MATLAB®

Mathematics

MATLAB®

R2021a

MathWorks®

How to Contact MathWorks

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

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

MATLAB$^{®}$ Mathematics

Trademarks

Patents

Revision History

June 2004	First printing	New for MATLAB 7.0 (Release 14), formerly part of *Using MATLAB*
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
June 2005	Second printing	Minor revision for MATLAB 7.0.4
September 2005	Second printing	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Second printing	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Second printing	Revised for MATLAB 7.3 (Release 2006b)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b)
March 2008	Online only	Revised for MATLAB 7.6 (Release 2008a)
October 2008	Online only	Revised for MATLAB 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised for MATLAB 7.10 (Release 2010a)
September 2010	Online only	Revised for MATLAB 7.11 (Release 2010b)
April 2011	Online only	Revised for MATLAB 7.12 (Release 2011a)
September 2011	Online only	Revised for MATLAB 7.13 (Release 2011b)
March 2012	Online only	Revised for MATLAB 7.14 (Release 2012a)
September 2012	Online only	Revised for MATLAB 8.0 (Release 2012b)
March 2013	Online only	Revised for MATLAB 8.1 (Release 2013a)
September 2013	Online only	Revised for MATLAB 8.2 (Release 2013b)
March 2014	Online only	Revised for MATLAB 8.3 (Release 2014a)
October 2014	Online only	Revised for MATLAB 8.4 (Release 2014b)
March 2015	Online only	Revised for MATLAB 8.5 (Release 2015a)
September 2015	Online only	Revised for MATLAB 8.6 (Release 2015b)
October 2015	Online only	Rereleased for Version 8.5.1 (Release 2015aSP1)
March 2016	Online only	Revised for MATLAB 9.0 (Release 2016a)
September 2016	Online only	Revised for MATLAB 9.1 (Release 2016b)
March 2017	Online only	Revised for MATLAB 9.2 (Release 2017a)
September 2017	Online only	Revised for MATLAB 9.3 (Release 2017b)
March 2018	Online only	Revised for MATLAB 9.4 (Release 2018a)
September 2018	Online only	Revised for MATLAB 9.5 (Release 2018b)
March 2019	Online only	Revised for MATLAB 9.6 (Release 2019a)
September 2019	Online only	Revised for MATLAB 9.7 (Release 2019b)
March 2020	Online only	Revised for MATLAB 9.8 (Release 2020a)
September 2020	Online only	Revised for MATLAB 9.9 (Release 2020b)
March 2021	Online only	Revised for MATLAB 9.10 (Release 2021a)

Contents

1 Matrices and Arrays

2 Linear Algebra

3 Random Numbers

4 Sparse Matrices

5 Graph and Network Algorithms

6 Functions of One Variable

7 Computational Geometry

8 Interpolation

9 Optimization

10 Function Handles

11 Ordinary Differential Equations (ODEs)

12 Boundary Value Problems (BVPs)

13 Partial Differential Equations (PDEs)

14 Delay Differential Equations (DDEs)

15 Numerical Integration

16 Fourier Transforms

1

Matrices and Arrays

Creating, Concatenating, and Expanding Matrices

The most basic MATLAB® data structure is the matrix. A matrix is a two-dimensional, rectangular array of data elements arranged in rows and columns. The elements can be numbers, logical values (`true` or `false`), dates and times, strings, or some other MATLAB data type.

Even a single number is stored as a matrix. For example, a variable containing the value 100 is stored as a 1-by-1 matrix of type `double`.

```
A = 100;
whos A
```

```
  Name      Size            Bytes  Class     Attributes

  A         1x1                 8  double
```

Constructing a Matrix of Data

If you have a specific set of data, you can arrange the elements in a matrix using square brackets. A single row of data has spaces or commas in between the elements, and a semicolon separates the rows. For example, create a single row of four numeric elements. The size of the resulting matrix is 1-by-4, since it has one row and four columns. A matrix of this shape is often referred to as a row vector.

```
A = [12 62 93 -8]
```

```
A = 1×4

    12    62    93    -8
```

```
sz = size(A)
```

```
sz = 1×2

     1     4
```

Now create a matrix with the same numbers, but arrange them in two rows. This matrix has two rows and two columns.

```
A = [12 62; 93 -8]
```

```
A = 2×2

    12    62
    93    -8
```

```
sz = size(A)
```

```
sz = 1×2

     2     2
```

Specialized Matrix Functions

MATLAB has many functions that help create matrices with certain values or a particular structure. For example, the `zeros` and `ones` functions create matrices of all zeros or all ones. The first and second arguments of these functions are the number of rows and number of columns of the matrix, respectively.

```
A = zeros(3,2)
```

```
A = 3×2

     0     0
     0     0
     0     0
```

```
B = ones(2,4)
```

```
B = 2×4

     1     1     1     1
     1     1     1     1
```

The `diag` function places the input elements on the diagonal of a matrix. For example, create a row vector `A` containing four elements. Then, create a 4-by-4 matrix whose diagonal elements are the elements of `A`.

```
A = [12 62 93 -8];
B = diag(A)
```

```
B = 4×4

    12     0     0     0
     0    62     0     0
     0     0    93     0
     0     0     0    -8
```

Concatenating Matrices

You can also use square brackets to join existing matrices together. This way of creating a matrix is called *concatenation*. For example, concatenate two row vectors to make an even longer row vector.

```
A = ones(1,4);
B = zeros(1,4);
C = [A B]
```

```
C = 1×8

     1     1     1     1     0     0     0     0
```

To arrange `A` and `B` as two rows of a matrix, use the semicolon.

```
D = [A;B]
```

```
D = 2×4
```

```
     1     1     1     1
     0     0     0     0
```

To concatenate two matrices, they must have compatible sizes. In other words, when you concatenate matrices horizontally, they must have the same number of rows. When you concatenate them vertically, they must have the same number of columns. For example, horizontally concatenate two matrices that both have two rows.

```
A = ones(2,3)
```

```
A = 2×3

     1     1     1
     1     1     1
```

```
B = zeros(2,2)
```

```
B = 2×2

     0     0
     0     0
```

```
C = [A B]
```

```
C = 2×5

     1     1     1     0     0
     1     1     1     0     0
```

An alternative way to concatenate matrices is to use concatenation functions such as `horzcat`, which horizontally concatenates two compatible input matrices.

```
D = horzcat(A,B)
```

```
D = 2×5

     1     1     1     0     0
     1     1     1     0     0
```

Generating a Numeric Sequence

The `colon` is a handy way to create matrices whose elements are sequential and evenly spaced. For example, create a row vector whose elements are the integers from 1 to 10.

```
A = 1:10
```

```
A = 1×10

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

You can use the colon operator to create a sequence of numbers within any range, incremented by one.

```
A = -2.5:2.5
```

```
A = 1×6

   -2.5000   -1.5000   -0.5000    0.5000    1.5000    2.5000
```

To change the value of the sequence increment, specify the increment value in between the starting and ending range values, separated by colons.

```
A = 0:2:10
```

```
A = 1×6

     0     2     4     6     8    10
```

To decrement, use a negative number.

```
A = 6:-1:0
```

```
A = 1×7

     6     5     4     3     2     1     0
```

You can also increment by noninteger values. If an increment value does not evenly partition the specified range, MATLAB automatically ends the sequence at the last value it can reach before exceeding the range.

```
A = 1:0.2:2.1
```

```
A = 1×6

    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000
```

Expanding a Matrix

You can add one or more elements to a matrix by placing them outside of the existing row and column index boundaries. MATLAB automatically pads the matrix with zeros to keep it rectangular. For example, create a 2-by-3 matrix and add an additional row and column to it by inserting an element in the (3,4) position.

```
A = [10  20  30; 60  70  80]
```

```
A = 2×3

    10    20    30
    60    70    80
```

```
A(3,4) = 1
```

```
A = 3×4

    10    20    30     0
    60    70    80     0
     0     0     0     1
```

You can also expand the size by inserting a new matrix outside of the existing index ranges.

```
A(4:5,5:6) = [2 3; 4 5]
```

```
A = 5×6

    10    20    30     0     0     0
    60    70    80     0     0     0
     0     0     0     1     0     0
     0     0     0     0     2     3
     0     0     0     0     4     5
```

To expand the size of a matrix repeatedly, such as within a `for` loop, it's usually best to preallocate space for the largest matrix you anticipate creating. Without preallocation, MATLAB has to allocate memory every time the size increases, slowing down operations. For example, preallocate a matrix that holds up to 10,000 rows and 10,000 columns by initializing its elements to zero.

```
A = zeros(10000,10000);
```

If you need to preallocate additional elements later, you can expand it by assigning outside of the matrix index ranges or concatenate another preallocated matrix to `A`.

Empty Arrays

An empty array in MATLAB is an array with at least one dimension length equal to zero. Empty arrays are useful for representing the concept of "nothing" programmatically. For example, suppose you want to find all elements of a vector that are less than 0, but there are none. The `find` function returns an empty vector of indices, indicating that it couldn't find any elements less than 0.

```
A = [1 2 3 4];
ind = find(A<0)
```

```
ind =

  1x0 empty double row vector
```

Many algorithms contain function calls that can return empty arrays. It is often useful to allow empty arrays to flow through these algorithms as function arguments instead of handling them as a special case. If you do need to customize empty array handling, you can check for them using the `isempty` function.

```
TF = isempty(ind)
```

```
TF = logical
   1
```

See Also

Related Examples

- “Array Indexing” on page 1-21
- “Reshaping and Rearranging Arrays” on page 1-9
- “Multidimensional Arrays” on page 1-14

- “Create String Arrays”
- “Represent Dates and Times in MATLAB”

Removing Rows or Columns from a Matrix

The easiest way to remove a row or column from a matrix is to set that row or column equal to a pair of empty square brackets `[]`. For example, create a 4-by-4 matrix and remove the second row.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
A(2,:) = []
```

```
A = 3×4

    16     2     3    13
     9     7     6    12
     4    14    15     1
```

Now remove the third column.

```
A(:,3) = []
```

```
A = 3×3

    16     2    13
     9     7    12
     4    14     1
```

You can extend this approach to any array. For example, create a random 3-by-3-by-3 array and remove all of the elements in the first matrix of the third dimension.

```
B = rand(3,3,3);
B(:,:,1) = [];
```

See Also

Related Examples

- "Reshaping and Rearranging Arrays" on page 1-9
- "Array Indexing" on page 1-21

Reshaping and Rearranging Arrays

Many functions in MATLAB® can take the elements of an existing array and put them in a different shape or sequence. This can be helpful for preprocessing your data for subsequent computations or analyzing the data.

Reshaping

The `reshape` function changes the size and shape of an array. For example, reshape a 3-by-4 matrix to a 2-by-6 matrix.

```
A = [1 4 7 10; 2 5 8 11; 3 6 9 12]
```

```
A = 3×4

     1     4     7    10
     2     5     8    11
     3     6     9    12
```

```
B = reshape(A,2,6)
```

```
B = 2×6

     1     3     5     7     9    11
     2     4     6     8    10    12
```

As long as the number of elements in each shape are the same, you can reshape them into an array with any number of dimensions. Using the elements from `A`, create a 2-by-2-by-3 multidimensional array.

```
C = reshape(A,2,2,3)
```

```
C = 
C(:,:,1) =

     1     3
     2     4


C(:,:,2) =

     5     7
     6     8


C(:,:,3) =

     9    11
    10    12
```

Transposing and Flipping

A common task in linear algebra is to work with the transpose of a matrix, which turns the rows into columns and the columns into rows. To do this, use the `transpose` function or the `.'` operator.

Create a 3-by-3 matrix and compute its transpose.

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

```
A = 3×3

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

```
B = A.'
```

```
B = 3×3

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

A similar operator `'` computes the conjugate transpose for complex matrices. This operation computes the complex conjugate of each element and transposes it. Create a 2-by-2 complex matrix and compute its conjugate transpose.

```
A = [1+i 1-i; -i i]
```

```
A = 2×2 complex

   1.0000 + 1.0000i   1.0000 - 1.0000i
   0.0000 - 1.0000i   0.0000 + 1.0000i
```

```
B = A'
```

```
B = 2×2 complex

   1.0000 - 1.0000i   0.0000 + 1.0000i
   1.0000 + 1.0000i   0.0000 - 1.0000i
```

`flipud` flips the rows of a matrix in an up-to-down direction, and `fliplr` flips the columns in a left-to-right direction.

```
A = [1 2; 3 4]
```

```
A = 2×2

     1     2
     3     4
```

```
B = flipud(A)
```

```
B = 2×2

     3     4
     1     2
```

```
C = fliplr(A)
```

```
C = 2×2

     2     1
     4     3
```

Shifting and Rotating

You can shift elements of an array by a certain number of positions using the `circshift` function. For example, create a 3-by-4 matrix and shift its columns to the right by 2. The second argument `[0 2]` tells `circshift` to shift the rows 0 places and shift the columns 2 places to the right.

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12]
```

```
A = 3×4

     1     2     3     4
     5     6     7     8
     9    10    11    12
```

```
B = circshift(A,[0 2])
```

```
B = 3×4

     3     4     1     2
     7     8     5     6
    11    12     9    10
```

To shift the rows of `A` up by 1 and keep the columns in place, specify the second argument as `[-1 0]`.

```
C = circshift(A,[-1 0])
```

```
C = 3×4

     5     6     7     8
     9    10    11    12
     1     2     3     4
```

The `rot90` function can rotate a matrix counterclockwise by 90 degrees.

```
A = [1 2; 3 4]
```

```
A = 2×2

     1     2
     3     4
```

```
B = rot90(A)
```

```
B = 2×2

     2     4
     1     3
```

If you rotate 3 more times by using the second argument to specify the number of rotations, you end up with the original matrix A.

```
C = rot90(B,3)
```

```
C = 2×2

     1     2
     3     4
```

Sorting

Sorting the data in an array is also a valuable tool, and MATLAB offers a number of approaches. For example, the sort function sorts the elements of each row or column of a matrix separately in ascending or descending order. Create a matrix A and sort each column of A in ascending order.

```
A = magic(4)
```

```
A = 4×4

    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

```
B = sort(A)
```

```
B = 4×4

     4     2     3     1
     5     7     6     8
     9    11    10    12
    16    14    15    13
```

Sort each row in descending order. The second argument value 2 specifies that you want to sort row-wise.

```
C = sort(A,2,'descend')
```

```
C = 4×4

    16    13     3     2
    11    10     8     5
    12     9     7     6
    15    14     4     1
```

To sort entire rows or columns relative to each other, use the sortrows function. For example, sort the rows of A in ascending order according to the elements in the first column. The positions of the rows change, but the order of the elements in each row are preserved.

```
D = sortrows(A)
```

```
D = 4×4

     4    14    15     1
     5    11    10     8
     9     7     6    12
    16     2     3    13
```

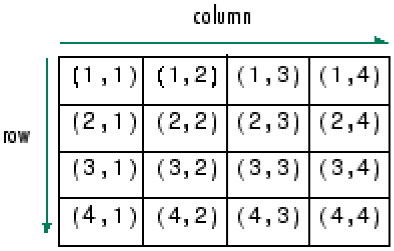
See Also

Related Examples

- “Removing Rows or Columns from a Matrix” on page 1-8
- “Array Indexing” on page 1-21

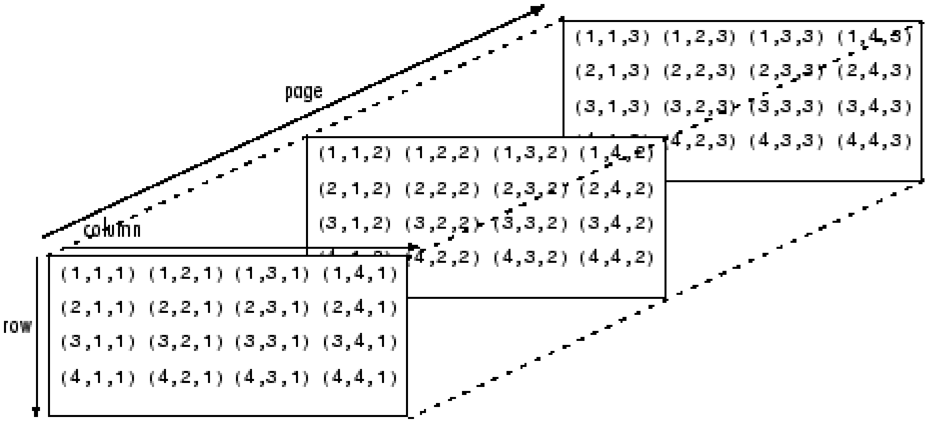
Multidimensional Arrays

A multidimensional array in MATLAB® is an array with more than two dimensions. In a matrix, the two dimensions are represented by rows and columns.



(1,1)	(1,2)	(1,3)	(1,4)
(2,1)	(2,2)	(2,3)	(2,4)
(3,1)	(3,2)	(3,3)	(3,4)
(4,1)	(4,2)	(4,3)	(4,4)

Each element is defined by two subscripts, the row index and the column index. Multidimensional arrays are an extension of 2-D matrices and use additional subscripts for indexing. A 3-D array, for example, uses three subscripts. The first two are just like a matrix, but the third dimension represents *pages* or *sheets* of elements.



Creating Multidimensional Arrays

You can create a multidimensional array by creating a 2-D matrix first, and then extending it. For example, first define a 3-by-3 matrix as the first page in a 3-D array.

```
A = [1 2 3; 4 5 6; 7 8 9]
```

```
A = 3×3

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

Now add a second page. To do this, assign another 3-by-3 matrix to the index value 2 in the third dimension. The syntax `A(:,:,2)` uses a colon in the first and second dimensions to include all rows and all columns from the right-hand side of the assignment.

```
A(:,:,2) = [10 11 12; 13 14 15; 16 17 18]
```

```
A = 
A(:,:,1) =
     1     2     3
     4     5     6
     7     8     9


A(:,:,2) =

    10    11    12
    13    14    15
    16    17    18
```

The `cat` function can be a useful tool for building multidimensional arrays. For example, create a new 3-D array `B` by concatenating `A` with a third page. The first argument indicates which dimension to concatenate along.

```
B = cat(3,A,[3 2 1; 0 9 8; 5 3 7])
```

```
B = 
B(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


B(:,:,2) =

    10    11    12
    13    14    15
    16    17    18


B(:,:,3) =

     3     2     1
     0     9     8
     5     3     7
```

Another way to quickly expand a multidimensional array is by assigning a single element to an entire page. For example, add a fourth page to `B` that contains all zeros.

```
B(:,:,4) = 0
```

```
B = 
B(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


B(:,:,2) =

    10    11    12
    13    14    15
    16    17    18


B(:,:,3) =

     3     2     1
     0     9     8
     5     3     7


B(:,:,4) =

     0     0     0
     0     0     0
     0     0     0
```

Accessing Elements

To access elements in a multidimensional array, use integer subscripts just as you would for vectors and matrices. For example, find the 1,2,2 element of A, which is in the first row, second column, and second page of A.

```
A
```

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

     1     2     3
     4     5     6
     7     8     9


A(:,:,2) =

    10    11    12
    13    14    15
    16    17    18
```

```
elA = A(1,2,2)
```

```
elA = 11
```

Use the index vector [1 3] in the second dimension to access only the first and last columns of each page of A.

```
C = A(:,[1 3],:)
```

```
C = 
C(:,:,1) =

     1     3
     4     6
     7     9


C(:,:,2) =
    10    12
    13    15
    16    18
```

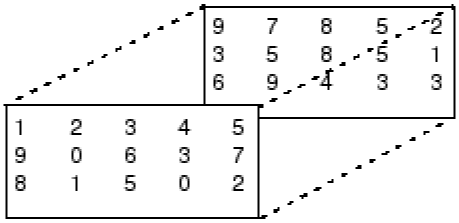
To find the second and third rows of each page, use the colon operator to create your index vector.

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

```
D = 
D(:,:,1) =

     4     5     6
     7     8     9


D(:,:,2) =

    13    14    15
    16    17    18
```

Manipulating Arrays

Elements of multidimensional arrays can be moved around in many ways, similar to vectors and matrices. `reshape`, `permute`, and `squeeze` are useful functions for rearranging elements. Consider a 3-D array with two pages.



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

1   2   3   4   5
9   0   6   3   7
8   1   5   0   2
```

Reshaping a multidimensional array can be useful for performing certain operations or visualizing the data. Use the `reshape` function to rearrange the elements of the 3-D array into a 6-by-5 matrix.

```
A = [1 2 3 4 5; 9 0 6 3 7; 8 1 5 0 2];
A(:,:,2) = [9 7 8 5 2; 3 5 8 5 1; 6 9 4 3 3];
B = reshape(A,[6 5])
```

```
B = 6×5

     1     3     5     7     5
     9     6     7     5     5
     8     5     2     9     3
     2     4     9     8     2
     0     3     3     8     1
     1     0     6     4     3
```

`reshape` operates columnwise, creating the new matrix by taking consecutive elements down each column of `A`, starting with the first page then moving to the second page.

Permutations are used to rearrange the order of the dimensions of an array. Consider a 3-D array `M`.

```
M(:,:,1) = [1 2 3; 4 5 6; 7 8 9];
M(:,:,2) = [0 5 4; 2 7 6; 9 3 1]
```

```
M =
M(:,:,1) =

     1     2     3
     4     5     6
     7     8     9


M(:,:,2) =

     0     5     4
     2     7     6
     9     3     1
```

Use the `permute` function to interchange row and column subscripts on each page by specifying the order of dimensions in the second argument. The original rows of `M` are now columns, and the columns are now rows.

```
P1 = permute(M,[2 1 3])
```

```
P1 =
P1(:,:,1) =

     1     4     7
     2     5     8
     3     6     9


P1(:,:,2) =

     0     2     9
     5     7     3
     4     6     1
```

Similarly, interchange row and page subscripts of `M`.

```
P2 = permute(M,[3 2 1])
```

```
P2 =
P2(:,:,1) =

     1     2     3
     0     5     4


P2(:,:,2) =

     4     5     6
     2     7     6


P2(:,:,3) =

     7     8     9
     9     3     1
```

When working with multidimensional arrays, you might encounter one that has an unnecessary dimension of length 1. The `squeeze` function performs another type of manipulation that eliminates dimensions of length 1. For example, use the `repmat` function to create a 2-by-3-by-1-by-4 array whose elements are each 5, and whose third dimension has length 1.

```
A = repmat(5,[2 3 1 4])
```

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

     5     5     5
     5     5     5


A(:,:,1,2) =

     5     5     5
     5     5     5


A(:,:,1,3) =

     5     5     5
     5     5     5


A(:,:,1,4) =

     5     5     5
     5     5     5
```

```
szA = size(A)
```

```
szA = 1×4

     2     3     1     4
```

```
numdimsA = ndims(A)
```

```
numdimsA = 4
```

Use the `squeeze` function to remove the third dimension, resulting in a 3-D array.

```
B = squeeze(A)
```

```
B = 
B(:,:,1) =

     5     5     5
     5     5     5


B(:,:,2) =
     5     5     5
     5     5     5


B(:,:,3) =

     5     5     5
     5     5     5


B(:,:,4) =

     5     5     5
     5     5     5
```

```
szB = size(B)
```

```
szB = 1×3

     2     3     4
```

```
numdimsB = ndims(B)
```

```
numdimsB = 3
```

See Also

Related Examples

- “Creating, Concatenating, and Expanding Matrices” on page 1-2
- “Array Indexing” on page 1-21
- “Reshaping and Rearranging Arrays” on page 1-9

Array Indexing

In MATLAB®, there are three primary approaches to accessing array elements based on their location (index) in the array. These approaches are indexing by position, linear indexing, and logical indexing.

Indexing with Element Positions

The most common way is to explicitly specify the indices of the elements. For example, to access a single element of a matrix, specify the row number followed by the column number of the element.

```
A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]
```

```
A = 4×4

     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

```
e = A(3,2)
```

```
e = 10
```

`e` is the element in the 3,2 position (third row, second column) of `A`.

You can also reference multiple elements at a time by specifying their indices in a vector. For example, access the first and third elements of the second row of `A`.

```
r = A(2,[1 3])
```

```
r = 1×2

     5     7
```

To access elements in a range of rows or columns, use the `colon`. For example, access the elements in the first through third row and the second through fourth column of `A`.

```
r = A(1:3,2:4)
```

```
r = 3×3

     2     3     4
     6     7     8
    10    11    12
```

An alternative way to compute `r` is to use the keyword `end` to specify the second column through the last column. This approach lets you specify the last column without knowing exactly how many columns are in `A`.

```
r = A(1:3,2:end)
```

```
r = 3×3

     2     3     4
     6     7     8
    10    11    12
```

If you want to access all of the rows or columns, use the colon operator by itself. For example, return the entire third column of A.

```
r = A(:,3)
```

```
r = 4×1

     3
     7
    11
    15
```

In general, you can use indexing to access elements of any array in MATLAB regardless of its data type or dimensions. For example, directly access a column of a datetime array.

```
t = [datetime(2018,1:5,1); datetime(2019,1:5,1)]
```

```
t = 2x5 datetime
   01-Jan-2018   01-Feb-2018   01-Mar-2018   01-Apr-2018   01-May-2018
   01-Jan-2019   01-Feb-2019   01-Mar-2019   01-Apr-2019   01-May-2019
```

```
march1 = t(:,3)
```

```
march1 = 2x1 datetime
   01-Mar-2018
   01-Mar-2019
```

For higher-dimensional arrays, expand the syntax to match the array dimensions. Consider a random 3-by-3-by-3 numeric array. Access the element in the second row, third column, and first sheet of the array.

```
A = rand(3,3,3);
e = A(2,3,1)
```

```
e = 0.5469
```

For more information on working with multidimensional arrays, see “Multidimensional Arrays” on page 1-14.

Indexing with a Single Index

Another method for accessing elements of an array is to use only a single index, regardless of the size or dimensions of the array. This method is known as *linear indexing*. While MATLAB displays arrays according to their defined sizes and shapes, they are actually stored in memory as a single column of elements. A good way to visualize this concept is with a matrix. While the following array is displayed as a 3-by-3 matrix, MATLAB stores it as a single column made up of the columns of A appended one after the other. The stored vector contains the sequence of elements 12, 45, 33, 36, 29, 25, 91, 48, 11, and can be displayed using a single colon.

```
A = [12 36 91; 45 29 48; 33 25 11]
```

```
A = 3×3

    12    36    91
    45    29    48
    33    25    11
```

```
Alinear = A(:)
```

```
Alinear = 9×1

    12
    45
    33
    36
    29
    25
    91
    48
    11
```

For example, the 3,2 element of `A` is `25`, and you can access it using the syntax `A(3,2)`. You can also access this element using the syntax `A(6)`, since `25` is sixth element of the stored vector sequence.

```
e = A(3,2)
```

```
e = 25
```

```
elinear = A(6)
```

```
elinear = 25
```

While linear indexing can be less intuitive visually, it can be powerful for performing certain computations that are not dependent on the size or shape of the array. For example, you can easily sum all of the elements of `A` without having to provide a second argument to the `sum` function.

```
s = sum(A(:))
```

```
s = 330
```

The `sub2ind` and `ind2sub` functions help to convert between original array indices and their linear version. For example, compute the linear index of the 3,2 element of `A`.

```
linearidx = sub2ind(size(A),3,2)
```

```
linearidx = 6
```

Convert from the linear index back to its row and column form.

```
[row,col] = ind2sub(size(A),6)
```

```
row = 3
```

```
col = 2
```

Indexing with Logical Values

Using true and false logical indicators is another useful way to index into arrays, particularly when working with conditional statements. For example, say you want to know if the elements of a matrix `A` are less than the corresponding elements of another matrix B. The less-than operator returns a logical array whose elements are 1 when an element in A is smaller than the corresponding element in B.

```
A = [1 2 6; 4 3 6]
```

```
A = 2×3

     1     2     6
     4     3     6

```

```
B = [0 3 7; 3 7 5]
```

```
B = 2×3

     0     3     7
     3     7     5

```

```
ind = A<B
```

```
ind = 2x3 logical array

   0   1   1
   0   1   0

```

Now that you know the locations of the elements meeting the condition, you can inspect the individual values using ind as the index array. MATLAB matches the locations of the value 1 in ind to the corresponding elements of A and B, and lists their values in a column vector.

```
Avals = A(ind)
```

```
Avals = 3×1

     2
     3
     6

```

```
Bvals = B(ind)
```

```
Bvals = 3×1

     3
     7
     7

```

MATLAB "is" functions also return logical arrays that indicate which elements of the input meet a certain condition. For example, check which elements of a string vector are missing using the ismissing function.

```
str = ["A" "B" missing "D" "E" missing];
ind = ismissing(str)
```

```
ind = 1x6 logical array
```

```
   0   0   1   0   0   1
```

Suppose you want to find the values of the elements that are *not* missing. Use the ~ operator with the index vector `ind` to do this.

```
strvals = str(~ind)
```

```
strvals = 1x4 string
    "A"    "B"    "D"    "E"
```

For more examples using logical indexing, see “Find Array Elements That Meet a Condition”.

See Also

Related Examples

- “Access Data Using Categorical Arrays”
- “Access Data in Tables”
- “Structure Arrays”
- “Access Data in Cell Array”

Linear Algebra

Matrices in the MATLAB Environment

This topic contains an introduction to creating matrices and performing basic matrix calculations in MATLAB.

The MATLAB environment uses the term *matrix* to indicate a variable containing real or complex numbers arranged in a two-dimensional grid. An *array* is, more generally, a vector, matrix, or higher dimensional grid of numbers. All arrays in MATLAB are rectangular, in the sense that the component vectors along any dimension are all the same length. The mathematical operations defined on matrices are the subject of linear algebra.

Creating Matrices

MATLAB has many functions that create different kinds of matrices. For example, you can create a symmetric matrix with entries based on Pascal's triangle:

```
A = pascal(3)
```

```
A =
       1     1     1
       1     2     3
       1     3     6
```

Or, you can create an unsymmetric *magic square matrix*, which has equal row and column sums:

```
B = magic(3)
```

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

Another example is a 3-by-2 rectangular matrix of random integers. In this case the first input to `randi` describes the range of possible values for the integers, and the second two inputs describe the number of rows and columns.

```
C = randi(10,3,2)
```

```
C =

     9    10
    10     7
     2     1
```

A column vector is an *m*-by-1 matrix, a row vector is a 1-by-*n* matrix, and a scalar is a 1-by-1 matrix. To define a matrix manually, use square brackets `[ ]` to denote the beginning and end of the array. Within the brackets, use a semicolon `;` to denote the end of a row. In the case of a scalar (1-by-1 matrix), the brackets are not required. For example, these statements produce a column vector, a row vector, and a scalar:

```
u = [3; 1; 4]

v = [2 0 -1]

s = 7
```

```
u =
       3
       1
       4

v =
       2     0    -1

s =
       7
```

For more information about creating and working with matrices, see “Creating, Concatenating, and Expanding Matrices” on page 1-2.

Adding and Subtracting Matrices

Addition and subtraction of matrices and arrays is performed element-by-element, or *element-wise*. For example, adding A to B and then subtracting A from the result recovers B:

```
X = A + B
```

```
X =
       9     2     7
       4     7    10
       5    12     8
```

```
Y = X - A
```

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

Addition and subtraction require both matrices to have compatible dimensions. If the dimensions are incompatible, an error results:

```
X = A + C
```

```
Error using  +
Matrix dimensions must agree.
```

For more information, see “Array vs. Matrix Operations”.

Vector Products and Transpose

A row vector and a column vector of the same length can be multiplied in either order. The result is either a scalar, called the *inner product*, or a matrix, called the *outer product*:

```
u = [3; 1; 4];
v = [2 0 -1];
x = v*u
```

```
x =

     2
```

```
X = u*v
```

```
X =

     6     0    -3
     2     0    -1
     8     0    -4
```

For real matrices, the *transpose* operation interchanges a_{ij} and a_{ji}. For complex matrices, another consideration is whether to take the complex conjugate of complex entries in the array to form the *complex conjugate transpose*. MATLAB uses the apostrophe operator (`'`) to perform a complex conjugate transpose, and the dot-apostrophe operator (`.'`) to transpose without conjugation. For matrices containing all real elements, the two operators return the same result.

The example matrix `A = pascal(3)` is *symmetric*, so `A'` is equal to `A`. However, `B = magic(3)` is not symmetric, so `B'` has the elements reflected along the main diagonal:

```
B = magic(3)
```

```
B =

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

```
X = B'
```

```
X =

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

For vectors, transposition turns a row vector into a column vector (and vice-versa):

```
x = v'

x =
       2
       0
      -1
```

If `x` and `y` are both real column vectors, then the product `x*y` is not defined, but the two products

```
x'*y
```

and

```
y'*x
```

produce the same scalar result. This quantity is used so frequently, it has three different names: *inner* product, *scalar* product, or *dot* product. There is even a dedicated function for dot products named `dot`.

For a complex vector or matrix, `z`, the quantity `z'` not only transposes the vector or matrix, but also converts each complex element to its complex conjugate. That is, the sign of the imaginary part of each complex element changes. For example, consider the complex matrix

```
z = [1+2i 7-3i 3+4i; 6-2i 9i 4+7i]
```

```
z =

   1.0000 + 2.0000i   7.0000 - 3.0000i   3.0000 + 4.0000i
   6.0000 - 2.0000i   0.0000 + 9.0000i   4.0000 + 7.0000i
```

The complex conjugate transpose of `z` is:

```
z'
```

```
ans =

   1.0000 - 2.0000i   6.0000 + 2.0000i
   7.0000 + 3.0000i   0.0000 - 9.0000i
   3.0000 - 4.0000i   4.0000 - 7.0000i
```

The unconjugated complex transpose, where the complex part of each element retains its sign, is denoted by `z.'`:

```
z.'
```

```
ans =

   1.0000 + 2.0000i   6.0000 - 2.0000i
   7.0000 - 3.0000i   0.0000 + 9.0000i
   3.0000 + 4.0000i   4.0000 + 7.0000i
```

For complex vectors, the two scalar products `x'*y` and `y'*x` are complex conjugates of each other, and the scalar product `x'*x` of a complex vector with itself is real.

Multiplying Matrices

Multiplication of matrices is defined in a way that reflects composition of the underlying linear transformations and allows compact representation of systems of simultaneous linear equations. The matrix product $C = AB$ is defined when the column dimension of A is equal to the row dimension of B, or when one of them is a scalar. If A is m-by-p and B is p-by-n, their product C is m-by-n. The product can actually be defined using MATLAB `for` loops, `colon` notation, and vector dot products:

```
A = pascal(3);
B = magic(3);
m = 3;
n = 3;
for i = 1:m
     for j = 1:n
        C(i,j) = A(i,:)*B(:,j);
     end
end
```

MATLAB uses an asterisk to denote matrix multiplication, as in `C = A*B`. Matrix multiplication is not commutative; that is, `A*B` is typically not equal to `B*A`:

```
X = A*B
```

```
X =
      15    15    15
      26    38    26
      41    70    39
```

```
Y = B*A
```

```
Y =
      15    28    47
      15    34    60
      15    28    43
```

A matrix can be multiplied on the right by a column vector and on the left by a row vector:

```
u = [3; 1; 4];
x = A*u
```

```
x =

     8
    17
    30
```

```
v = [2 0 -1];
y = v*B
```

```
y =

    12    -7    10
```

Rectangular matrix multiplications must satisfy the dimension compatibility conditions. Since `A` is 3-by-3 and `C` is 3-by-2, you can multiply them to get a 3-by-2 result (the common inner dimension cancels):

```
X = A*C
```

```
X =

    24    17
    47    42
    79    77
```

However, the multiplication does not work in the reverse order:

```
Y = C*A
```

```
Error using  *
Incorrect dimensions for matrix multiplication. Check that the number of columns
in the first matrix matches the number of rows in the second matrix. To perform
elementwise multiplication, use '.*'.
```

You can multiply anything with a scalar:

```
s = 10;
w = s*y
```

```
w =

   120   -70   100
```

When you multiply an array by a scalar, the scalar implicitly expands to be the same size as the other input. This is often referred to as *scalar expansion*.

Identity Matrix

Generally accepted mathematical notation uses the capital letter I to denote identity matrices, matrices of various sizes with ones on the main diagonal and zeros elsewhere. These matrices have the property that $AI = A$ and $IA = A$ whenever the dimensions are compatible.

The original version of MATLAB could not use I for this purpose because it did not distinguish between uppercase and lowercase letters and i already served as a subscript and as the complex unit. So an English language pun was introduced. The function

```
eye(m,n)
```

returns an m-by-n rectangular identity matrix and `eye(n)` returns an n-by-n square identity matrix.

Matrix Inverse

If a matrix `A` is square and nonsingular (nonzero determinant), then the equations $AX = I$ and $XA = I$ have the same solution X. This solution is called the *inverse* of `A` and is denoted A^{-1}. The `inv` function and the expression `A^-1` both compute the matrix inverse.

```
A = pascal(3)
```

```
A =
       1     1     1
       1     2     3
       1     3     6
```

```
X = inv(A)
```

```
X =

    3.0000   -3.0000    1.0000
   -3.0000    5.0000   -2.0000
    1.0000   -2.0000    1.0000
```

```
A*X
```

```
ans =

    1.0000         0         0
    0.0000    1.0000   -0.0000
   -0.0000    0.0000    1.0000
```

The *determinant* calculated by `det` is a measure of the scaling factor of the linear transformation described by the matrix. When the determinant is exactly zero, the matrix is *singular* and no inverse exists.

```
d = det(A)
```

```
d =

     1
```

Some matrices are *nearly singular*, and despite the fact that an inverse matrix exists, the calculation is susceptible to numerical errors. The `cond` function computes the *condition number for inversion*, which gives an indication of the accuracy of the results from matrix inversion. The condition number ranges from `1` for a numerically stable matrix to `Inf` for a singular matrix.

```
c = cond(A)

c =

   61.9839
```

It is seldom necessary to form the explicit inverse of a matrix. A frequent misuse of `inv` arises when solving the system of linear equations *Ax* = *b*. The best way to solve this equation, from the standpoint of both execution time and numerical accuracy, is to use the matrix backslash operator `x = A\b`. See `mldivide` for more information.

Kronecker Tensor Product

The Kronecker product, `kron(X,Y)`, of two matrices is the larger matrix formed from all possible products of the elements of `X` with those of `Y`. If `X` is *m*-by-*n* and `Y` is *p*-by-*q*, then `kron(X,Y)` is *mp*-by-*nq*. The elements are arranged such that each element of `X` is multiplied by the entire matrix `Y`:

```
[X(1,1)*Y  X(1,2)*Y  . . .  X(1,n)*Y
                     . . .
 X(m,1)*Y  X(m,2)*Y  . . .  X(m,n)*Y]
```

The Kronecker product is often used with matrices of zeros and ones to build up repeated copies of small matrices. For example, if `X` is the 2-by-2 matrix

```
X = [1   2
     3   4]
```

and `I = eye(2,2)` is the 2-by-2 identity matrix, then:

```
kron(X,I)

ans =

     1     0     2     0
     0     1     0     2
     3     0     4     0
     0     3     0     4
```

and

```
kron(I,X)

ans =

     1     2     0     0
     3     4     0     0
     0     0     1     2
     0     0     3     4
```

Aside from `kron`, some other functions that are useful to replicate arrays are `repmat`, `repelem`, and `blkdiag`.

Vector and Matrix Norms

The *p*-norm of a vector *x*,

$$\|x\|_p = \left(\sum |x_i|^p\right)^{1/p},$$

is computed by `norm(x,p)`. This operation is defined for any value of $p > 1$, but the most common values of p are 1, 2, and ∞. The default value is $p = 2$, which corresponds to *Euclidean length* or *vector magnitude*:

```
v = [2 0 -1];
[norm(v,1) norm(v) norm(v,inf)]

ans =

    3.0000    2.2361    2.0000
```

The p-norm of a matrix A,

$$\|A\|_p = \max_x \frac{\|Ax\|_p}{\|x\|_p},$$

can be computed for $p = 1$, 2, and ∞ by `norm(A,p)`. Again, the default value is $p = 2$:

```
A = pascal(3);
[norm(A,1) norm(A) norm(A,inf)]

ans =

   10.0000    7.8730   10.0000
```

In cases where you want to calculate the norm of each row or column of a matrix, you can use `vecnorm`:

```
vecnorm(A)

ans =

    1.7321    3.7417    6.7823
```

Using Multithreaded Computation with Linear Algebra Functions

MATLAB supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.
2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains several thousand elements or more.
3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complicated functions speed up more than simple functions.

The matrix multiply `(X*Y)` and matrix power `(X^p)` operators show significant increase in speed on large double-precision arrays (on order of 10,000 elements). The matrix analysis functions `det`, `rcond`, `hess`, and `expm` also show significant increase in speed on large double-precision arrays.

Systems of Linear Equations

In this section...

Computational Considerations

One of the most important problems in technical computing is the solution of systems of simultaneous linear equations.

In matrix notation, the general problem takes the following form: Given two matrices A and b, does there exist a unique matrix x, so that $Ax= b$ or $xA= b$?

It is instructive to consider a 1-by-1 example. For example, does the equation

$7x = 21$

have a unique solution?

The answer, of course, is yes. The equation has the unique solution $x = 3$. The solution is easily obtained by division:

$x = 21/7 = 3$.

The solution is *not* ordinarily obtained by computing the inverse of 7, that is 7^{-1}= 0.142857..., and then multiplying 7^{-1} by 21. This would be more work and, if 7^{-1} is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown; MATLAB solves such equations without computing the inverse of the matrix.

Although it is not standard mathematical notation, MATLAB uses the division terminology familiar in the scalar case to describe the solution of a general system of simultaneous equations. The two division symbols, *slash*, /, and *backslash*, \, correspond to the two MATLAB functions `mrdivide` and `mldivide`. These operators are used for the two situations where the unknown matrix appears on the left or right of the coefficient matrix:

`x = b/A`	Denotes the solution to the matrix equation $xA = b$, obtained using `mrdivide`.
`x = A\b`	Denotes the solution to the matrix equation $Ax = b$, obtained using `mldivide`.

Think of "dividing" both sides of the equation $Ax = b$ or $xA = b$ by A. The coefficient matrix `A` is always in the "denominator."

The dimension compatibility conditions for `x = A\b` require the two matrices `A` and `b` to have the same number of rows. The solution `x` then has the same number of columns as `b` and its row dimension is equal to the column dimension of `A`. For `x = b/A`, the roles of rows and columns are interchanged.

In practice, linear equations of the form $Ax = b$ occur more frequently than those of the form $xA = b$. Consequently, the backslash is used far more frequently than the slash. The remainder of this section concentrates on the backslash operator; the corresponding properties of the slash operator can be inferred from the identity:

```
(b/A)' = (A'\b').
```

The coefficient matrix `A` need not be square. If `A` has size *m*-by-*n*, then there are three cases:

m = n	Square system. Seek an exact solution.
m > n	Overdetermined system, with more equations than unknowns. Find a least-squares solution.
m < n	Underdetermined system, with fewer equations than unknowns. Find a basic solution with at most *m* nonzero components.

The mldivide Algorithm

The `mldivide` operator employs different solvers to handle different kinds of coefficient matrices. The various cases are diagnosed automatically by examining the coefficient matrix. For more information, see the “Algorithms” section of the `mldivide` reference page.

General Solution

The general solution to a system of linear equations $Ax= b$ describes all possible solutions. You can find the general solution by:

1. Solving the corresponding homogeneous system $Ax = 0$. Do this using the `null` command, by typing `null(A)`. This returns a basis for the solution space to $Ax = 0$. Any solution is a linear combination of basis vectors.
2. Finding a particular solution to the nonhomogeneous system $Ax =b$.

You can then write any solution to $Ax= b$ as the sum of the particular solution to $Ax =b$, from step 2, plus a linear combination of the basis vectors from step 1.

The rest of this section describes how to use MATLAB to find a particular solution to $Ax =b$, as in step 2.

Square Systems

The most common situation involves a square coefficient matrix `A` and a single right-hand side column vector `b`.

Nonsingular Coefficient Matrix

If the matrix `A` is nonsingular, then the solution, `x = A\b`, is the same size as `b`. For example:

```
A = pascal(3);
u = [3; 1; 4];
```

```
x = A\u

x =
      10
     -12
       5
```

It can be confirmed that `A*x` is exactly equal to `u`.

If `A` and `b` are square and the same size, `x= A\b` is also that size:

```
b = magic(3);
X = A\b

X =
      19    -3    -1
     -17     4    13
       6     0    -6
```

It can be confirmed that `A*x` is exactly equal to `b`.

Both of these examples have exact, integer solutions. This is because the coefficient matrix was chosen to be `pascal(3)`, which is a full rank matrix (nonsingular).

Singular Coefficient Matrix

A square matrix A is singular if it does not have linearly independent columns. If A is singular, the solution to $Ax = b$ either does not exist, or is not unique. The backslash operator, `A\b`, issues a warning if `A` is nearly singular or if it detects exact singularity.

If A is singular and $Ax = b$ has a solution, you can find a particular solution that is not unique, by typing

```
P = pinv(A)*b
```

`pinv(A)` is a pseudoinverse of A. If $Ax = b$ does not have an exact solution, then `pinv(A)` returns a least-squares solution.

For example:

```
A = [ 1     3     7
     -1     4     4
      1    10    18 ]
```

is singular, as you can verify by typing

```
rank(A)

ans =

     2
```

Since A is not full rank, it has some singular values equal to zero.

Exact Solutions. For `b =[5;2;12]`, the equation $Ax = b$ has an exact solution, given by

```
pinv(A)*b
```

```
ans =
    0.3850
   -0.1103
    0.7066
```

Verify that `pinv(A)*b` is an exact solution by typing

```
A*pinv(A)*b

ans =
    5.0000
    2.0000
   12.0000
```

Least-Squares Solutions. However, if `b = [3;6;0]`, *Ax* = *b* does not have an exact solution. In this case, `pinv(A)*b` returns a least-squares solution. If you type

```
A*pinv(A)*b

ans =
   -1.0000
    4.0000
    2.0000
```

you do not get back the original vector `b`.

You can determine whether *Ax* =*b* has an exact solution by finding the row reduced echelon form of the augmented matrix `[A b]`. To do so for this example, enter

```
rref([A b])
ans =
    1.0000         0    2.2857         0
         0    1.0000    1.5714         0
         0         0         0    1.0000
```

Since the bottom row contains all zeros except for the last entry, the equation does not have a solution. In this case, `pinv(A)` returns a least-squares solution.

Overdetermined Systems

This example shows how overdetermined systems are often encountered in various kinds of curve fitting to experimental data.

A quantity `y` is measured at several different values of time `t` to produce the following observations. You can enter the data and view it in a table with the following statements.

```
t = [0 .3 .8 1.1 1.6 2.3]';
y = [.82 .72 .63 .60 .55 .50]';
B = table(t,y)
```

```
B=6×2 table
     t      y  
    ___    ____

      0    0.82
    0.3    0.72
    0.8    0.63
    1.1     0.6
    1.6    0.55
    2.3     0.5
```

Try modeling the data with a decaying exponential function

$$y(t) = c_1 + c_2 e^{-t}.$$

The preceding equation says that the vector `y` should be approximated by a linear combination of two other vectors. One is a constant vector containing all ones and the other is the vector with components `exp(-t)`. The unknown coefficients, c_1 and c_2, can be computed by doing a least-squares fit, which minimizes the sum of the squares of the deviations of the data from the model. There are six equations in two unknowns, represented by a 6-by-2 matrix.

```
E = [ones(size(t)) exp(-t)]
```

```
E = 6×2

    1.0000    1.0000
    1.0000    0.7408
    1.0000    0.4493
    1.0000    0.3329
    1.0000    0.2019
    1.0000    0.1003
```

Use the backslash operator to get the least-squares solution.

```
c = E\y
```

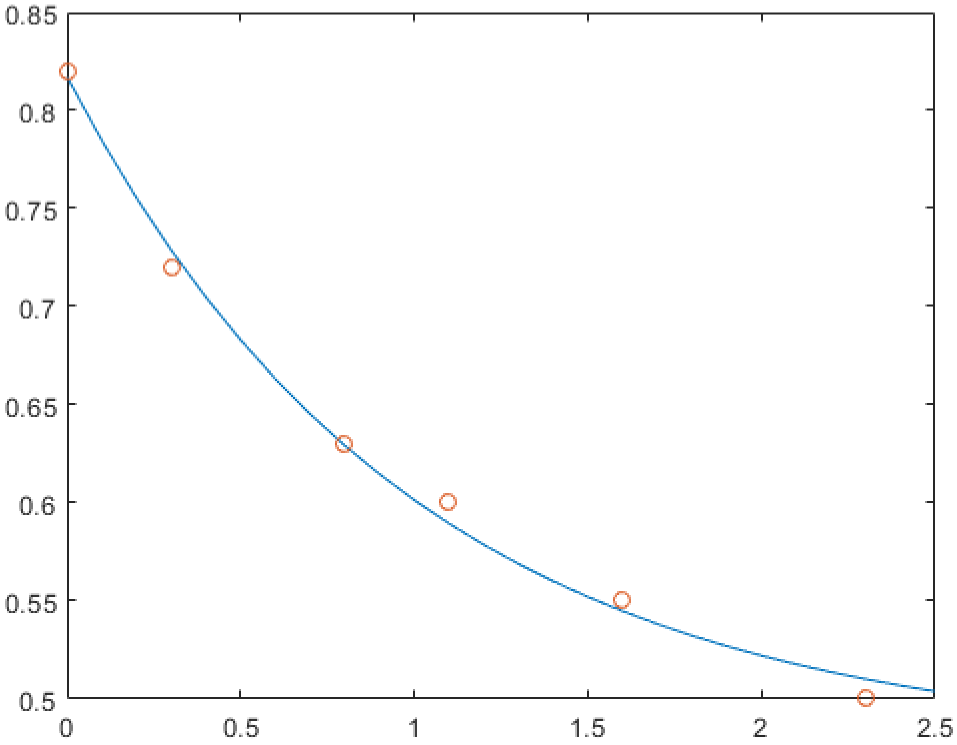
```
c = 2×1

    0.4760
    0.3413
```

In other words, the least-squares fit to the data is

$$y(t) = 0.4760 + 0.3413e^{-t}.$$

The following statements evaluate the model at regularly spaced increments in `t`, and then plot the result together with the original data:

```
T = (0:0.1:2.5)';
Y = [ones(size(T)) exp(-T)]*c;
plot(T,Y,'-',t,y,'o')
```



E*c is not exactly equal to y, but the difference might well be less than measurement errors in the original data.

A rectangular matrix A is rank deficient if it does not have linearly independent columns. If A is rank deficient, then the least-squares solution to AX = B is not unique. A\B issues a warning if A is rank deficient and produces a least-squares solution. You can use lsqminnorm to find the solution X that has the minimum norm among all solutions.

Underdetermined Systems

This example shows how the solution to underdetermined systems is not unique. Underdetermined linear systems involve more unknowns than equations. The matrix left division operation in MATLAB finds a basic least-squares solution, which has at most m nonzero components for an m-by-n coefficient matrix.

Here is a small, random example:

```
R = [6 8 7 3; 3 5 4 1]
rng(0);
b = randi(8,2,1)
```

```
R =

     6     8     7     3
     3     5     4     1
```

```
b =

       7
       8
```

The linear system `Rp = b` involves two equations in four unknowns. Since the coefficient matrix contains small integers, it is appropriate to use the `format` command to display the solution in rational format. The particular solution is obtained with

```
format rat
p = R\b
```

```
p =

       0
      17/7
       0
     -29/7
```

One of the nonzero components is `p(2)` because `R(:,2)` is the column of `R` with largest norm. The other nonzero component is `p(4)` because `R(:,4)` dominates after `R(:,2)` is eliminated.

The complete general solution to the underdetermined system can be characterized by adding `p` to an arbitrary linear combination of the null space vectors, which can be found using the `null` function with an option requesting a rational basis.

```
Z = null(R,'r')
```

```
Z =

      -1/2           -7/6
      -1/2            1/2
       1              0
       0              1
```

It can be confirmed that `R*Z` is zero and that the residual `R*x - b` is small for any vector `x`, where

```
x = p + Z*q
```

Since the columns of `Z` are the null space vectors, the product `Z*q` is a linear combination of those vectors:

$$Zq = \left(\vec{x}_1\ \vec{x}_2\right)\begin{pmatrix}u\\ w\end{pmatrix} = u\vec{x}_1 + w\vec{x}_2\ .$$

To illustrate, choose an arbitrary `q` and construct `x`.

```
q = [-2; 1];
x = p + Z*q;
```

Calculate the norm of the residual.

```
format short
norm(R*x - b)
```

```
ans =

   2.6645e-15
```

When infinitely many solutions are available, the solution with minimum norm is of particular interest. You can use `lsqminnorm` to compute the minimum-norm least-squares solution. This solution has the smallest possible value for `norm(p)`.

```
p = lsqminnorm(R,b)
```

```
p =

    -207/137
     365/137
      79/137
    -424/137
```

Solving for Several Right-Hand Sides

Some problems are concerned with solving linear systems that have the same coefficient matrix `A`, but different right-hand sides `b`. When the different values of `b` are available at the same time, you can construct `b` as a matrix with several columns and solve all of the systems of equations at the same time using a single backslash command: `X = A\[b1 b2 b3 …]`.

However, sometimes the different values of `b` are not all available at the same time, which means you need to solve several systems of equations consecutively. When you solve one of these systems of equations using slash (/) or backslash (\), the operator factorizes the coefficient matrix `A` and uses this matrix decomposition to compute the solution. However, each subsequent time you solve a similar system of equations with a different `b`, the operator computes the same decomposition of `A`, which is a redundant computation.

The solution to this problem is to precompute the decomposition of `A`, and then reuse the factors to solve for the different values of `b`. In practice, however, precomputing the decomposition in this manner can be difficult since you need to know which decomposition to compute (LU, LDL, Cholesky, and so on) as well as how to multiply the factors to solve the problem. For example, with LU decomposition you need to solve two linear systems to solve the original system $Ax = b$:

```
[L,U] = lu(A);
x = U \ (L \ b);
```

Instead, the recommended method for solving linear systems with several consecutive right-hand sides is to use `decomposition` objects. These objects enable you to leverage the performance benefits of precomputing the matrix decomposition, but they *do not* require knowledge of how to use the matrix factors. You can replace the previous LU decomposition with:

```
dA = decomposition(A,'lu');
x = dA\b;
```

If you are unsure which decomposition to use, `decomposition(A)` chooses the correct type based on the properties of `A`, similar to what backslash does.

Here is a simple test of the possible performance benefits of this approach. The test solves the same sparse linear system 100 times using both backslash (\) and `decomposition`.

```
n = 1e3;
A = sprand(n,n,0.2) + speye(n);
```

```
b = ones(n,1);

% Backslash solution
tic
for k = 1:100
    x = A\b;
end
toc
```

```
Elapsed time is 9.006156 seconds.
```

```
% decomposition solution
tic
dA = decomposition(A);
for k = 1:100
    x = dA\b;
end
toc
```

```
Elapsed time is 0.374347 seconds.
```

For this problem, the `decomposition` solution is much faster than using backslash alone, yet the syntax remains simple.

Iterative Methods

If the coefficient matrix A is large and sparse, factorization methods are generally not efficient. Iterative methods generate a series of approximate solutions. MATLAB provides several iterative methods to handle large, sparse input matrices.

Function	Description
`pcg`	Preconditioned conjugate gradients method. This method is appropriate for Hermitian positive definite coefficient matrix A.
`bicg`	BiConjugate Gradients Method
`bicgstab`	BiConjugate Gradients Stabilized Method
`bicgstabl`	BiCGStab(l) Method
`cgs`	Conjugate Gradients Squared Method
`gmres`	Generalized Minimum Residual Method
`lsqr`	LSQR Method
`minres`	Minimum Residual Method. This method is appropriate for Hermitian coefficient matrix A.
`qmr`	Quasi-Minimal Residual Method
`symmlq`	Symmetric LQ Method
`tfqmr`	Transpose-Free QMR Method

Multithreaded Computation

MATLAB supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.
2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains several thousand elements or more.
3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complicated functions speed up more than simple functions.

`inv`, `lscov`, `linsolve`, and `mldivide` show significant increase in speed on large double-precision arrays (on order of 10,000 elements or more) when multithreading is enabled.

Factorizations

In this section...

Introduction

All three of the matrix factorizations discussed in this section make use of *triangular* matrices, where all the elements either above or below the diagonal are zero. Systems of linear equations involving triangular matrices are easily and quickly solved using either *forward* or *back substitution*.

Cholesky Factorization

The Cholesky factorization expresses a symmetric matrix as the product of a triangular matrix and its transpose

$A = R'R,$

where R is an upper triangular matrix.

Not all symmetric matrices can be factored in this way; the matrices that have such a factorization are said to be positive definite. This implies that all the diagonal elements of A are positive and that the off-diagonal elements are "not too big." The Pascal matrices provide an interesting example. Throughout this chapter, the example matrix A has been the 3-by-3 Pascal matrix. Temporarily switch to the 6-by-6:

```
A = pascal(6)

A =
       1     1     1     1     1     1
       1     2     3     4     5     6
       1     3     6    10    15    21
       1     4    10    20    35    56
       1     5    15    35    70   126
       1     6    21    56   126   252
```

The elements of A are binomial coefficients. Each element is the sum of its north and west neighbors. The Cholesky factorization is

```
R = chol(A)

R =
     1     1     1     1     1     1
     0     1     2     3     4     5
     0     0     1     3     6    10
     0     0     0     1     4    10
     0     0     0     0     1     5
     0     0     0     0     0     1
```

The elements are again binomial coefficients. The fact that `R'*R` is equal to `A` demonstrates an identity involving sums of products of binomial coefficients.

Note The Cholesky factorization also applies to complex matrices. Any complex matrix that has a Cholesky factorization satisfies

$A' = A$

and is said to be *Hermitian positive definite*.

The Cholesky factorization allows the linear system

$Ax = b$

to be replaced by

$R'Rx = b.$

Because the backslash operator recognizes triangular systems, this can be solved in the MATLAB environment quickly with

```
x = R\(R'\b)
```

If `A` is *n*-by-*n*, the computational complexity of `chol(A)` is $O(n^3)$, but the complexity of the subsequent backslash solutions is only $O(n^2)$.

LU Factorization

LU factorization, or Gaussian elimination, expresses any square matrix *A* as the product of a permutation of a lower triangular matrix and an upper triangular matrix

$A = LU,$

where *L* is a permutation of a lower triangular matrix with ones on its diagonal and *U* is an upper triangular matrix.

The permutations are necessary for both theoretical and computational reasons. The matrix

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

cannot be expressed as the product of triangular matrices without interchanging its two rows. Although the matrix

$$\begin{bmatrix} \varepsilon & 1 \\ 1 & 0 \end{bmatrix}$$

can be expressed as the product of triangular matrices, when ε is small, the elements in the factors are large and magnify errors, so even though the permutations are not strictly necessary, they are desirable. Partial pivoting ensures that the elements of *L* are bounded by one in magnitude and that the elements of *U* are not much larger than those of *A*.

For example:

```
[L,U] = lu(B)

L =
    1.0000         0         0
    0.3750    0.5441    1.0000
    0.5000    1.0000         0

U =
    8.0000    1.0000    6.0000
         0    8.5000   -1.0000
         0         0    5.2941
```

The LU factorization of `A` allows the linear system

```
A*x = b
```

to be solved quickly with

```
x = U\(L\b)
```

Determinants and inverses are computed from the LU factorization using

```
det(A) = det(L)*det(U)
```

and

```
inv(A) = inv(U)*inv(L)
```

You can also compute the determinants using `det(A) = prod(diag(U))`, though the signs of the determinants might be reversed.

QR Factorization

An *orthogonal* matrix, or a matrix with orthonormal columns, is a real matrix whose columns all have unit length and are perpendicular to each other. If Q is orthogonal, then

$Q^{T}Q = I$,

where I is the identity matrix.

The simplest orthogonal matrices are two-dimensional coordinate rotations:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}.$$

For complex matrices, the corresponding term is *unitary*. Orthogonal and unitary matrices are desirable for numerical computation because they preserve length, preserve angles, and do not magnify errors.

The orthogonal, or QR, factorization expresses any rectangular matrix as the product of an orthogonal or unitary matrix and an upper triangular matrix. A column permutation might also be involved:

$A = QR$

or

$AP = QR$,

where Q is orthogonal or unitary, R is upper triangular, and P is a permutation.

There are four variants of the QR factorization—full or economy size, and with or without column permutation.

Overdetermined linear systems involve a rectangular matrix with more rows than columns, that is m-by-n with $m > n$. The full-size QR factorization produces a square, m-by-m orthogonal Q and a rectangular m-by-n upper triangular R:

```
C=gallery('uniformdata',[5 4], 0);
[Q,R] = qr(C)

Q =

    0.6191    0.1406   -0.1899   -0.5058    0.5522
    0.1506    0.4084    0.5034    0.5974    0.4475
    0.3954   -0.5564    0.6869   -0.1478   -0.2008
    0.3167    0.6676    0.1351   -0.1729   -0.6370
    0.5808   -0.2410   -0.4695    0.5792   -0.2207


R =

    1.5346    1.0663    1.2010    1.4036
         0    0.7245    0.3474   -0.0126
         0         0    0.9320    0.6596
         0         0         0    0.6648
         0         0         0         0
```

In many cases, the last $m - n$ columns of Q are not needed because they are multiplied by the zeros in the bottom portion of R. So the economy-size QR factorization produces a rectangular, m-by-n Q with orthonormal columns and a square n-by-n upper triangular R. For the 5-by-4 example, this is not much of a saving, but for larger, highly rectangular matrices, the savings in both time and memory can be quite important:

```
[Q,R] = qr(C,0)
Q =

    0.6191    0.1406   -0.1899   -0.5058
    0.1506    0.4084    0.5034    0.5974
    0.3954   -0.5564    0.6869   -0.1478
    0.3167    0.6676    0.1351   -0.1729
    0.5808   -0.2410   -0.4695    0.5792


R =

    1.5346    1.0663    1.2010    1.4036
         0    0.7245    0.3474   -0.0126
         0         0    0.9320    0.6596
         0         0         0    0.6648
```

In contrast to the LU factorization, the QR factorization does not require any pivoting or permutations. But an optional column permutation, triggered by the presence of a third output argument, is useful for detecting singularity or rank deficiency. At each step of the factorization, the column of the remaining unfactored matrix with largest norm is used as the basis for that step. This ensures that the diagonal elements of *R* occur in decreasing order and that any linear dependence among the columns is almost certainly be revealed by examining these elements. For the small example given here, the second column of C has a larger norm than the first, so the two columns are exchanged:

```
[Q,R,P] = qr(C)

Q =
   -0.3522    0.8398   -0.4131
   -0.7044   -0.5285   -0.4739
   -0.6163    0.1241    0.7777

R =
  -11.3578   -8.2762
         0    7.2460
         0         0

P =
     0     1
     1     0
```

When the economy-size and column permutations are combined, the third output argument is a permutation vector, rather than a permutation matrix:

```
[Q,R,p] = qr(C,0)

Q =
   -0.3522    0.8398
   -0.7044   -0.5285
   -0.6163    0.1241

R =
  -11.3578   -8.2762
         0    7.2460


p =
     2     1
```

The QR factorization transforms an overdetermined linear system into an equivalent triangular system. The expression

```
norm(A*x - b)
```

equals

```
norm(Q*R*x - b)
```

Multiplication by orthogonal matrices preserves the Euclidean norm, so this expression is also equal to

```
norm(R*x - y)
```

where y = Q'*b. Since the last *m*-*n* rows of *R* are zero, this expression breaks into two pieces:

```
norm(R(1:n,1:n)*x - y(1:n))
```

and

```
norm(y(n+1:m))
```

When `A` has full rank, it is possible to solve for `x` so that the first of these expressions is zero. Then the second expression gives the norm of the residual. When `A` does not have full rank, the triangular structure of `R` makes it possible to find a basic solution to the least-squares problem.

Using Multithreaded Computation for Factorization

MATLAB software supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads. For a function or expression to execute faster on multiple CPUs, a number of conditions must be true:

1. The function performs operations that easily partition into sections that execute concurrently. These sections must be able to execute with little communication between processes. They should require few sequential operations.
2. The data size is large enough so that any advantages of concurrent execution outweigh the time required to partition the data and manage separate execution threads. For example, most functions speed up only when the array contains several thousand elements or more.
3. The operation is not memory-bound; processing time is not dominated by memory access time. As a general rule, complicated functions speed up more than simple functions.

`lu` and `qr` show significant increase in speed on large double-precision arrays (on order of 10,000 elements).

Powers and Exponentials

This topic shows how to compute matrix powers and exponentials using a variety of methods.

Positive Integer Powers

If `A` is a square matrix and `p` is a positive integer, then `A^p` effectively multiplies `A` by itself `p-1` times. For example:

```
A = [1 1 1
     1 2 3
     1 3 6];
A^2
```

```
ans = 3×3

     3     6    10
     6    14    25
    10    25    46
```

Inverse and Fractional Powers

If `A` is square and nonsingular, then `A^(-p)` effectively multiplies `inv(A)` by itself `p-1` times.

```
A^(-3)
```

```
ans = 3×3

  145.0000 -207.0000   81.0000
 -207.0000  298.0000 -117.0000
   81.0000 -117.0000   46.0000
```

MATLAB® calculates `inv(A)` and `A^(-1)` with the same algorithm, so the results are exactly the same. Both `inv(A)` and `A^(-1)` produce warnings if the matrix is close to being singular.

```
isequal(inv(A),A^(-1))
```

```
ans = logical
   1
```

Fractional powers, such as `A^(2/3)`, are also permitted. The results using fractional powers depend on the distribution of the eigenvalues of the matrix.

```
A^(2/3)
```

```
ans = 3×3

    0.8901    0.5882    0.3684
    0.5882    1.2035    1.3799
    0.3684    1.3799    3.1167
```

Element-by-Element Powers

The `.^` operator calculates element-by-element powers. For example, to square each element in a matrix you can use `A.^2`.

```
A.^2
```

```
ans = 3×3

     1     1     1
     1     4     9
     1     9    36
```

Square Roots

The `sqrt` function is a convenient way to calculate the square root of each element in a matrix. An alternate way to do this is `A.^(1/2)`.

```
sqrt(A)
```

```
ans = 3×3

    1.0000    1.0000    1.0000
    1.0000    1.4142    1.7321
    1.0000    1.7321    2.4495
```

For other roots, you can use `nthroot`. For example, calculate `A.^(1/3)`.

```
nthroot(A,3)
```

```
ans = 3×3

    1.0000    1.0000    1.0000
    1.0000    1.2599    1.4422
    1.0000    1.4422    1.8171
```

These element-wise roots differ from the matrix square root, which calculates a second matrix *B* such that *A* = BB. The function `sqrtm(A)` computes `A^(1/2)` by a more accurate algorithm. The `m` in `sqrtm` distinguishes this function from `sqrt(A)`, which, like `A.^(1/2)`, does its job element-by-element.

```
B = sqrtm(A)
```

```
B = 3×3

    0.8775    0.4387    0.1937
    0.4387    1.0099    0.8874
    0.1937    0.8874    2.2749
```

```
B^2
```

```
ans = 3×3

    1.0000    1.0000    1.0000
    1.0000    2.0000    3.0000
    1.0000    3.0000    6.0000
```

Scalar Bases

In addition to raising a matrix to a power, you also can raise a scalar to the power of a matrix.

```
2^A
```

```
ans = 3×3

   10.4630   21.6602   38.5862
   21.6602   53.2807   94.6010
   38.5862   94.6010  173.7734
```

When you raise a scalar to the power of a matrix, MATLAB uses the eigenvalues and eigenvectors of the matrix to calculate the matrix power. If `[V,D] = eig(A)`, then $2^A = V\,2^D\,V^{-1}$.

```
[V,D] = eig(A);
V*2^D*V^(-1)
```

```
ans = 3×3

   10.4630   21.6602   38.5862
   21.6602   53.2807   94.6010
   38.5862   94.6010  173.7734
```

Matrix Exponentials

The matrix exponential is a special case of raising a scalar to a matrix power. The base for a matrix exponential is Euler's number `e = exp(1)`.

```
e = exp(1);
e^A
```

```
ans = 3×3
10^3 ×

    0.1008    0.2407    0.4368
    0.2407    0.5867    1.0654
    0.4368    1.0654    1.9418
```

The `expm` function is a more convenient way to calculate matrix exponentials.

```
expm(A)
```

```
ans = 3×3
10^3 ×

    0.1008    0.2407    0.4368
    0.2407    0.5867    1.0654
    0.4368    1.0654    1.9418
```

The matrix exponential can be calculated in a number of ways. See “Matrix Exponentials” on page 2-38 for more information.

Dealing with Small Numbers

The MATLAB functions `log1p` and `expm1` calculate $\log(1 + x)$ and $e^x - 1$ accurately for very small values of x. For example, if you try to add a number smaller than machine precision to 1, then the result gets rounded to 1.

```
log(1+eps/2)
```

```
ans = 0
```

However, `log1p` is able to return a more accurate answer.

```
log1p(eps/2)
```

```
ans = 1.1102e-16
```

Likewise for $e^x - 1$, if x is very small then it is rounded to zero.

```
exp(eps/2)-1
```

```
ans = 0
```

Again, `expm1` is able to return a more accurate answer.

```
expm1(eps/2)
```

```
ans = 1.1102e-16
```

Eigenvalues

In this section...

Eigenvalue Decomposition

An *eigenvalue* and *eigenvector* of a square matrix A are, respectively, a scalar λ and a nonzero vector υ that satisfy

$A\upsilon = \lambda\upsilon$.

With the eigenvalues on the diagonal of a diagonal matrix Λ and the corresponding eigenvectors forming the columns of a matrix V, you have

$AV = V\Lambda$.

If V is nonsingular, this becomes the eigenvalue decomposition

$A = V\Lambda V^{-1}$.

A good example is the coefficient matrix of the differential equation $dx/dt = Ax$:

```
A =
     0    -6    -1
     6     2   -16
    -5    20   -10
```

The solution to this equation is expressed in terms of the matrix exponential $x(t) = e^{tA}x(0)$. The statement

```
lambda = eig(A)
```

produces a column vector containing the eigenvalues of `A`. For this matrix, the eigenvalues are complex:

```
lambda =
     -3.0710
     -2.4645+17.6008i
     -2.4645-17.6008i
```

The real part of each of the eigenvalues is negative, so $e^{\lambda t}$ approaches zero as t increases. The nonzero imaginary part of two of the eigenvalues, $\pm\omega$, contributes the oscillatory component, $\sin(\omega t)$, to the solution of the differential equation.

With two output arguments, `eig` computes the eigenvectors and stores the eigenvalues in a diagonal matrix:

```
[V,D] = eig(A)
```

```
V =
  -0.8326         0.2003 - 0.1394i   0.2003 + 0.1394i
  -0.3553        -0.2110 - 0.6447i  -0.2110 + 0.6447i
  -0.4248        -0.6930            -0.6930

D =
  -3.0710                 0                 0
        0           -2.4645+17.6008i        0
        0                 0          -2.4645-17.6008i
```

The first eigenvector is real and the other two vectors are complex conjugates of each other. All three vectors are normalized to have Euclidean length, `norm(v,2)`, equal to one.

The matrix `V*D*inv(V)`, which can be written more succinctly as `V*D/V`, is within round-off error of `A`. And, `inv(V)*A*V`, or `V\A*V`, is within round-off error of `D`.

Multiple Eigenvalues

Some matrices do not have an eigenvector decomposition. These matrices are not diagonalizable. For example:

```
A = [ 1    -2    1
      0     1    4
      0     0    3 ]
```

For this matrix

```
[V,D] = eig(A)
```

produces

```
V =

    1.0000    1.0000   -0.5571
         0    0.0000    0.7428
         0         0    0.3714


D =

     1     0     0
     0     1     0
     0     0     3
```

There is a double eigenvalue at $\lambda = 1$. The first and second columns of `V` are the same. For this matrix, a full set of linearly independent eigenvectors does not exist.

Schur Decomposition

Many advanced matrix computations do not require eigenvalue decompositions. They are based, instead, on the Schur decomposition

$A = USU'$,

where U is an orthogonal matrix and S is a block upper-triangular matrix with 1-by-1 and 2-by-2 blocks on the diagonal. The eigenvalues are revealed by the diagonal elements and blocks of S, while the columns of *U* provide an orthogonal basis, which has much better numerical properties than a set of eigenvectors.

For example, compare the eigenvalue and Schur decompositions of this defective matrix:

```
A = [ 6    12    19
     -9   -20   -33
      4     9    15 ];

[V,D] = eig(A)
```

```
V =

  -0.4741 + 0.0000i  -0.4082 - 0.0000i  -0.4082 + 0.0000i
   0.8127 + 0.0000i   0.8165 + 0.0000i   0.8165 + 0.0000i
  -0.3386 + 0.0000i  -0.4082 + 0.0000i  -0.4082 - 0.0000i


D =

  -1.0000 + 0.0000i   0.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   1.0000 + 0.0000i   0.0000 + 0.0000i
   0.0000 + 0.0000i   0.0000 + 0.0000i   1.0000 - 0.0000i
```

```
[U,S] = schur(A)
```

```
U =

   -0.4741    0.6648    0.5774
    0.8127    0.0782    0.5774
   -0.3386   -0.7430    0.5774


S =

   -1.0000   20.7846  -44.6948
         0    1.0000   -0.6096
         0    0.0000    1.0000
```

The matrix `A` is defective since it does not have a full set of linearly independent eigenvectors (the second and third columns of `V` are the same). Since not all columns of `V` are linearly independent, it has a large condition number of about `~1e8`. However, `schur` is able to calculate three different basis vectors in `U`. Since `U` is orthogonal, `cond(U) = 1`.

The matrix `S` has the real eigenvalue as the first entry on the diagonal and the repeated eigenvalue represented by the lower right 2-by-2 block. The eigenvalues of the 2-by-2 block are also eigenvalues of `A`:

```
eig(S(2:3,2:3))
```

```
ans =

   1.0000 + 0.0000i
   1.0000 - 0.0000i
```

Singular Values

A singular value and corresponding singular vectors of a rectangular matrix A are, respectively, a scalar σ and a pair of vectors u and v that satisfy

$$Av = \sigma u$$
$$A^H u = \sigma v,$$

where A^H is the Hermitian transpose of A. The singular vectors u and v are typically scaled to have a norm of 1. Also, if u and v are singular vectors of A, then $-u$ and $-v$ are singular vectors of `A` as well.

The singular values σ are always real and nonnegative, even if A is complex. With the singular values in a diagonal matrix Σ and the corresponding singular vectors forming the columns of two orthogonal matrices U and V, you obtain the equations

$$AV = U\Sigma$$
$$A^H U = V\Sigma .$$

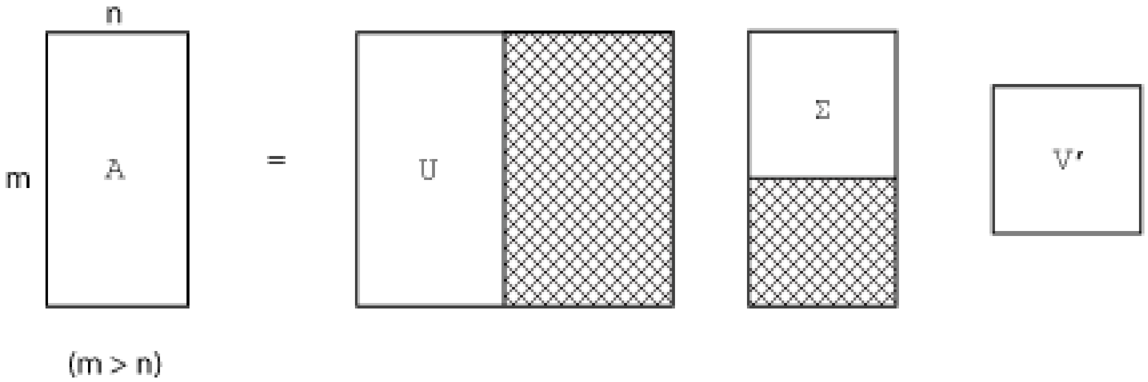
Since U and V are unitary matrices, multiplying the first equation by V^H on the right yields the singular value decomposition equation

$$A = U\Sigma V^H .$$

The full singular value decomposition of an m-by-n matrix involves:

- m-by-m matrix U
- m-by-n matrix Σ
- n-by-n matrix V

In other words, U and V are both square, and Σ is the same size as A. If A has many more rows than columns (`m > n`), then the resulting `m`-by-`m` matrix U is large. However, most of the columns in U are multiplied by zeros in Σ. In this situation, the *economy-sized* decomposition saves both time and storage by producing an m-by-n U, an n-by-n Σ and the same V:



The eigenvalue decomposition is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space into itself, as it does for an ordinary differential equation. However, the singular value decomposition is the appropriate tool for analyzing a mapping from one vector space into another vector space, possibly with a different dimension. Most systems of simultaneous linear equations fall into this second category.

If *A* is square, symmetric, and positive definite, then its eigenvalue and singular value decompositions are the same. But, as *A* departs from symmetry and positive definiteness, the difference between the two decompositions increases. In particular, the singular value decomposition of a real matrix is always real, but the eigenvalue decomposition of a real, nonsymmetric matrix might be complex.

For the example matrix

```
A =
     9     4
     6     8
     2     7
```

the full singular value decomposition is

```
[U,S,V] = svd(A)

U =

    0.6105   -0.7174    0.3355
    0.6646    0.2336   -0.7098
    0.4308    0.6563    0.6194


S =

   14.9359         0
         0    5.1883
         0         0


V =

    0.6925   -0.7214
    0.7214    0.6925
```

You can verify that `U*S*V'` is equal to `A` to within round-off error. For this small problem, the economy size decomposition is only slightly smaller.

```
[U,S,V] = svd(A,0)

U =

    0.6105   -0.7174
    0.6646    0.2336
    0.4308    0.6563


S =

   14.9359         0
         0    5.1883


V =
    0.6925   -0.7214
    0.7214    0.6925
```

Again, `U*S*V'` is equal to `A` to within round-off error.

Low-Rank SVD Approximations

For large sparse matrices, using `svd` to calculate *all* of the singular values and singular vectors is not always practical. For example, if you need to know just a few of the largest singular values, then calculating all of the singular values of a 5000-by-5000 sparse matrix is extra work.

In cases where only a subset of the singular values and singular vectors are required, the `svds` and `svdsketch` functions are preferred over `svd`.

Function	Usage
`svds`	Use `svds` to calculate a rank-*k* approximation of the SVD. You can specify whether the subset of singular values should be the largest, the smallest, or the closest to a specific number. `svds` generally calculates the best possible rank-*k* approximation.
`svdsketch`	Use `svdsketch` to calculate a partial SVD of the input matrix satisfying a specified tolerance. While `svds` requires that you specify the rank, `svdsketch` adaptively determines the rank of the matrix sketch based on the specified tolerance. The rank-*k* approximation that `svdsketch` ultimately uses satisfies the tolerance, but unlike `svds`, it is not guaranteed to be the best one possible.

For example, consider a 1000-by-1000 random sparse matrix with a density of about 30%.

```
n = 1000;
A = sprand(n,n,0.3);
```

The six largest singular values are

```
S = svds(A)

S =

  130.2184
   16.4358
   16.4119
   16.3688
   16.3242
   16.2838
```

Also, the six smallest singular values are

```
S = svds(A,6,'smallest')

S =
    0.0740
    0.0574
    0.0388
    0.0282
    0.0131
    0.0066
```

For smaller matrices that can fit in memory as a full matrix, `full(A)`, using `svd(full(A))` might still be quicker than `svds` or `svdsketch`. However, for truly large and sparse matrices, using `svds` or `svdsketch` becomes necessary.

LAPACK in MATLAB

LAPACK (Linear Algebra Package) is a library of routines that provides fast, robust algorithms for numerical linear algebra and matrix computations. Linear algebra functions and matrix operations in MATLAB are built on LAPACK, and they continue to benefit from the performance and accuracy of its routines.

A Brief History

MATLAB started its life in the late 1970s as an interactive calculator built on top of LINPACK and EISPACK, which were the state-of-the-art Fortran subroutine libraries for matrix computation of the time. For many years MATLAB used translations to C of about a dozen Fortran subroutines from LINPACK and EISPACK.

In the year 2000, MATLAB migrated to using LAPACK, which is the modern replacement for LINPACK and EISPACK. It is a large, multi-author, Fortran library for numerical linear algebra. LAPACK was originally intended for use on supercomputers because of its ability to operate on several columns of a matrix at a time. The speed of LAPACK routines is closely connected to the speed of the Basic Linear Algebra Subroutines (BLAS). The BLAS version is typically hardware-specific and highly optimized.

See Also

More About

- “Call LAPACK and BLAS Functions”

External Websites

- MATLAB Incorporates LAPACK

Matrix Exponentials

This example shows 3 of the 19 ways to compute the exponential of a matrix.

For background on the computation of matrix exponentials, see:

Moler, C. and C. Van Loan. "Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later." *SIAM Review.* Vol. 45, Number 1, 2003, pp. 3-49.

Start by creating a matrix `A`.

```
A = [0 1 2; 0.5 0 1; 2 1 0]
```

```
A = 3×3

         0    1.0000    2.0000
    0.5000         0    1.0000
    2.0000    1.0000         0
```

```
Asave = A;
```

Method 1: Scaling and Squaring

`expmdemo1` is an implementation of algorithm 11.3.1 in the book:

Golub, Gene H. and Charles Van Loan. *Matrix Computations, 3rd edition.* Baltimore, MD: Johns Hopkins University Press, 1996.

```
% Scale A by power of 2 so that its norm is < 1/2 .
[f,e] = log2(norm(A,'inf'));
s = max(0,e+1);
A = A/2^s;

% Pade approximation for exp(A)
X = A;
c = 1/2;
E = eye(size(A)) + c*A;
D = eye(size(A)) - c*A;
q = 6;
p = 1;
for k = 2:q
   c = c * (q-k+1) / (k*(2*q-k+1));
   X = A*X;
   cX = c*X;
   E = E + cX;
   if p
     D = D + cX;
   else
     D = D - cX;
   end
   p = ~p;
end
E = D\E;

% Undo scaling by repeated squaring
for k = 1:s
    E = E*E;
```

```
end

E1 = E
```

```
E1 = 3×3

    5.3091    4.0012    5.5778
    2.8088    2.8845    3.1930
    5.1737    4.0012    5.7132
```

Method 2: Taylor Series

`expmdemo2` uses the classic definition for the matrix exponential given by the power series

$$e^A = \sum_{k=0}^{\infty} \frac{1}{k!} A^k \, .$$

A^0 is the identity matrix with the same dimensions as A. As a practical numerical method, this approach is slow and inaccurate if `norm(A)` is too large.

```
A = Asave;

% Taylor series for exp(A)
E = zeros(size(A));
F = eye(size(A));
k = 1;

while norm(E+F-E,1) > 0
   E = E + F;
   F = A*F/k;
   k = k+1;
end

E2 = E
```

```
E2 = 3×3

    5.3091    4.0012    5.5778
    2.8088    2.8845    3.1930
    5.1737    4.0012    5.7132
```

Method 3: Eigenvalues and Eigenvectors

`expmdemo3` assumes that the matrix has a full set of eigenvectors V such that $A = \mathrm{VD}V^{-1}$. The matrix exponential can be calculated by exponentiating the diagonal matrix of eigenvalues:

$$e^A = Ve^D V^{-1} \, .$$

As a practical numerical method, the accuracy is determined by the condition of the eigenvector matrix.

```
A = Asave;

[V,D] = eig(A);
E = V * diag(exp(diag(D))) / V;
```

```
E3 = E
```

```
E3 = 3×3

    5.3091    4.0012    5.5778
    2.8088    2.8845    3.1930
    5.1737    4.0012    5.7132
```

Compare Results

For the matrix in this example, all three methods work equally well.

```
E = expm(Asave);
err1 = E - E1
```

```
err1 = 3×3
10^-14 ×

    0.3553    0.1776    0.0888
    0.0888    0.1332   -0.0444
         0         0   -0.2665
```

```
err2 = E - E2
```

```
err2 = 3×3
10^-14 ×

         0         0   -0.1776
   -0.0444         0   -0.0888
    0.1776         0    0.0888
```

```
err3 = E - E3
```

```
err3 = 3×3
10^-13 ×

   -0.0711   -0.0444   -0.0799
   -0.0622   -0.0488   -0.0933
   -0.0711   -0.0533   -0.1066
```

Taylor Series Failure

For some matrices the terms in the Taylor series become very large before they go to zero. Consequently, `expmdemo2` fails.

```
A = [-147 72; -192 93];
E1 = expmdemo1(A)
```

```
E1 = 2×2

   -0.0996    0.0747
   -0.1991    0.1494
```

```
E2 = expmdemo2(A)
```

```
E2 = 2×2
10^6 ×

   -1.1985   -0.5908
   -2.7438   -2.0442
```

```
E3 = expmdemo3(A)
```

```
E3 = 2×2

   -0.0996    0.0747
   -0.1991    0.1494
```

Eigenvalues and Eigenvectors Failure

Here is a matrix that does not have a full set of eigenvectors. Consequently, `expmdemo3` fails.

```
A = [-1 1; 0 -1];
E1 = expmdemo1(A)
```

```
E1 = 2×2

    0.3679    0.3679
         0    0.3679
```

```
E2 = expmdemo2(A)
```

```
E2 = 2×2

    0.3679    0.3679
         0    0.3679
```

```
E3 = expmdemo3(A)
```

```
E3 = 2×2

    0.3679         0
         0    0.3679
```

See Also

`expm`

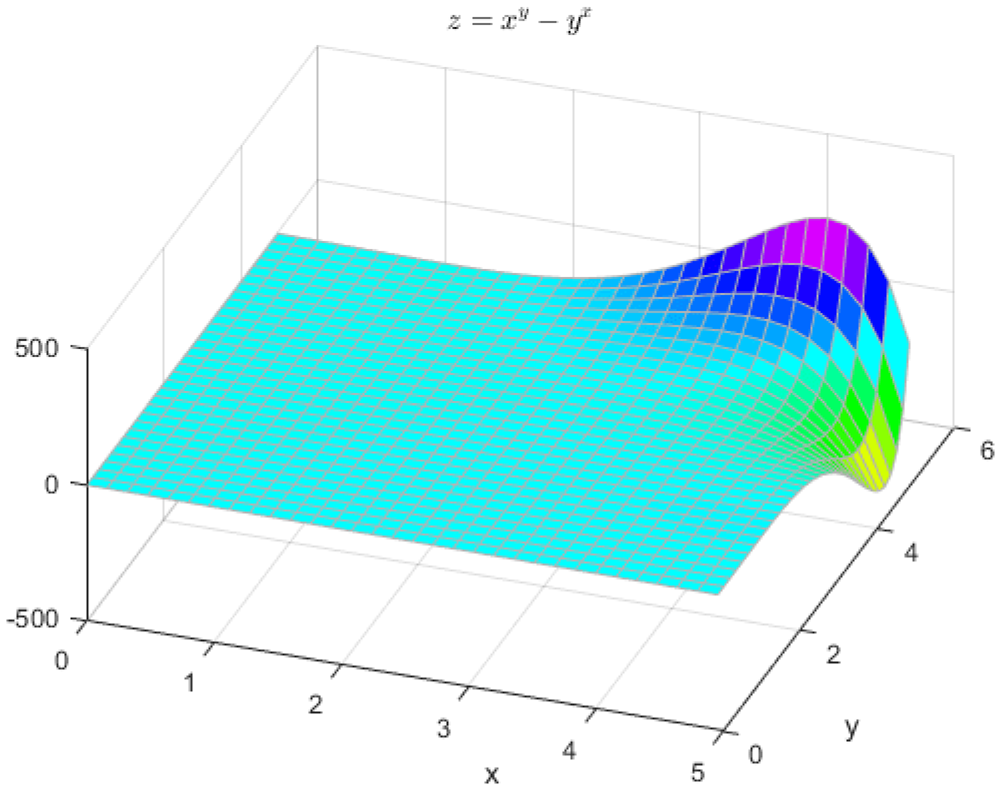
Graphical Comparison of Exponential Functions

This example shows an interesting graphical approach for discovering whether e^{π} is greater than π^{e}.

The question is: which is greater, e^{π} or π^{e}? The easy way to find out is to type it directly at the MATLAB® command prompt. But another way to analyze the situation is to ask a more general question: what is the shape of the function $z(x, y) = x^{y} - y^{x}$?

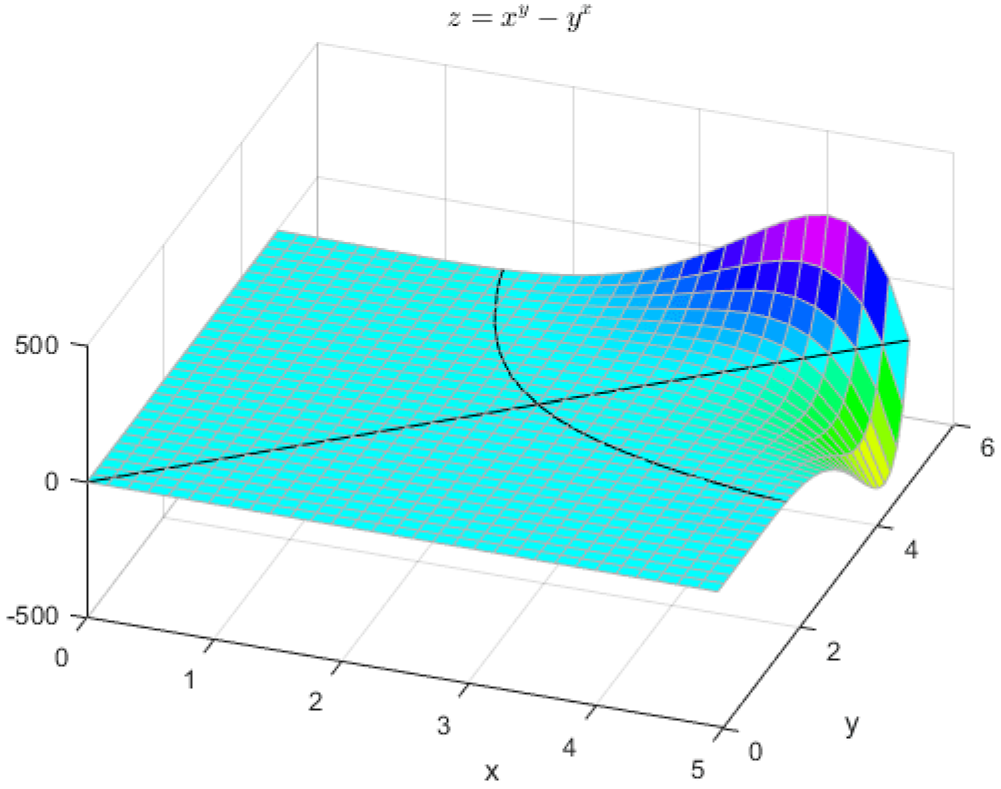
Here is a plot of z.

```
% Define the mesh
x = 0:0.16:5;
y = 0:0.16:5;
[xx,yy] = meshgrid(x,y);

% The plot
zz = xx.^yy-yy.^xx;
h = surf(x,y,zz);
h.EdgeColor = [0.7 0.7 0.7];
view(20,50);
colormap(hsv);
title('$z = x^y-y^x$','Interpreter','latex')
xlabel('x')
ylabel('y')
hold on
```



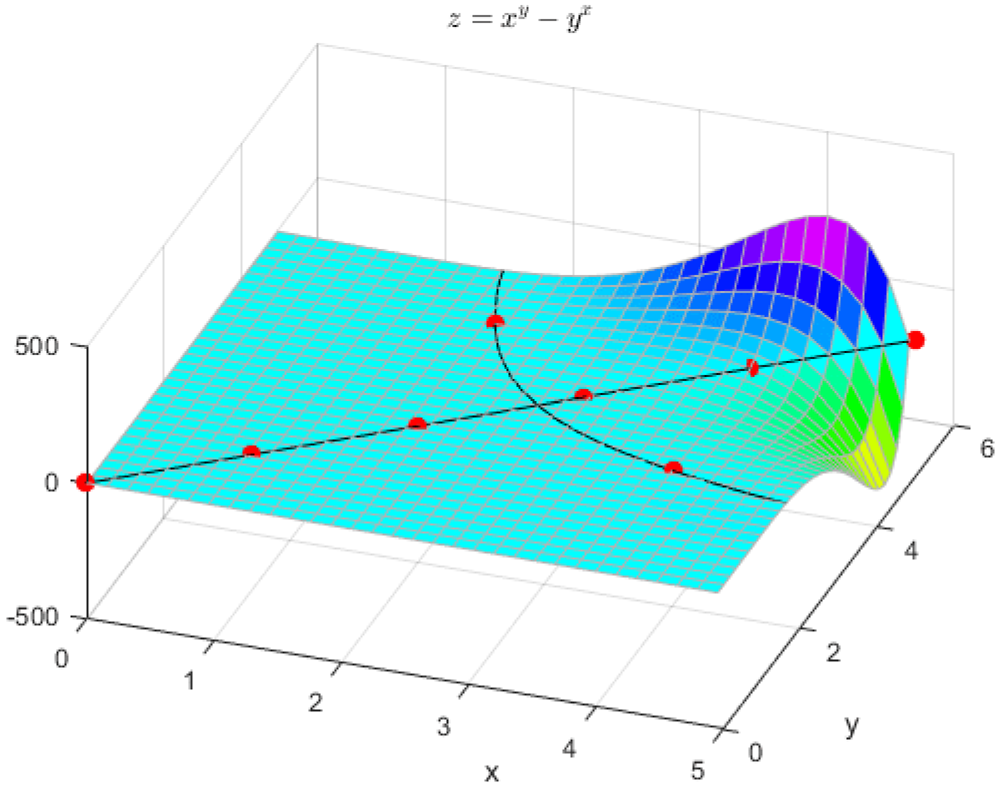
The solution of the equation $x^y - y^x = 0$ has a very interesting shape, and our original question is not easily solved by inspection. Here is a plot of the xy values that yield $z = 0$.

```
c = contourc(x,y,zz,[0 0]);
list1Len = c(2,1);
xContour = [c(1,2:1+list1Len) NaN c(1,3+list1Len:size(c,2))];
yContour = [c(2,2:1+list1Len) NaN c(2,3+list1Len:size(c,2))];
% Note that the NAN above prevents the end of the first contour line from being
% connected to the beginning of the second line
line(xContour,yContour,'Color','k');
```



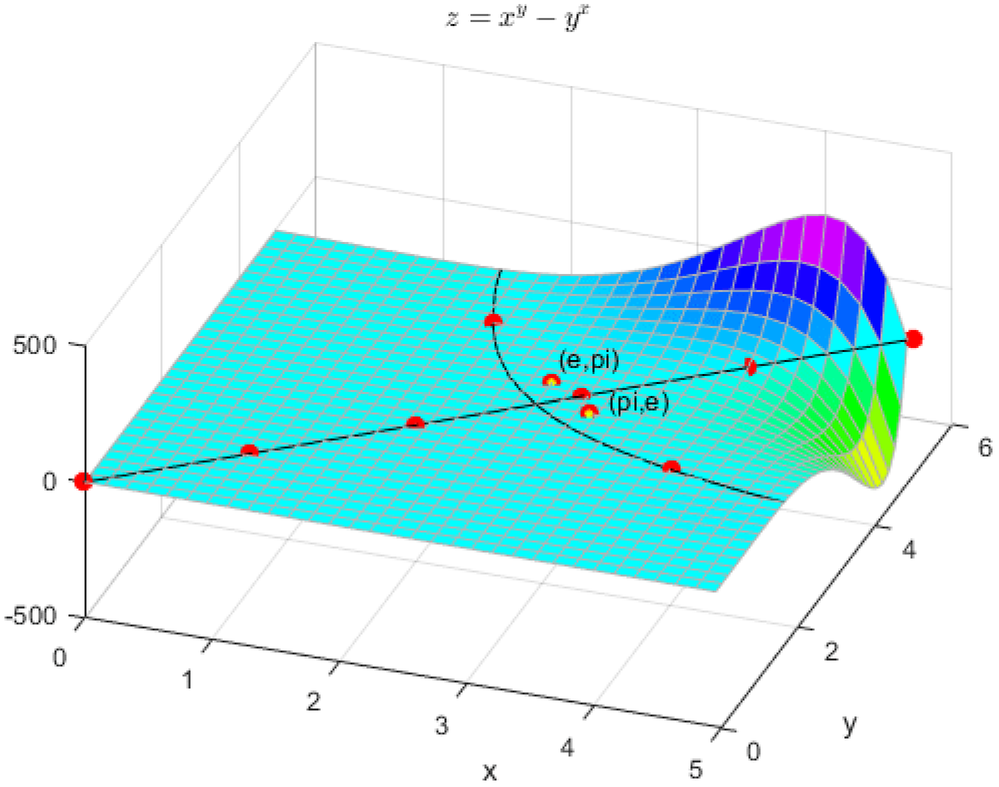
Some combinations of x and y along the black curve are integers. This next plot is of the integer solutions to the equation $x^y - y^x = 0$. Notice that $2^4 = 4^2$ is the *only* integer solution where $x \neq y$.

```
plot([0:5 2 4],[0:5 4 2],'r.','MarkerSize',25);
```



Finally, plot the points (π, e) and (e, π) on the surface. The result shows that e^{π} is indeed larger than π^{e} (though not by much).

```
e = exp(1);
plot([e pi],[pi e],'r.','MarkerSize',25);
plot([e pi],[pi e],'y.','MarkerSize',10);
text(e,3.3,'(e,pi)','Color','k', ...
   'HorizontalAlignment','left','VerticalAlignment','bottom');
text(3.3,e,'(pi,e)','Color','k','HorizontalAlignment','left',...
   'VerticalAlignment','bottom');
hold off;
```



Verify the results.

```
e = exp(1);
e^pi
```

```
ans = 23.1407
```

```
pi^e
```

```
ans = 22.4592
```

See Also

exp | pi

Basic Matrix Operations

This example shows basic techniques and functions for working with matrices in the MATLAB® language.

First, let's create a simple vector with 9 elements called `a`.

```
a = [1 2 3 4 6 4 3 4 5]
```

```
a = 1×9

     1     2     3     4     6     4     3     4     5
```

Now let's add 2 to each element of our vector, `a`, and store the result in a new vector.

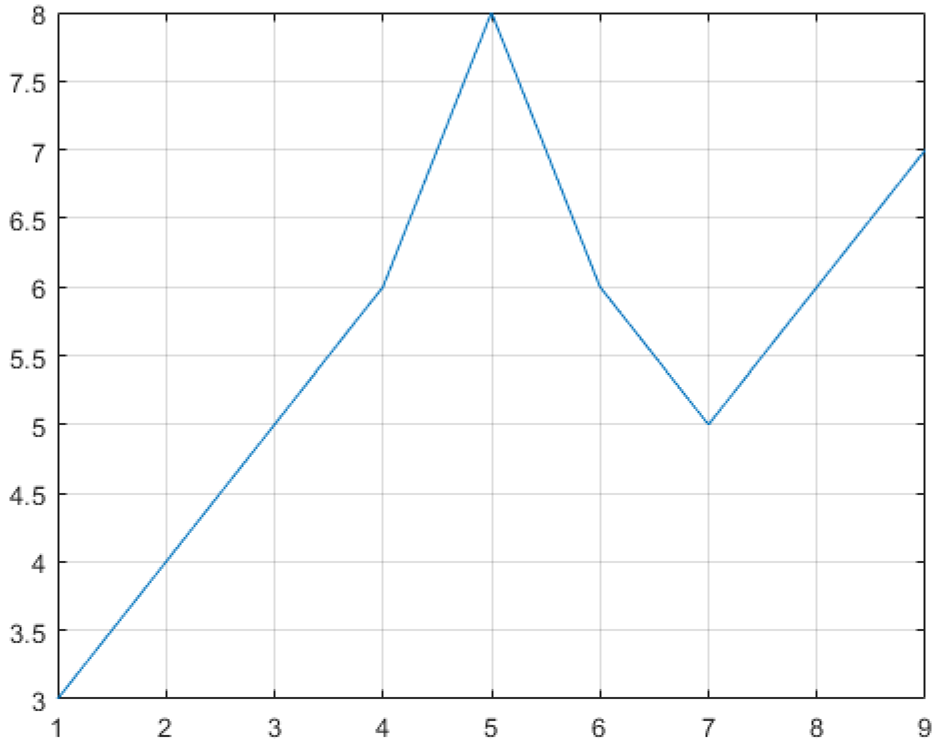
Notice how MATLAB requires no special handling of vector or matrix math.

```
b = a + 2
```

```
b = 1×9

     3     4     5     6     8     6     5     6     7
```

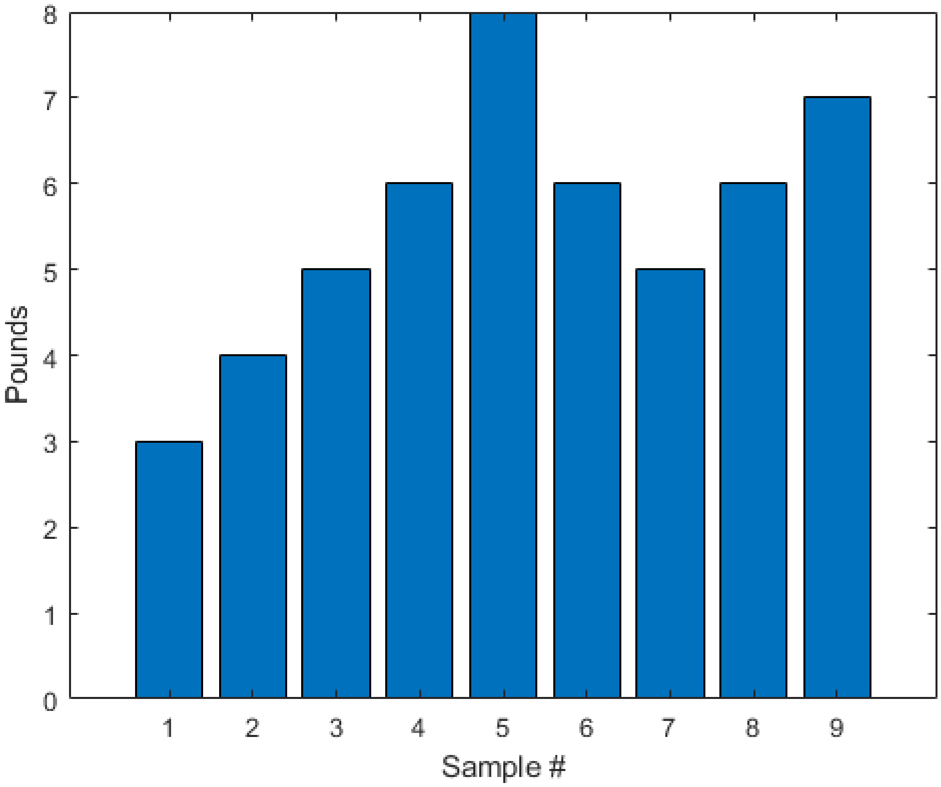
Creating graphs in MATLAB is as easy as one command. Let's plot the result of our vector addition with grid lines.

```
plot(b)
grid on
```



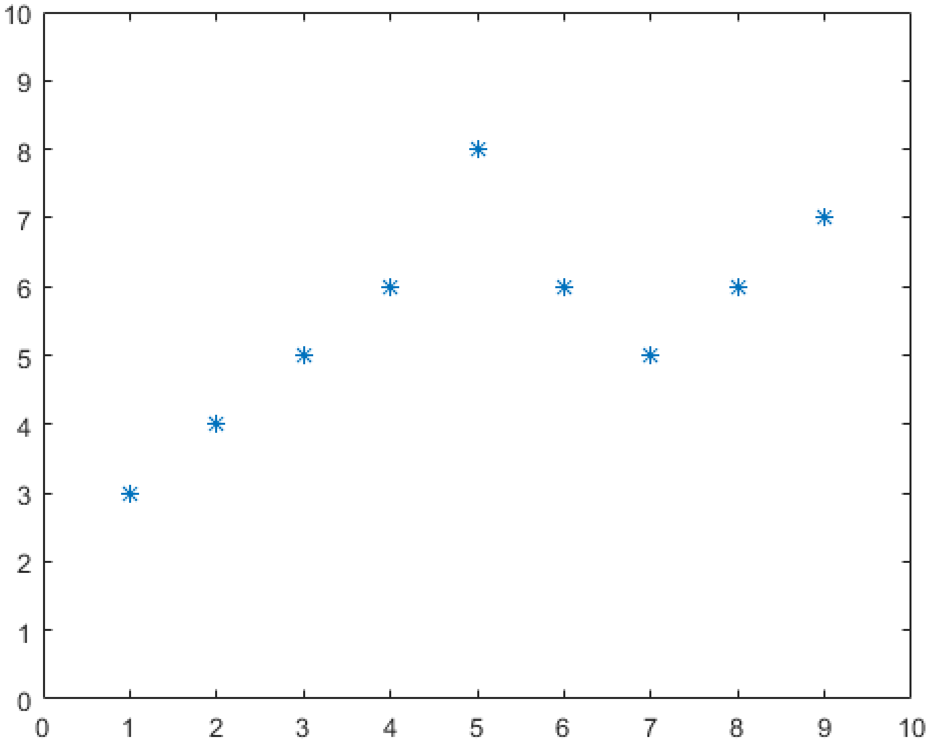
MATLAB can make other graph types as well, with axis labels.

```
bar(b)
xlabel('Sample #')
ylabel('Pounds')
```



MATLAB can use symbols in plots as well. Here is an example using stars to mark the points. MATLAB offers a variety of other symbols and line types.

```
plot(b,'*')
axis([0 10 0 10])
```



One area in which MATLAB excels is matrix computation.

Creating a matrix is as easy as making a vector, using semicolons (;) to separate the rows of a matrix.

```
A = [1 2 0; 2 5 -1; 4 10 -1]
```

```
A = 3×3

     1     2     0
     2     5    -1
     4    10    -1
```

We can easily find the transpose of the matrix `A`.

```
B = A'
```

```
B = 3×3

     1     2     4
     2     5    10
     0    -1    -1
```

Now let's multiply these two matrices together.

Note again that MATLAB doesn't require you to deal with matrices as a collection of numbers. MATLAB knows when you are dealing with matrices and adjusts your calculations accordingly.

```
C = A * B
```

```
C = 3×3

     5    12    24
    12    30    59
    24    59   117
```

Instead of doing a matrix multiply, we can multiply the corresponding elements of two matrices or vectors using the .* operator.

```
C = A .* B
```

```
C = 3×3

     1     4     0
     4    25   -10
     0   -10     1
```

Let's use the matrix A to solve the equation, A*x = b. We do this by using the \ (backslash) operator.

```
b = [1;3;5]
```

```
b = 3×1

     1
     3
     5
```

```
x = A\b
```

```
x = 3×1

     1
     0
    -1
```

Now we can show that A*x is equal to b.

```
r = A*x - b
```

```
r = 3×1

     0
     0
     0
```

MATLAB has functions for nearly every type of common matrix calculation.

There are functions to obtain eigenvalues ...

```
eig(A)
```

```
ans = 3×1
```

```
    3.7321
    0.2679
    1.0000
```

... as well as the singular values.

```
svd(A)
```

```
ans = 3×1

   12.3171
    0.5149
    0.1577
```

The "poly" function generates a vector containing the coefficients of the characteristic polynomial.

The characteristic polynomial of a matrix `A` is

$$det(\lambda I - A)$$

```
p = round(poly(A))
```

```
p = 1×4

     1    -5     5    -1
```

We can easily find the roots of a polynomial using the `roots` function.

These are actually the eigenvalues of the original matrix.

```
roots(p)
```

```
ans = 3×1

    3.7321
    1.0000
    0.2679
```

MATLAB has many applications beyond just matrix computation.

To convolve two vectors ...

```
q = conv(p,p)
```

```
q = 1×7

     1   -10    35   -52    35   -10     1
```

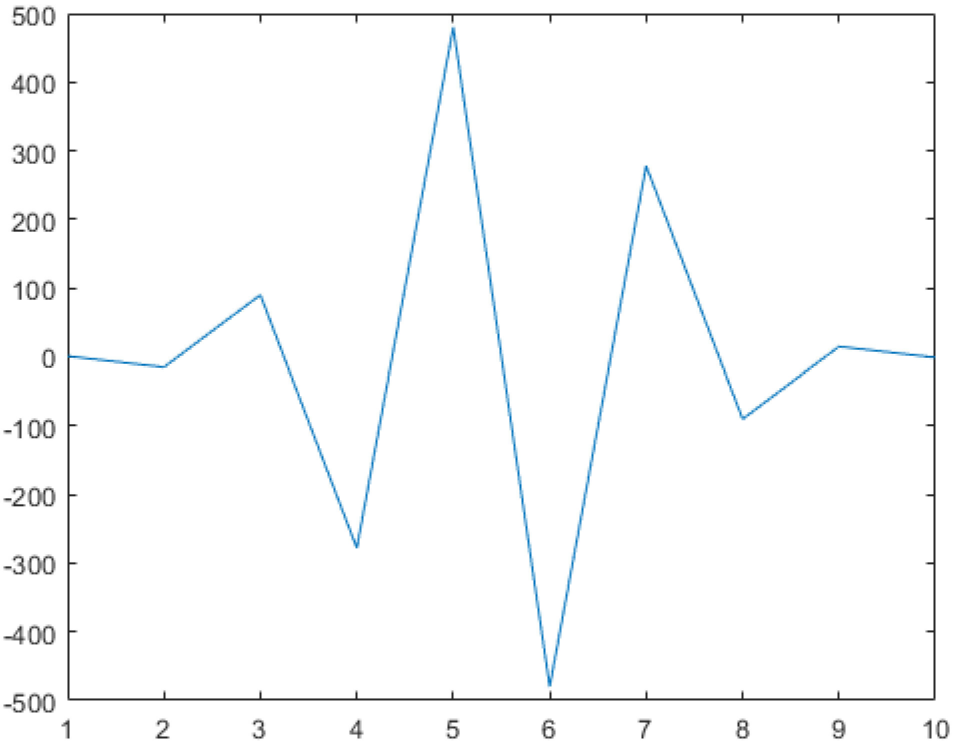
... or convolve again and plot the result.

```
r = conv(p,q)
```

```
r = 1×10
```

```
     1   -15    90  -278   480  -480   278   -90    15    -1
```

```
plot(r);
```



At any time, we can get a listing of the variables we have stored in memory using the `who` or `whos` command.

```
whos
```

```
  Name      Size            Bytes  Class     Attributes

  A         3x3                72  double              
  B         3x3                72  double              
  C         3x3                72  double              
  a         1x9                72  double              
  ans       3x1                24  double              
  b         3x1                24  double              
  p         1x4                32  double              
  q         1x7                56  double              
  r         1x10               80  double              
  x         3x1                24  double              
```

You can get the value of a particular variable by typing its name.

```
A
```

```
A = 3×3
```

```
     1     2     0
     2     5    -1
     4    10    -1
```

You can have more than one statement on a single line by separating each statement with commas or semicolons.

If you don't assign a variable to store the result of an operation, the result is stored in a temporary variable called `ans`.

```
sqrt(-1)
```

```
ans = 0.0000 + 1.0000i
```

As you can see, MATLAB easily deals with complex numbers in its calculations.

See Also

More About

- “Array vs. Matrix Operations”

Determine Whether Matrix Is Symmetric Positive Definite

This topic explains how to use the `chol` and `eig` functions to determine whether a matrix is symmetric positive definite (a symmetric matrix with all positive eigenvalues).

Method 1: Attempt Cholesky Factorization

The most efficient method to check whether a matrix is symmetric positive definite is to simply attempt to use `chol` on the matrix. If the factorization fails, then the matrix is not symmetric positive definite. This method does not require the matrix to be symmetric for a successful test (if the matrix is not symmetric, then the factorization fails).

```
A = [1 -1 0; -1 5 0; 0 0 7]
```

```
A = 3×3

     1    -1     0
    -1     5     0
     0     0     7
```

```
try chol(A)
    disp('Matrix is symmetric positive definite.')
catch ME
    disp('Matrix is not symmetric positive definite')
end
```

```
ans = 3×3

    1.0000   -1.0000         0
         0    2.0000         0
         0         0    2.6458
```

```
Matrix is symmetric positive definite.
```

The drawback of this method is that it cannot be extended to also check whether the matrix is symmetric positive semi-definite (where the eigenvalues can be positive or zero).

Method 2: Check Eigenvalues

While it is less efficient to use `eig` to calculate all of the eigenvalues and check their values, this method is more flexible since you can also use it to check whether a matrix is symmetric positive semi-definite. Still, for small matrices the difference in computation time between the methods is negligible to check whether a matrix is symmetric positive definite.

This method requires that you use `issymmetric` to check whether the matrix is symmetric before performing the test (if the matrix is not symmetric, then there is no need to calculate the eigenvalues).

```
tf = issymmetric(A)
```

```
tf = logical
   1
```

```
d = eig(A)
```

```
d = 3×1

    0.7639
    5.2361
    7.0000
```

```
isposdef = all(d > 0)
```

```
isposdef = logical
   1
```

You can extend this method to check whether a matrix is symmetric positive semi-definite with the command `all(d >= 0)`.

Numerical Considerations

The methods outlined here might give different results for the same matrix. Since both calculations involve round-off errors, each algorithm checks the definiteness of a matrix that is slightly different from `A`. In practice, the use of a tolerance is a more robust comparison method, since eigenvalues can be numerically zero within machine precision and be slightly positive or slightly negative.

For example, if a matrix has an eigenvalue on the order of `eps`, then using the comparison `isposdef = all(d > 0)` returns `true`, even though the eigenvalue is numerically zero and the matrix is better classified as symmetric positive *semi*-definite.

To perform the comparison using a tolerance, you can use the modified commands

```
tf = issymmetric(A)
d = eig(A)
isposdef = all(d > tol)
issemidef = all(d > -tol)
```

The tolerance defines a radius around zero, and any eigenvalues within that radius are treated as zeros. A good choice for the tolerance in most cases is `length(d)*eps(max(d))`, which takes into account the magnitude of the largest eigenvalue.

See Also

`chol` | `eig`

More About

- “Factorizations” on page 2-20

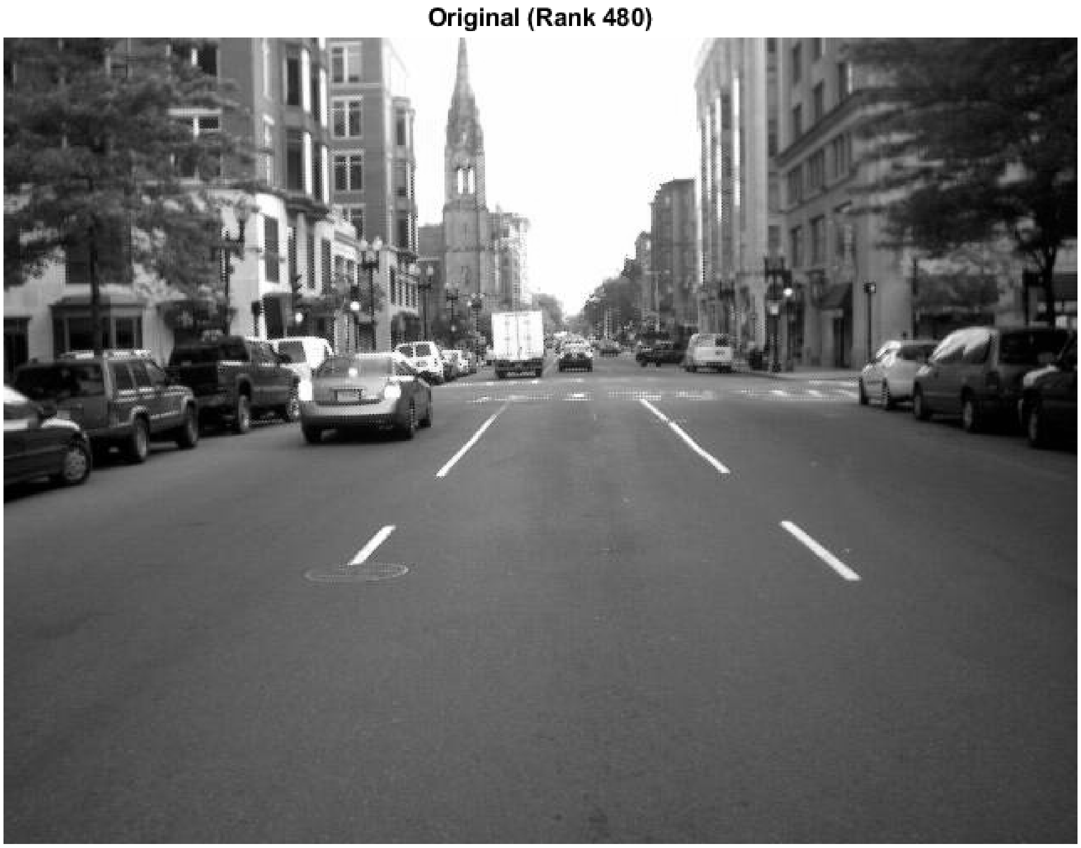
Image Compression with Low-Rank SVD

This example shows how to use svdsketch to compress an image. svdsketch uses a low-rank matrix approximation to preserve important features of the image, while filtering out less important features. As the tolerance used with svdsketch increases in magnitude, more features are filtered out, changing the level of detail in the image.

Load Image

Load the image street1.jpg, which is a picture of a city street. The 3-D matrix that forms this image is uint8, so convert the image to a grayscale matrix. View the image with an annotation of the original matrix rank.

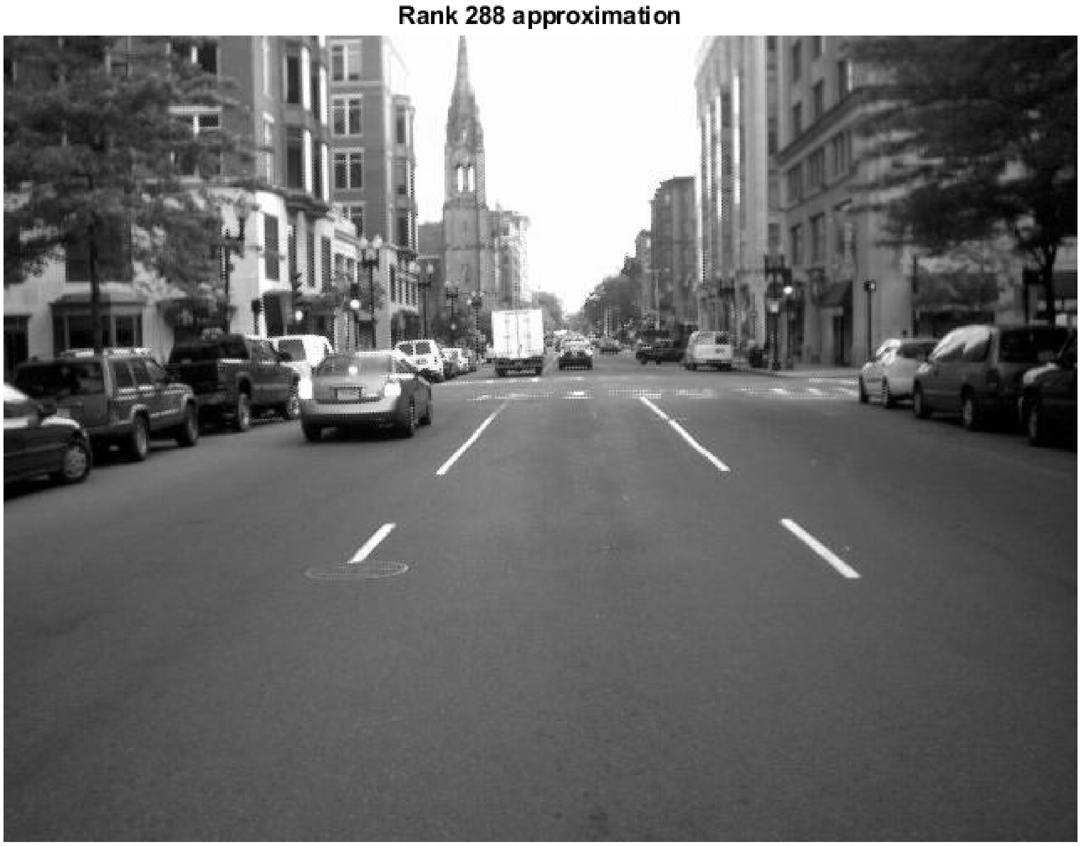
```
A = imread('street1.jpg');
A = rgb2gray(A);
imshow(A)
title(['Original (',sprintf('Rank %d)',rank(double(A)))])
```



Compress Image

Use svdsketch to calculate a low-rank matrix that approximates A within a tolerance of 1e-2. Form the low-rank matrix by multiplying the SVD factors returned by svdsketch, convert the result to uint8, and view the resulting image.

```
[U1,S1,V1] = svdsketch(double(A),1e-2);
Anew1 = uint8(U1*S1*V1');
imshow(uint8(Anew1))
title(sprintf('Rank %d approximation',size(S1,1)))
```



svdsketch produces a rank 288 approximation, which results in some minor graininess in some of the boundary lines of the image.

Now, compress the image a second time using a tolerance of 1e-1. As the magnitude of the tolerance increases, the rank of the approximation produced by svdsketch generally decreases.

```
[U2,S2,V2] = svdsketch(double(A),1e-1);
Anew2 = uint8(U2*S2*V2');
imshow(Anew2)
title(sprintf('Rank %d approximation',size(S2,1)))
```

This time, `svdsketch` produces a rank 48 approximation. Most of the major aspects of the image are still visible, but the additional compression increases the blurriness.

Limit Subspace Size

`svdsketch` adaptively determines what rank to use for the matrix sketch based on the specified tolerance. However, you can use the `MaxSubspaceDimension` name-value pair to specify the maximum subspace size that should be used to form the matrix sketch. This option can produce matrices that do not satisfy the tolerance, since the subspace you specify might be too small. In these cases, `svdsketch` returns a matrix sketch with the maximum allowed subspace size.

Use `svdsketch` with a tolerance of `1e-1` and a maximum subspace size of 15. Specify a fourth output to return the relative approximation error.

```
[U3,S3,V3,apxErr] = svdsketch(double(A),1e-1,'MaxSubspaceDimension',15);
```

Compare the relative approximation error of the result with the specified tolerance. `apxErr` contains one element since `svdsketch` only needs one iteration to compute the answer.

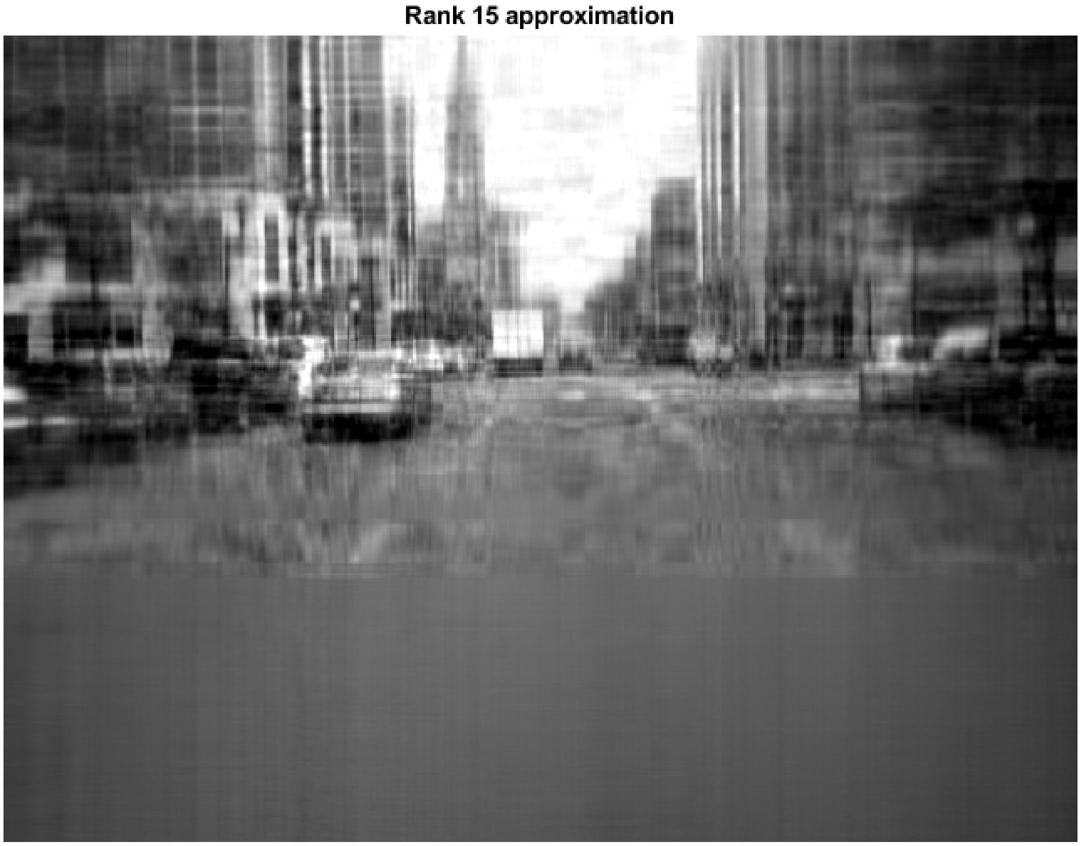
```
apxErr <= 1e-1
```

```
ans = logical
   0
```

The result indicates that the matrix sketch does not satisfy the specified tolerance.

View the heavily compressed rank 15 image.

```
Anew3 = uint8(U3*S3*V3');
imshow(Anew3)
title(sprintf('Rank %d approximation',size(S3,1)))
```

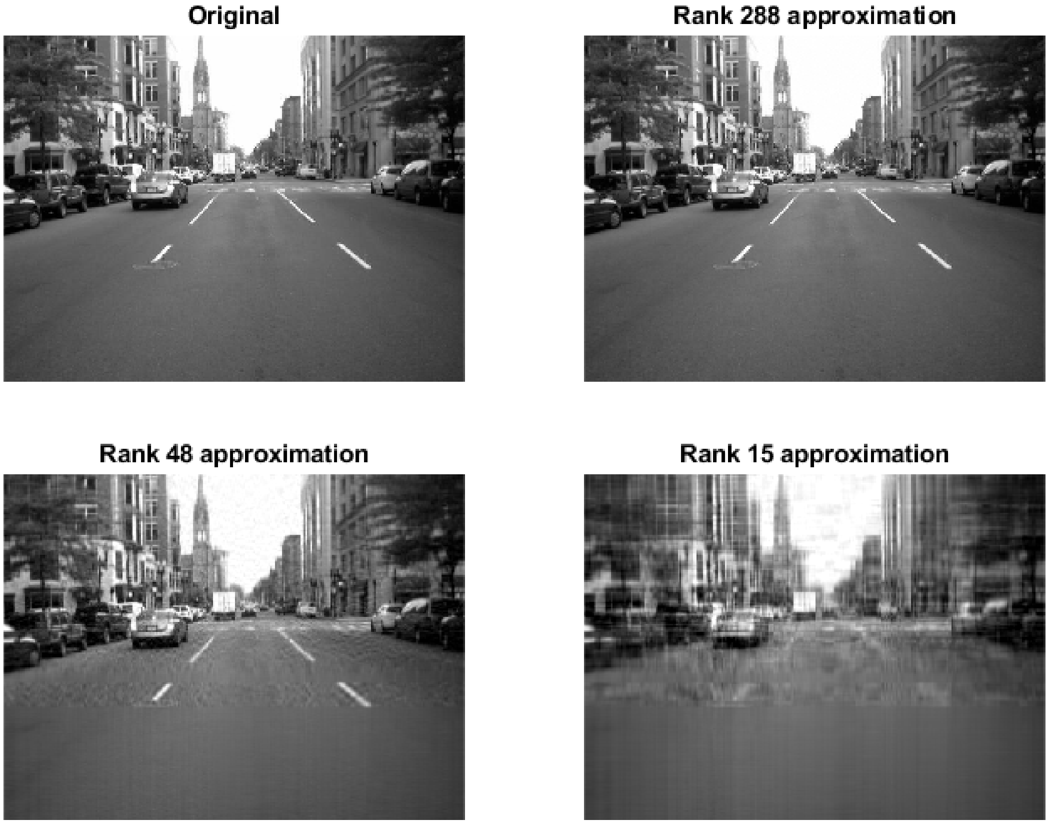


Compare Results

Finally, view all of the images side-by-side for comparison.

```
tiledlayout(2,2,'TileSpacing','Compact')
nexttile
imshow(A)
title('Original')
nexttile
imshow(Anew1)
title(sprintf('Rank %d approximation',size(S1,1)))
nexttile
```

```
imshow(Anew2)
title(sprintf('Rank %d approximation',size(S2,1)))
nexttile
imshow(Anew3)
title(sprintf('Rank %d approximation',size(S3,1)))
```



Original

Rank 288 approximation

Rank 48 approximation

Rank 15 approximation

See Also

svd | svds | svdsketch

More About

- “Singular Values” on page 2-33
- “Resample Image with Gridded Interpolation” on page 8-53

Random Numbers

Why Do Random Numbers Repeat After Startup?

All the random number functions, `rand`, `randn`, `randi`, and `randperm`, draw values from a shared random number generator. Every time you start MATLAB, the generator resets itself to the same state. Therefore, a command such as `rand(2,2)` returns the same result any time you execute it immediately following startup. Also, any script or function that calls the random number functions returns the same result whenever you restart.

If you want to avoid repeating the same random number arrays when MATLAB restarts, then execute the command,

```
rng('shuffle');
```

before calling `rand`, `randn`, `randi`, or `randperm`. This command ensures that you do not repeat a result from a previous MATLAB session.

If you want to repeat a result that you got at the start of a MATLAB session without restarting, you can reset the generator to the startup state at any time using

```
rng('default');
```

When you execute `rng('default')`, the ensuing random number commands return results that match the output of a new MATLAB session. For example,

```
rng('default');
A = rand(2,2)
```

```
A =

    0.8147    0.1270
    0.9058    0.9134
```

The values in `A` match the output of `rand(2,2)` whenever you restart MATLAB.

See Also

`rng`

Create Arrays of Random Numbers

In this section...

MATLAB uses algorithms to generate pseudorandom and pseudoindependent numbers. These numbers are not strictly random and independent in the mathematical sense, but they pass various statistical tests of randomness and independence, and their calculation can be repeated for testing or diagnostic purposes.

The `rand`, `randi`, `randn`, and `randperm` functions are the primary functions for creating arrays of random numbers. The `rng` function allows you to control the seed and algorithm that generates random numbers.

Random Number Functions

There are four fundamental random number functions: `rand`, `randi`, `randn`, and `randperm`. The `rand` function returns real numbers between 0 and 1 that are drawn from a uniform distribution. For example:

```
rng('default')
r1 = rand(1000,1);
```

`r1` is a 1000-by-1 column vector containing real floating-point numbers drawn from a uniform distribution. All the values in `r1` are in the open interval (0, 1). A histogram of these values is roughly flat, which indicates a fairly uniform sampling of numbers.

The `randi` function returns `double` integer values drawn from a discrete uniform distribution. For example,

```
r2 = randi(10,1000,1);
```

`r2` is a 1000-by-1 column vector containing integer values drawn from a discrete uniform distribution whose range is in the close interval [1, 10]. A histogram of these values is roughly flat, which indicates a fairly uniform sampling of integers between 1 and 10.

The `randn` function returns arrays of real floating-point numbers that are drawn from a standard normal distribution. For example:

```
r3 = randn(1000,1);
```

`r3` is a 1000-by-1 column vector containing numbers drawn from a standard normal distribution. A histogram of `r3` looks like a roughly normal distribution whose mean is 0 and standard deviation is 1.

You can use the `randperm` function to create a `double` array of random integer values that have no repeated values. For example,

```
r4 = randperm(15,5);
```

`r4` is a 1-by-5 array containing integers randomly selected from the range [1, 15]. Unlike `randi`, which can return an array containing repeated values, the array returned by `randperm` has no repeated values.

Successive calls to any of these functions return different results. This behavior is useful for creating several different arrays of random values.

Random Number Generators

MATLAB offers several generator algorithm options, which are summarized in the table.

Value	Generator Name	Generator Keyword
`'twister'`	Mersenne Twister (used by default stream at MATLAB startup)	mt19937ar
`'simdTwister'`	SIMD-oriented Fast Mersenne Twister	dsfmt19937
`'combRecursive'`	Combined multiple recursive	mrg32k3a
`'multFibonacci'`	Multiplicative Lagged Fibonacci	mlfg6331_64
`'philox'`	Philox 4x32 generator with 10 rounds	philox4x32_10
`'threefry'`	Threefry 4x64 generator with 20 rounds	threefry4x64_20
`'v4'`	Legacy MATLAB version 4.0 generator	mcg16807
`'v5uniform'`	Legacy MATLAB version 5.0 uniform generator	swb2712
`'v5normal'`	Legacy MATLAB version 5.0 normal generator	shr3cong

Use the `rng` function to set the seed and generator used by the `rand`, `randi`, `randn`, and `randperm` functions. For example, `rng(0,'twister')` reset the generator to its default state. To avoid repetition of random number arrays when MATLAB restarts, see “Why Do Random Numbers Repeat After Startup?” on page 3-2

For more information about controlling the random number generator's state to repeat calculations using the same random numbers, or to guarantee that different random numbers are used in repeated calculations, see “Controlling Random Number Generation” on page 3-36.

Random Number Data Types

`rand` and `randn` functions generate values in double precision by default.

```
rng('default')
A = rand(1,5);
class(A)
```

```
ans = 'double'
```

To specify the class as double explicitly:

```
rng('default')
B = rand(1,5,'double');
class(B)
```

```
ans = 'double'

isequal(A,B)

ans =
1
```

`rand` and `randn` can also generate values in single precision.

```
rng('default')
A = rand(1,5,'single');
class(A)

ans = 'single'
```

The values are the same as if you had cast the double precision values from the previous example. The random stream that the functions draw from advances the same way regardless of what class of values is returned.

```
A,B


A =
    0.8147    0.9058    0.1270    0.9134    0.6324


B =
    0.8147    0.9058    0.1270    0.9134    0.6324
```

`randi` supports both integer types and single or double precision.

```
A = randi([1 10],1,5,'double');
class(A)

ans = 'double'

B = randi([1 10],1,5,'uint8');
class(B)

ans = 'uint8'
```

See Also

`rand` | `randi` | `randn` | `randperm` | `rng`

Related Examples

- “Controlling Random Number Generation” on page 3-36
- “Generate Random Numbers That Are Repeatable” on page 3-11
- “Generate Random Numbers That Are Different” on page 3-14
- “Random Numbers Within a Specific Range” on page 3-6
- “Random Integers” on page 3-7
- “Random Numbers from Normal Distribution with Specific Mean and Variance” on page 3-8

Random Numbers Within a Specific Range

This example shows how to create an array of random floating-point numbers that are drawn from a uniform distribution in the open interval (50, 100).

By default, `rand` returns normalized values (between 0 and 1) that are drawn from a uniform distribution. To change the range of the distribution to a new range, (*a*, *b*), multiply each value by the width of the new range, (*b* – *a*) and then shift every value by *a*.

First, initialize the random number generator to make the results in this example repeatable.

```
rng(0,'twister');
```

Create a vector of 1000 random values. Use the `rand` function to draw the values from a uniform distribution in the open interval, (50,100).

```
a = 50;
b = 100;
r = (b-a).*rand(1000,1) + a;
```

Verify the values in `r` are within the specified range.

```
r_range = [min(r) max(r)]
```

```
r_range =

   50.0261   99.9746
```

The result is in the open interval, (50,100).

Note Some combinations of *a* and *b* make it theoretically possible for your results to include *a* or *b*. In practice, this is extremely unlikely to happen.

See Also

`rng`

Related Examples

- “Random Numbers from Normal Distribution with Specific Mean and Variance” on page 3-8
- “Random Numbers Within a Sphere” on page 3-9
- “Create Arrays of Random Numbers” on page 3-3

Random Integers

This example shows how to create an array of random integer values that are drawn from a discrete uniform distribution on the set of numbers –10, –9,...,9, 10.

The simplest `randi` syntax returns double-precision integer values between 1 and a specified value, `imax`. To specify a different range, use the `imin` and `imax` arguments together.

First, initialize the random number generator to make the results in this example repeatable.

```
rng(0,'twister');
```

Create a 1-by-1000 array of random integer values drawn from a discrete uniform distribution on the set of numbers -10, -9,...,9, 10. Use the syntax, `randi([imin imax],m,n)`.

```
r = randi([-10 10],1,1000);
```

Verify that the values in `r` are within the specified range.

```
[rmin,rmax] = bounds(r)
```

```
rmin = -10
```

```
rmax = 10
```

See Also

`randi` | `rng`

Related Examples

- “Create Arrays of Random Numbers” on page 3-3

Random Numbers from Normal Distribution with Specific Mean and Variance

This example shows how to create an array of random floating-point numbers that are drawn from a normal distribution having a mean of 500 and variance of 25.

The `randn` function returns a sample of random numbers from a normal distribution with mean 0 and variance 1. The general theory of random variables states that if x is a random variable whose mean is μ_x and variance is σ_x^2, then the random variable, y, defined by $y = ax + b$,where a and b are constants, has mean $\mu_y = a\mu_x + b$ and variance $\sigma_y^2 = a^2\sigma_x^2$. You can apply this concept to get a sample of normally distributed random numbers with mean 500 and variance 25.

First, initialize the random number generator to make the results in this example repeatable.

```
rng(0,'twister');
```

Create a vector of 1000 random values drawn from a normal distribution with a mean of 500 and a standard deviation of 5.

```
a = 5;
b = 500;
y = a.*randn(1000,1) + b;
```

Calculate the sample mean, standard deviation, and variance.

```
stats = [mean(y) std(y) var(y)]
```

```
stats = 1×3

  499.8368    4.9948   24.9483
```

The mean and variance are not 500 and 25 exactly because they are calculated from a sampling of the distribution.

See Also

`randn` | `rng`

Related Examples

- “Random Numbers Within a Specific Range” on page 3-6
- “Random Numbers Within a Sphere” on page 3-9
- “Create Arrays of Random Numbers” on page 3-3

Random Numbers Within a Sphere

This example shows how to create random points within the volume of a sphere, as described by Knuth [1]. The sphere in this example is centered at the origin and has a radius of 3.

One way to create points inside a sphere is to specify them in spherical coordinates. Then you can convert them to Cartesian coordinates to plot them.

First, initialize the random number generator to make the results in this example repeatable.

```
rng(0,'twister')
```

Calculate an elevation angle for each point in the sphere. These values are in the open interval, $(-\pi/2, \pi/2)$, but are not uniformly distributed.

```
rvals = 2*rand(1000,1)-1;
elevation = asin(rvals);
```

Create an azimuth angle for each point in the sphere. These values are uniformly distributed in the open interval, $(0, 2\pi)$.

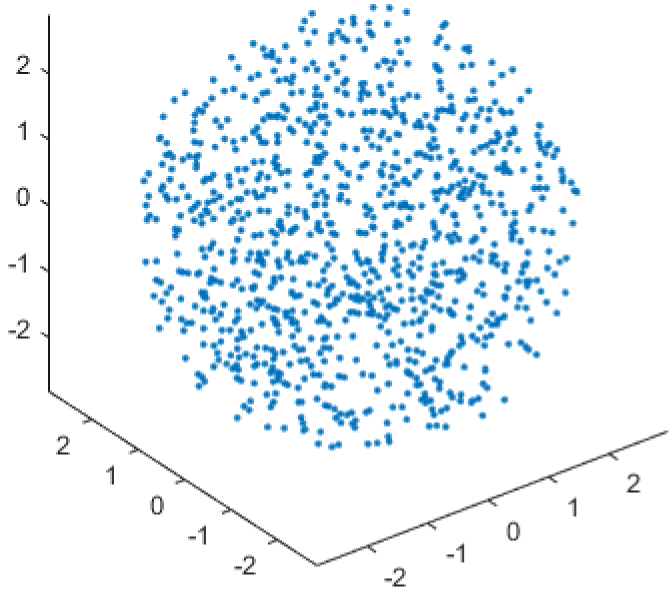
```
azimuth = 2*pi*rand(1000,1);
```

Create a radius value for each point in the sphere. These values are in the open interval, $(0, 3)$, but are not uniformly distributed.

```
radii = 3*(rand(1000,1).^(1/3));
```

Convert to Cartesian coordinates and plot the result.

```
[x,y,z] = sph2cart(azimuth,elevation,radii);
figure
plot3(x,y,z,'.')
axis equal
```



If you want to place random numbers *on the surface* of the sphere, then specify a constant radius value to be the last input argument to `sph2cart`. In this case, the value is `3`.

```
[x,y,z] = sph2cart(azimuth,elevation,3);
```

References

[1] Knuth, D. *The Art of Computer Programming*. Vol. 2, 3rd ed. Reading, MA: Addison-Wesley Longman, 1998, pp. 134–136.

See Also

`rand` | `rng` | `sph2cart`

Related Examples

- "Random Numbers Within a Specific Range" on page 3-6
- "Random Numbers from Normal Distribution with Specific Mean and Variance" on page 3-8
- "Create Arrays of Random Numbers" on page 3-3

Generate Random Numbers That Are Repeatable

Specify the Seed

This example shows how to repeat arrays of random numbers by specifying the seed first. Every time you initialize the generator using the same seed, you always get the same result.

First, initialize the random number generator to make the results in this example repeatable.

```
rng('default');
```

Now, initialize the generator using a seed of `1`.

```
rng(1);
```

Then, create an array of random numbers.

```
A = rand(3,3)
```

```
A =

    0.4170    0.3023    0.1863
    0.7203    0.1468    0.3456
    0.0001    0.0923    0.3968
```

Repeat the same command.

```
A = rand(3,3)
```

```
A =

    0.5388    0.2045    0.6705
    0.4192    0.8781    0.4173
    0.6852    0.0274    0.5587
```

The first call to `rand` changed the state of the generator, so the second result is different.

Now, reinitialize the generator using the same seed as before. Then reproduce the first matrix, `A`.

```
rng(1);
A = rand(3,3)
```

```
A =

    0.4170    0.3023    0.1863
    0.7203    0.1468    0.3456
    0.0001    0.0923    0.3968
```

In some situations, setting the seed alone will not guarantee the same results. This is because the generator that the random number functions draw from might be different than you expect when your code executes. For long-term repeatability, specify the seed and the generator type together.

For example, the following code sets the seed to `1` and the generator to Mersenne Twister.

```
rng(1,'twister');
```

Set the seed and generator type together when you want to:

- Ensure that the behavior of code you write today returns the same results when you run that code in a future MATLAB release.
- Ensure that the behavior of code you wrote in a previous MATLAB release returns the same results using the current release.
- Repeat random numbers in your code after running someone else's random number code.

See the `rng` reference page for a list of available generators.

Save and Restore the Generator Settings

This example shows how to create repeatable arrays of random numbers by saving and restoring the generator settings. The most common reason to save and restore generator settings is to reproduce the random numbers generated at a specific point in an algorithm or iteration. For example, you can use the generator settings as an aid in debugging. Unlike reseeding, which reinitializes the generator, this approach allows you to save and restore the generator settings at any point.

First, initialize the random number generator to make the results in this example repeatable.

```
rng(1,'twister');
```

Create an array of random integer values between 1 and 10.

```
A = randi(10,3,3)
```

```
A = 3×3

     5     4     2
     8     2     4
     1     1     4
```

The first call to `randi` changed the state of the generator. Save the generator settings after the first call to `randi` in a structure `s`.

```
s = rng;
```

Create another array of random integer values between 1 and 10.

```
A = randi(10,3,3)
```

```
A = 3×3

     6     3     7
     5     9     5
     7     1     6
```

Now, return the generator to the previous state stored in `s` and reproduce the second array `A`.

```
rng(s);
A = randi(10,3,3)
```

```
A = 3×3

     6     3     7
     5     9     5
     7     1     6
```

See Also

`rng`

Related Examples

- “Generate Random Numbers That Are Different” on page 3-14
- “Controlling Random Number Generation” on page 3-36

Generate Random Numbers That Are Different

This example shows how to avoid repeating the same random number arrays when MATLAB restarts. This technique is useful when you want to combine results from the same random number commands executed at different MATLAB sessions.

All the random number functions, `rand`, `randn`, `randi`, and `randperm`, draw values from a shared random number generator. Every time you start MATLAB, the generator resets itself to the same state. Therefore, a command such as `rand(2,2)` returns the same result any time you execute it immediately following startup. Also, any script or function that calls the random number functions returns the same result whenever you restart.

One way to get different random numbers is to initialize the generator using a different seed every time. Doing so ensures that you don't repeat results from a previous session.

Execute the `rng('shuffle')` command once in your MATLAB session before calling any of the random number functions.

```
rng('shuffle')
```

You can execute this command in a MATLAB Command Window, or you can add it to your startup file, which is a special script that MATLAB executes every time you restart.

Now, execute a random number command.

```
A = rand(2,2);
```

Each time you call `rng('shuffle')`, it reseeds the generator using a different seed based on the current time.

Note Frequent reseeding of the generator does not improve the statistical properties of the output and does not make the output more random in any real sense. Reseeding can be useful when you restart MATLAB or before you run a large calculation involving random numbers. However, reseeding the generator too frequently within a session is not a good idea because the statistical properties of your random numbers can be adversely affected.

Alternatively, specify different seeds explicitly in different MATLAB sessions. For example, generate random numbers in one MATLAB session.

```
rng(1);
A = rand(2,2);
```

Use different seeds to generate random numbers in another MATLAB session.

```
rng(2);
B = rand(2,2);
```

Arrays `A` and `B` are different because the generator is initialized with a different seed before each call to the `rand` function.

To generate multiple independent streams that are guaranteed to not overlap, and for which tests that demonstrate independence of the values between streams have been carried out, you can use

`RandStream.create`. For more information about generating multiple streams, see “Multiple Streams” on page 3-28.

See Also

`rng`

Related Examples

- “Generate Random Numbers That Are Repeatable” on page 3-11
- “Controlling Random Number Generation” on page 3-36
- “Startup Options in MATLAB Startup File”

Managing the Global Stream Using RandStream

The `rand`, `randn`, `randi`, and `randperm` functions draw random numbers from an underlying random number stream, called the *global stream*. The global stream is a `RandStream` object. A simple way to control the global stream is to use the `rng` function. For more comprehensive control, the `RandStream` class enables you to create a separate stream from the global stream, get a handle to the global stream, and control random number generation.

Use `rng` to set the random number generator to the default seed (`0`) and algorithm (Mersenne Twister). Save the generator settings.

```
rng('default')
s = rng
```

```
s = struct with fields:
     Type: 'twister'
     Seed: 0
    State: [625x1 uint32]
```

Create a 1-by-6 row vector of uniformly distributed random values between 0 and 1.

```
x = rand(1,6)
```

```
x = 1×6

    0.8147    0.9058    0.1270    0.9134    0.6324    0.0975
```

Use `RandStream.getGlobalStream` to return a handle to the global stream, that is, the current global stream that `rand` generates random numbers from. If you use `RandStream.getGlobalStream` to get a handle to the global stream, you can see the changes you made to the global stream using `rng`.

```
globalStream = RandStream.getGlobalStream
```

```
globalStream = 
mt19937ar random stream (current global stream)
             Seed: 0
  NormalTransform: Ziggurat
```

Change the generator seed and algorithm, and create a new random row vector. Show the current global stream that `rand` generates random numbers from.

```
rng(1,'philox')
xnew = rand(1,6)
```

```
xnew = 1×6

    0.5361    0.2319    0.7753    0.2390    0.0036    0.5262
```

```
globalStream = RandStream.getGlobalStream
```

```
globalStream = 
philox4x32_10 random stream (current global stream)
             Seed: 1
  NormalTransform: Inversion
```

Next, restore the original generator settings and create a random vector. The result matches the original row vector `x` created with the default generator.

```
rng(s)
xold = rand(1,6)
```

```
xold = 1×6

    0.8147    0.9058    0.1270    0.9134    0.6324    0.0975
```

By default, random number generation functions, such as `rand`, use the global random number stream. To specify a different stream, create another `RandStream` object. Pass it as the first input argument to `rand`. For example, create a 1-by-6 vector of random numbers using the SIMD-oriented Fast Mersenne Twister.

```
myStream = RandStream('dsfmt19937')
```

```
myStream = 
dsfmt19937 random stream
             Seed: 0
  NormalTransform: Ziggurat
```

```
r = rand(myStream,1,6)
```

```
r = 1×6

    0.0306    0.2131    0.2990    0.3811    0.8635    0.1334
```

When you call the `rand` function with `myStream` as the first input argument, it draws numbers from `myStream` and does not affect the results of the global stream.

If you want to set `myStream` as a global stream, you can use the `RandStream.setGlobalStream` object function.

```
RandStream.setGlobalStream(myStream)
globalStream = RandStream.getGlobalStream
```

```
globalStream = 
dsfmt19937 random stream (current global stream)
             Seed: 0
  NormalTransform: Ziggurat
```

In many cases, the `rng` function is all you need to control the global stream, but the `RandStream` class allows control over some advanced features, such as the choice of algorithm used for normal random values.

For example, create a `RandStream` object and specify the transformation algorithm to generate normally distributed pseudorandom values when using `randn`. Generate normally distributed pseudorandom values using the `Polar` transformation algorithm, instead of the default `Ziggurat` transformation algorithm.

```
myStream = RandStream('mt19937ar','NormalTransform','Polar')
```

```
myStream = 
mt19937ar random stream
             Seed: 0
  NormalTransform: Polar
```

Set myStream as the global stream. Create 6 random numbers with normal distribution from the global stream.

```
RandStream.setGlobalStream(myStream)
randn(1,6)
```

```
ans = 1×6

    0.2543   -0.7733   -1.7416    0.3686    0.5965   -0.0191
```

See Also

RandStream | rng

Related Examples

- “Multiple Streams” on page 3-28
- “Creating and Controlling a Random Number Stream” on page 3-19
- “Controlling Random Number Generation” on page 3-36
- “Create Arrays of Random Numbers” on page 3-3

Creating and Controlling a Random Number Stream

In this section...

The RandStream class allows you to create a random number stream. This is useful for several reasons:

- You can generate random values without affecting the state of the global stream.
- You can separate sources of randomness in a simulation.
- You can use a generator that is configured differently than the one MATLAB software uses at startup.

With a RandStream object, you can create your own stream, set the writable properties, and use the stream to generate random numbers. You can control the stream you create the same way you control the global stream. You can even replace the global stream with the stream you create.

To create a stream, use the RandStream function.

```
myStream = RandStream('mlfg6331_64');
rand(myStream,1,5)
```

```
ans =
    0.6986    0.7413    0.4239    0.6914    0.7255
```

The random stream myStream acts separately from the global stream. If you call the rand, randn, randi, and randperm functions with myStream as the first argument, they draw from the stream you created. If you call rand, randn, randi, and randperm without myStream, they draw from the global stream.

You can make myStream the global stream using the RandStream.setGlobalStream method.

```
RandStream.setGlobalStream(myStream)
RandStream.getGlobalStream
```

```
ans =

mlfg6331_64 random stream (current global stream)
             Seed: 0
  NormalTransform: Ziggurat
```

```
RandStream.getGlobalStream == myStream
```

```
ans =
     1
```

Substreams

You can use substreams to get different results that are statistically independent from a stream. Unlike seeds, where the locations along the sequence of random numbers are not exactly known, the spacing between substreams is known, so any chance of overlap can be eliminated. In short, substreams are a more-controlled way to do many of the same things that seeds have traditionally been used for. Substreams are also a more lightweight solution than parallel streams.

Substreams provide a quick and easy way to ensure that you get different results from the same code at different times. To use the `Substream` property of a `RandStream` object, create a stream using a generator that supports substreams. For a list of generator algorithms that support substreams and their properties, see the table in the next section. For example, generate several random numbers in a loop.

```
myStream = RandStream('mlfg6331_64');
RandStream.setGlobalStream(myStream)
for i = 1:5
    myStream.Substream = i;
    z = rand(1,i)
end

z =
    0.6986


z =
    0.9230    0.2489


z =
    0.0261    0.2530    0.0737


z =
    0.3220    0.7405    0.1983    0.1052


z =
    0.2067    0.2417    0.9777    0.5970    0.4187
```

In another loop, you can generate random values that are independent from the first set of 5 iterations.

```
for i = 6:10
    myStream.Substream = i;
    z = rand(1,11-i)
end

z =
    0.2650    0.8229    0.2479    0.0247    0.4581


z =
    0.3963    0.7445    0.7734    0.9113


z =
    0.2758    0.3662    0.7979


z =
    0.6814    0.5150


z =
    0.5247
```

Substreams are useful in serial computation. Substreams can recreate all or part of a simulation by returning to a particular checkpoint in stream. For example, you can return to the 6th substream in the loop. The result contains the same values as the 6th output above.

```
myStream.Substream = 6;
z = rand(1,5)

z =
    0.2650    0.8229    0.2479    0.0247    0.4581
```

Choosing a Random Number Generator

MATLAB offers several generator algorithm options. The table summarizes the key properties of the available generator algorithms and the keywords used to create them. To return a list of all the available generator algorithms, use the `RandStream.list` method.

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
`mt19937ar`	Mersenne twister (used by default stream at MATLAB startup)	No	2^{19937}-1
`dsfmt19937`	SIMD-oriented fast Mersenne twister	No	2^{19937}-1
`mcg16807`	Multiplicative congruential generator	No	2^{31}-2
`mlfg6331_64`	Multiplicative lagged Fibonacci generator	Yes	2^{124} (2^{51} streams of length 2^{72})
`mrg32k3a`	Combined multiple recursive generator	Yes	2^{191} (2^{63} streams of length 2^{127})
`philox4x32_10`	Philox 4x32 generator with 10 rounds	Yes	2^{193} (2^{64} streams of length 2^{129})
`threefry4x64_20`	Threefry 4x64 generator with 20 rounds	Yes	2^{514} (2^{256} streams of length 2^{258})
`shr3cong`	Shift-register generator summed with linear congruential generator	No	2^{64}
`swb2712`	Modified subtract with borrow generator	No	2^{1492}

The generators `mcg16807`, `shr3cong`, and `swb2712` provide for backwards compatibility with earlier versions of MATLAB. `mt19937ar` and `dsfmt19937` are designed primarily for sequential applications. The remaining generators provide explicit support for parallel random number generation.

Depending on the application, some generators might be faster or return values with more precision. All pseudorandom number generators are based on deterministic algorithms, and all generators pass a sufficiently specific statistical test for randomness. One way to check the results of a Monte Carlo simulation is to rerun the simulation with two or more different generator algorithms, and the choice of generators in MATLAB provides you with the means to do that. Although it is unlikely that your results will differ by more than the Monte Carlo sampling error when using different generators, there are examples in the literature where this kind of validation has turned up flaws in a particular generator algorithm. (See [13] for an example.)

Generator Algorithms

`mt19937ar`

The Mersenne Twister, as described in [11], has period $2^{19937} - 1$ and each U(0,1) value is created using two 32-bit integers. The possible values are multiples of 2^{-53} in the interval (0, 1). This generator does not support multiple streams or substreams. The `randn` algorithm used by default for `mt19937ar` streams is the ziggurat algorithm [7], but with the `mt19937ar` generator underneath.

Note This generator is identical to the one used by the `rand` function beginning in MATLAB Version 7, activated by `rand('twister',s)`.

`dsfmt19937`

The double precision SIMD-oriented Fast Mersenne Twister, as described in [12], is a faster implementation of the Mersenne Twister algorithm. The period is $2^{19937} - 1$ and the possible values are multiples of 2^{-52} in the interval (0, 1). The generator produces double precision values in [1, 2) natively, which are transformed to create U(0,1) values. This generator does not support multiple streams or substreams.

`mcg16807`

A 32-bit multiplicative congruential generator, as described in [14], with multiplier $a = 7^5$, modulo $m = 2^{31} - 1$. This generator has a period of $2^{31} - 2$ and does not support multiple streams or substreams. Each U(0,1) value is created using a single 32-bit integer from the generator; the possible values are all multiples of $(2^{31} - 1)^{-1}$ strictly within the interval (0, 1). For `mcg16807` streams, the default algorithm used by `randn` is the polar algorithm (described in [1]).

Note This generator is identical to the one used beginning in MATLAB Version 4 by both the `rand` and `randn` functions, activated using `rand('seed',s)` or `randn('seed',s)`.

`mlfg6331_64`

A 64-bit multiplicative lagged Fibonacci generator, as described in [10], with lags $l = 63$, $k = 31$. This generator is similar to the MLFG implemented in the SPRNG package. It has a period of approximately 2^{124}. It supports up to 2^{61} parallel streams, via parameterization, and 2^{51} substreams each of length 2^{72}. Each U(0,1) value is created using one 64-bit integer from the generator; the possible values are all multiples of 2^{-64} strictly within the interval (0, 1). The `randn` algorithm used by default for `mlfg6331_64` streams is the ziggurat algorithm [7], but with the `mlfg6331_64` generator underneath.

`mrg32k3a`

A 32-bit combined multiple recursive generator, as described in [2]. This generator is similar to the CMRG implemented in the RngStreams package in C. It has a period of 2^{191} and supports up to 2^{63} parallel streams through sequence splitting, each of length 2^{127}. It also supports 2^{51} substreams, each of length 2^{76}. Each U(0,1) value is created using two 32-bit integers from the generator; the possible values are multiples of 2^{-53} strictly within the interval (0, 1). The `randn` algorithm used by default for `mrg32k3a` streams is the ziggurat algorithm [7], but with the `mrg32k3a` generator underneath.

`philox4x32_10`

A 4x32 generator with 10 rounds as described in [15]. This generator uses a Feistel network and integer multiplication. The generator is specifically designed for high performance in highly parallel systems such as GPUs. It has a period of 2^{193} (2^{64} streams of length 2^{129}).

`threefry4x64_20`

A 4x64 generator with 20 rounds as described in [15]. This generator is a non-cryptographic adaptation of the Threefish block cipher from the Skein Hash Function. It has a period of 2^{514} (2^{256} streams of length 2^{258}).

`shr3cong`

Marsaglia's SHR3 shift-register generator summed with a linear congruential generator with multiplier $a = 69069$, addend $b = 1234567$, and modulus 2^{-32}. SHR3 is a 3-shift-register generator defined as $u = u(\mathbf{I} + \mathbf{L}^{13})(\mathbf{I} + \mathbf{R}^{17})(\mathbf{I} + \mathbf{L}^{5})$, where **I** is the identity operator, **L** is the left shift operator, and **R** is the right shift operator. The combined generator (the SHR3 part is described in [7]) has a period of approximately 2^{64}. This generator does not support multiple streams or substreams. Each U(0,1) value is created using one 32-bit integer from the generator; the possible values are all multiples of 2^{-32} strictly within the interval (0, 1). The `randn` algorithm used by default for `shr3cong` streams is the earlier form of the ziggurat algorithm [9], but with the `shr3cong` generator underneath. This generator is identical to the one used by the `randn` function beginning in MATLAB Version 5, activated using `randn('state',s)`.

Note The SHR3 generator used in [6] (1999) differs from the one used in [7] (2000). MATLAB uses the most recent version of the generator, presented in [7].

`swb2712`

A modified Subtract-with-Borrow generator, as described in [8]. This generator is similar to an additive lagged Fibonacci generator with lags 27 and 12, but it is modified to have a much longer period of approximately 2^{1492}. The generator works natively in double precision to create U(0,1) values, and all values in the open interval (0, 1) are possible. The `randn` algorithm used by default for `swb2712` streams is the ziggurat algorithm [7], but with the `swb2712` generator underneath.

Note This generator is identical to the one used by the `rand` function beginning in MATLAB Version 5, activated using `rand('state',s)`.

Transformation Algorithms

`Inversion`

Computes a normal random variate by applying the standard normal inverse cumulative distribution function to a uniform random variate. Exactly one uniform value is consumed per normal value.

`Polar`

The polar rejection algorithm, as described in [1]. Approximately 1.27 uniform values are consumed per normal value, on average.

`Ziggurat`

The ziggurat algorithm, as described in [7]. Approximately 2.02 uniform values are consumed per normal value, on average.

Configuring a Stream

A random number stream `s` has properties that control its behavior. To access or change a property, use the syntax `p = s.Property` and `s.Property = p`.

For example, you can configure the transformation algorithm to generate normally distributed pseudorandom values when using `randn`. Generate normally distributed pseudorandom values using the default `Ziggurat` transformation algorithm.

```
s1 = RandStream('mt19937ar');
s1.NormalTransform
```

```
ans = 'Ziggurat'
```

```
r1 = randn(s1,1,10);
```

Configure the stream to use the `Polar` transformation algorithm to generate normally distributed pseudorandom values.

```
s1.NormalTransform = 'Polar'
```

```
s1 = 
mt19937ar random stream
             Seed: 0
  NormalTransform: Polar
```

```
r2 = randn(s1,1,10);
```

When generating random numbers with uniform distribution using `rand`, you can also configure the stream to generate antithetic pseudorandom values, that is, the usual values subtracted from 1 for uniform values.

Create 6 random numbers with uniform distribution from the stream s.

```
s2 = RandStream('mt19937ar');
r1 = rand(s2,1,6)
```

```
r1 = 
    0.8147    0.9058    0.1270    0.9134    0.6324    0.0975
```

Restore the initial state of the stream. Create another 6 random numbers with the `Antithetic` property set to true. Check that these 6 random numbers are equal to the previously generated random numbers subtracted from 1.

```
reset(s2)
s2.Antithetic = true;
r2 = rand(s2,1,6)
```

```
r2 = 
    0.1853    0.0942    0.8730    0.0866    0.3676    0.9025
```

```
isequal(r1,1 - r2)
```

```
ans =
   1
```

Instead of setting the properties of a stream one-by-one, you can save and restore all properties of a stream s by using A = get(s) and set(s,A), respectively. For example, configure all properties of the stream s2 to be the same as the stream s1.

```
A = get(s1)
```

```
A = 
               Type: 'mt19937ar'
         NumStreams: 1
        StreamIndex: 1
          Substream: 1
               Seed: 0
              State: [625x1 uint32]
    NormalTransform: 'Polar'
         Antithetic: 0
      FullPrecision: 1
```

```
set(s2,A)
```

```
               Type: 'mt19937ar'
         NumStreams: 1
        StreamIndex: 1
          Substream: 1
               Seed: 0
              State: [625x1 uint32]
    NormalTransform: 'Polar'
         Antithetic: 0
      FullPrecision: 1
```

The get and set functions enable you to save and restore a stream's entire configuration so that results are exactly reproducible later on.

Restore State of Random Number Generator to Reproduce Output

The State property is the internal state of the random number generator. You can save the state of the global stream at a certain point in your simulation when generating random numbers to reproduce the results later on.

Use RandStream.getGlobalStream to return a handle to the global stream, that is, the current global stream that rand generates random numbers from. Save the state of the global stream.

```
globalStream = RandStream.getGlobalStream;
myState = globalStream.State;
```

Using myState, you can restore the state of globalStream and reproduce previous results.

```
A = rand(1,100);
globalStream.State = myState;
B = rand(1,100);
isequal(A,B)
```

```
ans = logical
   1
```

rand, randi, randn, and randperm access the global stream. Since all of these functions access the same underlying stream, a call to one affects the values produced by the others at subsequent calls.

```
globalStream.State = myState;
A = rand(1,100);
globalStream.State = myState;
C = randi(100);
B = rand(1,100);
isequal(A,B)

ans = logical
   0
```

You can also reset a stream to its initial settings using the reset function.

```
reset(globalStream)
A = rand(1,100);
reset(globalStream)
B = rand(1,100);
isequal(A,B)

ans = logical
   1
```

References

[1] Devroye, L. *Non-Uniform Random Variate Generation*, Springer-Verlag, 1986.

[2] L’Ecuyer, P. “Good Parameter Sets for Combined Multiple Recursive Random Number Generators”, *Operations Research*, 47(1): 159–164. 1999.

[3] L'Ecuyer, P. and S. Côté. “Implementing A Random Number Package with Splitting Facilities”, *ACM Transactions on Mathematical Software*, 17: 98–111. 1991.

[4] L'Ecuyer, P. and R. Simard. “TestU01: A C Library for Empirical Testing of Random Number Generators,” *ACM Transactions on Mathematical Software*, 33(4): Article 22. 2007.

[5] L'Ecuyer, P., R. Simard, E. J. Chen, and W. D. Kelton. “An Objected-Oriented Random-Number Package with Many Long Streams and Substreams.” *Operations Research*, 50(6):1073–1075. 2002.

[6] Marsaglia, G. “Random numbers for C: The END?” Usenet posting to sci.stat.math. 1999. Available online at https://groups.google.com/group/sci.crypt/browse_thread/thread/ca8682a4658a124d/.

[7] Marsaglia G., and W. W. Tsang. “The ziggurat method for generating random variables.” *Journal of Statistical Software*, 5:1–7. 2000. Available online at https://www.jstatsoft.org/v05/i08.

[8] Marsaglia, G., and A. Zaman. “A new class of random number generators.” *Annals of Applied Probability* 1(3):462–480. 1991.

[9] Marsaglia, G., and W. W. Tsang. “A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions.” *SIAM J. Sci. Stat. Comput*. 5(2):349–359. 1984.

[10] Mascagni, M., and A. Srinivasan. “Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators.” *Parallel Computing*, 30: 899–916. 2004.

[11] Matsumoto, M., and T. Nishimura.“Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator.” *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30. 1998.

[12] Matsumoto, M., and M. Saito.“A PRNG Specialized in Double Precision Floating Point Numbers Using an Affine Transition.” *Monte Carlo and Quasi-Monte Carlo Methods 2008*, 10.1007/978-3-642-04107-5_38. 2009.

[13] Moler, C.B. *Numerical Computing with MATLAB*. SIAM, 2004. Available online at `https://www.mathworks.com/moler`

[14] Park, S.K., and K.W. Miller. “Random Number Generators: Good Ones Are Hard to Find.” *Communications of the ACM*, 31(10):1192–1201. 1998.

[15] Salmon, J. K., M. A. Moraes, R. O. Dror, and D. E. Shaw. "Parallel Random Numbers: As Easy As 1, 2, 3." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC11)*. New York, NY: ACM, 2011.

See Also

`RandStream` | `rng`

Related Examples

- “Managing the Global Stream Using RandStream” on page 3-16
- “Multiple Streams” on page 3-28
- “Controlling Random Number Generation” on page 3-36
- “Create Arrays of Random Numbers” on page 3-3

Multiple Streams

Using Multiple Independent Streams

MATLAB® software includes generator algorithms that enable you to create multiple independent random number streams. For example, the four generator types that support multiple independent streams are the Combined Multiple Recursive ('mrg32k3a'), the Multiplicative Lagged Fibonacci ('mlfg6331_64'), the Philox 4x32 ('philox4x32_10'), and the Threefry 4x64 ('threefry4x64_20') generators. You can create multiple independent streams that are guaranteed to not overlap, and for which tests that demonstrate (pseudo)independence of the values between streams have been carried out. For more information about generator algorithms that support multiple streams, see the table of generator algorithms in “Creating and Controlling a Random Number Stream” on page 3-19.

The RandStream.create function enables you to create streams that have the same generator algorithm and seed value, but are statistically independent.

```
[s1,s2,s3] = RandStream.create('mlfg6331_64','NumStreams',3)
```

```
s1 =
mlfg6331_64 random stream
      StreamIndex: 1
       NumStreams: 3
             Seed: 0
  NormalTransform: Ziggurat

s2 =
mlfg6331_64 random stream
      StreamIndex: 2
       NumStreams: 3
             Seed: 0
  NormalTransform: Ziggurat

s3 =
mlfg6331_64 random stream
      StreamIndex: 3
       NumStreams: 3
             Seed: 0
  NormalTransform: Ziggurat
```

As evidence of independence, you can see that these streams are largely uncorrelated.

```
r1 = rand(s1,100000,1);
r2 = rand(s2,100000,1);
r3 = rand(s3,100000,1);
corrcoef([r1,r2,r3])
```

```
ans = 3×3

    1.0000    0.0007    0.0052
    0.0007    1.0000    0.0000
    0.0052    0.0000    1.0000
```

Depending on the application, creating only some of the streams in a set of independent streams can be useful if you need to simulate some events. Specify the StreamIndices parameter to create only some of the streams from a set of multiple streams. The StreamIndex property returns the index of each stream you create.

```
numLabs = 256;
labIndex = 4;
s4 = RandStream.create('mlfg6331_64','NumStreams',numLabs,'StreamIndices',labIndex)
```

```
s4 = 
mlfg6331_64 random stream
      StreamIndex: 4
       NumStreams: 256
             Seed: 0
  NormalTransform: Ziggurat
```

Multiple streams, since they are statistically independent, can be used to verify the precision of a simulation. For example, a set of independent streams can be used to repeat a Monte Carlo simulation several times in different MATLAB sessions or on different processors and determine the variance in the results. This makes multiple streams useful in large-scale parallel simulations.

Using Seeds to Get Different Results

For generator types that do not explicitly support independent streams, different seeds provide a method to create multiple streams. By using different seeds, you can create streams that return different values and act separately from one another. However, using a generator specifically designed for multiple independent streams is a better option, as the statistical properties across streams have been carefully verified.

Create two streams with different seeds by using the Mersenne twister generator.

```
s1 = RandStream('mt19937ar','Seed',1)
```

```
s1 = 
mt19937ar random stream
             Seed: 1
  NormalTransform: Ziggurat
```

```
s2 = RandStream('mt19937ar','Seed',2)
```

```
s2 = 
mt19937ar random stream
             Seed: 2
  NormalTransform: Ziggurat
```

Use the first stream in one MATLAB session to generate random numbers.

```
r1 = rand(s1,100000,1);
```

Use the second stream in another MATLAB session to generate random numbers.

```
r2 = rand(s2,100000,1);
```

With different seeds, streams typically return values that are uncorrelated.

```
corrcoef([r1,r2])
```

```
ans = 2×2

    1.0000    0.0030
    0.0030    1.0000
```

The two streams with different seeds may appear uncorrelated since the state space of the Mersenne Twister is so much larger (2^{19937} elements) than the number of possible seeds (2^{32}). The chances of overlap in different simulation runs are pretty remote unless you use a large number of different seeds. Using widely spaced seeds does not increase the level of randomness. In fact, taking this strategy to the extreme and reseeding a generator before each call can result in the sequence of values that are not statistically independent and identically distributed.

Seeding a stream is most useful if you use it as an initialization step, perhaps at MATLAB startup, or before running a simulation.

Using Substreams to Get Different Results

Another method to get different results from a stream is to use substreams. Unlike seeds, where the locations along the sequence of random numbers are not exactly known, the spacing between substreams is known, so any chance of overlap can be eliminated. Like independent parallel streams, research has been done to demonstrate statistical independence across substreams. In short, substreams are a more controlled way to do many of the same things that seeds have traditionally been used for, and a more lightweight solution than parallel streams.

Substreams provide a quick and easy way to ensure that you get different results from the same code at different times. For example, generate several random numbers in a loop.

```
defaultStream = RandStream('mlfg6331_64');
RandStream.setGlobalStream(defaultStream)
for i = 1:5
    defaultStream.Substream = i;
    z = rand(1,i)
end
```

```
z = 0.6986

z = 1×2

    0.9230    0.2489


z = 1×3

    0.0261    0.2530    0.0737


z = 1×4

    0.3220    0.7405    0.1983    0.1052


z = 1×5

    0.2067    0.2417    0.9777    0.5970    0.4187
```

In another loop, you can generate random values that are independent from the first set of 5 iterations.

```
for i = 6:10
    defaultStream.Substream = i;
    z = rand(1,11-i)
end
```

```
z = 1×5

    0.2650    0.8229    0.2479    0.0247    0.4581


z = 1×4

    0.3963    0.7445    0.7734    0.9113


z = 1×3

    0.2758    0.3662    0.7979


z = 1×2

    0.6814    0.5150


z = 0.5247
```

Each of these substreams can reproduce its loop iteration. For example, you can return to the 6th substream in the loop.

```
defaultStream.Substream = 6;
z = rand(1,5)
```

```
z = 1×5

    0.2650    0.8229    0.2479    0.0247    0.4581
```

See Also

RandStream | rng

Related Examples

- “Managing the Global Stream Using RandStream” on page 3-16
- “Controlling Random Number Generation” on page 3-36
- “Creating and Controlling a Random Number Stream” on page 3-19
- “Create Arrays of Random Numbers” on page 3-3

Replace Discouraged Syntaxes of rand and randn

In this section...

Description of the Discouraged Syntaxes

In earlier versions of MATLAB, you controlled the random number generator used by the `rand` and `randn` functions with the `'seed'`, `'state'` or `'twister'` inputs. For example:

```
rand('seed',sd)
randn('seed',sd)
rand('state',s)
randn('state',s)
rand('twister',5489)
```

These syntaxes referred to different types of generators, and they are no longer recommended for the following reasons:

- The terms `'seed'` and `'state'` are misleading names for the generators.
- All of the generators except `'twister'` are flawed.
- They unnecessarily use different generators for `rand` and `randn`.

To assess the impact of replacing discouraged syntaxes in your existing code, execute the following commands at the start of your MATLAB session:

```
warning('on','MATLAB:RandStream:ActivatingLegacyGenerators')
warning('on','MATLAB:RandStream:ReadingInactiveLegacyGeneratorState')
```

Description of Replacement Syntaxes

Use the `rng` function to control the shared generator used by `rand`, `randn`, `randi` and all other random number generation functions like `randperm`, `sprand`, and so on. To learn how to use the `rng` function when replacing discouraged syntaxes, take a few moments to understand their function. This should help you to see which new `rng` syntax best suits your needs.

The first input to `rand(Generator,s)` or `randn(Generator,s)` specified the type of the generator, as described here.

`Generator = 'seed'` referred to the MATLAB `v4` generator, not to the seed initialization value.

`Generator = 'state'` referred to the MATLAB `v5` generators, not to the internal state of the generator.

`Generator = 'twister'` referred to the Mersenne Twister generator, now the MATLAB startup generator.

The `v4` and `v5` generators are no longer recommended unless you are trying to exactly reproduce the random numbers generated in earlier versions of MATLAB. The simplest way to update your code is to use `rng`. The `rng` function replaces the names for the `rand` and `randn` generators as follows.

rand/randn Generator Name	rng Generator Name
`'seed'`	`'v4'`
`'state'`	`'v5uniform'` (for `rand`) or `'v5normal'` (for `randn`)
`'twister'`	`'twister'` (recommended)

Replacement Syntaxes for Initializing the Generator with an Integer Seed

The most common uses of the integer seed `sd` in the `rand(Generator,sd)` syntax were to:

- Reproduce exactly the same random numbers each time (e.g., by using a seed such as 0, 1, or 3141879)
- Try to ensure that MATLAB always gives different random numbers in separate runs (for example, by using a seed such as `sum(100*clock)`)

The following table shows replacements for syntaxes with an integer seed `sd`.

- The first column shows the discouraged syntax with `rand` and `randn`.
- The second column shows how to exactly reproduce the discouraged behavior with the new `rng` function. In most cases, this is done by specifying a legacy generator type such as the v4 or v5 generators, which is no longer recommended.
- The third column shows the recommended alternative, which does not specify the optional generator type input to `rng`. Therefore, if you *always* omit the `Generator` input, `rand`, `randn`, and `randi` just use the default Mersenne Twister generator that is used at MATLAB startup. In future releases when new generators supersede the Mersenne Twister, this code will use the new default.

Discouraged rand/randn Syntax	Not Recommended: Reproduce Discouraged Behavior Exactly By Specifying Generator Type	Recommended Alternative: Does Not Override Generator Type
`rand('twister',5489)`	`rng(5489,'twister')`	`rng('default')`
`rand('seed',sd)`	`rng(sd,'v4')`	`rng(sd)`
`randn('seed',sd)`		
`rand('state',sd)`	`rng(sd,'v5uniform')`	
`randn('state',sd)`	`rng(sd,'v5normal')`	
`rand('seed',sum(100*clock))`	`rng(sum(100*clock),'v4')`	`rng('shuffle')`

Replacement Syntaxes for Initializing the Generator with a State Vector

The most common use of the state vector (shown here as `st`) in the `rand(Generator,st)` syntax was to reproduce exactly the random numbers generated at a specific point in an algorithm or iteration. For example, you could use this vector as an aid in debugging.

The `rng` function changes the pattern of saving and restoring the state of the random number generator as shown in the next table. The example in the left column assumes that you are using the `v5` uniform generator. The example in the right column uses the new syntax, and works for any generator you use.

Discouraged Syntax Using rand/randn	New Syntax Using rng

`% Save v5 generator state.`
`st = rand('state');`
 	
`% Call rand.`
`x = rand;`
 	
`% Restore v5 generator state.`
`rand('state',st);`
 	
`% Call rand again and hope`
`% for the same results.`
`y = rand`
	`% Get generator settings.`
	`s = rng;`
	
	`% Call rand.`
	`x = rand;`
	
	`% Restore previous generator`
	`% settings.`
	`rng(s);`
	
	`% Call rand again and`
	`% get the same results.`
	`y = rand`

For a demonstration, see this instructional video.

If You Are Unable to Upgrade from Discouraged Syntax

If there is code that you are not able or not permitted to modify and you know that it uses the discouraged random number generator control syntaxes, it is important to remember that when you use that code MATLAB will switch into legacy mode. In legacy mode, `rand` and `randn` are controlled by separate generators, each with their own settings.

Calls to `rand` in legacy mode use one of the following:

- The `'v4'` generator, controlled by `rand('seed', ...)`
- The `'v5uniform'` generator, controlled by `rand('state', ...)`
- The `'twister'` generator, controlled by `rand('twister', ...)`

Calls to `randn` in legacy mode use one of the following:

- The `'v4'` generator, controlled by `randn('seed', ...)`
- The `'v5normal'` generator, controlled by `randn('state', ...)`

If code that you rely on puts MATLAB into legacy mode, use the following command to escape legacy mode and get back to the default startup generator:

```
rng default
```

Alternatively, to guard around code that puts MATLAB into legacy mode, use:

```
s = rng     % Save current settings of the generator.
  ...       % Call code using legacy random number generator syntaxes.
rng(s)      % Restore previous settings of the generator.
```

See Also

`rand` | `randn` | `rng`

Related Examples

- "Create Arrays of Random Numbers" on page 3-3

Controlling Random Number Generation

This example shows how to use the `rng` function, which provides control over random number generation.

(Pseudo)Random numbers in MATLAB come from the `rand`, `randi`, and `randn` functions. Many other functions call those three, but those are the fundamental building blocks. All three depend on a single shared random number generator that you can control using `rng`.

It's important to realize that "random" numbers in MATLAB are not unpredictable at all, but are generated by a deterministic algorithm. The algorithm is designed to be sufficiently complicated so that its output *appears* to be an independent random sequence to someone who does not know the algorithm, and can pass various statistical tests of randomness. The function that is introduced here provides ways to take advantage of the determinism to

- repeat calculations that involve random numbers, and get the same results, or
- guarantee that different random numbers are used in repeated calculations

and to take advantage of the apparent randomness to justify combining results from separate calculations.

"Starting Over"

If you look at the output from `rand`, `randi`, or `randn` in a new MATLAB session, you'll notice that they return the same sequences of numbers each time you restart MATLAB. It's often useful to be able to reset the random number generator to that startup state, without actually restarting MATLAB. For example, you might want to repeat a calculation that involves random numbers, and get the same result.

`rng` provides a very simple way to put the random number generator back to its default settings.

```
rng default
rand % returns the same value as at startup
```

```
ans = 0.8147
```

What are the "default" random number settings that MATLAB starts up with, or that `rng default` gives you? If you call `rng` with no inputs, you can see that it is the Mersenne Twister generator algorithm, seeded with 0.

```
rng
```

```
ans = struct with fields:
     Type: 'twister'
     Seed: 0
    State: [625x1 uint32]
```

You'll see in more detail below how to use the above output, including the `State` field, to control and change how MATLAB generates random numbers. For now, it serves as a way to see what generator `rand`, `randi`, and `randn` are currently using.

Non-Repeatability

Each time you call `rand`, `randi`, or `randn`, they draw a new value from their shared random number generator, and successive values can be treated as statistically independent. But as mentioned above, each time you restart MATLAB those functions are reset and return the same sequences of numbers. Obviously, calculations that use the *same* "random" numbers cannot be thought of as statistically independent. So when it's necessary to combine calculations done in two or more MATLAB sessions as if they *were* statistically independent, you cannot use the default generator settings.

One simple way to avoid repeating the same random numbers in a new MATLAB session is to choose a different seed for the random number generator. `rng` gives you an easy way to do that, by creating a seed based on the current time.

```
rng shuffle
rand
```

```
ans = 0.8696
```

Each time you use `'shuffle'`, it reseeds the generator with a different seed. You can call `rng` with no inputs to see what seed it actually used.

```
rng
```

```
ans = struct with fields:
     Type: 'twister'
     Seed: 117019470
    State: [625x1 uint32]
```

```
rng shuffle % creates a different seed each time
rng
```

```
ans = struct with fields:
     Type: 'twister'
     Seed: 117019475
    State: [625x1 uint32]
```

```
rand
```

```
ans = 0.4077
```

`'shuffle'` is a very easy way to reseed the random number generator. You might think that it's a good idea, or even necessary, to use it to get "true" randomness in MATLAB. For most purposes, though, *it is not necessary to use `'shuffle'` at all*. Choosing a seed based on the current time does not improve the statistical properties of the values you'll get from `rand`, `randi`, and `randn`, and does not make them "more random" in any real sense. While it is perfectly fine to reseed the generator each time you start up MATLAB, or before you run some kind of large calculation involving random numbers, it is actually not a good idea to reseed the generator too frequently within a session, because this can affect the statistical properties of your random numbers.

What `'shuffle'` does provide is a way to avoid repeating the same sequences of values. Sometimes that is critical, sometimes it's just "nice", but often it is not important at all. Bear in mind that if you use `'shuffle'`, you may want to save the seed that `rng` created so that you can repeat your calculations later on. You'll see how to do that below.

More Control over Repeatability and Non-Repeatability

So far, you've seen how to reset the random number generator to its default settings, and reseed it using a seed that is created using the current time. `rng` also provides a way to reseed it using a specific seed.

You can use the same seed several times, to repeat the same calculations. For example, if you run this code twice ...

```
rng(1) % the seed is any non-negative integer < 2^32
x = randn(1,5)
```

```
x = 1×5

   -0.6490    1.1812   -0.7585   -1.1096   -0.8456
```

```
rng(1)
x = randn(1,5)
```

```
x = 1×5

   -0.6490    1.1812   -0.7585   -1.1096   -0.8456
```

... you get exactly the same results. You might do this to recreate `x` after having cleared it, so that you can repeat what happens in subsequent calculations that depend on `x`, using those specific values.

On the other hand, you might want to choose *different* seeds to ensure that you don't repeat the same calculations. For example, if you run this code in one MATLAB session ...

```
rng(2)
x2 = sum(randn(50,1000),1); % 1000 trials of a random walk
```

and this code in another ...

```
rng(3)
x3 = sum(randn(50,1000),1);
```

... you could combine the two results and be confident that they are not simply the same results repeated twice.

```
x = [x2 x3];
```

As with `'shuffle'` there is a caveat when reseeding MATLAB's random number generator, because it affects all subsequent output from `rand`, `randi`, and `randn`. Unless you need repeatability or uniqueness, it is usually advisable to simply generate random values without reseeding the generator. If you do need to reseed the generator, that is usually best done at the command line, or in a spot in your code that is not easily overlooked.

Choosing a Generator Type

Not only can you reseed the random number generator as shown above, you can also choose the type of random number generator that you want to use. Different generator types produce different sequences of random numbers, and you might, for example, choose a specific type because of its statistical properties. Or you might need to recreate results from an older version of MATLAB that used a different default generator type.

One other common reason for choosing the generator type is that you are writing a validation test that generates "random" input data, and you need to guarantee that your test can always expect exactly the same predictable result. If you call `rng` with a seed before creating the input data, it reseeds the random number generator. But if the generator type has been changed for some reason, then the output from `rand`, `randi`, and `randn` will not be what you expect from that seed. Therefore, to be 100% certain of repeatability, you can also specify a generator type.

For example,

```
rng(0,'twister')
```

causes `rand`, `randi`, and `randn` to use the Mersenne Twister generator algorithm, after seeding it with 0.

Using `'combRecursive'`

```
rng(0,'combRecursive')
```

selects the Combined Multiple Recursive generator algorithm, which supports some parallel features that the Mersenne Twister does not.

This command

```
rng(0,'v4')
```

selects the generator algorithm that was the default in MATLAB 4.0.

And of course, this command returns the random number generator to its default settings.

```
rng default
```

However, because the default random number generator settings may change between MATLAB releases, using `'default'` does not guarantee predictable results over the long-term. `'default'` is a convenient way to reset the random number generator, but for even more predictability, specify a generator type and a seed.

On the other hand, when you are working interactively and need repeatability, it is simpler, and usually sufficient, to call `rng` with just a seed.

Saving and Restoring Random Number Generator Settings

Calling `rng` with no inputs returns a scalar structure with fields that contain two pieces of information described already: the generator type, and the integer with which the generator was last reseeded.

```
s = rng
```

```
s = struct with fields:
     Type: 'twister'
     Seed: 0
    State: [625x1 uint32]
```

The third field, `State`, contains a copy of the generator's current state vector. This state vector is the information that the generator maintains internally in order to generate the next value in its sequence of random numbers. Each time you call `rand`, `randi`, or `randn`, the generator that they share updates its internal state. Thus, the state vector in the settings structure returned by `rng` contains the information necessary to repeat the sequence, beginning from the point at which the state was captured.

While just being able to see this output is informative, `rng` also accepts a settings structure as an *input*, so that you can save the settings, including the state vector, and restore them later to repeat calculations. Because the settings contain the generator type, you'll know exactly what you're getting, and so "later" might mean anything from moments later in the same MATLAB session, to years (and multiple MATLAB releases) later. You can repeat results from any point in the random number sequence at which you saved the generator settings. For example

```
x1 = randn(10,10); % move ahead in the random number sequence
s = rng;           % save the settings at this point
x2 = randn(1,5)

x2 = 1×5

    0.8404   -0.8880    0.1001   -0.5445    0.3035


x3 = randn(5,5);   % move ahead in the random number sequence
rng(s);            % return the generator back to the saved state
x2 = randn(1,5)    % repeat the same numbers

x2 = 1×5

    0.8404   -0.8880    0.1001   -0.5445    0.3035
```

Notice that while reseeding provides only a coarse reinitialization, saving and restoring the generator state using the settings structure allows you to repeat *any* part of the random number sequence.

The most common way to use a settings structure is to restore the generator state. However, because the structure contains not only the state, but also the generator type and seed, it's also a convenient way to temporarily switch generator types. For example, if you need to create values using one of the legacy generators from MATLAB 5.0, you can save the current settings at the same time that you switch to use the old generator ...

```
previousSettings = rng(0,'v5uniform')

previousSettings = struct with fields:
     Type: 'twister'
     Seed: 0
    State: [625x1 uint32]
```

... and then restore the original settings later.

```
rng(previousSettings)
```

You should not modify the contents of any of the fields in a settings structure. In particular, you should not construct your own state vector, or even depend on the format of the generator state.

Writing Simpler, More Flexible, Code

`rng` allows you to either

- reseed the random number generator, or
- save and restore random number generator settings

without having to know what type it is. You can also return the random number generator to its default settings without having to know what those settings are. While there are situations when you might *want* to specify a generator type, `rng` affords you the simplicity of not *having* to specify it.

If you are able to avoid specifying a generator type, your code will automatically adapt to cases where a different generator needs to be used, and will automatically benefit from improved properties in a new default random number generator type.

`rng` and `RandStream`

`rng` provides a convenient way to control random number generation in MATLAB for the most common needs. However, more complicated situations involving multiple random number streams and parallel random number generation require a more complicated tool. The `RandStream` class is that tool, and it provides the most powerful way to control random number generation. The two tools are complementary, with `rng` providing a much simpler and concise syntax that is built on top of the flexibility of `RandStream`.

Sparse Matrices

Computational Advantages of Sparse Matrices

In this section...

Memory Management

Using sparse matrices to store data that contains a large number of zero-valued elements can both save a significant amount of memory and speed up the processing of that data. `sparse` is an attribute that you can assign to any two-dimensional MATLAB matrix that is composed of `double` or `logical` elements.

The `sparse` attribute allows MATLAB to:

- Store only the nonzero elements of the matrix, together with their indices.
- Reduce computation time by eliminating operations on zero elements.

For full matrices, MATLAB stores every matrix element internally. Zero-valued elements require the same amount of storage space as any other matrix element. For sparse matrices, however, MATLAB stores only the nonzero elements and their indices. For large matrices with a high percentage of zero-valued elements, this scheme significantly reduces the amount of memory required for data storage.

The `whos` command provides high-level information about matrix storage, including size and storage class. For example, this `whos` listing shows information about sparse and full versions of the same matrix.

```
M_full = magic(1100);          % Create 1100-by-1100 matrix.
M_full(M_full > 50) = 0;       % Set elements >50 to zero.
M_sparse = sparse(M_full);     % Create sparse matrix of same.

whos
```

```
  Name                Size                Bytes  Class     Attributes

  M_full         1100x1100              9680000  double
  M_sparse       1100x1100                 9608  double    sparse
```

Notice that the number of bytes used is fewer in the sparse case, because zero-valued elements are not stored.

Computational Efficiency

Sparse matrices also have significant advantages in terms of computational efficiency. Unlike operations with full matrices, operations with sparse matrices do not perform unnecessary low-level arithmetic, such as zero-adds (`x+0` is always `x`). The resulting efficiencies can lead to dramatic improvements in execution time for programs working with large amounts of sparse data.

See Also

More About

- “Sparse Matrix Operations” on page 4-14

Constructing Sparse Matrices

In this section...

Creating Sparse Matrices

MATLAB never creates sparse matrices automatically. Instead, you must determine if a matrix contains a large enough percentage of zeros to benefit from sparse techniques.

The *density* of a matrix is the number of nonzero elements divided by the total number of matrix elements. For matrix M, this would be

```
nnz(M) / prod(size(M));
```

or

```
nnz(M) / numel(M);
```

Matrices with very low density are often good candidates for use of the sparse format.

Converting Full to Sparse

You can convert a full matrix to sparse storage using the sparse function with a single argument.

For example:

```
A = [ 0   0   0   5
      0   2   0   0
      1   3   0   0
      0   0   4   0];
S = sparse(A)
```

```
 S =

   (3,1)        1
   (2,2)        2
   (3,2)        3
   (4,3)        4
   (1,4)        5
```

The printed output lists the nonzero elements of S, together with their row and column indices. The elements are sorted by columns, reflecting the internal data structure.

You can convert a sparse matrix to full storage using the full function, provided the matrix order is not too large. For example, A = full(S) reverses the example conversion.

Converting a full matrix to sparse storage is not the most frequent way of generating sparse matrices. If the order of a matrix is small enough that full storage is possible, then conversion to sparse storage rarely offers significant savings.

Creating Sparse Matrices Directly

You can create a sparse matrix from a list of nonzero elements using the `sparse` function with five arguments.

```
S = sparse(i,j,s,m,n)
```

`i` and `j` are vectors of row and column indices, respectively, for the nonzero elements of the matrix. `s` is a vector of nonzero values whose indices are specified by the corresponding `(i,j)` pairs. `m` is the row dimension of the resulting matrix, and `n` is the column dimension.

The matrix `S` of the previous example can be generated directly with

```
S = sparse([3 2 3 4 1],[1 2 2 3 4],[1 2 3 4 5],4,4)
```

```
S =

   (3,1)        1
   (2,2)        2
   (3,2)        3
   (4,3)        4
   (1,4)        5
```

The `sparse` command has a number of alternate forms. The example above uses a form that sets the maximum number of nonzero elements in the matrix to `length(s)`. If desired, you can append a sixth argument that specifies a larger maximum, allowing you to add nonzero elements later without reallocating the sparse matrix.

The matrix representation of the second difference operator is a good example of a sparse matrix. It is a tridiagonal matrix with -2s on the diagonal and 1s on the super- and subdiagonal. There are many ways to generate it—here's one possibility.

```
n = 5;
D = sparse(1:n,1:n,-2*ones(1,n),n,n);
E = sparse(2:n,1:n-1,ones(1,n-1),n,n);
S = E+D+E'
```

```
S =

   (1,1)       -2
   (2,1)        1
   (1,2)        1
   (2,2)       -2
   (3,2)        1
   (2,3)        1
   (3,3)       -2
   (4,3)        1
   (3,4)        1
   (4,4)       -2
   (5,4)        1
   (4,5)        1
   (5,5)       -2
```

Now `F = full(S)` displays the corresponding full matrix.

```
F = full(S)
```

```
F =
    -2     1     0     0     0
     1    -2     1     0     0
     0     1    -2     1     0
     0     0     1    -2     1
     0     0     0     1    -2
```

Creating Sparse Matrices from Their Diagonal Elements

Creating sparse matrices based on their diagonal elements is a common operation, so the function `spdiags` handles this task. Its syntax is

```
S = spdiags(B,d,m,n)
```

To create an output matrix `S` of size *m*-by-*n* with elements on `p` diagonals:

- `B` is a matrix of size `min(m,n)`-by-*p*. The columns of `B` are the values to populate the diagonals of `S`.
- `d` is a vector of length `p` whose integer elements specify which diagonals of `S` to populate.

That is, the elements in column `j` of `B` fill the diagonal specified by element `j` of `d`.

Note If a column of `B` is longer than the diagonal it's replacing, super-diagonals are taken from the lower part of the column of `B`, and sub-diagonals are taken from the upper part of the column of `B`.

As an example, consider the matrix `B` and the vector `d`.

```
B = [ 41    11     0
      52    22     0
      63    33    13
      74    44    24 ];

d = [-3
      0
      2];
```

Use these matrices to create a 7-by-4 sparse matrix `A`:

```
A = spdiags(B,d,7,4)

A =

   (1,1)       11
   (4,1)       41
   (2,2)       22
   (5,2)       52
   (1,3)       13
   (3,3)       33
   (6,3)       63
   (2,4)       24
   (4,4)       44
   (7,4)       74
```

In its full form, `A` looks like this:

```
full(A)
```

```
ans =

    11     0    13     0
     0    22     0    24
     0     0    33     0
    41     0     0    44
     0    52     0     0
     0     0    63     0
     0     0     0    74
```

spdiags can also extract diagonal elements from a sparse matrix, or replace matrix diagonal elements with new values. Type help spdiags for details.

Importing Sparse Matrices

You can import sparse matrices from computations outside the MATLAB environment. Use the spconvert function in conjunction with the load command to import text files containing lists of indices and nonzero elements. For example, consider a three-column text file T.dat whose first column is a list of row indices, second column is a list of column indices, and third column is a list of nonzero values. These statements load T.dat into MATLAB and convert it into a sparse matrix S:

```
load T.dat
S = spconvert(T)
```

The save and load commands can also process sparse matrices stored as binary data in MAT-files.

See Also

sparse | spconvert

More About

- “Sparse Matrix Operations” on page 4-14

Accessing Sparse Matrices

In this section...

Nonzero Elements

There are several commands that provide high-level information about the nonzero elements of a sparse matrix:

- nnz returns the number of nonzero elements in a sparse matrix.
- nonzeros returns a column vector containing all the nonzero elements of a sparse matrix.
- nzmax returns the amount of storage space allocated for the nonzero entries of a sparse matrix.

To try some of these, load the supplied sparse matrix west0479, one of the Harwell-Boeing collection.

```
load west0479
whos
```

```
  Name              Size              Bytes  Class     Attributes

  west0479      479x479               34032  double    sparse
```

This matrix models an eight-stage chemical distillation column.

Try these commands.

```
nnz(west0479)
```

```
ans =

        1887
```

```
format short e
west0479
```

```
west0479 =

  (25,1)      1.0000e+00
  (31,1)     -3.7648e-02
  (87,1)     -3.4424e-01
  (26,2)      1.0000e+00
  (31,2)     -2.4523e-02
  (88,2)     -3.7371e-01
  (27,3)      1.0000e+00
  (31,3)     -3.6613e-02
  (89,3)     -8.3694e-01
  (28,4)      1.3000e+02
     .
     .
     .
```

```
nonzeros(west0479)
```

```
ans =

   1.0000e+00
  -3.7648e-02
  -3.4424e-01
   1.0000e+00
  -2.4523e-02
  -3.7371e-01
   1.0000e+00
  -3.6613e-02
  -8.3694e-01
   1.3000e+02
    .
    .
    .
```

Note Use **Ctrl+C** to stop the `nonzeros` listing at any time.

Note that initially `nnz` has the same value as `nzmax` by default. That is, the number of nonzero elements is equivalent to the number of storage locations allocated for nonzeros. However, MATLAB does not dynamically release memory if you zero out additional array elements. Changing the value of some matrix elements to zero changes the value of `nnz`, but not that of `nzmax`.

However, you can add as many nonzero elements to the matrix as desired. You are not constrained by the original value of `nzmax`.

Indices and Values

For any matrix, full or sparse, the `find` function returns the indices and values of nonzero elements. Its syntax is

```
[i,j,s] = find(S);
```

`find` returns the row indices of nonzero values in vector `i`, the column indices in vector `j`, and the nonzero values themselves in the vector `s`. The example below uses `find` to locate the indices and values of the nonzeros in a sparse matrix. The `sparse` function uses the `find` output, together with the size of the matrix, to recreate the matrix.

```
S1 = west0479;
[i,j,s] = find(S1);
[m,n] = size(S1);
S2 = sparse(i,j,s,m,n);
```

Indexing in Sparse Matrix Operations

Because sparse matrices are stored in compressed sparse column format, there are different costs associated with indexing into a sparse matrix than there are with indexing into a full matrix. Such costs are negligible when you need to change only a few elements in a sparse matrix, so in those cases it’s normal to use regular array indexing to reassign values:

```
B = speye(4);
[i,j,s] = find(B);
[i,j,s]

ans =

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

B(3,1) = 42;
[i,j,s] = find(B);
[i,j,s]

ans =

     1     1     1
     3     1    42
     2     2     1
     3     3     1
     4     4     1
```

In order to store the new matrix with 42 at (3,1), MATLAB inserts an additional row into the nonzero values vector and subscript vectors, then shifts all matrix values after (3,1).

Using linear indexing to access or assign an element in a large sparse matrix will fail if the linear index exceeds 2^48-1, which is the current upper bound for the number of elements allowed in a matrix.

```
S = spalloc(2^30,2^30,2);
S(end) = 1

Maximum variable size allowed by the program is exceeded.
```

To access an element whose linear index is greater than intmax, use array indexing:

```
S(2^30,2^30) = 1

S =

         (1073741824,1073741824)             1
```

While the cost of indexing into a sparse matrix to change a single element is negligible, it is compounded in the context of a loop and can become quite slow for large matrices. For that reason, in cases where many sparse matrix elements need to be changed, it is best to vectorize the operation instead of using a loop. For example, consider a sparse identity matrix:

```
n = 10000;
A = 4*speye(n);
```

Changing the elements of A within a loop takes is slower than a similar vectorized operation:

```
tic
A(1:n-1,n) = -1;
A(n,1:n-1) = -1;
toc

Elapsed time is 0.003344 seconds.
```

```
tic
for k = 1:n-1
  C(k,n) = -1;
  C(n,k) = -1;
end
toc
```

```
Elapsed time is 0.448069 seconds.
```

Since MATLAB stores sparse matrices in compressed sparse column format, it needs to shift multiple entries in `A` during each pass through the loop.

Preallocating the memory for a sparse matrix and then filling it in an element-wise manner similarly causes a significant amount of overhead in indexing into the sparse array:

```
S1 = spalloc(1000,1000,100000);
tic;
for n = 1:100000
    i = ceil(1000*rand(1,1));
    j = ceil(1000*rand(1,1));
    S1(i,j) = rand(1,1);
end
toc
```

```
Elapsed time is 2.577527 seconds.
```

Constructing the vectors of indices and values eliminates the need to index into the sparse array, and thus is significantly faster:

```
i = ceil(1000*rand(100000,1));
j = ceil(1000*rand(100000,1));
v = zeros(size(i));
for n = 1:100000
    v(n) = rand(1,1);
end

tic;
S2 = sparse(i,j,v,1000,1000);
toc
```

```
Elapsed time is 0.017676 seconds.
```

For that reason, it’s best to construct sparse matrices all at once using a construction function, like the `sparse` or `spdiags` functions.

For example, suppose you wanted the sparse form of the coordinate matrix `C`:

$$C = \begin{pmatrix} 4 & 0 & 0 & 0 & -1 \\ 0 & 4 & 0 & 0 & -1 \\ 0 & 0 & 4 & 0 & -1 \\ 0 & 0 & 0 & 4 & -1 \\ 1 & 1 & 1 & 1 & 4 \end{pmatrix}$$

Construct the five-column matrix directly with the `sparse` function using the triplet pairs for the row subscripts, column subscripts, and values:

```
i = [1 5 2 5 3 5 4 5 1 2 3 4 5]';
j = [1 1 2 2 3 3 4 4 5 5 5 5 5]';
```

```
s = [4 1 4 1 4 1 4 1 -1 -1 -1 -1 4]';
C = sparse(i,j,s)
```

```
C =

   (1,1)        4
   (5,1)        1
   (2,2)        4
   (5,2)        1
   (3,3)        4
   (5,3)        1
   (4,4)        4
   (5,4)        1
   (1,5)       -1
   (2,5)       -1
   (3,5)       -1
   (4,5)       -1
   (5,5)        4
```

The ordering of the values in the output reflects the underlying storage by columns. For more information on how MATLAB stores sparse matrices, see John R. Gilbert, Cleve Moler, and Robert Schreiber's Sparse Matrices In MATLAB: Design and Implementation, (*SIAM Journal on Matrix Analysis and Applications*, 13:1, 333–356 (1992)).

Visualizing Sparse Matrices

It is often useful to use a graphical format to view the distribution of the nonzero elements within a sparse matrix. The MATLAB `spy` function produces a template view of the sparsity structure, where each point on the graph represents the location of a nonzero array element.

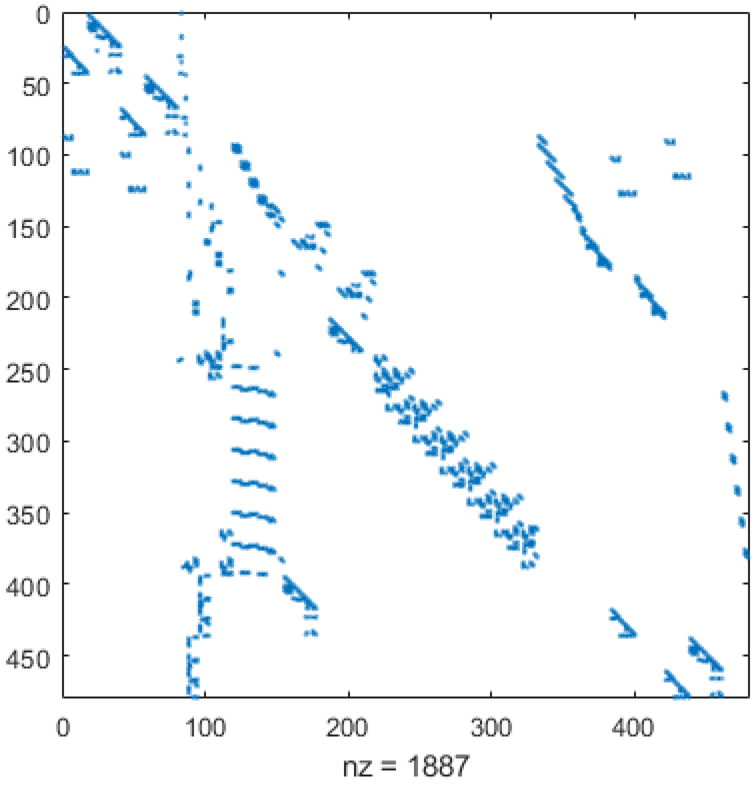
For example:

Load the supplied sparse matrix `west0479`, one of the Harwell-Boeing collection.

```
load west0479
```

View the sparsity structure.

```
spy(west0479)
```



See Also

`sparse`

More About

- “Computational Advantages of Sparse Matrices” on page 4-2
- “Constructing Sparse Matrices” on page 4-4
- “Sparse Matrix Operations” on page 4-14

Sparse Matrix Operations

Efficiency of Operations

Computational Complexity

The computational complexity of sparse operations is proportional to nnz, the number of nonzero elements in the matrix. Computational complexity also depends linearly on the row size m and column size n of the matrix, but is independent of the product m*n, the total number of zero and nonzero elements.

The complexity of fairly complicated operations, such as the solution of sparse linear equations, involves factors like ordering and fill-in, which are discussed in the previous section. In general, however, the computer time required for a sparse matrix operation is proportional to the number of arithmetic operations on nonzero quantities.

Algorithmic Details

Sparse matrices propagate through computations according to these rules:

- Functions that accept a matrix and return a scalar or constant-size vector always produce output in full storage format. For example, the size function always returns a full vector, whether its input is full or sparse.
- Functions that accept scalars or vectors and return matrices, such as zeros, ones, rand, and eye, always return full results. This is necessary to avoid introducing sparsity unexpectedly. The sparse analog of zeros(m,n) is simply sparse(m,n). The sparse analogs of rand and eye are sprand and speye, respectively. There is no sparse analog for the function ones.
- Unary functions that accept a matrix and return a matrix or vector preserve the storage class of the operand. If S is a sparse matrix, then chol(S) is also a sparse matrix, and diag(S) is a sparse vector. Columnwise functions such as max and sum also return sparse vectors, even though these vectors can be entirely nonzero. Important exceptions to this rule are the sparse and full functions.
- Binary operators yield sparse results if both operands are sparse, and full results if both are full. For mixed operands, the result is full unless the operation preserves sparsity. If S is sparse and F is full, then S+F, S*F, and F\S are full, while S.*F and S&F are sparse. In some cases, the result might be sparse even though the matrix has few zero elements.
- Matrix concatenation using either the cat function or square brackets produces sparse results for mixed operands.

Permutations and Reordering

A permutation of the rows and columns of a sparse matrix S can be represented in two ways:

- A permutation matrix P acts on the rows of S as P*S or on the columns as S*P'.
- A permutation vector p, which is a full vector containing a permutation of 1:n, acts on the rows of S as S(p,:), or on the columns as S(:,p).

For example:

```
p = [1 3 4 2 5]
I = eye(5,5);
```

```
P = I(p,:)
e = ones(4,1);
S = diag(11:11:55) + diag(e,1) + diag(e,-1)

p =

     1     3     4     2     5


P =

     1     0     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     1     0     0     0
     0     0     0     0     1


S =

    11     1     0     0     0
     1    22     1     0     0
     0     1    33     1     0
     0     0     1    44     1
     0     0     0     1    55
```

You can now try some permutations using the permutation vector `p` and the permutation matrix `P`. For example, the statements `S(p,:)` and `P*S` return the same matrix.

```
S(p,:)

ans =

    11     1     0     0     0
     0     1    33     1     0
     0     0     1    44     1
     1    22     1     0     0
     0     0     0     1    55
```

```
P*S

ans =

    11     1     0     0     0
     0     1    33     1     0
     0     0     1    44     1
     1    22     1     0     0
     0     0     0     1    55
```

Similarly, `S(:,p)` and `S*P'` both produce

```
S*P'

ans =

    11     0     0     1     0
     1     1     0    22     0
     0    33     1     1     0
     0     1    44     0     1
     0     0     1     0    55
```

If P is a sparse matrix, then both representations use storage proportional to n and you can apply either to S in time proportional to nnz(S). The vector representation is slightly more compact and efficient, so the various sparse matrix permutation routines all return full row vectors with the exception of the pivoting permutation in LU (triangular) factorization, which returns a matrix compatible with the full LU factorization.

To convert between the two permutation representations:

```
n = 5;
I = speye(n);
Pr = I(p,:);
Pc = I(:,p);
pc = (1:n)*Pc

pc =

     1     3     4     2     5

pr = (Pr*(1:n)')'

pr =

     1     3     4     2     5
```

The inverse of P is simply R = P'. You can compute the inverse of p with r(p) = 1:n.

```
r(p) = 1:5

r =

     1     4     2     3     5
```

Reordering for Sparsity

Reordering the columns of a matrix can often make its LU or QR factors sparser. Reordering the rows and columns can often make its Cholesky factors sparser. The simplest such reordering is to sort the columns by nonzero count. This is sometimes a good reordering for matrices with very irregular structures, especially if there is great variation in the nonzero counts of rows or columns.

The colperm computes a permutation that orders the columns of a matrix by the number of nonzeros in each column from smallest to largest.

Reordering to Reduce Bandwidth

The reverse Cuthill-McKee ordering is intended to reduce the profile or bandwidth of the matrix. It is not guaranteed to find the smallest possible bandwidth, but it usually does. The symrcm function actually operates on the nonzero structure of the symmetric matrix A + A', but the result is also useful for nonsymmetric matrices. This ordering is useful for matrices that come from one-dimensional problems or problems that are in some sense *long and thin*.

Approximate Minimum Degree Ordering

The degree of a node in a graph is the number of connections to that node. This is the same as the number of off-diagonal nonzero elements in the corresponding row of the adjacency matrix. The approximate minimum degree algorithm generates an ordering based on how these degrees are altered during Gaussian elimination or Cholesky factorization. It is a complicated and powerful algorithm that usually leads to sparser factors than most other orderings, including column count and reverse Cuthill-McKee. Because keeping track of the degree of each node is very time-consuming, the approximate minimum degree algorithm uses an approximation to the degree, rather than the exact degree.

These MATLAB functions implement the approximate minimum degree algorithm:

- `symamd` — Use with symmetric matrices.
- `colamd` — Use with nonsymmetric matrices and symmetric matrices of the form `A*A'` or `A'*A`.

See “Reordering and Factorization of Sparse Matrices” on page 4-18 for an example using `symamd`.

You can change various parameters associated with details of the algorithms using the `spparms` function.

For details on the algorithms used by `colamd` and `symamd`, see [5]. The approximate degree the algorithms use is based on [1].

Nested Dissection Ordering

Like the approximate minimum degree ordering, the nested dissection ordering algorithm implemented by the `dissect` function reorders the matrix rows and columns by considering the matrix to be the adjacency matrix of a graph. The algorithm reduces the problem down to a much smaller scale by collapsing together pairs of vertices in the graph. After reordering the small graph, the algorithm then applies projection and refinement steps to expand the graph back to the original size.

The nested dissection algorithm produces high quality reorderings and performs particularly well with finite element matrices compared to other reordering techniques. For more information about the nested dissection ordering algorithm, see [7].

Factoring Sparse Matrices

LU Factorization

If `S` is a sparse matrix, the following command returns three sparse matrices `L`, `U`, and `P` such that `P*S = L*U`.

```
[L,U,P] = lu(S);
```

`lu` obtains the factors by Gaussian elimination with partial pivoting. The permutation matrix `P` has only `n` nonzero elements. As with dense matrices, the statement `[L,U] = lu(S)` returns a permuted unit lower triangular matrix and an upper triangular matrix whose product is `S`. By itself, `lu(S)` returns `L` and `U` in a single matrix without the pivot information.

The three-output syntax `[L,U,P] = lu(S)` selects `P` via numerical partial pivoting, but does not pivot to improve sparsity in the `LU` factors. On the other hand, the four-output syntax `[L,U,P,Q] = lu(S)` selects `P` via threshold partial pivoting, and selects `P` and `Q` to improve sparsity in the `LU` factors.

You can control pivoting in sparse matrices using

```
lu(S,thresh)
```

where `thresh` is a pivot threshold in [0,1]. Pivoting occurs when the diagonal entry in a column has magnitude less than `thresh` times the magnitude of any sub-diagonal entry in that column. `thresh = 0` forces diagonal pivoting. `thresh = 1` is the default. (The default for `thresh` is `0.1` for the four-output syntax).

When you call `lu` with three or less outputs, MATLAB automatically allocates the memory necessary to hold the sparse `L` and `U` factors during the factorization. Except for the four-output syntax, MATLAB does not use any symbolic LU prefactorization to determine the memory requirements and set up the data structures in advance.

Reordering and Factorization of Sparse Matrices

This example shows the effects of reordering and factorization on sparse matrices.

If you obtain a good column permutation `p` that reduces fill-in, perhaps from `symrcm` or `colamd`, then computing `lu(S(:,p))` takes less time and storage than computing `lu(S)`.

Create a sparse matrix using the Bucky ball example.

```
B = bucky;
```

`B` has exactly three nonzero elements in each row and column.

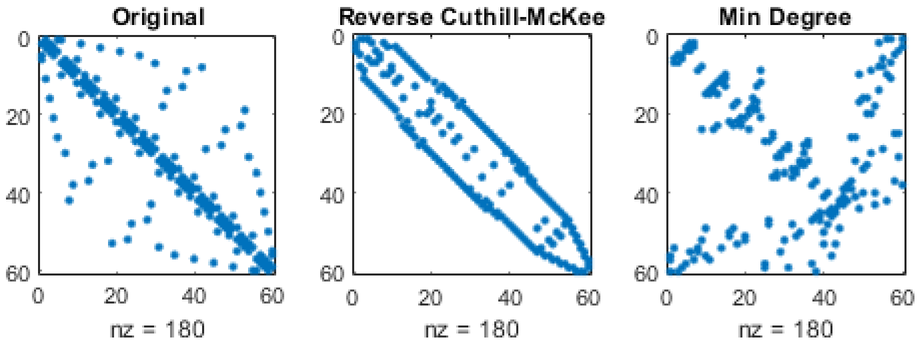
Create two permutations, `r` and `m` using `symrcm` and `symamd` respectively.

```
r = symrcm(B);
m = symamd(B);
```

The two permutations are the symmetric reverse Cuthill-McKee ordering and the symmetric approximate minimum degree ordering.

Create spy plots to show the three adjacency matrices of the Bucky Ball graph with these three different numberings. The local, pentagon-based structure of the original numbering is not evident in the others.

```
figure
subplot(1,3,1)
spy(B)
title('Original')

subplot(1,3,2)
spy(B(r,r))
title('Reverse Cuthill-McKee')

subplot(1,3,3)
spy(B(m,m))
title('Min Degree')
```



The reverse Cuthill-McKee ordering, `r`, reduces the bandwidth and concentrates all the nonzero elements near the diagonal. The approximate minimum degree ordering, `m`, produces a fractal-like structure with large blocks of zeros.

To see the fill-in generated in the LU factorization of the Bucky ball, use `speye`, the sparse identity matrix, to insert -3s on the diagonal of `B`.

```
B = B - 3*speye(size(B));
```

Since each row sum is now zero, this new `B` is actually singular, but it is still instructive to compute its LU factorization. When called with only one output argument, `lu` returns the two triangular factors, `L` and `U`, in a single sparse matrix. The number of nonzeros in that matrix is a measure of the time and storage required to solve linear systems involving `B`.

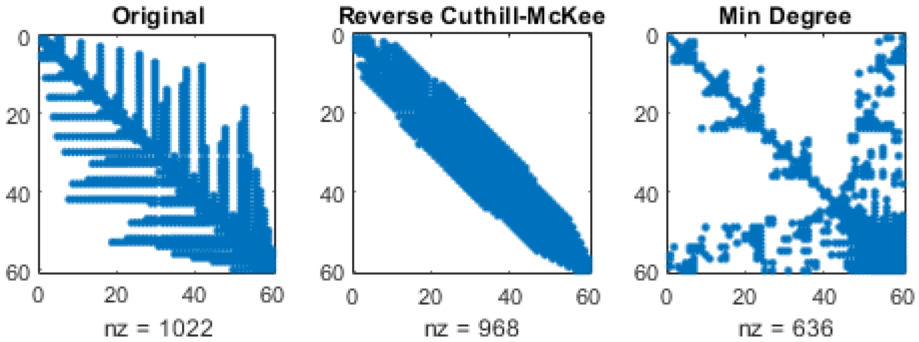
Here are the nonzero counts for the three permutations being considered.

- `lu(B)` (Original): 1022
- `lu(B(r,r))` (Reverse Cuthill-McKee): 968
- `lu(B(m,m))` (Approximate minimum degree): 636

Even though this is a small example, the results are typical. The original numbering scheme leads to the most fill-in. The fill-in for the reverse Cuthill-McKee ordering is concentrated within the band, but it is almost as extensive as the first two orderings. For the approximate minimum degree ordering, the relatively large blocks of zeros are preserved during the elimination and the amount of fill-in is significantly less than that generated by the other orderings.

The spy plots below reflect the characteristics of each reordering.

```
figure
subplot(1,3,1)
spy(lu(B))
title('Original')

subplot(1,3,2)
spy(lu(B(r,r)))
title('Reverse Cuthill-McKee')

subplot(1,3,3)
spy(lu(B(m,m)))
title('Min Degree')
```



Cholesky Factorization

If S is a symmetric (or Hermitian), positive definite, sparse matrix, the statement below returns a sparse, upper triangular matrix R so that R'*R = S.

```
R = chol(S)
```

chol does not automatically pivot for sparsity, but you can compute approximate minimum degree and profile limiting permutations for use with chol(S(p,p)).

Since the Cholesky algorithm does not use pivoting for sparsity and does not require pivoting for numerical stability, chol does a quick calculation of the amount of memory required and allocates all the memory at the start of the factorization. You can use `symbfact`, which uses the same algorithm as `chol`, to calculate how much memory is allocated.

QR Factorization

MATLAB computes the complete QR factorization of a sparse matrix `S` with

```
 [Q,R] = qr(S)
```

or

```
[Q,R,E] = qr(S)
```

but this is often impractical. The unitary matrix `Q` often fails to have a high proportion of zero elements. A more practical alternative, sometimes known as “the Q-less QR factorization,” is available.

With one sparse input argument and one output argument

```
R = qr(S)
```

returns just the upper triangular portion of the QR factorization. The matrix `R` provides a Cholesky factorization for the matrix associated with the normal equations:

```
R'*R = S'*S
```

However, the loss of numerical information inherent in the computation of `S'*S` is avoided.

With two input arguments having the same number of rows, and two output arguments, the statement

```
[C,R] = qr(S,B)
```

applies the orthogonal transformations to `B`, producing `C = Q'*B` without computing `Q`.

The Q-less QR factorization allows the solution of sparse least squares problems

$$\text{minimize}\|Ax - b\|_2$$

with two steps:

```
[c,R] = qr(A,b);
x = R\c
```

If `A` is sparse, but not square, MATLAB uses these steps for the linear equation solving backslash operator:

```
x = A\b
```

Or, you can do the factorization yourself and examine `R` for rank deficiency.

It is also possible to solve a sequence of least squares linear systems with different right-hand sides, `b`, that are not necessarily known when `R = qr(A)` is computed. The approach solves the “semi-normal equations `R'*R*x = A'*b` with

```
x = R\(R'\(A'*b))
```

and then employs one step of iterative refinement to reduce round off error:

```
r = b - A*x;
e = R\(R'\(A'*r));
x = x + e
```

Incomplete Factorizations

The ilu and ichol functions provide approximate, *incomplete* factorizations, which are useful as preconditioners for sparse iterative methods.

The ilu function produces three incomplete lower-upper (ILU) factorizations: the zero-fill ILU (ILU(0)), a Crout version of ILU (ILUC(tau)), and ILU with threshold dropping and pivoting (ILUTP(tau)). The ILU(0) never pivots and the resulting factors only have nonzeros in positions where the input matrix had nonzeros. Both ILUC(tau) and ILUTP(tau), however, do threshold-based dropping with the user-defined drop tolerance tau.

For example:

```
A = gallery('neumann', 1600) + speye(1600);
```

```
ans =

        7840
```

```
nnz(lu(A))
```

```
ans =

      126478
```

shows that A has 7840 nonzeros, and its complete LU factorization has 126478 nonzeros. On the other hand, the following code shows the different ILU outputs:

```
[L,U] = ilu(A);
nnz(L)+nnz(U)-size(A,1)
```

```
ans =

        7840
```

```
norm(A-(L*U).*spones(A),'fro')./norm(A,'fro')
```

```
ans =

   4.8874e-17
```

```
opts.type = 'ilutp';
opts.droptol = 1e-4;
[L,U,P] = ilu(A, opts);
nnz(L)+nnz(U)-size(A,1)
```

```
ans =

       31147
```

```
norm(P*A - L*U,'fro')./norm(A,'fro')
```

```
ans =

   9.9224e-05
```

```
opts.type = 'crout';
[L,U,P] = ilu(A, opts);
nnz(L)+nnz(U)-size(A,1)

ans =

       31083

norm(P*A-L*U,'fro')./norm(A,'fro')

ans =

   9.7344e-05
```

These calculations show that the zero-fill factors have 7840 nonzeros, the `ILUTP(1e-4)` factors have 31147 nonzeros, and the `ILUC(1e-4)` factors have 31083 nonzeros. Also, the relative error of the product of the zero-fill factors is essentially zero on the pattern of `A`. Finally, the relative error in the factorizations produced with threshold dropping is on the same order of the drop tolerance, although this is not guaranteed to occur. See the `ilu` reference page for more options and details.

The `ichol` function provides zero-fill incomplete Cholesky factorizations (`IC(0)`) as well as threshold-based dropping incomplete Cholesky factorizations (`ICT(tau)`) of symmetric, positive definite sparse matrices. These factorizations are the analogs of the incomplete LU factorizations above and have many of the same characteristics. For example:

```
A = delsq(numgrid('S',200));
nnz(A)

ans =

      195228

nnz(chol(A,'lower'))

ans =

     7762589
```

shows that `A` has 195228 nonzeros, and its complete Cholesky factorization without reordering has 7762589 nonzeros. By contrast:

```
L = ichol(A);
nnz(L)

ans =

      117216

norm(A-(L*L').*spones(A),'fro')./norm(A,'fro')

ans =

   3.5805e-17

opts.type = 'ict';
opts.droptol = 1e-4;
L = ichol(A,opts);
nnz(L)
```

```
ans =

     1166754

norm(A-L*L','fro')./norm(A,'fro')

ans =

   2.3997e-04
```

IC(0) has nonzeros only in the pattern of the lower triangle of A, and on the pattern of A, the product of the factors matches. Also, the ICT(1e-4) factors are considerably sparser than the complete Cholesky factor, and the relative error between A and L*L' is on the same order of the drop tolerance. It is important to note that unlike the factors provided by chol, the default factors provided by ichol are lower triangular. See the ichol reference page for more information.

Eigenvalues and Singular Values

Two functions are available that compute a few specified eigenvalues or singular values. svds is based on eigs.

Functions to Compute a Few Eigenvalues or Singular Values

Function	Description
eigs	Few eigenvalues
svds	Few singular values

These functions are most frequently used with sparse matrices, but they can be used with full matrices or even with linear operators defined in MATLAB code.

The statement

```
[V,lambda] = eigs(A,k,sigma)
```

finds the k eigenvalues and corresponding eigenvectors of the matrix A that are nearest the “shift” sigma. If sigma is omitted, the eigenvalues largest in magnitude are found. If sigma is zero, the eigenvalues smallest in magnitude are found. A second matrix, B, can be included for the generalized eigenvalue problem: $A\upsilon = \lambda B\upsilon$.

The statement

```
[U,S,V] = svds(A,k)
```

finds the k largest singular values of A and

```
[U,S,V] = svds(A,k,'smallest')
```

finds the k smallest singular values.

The numerical techniques used in eigs and svds are described in [6].

Smallest Eigenvalue of Sparse Matrix

This example shows how to find the smallest eigenvalue and eigenvector of a sparse matrix.

Set up the five-point Laplacian difference operator on a 65-by-65 grid in an *L*-shaped, two-dimensional domain.

```
L = numgrid('L',65);
A = delsq(L);
```

Determine the order and number of nonzero elements.

```
size(A)
```

```
ans = 1×2

        2945        2945
```

```
nnz(A)
```

```
ans = 14473
```

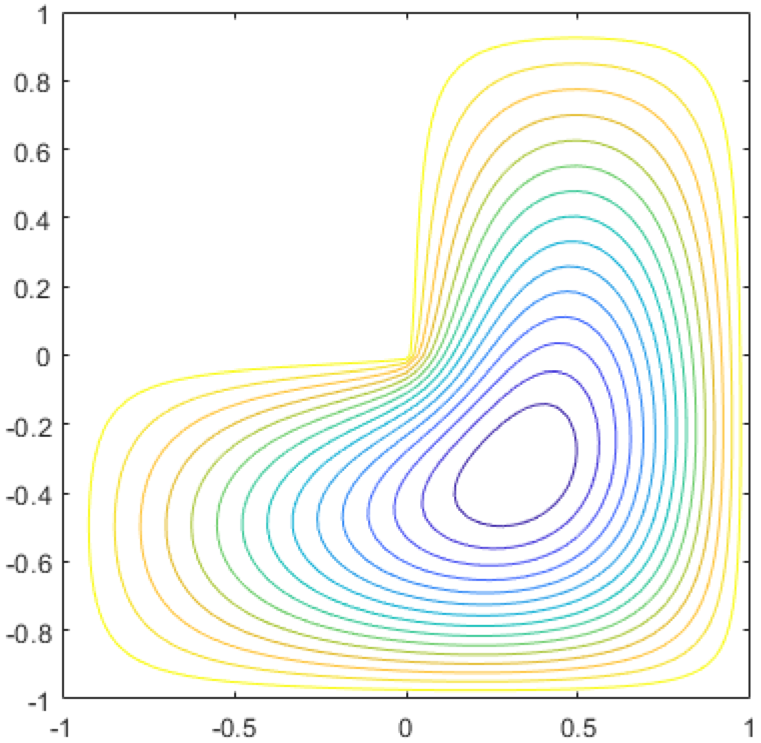
A is a matrix of order 2945 with 14,473 nonzero elements.

Compute the smallest eigenvalue and eigenvector.

```
[v,d] = eigs(A,1,'smallestabs');
```

Distribute the components of the eigenvector over the appropriate grid points and produce a contour plot of the result.

```
L(L>0) = full(v(L(L>0)));
x = -1:1/32:1;
contour(x,x,L,15)
axis square
```



References

[1] Amestoy, P. R., T. A. Davis, and I. S. Duff, “An Approximate Minimum Degree Ordering Algorithm,” *SIAM Journal on Matrix Analysis and Applications*, Vol. 17, No. 4, Oct. 1996, pp. 886-905.

[2] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[3] Davis, T.A., Gilbert, J. R., Larimore, S.I., Ng, E., Peyton, B., “A Column Approximate Minimum Degree Ordering Algorithm,” *Proc. SIAM Conference on Applied Linear Algebra*, Oct. 1997, p. 29.

[4] Gilbert, John R., Cleve Moler, and Robert Schreiber, “Sparse Matrices in MATLAB: Design and Implementation,” *SIAM J. Matrix Anal. Appl.*, Vol. 13, No. 1, January 1992, pp. 333-356.

[5] Larimore, S. I., *An Approximate Minimum Degree Column Ordering Algorithm*, MS Thesis, Dept. of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1998.

[6] Saad, Yousef, *Iterative Methods for Sparse Linear Equations*. PWS Publishing Company, 1996.

[7] Karypis, George and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs." *SIAM Journal on Scientific Computing*. Vol. 20, Number 1, 1999, pp. 359–392.

See Also

More About

- “Computational Advantages of Sparse Matrices” on page 4-2
- “Constructing Sparse Matrices” on page 4-4
- “Accessing Sparse Matrices” on page 4-8

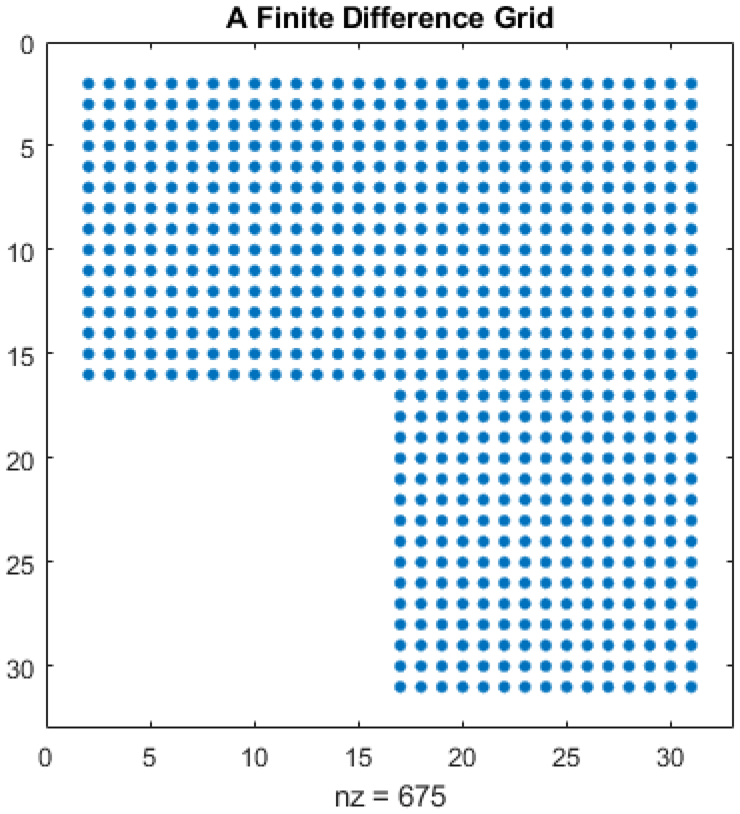
Finite Difference Laplacian

This example shows how to compute and represent the finite difference Laplacian on an L-shaped domain.

Domain

The `numgrid` function numbers points within an L-shaped domain. The `spy` function is a useful tool for visualizing the pattern of nonzero elements in a matrix. Use these two functions to generate and display an L-shaped domain.

```
n = 32;
R = 'L';
G = numgrid(R,n);
spy(G)
title('A Finite Difference Grid')
```



Show a smaller version of the matrix as a sample.

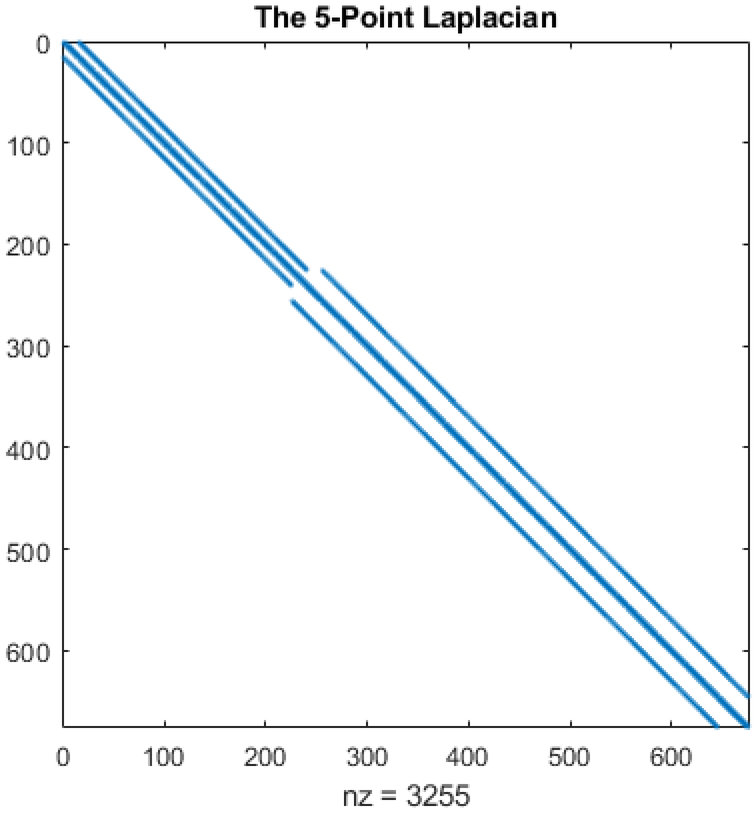
```
g = numgrid(R,10)
```

```
g = 10×10

     0     0     0     0     0     0     0     0     0     0
     0     1     5     9    13    17    25    33    41     0
     0     2     6    10    14    18    26    34    42     0
     0     3     7    11    15    19    27    35    43     0
     0     4     8    12    16    20    28    36    44     0
     0     0     0     0     0    21    29    37    45     0
     0     0     0     0     0    22    30    38    46     0
     0     0     0     0     0    23    31    39    47     0
     0     0     0     0     0    24    32    40    48     0
     0     0     0     0     0     0     0     0     0     0
```

Discrete Laplacian

Use `delsq` to generate the discrete Laplacian. Use the `spy` function again to get a graphical feel of the matrix elements.

```
D = delsq(G);
spy(D)
title('The 5-Point Laplacian')
```



Determine the number of interior points.

```
N = sum(G(:)>0)
```

```
N = 675
```

Dirichlet Boundary Value Problem

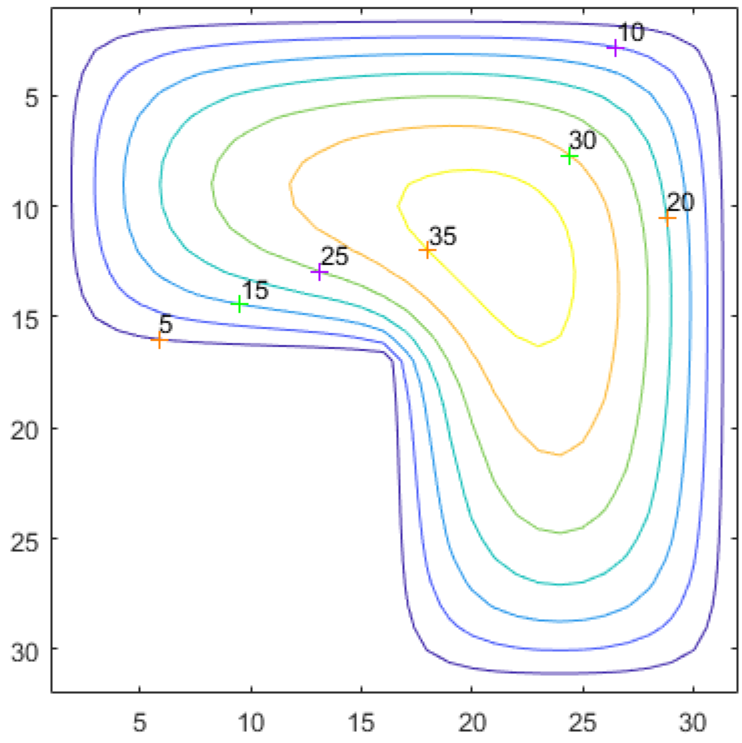
Solve the Dirichlet boundary value problem for the sparse linear system. The problem setup is:

`delsq(u) = 1` in the interior, `u = 0` on the boundary.

```
rhs = ones(N,1);
if (R == 'N') % For nested dissection, turn off minimum degree ordering.
   spparms('autommd',0)
   u = D\rhs;
   spparms('autommd',1)
else
   u = D\rhs; % This is used for R=='L' as in this example
end
```

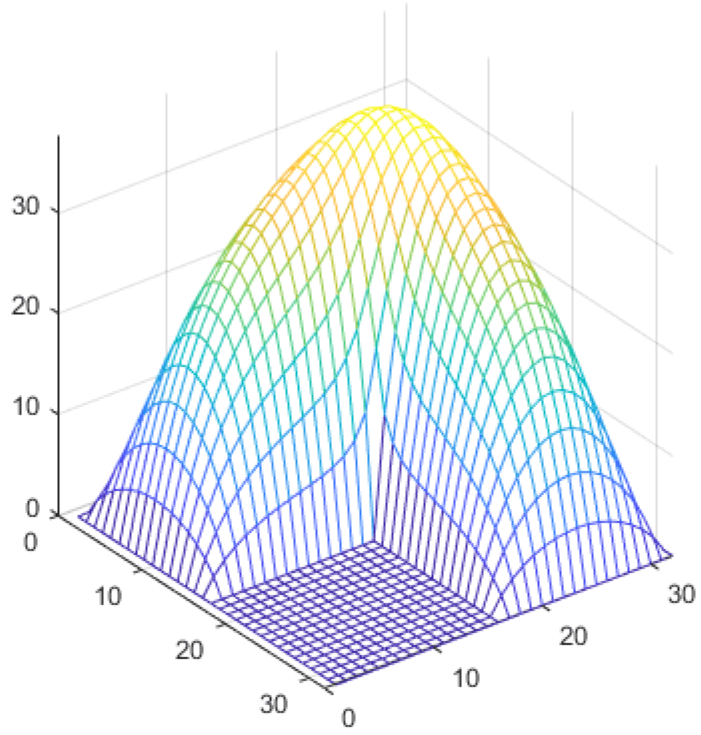
Map the solution onto the L-shaped grid and plot it as a contour map.

```
U = G;
U(G>0) = full(u(G(G>0)));
clabel(contour(U));
prism
axis square ij
```



Now show the solution as a mesh plot.

```
mesh(U)
axis([0 n 0 n 0 max(max(U))])
axis square ij
```



See Also

spy

Graphical Representation of Sparse Matrices

This example shows the finite element mesh for a NASA airfoil, including two trailing flaps. More information about the history of airfoils is available at NACA Airfoils (nasa.gov).

The data is stored in the file `airfoil.mat`. The data consists of 4253 pairs of (x,y) coordinates of the mesh points. It also contains an array of 12,289 pairs of indices, (i,j), specifying connections between the mesh points.

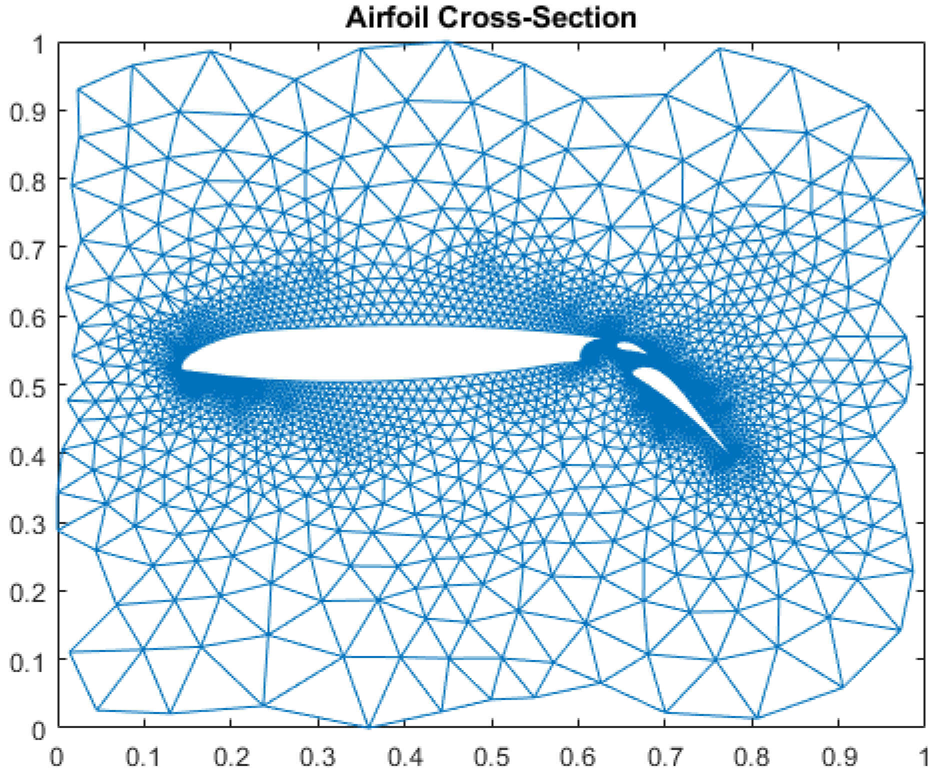
Load the data file into the workspace.

```
load airfoil
```

View Finite Element Mesh

First, scale `x` and `y` by 2^{-32} to bring them into the range $[0, 1]$. Then form a sparse adjacency matrix from the (i,j) connections and make it positive definite. Finally, plot the adjacency matrix using (x,y) as the coordinates for the vertices (mesh points).

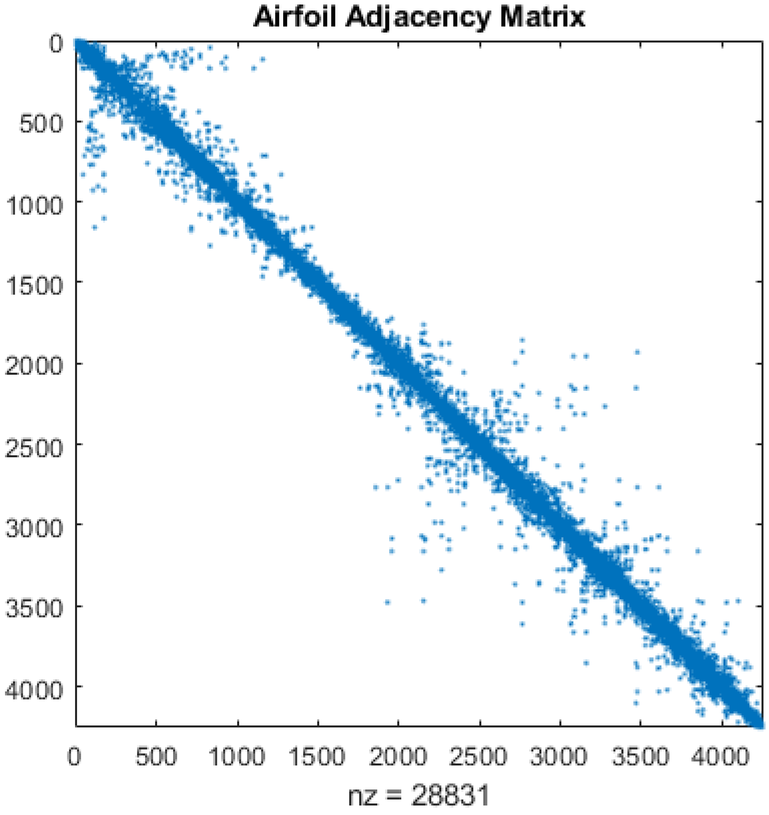
```
% Scaling x and y
x = pow2(x,-32);
y = pow2(y,-32);

% Forming the sparse adjacency matrix and making it positive definite
n = max(max(i),max(j));
A = sparse(i,j,-1,n,n);
A = A + A';
d = abs(sum(A)) + 1;
A = A + diag(sparse(d));

% Plotting the finite element mesh
gplot(A,[x y])
title('Airfoil Cross-Section')
```



Visualize Sparsity Pattern

You can use `spy` to visualize the nonzero elements in a matrix, so it is a particularly useful function to see the sparsity pattern in sparse matrices. `spy(A)` plots the sparsity pattern of the matrix `A`.

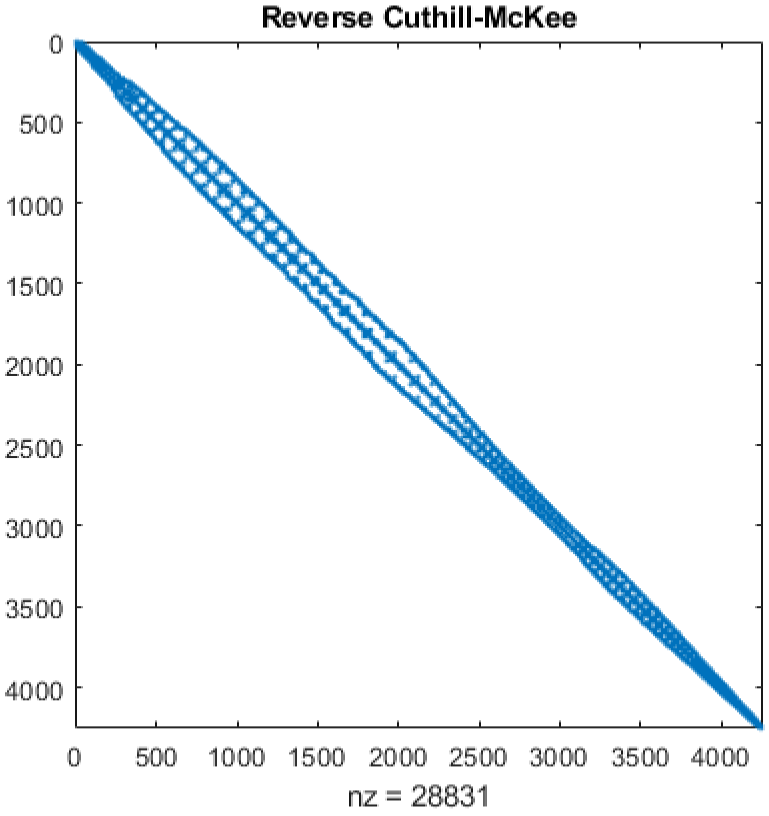
```
spy(A)
title('Airfoil Adjacency Matrix')
```



Symmetric Reordering - Reverse Cuthill-McKee

`symrcm` uses the Reverse Cuthill-McKee technique for reordering the adjacency matrix. `r = symrcm(A)` returns a permutation vector `r` such that `A(r,r)` tends to have its diagonal elements closer to the diagonal than `A`. This form is a good preordering for LU or Cholesky factorization of matrices that come from "long, skinny" problems. It works for both symmetric and nonsymmetric matrices.

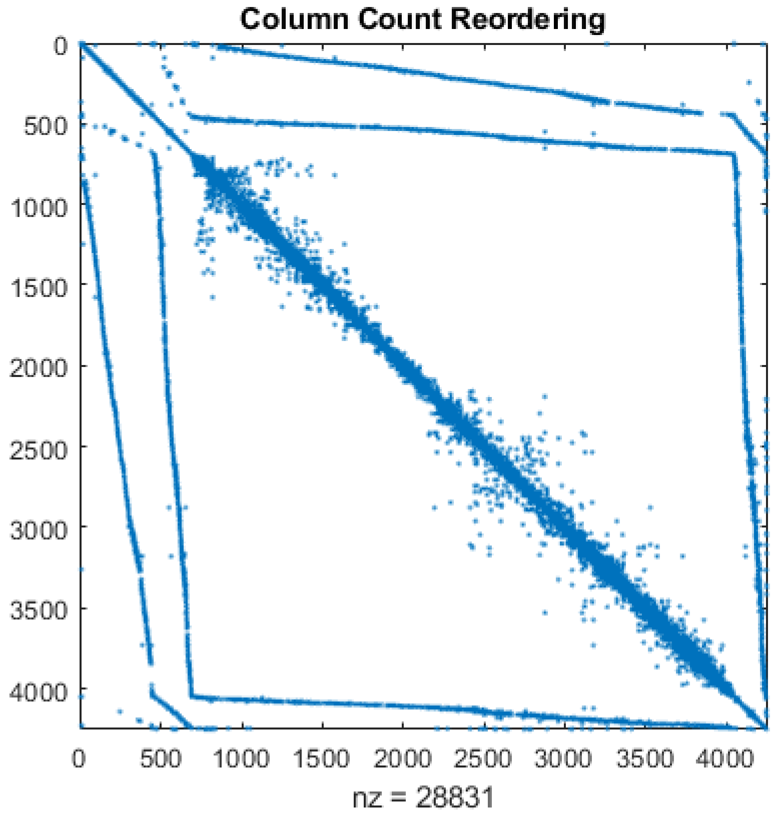
```
r = symrcm(A);
spy(A(r,r))
title('Reverse Cuthill-McKee')
```



Symmetric Reordering - Column Permutations

Use `j = COLPERM(A)` to return a permutation vector that reorders the columns of the sparse matrix `A` in nondecreasing order of nonzero count. This form is sometimes useful as a preordering for LU factorization, as in `lu(A(:,j))`.

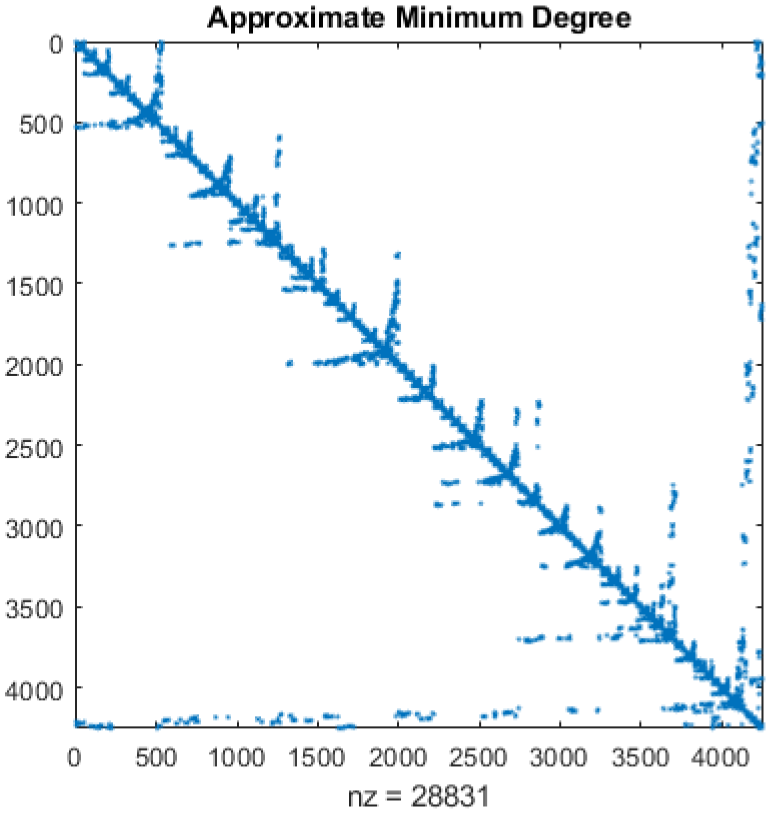
```
j = colperm(A);
spy(A(j,j))
title('Column Count Reordering')
```



Symmetric Reordering - Symmetric Approximate Minimum Degree

symamd gives a symmetric approximate minimum degree permutation. For a symmetric positive definite matrix A, the command p = symamd(S) returns the permutation vector p such that S(p,p) tends to have a sparser Cholesky factor than S. Sometimes symamd works well for symmetric indefinite matrices too.

```
m = symamd(A);
spy(A(m,m))
title('Approximate Minimum Degree')
```



See Also

`colperm` | `spy` | `symamd` | `symrcm`

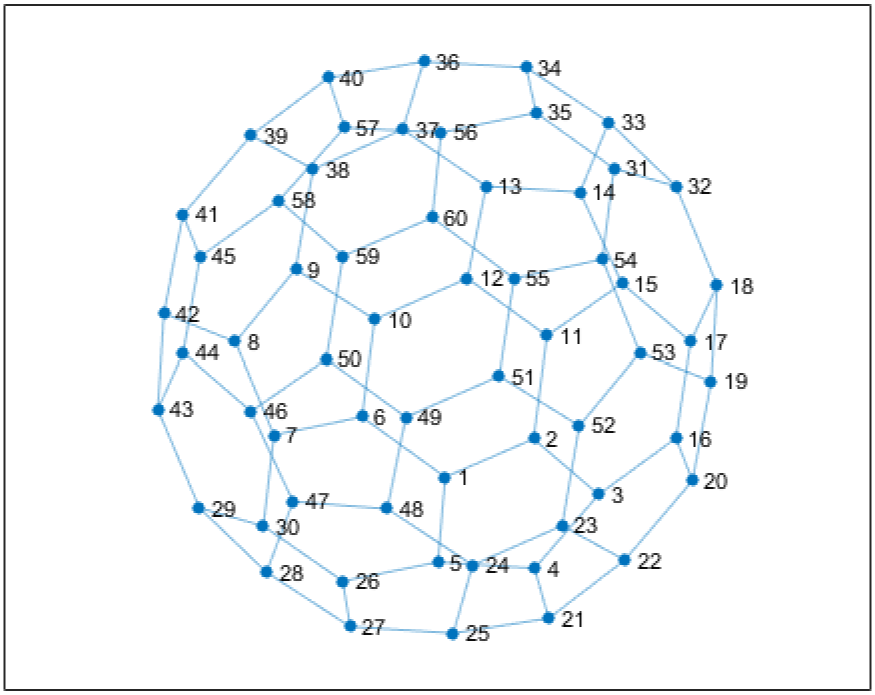
Graphs and Matrices

This example shows an application of sparse matrices and explains the relationship between graphs and matrices.

A graph is a set of nodes with specified connections, or edges, between them. Graphs come in many shapes and sizes. One example is the connectivity graph of the Buckminster Fuller geodesic dome, which is also in the shape of a soccer ball or a carbon-60 molecule.

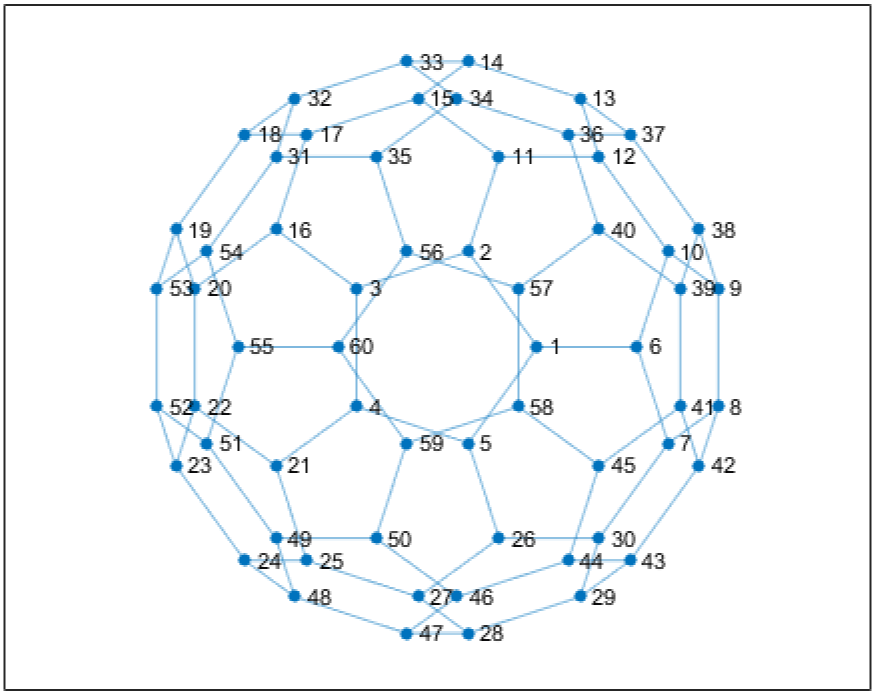
In MATLAB®, you can use the bucky function to generate the graph of the geodesic dome.

```
[B,V] = bucky;
G = graph(B);
p = plot(G);
axis equal
```



You also can specify coordinates for the nodes to change the display of the graph.

```
p.XData = V(:,1);
p.YData = V(:,2);
```

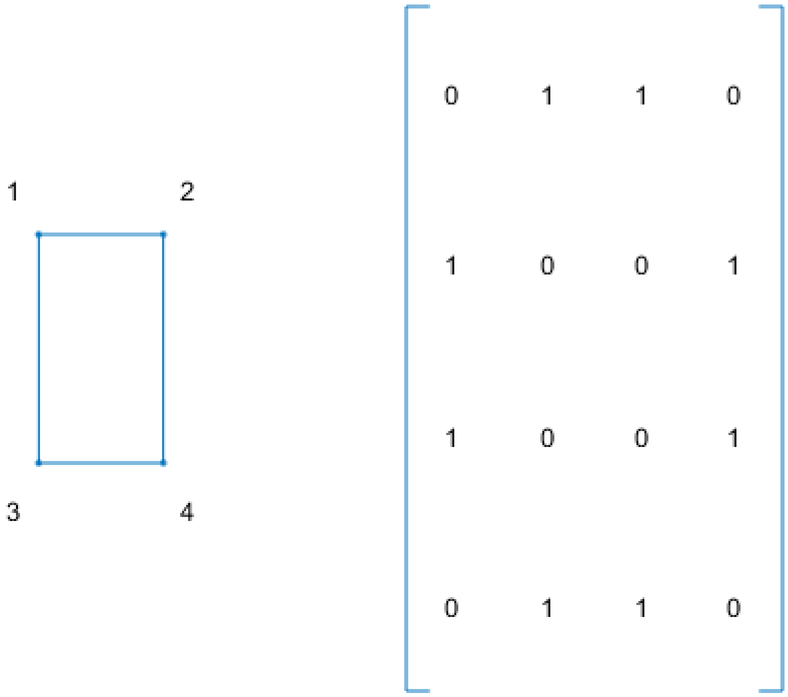


The `bucky` function can be used to create the graph because it returns an adjacency matrix. An adjacency matrix is one way to represent the nodes and edges in a graph.

To construct the adjacency matrix of a graph, the nodes are numbered 1 to N. Then each element (i,j) of the N-by-N matrix is set to 1 if node i is connected to node j, and 0 otherwise. Thus, for undirected graphs the adjacency matrix is symmetric, but this need not be the case for directed graphs.

For example, here is a simple graph and its associated adjacency matrix.

```
% Define a matrix A.
A = [0 1 1 0 ; 1 0 0 1 ; 1 0 0 1 ; 0 1 1 0];

% Draw a picture showing the connected nodes.
cla
subplot(1,2,1);
gplot(A,[0 1;1 1;0 0;1 0],'.-');
text([-0.2, 1.2 -0.2, 1.2],[1.2, 1.2, -.2, -.2],('1234')', ...
   'HorizontalAlignment','center')
axis([-1 2 -1 2],'off')

% Draw a picture showing the adjacency matrix.
subplot(1,2,2);
xtemp = repmat(1:4,1,4);
ytemp = reshape(repmat(1:4,4,1),16,1)';
text(xtemp-.5,ytemp-.5,char('0'+A(:)),'HorizontalAlignment','center');
line([.25 0 0 .25 NaN 3.75 4 4 3.75],[0 0 4 4 NaN 0 0 4 4])
axis off tight
```



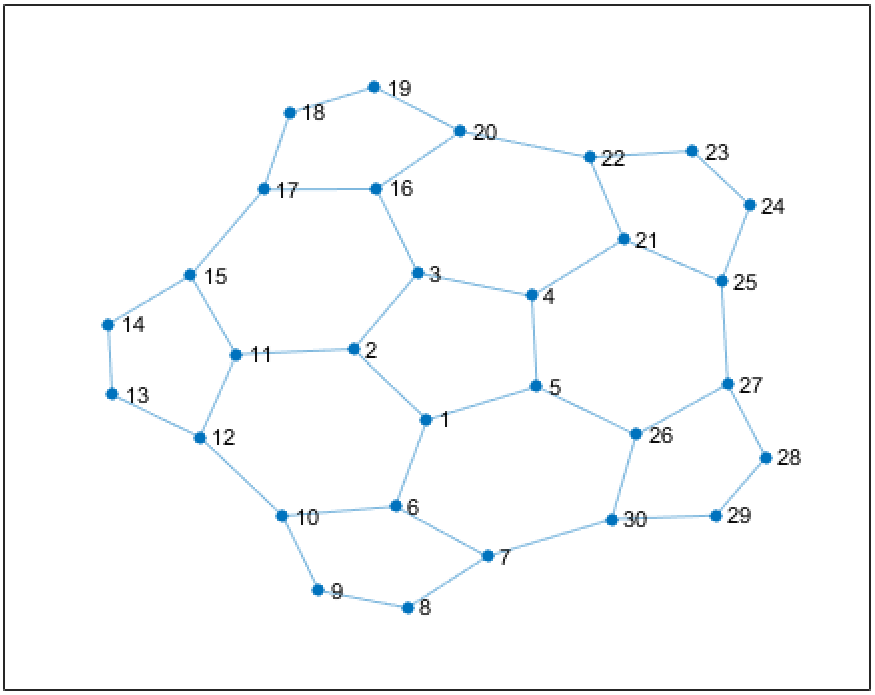
$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Sparse matrices are particularly helpful for representing very large graphs. This is because each node is usually connected to only a few other nodes. As a result, the density of nonzero entries in the adjacency matrix is often relatively small for large graphs. The bucky ball adjacency matrix is a good example, since it is a 60-by-60 symmetric sparse matrix with only 180 nonzero elements. The density of this matrix is just 5%.

Since the adjacency matrix defines the graph, you can plot a portion of the bucky ball by using a subset of the entries in the adjacency matrix.

Use the `adjacency` function to create a new adjacency matrix for the graph. Display the nodes in one hemisphere of the bucky ball by indexing into the adjacency matrix to create a new, smaller graph.

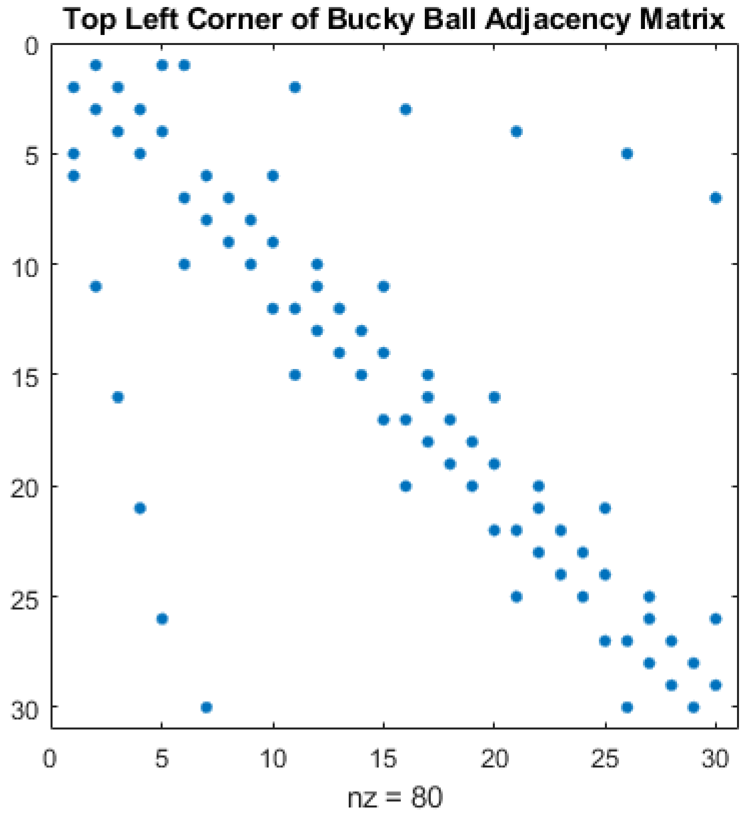
```
figure
A = adjacency(G);
H = graph(A(1:30,1:30));
h = plot(H);
```



To visualize the adjacency matrix of this hemisphere, use the spy function to plot the silhouette of the nonzero elements in the adjacency matrix.

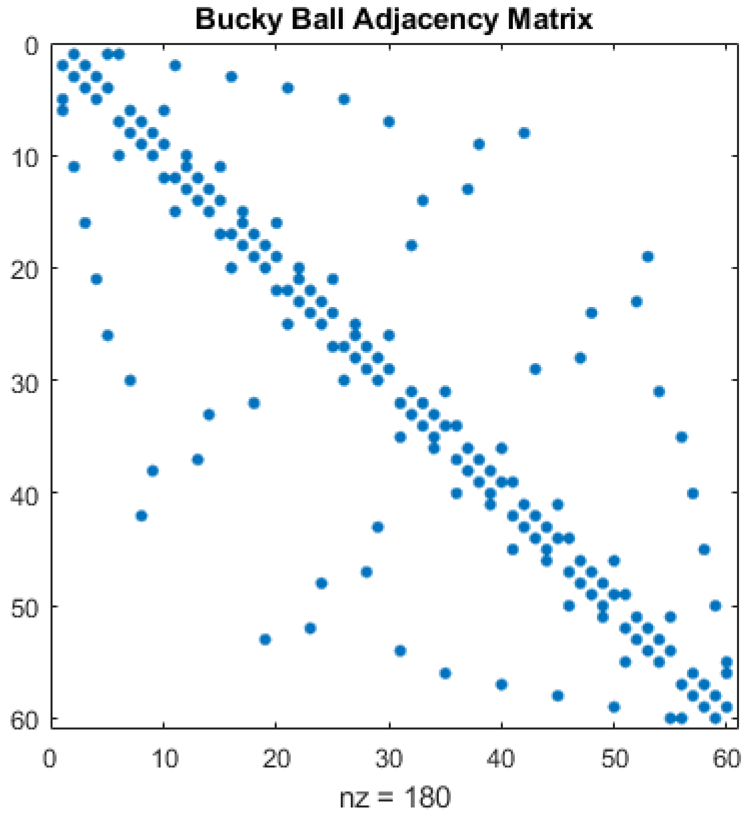
Note that the matrix is symmetric, since if node i is connected to node j, then node j is connected to node i.

```
spy(A(1:30,1:30))
title('Top Left Corner of Bucky Ball Adjacency Matrix')
```



Finally, here is a `spy` plot of the entire bucky ball adjacency matrix.

```
spy(A)
title('Bucky Ball Adjacency Matrix')
```



See Also

`graph` | `spy`

Sparse Matrix Reordering

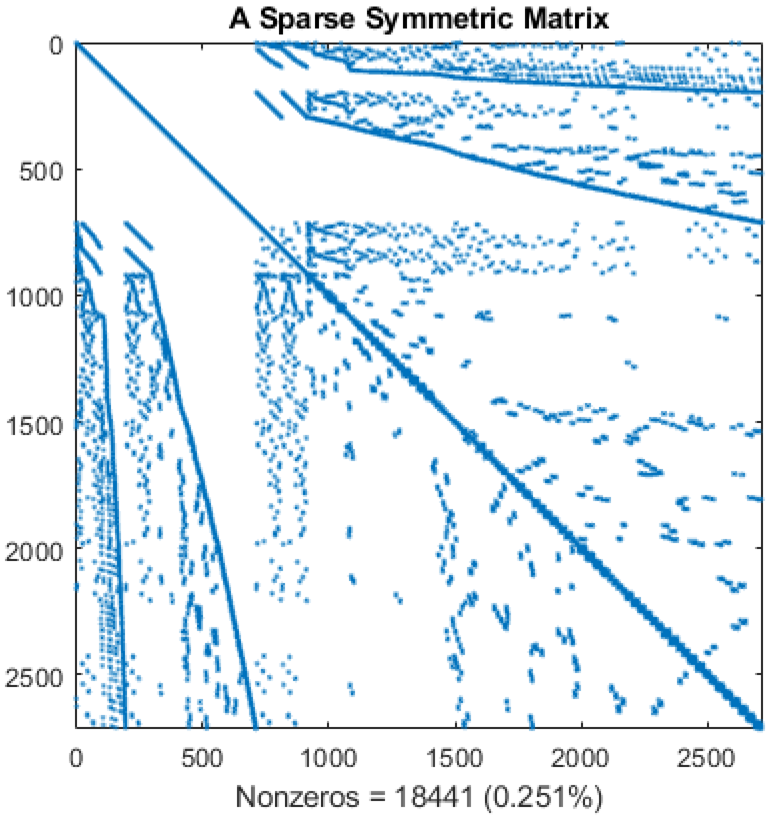
This example shows how reordering the rows and columns of a sparse matrix can influence the speed and storage requirements of a matrix operation.

Visualizing a Sparse Matrix

A spy plot shows the nonzero elements in a matrix.

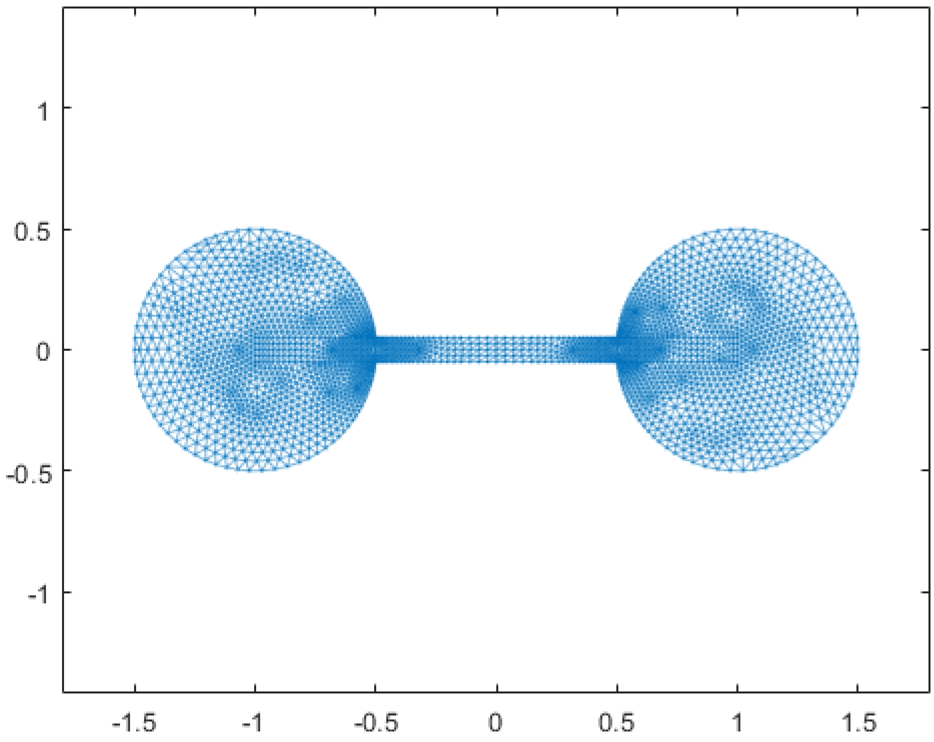
This spy plot shows a sparse symmetric positive definite matrix derived from a portion of the barbell matrix. This matrix describes connections in a graph that resembles a barbell.

```
load barbellgraph.mat
S = A + speye(size(A));
pct = 100 / numel(A);
spy(S)
title('A Sparse Symmetric Matrix')
nz = nnz(S);
xlabel(sprintf('Nonzeros = %d (%.3f%%)',nz,nz*pct));
```



Here is a plot of the barbell graph.

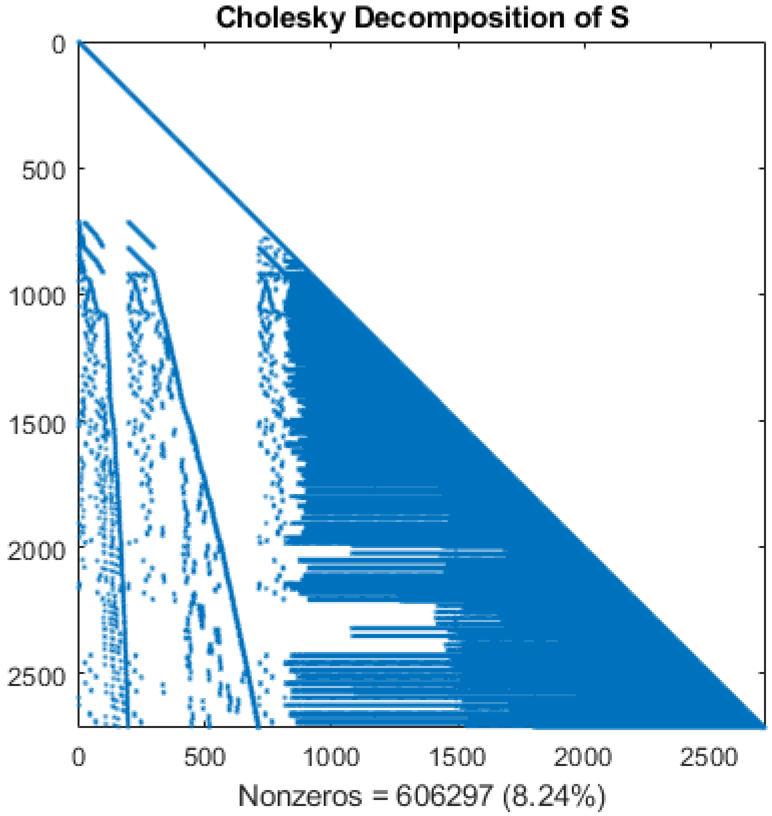
```
G = graph(S,'omitselfloops');
p = plot(G,'XData',xy(:,1),'YData',xy(:,2),'Marker','.');
axis equal
```



Computing the Cholesky Factor

Compute the Cholesky factor L, where S = L*L'. Notice that L contains *many* more nonzero elements than the unfactored S, because the computation of the Cholesky factorization creates **fill-in** nonzeros. These fill-in values slow down the algorithm and increase storage cost.

```
L = chol(S,'lower');
spy(L)
title('Cholesky Decomposition of S')
nc(1) = nnz(L);
xlabel(sprintf('Nonzeros = %d (%.2f%%)',nc(1),nc(1)*pct));
```



Reordering to Speed Up Calculation

By reordering the rows and columns of a matrix, it is possible to reduce the amount of fill-in that factorization creates, thereby reducing the time and storage cost of subsequent calculations.

Several different reorderings supported by MATLAB® are:

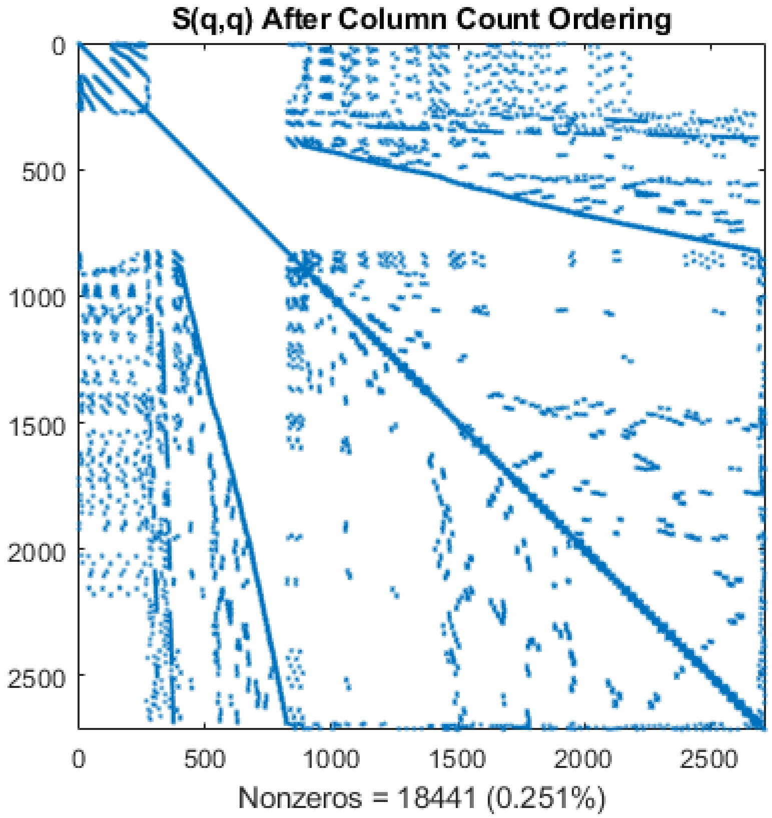
- `colperm`: Column count
- `symrcm`: Reverse Cuthill-McKee
- `amd`: Minimum degree
- `dissect`: Nested dissection

Test the effects of these sparse matrix reorderings on the barbell matrix.

Column Count Reordering

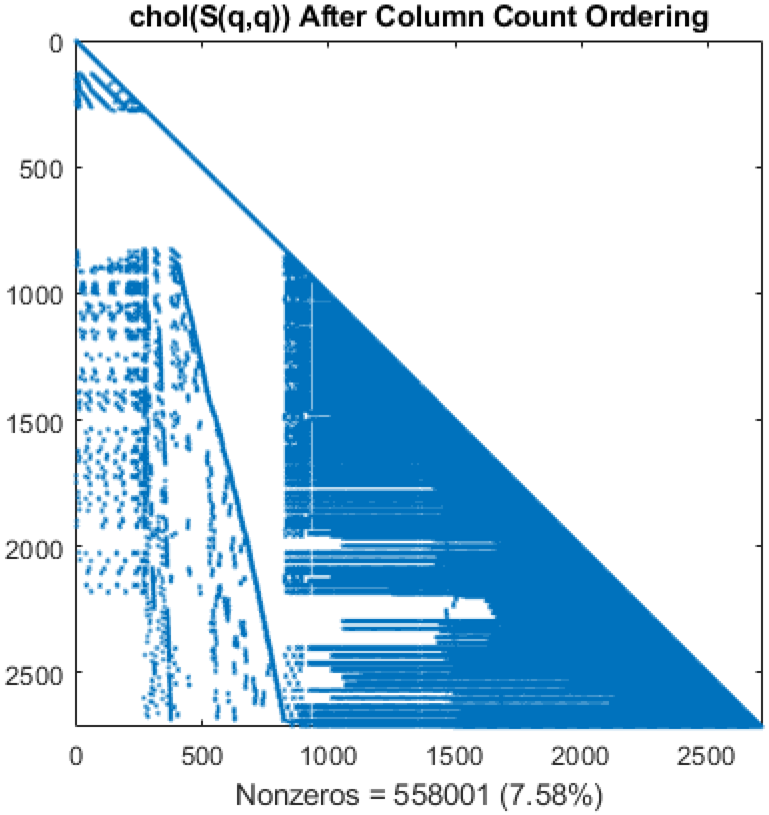
The `colperm` command uses the column count reordering algorithm to move rows and columns with higher nonzero count towards the end of the matrix.

```
q = colperm(S);
spy(S(q,q))
title('S(q,q) After Column Count Ordering')
nz = nnz(S);
xlabel(sprintf('Nonzeros = %d (%.3f%%)',nz,nz*pct));
```



For this matrix, the column count ordering can barely reduce the time and storage for Cholesky factorization.

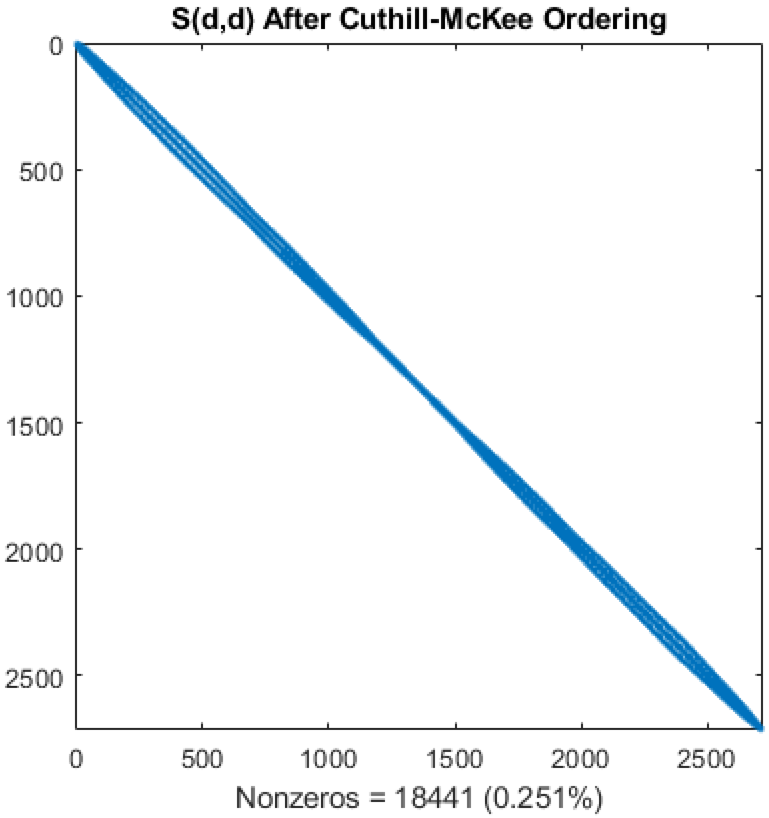
```
L = chol(S(q,q),'lower');
spy(L)
title('chol(S(q,q)) After Column Count Ordering')
nc(2) = nnz(L);
xlabel(sprintf('Nonzeros = %d (%.2f%%)',nc(2),nc(2)*pct));
```



Reverse Cuthill-McKee Reordering

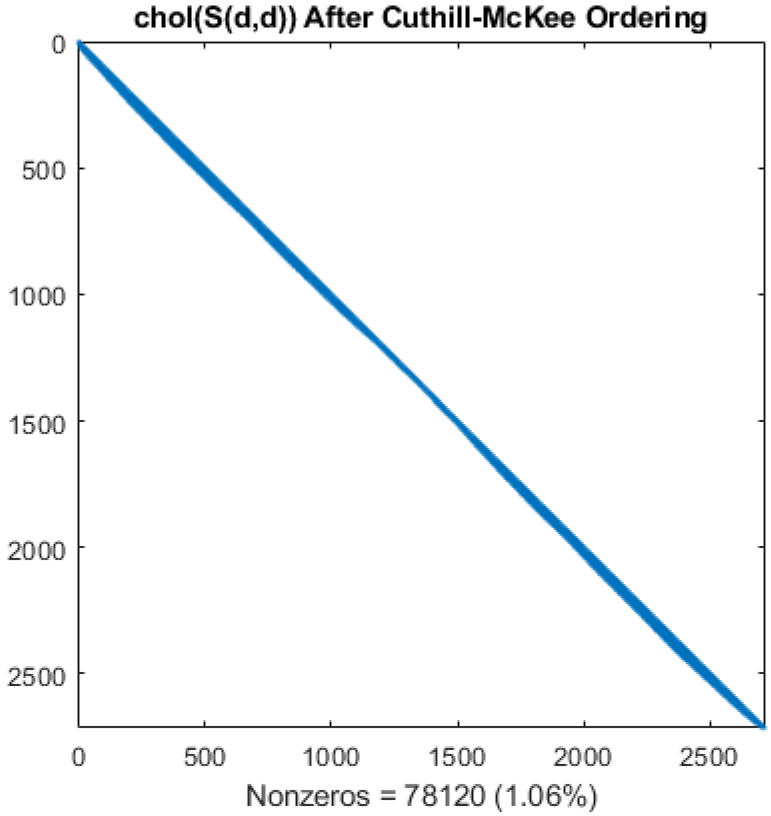
The `symrcm` command uses the reverse Cuthill-McKee reordering algorithm to move all nonzero elements closer to the diagonal, reducing the bandwidth of the original matrix.

```
d = symrcm(S);
spy(S(d,d))
title('S(d,d) After Cuthill-McKee Ordering')
nz = nnz(S);
xlabel(sprintf('Nonzeros = %d (%.3f%%)',nz,nz*pct));
```



The fill-in produced by Cholesky factorization is confined to the band, so factorizing the reordered matrix takes less time and less storage.

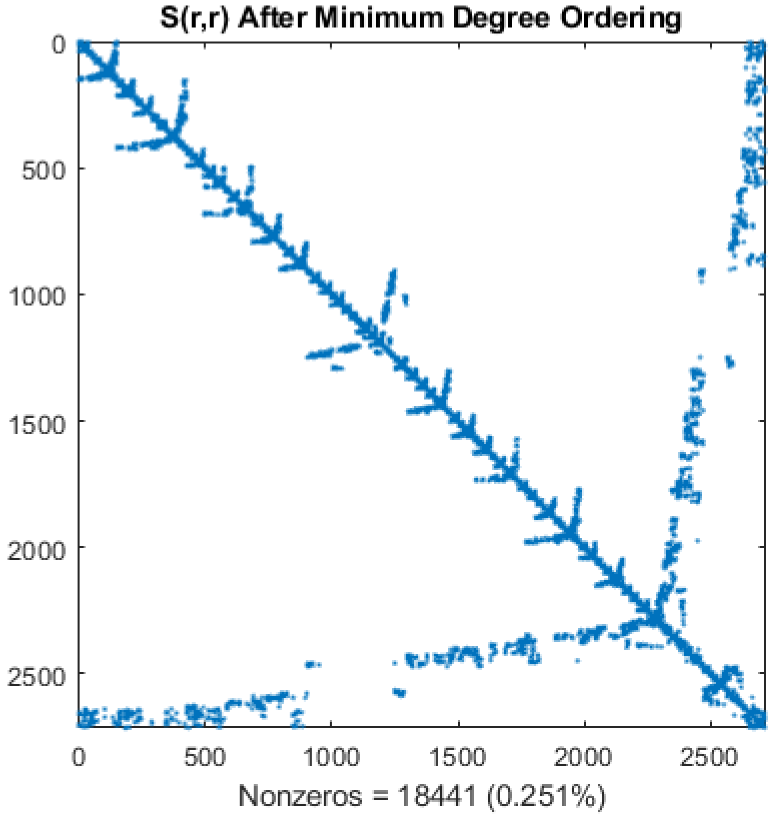
```
L = chol(S(d,d),'lower');
spy(L)
title('chol(S(d,d)) After Cuthill-McKee Ordering')
nc(3) = nnz(L);
xlabel(sprintf('Nonzeros = %d (%.2f%%)', nc(3),nc(3)*pct));
```



Minimum Degree Reordering

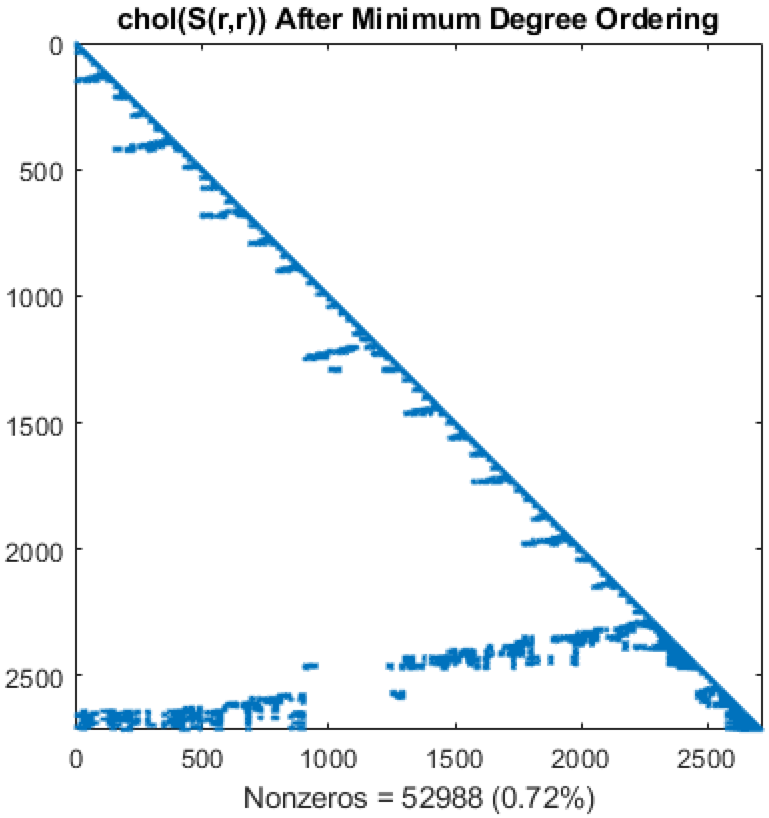
The amd command uses an approximate minimum degree algorithm (a powerful graph-theoretic technique) to produce large blocks of zeros in the matrix.

```
r = amd(S);
spy(S(r,r))
title('S(r,r) After Minimum Degree Ordering')
nz = nnz(S);
xlabel(sprintf('Nonzeros = %d (%.3f%%)',nz,nz*pct));
```



The Cholesky factorization preserves the blocks of zeros produced by the minimum degree algorithm. This structure can significantly reduce time and storage costs.

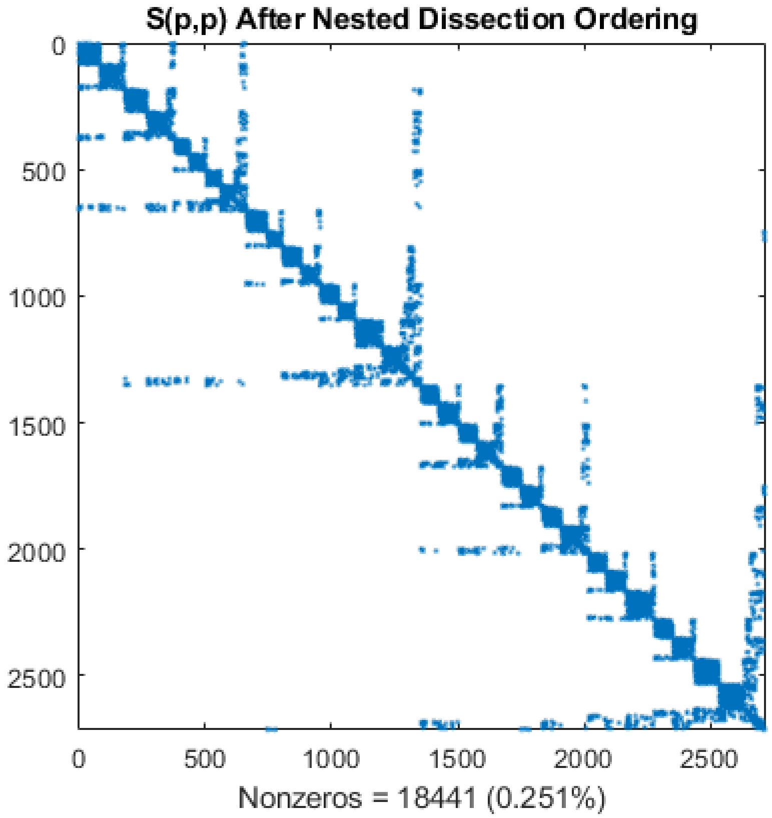
```
L = chol(S(r,r),'lower');
spy(L)
title('chol(S(r,r)) After Minimum Degree Ordering')
nc(4) = nnz(L);
xlabel(sprintf('Nonzeros = %d (%.2f%%)',nc(4),nc(4)*pct));
```



Nested Dissection Permutation

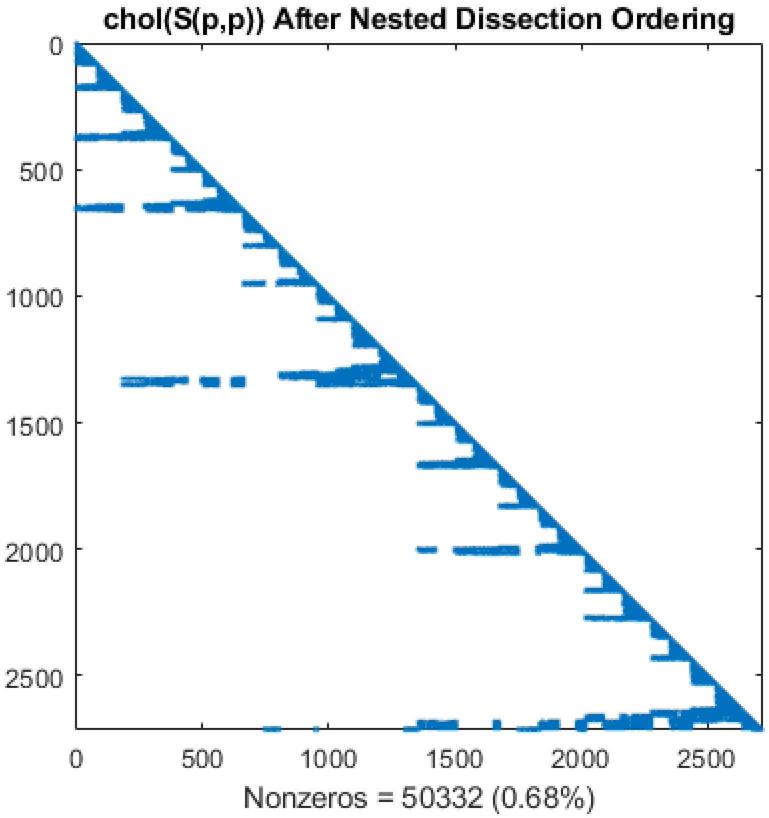
The `dissect` function uses graph-theoretic techniques to produce fill-reducing orderings. The algorithm treats the matrix as the adjacency matrix of a graph, coarsens the graph by collapsing vertices and edges, reorders the smaller graph, and then uses refinement steps to uncoarsen the small graph and produce a reordering of the original graph. The result is a powerful algorithm that frequently produces the least amount of fill-in compared to the other reordering algorithms.

```
p = dissect(S);
spy(S(p,p))
title('S(p,p) After Nested Dissection Ordering')
nz = nnz(S);
xlabel(sprintf('Nonzeros = %d (%.3f%%)',nz,nz*pct));
```



Similar to the minimum degree ordering, the Cholesky factorization of the nested dissection ordering mostly preserves the nonzero structure of S(d,d) below the main diagonal.

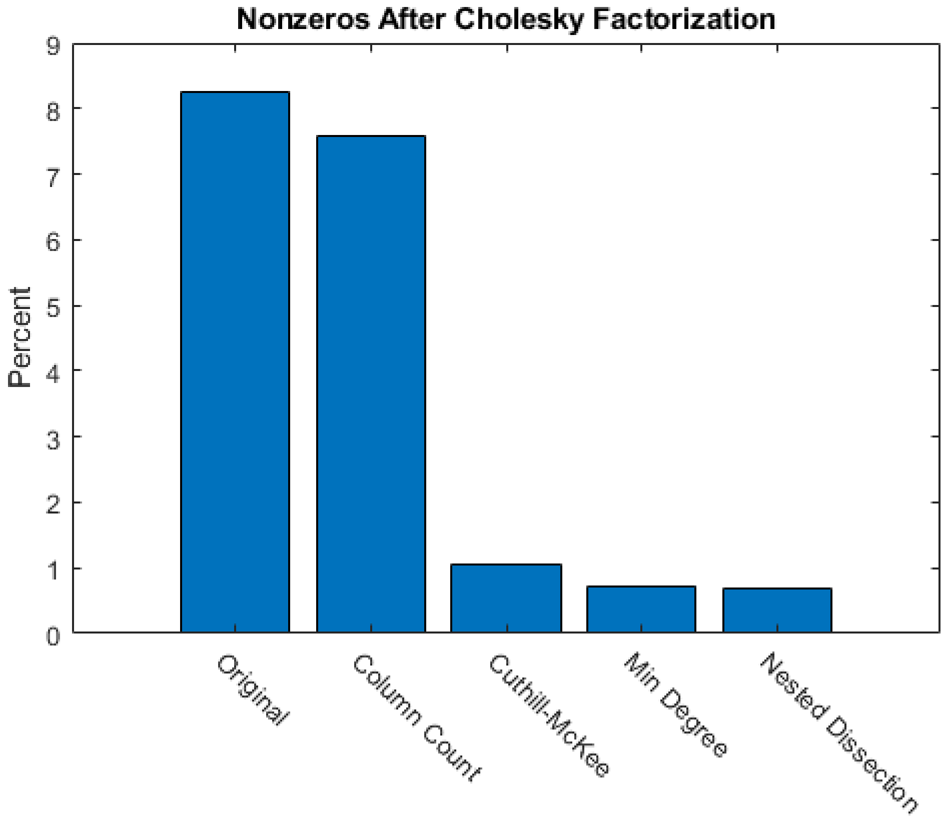
```
L = chol(S(p,p),'lower');
spy(L)
title('chol(S(p,p)) After Nested Dissection Ordering')
nc(5) = nnz(L);
xlabel(sprintf('Nonzeros = %d (%.2f%%)',nc(5),nc(5)*pct));
```



Summarizing Results

This bar chart summarizes the effects of reordering the matrix before performing the Cholesky factorization. While the Cholesky factorization of the original matrix had about 8% of its elements as nonzeros, using `dissect` or `symamd` reduces that density to less than 1%.

```
labels = {'Original','Column Count','Cuthill-McKee',...
    'Min Degree','Nested Dissection'};
bar(nc*pct)
title('Nonzeros After Cholesky Factorization')
ylabel('Percent');
ax = gca;
ax.XTickLabel = labels;
ax.XTickLabelRotation = -45;
```



See Also

chol | colperm | nnz | spy | symamd | symrcm

Iterative Methods for Linear Systems

One of the most important and common applications of numerical linear algebra is the solution of linear systems that can be expressed in the form `A*x = b`. When `A` is a large sparse matrix, you can solve the linear system using iterative methods, which enable you to trade-off between the run time of the calculation and the precision of the solution. This topic describes the iterative methods available in MATLAB to solve the equation `A*x = b`.

Direct vs. Iterative Methods

There are two types of methods for solving linear equations `A*x = b`:

- **Direct methods** are variants of Gaussian elimination. These methods use the individual matrix elements directly, through matrix operations such as LU, QR, or Cholesky factorization. You can use direct methods to solve linear equations with a high level of precision, but these methods can be slow when operating on large sparse matrices. The speed of solving a linear system with a direct method strongly depends on the density and fill pattern of the coefficient matrix.

  For example, this code solves a small linear system.

  ```
  A = magic(5);
  b = sum(A,2);
  x = A\b;
  norm(A*x-b)
  ```

  ```
  ans =

     1.4211e-14
  ```

  MATLAB implements direct methods through the matrix division operators `/` and `\`, as well as functions such as `decomposition`, `lsqminnorm`, and `linsolve`.
- **Iterative methods** produce an approximate solution to the linear system after a finite number of steps. These methods are useful for large systems of equations where it is reasonable to trade-off precision for a shorter run time. Iterative methods use the coefficient matrix only indirectly, through a matrix-vector product or an abstract linear operator. Iterative methods can be used with any matrix, but they are typically applied to large sparse matrices for which direct solves are slow. The speed of solving a linear system with an indirect method does not depend as strongly on the fill pattern of the coefficient matrix as a direct method. However, using an iterative method typically requires tuning parameters for each specific problem.

  For example, this code solves a large sparse linear system that has a symmetric positive definite coefficient matrix.

  ```
  A = delsq(numgrid('L',400));
  b = ones(size(A,1),1);
  x = pcg(A,b,[],1000);
  norm(b-A*x)
  ```

  ```
  pcg converged at iteration 796 to a solution with relative residual 9.9e-07.

  ans =

     3.4285e-04
  ```

  MATLAB implements a variety of iterative methods that have different strengths and weaknesses depending on the properties of the coefficient matrix `A`.

Direct methods are usually faster and more generally applicable than indirect methods if there is enough storage available to carry them out. Generally, you should attempt to use `x = A\b` first. If the direct solve is too slow, then you can try using iterative methods.

Generic Iterative Algorithm

Most iterative algorithms that solve linear equations follow a similar process:

1. Start with an initial guess for the solution vector `x0`. (This is usually a vector of zeros unless you specify a better guess.)
2. Compute the residual norm `res = norm(b-A*x0)`.
3. Compare the residual against the specified tolerance. If `res <= tol`, end the computation and return the computed answer for `x0`.
4. Apply `A*x0` and update the magnitude and direction of the vector `x0` based on the value of the residual and other calculated quantities. This is the step where most computation is done.
5. Repeat Steps 2 through 4 until the value of `x0` is good enough to satisfy the tolerance.

Iterative methods differ in how they update the magnitude and direction of `x0` in Step 4, and some have slightly different convergence criteria in Steps 2 and 3, but this captures the basic process that all iterative solvers follow.

Summary of Iterative Methods

MATLAB has several functions that implement iterative methods for systems of linear equations. These methods are designed to solve $Ax = b$ or minimize the norm $||b - Ax||$. Several of these methods have similarities and are based on the same underlying algorithms, but each algorithm has benefits in certain situations [1], [2].

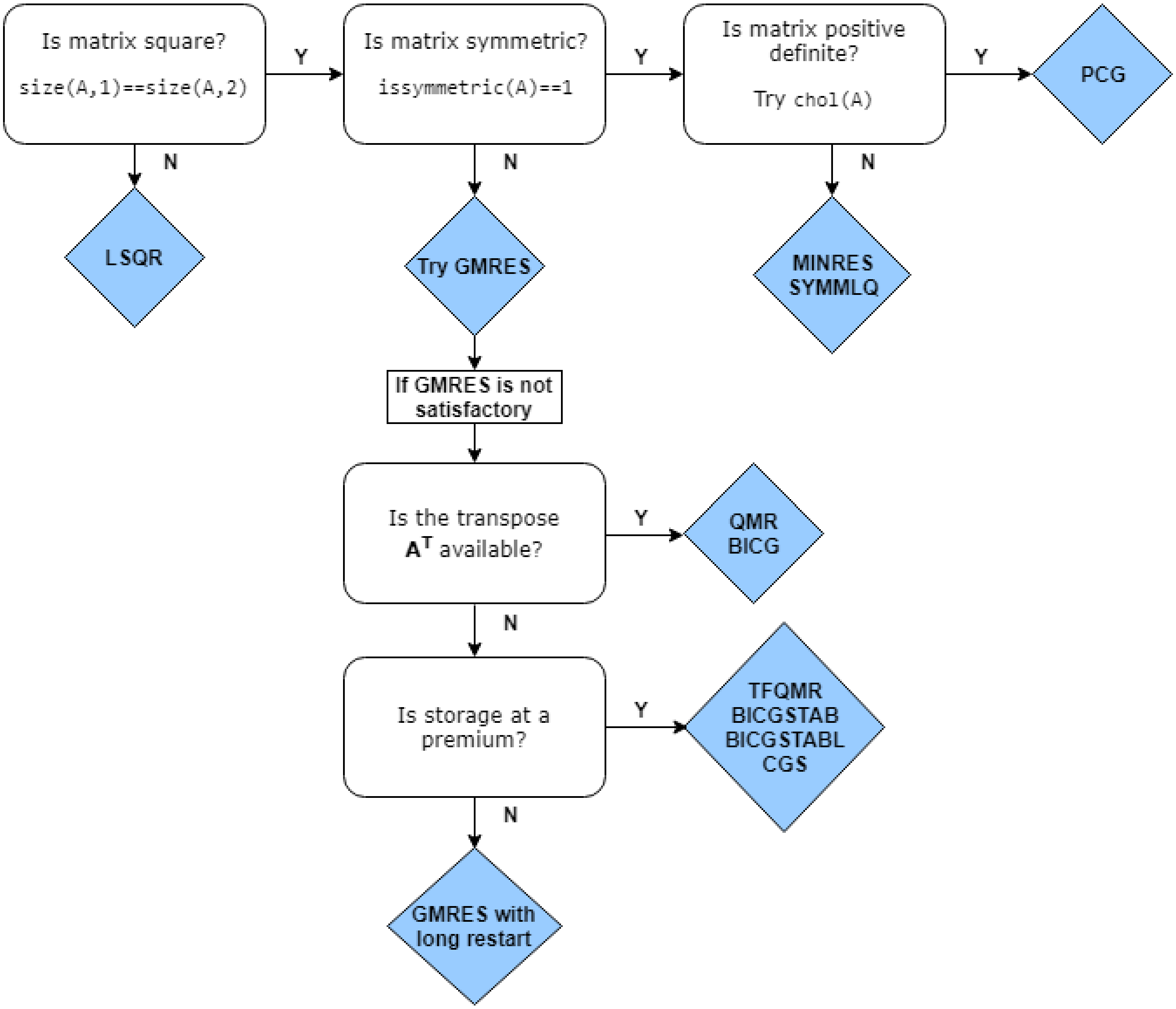
Description	Notes
`pcg` (preconditioned conjugate gradients)	• Coefficient matrix must be symmetric positive definite. • Most effective solver for symmetric positive definite systems since storage for only a limited number of vectors is required.
`lsqr` (least squares)	• The only solver available for rectangular systems. • Analytically equivalent to the method of conjugate gradients (PCG) applied to the normal equations `(A'*A)*x = A'*b`.

Description	Notes
`minres` (minimum residual)	• Coefficient matrix must be symmetric but does not need to be positive definite. • Each iteration minimizes the residual error in the 2-norm, so the algorithm is guaranteed to make progress from step to step. • Does not suffer from breakdowns (when an algorithm becomes unable to make progress toward a solution and halts).
`symmlq` (symmetric LQ)	• Coefficient matrix must be symmetric but does not need to be positive definite. • Solves a projected system and keeps the residual orthogonal to all previous ones. • Does not suffer from breakdowns (when an algorithm becomes unable to make progress toward a solution and halts).
`bicg` (biconjugate gradient)	• Coefficient matrix must be square. • `bicg` is computationally cheap, but convergence is irregular and unreliable. • `bicg` is historically important because many other iterative algorithms were developed as improvements on it.
`bicgstab` (biconjugate gradient stabilized)	• Coefficient matrix must be square. • Uses BiCG steps alternating with GMRES(1) steps for additional stability.
`bicgstabl` (biconjugate gradient stabilized (l))	• Coefficient matrix must be square. • Uses BiCG steps alternating with GMRES(2) steps for additional stability.
`cgs` (conjugate gradient squared)	• Coefficient matrix must be square. • Requires the same number of operations per iteration as `bicg`, but avoids using the transpose by working with a squared residual.

Description	Notes
`gmres` (generalized minimum residual)	• Coefficient matrix must be square. • One of the most dependable algorithms, since the residual norm is minimized in each iteration. • Work and required storage rise linearly with iteration count. • Choosing an appropriate `restart` value is essential to avoid unnecessary work and storage.
`qmr` (quasi-minimal residual)	• Coefficient matrix must be square. • Overhead per iteration is slightly more than `bicg`, but this provides more stability.
`tfqmr` (transpose-free quasi-minimal residual)	• Coefficient matrix must be square. • Best solver to try for symmetric indefinite systems when memory is limited.

Choosing an Iterative Solver

This flowchart of iterative solvers in MATLAB gives a rough idea of the situations where each solver is useful. You can generally use `gmres` for almost all square, nonsymmetric problems. There are some cases where the biconjugate gradients algorithms (`bicg`, `bicgstab`, `cgs`, and so on) are more efficient than `gmres`, but their unpredictable convergence behavior often makes `gmres` a better initial choice.



Preconditioners

The convergence rate of iterative methods is dependent on the spectrum (eigenvalues) of the coefficient matrix. Therefore, you can improve the convergence and stability of most iterative methods by transforming the linear system to have a more favorable spectrum (clustered eigenvalues or a condition number near 1). This transformation is performed by applying a second matrix, called a preconditioner, to the system. This process transforms the linear system

$$Ax = b$$

into an equivalent system

$$\widetilde{A}\widetilde{x} = \widetilde{b}\ .$$

The ideal preconditioner transforms the coefficient matrix A into an identity matrix, since any iterative method will converge in one iteration with such a preconditioner. In practice, finding a good preconditioner requires trade-offs. The transformation is performed in one of three ways: left preconditioning, right preconditioning, or split preconditioning.

The first case is called *left preconditioning* since the preconditioner matrix M appears on the left of A:

$$\left(M^{-1}A\right)x = \left(M^{-1}b\right) .$$

These iterative solvers use left preconditioning:

- `bicg`
- `gmres`
- `qmr`

In *right preconditioning*, M appears on the right of A:

$$\left(A\,M^{-1}\right)(M\,x) = b\ .$$

These iterative solvers use right preconditioning:

- `lsqr`
- `bicgstab`
- `bicgstabl`
- `cgs`
- `tfqmr`

Finally, for symmetric coefficient matrices `A`, *split preconditioning* ensures that the transformed system is still symmetric. The preconditioner $M = HH^T$ gets split and the factors appear on different sides of A:

$$\left(H^{-1}A\,H^{-T}\right)H^Tx = \left(H^{-1}b\right)$$

The solver algorithm for split preconditioned systems is based on the above equation, but in practice there is no need to compute H. The solver algorithm multiplies and solves with `M` directly.

These iterative solvers use split preconditioning:

- `pcg`
- `minres`
- `symmlq`

In all cases, the preconditioner M is chosen to accelerate convergence of the iterative method. When the residual error of an iterative solution stagnates or makes little progress between iterations, it often means you need to generate a preconditioner matrix to incorporate into the problem.

The iterative solvers in MATLAB allow you to specify a single preconditioner matrix M, or two preconditioner matrix factors such that $M = M_1M_2$. This makes it easy to specify a preconditioner in its factorized form, such as $M = LU$. Note that in the split preconditioned case, where $M = HH^{\mathrm{T}}$ also holds, there is not a relation between the `M1` and `M2` inputs and the H factors.

In some cases, preconditioners occur naturally in the mathematical model of a given problem. In the absence of natural preconditioners, you can use one of the incomplete factorizations in this table to generate a preconditioner matrix. Incomplete factorizations are essentially incomplete direct solves that are quick to calculate.

Function	Factorization	Description
`ilu`	A≈LU	Incomplete LU factorization for square or rectangular matrices.
`ichol`	A ≈ L L*	Incomplete Cholesky factorization for symmetric positive definite matrices.

See “Incomplete Factorizations” on page 4-22 for more information about `ilu` and `ichol`.

Preconditioner Example

Consider the five-point finite difference approximation to Laplace's equation on a square, two-dimensional domain. The following commands use the preconditioned conjugate gradient (PCG) method with the preconditioner `M = L*L'`, where `L` is the zero-fill incomplete Cholesky factor of `A`. For this system, `pcg` is unable to find a solution without specifying a preconditioner matrix.

```
A = delsq(numgrid('S',250));
b = ones(size(A,1),1);
tol = 1e-3;
maxit = 100;
L = ichol(A);
x = pcg(A,b,tol,maxit,L,L');
```

```
pcg converged at iteration 92 to a solution with relative residual 0.00076.
```

`pcg` requires 92 iterations to achieve the specified tolerance. However, using a different preconditioner can yield better results. For example, using `ichol` to construct a modified incomplete Cholesky allows `pcg` to meet the specified tolerance after only 39 iterations.

```
L = ichol(A,struct('type','nofill','michol','on'));
x = pcg(A,b,tol,maxit,L,L');
```

```
pcg converged at iteration 39 to a solution with relative residual 0.00098.
```

Equilibration and Reordering

For computationally tough problems, you might need a better preconditioner than the one generated by `ilu` or `ichol` directly. For example, you might want to generate a better quality preconditioner or minimize the amount of computation being done. In these cases, you can use *equilibration* to make the coefficient matrix more diagonally dominant (which can lead to a better quality preconditioner) and *reordering* to minimize the number of nonzeros in matrix factors (which can reduce memory requirements and may improve the efficiency of subsequent calculations).

If you use both equilibration and reordering to generate a preconditioner, the process is:

1. Use `equilibrate` on the coefficient matrix.
2. Reorder the equilibrated matrix using a sparse matrix reordering function, such as `dissect` or `symrcm`.
3. Generate the final preconditioner using `ilu` or `ichol`.

Here is an example that uses equilibration and reordering to generate a preconditioner for a sparse coefficient matrix.

1. Create the coefficient matrix A and a vector of ones b for the right-hand side of the linear equation. Calculate an estimation of the condition number for A.

   ```
   load west0479;
   A = west0479;
   b = ones(size(A,1),1);
   condest(A)
   ```

   ```
   ans =

      1.4244e+12
   ```

   Use equilibrate to improve the condition number of the coefficient matrix.

   ```
   [P,R,C] = equilibrate(A);
   Anew = R*P*A*C;
   bnew = R*P*b;
   condest(Anew)
   ```

   ```
   ans =

      5.1042e+04
   ```
2. Reorder the equilibrated matrix using dissect.

   ```
   q = dissect(Anew);
   Anew = Anew(q,q);
   bnew = bnew(q);
   ```
3. Generate a preconditioner using an incomplete LU factorization.

   ```
   [L,U] = ilu(Anew);
   ```
4. Solve the linear system with gmres using the preconditioner matrices, a tolerance of 1e-10, 50 maximum outer iterations, and 30 inner iterations.

   ```
   tol = 1e-10;
   maxit = 50;
   restart = 30;
   [xnew, flag, relres] = gmres(Anew,bnew,restart,tol,maxit,L,U);
   x(q) = xnew;
   x = C*x(:);
   ```

   Now, compare the relres relative residual returned by gmres (which includes the preconditioners) to the relative residual without the preconditioners resnew and the relative residual without equilibration res. The results show that even though the linear systems are all equivalent, the different methods apply different weights to each element, and this can significantly affect the value of the residual.

   ```
   relres
   resnew = norm(Anew*xnew - bnew) / norm(bnew)
   res = norm(A*x - b) / norm(b)
   ```

   ```
   relres =
      8.7537e-11
   resnew =
      3.6805e-08
   res =
      5.1415e-04
   ```

Using Linear Operators Instead of Matrices

The iterative solvers in MATLAB do not *require* that you provide a numeric matrix for A. Since the calculations performed by the solvers use the result of the matrix-vector multiplication A*x or A'*x, you can instead provide a function that calculates the result of those linear operations. A function that calculates these quantities is often called a linear operator.

In addition to using a linear operator instead of a coefficient matrix A, you can also use a linear operator instead of a matrix for the preconditioner M. In that case, the function needs to calculate M\x or M'\x, as indicated on the reference page for the solver.

Using linear operators enables you to exploit patterns in A or M to calculate the value of the linear operations more efficiently than if the solver used the matrix explicitly to carry out the full matrix-vector multiplication. It also means you do not need the memory to store the coefficient or preconditioner matrices, since the linear operator typically calculates the result of the matrix-vector multiplication without forming the matrix at all.

For example, consider the coefficient matrix

```
A = [2 -1  0  0  0  0;
    -1  2 -1  0  0  0;
     0 -1  2 -1  0  0;
     0  0 -1  2 -1  0;
     0  0  0 -1  2 -1;
     0  0  0  0 -1  2];
```

When A multiplies a vector, most of the elements in the resulting vector are zeros. The nonzero elements in the result correspond with the nonzero tridiagonal elements of A. So, for a given vector x, the linear operator function simply needs to add together three vectors to calculate the value of A*x:

```
function y = linearOperatorA(x)
y = -1*[0; x(1:end-1)] ...
  + 2*x ...
  + -1*[x(2:end); 0];
end
```

Most iterative solvers require the linear operator function for A to return the value of A*x. Likewise, for the preconditioner matrix M, the function generally must calculate M\x. For the solvers lsqr, qmr, and bicg, the linear operator functions must also return the value for A'*x or M'\x when requested. See the iterative solver reference pages for examples and descriptions of linear operator functions.

References

[1] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[2] Saad, Yousef, *Iterative Methods for Sparse Linear Equations*. PWS Publishing Company, 1996.

See Also

More About

- “Systems of Linear Equations” on page 2-10

Graph and Network Algorithms

Directed and Undirected Graphs

In this section...

What Is a Graph?

A graph is a collection of *nodes* and *edges* that represents relationships:

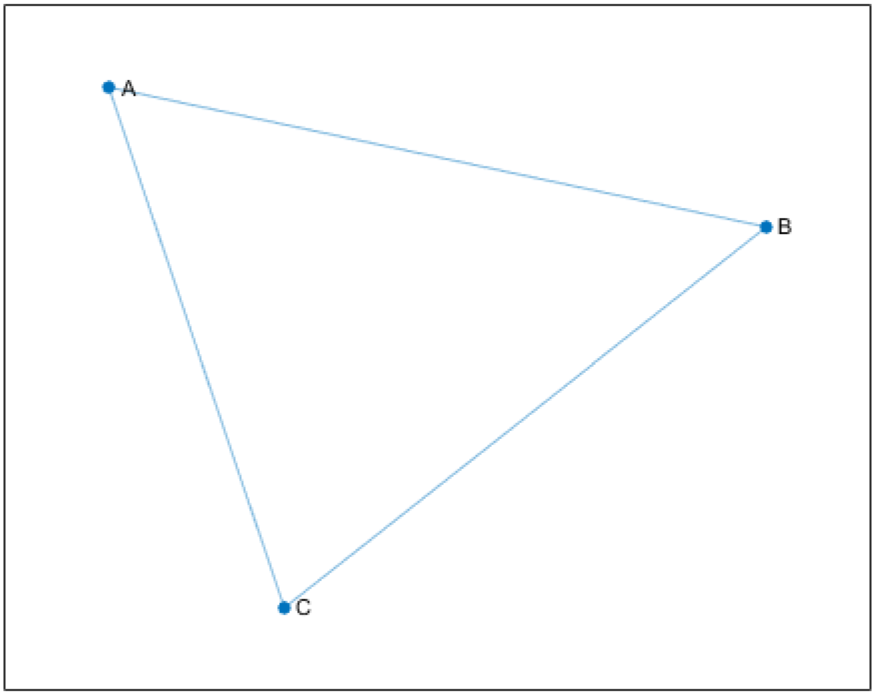
- **Nodes** are vertices that correspond to objects.
- **Edges** are the connections between objects.
- The graph edges sometimes have **Weights**, which indicate the strength (or some other attribute) of each connection between the nodes.

These definitions are general, as the exact meaning of the nodes and edges in a graph depends on the specific application. For instance, you can model the friendships in a social network using a graph. The graph nodes are people, and the edges represent friendships. The natural correspondence of graphs to physical objects and situations means that you can use graphs to model a wide variety of systems. For example:

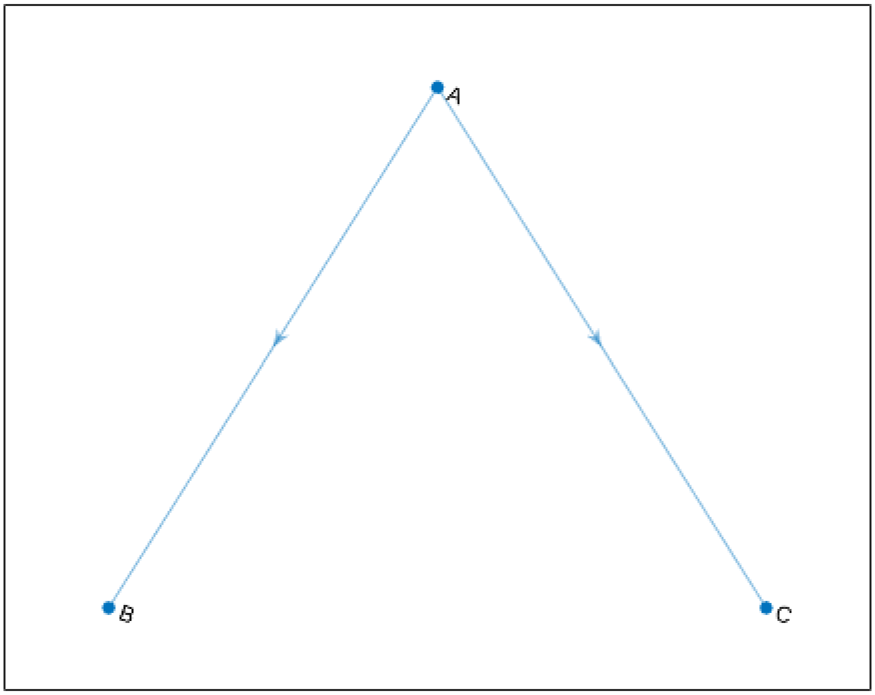
- Web page linking — The graph nodes are web pages, and the edges represent hyperlinks between pages.
- Airports — The graph nodes are airports, and the edges represent flights between airports.

In MATLAB, the `graph` and `digraph` functions construct objects that represent undirected and directed graphs.

- **Undirected graphs** have edges that do not have a direction. The edges indicate a *two-way* relationship, in that each edge can be traversed in both directions. This figure shows a simple undirected graph with three nodes and three edges.



- **Directed graphs** have edges with direction. The edges indicate a *one-way* relationship, in that each edge can only be traversed in a single direction. This figure shows a simple directed graph with three nodes and two edges.



The exact position, length, or orientation of the edges in a graph illustration typically do not have meaning. In other words, the same graph can be visualized in several different ways by rearranging the nodes and/or distorting the edges, as long as the underlying structure does not change.

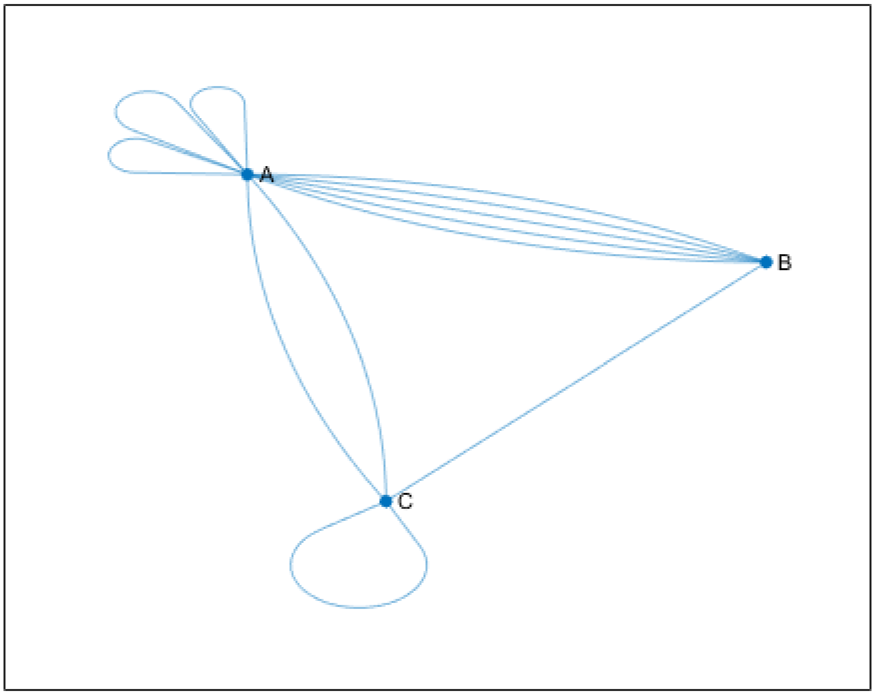
Self-loops and Multigraphs

Graphs created using `graph` and `digraph` can have one or more *self-loops*, which are edges connecting a node to itself. Additionally, graphs can have multiple edges with the same source and target nodes, and the graph is then known as a *multigraph*. A multigraph may or may not contain self-loops.

For the purposes of graph algorithm functions in MATLAB, a graph containing a node with a single self-loop is not a multigraph. However, if the graph contains a node with *multiple* self-loops, it is a multigraph.

For example, the following figure shows an undirected multigraph with self-loops. Node A has three self-loops, while node C has one. The graph contains these three conditions, any one of which makes it a multigraph.

- Node A has three self-loops.
- Nodes A and B have five edges between them.
- Nodes A and C have two edges between them.



To determine whether a given graph is a multigraph, use the `ismultigraph` function.

Creating Graphs

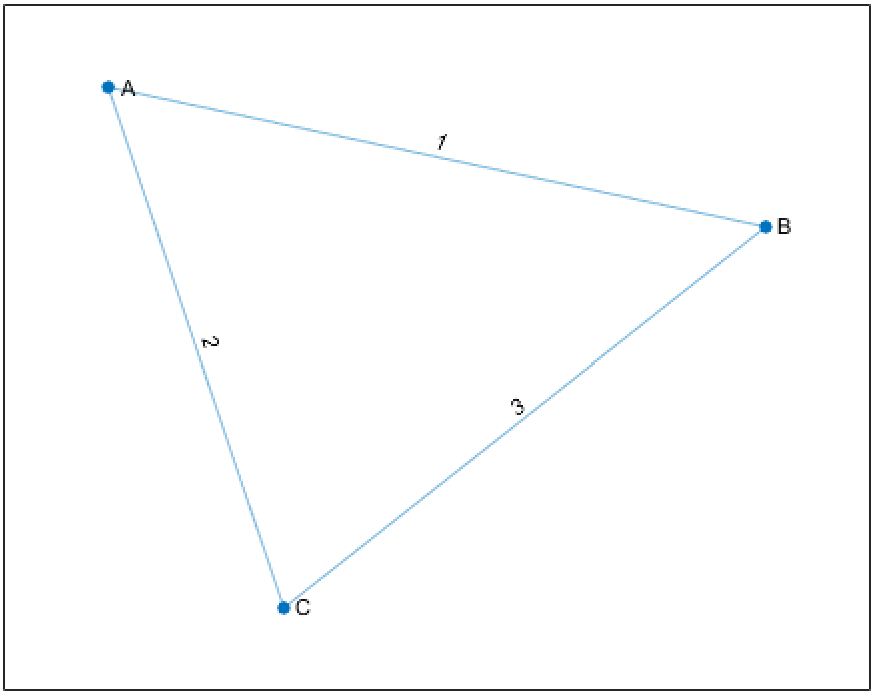
The primary ways to create a graph include using an adjacency matrix or an edge list.

Adjacency Matrix

One way to represent the information in a graph is with a square *adjacency matrix*. The nonzero entries in an adjacency matrix indicate an edge between two nodes, and the value of the entry indicates the weight of the edge. The diagonal elements of an adjacency matrix are typically zero, but a nonzero diagonal element indicates a *self-loop*, or a node that is connected to itself by an edge.

- When you use `graph` to create an undirected graph, the adjacency matrix must be symmetric. In practice, the matrices are frequently triangular to avoid repetition. To construct an undirected graph using only the upper or lower triangle of the adjacency matrix, use `graph(A,'upper')` or `graph(A,'lower')` .
- When you use `digraph` to create a directed graph, the adjacency matrix does not need to be symmetric.
- For large graphs, the adjacency matrix contains many zeros and is typically a sparse matrix.
- You cannot create a multigraph from an adjacency matrix.

For example, consider this undirected graph.



You can represent the graph with this adjacency matrix:

$$\begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 3 \\ 2 & 3 & 0 \end{pmatrix} .$$

To construct the graph in MATLAB, input:

```
A = [0 1 2; 1 0 3; 2 3 0];
node_names = {'A','B','C'};
G = graph(A,node_names)
```

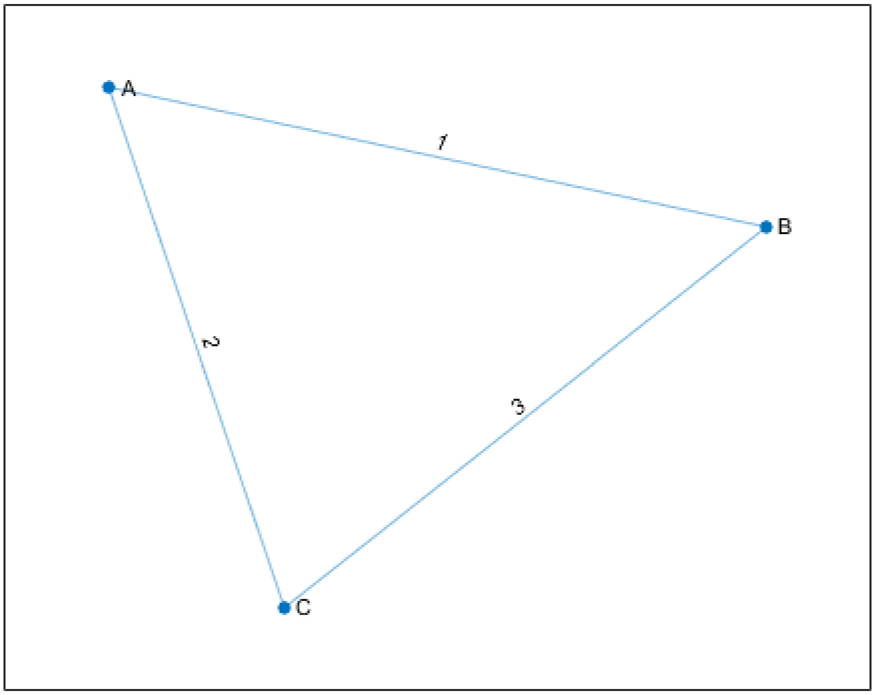
```
G = 

  graph with properties:

    Edges: [3×2 table]
    Nodes: [3×1 table]
```

You can use the `graph` or `digraph` functions to create a graph using an adjacency matrix, or you can use the `adjacency` function to find the weighted or unweighted sparse adjacency matrix of a preexisting graph.

Edge List

Another way to represent the information in a graph is by listing all of the edges.

For example, consider the same undirected graph.



Now represent the graph by the edge list

Edge	Weight
(A, B)	1
(A, C)	2
(B, C)	3

From the edge list it is easy to conclude that the graph has three unique nodes, `A`, `B`, and `C`, which are connected by the three listed edges. If the graph had disconnected nodes, they would not be found in the edge list, and would have to be specified separately.

In MATLAB, the list of edges is separated by column into *source* nodes and *target* nodes. For directed graphs the edge direction (from source to target) is important, but for undirected graphs the source and target node are interchangeable. One way to construct this graph using the edge list is to use separate inputs for the source nodes, target nodes, and edge weights:

```
source_nodes = {'A','A','B'};
target_nodes = {'B','C','C'};
edge_weights = [1 2 3];
G = graph(source_nodes, target_nodes, edge_weights);
```

Both `graph` and `digraph` permit construction of a simple graph or multigraph from an edge list. After constructing a graph, `G`, you can look at the edges (and their properties) with the command `G.Edges`. The order of the edges in `G.Edges` is sorted by source node (first column) and secondarily by target node (second column). For undirected graphs, the node with the smaller index is listed as the source node, and the node with the larger index is listed as the target node.

Since the underlying implementation of `graph` and `digraph` depends on sparse matrices, many of the same indexing costs apply. Using one of the previous methods to construct a graph all at once from the triplet pairs `(source,target,weight)` is quicker than creating an empty graph and iteratively adding more nodes and edges. For best performance, minimize the number of calls to `graph`, `digraph`, `addedge`, `addnode`, `rmedge`, and `rmnode`.

Graph Node IDs

By default, all of the nodes in a graph created using `graph` or `digraph` are numbered. Therefore, you always can refer to them by their numeric *node index*.

If the graph has node names (that is, `G.Nodes` contains a variable `Name`), then you also can refer to the nodes in a graph using their names. Thus, named nodes in a graph can be referred to by either their node indices or node names. For example, node `1` can be called, `'A'`.

The term *node ID* encompasses both aspects of node identification. The node ID refers to both the node index and the node name.

For convenience, MATLAB remembers which type of node ID you use when you call most graph functions. So if you refer to the nodes in a graph by their node indices, most graph functions return a numeric answer that also refers to the nodes by their indices.

```
A = [0 1 1 0; 1 0 1 0; 1 1 0 1; 0 0 1 0];
G = graph(A,{'a','b','c','d'});
p = shortestpath(G,1,4)
```

```
p =

     1     3     4
```

However, if you refer to the nodes by their names, then most graph functions return an answer that also refers to the nodes by their names (contained in a cell array of character vectors or string array).

```
p1 = shortestpath(G,'a','d')
```

```
p1 =

  1×3 cell array

    {'a'}    {'c'}    {'d'}
```

Use `findnode` to find the numeric node ID for a given node name. Conversely, for a given numeric node ID, index into `G.Nodes.Name` to determine the corresponding node name.

Modify or Query Existing Graph

After you construct a `graph` or `digraph` object, you can use a variety of functions to modify the graph structure or to determine how many nodes or edges the graph has. This table lists some available functions for modifying or querying `graph` and `digraph` objects.

`addedge`	Add one or more edges to a graph

`rmedge`	Remove one or more edges from a graph
`addnode`	Add one or more nodes to a graph
`rmnode`	Remove one or more nodes from a graph
`findnode`	Locate a specific node in a graph
`findedge`	Locate a specific edge in a graph
`numnodes`	Find the number of nodes in a graph
`numedges`	Find the number of edges in a graph
`edgecount`	Number of edges between specified nodes
`flipedge`	Reverse the direction of directed graph edges
`reordernodes`	Permute the order of the nodes in a graph
`subgraph`	Extract subgraph

See “Modify Nodes and Edges of Existing Graph” on page 5-10 for some common graph modification examples.

See Also

`digraph` | `graph`

More About

- “Modify Nodes and Edges of Existing Graph” on page 5-10
- “Add Graph Node Names, Edge Weights, and Other Attributes” on page 5-13
- “Graph Plotting and Customization” on page 5-17

Modify Nodes and Edges of Existing Graph

This example shows how to access and modify the nodes and/or edges in a `graph` or `digraph` object using the `addedge`, `rmedge`, `addnode`, `rmnode`, `findedge`, `findnode`, and `subgraph` functions.

Add Nodes

Create a graph with four nodes and four edges. The corresponding elements in `s` and `t` specify the end nodes of each graph edge.

```
s = [1 1 1 2];
t = [2 3 4 3];
G = graph(s,t)
```

```
G = 
  graph with properties:

    Edges: [4x1 table]
    Nodes: [4x0 table]
```

View the edge list of the graph.

```
G.Edges
```

```
ans=4×1 table
    EndNodes
    ________

      1    2
      1    3
      1    4
      2    3
```

Use `addnode` to add five nodes to the graph. This command adds five disconnected nodes with node IDs `5`, `6`, `7`, `8`, and `9`.

```
G = addnode(G,5)
```

```
G = 
  graph with properties:

    Edges: [4x1 table]
    Nodes: [9x0 table]
```

Remove Nodes

Use `rmnode` to remove nodes 3, 5, and 6 from the graph. All edges connected to one of the removed nodes also are removed. The remaining six nodes in the graph are renumbered to reflect the new number of nodes.

```
G = rmnode(G,[3 5 6])
```

```
G = 
  graph with properties:
    Edges: [2x1 table]
    Nodes: [6x0 table]
```

Add Edges

Use `addedge` to add two edges to `G`. The first edge is between node 1 and node 5, and the second edge is between node 2 and node 5. This command adds two new rows to `G.Edges`.

```
G = addedge(G,[1 2],[5 5])
```

```
G = 
  graph with properties:

    Edges: [4x1 table]
    Nodes: [6x0 table]
```

Remove Edges

Use `rmedge` to remove the edge between node 1 and node 3. This command removes a row from `G.Edges`.

```
G = rmedge(G,1,3)
```

```
G = 
  graph with properties:

    Edges: [3x1 table]
    Nodes: [6x0 table]
```

Determine Edge Index

Determine the edge index for the edge between nodes 1 and 5. The edge index, `ei`, is a row number in `G.Edges`.

```
ei = findedge(G,1,5)
```

```
ei = 2
```

Determine Node Index

Add node names to the graph, and then determine the node index for node `'d'`. The numeric node index, `ni`, is a row number in `G.Nodes`. You can use both `ni` and the node name, `'d'`, to refer to the node when using other graph functions, like `shortestpath`.

```
G.Nodes.Name = {'a' 'b' 'c' 'd' 'e' 'f'}';
ni = findnode(G,'d')
```

```
ni = 4
```

Extract Subgraph

Use `subgraph` to extract a piece of the graph containing only two nodes.

```
H = subgraph(G,[1 2])
```

```
H = 
  graph with properties:
    Edges: [1x1 table]
    Nodes: [2x1 table]
```

View the edge list of the subgraph.

```
H.Edges
```

```
ans=table
       EndNodes
    ______________

    {'a'}    {'b'}
```

Modify Node and Edge Tables with Variables Editor

The node and edge information for a graph object is contained in two properties: `Nodes` and `Edges`. Both of these properties are tables containing variables to describe the attributes of the nodes and edges in the graph. Since `Nodes` and `Edges` are both tables, you can use the Variables editor to interactively view or edit the tables. You cannot add or remove nodes or edges using the Variables editor, and you also cannot edit the `EndNodes` property of the `Edges` table. The Variables editor is useful for managing extra node and edge attributes in the `Nodes` and `Edges` tables. For more information, see “Create and Edit Variables”.

See Also

`addedge` | `addnode` | `digraph` | `findedge` | `findnode` | `graph` | `rmedge` | `rmnode` | `subgraph`

More About

- “Directed and Undirected Graphs” on page 5-2

Add Graph Node Names, Edge Weights, and Other Attributes

This example shows how to add attributes to the nodes and edges in graphs created using `graph` and `digraph`. You can specify node names or edge weights when you originally call `graph` or `digraph` to create a graph. However, this example shows how to add attributes to a graph after it has been created.

Create Graph

Create a directed graph. The corresponding elements in `s` and `t` define the source and target nodes of each edge in the graph.

```
s = [1 1 2 2 3];
t = [2 4 3 4 4];
G = digraph(s,t)
```

```
G = 
  digraph with properties:

    Edges: [5x1 table]
    Nodes: [4x0 table]
```

Add Node Names

Add node names to the graph by adding the variable, `Name`, to the `G.Nodes` table. The `Name` variable must be specified as an `N`-by-`1` cell array of character vectors or string array, where `N = numnodes(G)`. It is important to use the `Name` variable when adding node names, as this variable name is treated specially by some graph functions.

```
G.Nodes.Name = {'First' 'Second' 'Third' 'Fourth'}';
```

View the new `Nodes` table.

```
G.Nodes
```

```
ans=4×1 table
       Name   
    __________

    {'First' }
    {'Second'}
    {'Third' }
    {'Fourth'}
```

Use table indexing to view the names of nodes 1 and 4.

```
G.Nodes.Name([1 4])
```

```
ans = 2x1 cell
    {'First' }
    {'Fourth'}
```

Add Edge Weights

Add edge weights to the graph by adding the variable, `Weight`, to the `G.Edges` table. The `Weight` variable must be an `M`-by-1 numeric vector, where `M = numedges(G)`. It is important to use the

Weight variable when adding edge weights, as this variable name is treated specially by some graph functions.

```
G.Edges.Weight = [10 20 30 40 50]';
```

View the new Edges table.

```
G.Edges
```

```
ans=5×2 table
            EndNodes             Weight
    ________________________    ______

    {'First' }    {'Second'}      10
    {'First' }    {'Fourth'}      20
    {'Second'}    {'Third' }      30
    {'Second'}    {'Fourth'}      40
    {'Third' }    {'Fourth'}      50
```

Use table indexing to view the first and third rows of G.Edges.

```
G.Edges([1 3],:)
```

```
ans=2×2 table
            EndNodes             Weight
    ________________________    ______

    {'First' }    {'Second'}      10
    {'Second'}    {'Third' }      30
```

Add Custom Attributes

In principle you can add any variable to G.Nodes and G.Edges that defines an attribute of the graph nodes or edges. Adding custom attributes can be useful, since functions like subgraph and reordernodes preserve the graph attributes.

For example, add a variable named Power to G.Edges to indicate whether each edge is 'on' or 'off'.

```
G.Edges.Power = {'on' 'on' 'on' 'off' 'off'}';
G.Edges
```

```
ans=5×3 table
            EndNodes             Weight     Power
    ________________________    ______    _______

    {'First' }    {'Second'}      10      {'on' }
    {'First' }    {'Fourth'}      20      {'on' }
    {'Second'}    {'Third' }      30      {'on' }
    {'Second'}    {'Fourth'}      40      {'off'}
    {'Third' }    {'Fourth'}      50      {'off'}
```

Add a variable named Size to G.Nodes to indicate the physical size of each node.

```
G.Nodes.Size = [10 20 10 30]';
G.Nodes
```

```
ans=4×2 table
        Name        Size
    ____________    ____

    {'First' }       10 
    {'Second'}       20 
    {'Third' }       10 
    {'Fourth'}       30 
```

Modify Tables with Variables Editor

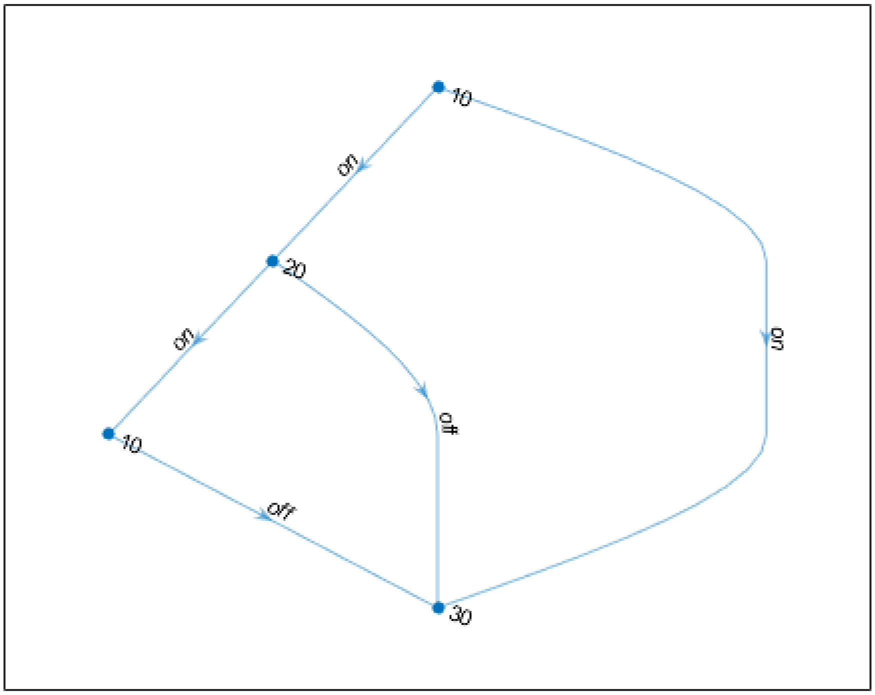
Since `Nodes` and `Edges` are both tables, you can use the Variables editor to interactively view or edit the tables. For more information, see “Create and Edit Variables”.

Label Nodes and Edges of Graph Plot

When you plot a graph, you can use the variables in `G.Nodes` and `G.Edges` to label the graph nodes and edges. This practice is convenient, since these variables are already guaranteed to have the correct number of elements.

Plot the graph and label the edges using the `Power` variable in `G.Edges`. Label the nodes using the `Size` variable in `G.Nodes`.

```
p = plot(G,'EdgeLabel',G.Edges.Power,'NodeLabel',G.Nodes.Size)
```



```
p = 
  GraphPlot with properties:
     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
        Marker: 'o'
     EdgeColor: [0 0.4470 0.7410]
     LineWidth: 0.5000
     LineStyle: '-'
     NodeLabel: {'10'  '20'  '10'  '30'}
     EdgeLabel: {'on'  'on'  'on'  'off'  'off'}
         XData: [2 1.5000 1 2]
         YData: [4 3 2 1]
         ZData: [0 0 0 0]

  Show all properties
```

See Also

`digraph` | `graph`

More About

- “Directed and Undirected Graphs” on page 5-2
- “Modify Nodes and Edges of Existing Graph” on page 5-10

Graph Plotting and Customization

This example shows how to plot graphs, and then customize the display to add labels or highlighting to the graph nodes and edges.

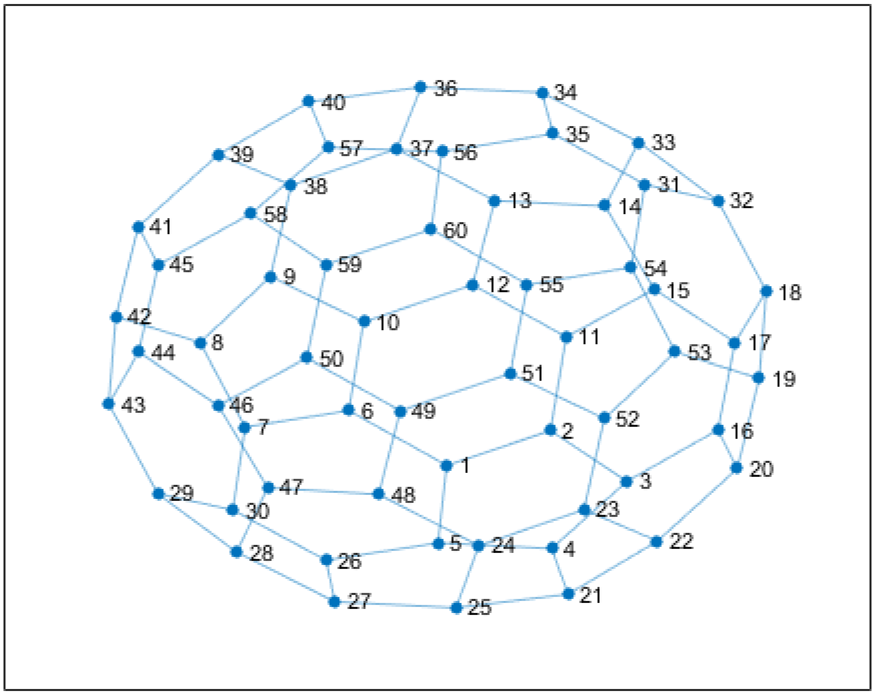
Graph Plotting Objects

Use the `plot` function to plot `graph` and `digraph` objects. By default, `plot` examines the size and type of graph to determine which layout to use. The resulting figure window contains no axes tick marks. However, if you specify the *(x,y)* coordinates of the nodes with the `XData`, `YData`, or `ZData` name-value pairs, then the figure includes axes ticks.

Node labels are included automatically in plots of graphs that have 100 or fewer nodes. The node labels use the node names if available; otherwise, the labels are numeric node indices.

For example, create a graph using the buckyball adjacency matrix, and then plot the graph using all of the default options. If you call `plot` and specify an output argument, then the function returns a handle to a `GraphPlot` object. Subsequently, you can use this object to adjust properties of the plot. For example, you can change the color or style of the edges, the size and color of the nodes, and so on.

```
G = graph(bucky);
p = plot(G)
```

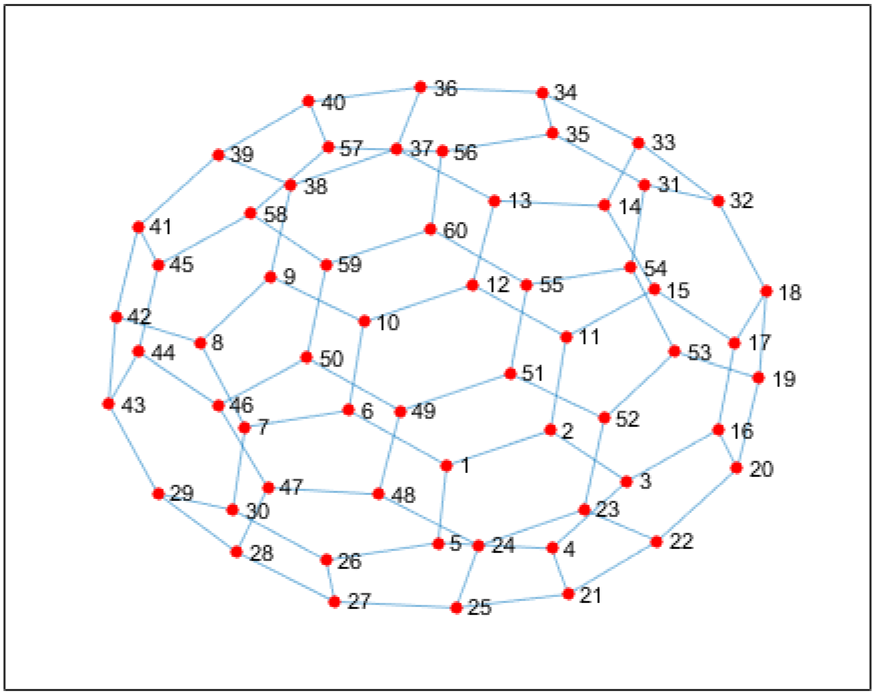


```
p =
  GraphPlot with properties:
     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
        Marker: 'o'
     EdgeColor: [0 0.4470 0.7410]
     LineWidth: 0.5000
     LineStyle: '-'
     NodeLabel: {1x60 cell}
     EdgeLabel: {}
         XData: [1x60 double]
         YData: [1x60 double]
         ZData: [1x60 double]

  Show all properties
```

After you have a handle to the GraphPlot object, use dot indexing to access or change the property values. For a complete list of the properties that you can adjust, see GraphPlot Properties.

Change the value of NodeColor to 'red'.

```
p.NodeColor = 'red';
```



Determine the line width of the edges.

```
p.LineWidth
```

```
ans = 0.5000
```

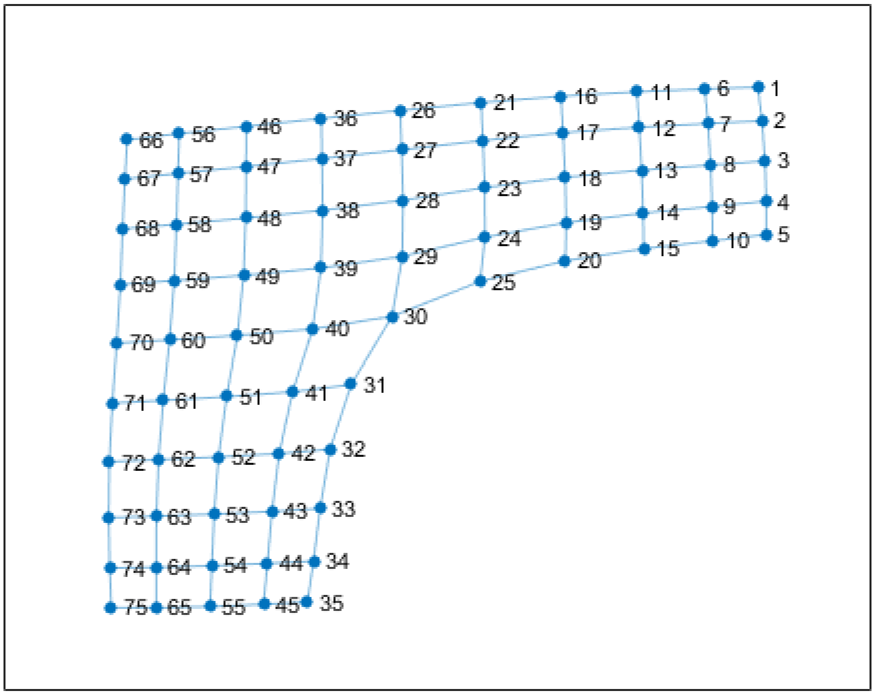
Create and Plot Graph

Create and plot a graph representing an L-shaped membrane constructed from a square grid with a side of 12 nodes. Specify an output argument with `plot` to return a handle to the `GraphPlot` object.

```
n = 12;
A = delsq(numgrid('L',n));
G = graph(A,'omitselfloops')
```

```
G = 
  graph with properties:

    Edges: [130x2 table]
    Nodes: [75x0 table]
```

```
p = plot(G)
```



```
p = 
  GraphPlot with properties:

     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
        Marker: 'o'
     EdgeColor: [0 0.4470 0.7410]
     LineWidth: 0.5000
     LineStyle: '-'
     NodeLabel: {1x75 cell}
```

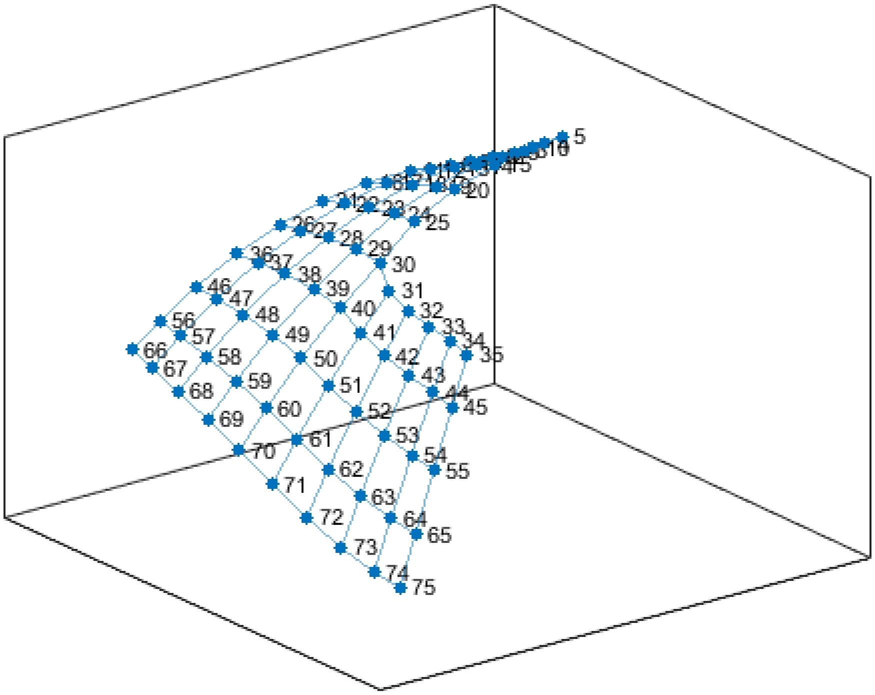
```
      EdgeLabel: {}
          XData: [1x75 double]
          YData: [1x75 double]
          ZData: [1x75 double]

  Show all properties
```

Change Graph Node Layout

Use the `layout` function to change the layout of the graph nodes in the plot. The different layout options automatically compute node coordinates for the plot. Alternatively, you can specify your own node coordinates with the `XData`, `YData`, and `ZData` properties of the `GraphPlot` object.

Instead of using the default 2-D layout method, use `layout` to specify the `'force3'` layout, which is a 3-D force directed layout.

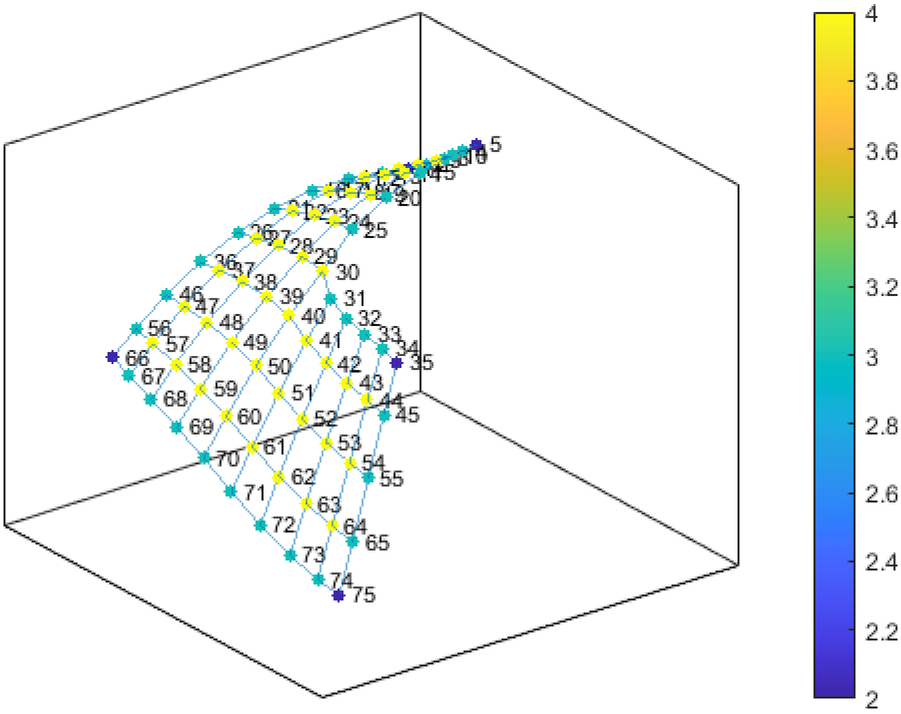
```
layout(p,'force3')
view(3)
```



Proportional Node Coloring

Color the graph nodes based on their degree. In this graph, all of the interior nodes have the same maximum degree of 4, nodes along the boundary of the graph have a degree of 3, and the corner nodes have the smallest degree of 2. Store this node coloring data as the variable `NodeColors` in `G.Nodes`.

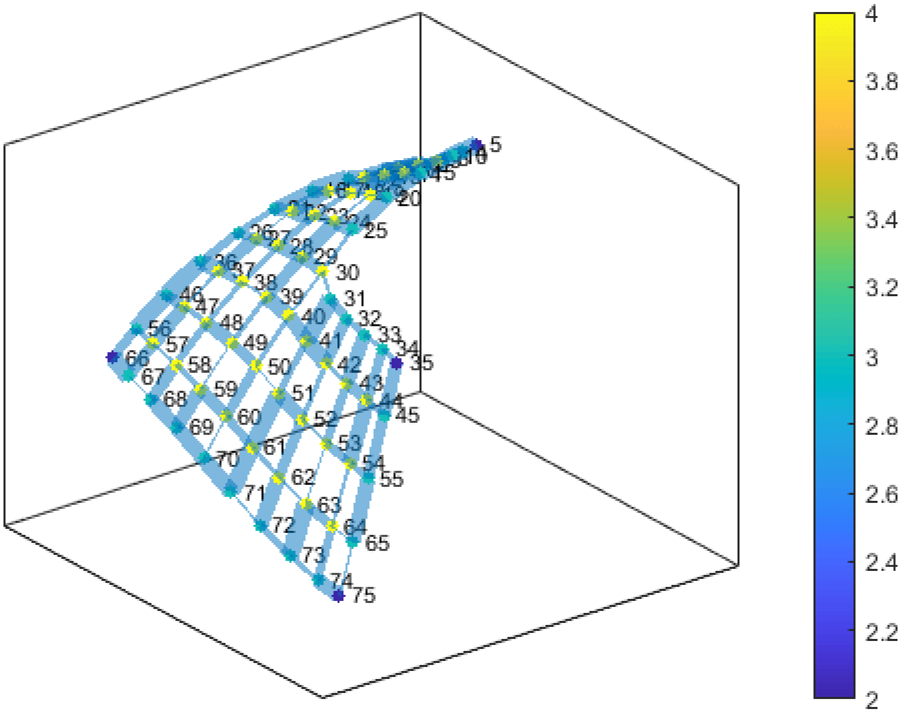
```
G.Nodes.NodeColors = degree(G);
p.NodeCData = G.Nodes.NodeColors;
colorbar
```



Edge Line Width by Weight

Add some random integer weights to the graph edges, and then plot the edges such that their line width is proportional to their weight. Since an edge line width approximately greater than 7 starts to become cumbersome, scale the line widths such that the edge with the greatest weight has a line width of 7. Store this edge width data as the variable `LWidths` in `G.Edges`.

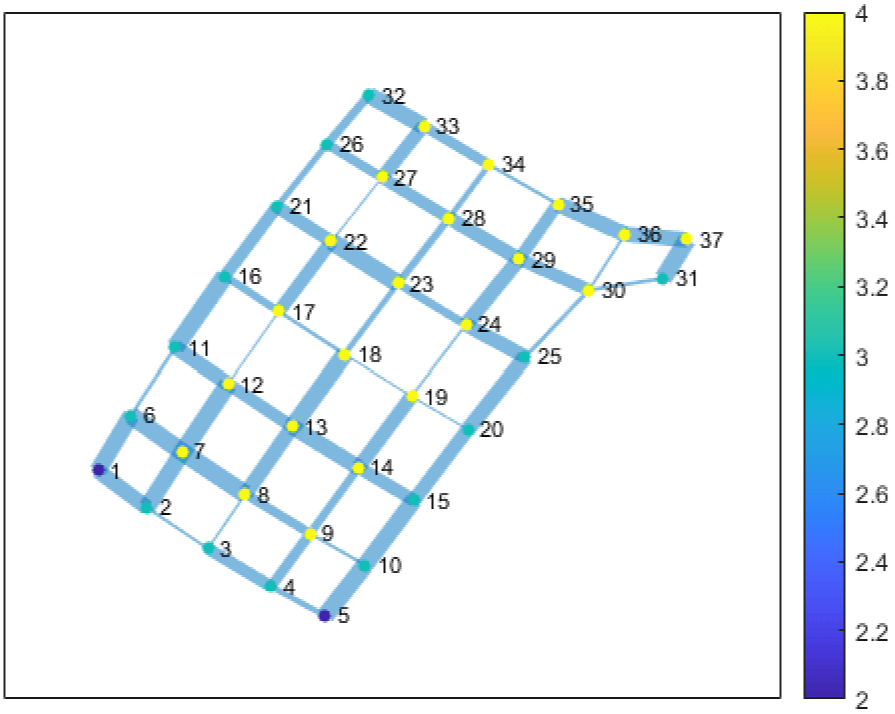
```
G.Edges.Weight = randi([10 250],130,1);
G.Edges.LWidths = 7*G.Edges.Weight/max(G.Edges.Weight);
p.LineWidth = G.Edges.LWidths;
```



Extract Subgraph

Extract and plot the top right corner of G as a subgraph, to make it easier to read the details on the graph. The new graph, H, inherits the NodeColors and LWidths variables from G, so that recreating the previous plot customizations is straightforward. However, the nodes in H are renumbered to account for the new number of nodes in the graph.

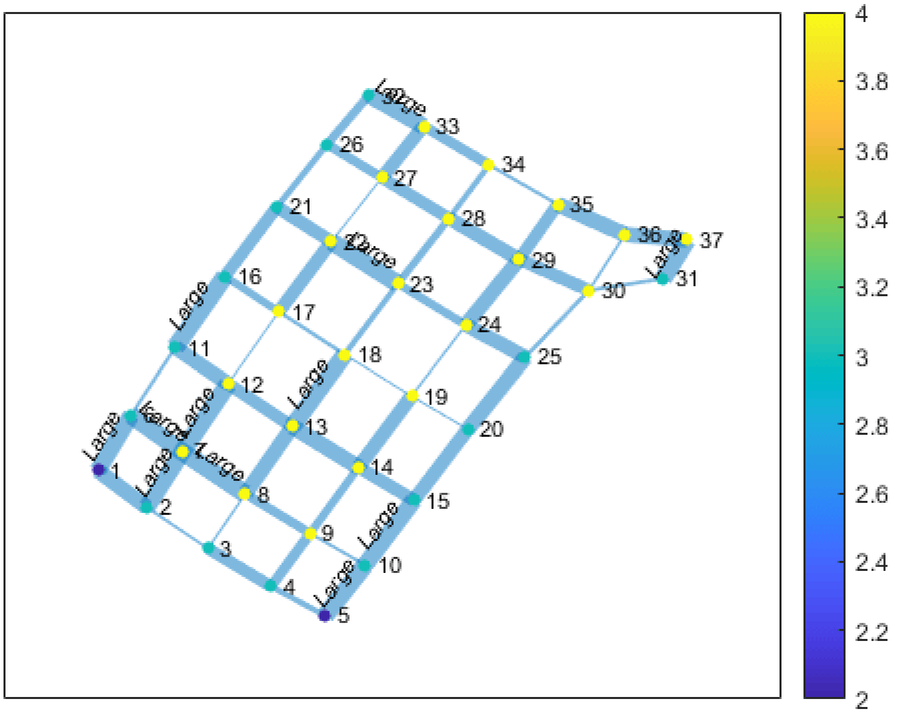
```
H = subgraph(G,[1:31 36:41]);
p1 = plot(H,'NodeCData',H.Nodes.NodeColors,'LineWidth',H.Edges.LWidths);
colorbar
```



Label Nodes and Edges

Use `labeledge` to label the edges whose width is larger than `6` with the label, `'Large'`. The `labelnode` function works in a similar manner for labeling nodes.

```
labeledge(p1,find(H.Edges.LWidths > 6),'Large')
```



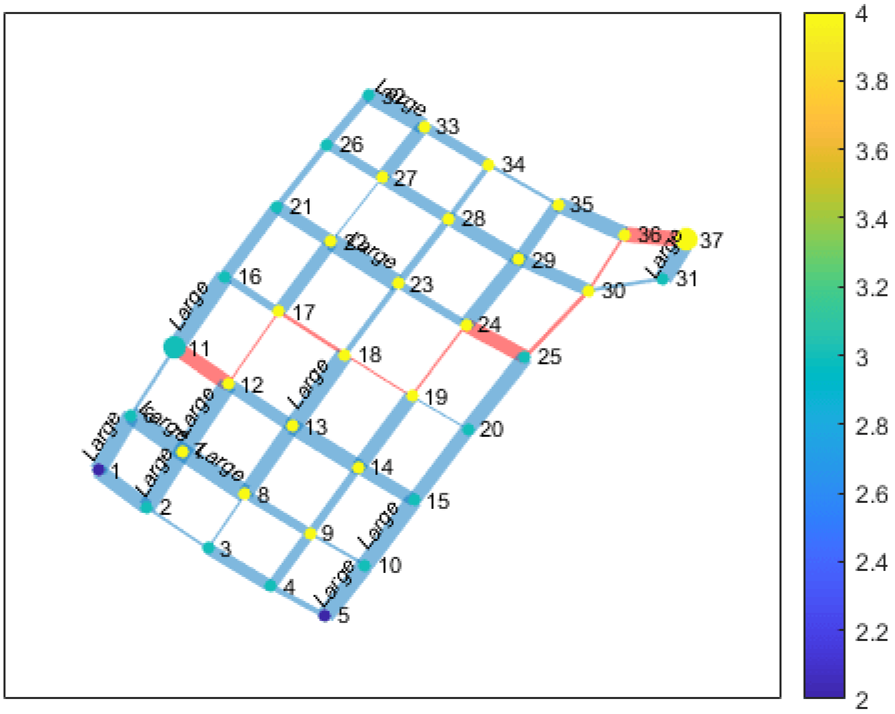
Highlight Shortest Path

Find the shortest path between node 11 and node 37 in the subgraph, H. Highlight the edges along this path in red, and increase the size of the end nodes on the path.

```
path = shortestpath(H,11,37)
```

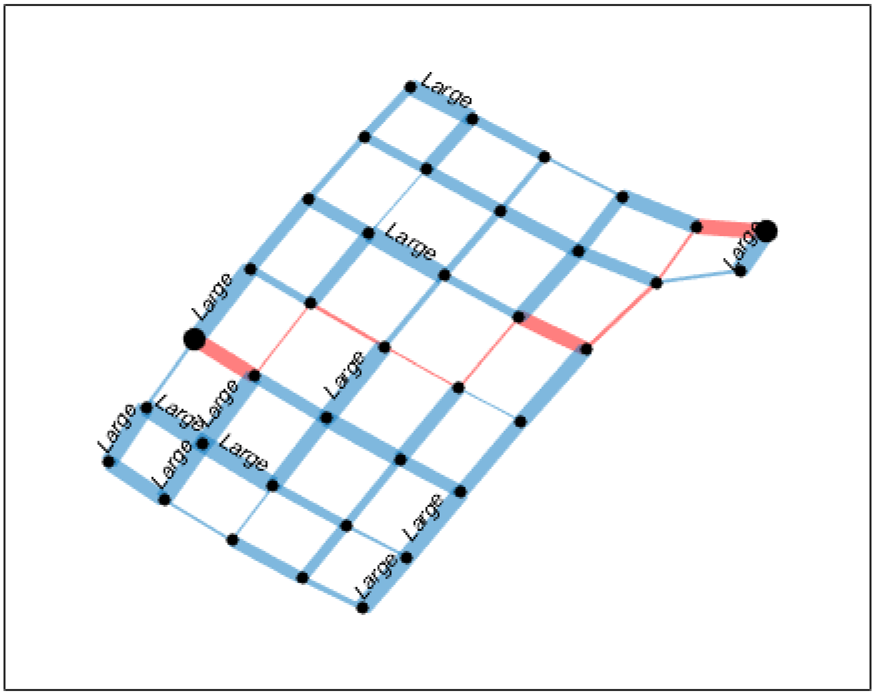
```
path = 1×10

    11    12    17    18    19    24    25    30    36    37
```

```
highlight(p1,[11 37])
highlight(p1,path,'EdgeColor','r')
```



Remove the node labels and colorbar, and make all of the nodes black.

```
p1.NodeLabel = {};
colorbar off
p1.NodeColor = 'black';
```

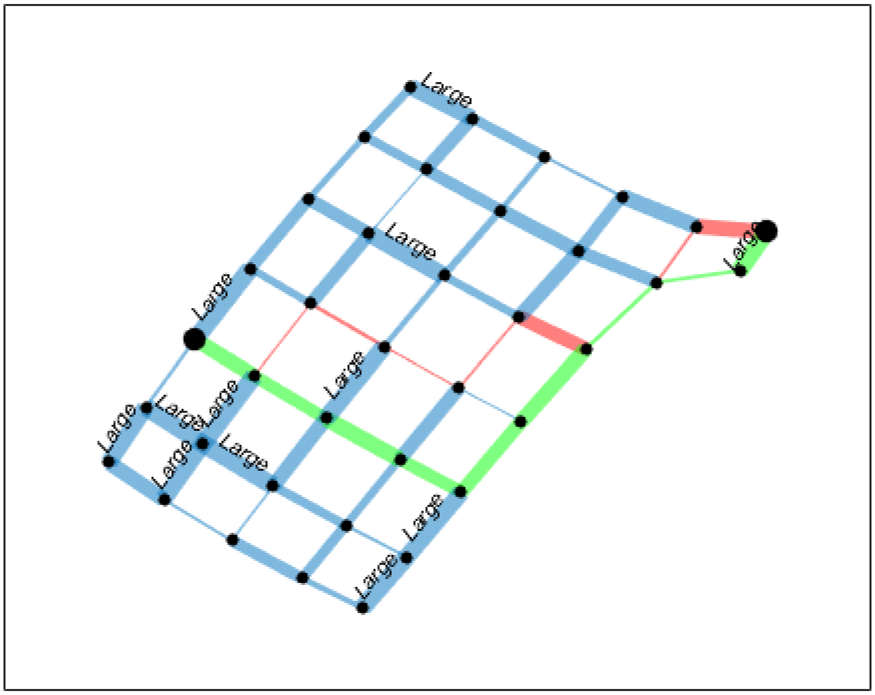


Find a different shortest path that ignores the edge weights. Highlight this path in green.

```
path2 = shortestpath(H,11,37,'Method','unweighted')
```

```
path2 = 1×10

    11    12    13    14    15    20    25    30    31    37
```

```
highlight(p1,path2,'EdgeColor','g')
```



Plotting Large Graphs

It is common to create graphs that have hundreds of thousands, or even millions, of nodes and/or edges. For this reason, `plot` treats large graphs slightly differently to maintain readability and performance. The `plot` function makes these adjustments when working with graphs that have more than 100 nodes:

1. The default graph layout method is always `'subspace'`.
2. The nodes are no longer labeled automatically.
3. The `MarkerSize` property is set to `2`. (Smaller graphs have a marker size of `4`).
4. The `ArrowSize` property of directed graphs is set to `4`. (Smaller directed graphs use an arrow size of `7`).

See Also

`GraphPlot` | `digraph` | `graph` | `plot`

More About

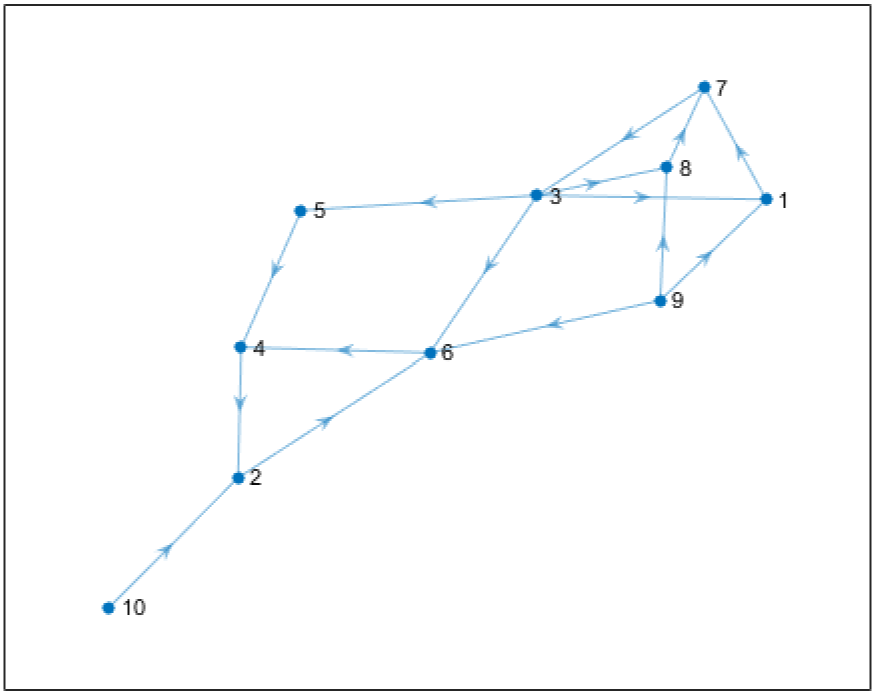
- “Directed and Undirected Graphs” on page 5-2
- GraphPlot
- “Add Node Properties to Graph Plot Data Tips” on page 5-36

Visualize Breadth-First and Depth-First Search

This example shows how to define a function that visualizes the results of `bfsearch` and `dfsearch` by highlighting the nodes and edges of a graph.

Create and plot a directed graph.

```
s = [1 2 3 3 3 3 4 5 6 7 8 9 9 9 10];
t = [7 6 1 5 6 8 2 4 4 3 7 1 6 8 2];
G = digraph(s,t);
plot(G)
```



Perform a depth-first search on the graph. Specify `'allevents'` to return all events in the algorithm. Also, specify `Restart` as `true` to ensure that the search visits every node in the graph.

```
T = dfsearch(G, 1, 'allevents', 'Restart', true)
```

```
T = 

  38x4 table

        Event          Node       Edge        EdgeIndex
    ________________   ____    __________    _________

    startnode            1     NaN    NaN       NaN   
    discovernode         1     NaN    NaN       NaN   
    edgetonew           NaN       1      7         1
    discovernode          7     NaN    NaN       NaN
    edgetonew           NaN       7      3        10
    discovernode          3     NaN    NaN       NaN
    edgetodiscovered    NaN       3      1         3
    edgetonew           NaN       3      5         4
    discovernode          5     NaN    NaN       NaN
    edgetonew           NaN       5      4         8
    discovernode          4     NaN    NaN       NaN
    edgetonew           NaN       4      2         7
    discovernode          2     NaN    NaN       NaN
    edgetonew           NaN       2      6         2
    discovernode          6     NaN    NaN       NaN
    edgetodiscovered    NaN       6      4         9
    finishnode            6     NaN    NaN       NaN
    finishnode            2     NaN    NaN       NaN
    finishnode            4     NaN    NaN       NaN
    finishnode            5     NaN    NaN       NaN
    edgetofinished      NaN       3      6         5
    edgetonew           NaN       3      8         6
    discovernode          8     NaN    NaN       NaN
    edgetodiscovered    NaN       8      7        11
    finishnode            8     NaN    NaN       NaN
    finishnode            3     NaN    NaN       NaN
    finishnode            7     NaN    NaN       NaN
    finishnode            1     NaN    NaN       NaN
    startnode             9     NaN    NaN       NaN
    discovernode          9     NaN    NaN       NaN
    edgetofinished      NaN       9      1        12
    edgetofinished      NaN       9      6        13
    edgetofinished      NaN       9      8        14
    finishnode            9     NaN    NaN       NaN
    startnode            10     NaN    NaN       NaN
    discovernode         10     NaN    NaN       NaN
    edgetofinished      NaN      10      2        15
    finishnode           10     NaN    NaN       NaN
```

The values in the table, T, are useful for visualizing the search. The function visualize_search.m shows one way to use the results of searches performed with bfsearch and dfsearch to highlight the nodes and edges in the graph according to the table of events, T. The function pauses before each step in the algorithm, so you can slowly step through the search by pressing any key.

Save visualize_search.m in the current folder.

```
function visualize_search(G,t)
% G is a graph or digraph object, and t is a table resulting from a call to
% BFSEARCH or DFSEARCH on that graph.
%
% Example inputs: G = digraph([1 2 3 3 3 3 4 5 6 7 8 9 9 9 10], ...
%     [7 6 1 5 6 8 2 4 4 3 7 1 6 8 2]);
% t = dfsearch(G, 1, 'allevents', 'Restart', true);

% Copyright 1984-2019 The MathWorks, Inc.

isundirected = isa(G, 'graph');
if isundirected
```

```
    % Replace graph with corresponding digraph, because we need separate
    % edges for both directions
    [src, tgt] = findedge(G);
    G = digraph([src; tgt], [tgt; src], [1:numedges(G), 1:numedges(G)]);
end

h = plot(G,'NodeColor',[0.5 0.5 0.5],'EdgeColor',[0.5 0.5 0.5], ...
    'EdgeLabelMode', 'auto');

for ii=1:size(t,1)
    switch t.Event(ii)
        case 'startnode'
            highlight(h,t.Node(ii),'MarkerSize',min(h.MarkerSize)*2);
        case 'discovernode'
            highlight(h,t.Node(ii),'NodeColor','r');
        case 'finishnode'
            highlight(h,t.Node(ii),'NodeColor','k');
        otherwise
            if isundirected
                a = G.Edges.Weight;
                b = t.EdgeIndex(ii);
                edgeind = intersect(find(a == b),...
                    findedge(G,t.Edge(ii,1),t.Edge(ii,2)));
            else
                edgeind = t.EdgeIndex(ii);
            end
            switch t.Event(ii)
                case 'edgetonew'
                    highlight(h,'Edges',edgeind,'EdgeColor','b');
                case 'edgetodiscovered'
                    highlight(h,'Edges',edgeind,'EdgeColor',[0.8 0 0.8]);
                case 'edgetofinished'
                    highlight(h,'Edges',edgeind,'EdgeColor',[0 0.8 0]);
            end
    end

    nodeStr = t.Node;
    if isnumeric(nodeStr)
        nodeStr = num2cell(nodeStr);
        nodeStr = cellfun(@num2str, nodeStr, 'UniformOutput', false);
    end

    edgeStr = t.Edge;
    if isnumeric(edgeStr)
        edgeStr = num2cell(edgeStr);
        edgeStr = cellfun(@num2str, edgeStr, 'UniformOutput', false);
    end

    if ~isnan(t.Node(ii))
        title([char(t{ii, 1}) ' on Node ' nodeStr{ii}]);
    else
        title([char(t{ii, 1}) ' on Edge (' edgeStr{ii, 1} ', '...
            edgeStr{ii, 2},') with edge index ' sprintf('%d ', t{ii, 4})]);
    end

    disp('Strike any key to continue...')
    pause
end
```

```
disp('Done.')
close all
```

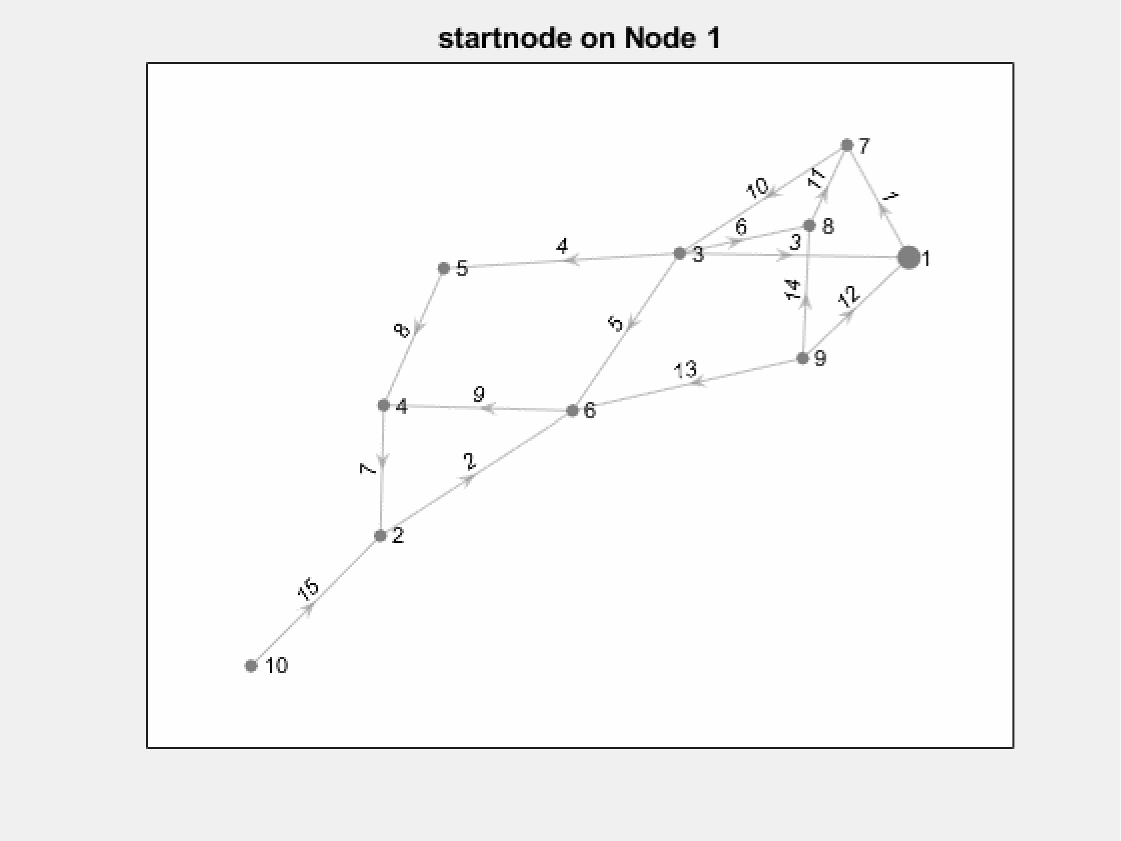
Use this command to run `visualize_search.m` on graph `G` and search result `T`:

```
visualize_search(G,T)
```

The graph begins as all gray, and then a new piece of the search result appears each time you press a key. The search results are highlighted according to:

- `'startnode'` - Starting nodes *increase* in size.
- `'discovernode'` - Nodes turn *red* as they are discovered.
- `'finishnode'` - Nodes turn *black* after they are finished.
- `'edgetonew'` - Edges that lead to undiscovered nodes turn *blue*.
- `'edgetodiscovered'` - Edges that lead to discovered nodes turn *magenta*.
- `'edgetofinished'` - Edges that lead to finished nodes turn *green*.

This `.gif` animation shows what you see when you step through the results of `visualize_search.m`.



See Also

`bfsearch` | `dfsearch` | `digraph` | `graph`

More About

- “Directed and Undirected Graphs” on page 5-2

Partition Graph with Laplacian Matrix

This example shows how to use the Laplacian matrix of a graph to compute the Fiedler vector. The Fiedler vector can be used to partition the graph into two subgraphs.

Load Data

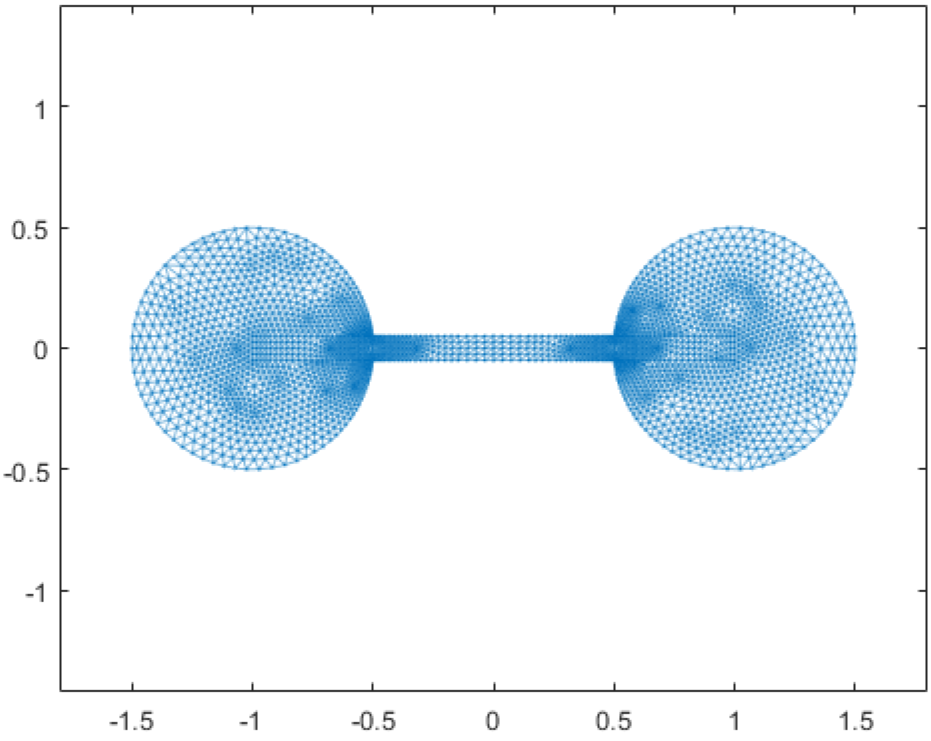
Load the data set `barbellgraph.mat`, which contains the sparse adjacency matrix and node coordinates of a barbell graph.

```
load barbellgraph.mat
```

Plot Graph

Plot the graph using the custom node coordinates `xy`.

```
G = graph(A,'omitselfloops');
p = plot(G,'XData',xy(:,1),'YData',xy(:,2),'Marker','.');
axis equal
```



Calculate Laplacian Matrix and Fiedler Vector

Calculate the Laplacian matrix of the graph. Then, calculate the two smallest magnitude eigenvalues and corresponding eigenvectors using `eigs`.

```
L = laplacian(G);
[V,D] = eigs(L,2,'smallestabs');
```

The Fiedler vector is the eigenvector corresponding to the second smallest eigenvalue of the graph. The *smallest* eigenvalue is zero, indicating that the graph has one connected component. In this case, the second column in V corresponds to the second smallest eigenvalue D(2,2).

```
D
```

```
D = 2×2
10^-3 ×

    0.0000         0
         0    0.2873
```

```
w = V(:,2);
```

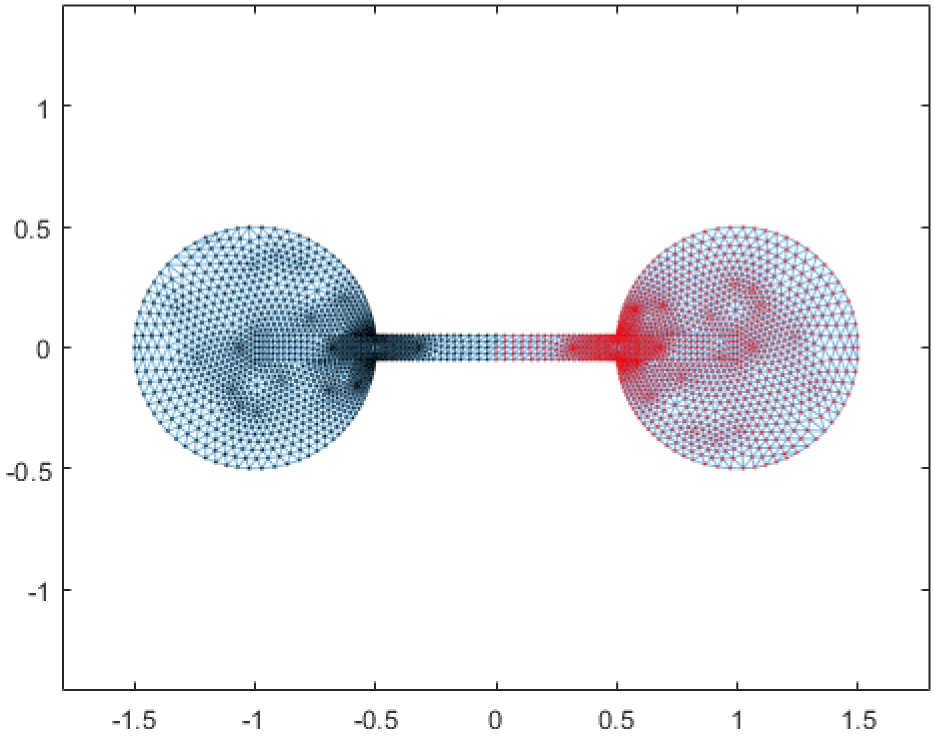
Finding the Fiedler vector using eigs is scalable to larger graphs, since only a subset of the eigenvalues and eigenvectors are computed, but for smaller graphs it is equally feasible to convert the Laplacian matrix to full storage and use eig(full(L)).

Partition Graph

Partition the graph into two subgraphs using the Fiedler vector w. A node is assigned to subgraph A if it has a positive value in w. Otherwise, the node is assigned to subgraph B. This practice is called a *sign cut* or *zero threshold cut*. The sign cut minimizes the weight of the cut, subject to the upper and lower bounds on the weight of any nontrivial cut of the graph.

Partition the graph using the sign cut. Highlight the subgraph of nodes with w>=0 in red, and the nodes with w<0 in black.

```
highlight(p,find(w>=0),'NodeColor','r') % subgraph A
highlight(p,find(w<0),'NodeColor','k') % subgraph B
```



For the bar bell graph, this partition bisects the graph nicely into two equal sets of nodes. However, the sign cut does not always produce a balanced cut.

It is always possible to bisect a graph by calculating the median of `w` and using it as a threshold value. This partition is called the *median cut*, and it guarantees an equal number of nodes in each subgraph.

You can use the median cut by first shifting the values in `w` by the median:

```
w_med = w - median(w);
```

Then, partition the graph by sign in `w_med`. For the bar bell graph, the median of `w` is close to zero, so the two cuts produce similar bisections.

See Also

`digraph` | `graph` | `laplacian` | `subgraph`

More About

- "Directed and Undirected Graphs" on page 5-2

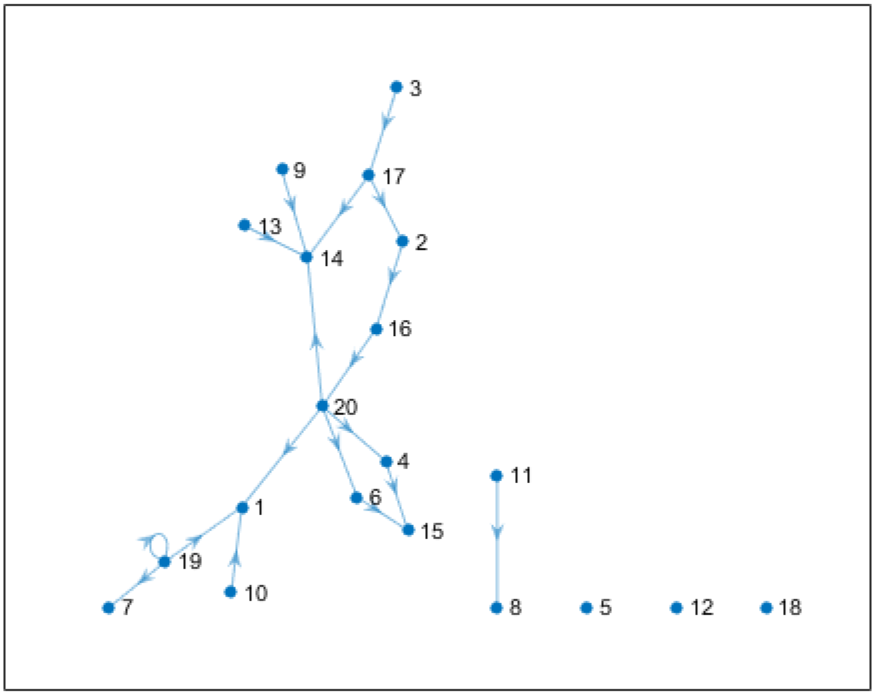
Add Node Properties to Graph Plot Data Tips

This example shows how to customize `GraphPlot` data tips to display extra node properties of a graph.

Plot `GraphPlot` object with Data Tip

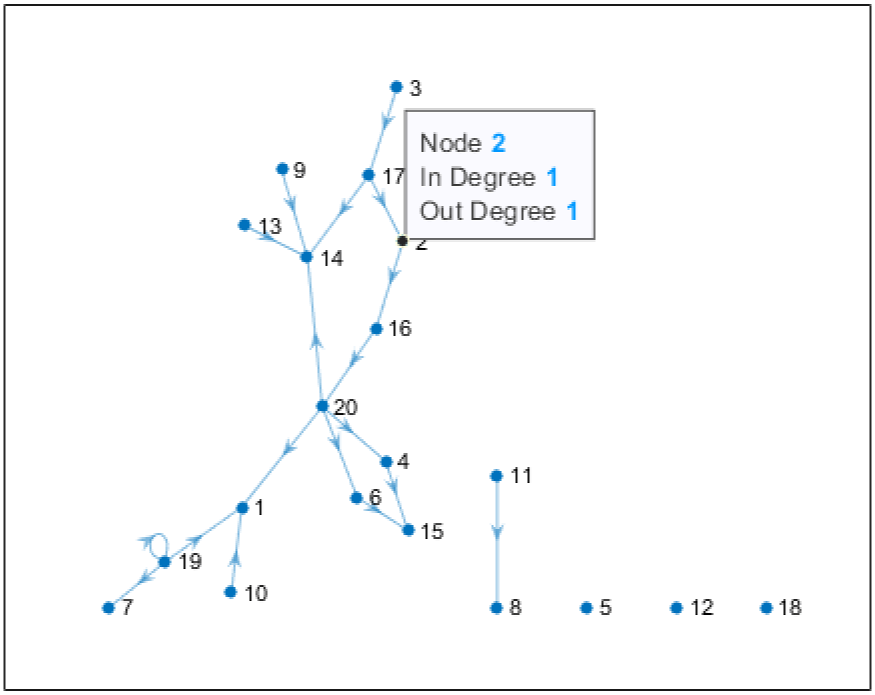
Create a `GraphPlot` graphics object for a random directed graph. Add an extra node property `wifi` to the graph.

```
rng default
G = digraph(sprandn(20, 20, 0.05));
G.Nodes.wifi = randi([0 1], 20, 1) == 1;
h = plot(G);
```



Add a data tip to the graph. The data tip enables you to select nodes in the graph plot and view properties of the nodes.

```
dt = datatip(h,4,3);
```



By default, the data tips for an undirected graph display the node number and degree. For directed graphs, the display includes the node number, in-degree, and out-degree.

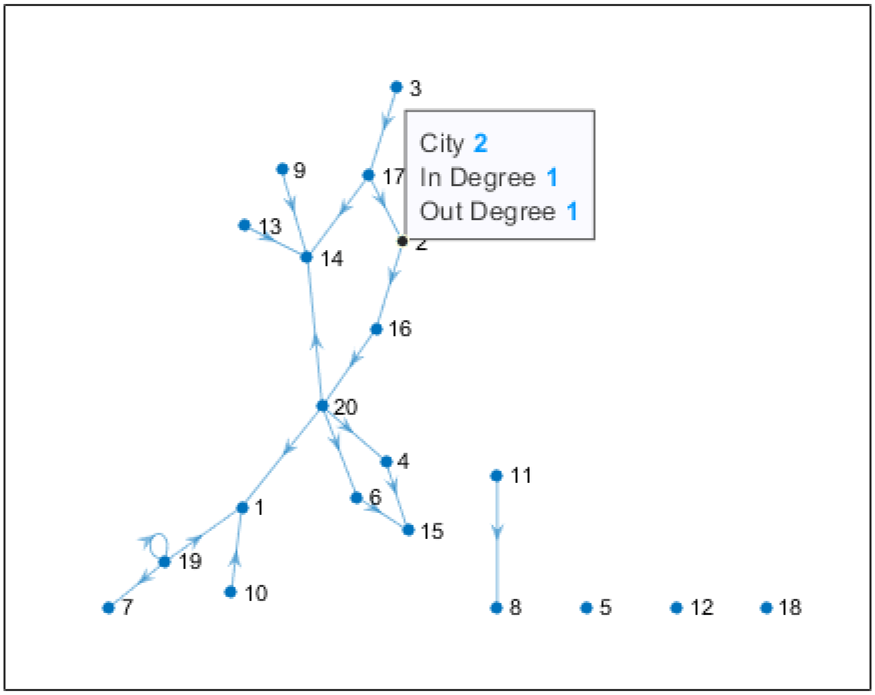
Customize Existing Data in Data Tip

You can customize the display of data tips for graphics objects by adding, editing, or removing rows of data from the appropriate object properties. For this `GraphPlot` object:

- The `GraphPlot` object handle is `h`.
- The `h.DataTipTemplate` property contains an object that controls the display of the data tips.
- The `h.DataTipTemplate.DataTipRows` property holds the data for the data tips as `DataTipTextRow` objects.
- Each `DataTipTextRow` object has `Label` and `Value` properties. You can adjust the label or data that is displayed in the data tip by modifying these properties.

Change the label for the Node row in the data tip so that it displays as "City".

```
h.DataTipTemplate.DataTipRows(1).Label = "City";
```

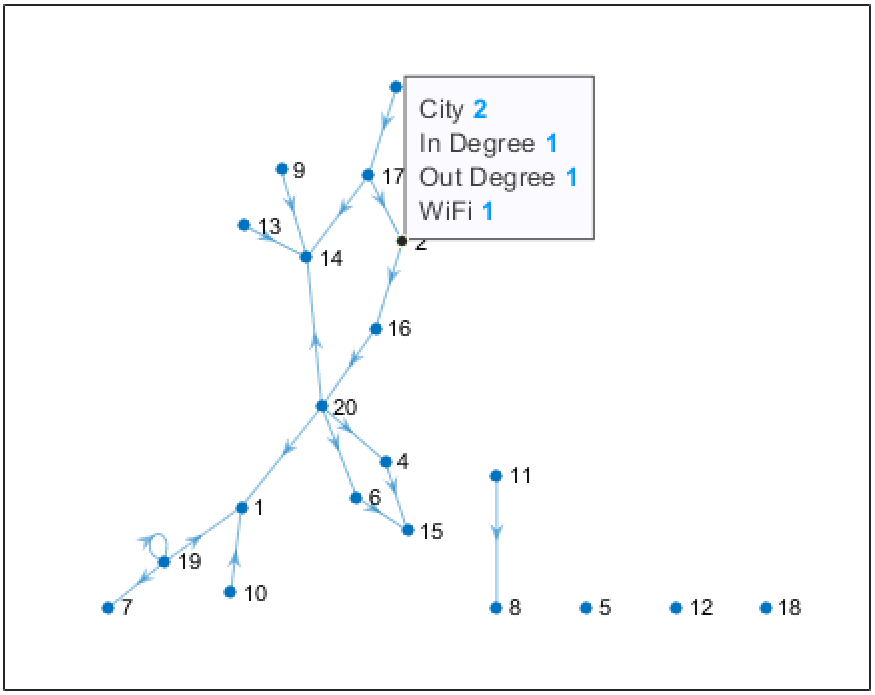


The data tip now displays a city number.

Add Data to Data Tip

The `dataTipTextRow` function creates a new row of data as an object that can be inserted into the `DataTipRows` property. Use `dataTipTextRow` to create a new row of data for the data tip labeled "WiFi" that references the values in the `G.Nodes.wifi` property of the graph. Add this data tip row to the `DataTipRows` property as the last row.

```
row = dataTipTextRow('WiFi',G.Nodes.wifi);
h.DataTipTemplate.DataTipRows(end+1) = row;
```



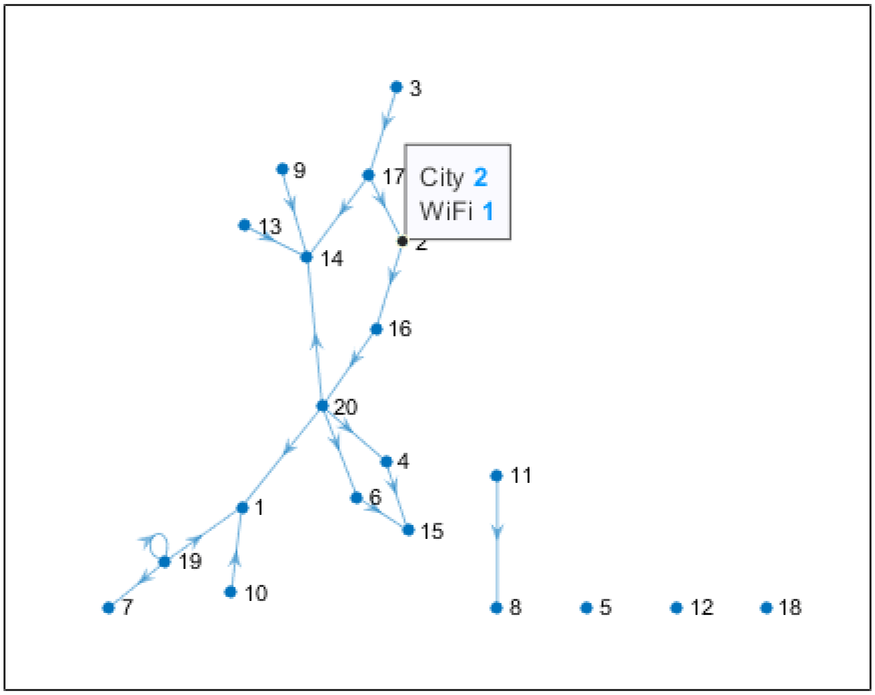
The data tip display now includes a WiFi value for each node.

Remove Data from Data Tip

To remove rows of data from the data tip, you can index into the `DataTipRows` property and assign the rows an empty matrix `[]`. This is the same method you might use to delete rows or columns from a matrix.

Delete the in-degree and out-degree rows from the data tip. Since these appear as the second and third rows in the data tip display, they correspond to the second and third rows of the `DataTipRows` property.

```
h.DataTipTemplate.DataTipRows(2:3) = [];
```



The data tip display now only displays the city number and WiFi status.

See Also

DataTipTemplate Properties | `datatip` | `digraph` | `graph`

More About

- “Create Custom Data Tips”
- “Interactively Explore Plotted Data”

Build Watts-Strogatz Small World Graph Model

This example shows how to construct and analyze a Watts-Strogatz small-world graph. The Watts-Strogatz model is a random graph that has small-world network properties, such as clustering and short average path length.

Algorithm Description

Creating a Watts-Strogatz graph has two basic steps:

1. Create a ring lattice with N nodes of mean degree $2K$. Each node is connected to its K nearest neighbors on either side.
2. For each edge in the graph, rewire the target node with probability β. The rewired edge cannot be a duplicate or self-loop.

After the first step the graph is a perfect ring lattice. So when $\beta = 0$, no edges are rewired and the model returns a ring lattice. In contrast, when $\beta = 1$, all of the edges are rewired and the ring lattice is transformed into a random graph.

The file `WattsStrogatz.m` implements this graph algorithm for undirected graphs. The input parameters are `N`, `K`, and `beta` according to the algorithm description above.

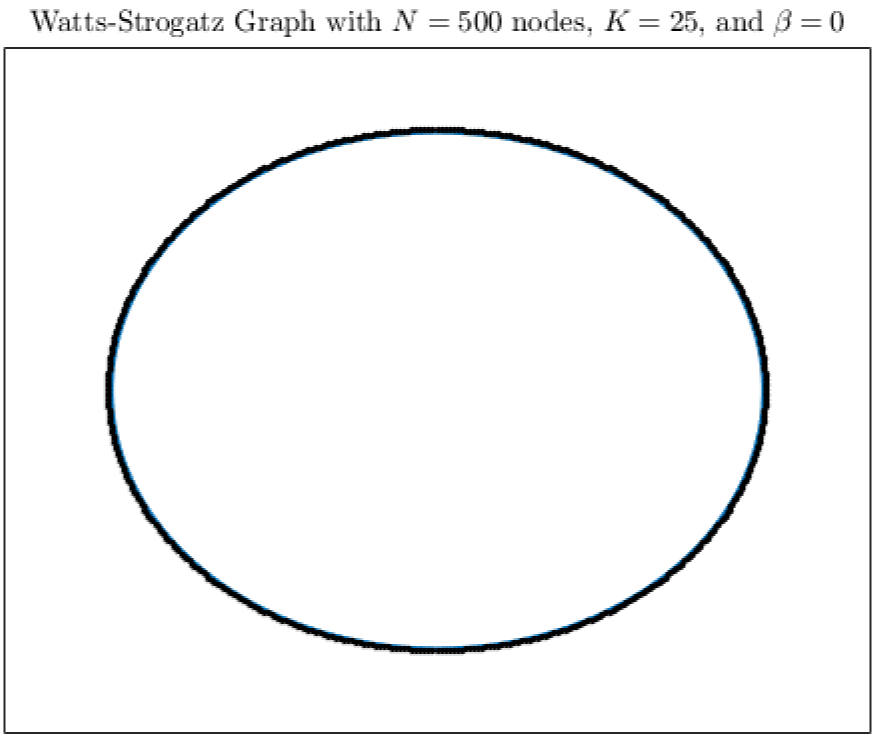
View the file `WattsStrogatz.m`.

```
% Copyright 2015 The MathWorks, Inc.

function h = WattsStrogatz(N,K,beta)
% H = WattsStrogatz(N,K,beta) returns a Watts-Strogatz model graph with N
% nodes, N*K edges, mean node degree 2*K, and rewiring probability beta.
%
% beta = 0 is a ring lattice, and beta = 1 is a random graph.

% Connect each node to its K next and previous neighbors. This constructs
% indices for a ring lattice.
s = repelem((1:N)',1,K);
t = s + repmat(1:K,N,1);
t = mod(t-1,N)+1;

% Rewire the target node of each edge with probability beta
for source=1:N
    switchEdge = rand(K, 1) < beta;

    newTargets = rand(N, 1);
    newTargets(source) = 0;
    newTargets(s(t==source)) = 0;
    newTargets(t(source, ~switchEdge)) = 0;

    [~, ind] = sort(newTargets, 'descend');
    t(source, switchEdge) = ind(1:nnz(switchEdge));
end

h = graph(s,t);
end
```

Ring Lattice

Construct a ring lattice with 500 nodes using the `WattsStrogatz` function. When `beta` is 0, the function returns a ring lattice whose nodes all have degree 2K.

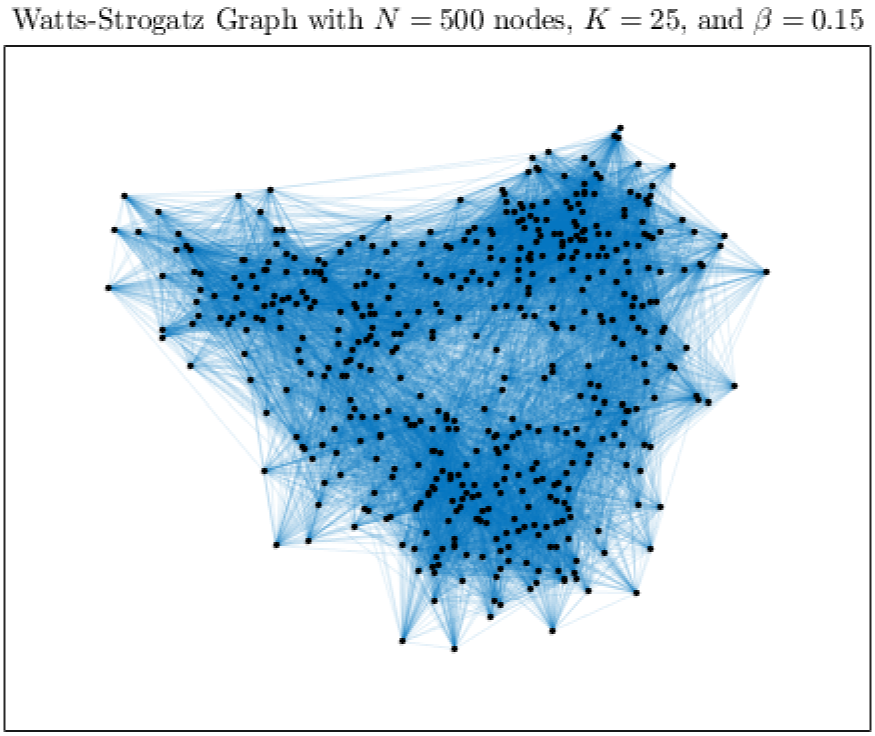
```
h = WattsStrogatz(500,25,0);
plot(h,'NodeColor','k','Layout','circle');
title('Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0$', ...
    'Interpreter','latex')
```



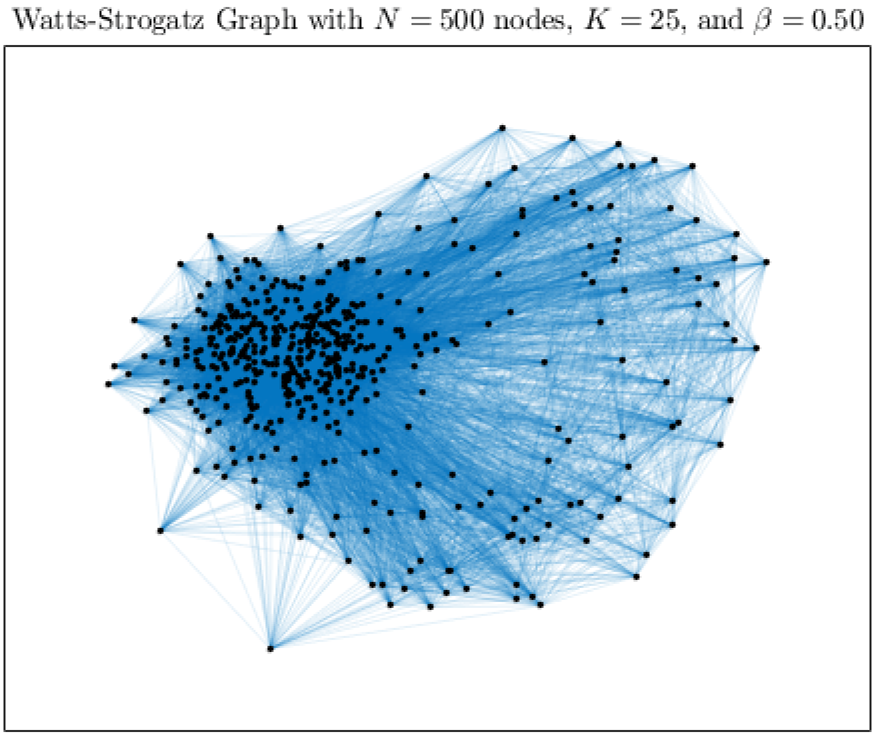
Some Random Edges

Increase the amount of randomness in the graph by raising `beta` to `0.15` and `0.50`.

```
h2 = WattsStrogatz(500,25,0.15);
plot(h2,'NodeColor','k','EdgeAlpha',0.1);
title('Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0.15$', ...
    'Interpreter','latex')
```



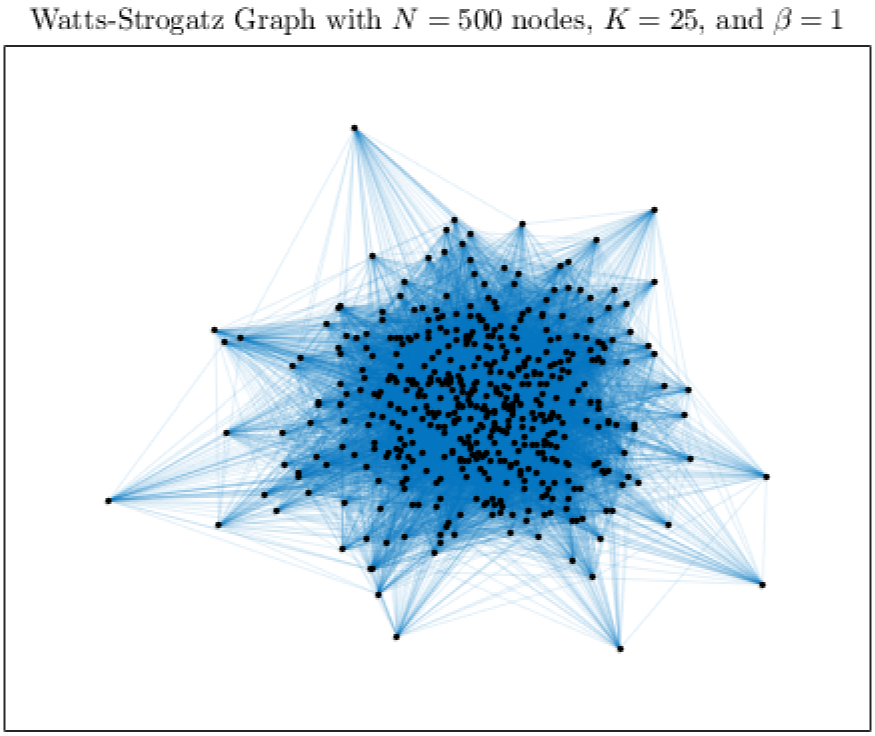
```
h3 = WattsStrogatz(500,25,0.50);
plot(h3,'NodeColor','k','EdgeAlpha',0.1);
title('Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0.50$', ...
    'Interpreter','latex')
```



Random Graph

Generate a completely random graph by increasing `beta` to its maximum value of `1.0`. This rewires all of the edges.

```
h4 = WattsStrogatz(500,25,1);
plot(h4,'NodeColor','k','EdgeAlpha',0.1);
title('Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 1$', ...
    'Interpreter','latex')
```

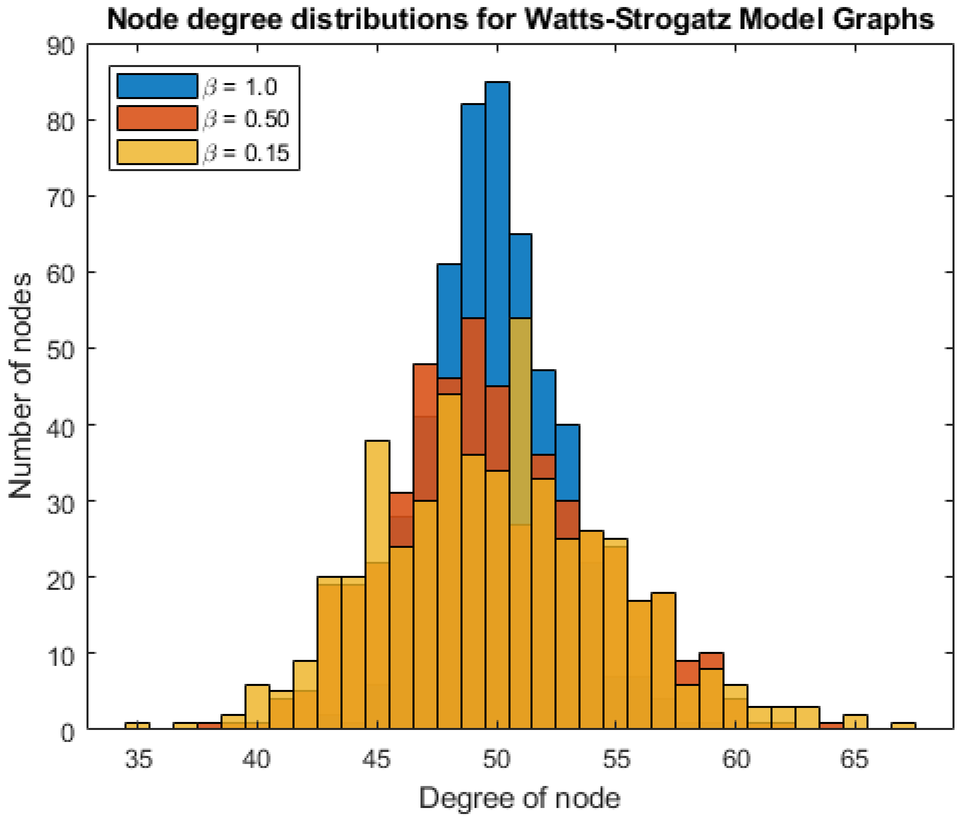

Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 1$

Degree Distribution

The degree distribution of the nodes in the different Watts-Strogatz graphs varies. When beta is 0, the nodes all have the same degree, 2K, so the degree distribution is just a Dirac-delta function centered on 2K, $\delta(x - 2K)$. However, as beta increases, the degree distribution changes.

This plot shows the degree distributions for the nonzero values of beta.

```
histogram(degree(h2),'BinMethod','integers','FaceAlpha',0.9);
hold on
histogram(degree(h3),'BinMethod','integers','FaceAlpha',0.9);
histogram(degree(h4),'BinMethod','integers','FaceAlpha',0.8);
hold off
title('Node degree distributions for Watts-Strogatz Model Graphs')
xlabel('Degree of node')
ylabel('Number of nodes')
legend('\beta = 1.0','\beta = 0.50','\beta = 0.15','Location','NorthWest')
```



Hub Formation

The Watts-Strogatz graph has a high clustering coefficient, so the nodes tend to form cliques, or small groups of closely interconnected nodes. As `beta` increases towards its maximum value of `1.0`, you see an increasingly large number of hub nodes, or nodes of high relative degree. The hubs are a common connection between other nodes and between cliques in the graph. The existence of hubs is what permits the formation of cliques while preserving a short average path length.

Calculate the average path length and number of hub nodes for each value of `beta`. For the purposes of this example, the hub nodes are nodes with degree greater than or equal to 55. These are all of the nodes whose degree increased 10% or more compared to the original ring lattice.

```
n = 55;
d = [mean(mean(distances(h))), nnz(degree(h)>=n); ...
     mean(mean(distances(h2))), nnz(degree(h2)>=n); ...
     mean(mean(distances(h3))), nnz(degree(h3)>=n);
     mean(mean(distances(h4))), nnz(degree(h4)>=n)];
T = table([0 0.15 0.50 1]', d(:,1), d(:,2),...
    'VariableNames',{'Beta','AvgPathLength','NumberOfHubs'})
```

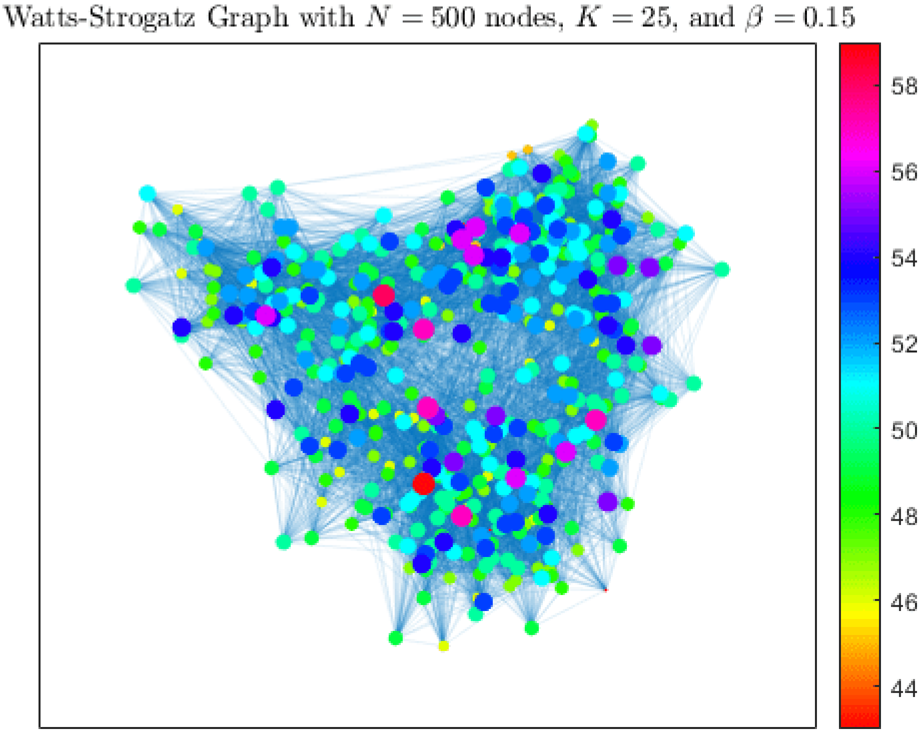
```
T =

  4x3 table

    Beta    AvgPathLength    NumberOfHubs
    ____    _____________    ____________
```

```
       0          5.48                  0
    0.15        2.0715                 20
     0.5        1.9101                 85
       1        1.9008                 92
```

As `beta` increases, the average path length in the graph quickly falls to its limiting value. This is due to the formation of the highly connected hub nodes, which become more numerous as `beta` increases.

Plot the $\beta = 0.15$ Watts-Strogatz model graph, making the size and color of each node proportional to its degree. This is an effective way to visualize the formation of hubs.

```
colormap hsv
deg = degree(h2);
nSizes = 2*sqrt(deg-min(deg)+0.2);
nColors = deg;
plot(h2,'MarkerSize',nSizes,'NodeCData',nColors,'EdgeAlpha',0.1)
title('Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0.15$', ...
    'Interpreter','latex')
colorbar
```



See Also

`digraph` | `graph`

Use PageRank Algorithm to Rank Websites

This example shows how to use a PageRank algorithm to rank a collection of websites. Although the PageRank algorithm was originally designed to rank search engine results, it also can be more broadly applied to the nodes in many different types of graphs. The PageRank score gives an idea of the relative importance of each graph node based on how it is connected to the other nodes.

Theoretically, the PageRank score is the limiting probability that someone randomly clicking links on each website will arrive at any particular page. So pages with a high score are highly connected and discoverable within the network, and it is more likely a random web surfer will visit that page.

Algorithm Description

At each step in the PageRank algorithm, the score of each page is updated according to,

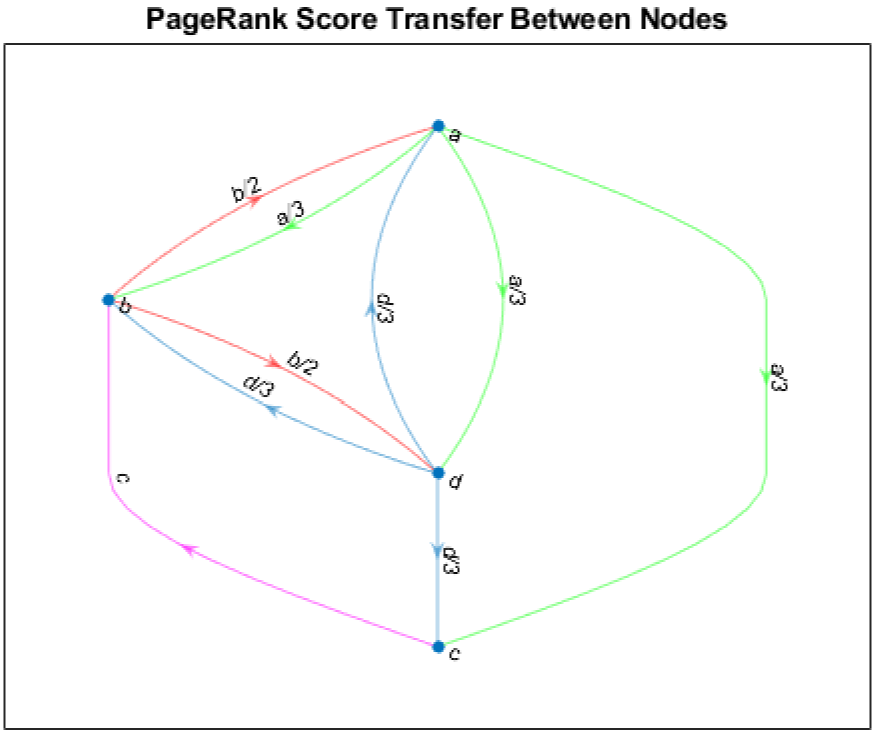
```
r = (1-P)/n + P*(A'*(r./d) + s/n);
```

- r is a vector of PageRank scores.
- P is a scalar damping factor (usually 0.85), which is the probability that a random surfer clicks on a link on the current page, instead of continuing on another random page.
- A' is the transpose of the adjacency matrix of the graph.
- d is a vector containing the out-degree of each node in the graph. d is set to 1 for nodes with no outgoing edges.
- n is the scalar number of nodes in the graph.
- s is the scalar sum of the PageRank scores for pages with no links.

In other words, the rank of each page is largely based on the ranks of the pages that link to it. The term A'*(r./d) picks out the scores of the source nodes that link to each node in the graph, and the scores are normalized by the total number of outbound links of those source nodes. This ensures that the sum of the PageRank scores is always 1. For example, if node 2 links to nodes 1, 3, and 4, then it transfers 1/3 of its PageRank score to each of those nodes during each iteration of the algorithm.

Create a graph that illustrates how each node confers its PageRank score to the other nodes in the graph.

```
s = {'a' 'a' 'a' 'b' 'b' 'c' 'd' 'd' 'd'};
t = {'b' 'c' 'd' 'd' 'a' 'b' 'c' 'a' 'b'};
G = digraph(s,t);
labels = {'a/3' 'a/3' 'a/3' 'b/2' 'b/2' 'c' 'd/3' 'd/3' 'd/3'};
p = plot(G,'Layout','layered','EdgeLabel',labels);
highlight(p,[1 1 1],[2 3 4],'EdgeColor','g')
highlight(p,[2 2],[1 4],'EdgeColor','r')
highlight(p,3,2,'EdgeColor','m')
title('PageRank Score Transfer Between Nodes')
```



The `centrality` function contains an option for calculating PageRank scores.

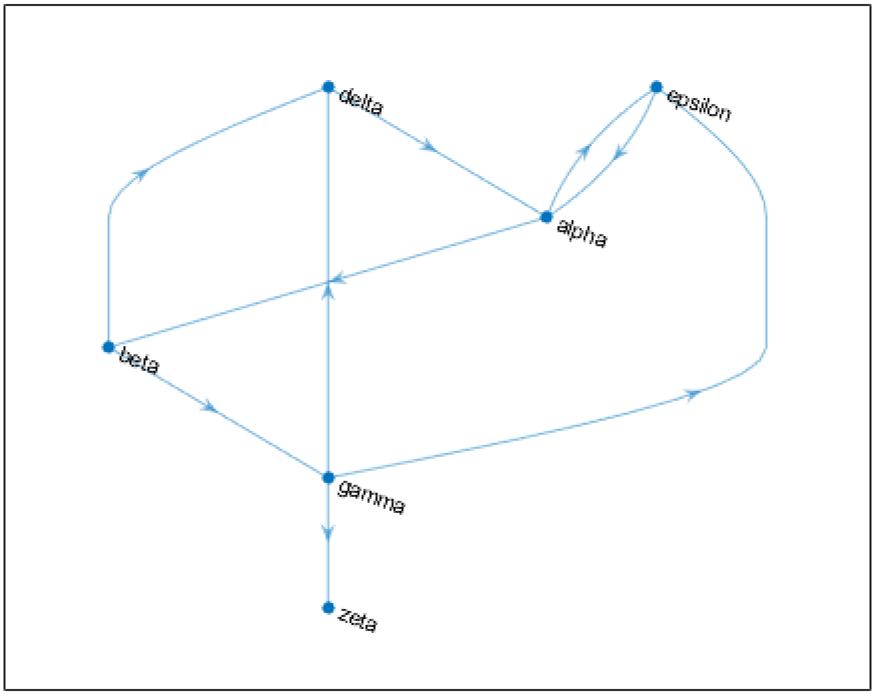
PageRank with 6 Nodes

Create and plot a directed graph containing six nodes representing fictitious websites.

```
s = [1 1 2 2 3 3 3 4 5];
t = [2 5 3 4 4 5 6 1 1];
names = {'http://www.example.com/alpha', 'http://www.example.com/beta', ...
         'http://www.example.com/gamma', 'http://www.example.com/delta', ...
         'http://www.example.com/epsilon', 'http://www.example.com/zeta'};
G = digraph(s,t,[],names)
```

```
G = 
  digraph with properties:

    Edges: [9x1 table]
    Nodes: [6x1 table]
```

```
plot(G,'Layout','layered', ...
    'NodeLabel',{'alpha','beta','gamma','delta','epsilon','zeta'})
```



Calculate the PageRank centrality score for this graph. Use a follow probability (otherwise known as a damping factor) of `0.85`.

```
pr = centrality(G,'pagerank','FollowProbability',0.85)
```

```
pr = 6×1

    0.3210
    0.1706
    0.1066
    0.1368
    0.2008
    0.0643
```

View the PageRank scores and degree information for each page.

```
G.Nodes.PageRank = pr;
G.Nodes.InDegree = indegree(G);
G.Nodes.OutDegree = outdegree(G);
G.Nodes
```

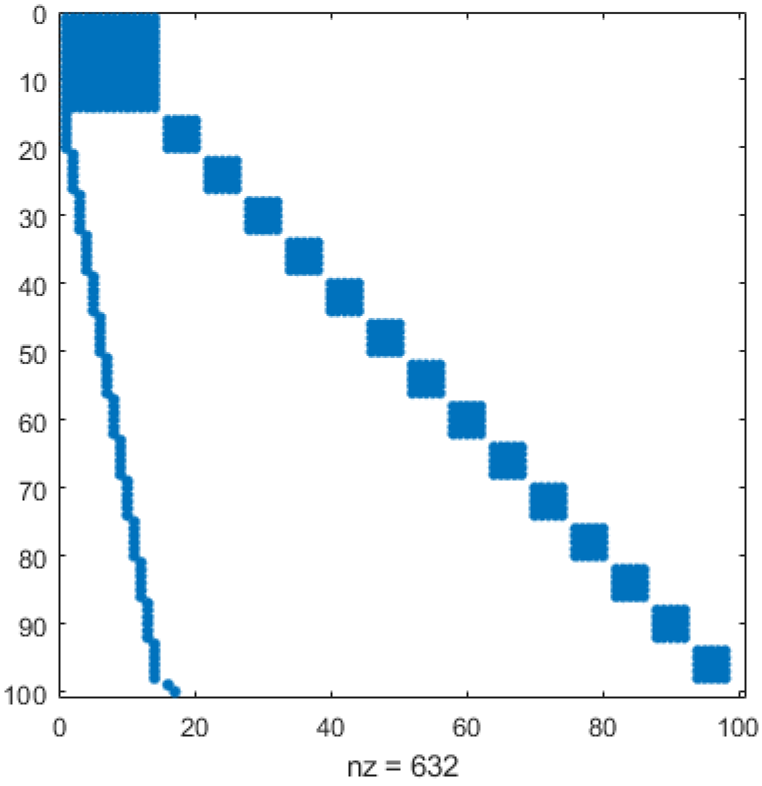
```
ans=6×4 table
                   Name                   PageRank    InDegree    OutDegree
    _________________________________    ________    ________    _________

    {'http://www.example.com/alpha'  }    0.32098        2            2    
    {'http://www.example.com/beta'   }    0.17057        1            2    
    {'http://www.example.com/gamma'  }    0.10657       1            3
    {'http://www.example.com/delta'  }    0.13678       2            1
    {'http://www.example.com/epsilon'}    0.20078       2            1
    {'http://www.example.com/zeta'   }    0.06432       1            0
```

The results show that it is not just the *number* of page links that determines the score, but also the *quality*. The `alpha` and `gamma` websites both have a total degree of 4, however `alpha` links to both `epsilon` and `beta`, which also are highly ranked. `gamma` is only linked to by one page, `beta`, which is in the middle of the list. Thus, `alpha` is scored higher than `gamma` by the algorithm.

PageRank of Websites on mathworks.com

Load the data in `mathworks100.mat` and view the adjacency matrix, `A`. This data was generated in 2015 using an automatic page crawler. The page crawler began at `https://www.mathworks.com` and followed links to subsequent web pages until the adjacency matrix contained information on the connections of 100 unique web pages.

```
load mathworks100.mat
spy(A)
```



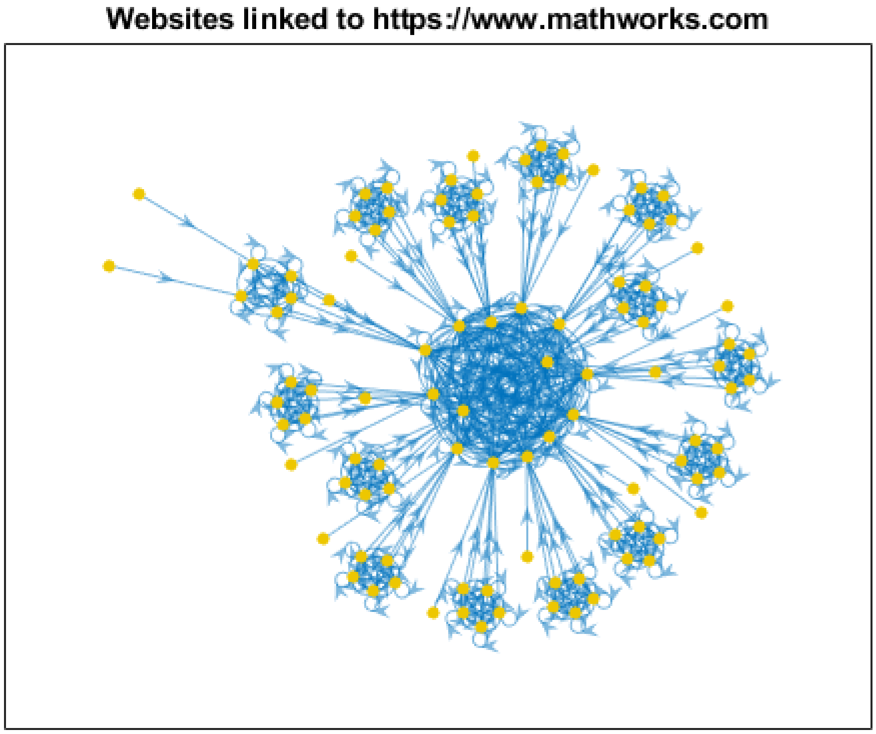
Create a directed graph with the sparse adjacency matrix, `A`, using the URLs contained in `U` as node names.

```
G = digraph(A,U)
```

```
G = 
  digraph with properties:
    Edges: [632x1 table]
    Nodes: [100x1 table]
```

Plot the graph using the force layout.

```
plot(G,'NodeLabel',{},'NodeColor',[0.93 0.78 0],'Layout','force');
title('Websites linked to https://www.mathworks.com')
```



Compute the PageRank scores for the graph, G, using 200 iterations and a damping factor of 0.85. Add the scores and degree information to the nodes table of the graph.

```
pr = centrality(G,'pagerank','MaxIterations',200,'FollowProbability',0.85);
G.Nodes.PageRank = pr;
G.Nodes.InDegree = indegree(G);
G.Nodes.OutDegree = outdegree(G);
```

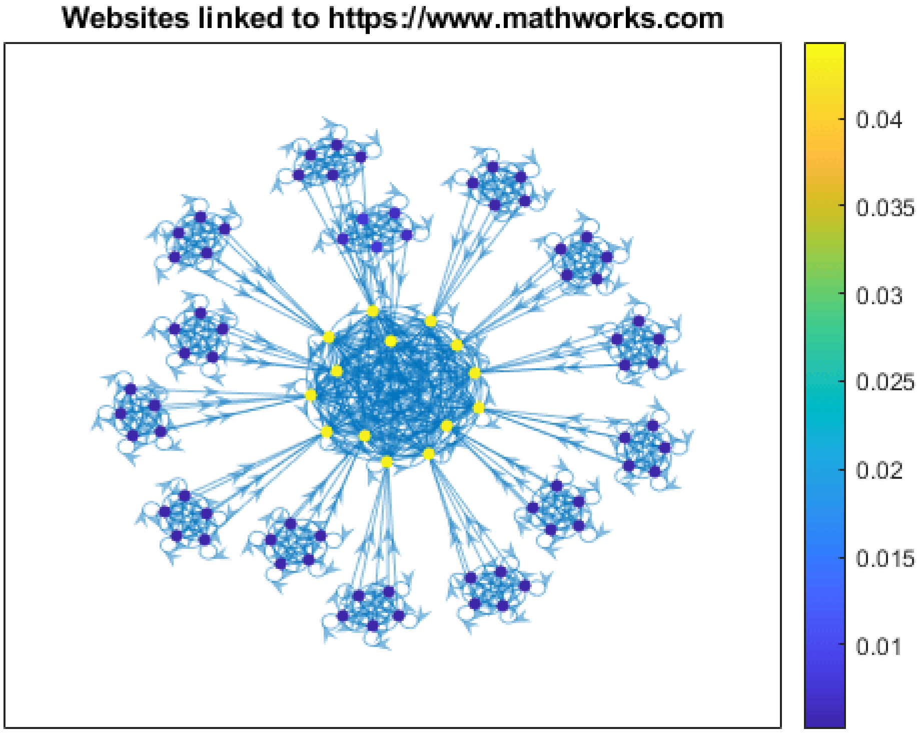
View the top 25 resulting scores.

```
G.Nodes(1:25,:)
```

```
ans=25×4 table
                                   Name                                          PageRank
    _______________________________________________________________________     ________

    {'https://www.mathworks.com'                                           }    0.044342
    {'https://ch.mathworks.com'                                            }    0.043085
    {'https://cn.mathworks.com'                                                  }    0.043085
    {'https://jp.mathworks.com'                                                  }    0.043085
    {'https://kr.mathworks.com'                                                  }    0.043085
    {'https://uk.mathworks.com'                                                  }    0.043085
    {'https://au.mathworks.com'                                                  }    0.043085
    {'https://de.mathworks.com'                                                  }    0.043085
    {'https://es.mathworks.com'                                                  }    0.043085
    {'https://fr.mathworks.com'                                                  }    0.043085
    {'https://in.mathworks.com'                                                  }    0.043085
    {'https://it.mathworks.com'                                                  }    0.043085
    {'https://nl.mathworks.com'                                                  }    0.043085
    {'https://se.mathworks.com'                                                  }    0.043085
    {'https://www.mathworks.com/index.html%3Fnocookie%3Dtrue'                    }      0.0015
    {'https://www.mathworks.com/company/aboutus/policies_statements/patents.html'}    0.007714
       ⋮
```

Extract and plot a subgraph containing all nodes whose score is greater than `0.005`. Color the graph nodes based on their PageRank score.

```
H = subgraph(G,find(G.Nodes.PageRank > 0.005));
plot(H,'NodeLabel',{},'NodeCData',H.Nodes.PageRank,'Layout','force');
title('Websites linked to https://www.mathworks.com')
colorbar
```



The PageRank scores for the top websites are all quite similar, such that a random web surfer has about a 4.5% chance to land on each page. This small group of highly connected pages forms a clique in the center of the plot. Connected to this central clique are several smaller cliques, which are highly connected amongst themselves.

References

Moler, C. *Experiments with MATLAB*. Chapter 7: Google PageRank. MathWorks, Inc., 2011.

See Also

`centrality` | `digraph` | `graph`

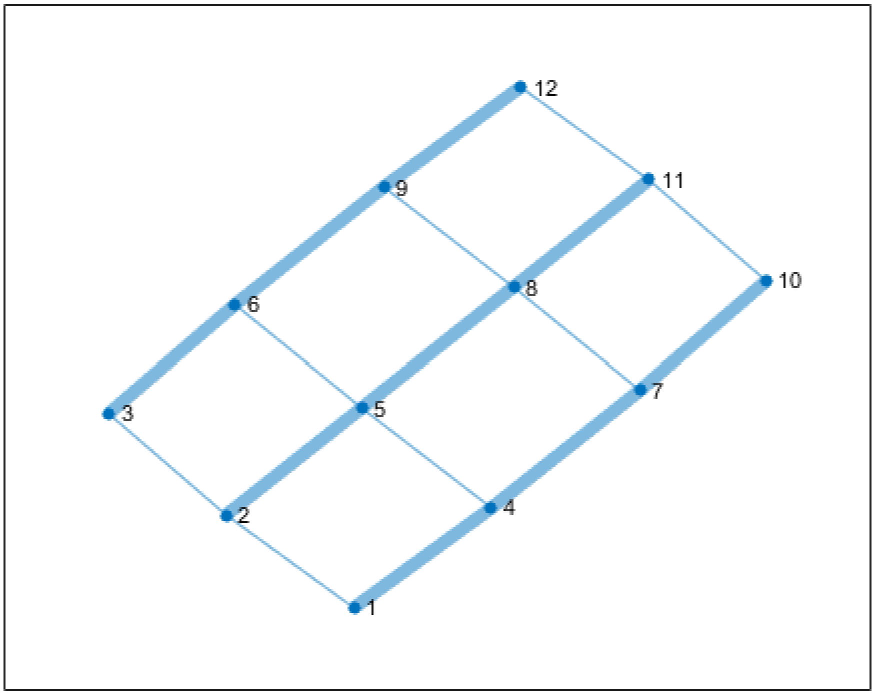
Label Graph Nodes and Edges

This example shows how to add and customize labels on graph nodes and edges.

Create and Plot Graph

Create a graph representing the gridded streets and intersections in a city. Add weights to the edges so that the main avenues and cross streets appear differently in the plot. Plot the graph with the edge line widths proportional to the weight of the edge.

```
s = [1 1 2 2 3 4 4 5 5 6 7 7 8 8 9 10 11];
t = [2 4 3 5 6 5 7 6 8 9 8 10 9 11 12 11 12];
weights = [1 5 1 5 5 1 5 1 5 5 1 5 1 5 5 1 1];
G = graph(s,t,weights);
P = plot(G,'LineWidth',G.Edges.Weight);
```

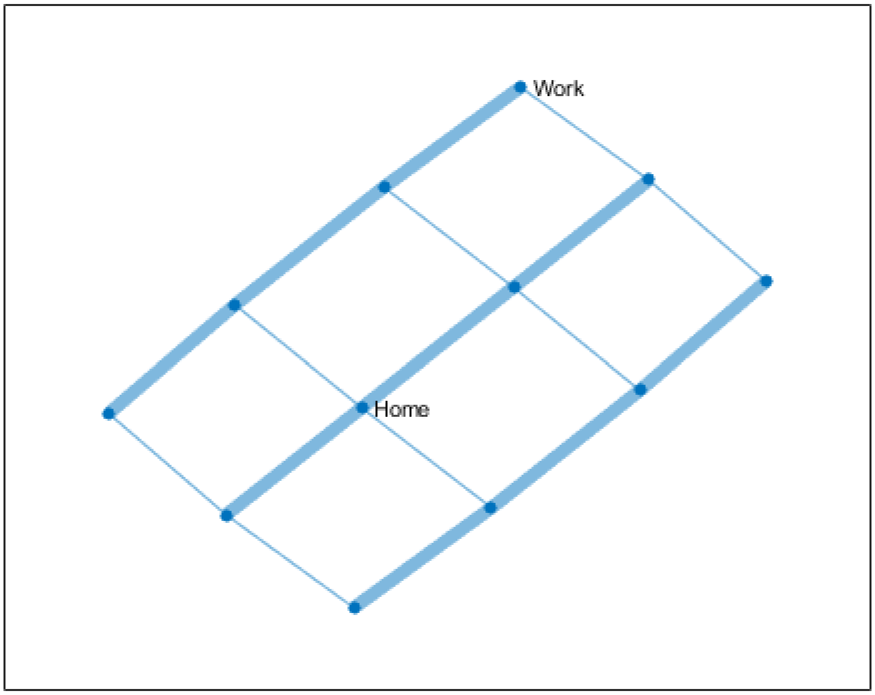


Add Node Labels

For graphs with 100 or fewer nodes, MATLAB® automatically labels the nodes using the numeric node indices or node names (larger graphs omit these labels by default). However, you can change the node labels by adjusting the `NodeLabel` property of the `GraphPlot` object `P` or by using the `labelnode` function. Therefore, even if the nodes have names, you can use labels that are different from the names.

Remove the default numeric node labels. Label one of the intersections as `Home` and another as `Work`.

```
labelnode(P,1:12,'')
labelnode(P,5,'Home')
labelnode(P,12,'Work')
```



Add Edge Labels

The edges in a plotted graph are *not* labeled automatically. You can add edge labels by changing the value of the `EdgeLabel` property of the `GraphPlot` object `P` or by using the `labeledge` function.

Add edge labels for streets in New York City. The order of the edges is defined in the `G.Edges` table of the graph, so the order of the labels you specify must respect that order. It is convenient to store edge labels directly in the `G.Edges` table, so that the edge name lives right next to the other edge information.

```
G.Edges
```

```
ans=17×2 table
    EndNodes     Weight
    ________     ______

     1     2       1
     1     4       5
     2     3       1
     2     5       5
     3     6       5
     4     5       1
     4     7       5
     5     6       1
     5     8       5
     6     9       5
     7     8       1
     7    10       5
     8     9       1
     8    11       5
     9    12       5
    10    11       1
      ⋮
```

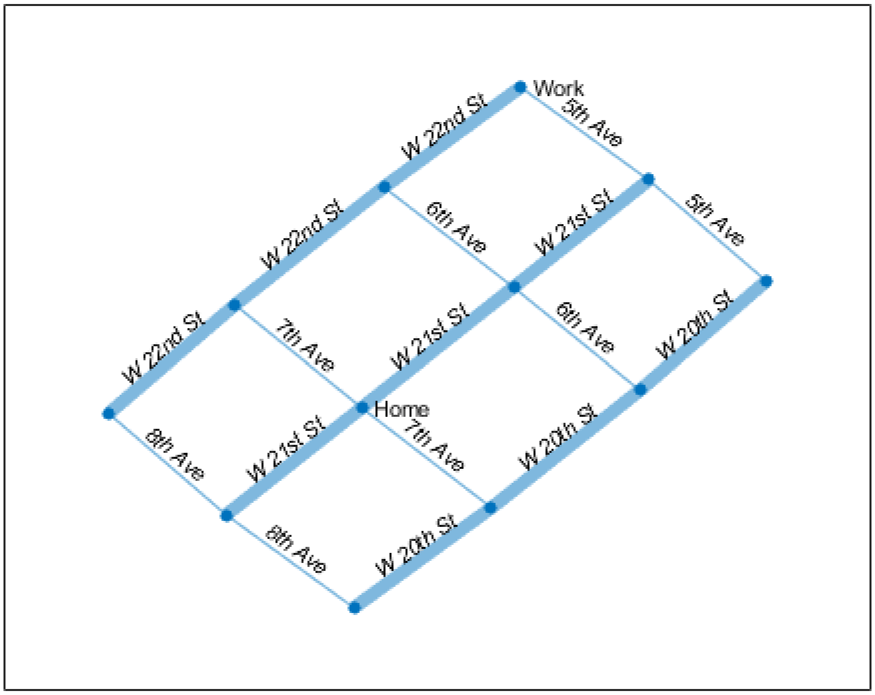
This example has 17 edges but only 7 unique street names. Therefore, it makes sense to define the street names in a cell array and then index into the cell array to retrieve the desired street name for each edge. Add a variable to the `G.Edges` table containing the street names.

```
streets = {'8th Ave' '7th Ave' '6th Ave' '5th Ave' ...
    'W 20th St' 'W 21st St' 'W 22nd St'}';
inds = [1 5 1 6 7 2 5 2 6 7 3 5 3 6 7 4 4];
G.Edges.StreetName = streets(inds);
G.Edges
```

```
ans=17×3 table
    EndNodes    Weight     StreetName 
    ________    ______    _____________

     1     2       1      {'8th Ave'  }
     1     4       5      {'W 20th St'}
     2     3       1      {'8th Ave'  }
     2     5       5      {'W 21st St'}
     3     6       5      {'W 22nd St'}
     4     5       1      {'7th Ave'  }
     4     7       5      {'W 20th St'}
     5     6       1      {'7th Ave'  }
     5     8       5      {'W 21st St'}
     6     9       5      {'W 22nd St'}
     7     8       1      {'6th Ave'  }
     7    10       5      {'W 20th St'}
     8     9       1      {'6th Ave'  }
     8    11       5      {'W 21st St'}
     9    12       5      {'W 22nd St'}
    10    11       1      {'5th Ave'  }
      ⋮
```

Update the `EdgeLabel` property to reference these street names.

```
P.EdgeLabel = G.Edges.StreetName;
```



Adjust Font Properties

The node and edge labels in a graph plot have their own properties that control the appearance and style of the labels. Since the properties are decoupled, you can use different styles for the node labels and the edge labels.

For **Node** labels, you can adjust:

- `NodeLabel`
- `NodeLabelColor`
- `NodeFontName`
- `NodeFontSize`
- `NodeFontWeight`
- `NodeFontAngle`

For **Edge** labels, you can adjust:

- `EdgeLabel`
- `EdgeLabelColor`
- `EdgeFontName`
- `EdgeFontSize`
- `EdgeFontWeight`

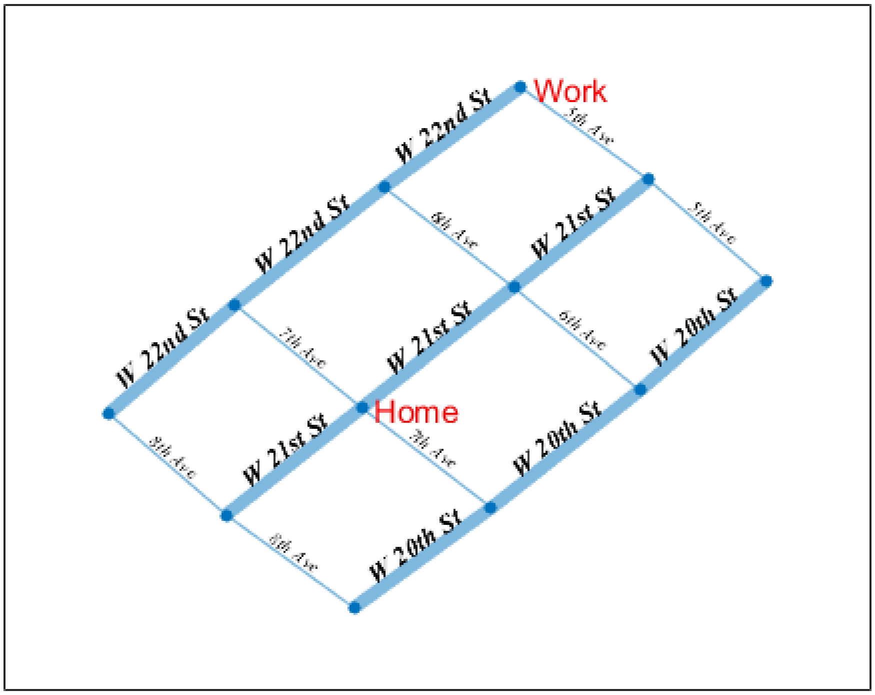
- `EdgeFontAngle`

Use these properties to adjust the fonts in this example with New York City Streets:

- Change `NodeFontSize` and `NodeLabelColor` so that the intersection labels are 12 pt. font and red.
- Change `EdgeFontWeight`, `EdgeFontAngle`, and `EdgeFontSize` to use a larger, bold font for streets in one direction and a smaller, italic font for streets in the other direction.
- Change `EdgeFontName` to use Times New Roman for the edge labels.

You can use the `highlight` function to change the graph properties of a subset of the graph edges. Create a logical index `isAvenue` that is `true` for edge labels containing the word `'Ave'`. Using this logical vector as an input to `highlight`, label all of the Avenues in one way, and all of the non-Avenues another way.

```
P.NodeFontSize = 12;
P.NodeLabelColor = 'r';
isAvenue = contains(P.EdgeLabel, 'Ave');
highlight(P, 'Edges', isAvenue, 'EdgeFontAngle', 'italic', 'EdgeFontSize', 7);
highlight(P, 'Edges', ~isAvenue, 'EdgeFontWeight', 'bold', 'EdgeFontSize', 10);
P.EdgeFontName = 'Times New Roman';
```



Highlight Edges

Find the shortest path between the Home and Work nodes and examine which streets are on the path. Highlight the nodes and edges on the path in red and remove the edge labels for all edges that are not on the path.

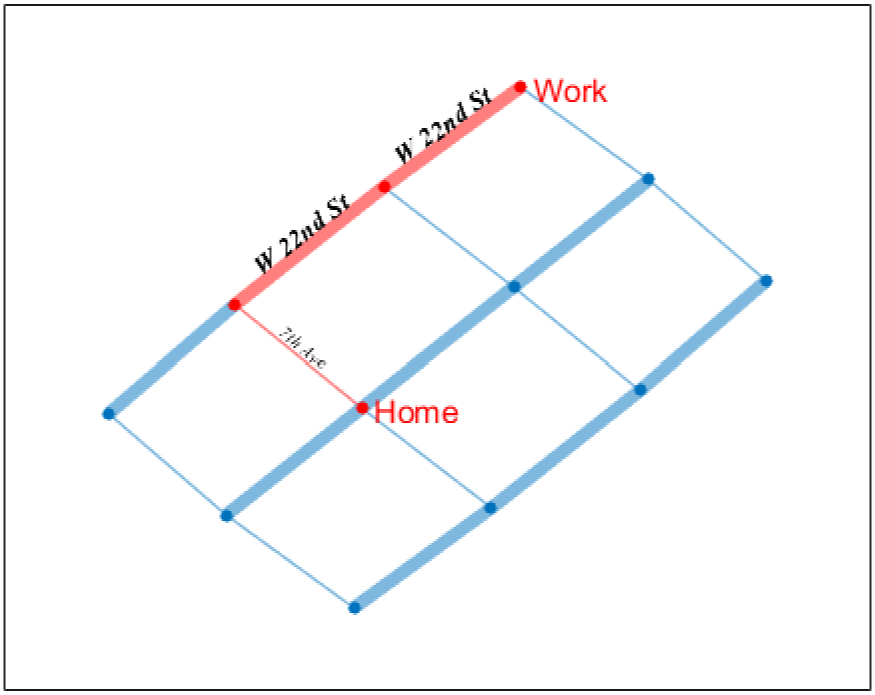
```
[path,d,pathEdges] = shortestpath(G,5,12)
```

```
path = 1×4

     5     6     9    12


d = 11

pathEdges = 1×3

     8    10    15
```

```
G.Edges.StreetName(pathEdges,:)
```

```
ans = 3x1 cell
    {'7th Ave'  }
    {'W 22nd St'}
    {'W 22nd St'}
```

```
highlight(P,'Edges',pathEdges,'EdgeColor','r')
highlight(P,path,'NodeColor','r')
labeledge(P, setdiff(1:numedges(G), pathEdges), '')
```



See Also

GraphPlot

More About

- “Graph Plotting and Customization” on page 5-17
- “Add Node Properties to Graph Plot Data Tips” on page 5-36

Functions of One Variable

Create and Evaluate Polynomials

This example shows how to represent a polynomial as a vector in MATLAB® and evaluate the polynomial at points of interest.

Representing Polynomials

MATLAB® represents polynomials as row vectors containing coefficients ordered by descending powers. For example, the three-element vector

```
p = [p2 p1 p0];
```

represents the polynomial

$$p(x) = p_2x^2 + p_1x + p_0 .$$

Create a vector to represent the quadratic polynomial $p(x) = x^2 - 4x + 4$.

```
p = [1 -4 4];
```

Intermediate terms of the polynomial that have a coefficient of `0` must also be entered into the vector, since the `0` acts as a placeholder for that particular power of `x`.

Create a vector to represent the polynomial $p(x) = 4x^5 - 3x^2 + 2x + 33$.

```
p = [4 0 0 -3 2 33];
```

Evaluating Polynomials

After entering the polynomial into MATLAB® as a vector, use the `polyval` function to evaluate the polynomial at a specific value.

Use `polyval` to evaluate $p(2)$.

```
polyval(p,2)
```

```
ans = 153
```

Alternatively, you can evaluate a polynomial in a matrix sense using `polyvalm`. The polynomial expression in one variable, $p(x) = 4x^5 - 3x^2 + 2x + 33$, becomes the matrix expression

$$p(X) = 4X^5 - 3X^2 + 2X + 33I,$$

where `X` is a square matrix and `I` is the identity matrix.

Create a square matrix, `X`, and evaluate `p` at `X`.

```
X = [2 4 5; -1 0 3; 7 1 5];
Y = polyvalm(p,X)
```

```
Y = 3×3

      154392         78561        193065
       49001         24104         59692
      215378      111419      269614
```

See Also

`poly` | `polyval` | `polyvalm` | `roots`

More About

- “Roots of Polynomials” on page 6-4
- “Integrate and Differentiate Polynomials” on page 6-9
- “Polynomial Curve Fitting” on page 6-11

Roots of Polynomials

This example shows several different methods to calculate the roots of a polynomial.

In this section...

Numeric Roots

The `roots` function calculates the roots of a single-variable polynomial represented by a vector of coefficients.

For example, create a vector to represent the polynomial $x^2 - x - 6$, then calculate the roots.

```
p = [1 -1 -6];
r = roots(p)
```

```
r =

     3
    -2
```

By convention, MATLAB returns the roots in a column vector.

The `poly` function converts the roots back to polynomial coefficients. When operating on vectors, `poly` and `roots` are inverse functions, such that `poly(roots(p))` returns `p` (up to roundoff error, ordering, and scaling).

```
p2 = poly(r)
```

```
p2 =

     1    -1    -6
```

When operating on a matrix, the `poly` function computes the characteristic polynomial of the matrix. The roots of the characteristic polynomial are the eigenvalues of the matrix. Therefore, `roots(poly(A))` and `eig(A)` return the same answer (up to roundoff error, ordering, and scaling).

Roots Using Substitution

You can solve polynomial equations involving trigonometric functions by simplifying the equation using a substitution. The resulting polynomial of one variable no longer contains any trigonometric functions.

For example, find the values of θ that solve the equation

$$3\cos^2(\theta) - \sin(\theta) + 3 = 0\,.$$

Use the fact that $\cos^2(\theta) = 1 - \sin^2(\theta)$ to express the equation entirely in terms of sine functions:

$$-3\sin^2(\theta) - \sin(\theta) + 6 = 0 \,.$$

Use the substitution $x = \sin(\theta)$ to express the equation as a simple polynomial equation:

$$-3x^2 - x + 6 = 0 \,.$$

Create a vector to represent the polynomial.

```
p = [-3 -1 6];
```

Find the roots of the polynomial.

```
r = roots(p)
```

```
r = 2×1

   -1.5907
    1.2573
```

To undo the substitution, use $\theta = \sin^{-1}(x)$. The `asin` function calculates the inverse sine.

```
theta = asin(r)
```

```
theta = 2×1 complex

  -1.5708 + 1.0395i
   1.5708 - 0.7028i
```

Verify that the elements in `theta` are the values of θ that solve the original equation (within roundoff error).

```
f = @(Z) 3*cos(Z).^2 - sin(Z) + 3;
f(theta)
```

```
ans = 2×1 complex
10^-14 ×

  -0.0888 + 0.0647i
   0.2665 + 0.0399i
```

Roots in a Specific Interval

Use the `fzero` function to find the roots of a polynomial in a specific interval. Among other uses, this method is suitable if you plot the polynomial and want to know the value of a particular root.

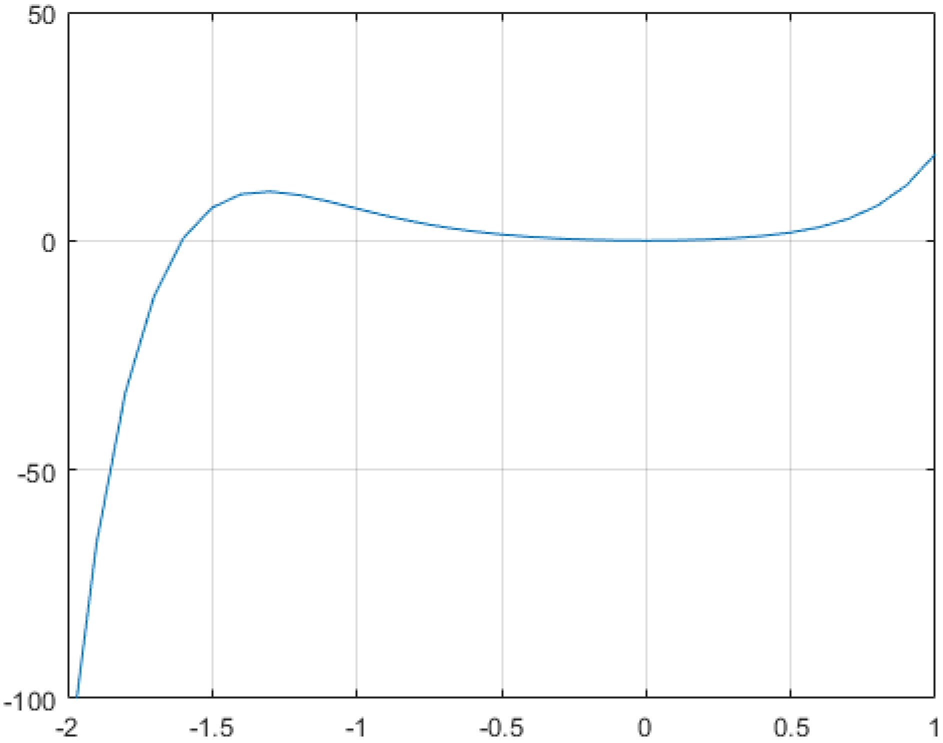
For example, create a function handle to represent the polynomial $3x^7 + 4x^6 + 2x^5 + 4x^4 + x^3 + 5x^2$.

```
p = @(x) 3*x.^7 + 4*x.^6 + 2*x.^5 + 4*x.^4 + x.^3 + 5*x.^2;
```

Plot the function over the interval $[-2, 1]$.

```
x = -2:0.1:1;
plot(x,p(x))
```

```
ylim([-100 50])
grid on
hold on
```

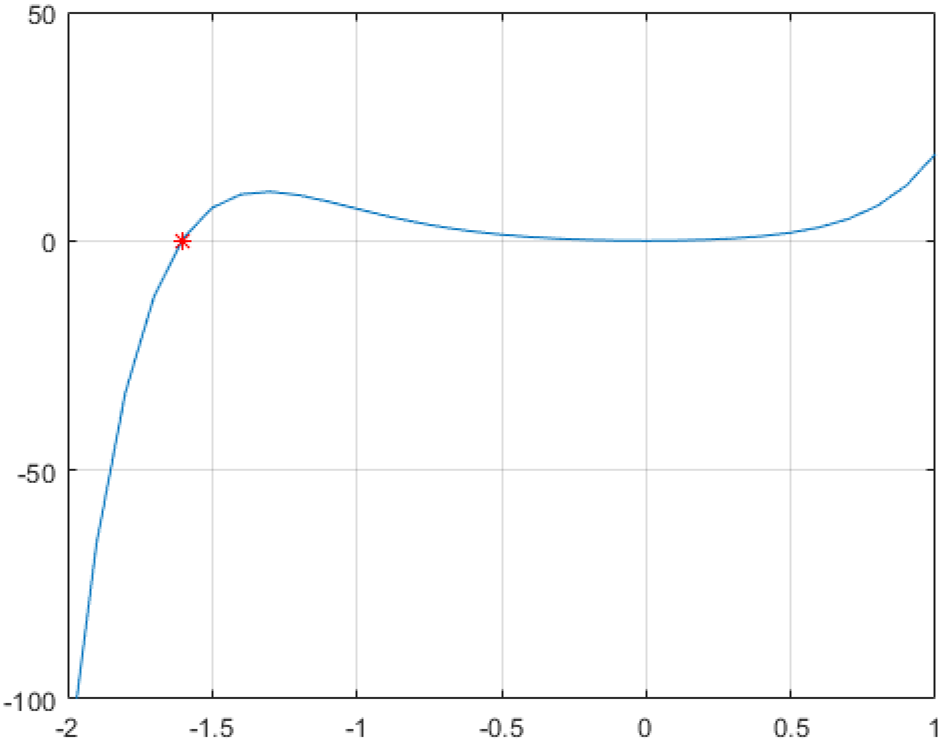


From the plot, the polynomial has a trivial root at `0` and another near `-1.5`. Use `fzero` to calculate and plot the root that is near `-1.5`.

```
Z = fzero(p, -1.5)
```

```
Z = -1.6056
```

```
plot(Z,p(Z),'r*')
```



Symbolic Roots

If you have Symbolic Math Toolbox™, then there are additional options for evaluating polynomials symbolically. One way is to use the `solve` function.

```
syms x
s = solve(x^2-x-6)
```

```
s =
 
 -2
  3
```

Another way is to use the `factor` function to factor the polynomial terms.

```
F = factor(x^2-x-6)
```

```
F =
 
[ x + 2, x - 3]
```

See “Solve Algebraic Equation” (Symbolic Math Toolbox) for more information.

See Also

`eig` | `poly` | `roots`

More About

- “Create and Evaluate Polynomials” on page 6-2
- “Roots of Scalar Functions” on page 6-18
- “Integrate and Differentiate Polynomials” on page 6-9

Integrate and Differentiate Polynomials

This example shows how to use the `polyint` and `polyder` functions to analytically integrate or differentiate any polynomial represented by a vector of coefficients.

Use `polyder` to obtain the derivative of the polynomial $p(x) = x^3 - 2x - 5$. The resulting polynomial is $q(x) = \frac{d}{dx}p(x) = 3x^2 - 2$.

```
p = [1 0 -2 -5];
q = polyder(p)
```

```
q = 1×3

     3     0    -2
```

Similarly, use `polyint` to integrate the polynomial $p(x) = 4x^3 - 3x^2 + 1$. The resulting polynomial is $q(x) = \int p(x)dx = x^4 - x^3 + x$.

```
p = [4 -3 0 1];
q = polyint(p)
```

```
q = 1×5

     1    -1     0     1     0
```

`polyder` also computes the derivative of the product or quotient of two polynomials. For example, create two vectors to represent the polynomials $a(x) = x^2 + 3x + 5$ and $b(x) = 2x^2 + 4x + 6$.

```
a = [1 3 5];
b = [2 4 6];
```

Calculate the derivative $\frac{d}{dx}[a(x)b(x)]$ by calling `polyder` with a single output argument.

```
c = polyder(a,b)
```

```
c = 1×4

     8    30    56    38
```

Calculate the derivative $\frac{d}{dx}\left[\frac{a(x)}{b(x)}\right]$ by calling `polyder` with two output arguments. The resulting polynomial is

$$\frac{d}{dx}\left[\frac{a(x)}{b(x)}\right] = \frac{-2x^2 - 8x - 2}{4x^4 + 16x^3 + 40x^2 + 48x + 36} = \frac{q(x)}{d(x)}.$$

```
[q,d] = polyder(a,b)
```

```
q = 1×3
```

```
    -2    -8    -2

d = 1×5

     4    16    40    48    36
```

See Also

`conv` | `deconv` | `polyder` | `polyint`

More About

- “Analytic Solution to Integral of Polynomial” on page 15-7
- “Create and Evaluate Polynomials” on page 6-2

Polynomial Curve Fitting

This example shows how to fit a polynomial curve to a set of data points using the polyfit function. You can use polyfit to find the coefficients of a polynomial that fits a set of data in a least-squares sense using the syntax

```
p = polyfit(x,y,n),
```

where:

- x and y are vectors containing the x and y coordinates of the data points
- n is the degree of the polynomial to fit

Create some *x*-*y* test data for five data points.

```
x = [1 2 3 4 5];
y = [5.5 43.1 128 290.7 498.4];
```

Use polyfit to find a third-degree polynomial that approximately fits the data.

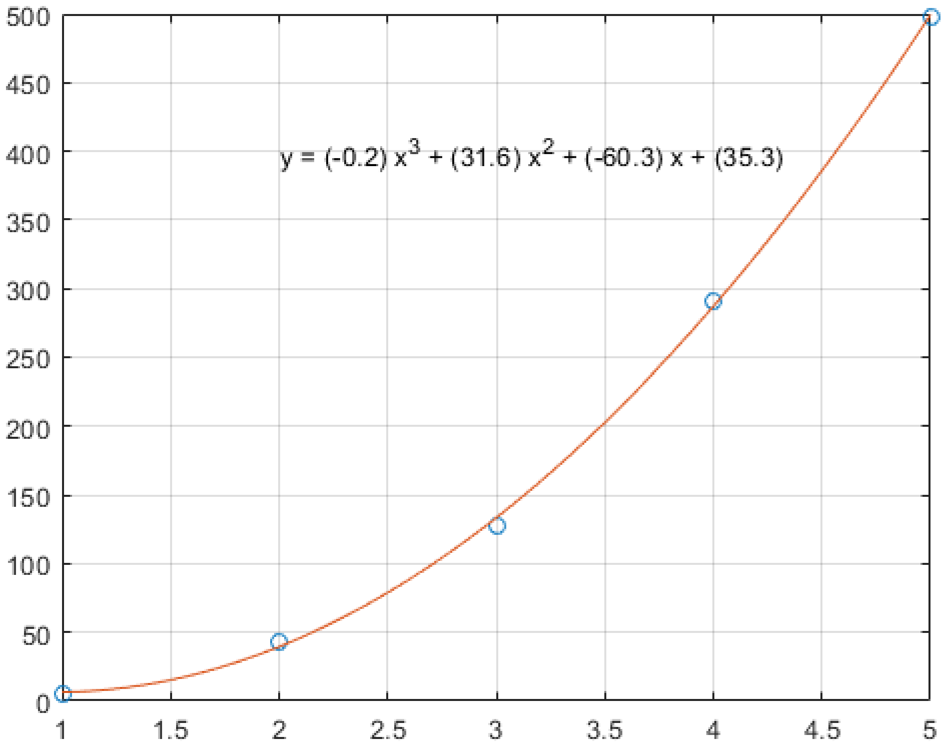
```
p = polyfit(x,y,3)
```

```
p = 1×4

   -0.1917   31.5821  -60.3262   35.3400
```

After you obtain the polynomial for the fit line using polyfit, you can use polyval to evaluate the polynomial at other points that might not have been included in the original data.

Compute the values of the polyfit estimate over a finer domain and plot the estimate over the real data values for comparison. Include an annotation of the equation for the fit line.

```
x2 = 1:.1:5;
y2 = polyval(p,x2);
plot(x,y,'o',x2,y2)
grid on
s = sprintf('y = (%.1f) x^3 + (%.1f) x^2 + (%.1f) x + (%.1f)',p(1),p(2),p(3),p(4));
text(2,400,s)
```



See Also

`polyfit` | `polyval`

More About

- "Programmatic Fitting"
- "Create and Evaluate Polynomials" on page 6-2

Predicting the US Population

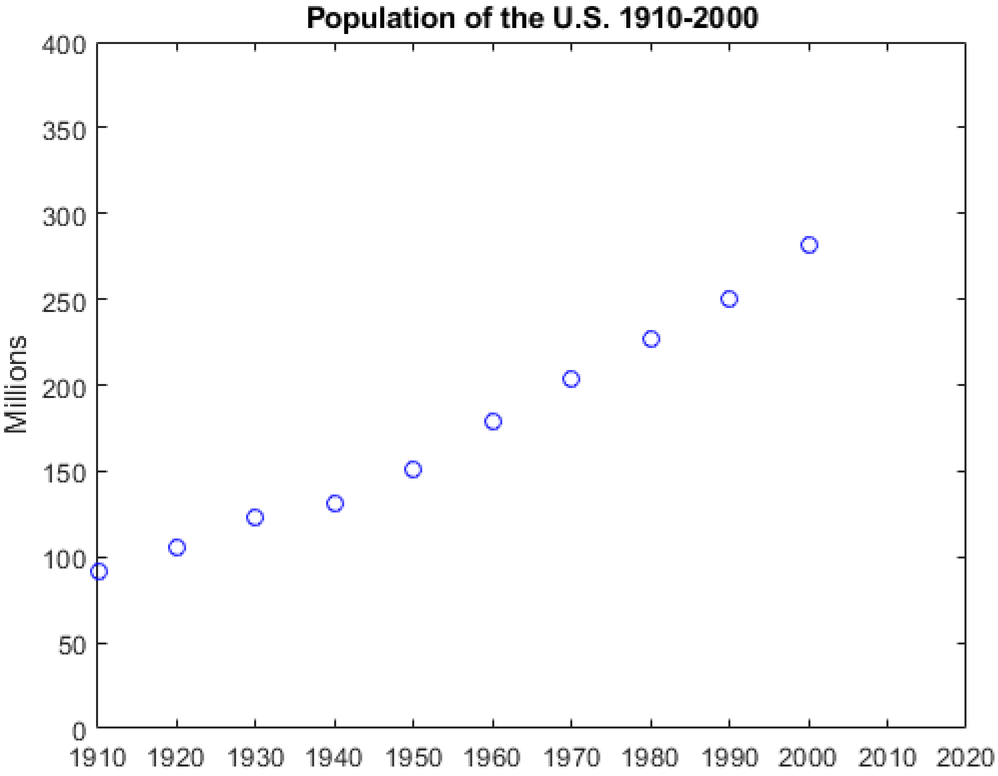
This example shows that extrapolating data using polynomials of even modest degree is risky and unreliable.

This example is older than MATLAB®. It started as an exercise in *Computer Methods for Mathematical Computations*, by Forsythe, Malcolm and Moler, published by Prentice-Hall in 1977.

Now, MATLAB makes it much easier to vary the parameters and see the results, but the underlying mathematical principles are unchanged.

Create and plot two vectors with US Census data from 1910 to 2000.

```
% Time interval
t = (1910:10:2000)';

% Population
p = [91.972 105.711 123.203 131.669 150.697...
    179.323 203.212 226.505 249.633 281.422]';

% Plot
plot(t,p,'bo');
axis([1910 2020 0 400]);
title('Population of the U.S. 1910-2000');
ylabel('Millions');
```



What is your guess for the population in the year 2010?

```
p
```

```
p = 10×1

   91.9720
  105.7110
  123.2030
  131.6690
  150.6970
  179.3230
  203.2120
  226.5050
  249.6330
  281.4220
```

Fit the data with a polynomial in `t` and use it to extrapolate the population at `t = 2010`. Obtain the coefficients in the polynomial by solving a linear system of equations involving an 11-by-11 Vandermonde matrix, with elements as powers of scaled time, `A(i,j) = s(i)^(n-j)`.

```
n = length(t);
s = (t-1950)/50;
A = zeros(n);
A(:,end) = 1;
for j = n-1:-1:1
   A(:,j) = s .* A(:,j+1);
end
```

Obtain the coefficients `c` for a polynomial of degree `d` that fits the data `p` by solving a linear system of equations involving the last `d+1` columns of the Vandermonde matrix:

```
A(:,n-d:n)*c ~= p
```

- If `d < 10`, then more equations than unknowns exist, and a least-squares solution is appropriate.
- If `d == 10`, then you can solve the equations exactly and the polynomial actually interpolates the data.

In either case, use the backslash operator to solve the system. The coefficients for the cubic fit are:

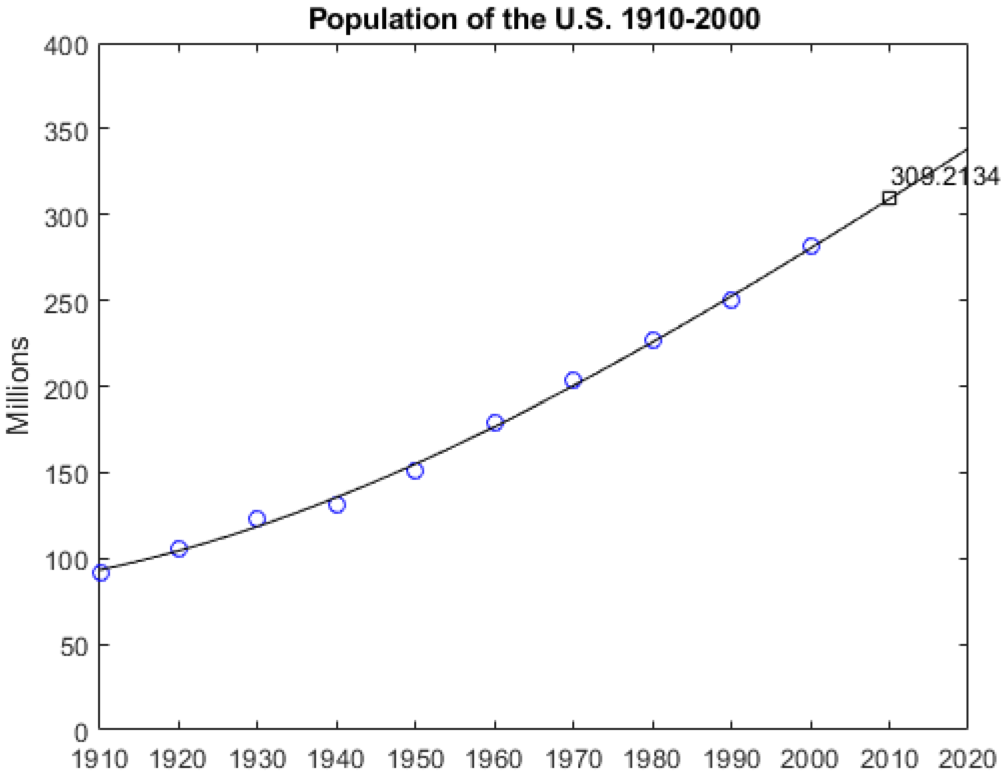
```
c = A(:,n-3:n)\p
```

```
c = 4×1

   -5.7042
   27.9064
  103.1528
  155.1017
```

Now evaluate the polynomial at every year from 1910 to 2010 and plot the results.

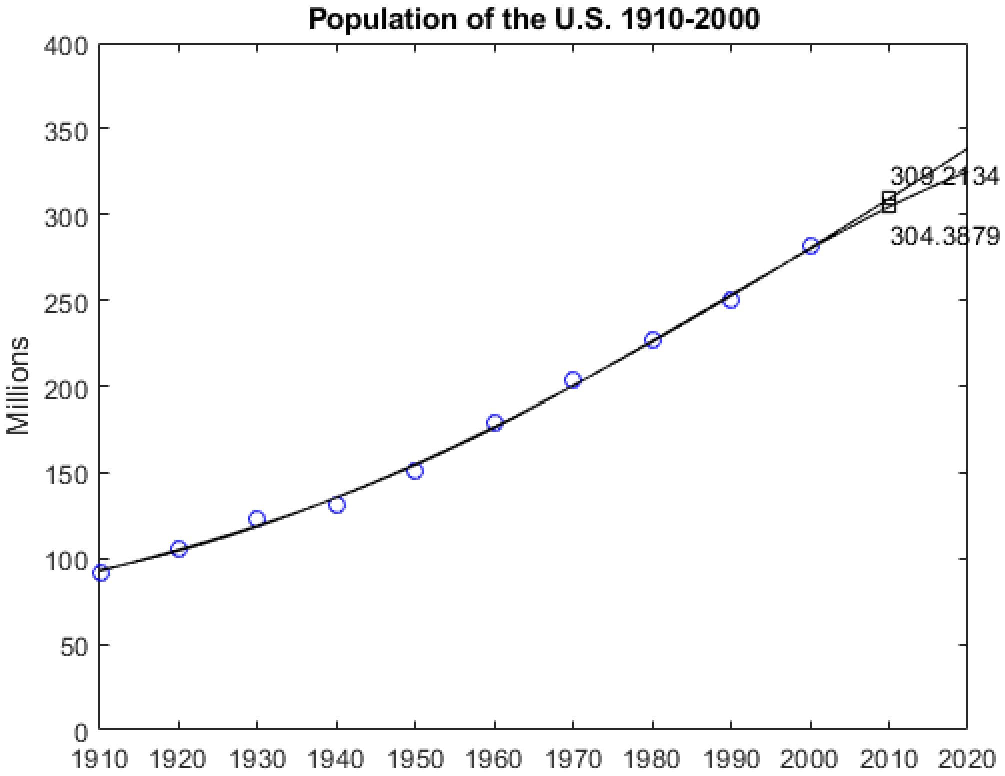
```
v = (1910:2020)';
x = (v-1950)/50;
w = (2010-1950)/50;
y = polyval(c,x);
z = polyval(c,w);

hold on
```

```
plot(v,y,'k-');
plot(2010,z,'ks');
text(2010,z+15,num2str(z));
hold off
```



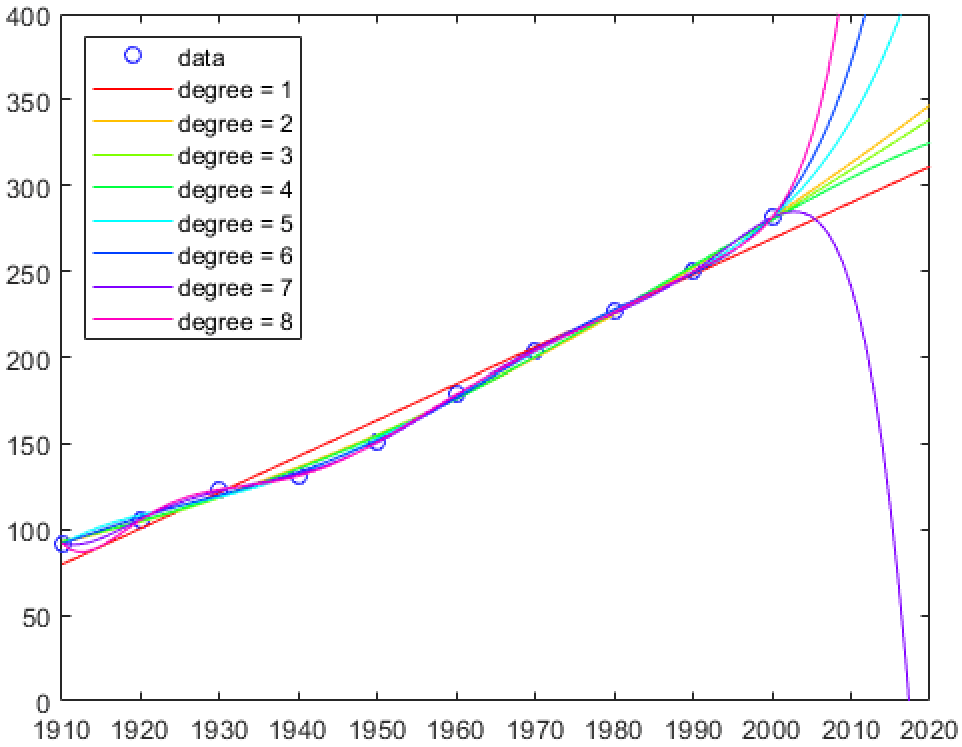
Compare the cubic fit with the quartic. Notice that the extrapolated point is very different.

```
c = A(:,n-4:n)\p;
y = polyval(c,x);
z = polyval(c,w);

hold on
plot(v,y,'k-');
plot(2010,z,'ks');
text(2010,z-15,num2str(z));
hold off
```



As the degree increases, the extrapolation becomes even more erratic.

```
cla
plot(t,p,'bo')
hold on
axis([1910 2020 0 400])
colors = hsv(8);
labels = {'data'};
for d = 1:8
   [Q,R] = qr(A(:,n-d:n));
   R = R(1:d+1,:);
   Q = Q(:,1:d+1);
   c = R\(Q'*p);    % Same as c = A(:,n-d:n)\p;
   y = polyval(c,x);
   z = polyval(c,11);
   plot(v,y,'color',colors(d,:));
   labels{end+1} = ['degree = ' int2str(d)];
end
legend(labels, 'Location', 'NorthWest')
hold off
```



See Also

`polyfit`

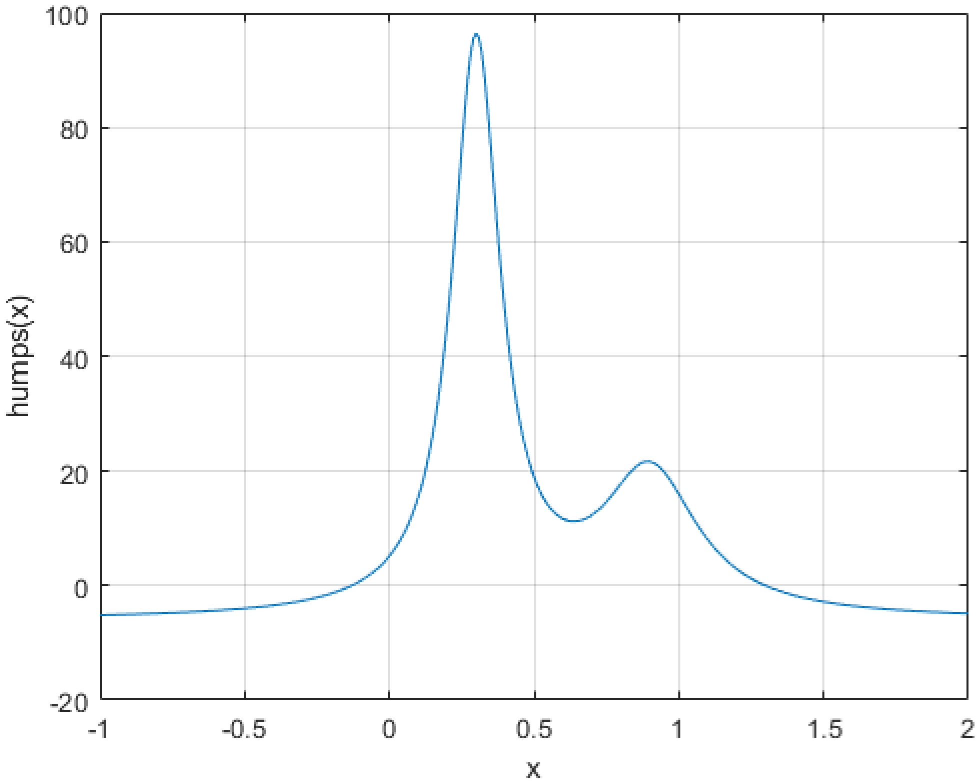
Roots of Scalar Functions

Solving a Nonlinear Equation in One Variable

The `fzero` function attempts to find a root of one equation with one variable. You can call this function with either a one-element starting point or a two-element vector that designates a starting interval. If you give `fzero` a starting point `x0`, `fzero` first searches for an interval around this point where the function changes sign. If the interval is found, `fzero` returns a value near where the function changes sign. If no such interval is found, `fzero` returns `NaN`. Alternatively, if you know two points where the function value differs in sign, you can specify this starting interval using a two-element vector; `fzero` is guaranteed to narrow down the interval and return a value near a sign change.

The following sections contain two examples that illustrate how to find a zero of a function using a starting interval and a starting point. The examples use the function `humps.m`, which is provided with MATLAB®. The following figure shows the graph of `humps`.

```
x = -1:.01:2;
y = humps(x);
plot(x,y)
xlabel('x');
ylabel('humps(x)')
grid on
```



Setting Options For fzero

You can control several aspects of the `fzero` function by setting options. You set options using `optimset`. Options include:

- Choosing the amount of display `fzero` generates — see “Set Optimization Options” on page 9-10, Using a Starting Interval on page 6-0 , and Using a Starting Point on page 6-0 .
- Choosing various tolerances that control how `fzero` determines it is at a root — see “Set Optimization Options” on page 9-10.
- Choosing a plot function for observing the progress of `fzero` towards a root — see “Optimization Solver Plot Functions” on page 9-20.
- Using a custom-programmed output function for observing the progress of `fzero` towards a root — see “Optimization Solver Output Functions” on page 9-14.

Using a Starting Interval

The graph of `humps` indicates that the function is negative at `x = -1` and positive at `x = 1`. You can confirm this by calculating `humps` at these two points.

```
humps(1)
```

```
ans = 16
```

```
humps(-1)
```

```
ans = -5.1378
```

Consequently, you can use `[-1 1]` as a starting interval for `fzero`.

The iterative algorithm for `fzero` finds smaller and smaller subintervals of `[-1 1]`. For each subinterval, the sign of `humps` differs at the two endpoints. As the endpoints of the subintervals get closer and closer, they converge to zero for `humps`.

To show the progress of `fzero` at each iteration, set the `Display` option to `iter` using the `optimset` function.

```
options = optimset('Display','iter');
```

Then call `fzero` as follows:

```
a = fzero(@humps,[-1 1],options)
```

```
 Func-count    x          f(x)             Procedure
    2              -1      -5.13779        initial
    3       -0.513876      -4.02235        interpolation
    4       -0.513876      -4.02235        bisection
    5       -0.473635      -3.83767        interpolation
    6       -0.115287      0.414441        bisection
    7       -0.115287      0.414441        interpolation
    8       -0.132562    -0.0226907        interpolation
    9       -0.131666    -0.0011492        interpolation
   10       -0.131618   1.88371e-07        interpolation
   11       -0.131618   -2.7935e-11        interpolation
   12       -0.131618   8.88178e-16        interpolation
   13       -0.131618   8.88178e-16        interpolation
```

```
Zero found in the interval [-1, 1]

a = -0.1316
```

Each value x represents the best endpoint so far. The Procedure column tells you whether each step of the algorithm uses bisection or interpolation.

You can verify that the function value at a is close to zero by entering

```
humps(a)
```

```
ans = 8.8818e-16
```

Using a Starting Point

Suppose you do not know two points at which the function values of humps differ in sign. In that case, you can choose a scalar x0 as the starting point for fzero. fzero first searches for an interval around this point on which the function changes sign. If fzero finds such an interval, it proceeds with the algorithm described in the previous section. If no such interval is found, fzero returns NaN.

For example, set the starting point to -0.2, the Display option to Iter, and call fzero:

```
options = optimset('Display','iter');
a = fzero(@humps,-0.2,options)
```

```
Search for an interval around -0.2 containing a sign change:
 Func-count    a          f(a)             b          f(b)        Procedure
    1            -0.2      -1.35385          -0.2      -1.35385   initial interval
    3       -0.194343      -1.26077     -0.205657      -1.44411   search
    5          -0.192      -1.22137        -0.208       -1.4807   search
    7       -0.188686      -1.16477     -0.211314      -1.53167   search
    9          -0.184      -1.08293        -0.216      -1.60224   search
   11       -0.177373     -0.963455     -0.222627      -1.69911   search
   13          -0.168     -0.786636        -0.232      -1.83055   search
   15       -0.154745      -0.51962     -0.245255      -2.00602   search
   17          -0.136     -0.104165        -0.264      -2.23521   search
   18        -0.10949      0.572246        -0.264      -2.23521   search

Search for a zero in the interval [-0.10949, -0.264]:
 Func-count    x          f(x)             Procedure
   18        -0.10949      0.572246        initial
   19       -0.140984     -0.219277        interpolation
   20       -0.132259    -0.0154224        interpolation
   21       -0.131617   3.40729e-05        interpolation
   22       -0.131618  -6.79505e-08        interpolation
   23       -0.131618  -2.98428e-13        interpolation
   24       -0.131618   8.88178e-16        interpolation
   25       -0.131618   8.88178e-16        interpolation

Zero found in the interval [-0.10949, -0.264]

a = -0.1316
```

The endpoints of the current subinterval at each iteration are listed under the headings a and b, while the corresponding values of humps at the endpoints are listed under f(a) and f(b), respectively.

Note: The endpoints a and b are not listed in any specific order: a can be greater than b or less than b.

For the first nine steps, the sign of humps is negative at both endpoints of the current subinterval, which is shown in the output. At the tenth step, the sign of humps is positive at a, -0.10949, but negative at b, -0.264. From this point on, the algorithm continues to narrow down the interval [-0.10949 -0.264], as described in the previous section, until it reaches the value -0.1316.

See Also

More About

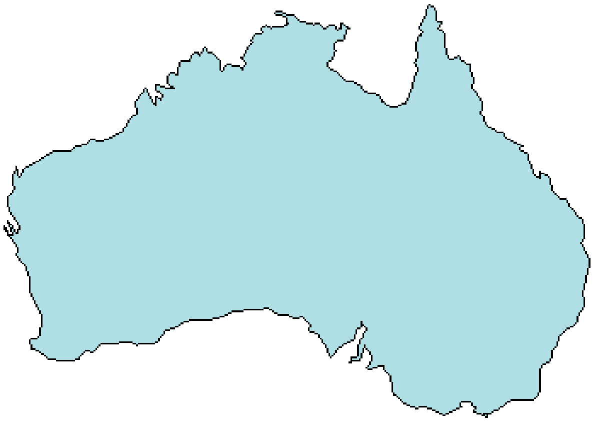
- “Roots of Polynomials” on page 6-4
- “Optimizing Nonlinear Functions” on page 9-2
- “Systems of Nonlinear Equations” (Optimization Toolbox)

Computational Geometry

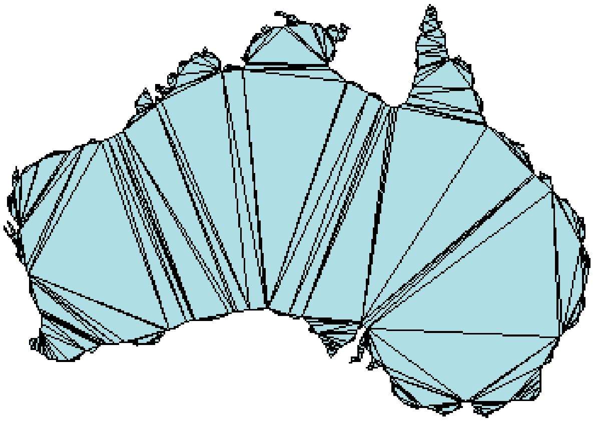
Triangulation Representations

2-D and 3-D Domains

Triangulations are often used to represent 2-D and 3-D geometric domains in application areas such as computer graphics, physical modeling, geographic information systems, medical imaging, and more. The map polygon shown here

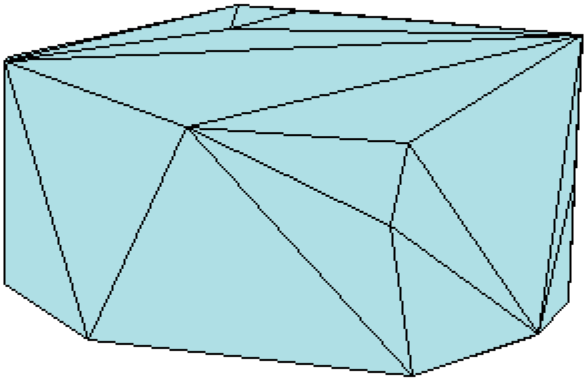


can be represented by the triangulation on the map shown below.



The triangulation decomposes a complex polygon into a collection of simpler triangular polygons. You can use these polygons for developing geometric-based algorithms or graphics applications.

Similarly, you can represent the boundary of a 3-D geometric domain using a triangulation. The figure below shows the convex hull of a set of points in 3-D space. Each facet of the hull is a triangle.

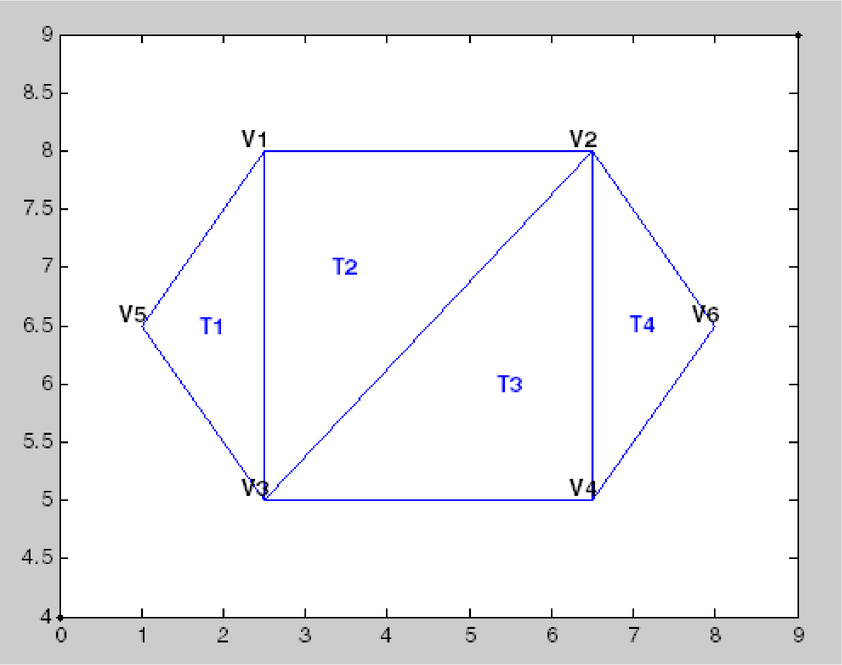


Triangulation Matrix Format

MATLAB uses a matrix format to represent triangulations. This format has two parts:

- The vertices, represented as a matrix in which each row contains the coordinates of a point in the triangulation.
- The triangulation connectivity, represented as a matrix in which each row defines a triangle or tetrahedron.

This figure shows a simple 2-D triangulation.



The following table shows the vertex information.

Vertices		
Vertex ID	*x*-coordinate	*y*-coordinate

Vertices		
V1	2.5	8.0
V2	6.5	8.0
V3	2.5	5.0
V4	6.5	5.0
V5	1.0	6.5
V6	8.0	6.5

The data in the previous table is stored as a matrix in the MATLAB environment. The vertex IDs are labels used for identifying specific vertices. They are shown to illustrate the concept of a vertex ID, but they are not stored explicitly. Instead, the row numbers of the matrix serve as the vertex IDs.

The triangulation connectivity data is shown in this table.

Connectivity			
Triangle ID	IDs of Bounding Vertices		
T1	5	3	1
T2	3	2	1
T3	3	4	2
T4	4	6	2

The data in this table is stored as a matrix in the MATLAB environment. The triangle IDs are labels used for identifying specific triangles. They are shown to illustrate the concept of a triangle ID, but they are not stored explicitly. Instead, the row numbers of the matrix serve as the triangle IDs.

You can see that triangle `T1` is defined by three vertices, `{V5, V3, V1}`. Similarly, `T4` is defined by the vertices, `{V4, V6, V2}`. This format extends naturally to higher dimensions, which require additional columns of data. For example, a tetrahedron in 3-D space is defined by four vertices, each of which have three coordinates, (x, y, z).

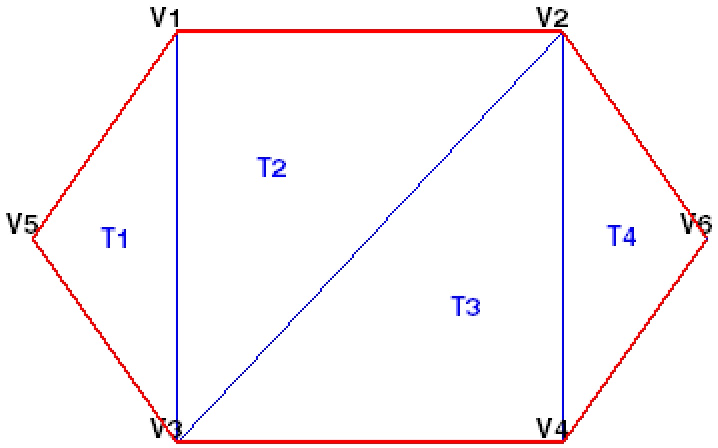
You can represent and query the following types of triangulations using MATLAB:

- 2-D triangulations consisting of triangles bounded by vertices and edges
- 3-D surface triangulations consisting of triangles bounded by vertices and edges
- 3-D triangulations consisting of tetrahedra bounded by vertices, edges, and faces

Querying Triangulations Using the triangulation Class

The matrix format provides a compact low-level, array-based representation for triangulations. When you use triangulations to develop algorithms, you might need more information about the geometric properties, topology, and adjacency information.

For example, you might compute the triangle incenters before plotting the annotated triangulation shown below. In this case, you use the incenters to display the triangle labels (T1, T2, etc.) within each triangle. If you want to plot the boundary in red, you need to determine the edges that are referenced by only one triangle.



The triangulation Class

You can use `triangulation` to create an in-memory representation of any 2-D or 3-D triangulation data that is in matrix format, such as the matrix output from the `delaunay` function or other software tools. When your data is represented using `triangulation`, you can perform topological and geometric queries, which you can use to develop geometric algorithms. For example, you can find the triangles or tetrahedra attached to a vertex, those that share an edge, their circumcenters, and other features.

You can create a `triangulation` in one of two ways:

- Pass existing data that you have in matrix format to `triangulation`. This data can be the output from a MATLAB function, such as `delaunay` or `convhull`. You also can import triangulation data that was created by another software application. When you work with imported data, be sure the connectivity data references the vertex array using 1-based indexing instead of 0-based indexing.
- Pass a set of points to `delaunayTriangulation`. The resulting Delaunay triangulation is a special kind of `triangulation`. This means you can perform any `triangulation` query on your data, as well as any Delaunay-specific query. In more formal MATLAB language terms, `delaunayTriangulation` is a subclass of `triangulation`.

Creating a triangulation from Matrix Data

This example shows how to use the triangulation matrix data to create a triangulation, explore what it is, and explore what it can do.

Create a matrix, `P`, that contains the vertex data.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];
```

Define the connectivity, `T`.

```
T = [5  3  1;
     3  2  1;
     3  4  2;
     4  6  2];
```

Create a `triangulation` from this data.

```
TR = triangulation(T,P)
```

```
TR = 
  triangulation with properties:

              Points: [6x2 double]
    ConnectivityList: [4x3 double]
```

Access the properties in a `triangulation` in the same way you access the fields of a `struct`. For example, examine the `Points` property, which contains the coordinates of the vertices.

```
TR.Points
```

```
ans = 6×2

    2.5000    8.0000
    6.5000    8.0000
    2.5000    5.0000
    6.5000    5.0000
    1.0000    6.5000
    8.0000    6.5000
```

Next, examine the connectivity.

```
TR.ConnectivityList
```

```
ans = 4×3

     5     3     1
     3     2     1
     3     4     2
     4     6     2
```

The `Points` and `ConnectivityList` properties define the matrix data for the triangulation.

The `triangulation` class is a wrapper around the matrix data. The real benefit is the usefulness of the `triangulation` class methods. The methods are like functions that accept a `triangulation` and other relevant input data.

The `triangulation` class provides an easy way to index into the `ConnectivityList` property matrix. Access the first triangle in the triangulation.

```
TR.ConnectivityList(1,:)
```

```
ans = 1×3

     5     3     1
```

Another way of getting the first triangle is `TR(1,:)`.

Examine the first vertex of the first triangle.

```
TR(1,1)
```

```
ans = 5
```

Examine the second vertex of the first triangle.

```
TR(1,2)
```

```
ans = 3
```

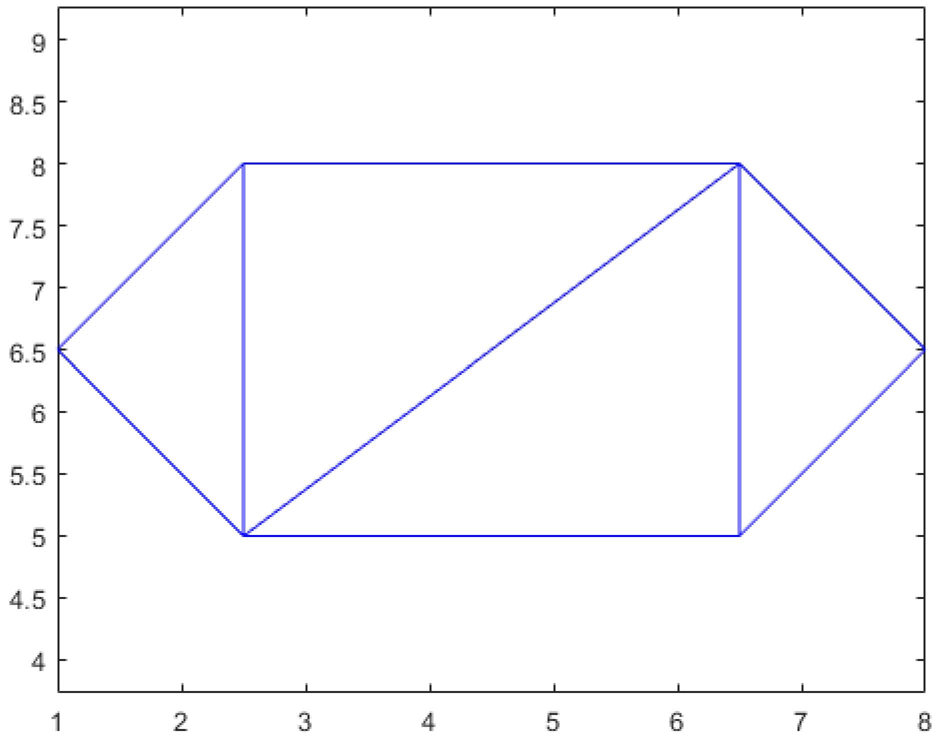
Now, examine all the triangles in the triangulation.

```
TR(:,:)
```

```
ans = 4×3

     5     3     1
     3     2     1
     3     4     2
     4     6     2
```

Use `triplot` to plot the `triangulation`. The `triplot` function is not a `triangulation` method, but it accepts and can plot a `triangulation`.

```
figure
triplot(TR)
axis equal
```

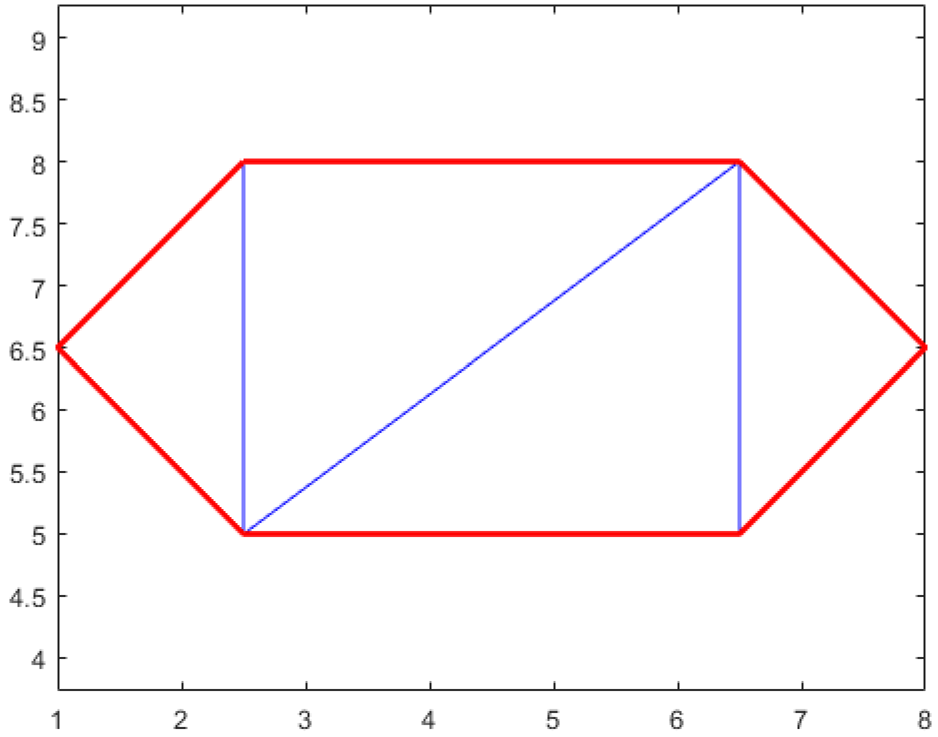


Use the `triangulation` method, `freeBoundary`, to query the free boundary and highlight it in a plot. This method returns the edges of the triangulation that are shared by only one triangle. The returned edges are expressed in terms of the vertex IDs.

```
boundaryedges = freeBoundary(TR)';
```

Now plot the boundary edges as a red line.

```
hold on
plot(P(boundaryedges,1),P(boundaryedges,2),'-r','LineWidth',2)
hold off
```



You can use the `freeBoundary` method to validate a triangulation. For example, if you observed red edges in the interior of the triangulation, then it would indicate a problem in how the triangles are connected.

Creating a triangulation Using delaunayTriangulation

This example shows how to create a Delaunay triangulation using `delaunayTriangulation`.

When you create a Delaunay triangulation using the `delaunayTriangulation` class, you automatically get access to the `triangulation` methods because `delaunayTriangulation` is a subclass of `triangulation`.

Create a `delaunayTriangulation` from a set of points.

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

DT = delaunayTriangulation(P)
```

```
DT = 
  delaunayTriangulation with properties:

              Points: [6x2 double]
```

```
    ConnectivityList: [4x3 double]
         Constraints: []
```

The resulting `delaunayTriangulation` object has the properties, `Points` and `ConnectivityList`, just like a `triangulation` object.

You can access the triangulation using direct indexing, just like `triangulation`. For example, examine the connectivity of the first triangle.

```
DT(1,:)
```

```
ans = 1×3

     5     3     1
```

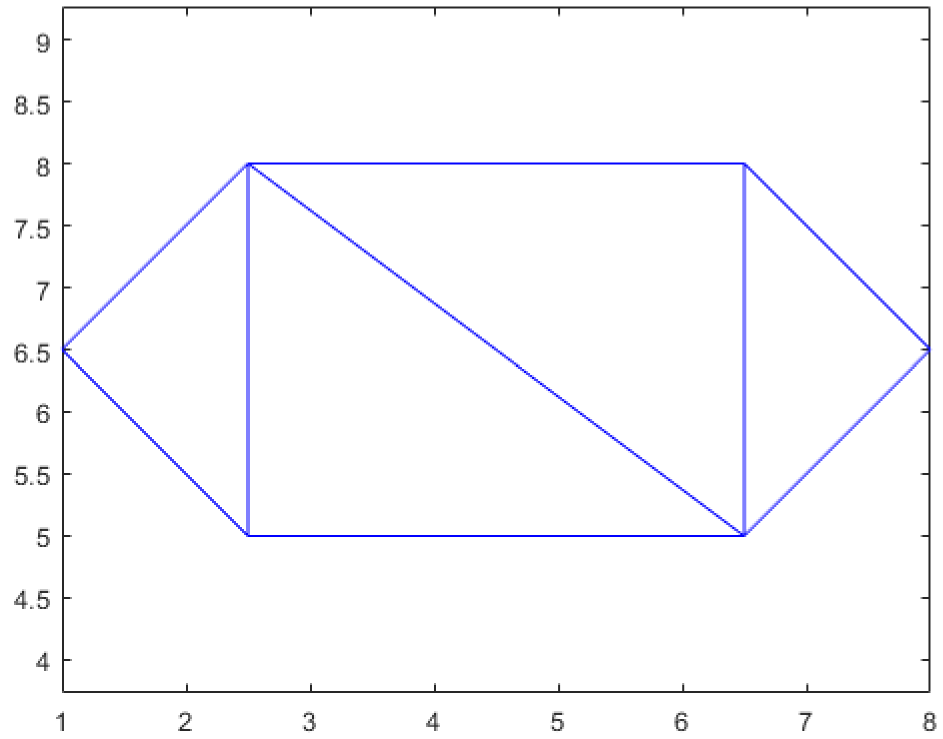
Next, examine the connectivity of the entire triangulation.

```
DT(:,:)
```

```
ans = 4×3

     5     3     1
     3     4     1
     1     4     2
     4     6     2
```

Use the `triplot` function to plot the triangulation.

```
triplot(DT)
axis equal
```



The parent class, `triangulation`, provides the `incenter` method to compute the incenters of each triangle.

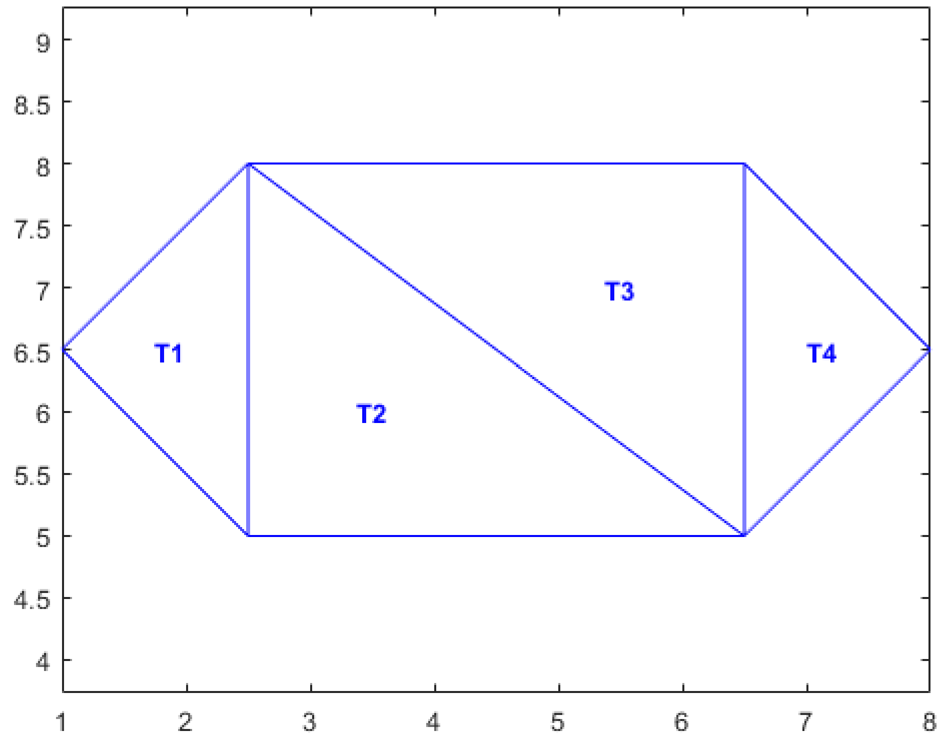
```
IC = incenter(DT)
```

```
IC = 4×2

    1.8787    6.5000
    3.5000    6.0000
    5.5000    7.0000
    7.1213    6.5000
```

The returned value, `IC`, is an array of coordinates representing the incenters of the triangles.

Now, use the incenters to find the positions for placing triangle labels on the plot.

```
hold on
numtri = size(DT,1);
trilabels = arrayfun(@(P) {sprintf('T%d', P)}, (1:numtri)');
Htl = text(IC(:,1),IC(:,2),trilabels,'FontWeight','bold', ...
'HorizontalAlignment','center','Color','blue');
hold off
```



Instead of creating a Delaunay triangulation using `delaunayTriangulation`, you could use the `delaunay` function to create the triangulation connectivity data, and then pass the connectivity data to `triangulation`. For example,

```
P = [ 2.5    8.0
      6.5    8.0
      2.5    5.0
      6.5    5.0
      1.0    6.5
      8.0    6.5];

T = delaunay(P);
TR = triangulation(T,P);
IC = incenter(TR);
```

Both approaches are valid in this example, but if you want to create a Delaunay triangulation and perform queries on it, then you should use `delaunayTriangulation` for these reasons:

- The `delaunayTriangulation` class provides additional methods that are useful for working with triangulations. For example, you can to perform nearest-neighbor and point-in-triangle searches.
- It allows you to edit the triangulation to add, move, or remove points.
- It allows you to create constrained Delaunay triangulations. This allows you to create a triangulation for a 2-D domain.

See Also

`delaunay` | `delaunayTriangulation` | `freeBoundary` | `triangulation` | `triplot`

More About

- “Triangulation Representations” on page 7-2
- “Working with Delaunay Triangulations” on page 7-14
- “Spatial Searching” on page 7-53

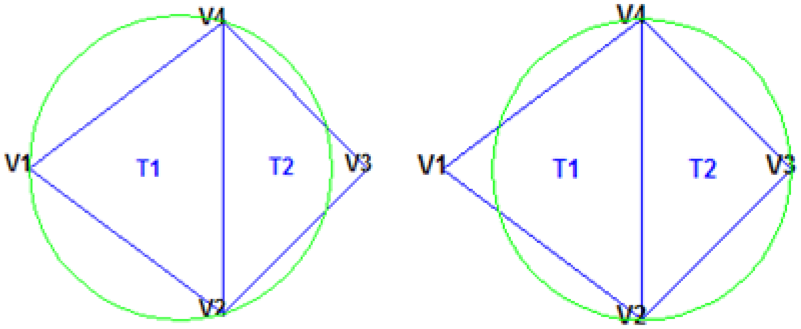
Working with Delaunay Triangulations

In this section...

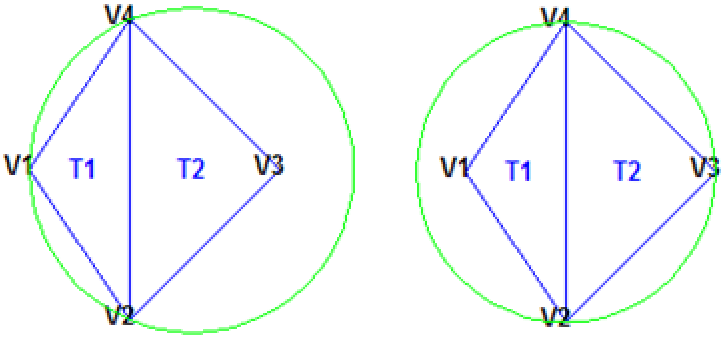
Definition of Delaunay Triangulation

Delaunay triangulations are widely used in scientific computing in many diverse applications. While there are numerous algorithms for computing triangulations, it is the favorable geometric properties of the Delaunay triangulation that make it so useful.

The fundamental property is the Delaunay criterion. In the case of 2-D triangulations, this is often called the empty circumcircle criterion. For a set of points in 2-D, a Delaunay triangulation of these points ensures the circumcircle associated with each triangle contains no other point in its interior. This property is important. In the illustration below, the circumcircle associated with T1 is empty. It does not contain a point in its interior. The circumcircle associated with T2 is empty. It does not contain a point in its interior. This triangulation is a Delaunay triangulation.

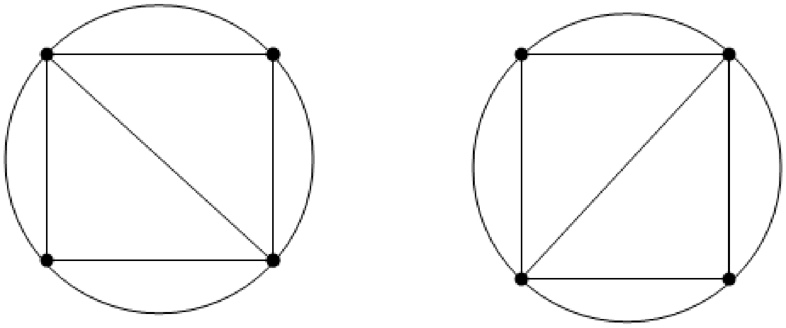


The triangles below are different. The circumcircle associated with T1 is not empty. It contains V3 in its interior. The circumcircle associated with T2 is not empty. It contains V1 in its interior. This triangulation is *not* a Delaunay triangulation.

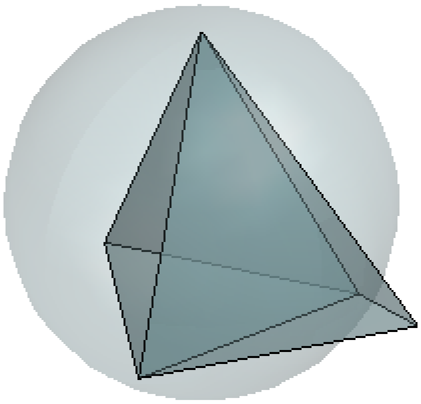


Delaunay triangles are said to be "well shaped" because in fulfilling the empty circumcircle property, triangles with large internal angles are selected over ones with small internal angles. The triangles in the non-Delaunay triangulation have sharp angles at vertices V2 and V4. If the edge {V2, V4} were replaced by an edge joining V1 and V3, the minimum angle would be maximized and the triangulation would become a Delaunay triangulation. Also, the Delaunay triangulation connects points in a nearest-neighbor manner. These two characteristics, well-shaped triangles and the nearest-neighbor relation, have important implications in practice and motivate the use of Delaunay triangulations in scattered data interpolation.

While the Delaunay property is well defined, the topology of the triangulation is not unique in the presence of degenerate point sets. In two dimensions, degeneracies arise when four or more unique points lie on the same circle. The vertices of a square, for example, have a nonunique Delaunay triangulation.



The properties of Delaunay triangulations extend to higher dimensions. The triangulation of a 3-D set of points is composed of tetrahedra. The next illustration shows a simple 3-D Delaunay triangulation made up of two tetrahedra. The circumsphere of one tetrahedron is shown to highlight the empty circumsphere criterion.



A 3-D Delaunay triangulation produces tetrahedra that satisfy the empty circumsphere criterion.

Creating Delaunay Triangulations

MATLAB provides two ways to create Delaunay triangulations:

- The functions delaunay and delaunayn

- The delaunayTriangulation class

The delaunay function supports the creation of 2-D and 3-D Delaunay triangulations. The delaunayn function supports creating Delaunay triangulations in 4-D and higher.

Tip Creating Delaunay triangulations in dimensions higher than 6-D is generally not practical for moderate to large point sets due to the exponential growth in required memory.

The delaunayTriangulation class supports creating Delaunay triangulations in 2-D and 3-D. It provides many methods that are useful for developing triangulation-based algorithms. These class methods are like functions, but they are restricted to work with triangulations created using delaunayTriangulation. The delaunayTriangulation class also supports the creation of related constructs such as the convex hull and Voronoi diagram. It also supports the creation of constrained Delaunay triangulations.

In summary:

- The delaunay function is useful when you only require the basic triangulation data, and that data is sufficiently complete for your application.
- The delaunayTriangulation class offers more functionality for developing triangulation-based applications. It is useful when you require the triangulation and you want to perform any of these operations:
  - Search the triangulation for triangles or tetrahedra enclosing a query point.
  - Use the triangulation to perform a nearest-neighbor point search.
  - Query the triangulation's topological adjacency or geometric properties.
  - Modify the triangulation to insert or remove points.
  - Constrain edges in the triangulation—this is called a constrained Delaunay triangulation.
  - Triangulate a polygon and optionally remove the triangles that are outside of the domain.
  - Use the Delaunay triangulation to compute the convex hull or Voronoi diagram.

Using the delaunay and delaunayn functions

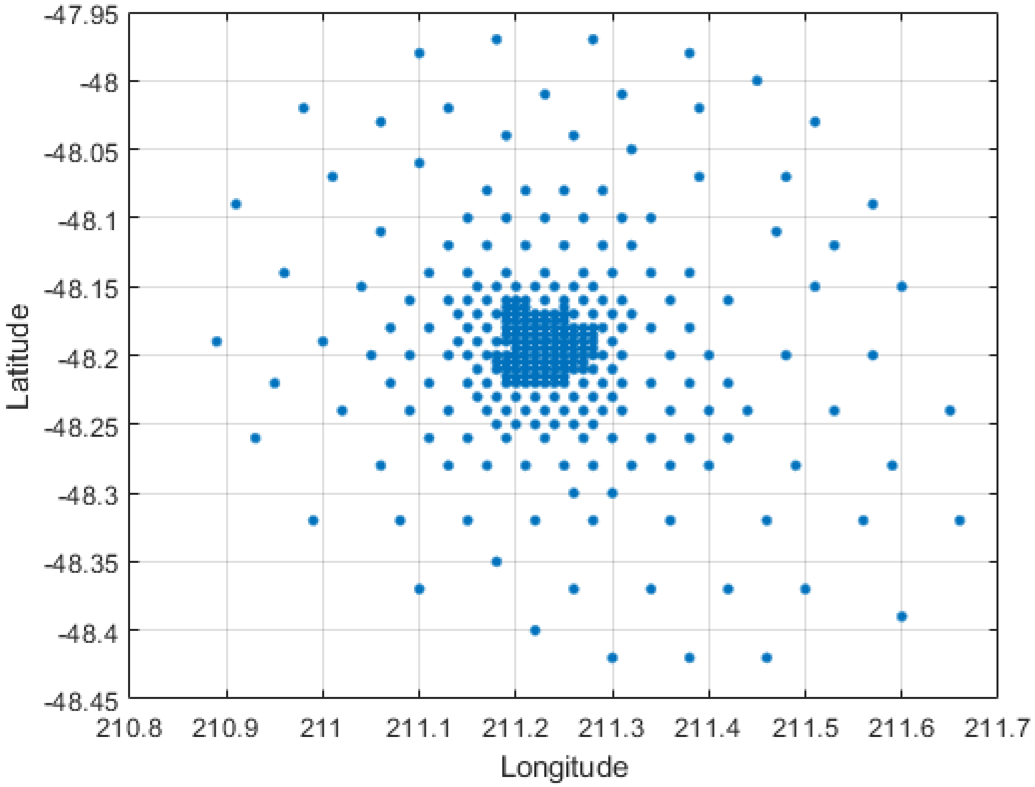
The delaunay and delaunayn functions take a set of points and produce a triangulation in matrix format. Refer to “Triangulation Matrix Format” on page 7-3 for more information on this data structure. In 2-D, the delaunay function is often used to produce a triangulation that can be used to plot a surface defined in terms of a set of scattered data points. In this application, it’s important to note that this approach can only be used if the surface is single-valued. For example, it could not be used to plot a spherical surface because there are two z values corresponding to a single (x, y) coordinate. A simple example demonstrates how the delaunay function can be used to plot a surface representing a sampled data set.

This example shows how to use the delaunay function to create a 2-D Delaunay triangulation from the seamount data set. A *seamount* is an underwater mountain. The data set consists of a set of longitude (x) and latitude (y) locations, and corresponding seamount elevations (z) measured at those coordinates.

Load the seamount data set and view the (x, y) data as a scatter plot.

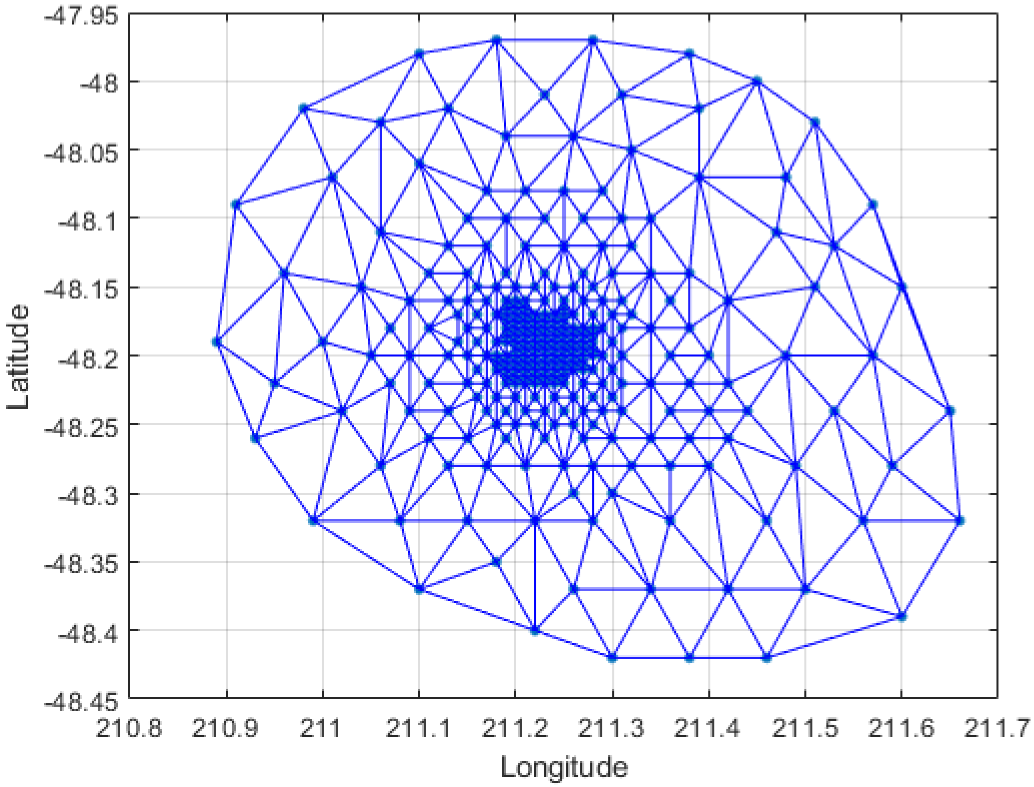
```
load seamount
plot(x,y,'.','markersize',12)
```

```
xlabel('Longitude'), ylabel('Latitude')
grid on
```



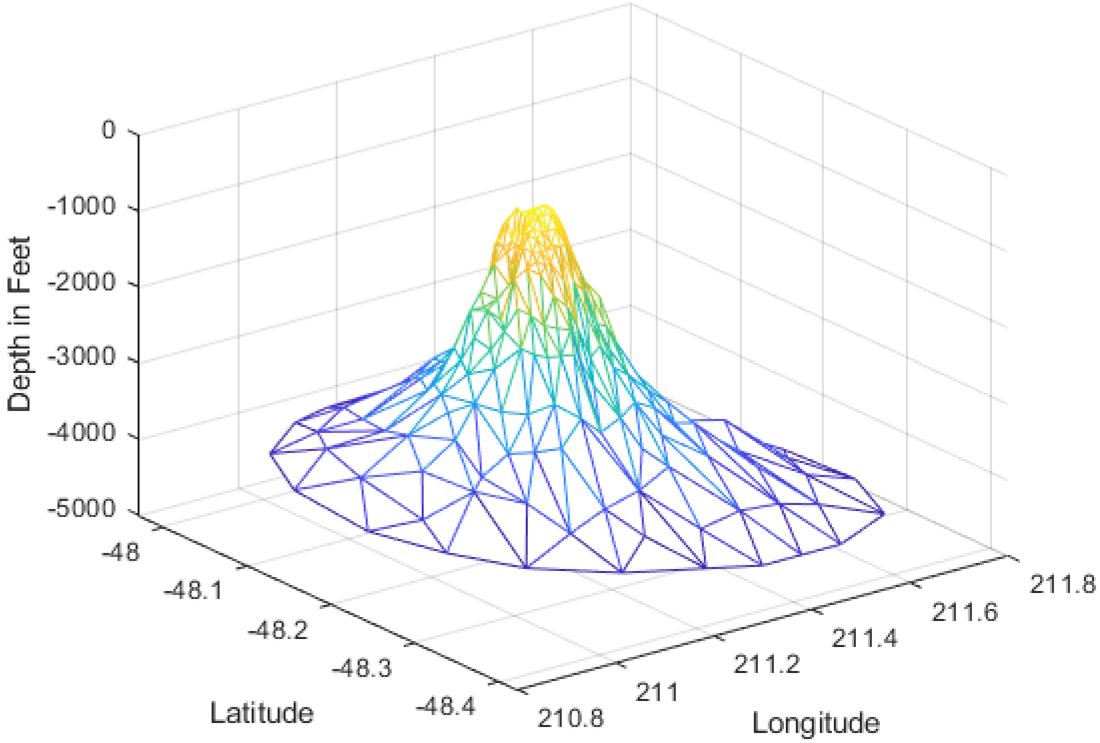
Construct a Delaunay triangulation from this point set and use `triplot` to plot the triangulation in the existing figure.

```
tri = delaunay(x,y);
hold on, triplot(tri,x,y), hold off
```



Add the depth data (`z`) from seamount to lift the vertices and create the surface. Create a new figure and use `trimesh` to plot the surface in wireframe mode.

```
figure
hidden on
trimesh(tri,x,y,z)
xlabel('Longitude'),ylabel('Latitude'),zlabel('Depth in Feet');
```

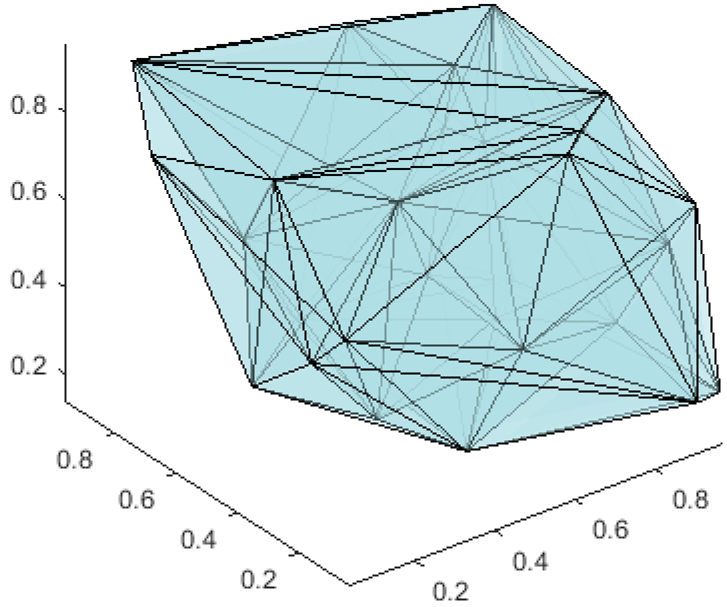


If you want to plot the surface in shaded mode, use `trisurf` instead of `trimesh`.

A 3-D Delaunay triangulation also can be created using the `delaunay` function. This triangulation is composed of tetrahedra.

This example shows how to create a 3-D Delaunay triangulation of a random data set. The triangulation is plotted using `tetramesh`, and the `FaceAlpha` option adds transparency to the plot.

```
rng('default')
X = rand([30 3]);
tet = delaunay(X);
faceColor  = [0.6875 0.8750 0.8984];
tetramesh(tet,X,'FaceColor', faceColor,'FaceAlpha',0.3);
```



MATLAB provides the delaunayn function to support the creation of Delaunay triangulations in dimension 4-D and higher. Two complementary functions tsearchn and dsearchn are also provided to support spatial searching for N-D triangulations. See “Spatial Searching” on page 7-53 for more information on triangulation-based search.

Using the delaunayTriangulation Class

The delaunayTriangulation class provides another way to create Delaunay triangulations in MATLAB. While delaunay and delaunayTriangulation use the same underlying algorithm and produce the same triangulation, delaunayTriangulation provides complementary methods that are useful for developing Delaunay-based algorithms. These methods are like functions that are packaged together with the triangulation data into a container called a class. Keeping everything together in a class provides a more organized setup that improves ease of use. It also improves the performance of triangulation-based searches such as point-location and nearest-neighbor. delaunayTriangulation supports incremental editing of the Delaunay triangulation. You also can impose edge constraints in 2-D.

“Triangulation Representations” on page 7-2 introduces the triangulation class, which supports topological and geometric queries for 2-D and 3-D triangulations. A delaunayTriangulation is a special kind of triangulation. This means you can perform any triangulation query on a delaunayTriangulation in addition to the Delaunay-specific queries. In more formal MATLAB language terms, delaunayTriangulation is a subclass of triangulation.

This example shows how to create, query, and edit a Delaunay triangulation from the seamount data using delaunayTriangulation. The seamount data set contains (x, y) locations and corresponding elevations (z) that define the surface of the seamount.

Load and triangulate the (x, y) data.

```
load seamount
DT = delaunayTriangulation(x,y)
```

```
DT = 
  delaunayTriangulation with properties:

              Points: [294x2 double]
    ConnectivityList: [566x3 double]
         Constraints: []
```

The `Constraints` property is empty because there aren't any imposed edge constraints. The `Points` property represents the coordinates of the vertices, and the `ConnectivityList` property represents the triangles. Together, these two properties define the matrix data for the triangulation.

The `delaunayTriangulation` class is a wrapper around the matrix data, and it offers a set of complementary methods. You access the properties in a `delaunayTriangulation` in the same way you access the fields of a struct.

Access the vertex data.

```
DT.Points;
```

Access the connectivity data.

```
DT.ConnectivityList;
```

Access the first triangle in the `ConnectivityList` property.

```
DT.ConnectivityList(1,:)
```

```
ans = 1×3

   205   230   262
```

`delaunayTriangulation` provides an easy way to index into the `ConnectivityList` property matrix.

Access the first triangle.

```
DT(1,:)
```

```
ans = 1×3

   205   230   262
```

Examine the first vertex of the first triangle.

```
DT(1,1)
```

```
ans = 205
```

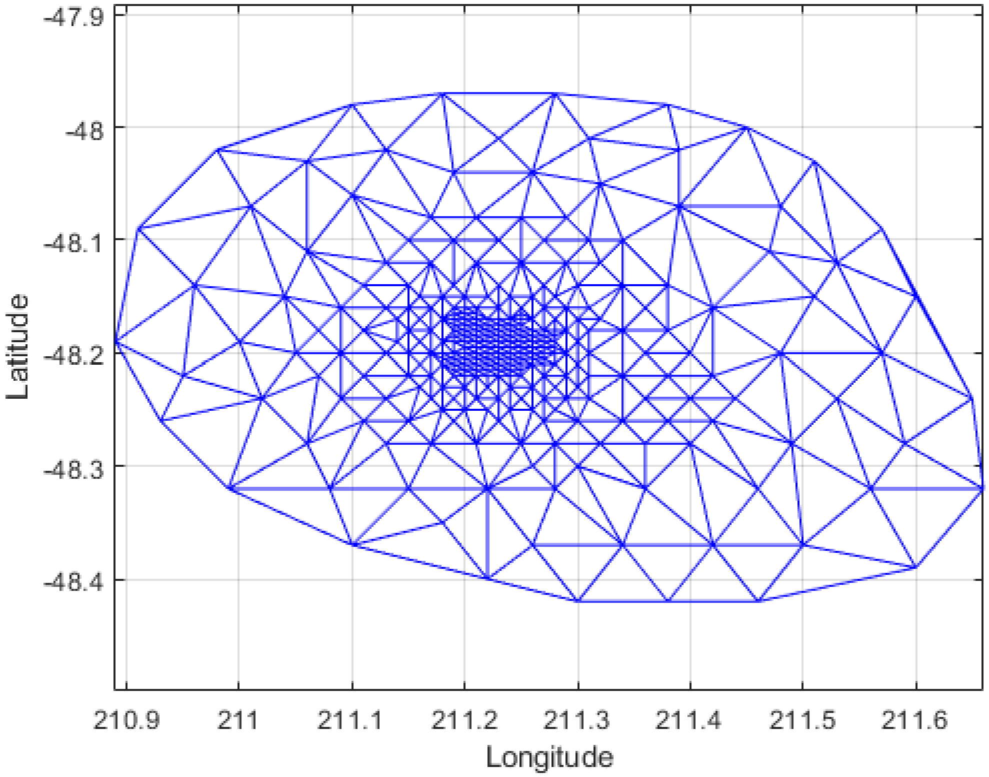
Examine all the triangles in the triangulation.

```
DT(:,:);
```

Indexing into the delaunayTriangulation output, DT, works like indexing into the triangulation array output from delaunay. The difference between the two are the extra methods that you can call on DT (for example, nearestNeighbor and pointLocation).

Use triplot to plot the delaunayTriangulation. The triplot function is not a delaunayTriangulation method, but it accepts and can plot a delaunayTriangulation.

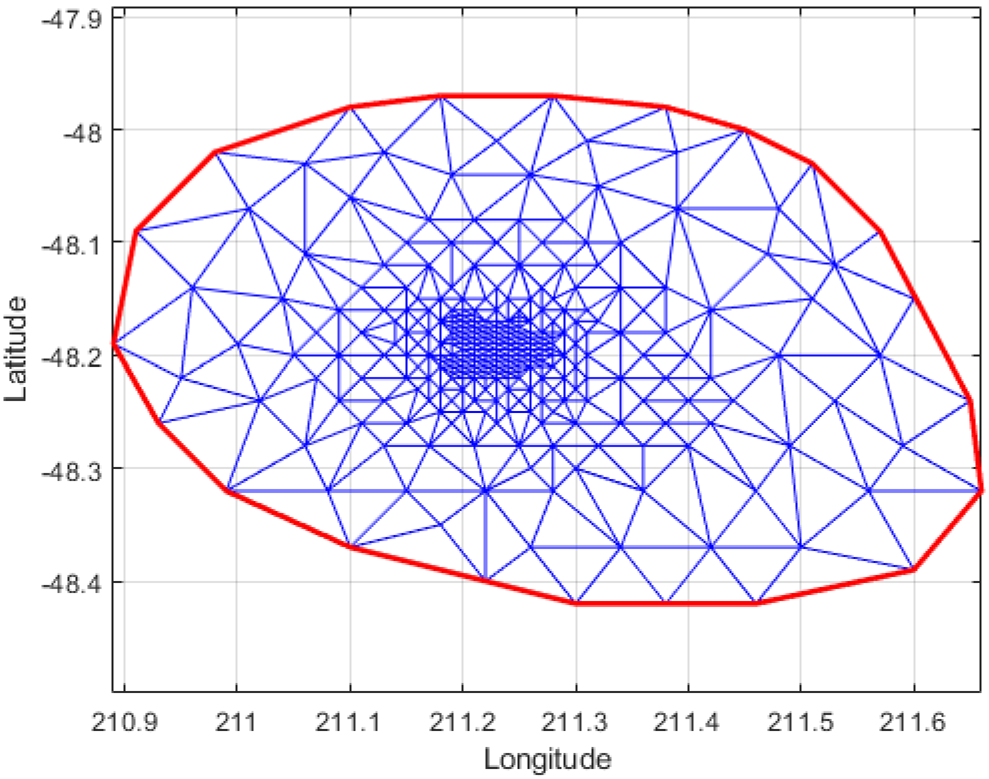
```
triplot(DT);
axis equal
xlabel('Longitude'), ylabel('Latitude')
grid on
```



Alternatively, you could use triplot(DT(:,:), DT.Points(:,1), DT.Points(:,2)); to get the same plot.

Use the delaunayTriangulation method, convexHull, to compute the convex hull and add it to the plot. Since you already have a Delaunay triangulation, this method allows you to derive the convex hull more efficiently than a full computation using convhull.

```
hold on
k = convexHull(DT);
xHull = DT.Points(k,1);
yHull = DT.Points(k,2);
plot(xHull,yHull,'r','LineWidth',2);
hold off
```



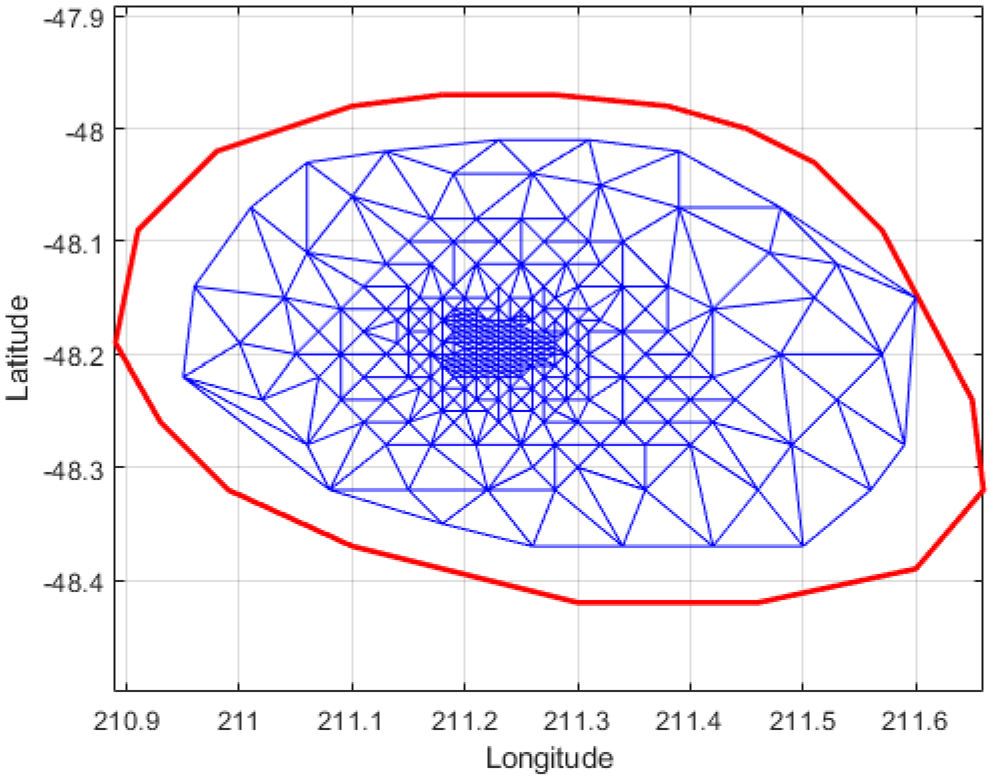
You can incrementally edit the `delaunayTriangulation` to add or remove points. If you need to add points to an existing triangulation, then an incremental addition is faster than a complete retriangulation of the augmented point set. Incremental removal of points is more efficient when the number of points to be removed is small relative to the existing number of points.

Edit the triangulation to remove the points on the convex hull from the previous computation.

```
figure
plot(xHull,yHull,'r','LineWidth',2);
axis equal
xlabel('Longitude'),ylabel('Latitude')
grid on

% The convex hull topology duplicates the start and end vertex.
% Remove the duplicate entry.
k(end) = [];

% Now remove the points on the convex hull.
DT.Points(k,:) = []
```

```
DT = 
  delaunayTriangulation with properties:

              Points: [274x2 double]
    ConnectivityList: [528x3 double]
         Constraints: []
```

```
% Plot the new triangulation.
hold on
triplot(DT);
hold off
```



There is one vertex that is just inside the boundary of the convex hull that was not removed. The fact that it is interior to the hull can be seen using the Zoom-In tool in the figure. You could plot the vertex labels to determine the index of this vertex and remove it from the triangulation. Alternatively, you can use the `nearestNeighbor` method to identify the index more readily.

The point is close to location (211.6, -48.15). Use the nearestNeighbor method to find the nearest vertex.

```
vertexId = nearestNeighbor(DT, 211.6, -48.15)
```

```
vertexId = 50
```

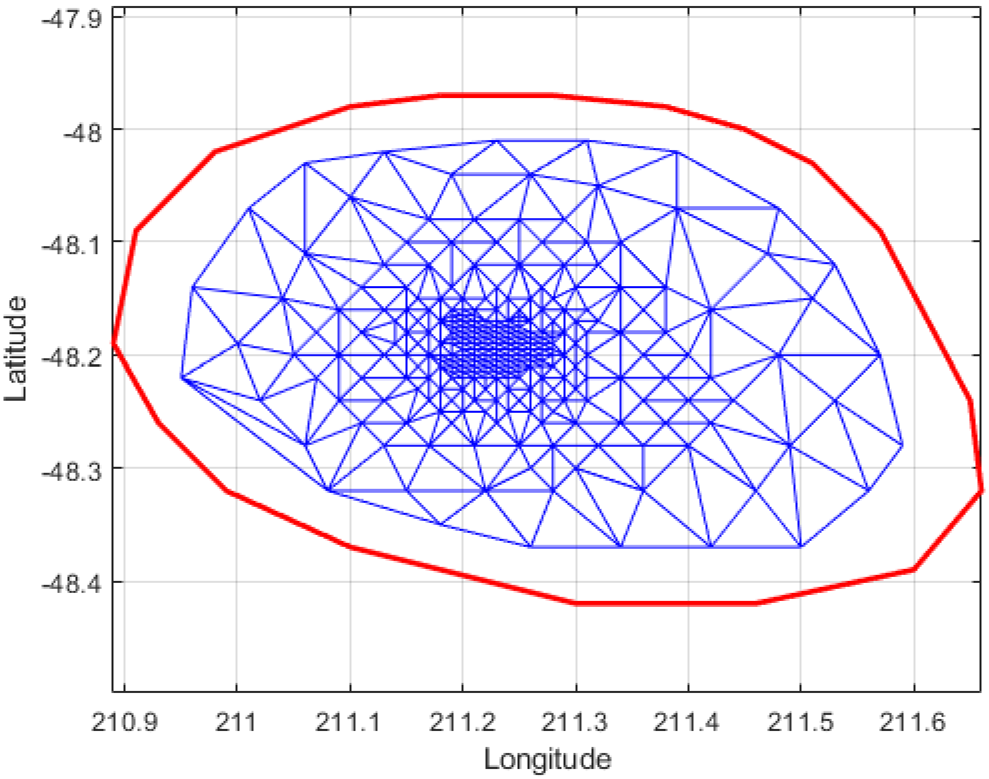
Now remove that vertex from the triangulation.

```
DT.Points(vertexId,:) = []
```

```
DT = 
  delaunayTriangulation with properties:

              Points: [273x2 double]
    ConnectivityList: [525x3 double]
         Constraints: []
```

Plot the new triangulation.

```
figure
plot(xHull,yHull,'r','LineWidth',2);
axis equal
xlabel('Longitude'),ylabel('Latitude')
grid on
hold on
triplot(DT);
hold off
```



Add points to the existing triangulation. Add 4 points to form a rectangle around the triangulation.

```
Padditional = [210.9 -48.5; 211.6 -48.5; ...
     211.6 -47.9; 210.9 -47.9];
DT.Points(end+(1:4),:) = Padditional
```

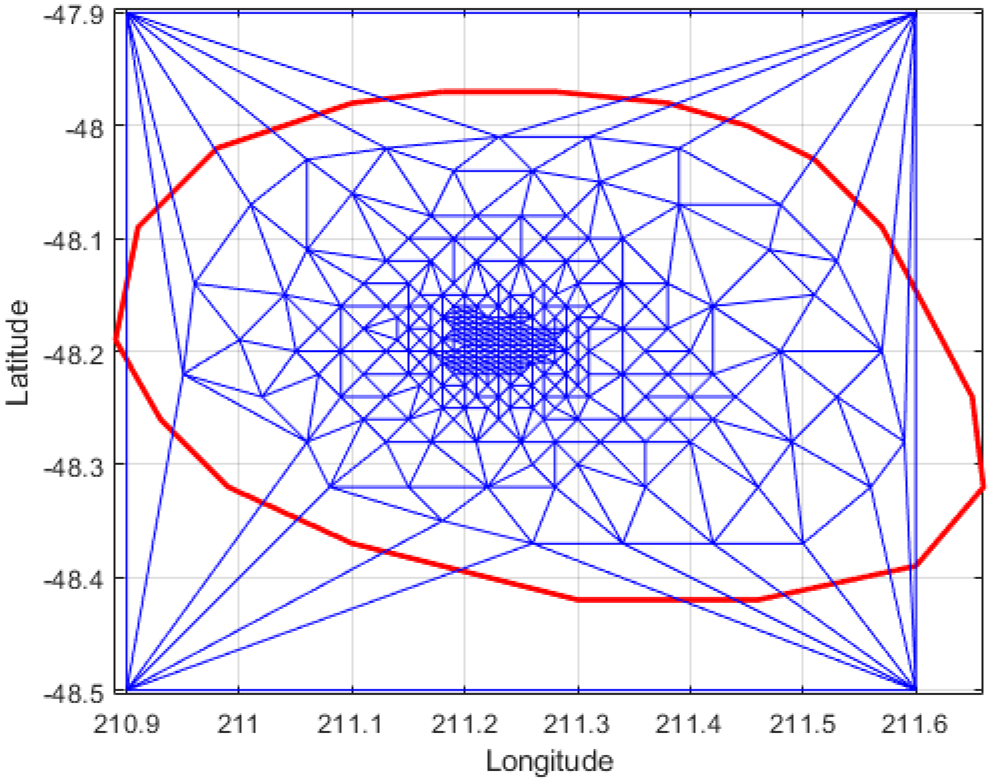
```
DT = 
  delaunayTriangulation with properties:

              Points: [277x2 double]
    ConnectivityList: [548x3 double]
         Constraints: []
```

Close all existing figures.

```
close all
```

Plot the new triangulation.

```
figure
plot(xHull,yHull,'r','LineWidth',2);
axis equal
xlabel('Longitude'),ylabel('Latitude')
grid on
hold on
triplot(DT);
hold off
```



You can edit the points in the triangulation to move them to a new location. Edit the first of the additional point set (the vertex ID 274).

```
DT.Points(274,:) = [211 -48.4];
```

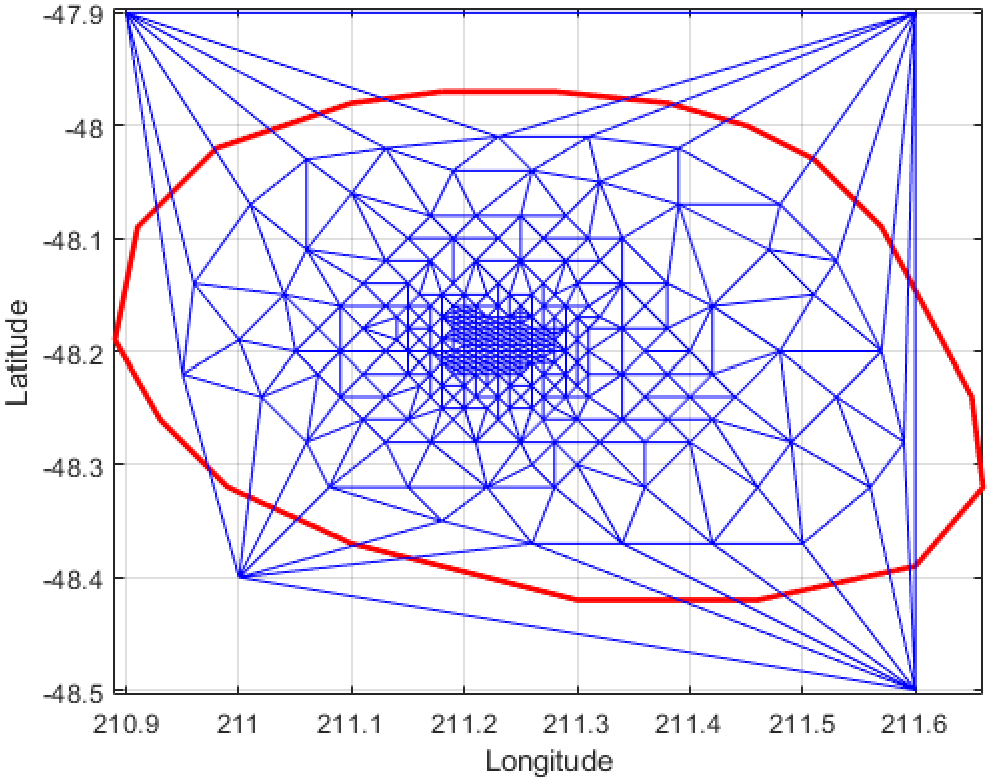
Close all existing figures.

```
close all
```

Plot the new triangulation

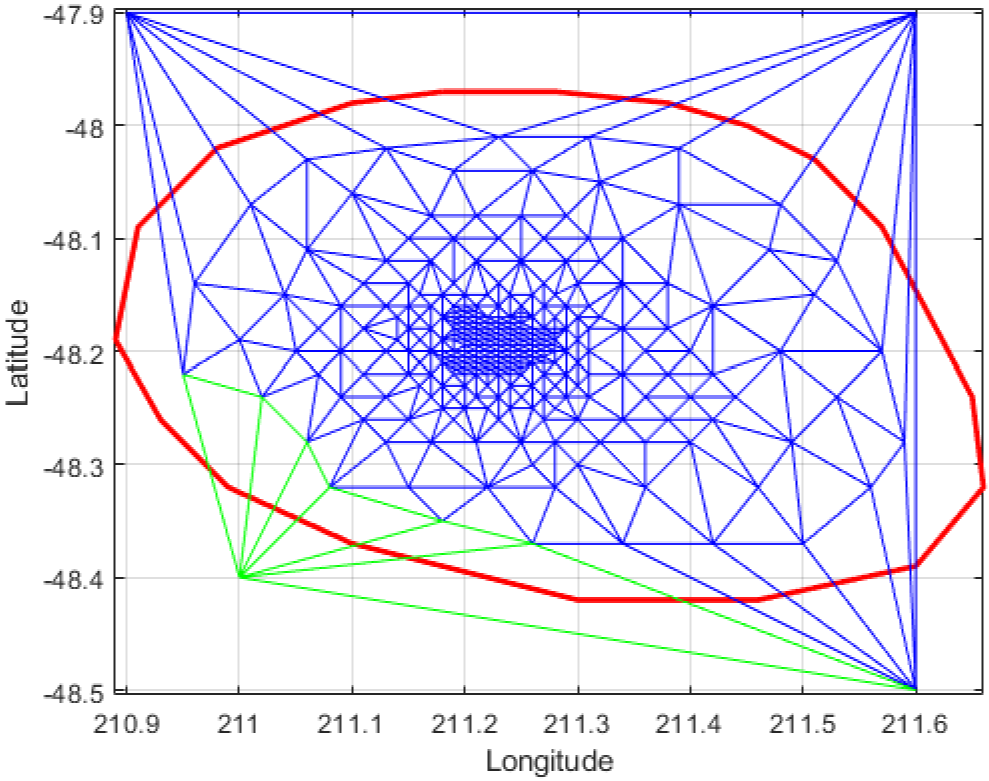
```
figure
plot(xHull,yHull,'r','LineWidth',2);
axis equal
xlabel('Longitude'),ylabel('Latitude')
grid on
hold on
```

```
triplot(DT);
hold off
```



Use the a method of the `triangulation` class, `vertexAttachments`, to find the attached triangles. Since the number of triangles attached to a vertex is variable, the method returns the attached triangle IDs in a cell array. You need braces to extract the contents.

```
attTris = vertexAttachments(DT,274);
hold on
triplot(DT(attTris{:},:),DT.Points(:,1),DT.Points(:,2),'g')
hold off
```



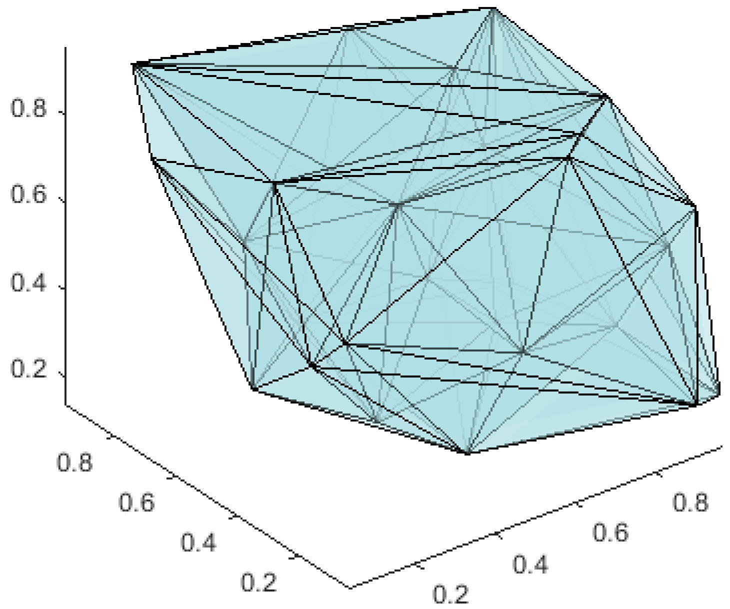
`delaunayTriangulation` also can be used to triangulate points in 3-D space. The resulting triangulation is composed of tetrahedra.

This example shows how to use a `delaunayTriangulation` to create and plot the triangulation of 3-D points.

```
rng('default')
P = rand(30,3);
DT = delaunayTriangulation(P)
```

```
DT = 
  delaunayTriangulation with properties:

              Points: [30x3 double]
    ConnectivityList: [102x4 double]
         Constraints: []
```

```
faceColor  = [0.6875 0.8750 0.8984];
tetramesh(DT,'FaceColor', faceColor,'FaceAlpha',0.3);
```

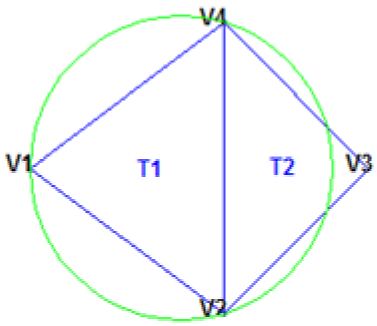


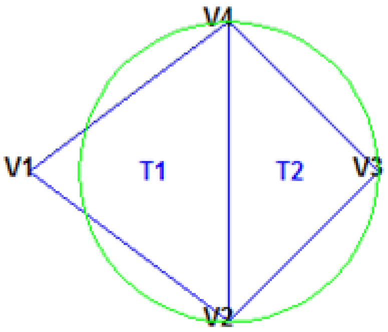
The `tetramesh` function plots both the internal and external faces of the triangulation. For large 3-D triangulations, plotting the internal faces might be an unnecessary use of resources. A plot of the boundary might be more appropriate. You can use the `freeBoundary` method to get the boundary triangulation in matrix format. Then pass the result to `trimesh` or `trisurf`.

Constrained Delaunay Triangulation

The `delaunayTriangulation` class allows you to constrain edges in a 2-D triangulation. This means you can choose a pair of points in the triangulation and constrain an edge to join those points. You can picture this as “forcing” an edge between one or more pairs of points. The following example shows how edge constraints can affect the triangulation.

The triangulation below is a Delaunay triangulation because it respects the empty circumcircle criterion.





Triangulate a set of points with an edge constraint specified between vertex V1 and V3.

Define the point set.

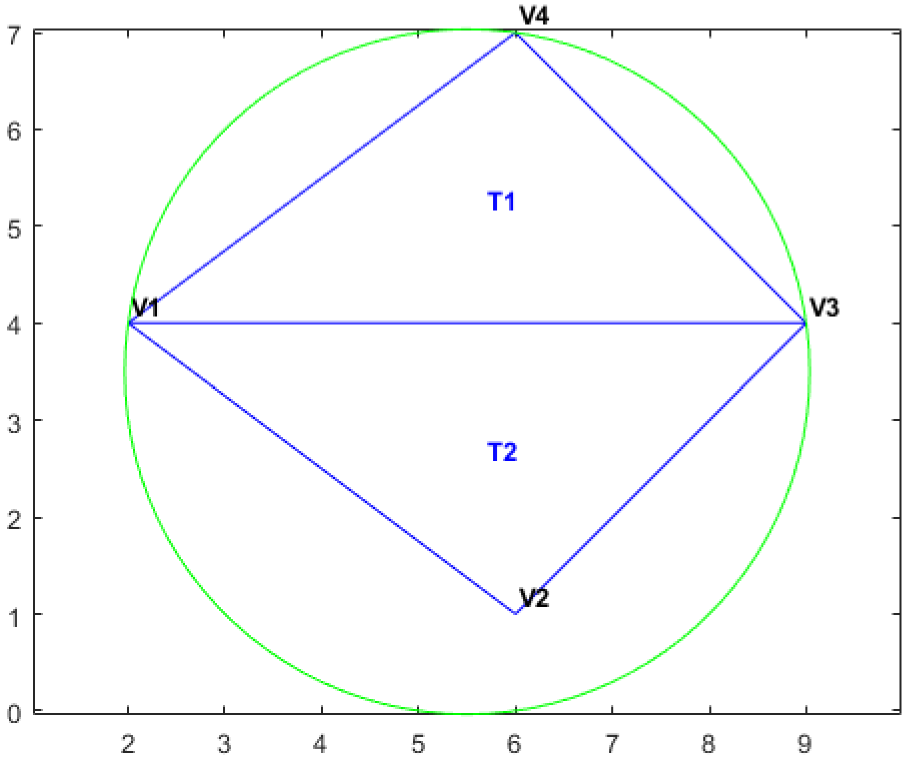
```
P = [2 4; 6 1; 9 4; 6 7];
```

Define a constraint, C, between V1 and V3.

```
C = [1 3];
DT = delaunayTriangulation(P,C);
```

Plot the triangulation and add annotations.

```
triplot(DT)

% Label the vertices.
hold on
numvx = size(P,1);
vxlabels = arrayfun(@(n) {sprintf('V%d', n)}, (1:numvx)');
Hpl = text(P(:,1)+0.2, P(:,2)+0.2, vxlabels, 'FontWeight', ...
  'bold', 'HorizontalAlignment','center', 'BackgroundColor', ...
  'none');
hold off


% Use the incenters to find the positions for placing triangle labels on the plot.
hold on
IC = incenter(DT);
numtri = size(DT,1);
trilabels = arrayfun(@(P) {sprintf('T%d', P)}, (1:numtri)');
Htl = text(IC(:,1),IC(:,2),trilabels,'FontWeight','bold', ...
'HorizontalAlignment','center','Color','blue');
hold off

% Plot the circumcircle associated with the triangle, T1.
hold on
[CC,r] = circumcenter(DT);
theta = 0:pi/50:2*pi;
xunit = r(1)*cos(theta) + CC(1,1);
yunit = r(1)*sin(theta) + CC(1,2);
plot(xunit,yunit,'g');
axis equal
hold off
```



The constraint between vertices (`V1`, `V3`) was honored, however, the Delaunay criterion was invalidated. This also invalidates the nearest-neighbor relation that is inherent in a Delaunay triangulation. This means the `nearestNeighbor` search method provided by `delaunayTriangulation` cannot be supported if the triangulation has constraints.

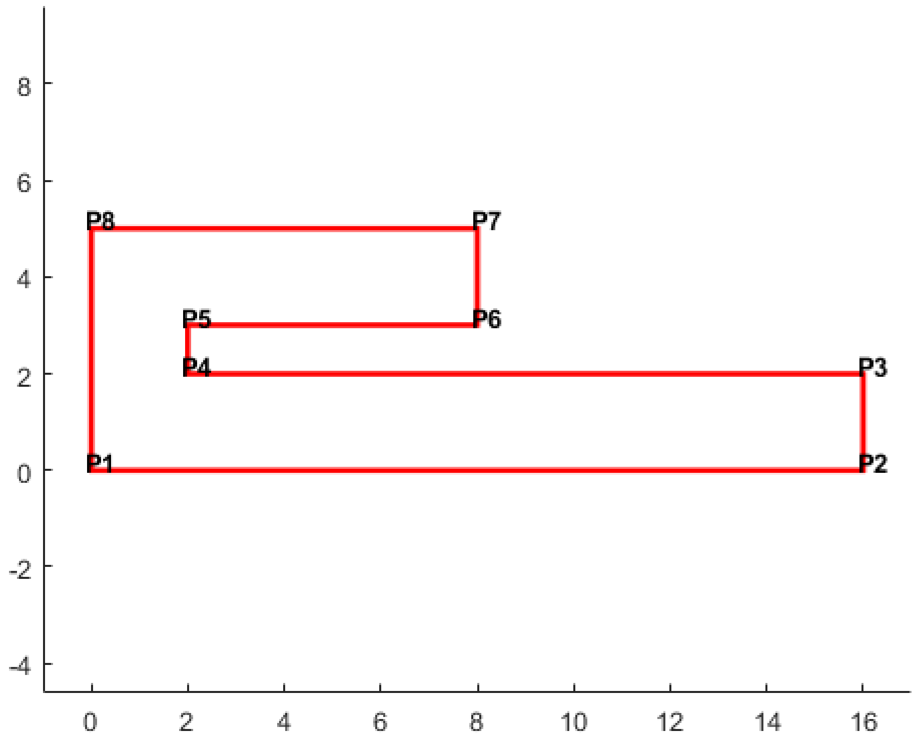
In typical applications, the triangulation might be composed of many points, and a relatively small number of edges in the triangulation might be constrained. Such a triangulation is said to be locally non-Delaunay, because many triangles in the triangulation might respect the Delaunay criterion, but locally there might be some triangles that do not. In many applications, local relaxation of the empty circumcircle property is not a concern.

Constrained triangulations are generally used to triangulate a nonconvex polygon. The constraints give us a correspondence between the polygon edges and the triangulation edges. This relationship enables you to extract a triangulation that represents the region. The following example shows how to use a constrained `delaunayTriangulation` to triangulate a nonconvex polygon.

Define and plot a polygon.

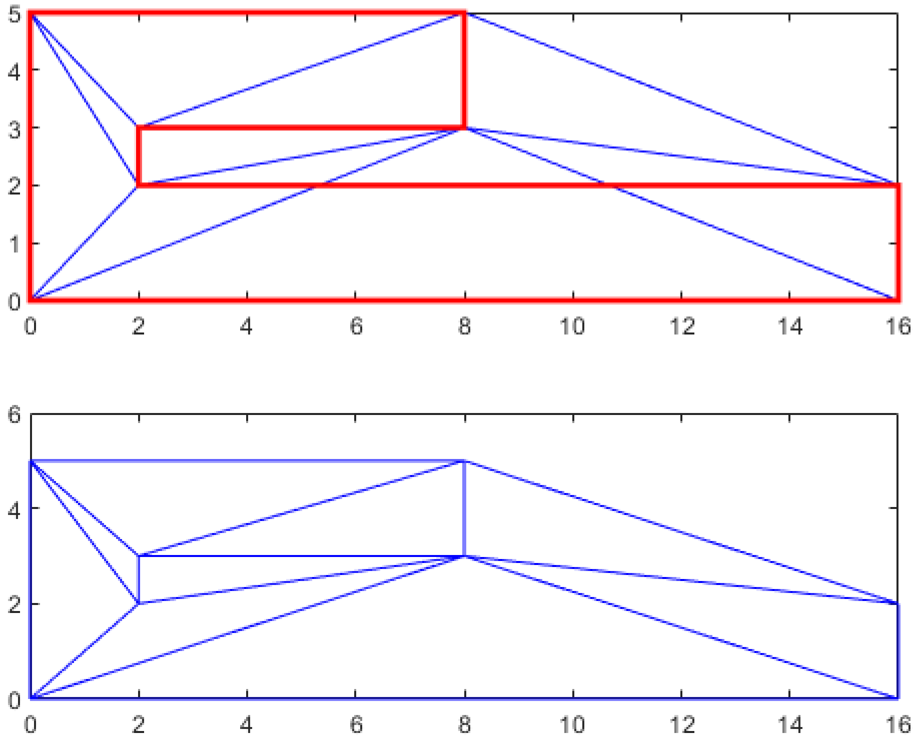
```
figure()
axis([-1 17 -1 6]);
axis equal
P = [0 0; 16 0; 16 2; 2 2; 2 3; 8 3; 8 5; 0 5];
patch(P(:,1),P(:,2),'-r','LineWidth',2,'FaceColor',...
     'none','EdgeColor','r');

% Label the points.
```

```
hold on
numvx = size(P,1);
vxlabels = arrayfun(@(n) {sprintf('P%d', n)}, (1:numvx)');
Hpl = text(P(:,1)+0.2, P(:,2)+0.2, vxlabels, 'FontWeight', ...
  'bold', 'HorizontalAlignment','center', 'BackgroundColor', ...
  'none');
hold off
```



Create and plot the triangulation together with the polygon boundary.

```
figure()
subplot(2,1,1);
axis([-1 17 -1 6]);
axis equal

P = [0 0; 16 0; 16 2; 2 2; 2 3; 8 3; 8 5; 0 5];
DT = delaunayTriangulation(P);
triplot(DT)
hold on;
patch(P(:,1),P(:,2),'-r','LineWidth',2,'FaceColor',...
      'none','EdgeColor','r');
hold off

% Plot the standalone triangulation in a subplot.
subplot(2,1,2);
axis([-1 17 -1 6]);
axis equal
triplot(DT)
```



This triangulation cannot be used to represent the domain of the polygon because some triangles cut across the boundary. You need to impose a constraint on the edges that are cut by triangulation edges. Since all edges have to be respected, you need to constrain all edges. The steps below show how to constrain all the edges.

Enter the constrained edge definition. Observe from the annotated figure where you need constraints (between (V1, V2), (V2, V3), and so on).

```
C = [1 2; 2 3; 3 4; 4 5; 5 6; 6 7; 7 8; 8 1];
```

In general, if you have `N` points in a sequence that define a polygonal boundary, the constraints can be expressed as `C = [(1:(N-1))' (2:N)'; N 1];`.

Specify the constraints when you create the `delaunayTriangulation`.

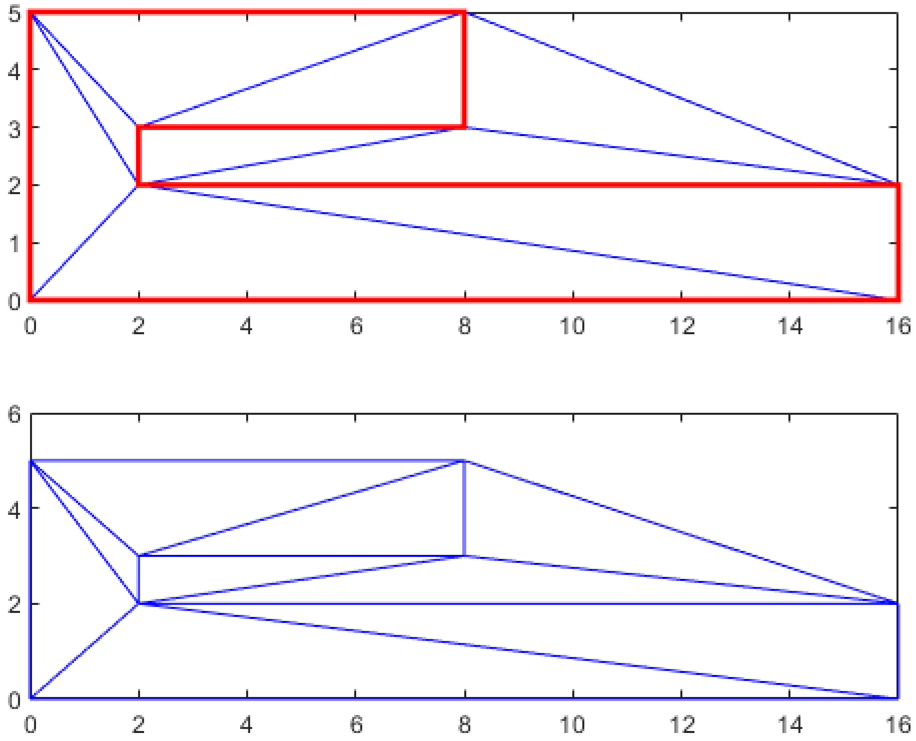
```
DT = delaunayTriangulation(P,C);
```

Alternatively, you can impose constraints on an existing triangulation by setting the `Constraints` property: `DT.Constraints = C;`.

Plot the triangulation and polygon.

```
figure('Color','white')
subplot(2,1,1);
axis([-1 17 -1 6]);
axis equal
triplot(DT)
```

```
hold on;
patch(P(:,1),P(:,2),'-r','LineWidth',2, ...
     'FaceColor','none','EdgeColor','r');
hold off

% Plot the standalone triangulation in a subplot.
subplot(2,1,2);
axis([-1 17 -1 6]);
axis equal
triplot(DT)
```

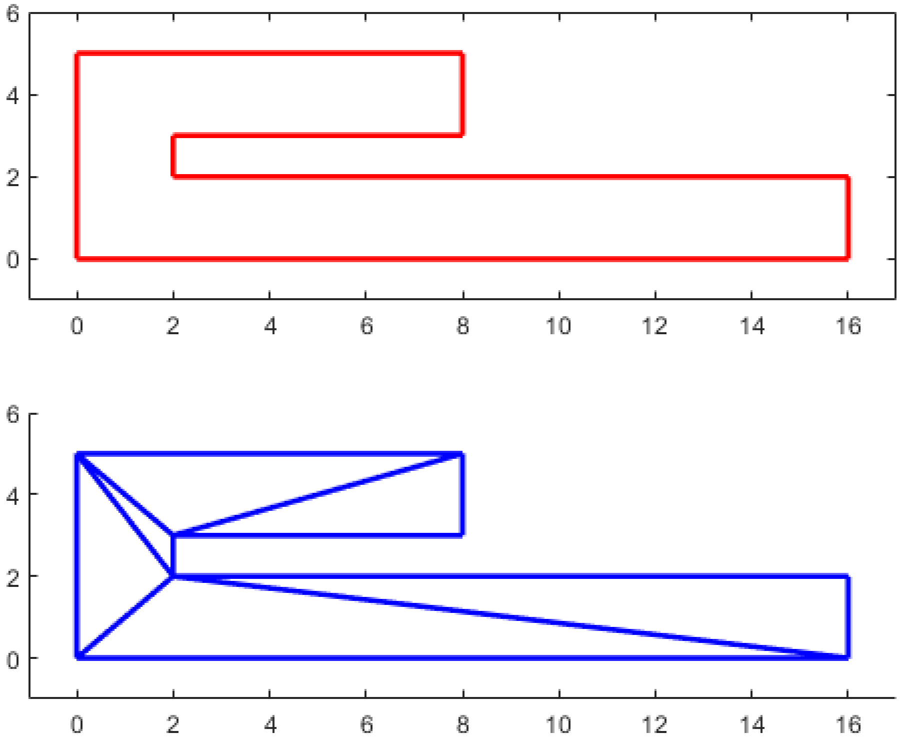


The plot shows that the edges of the triangulation respect the boundary of the polygon. However, the triangulation fills the concavities. What is needed is a triangulation that represents the polygonal domain. You can extract the triangles within the polygon using the `delaunayTriangulation` method, `isInterior`. This method returns a logical array whose `true` and `false` values that indicate whether the triangles are inside a bounded geometric domain. The analysis is based on the Jordan Curve theorem, and the boundaries are defined by the edge constraints. The ith triangle in the triangulation is considered to be inside the domain if the ith logical flag is true, otherwise it is outside.

Now use the `isInterior` method to compute and plot the set of domain triangles.

```
% Plot the constrained edges in red.
figure('Color','white')
subplot(2,1,1);
plot(P(C'),P(C'+size(P,1)),'-r','LineWidth', 2);
axis([-1 17 -1 6]);
```

```
% Compute the in/out status.
IO = isInterior(DT);
subplot(2,1,2);
hold on;
axis([-1 17 -1 6]);

% Use triplot to plot the triangles that are inside.
% Uses logical indexing and dt(i,j) shorthand
% format to access the triangulation.
triplot(DT(IO, :),DT.Points(:,1), DT.Points(:,2),'LineWidth', 2)
hold off;
```



Triangulation of Point Sets Containing Duplicate Locations

The Delaunay algorithms in MATLAB construct a triangulation from a unique set of points. If the points passed to the triangulation function, or class, are not unique, the duplicate locations are detected and the duplicate point is ignored. This produces a triangulation that does not reference some points in the original input, namely the duplicate points. When you work with the `delaunay` and `delaunayn` functions, the presence of duplicates may be of little consequence. However, since many of the queries provided by the `delaunayTriangulation` class are index based, it is important to understand that `delaunayTriangulation` triangulates and works with the unique data set. Therefore, indexing based on the unique point set is the convention. This data is maintained by the `Points` property of `delaunayTriangulation`.

The following example illustrates the importance of referencing the unique data set stored within the Points property when working with delaunayTriangulation:

```
rng('default')
P = rand([25 2]);
P(18,:) = P(8,:)
P(16,:) = P(6,:)
P(12,:) = P(2,:)

DT = delaunayTriangulation(P)
```

When the triangulation is created, MATLAB issues a warning. The Points property shows that the duplicate points have been removed from the data.

```
DT = 

  delaunayTriangulation with properties:

              Points: [22x2 double]
    ConnectivityList: [31x3 double]
         Constraints: []
```

If for example, the Delaunay triangulation is used to compute the convex hull, the indices of the points on the hull are indices with respect to the unique point set, DT.Points. Therefore, use the following code to compute and plot the convex hull:

```
K = DT.convexHull();
plot(DT.Points(:,1),DT.Points(:,2),'.');
hold on
plot(DT.Points(K,1),DT.Points(K,2),'-r');
```

If the original data set containing the duplicates were used in conjunction with the indices provided by delaunayTriangulation, then the result would be incorrect. The delaunayTriangulation works with indices that are based on the unique data set DT.Points. For example, the following would produce an incorrect plot, because K is indexed with respect to DT.Points and not P:

```
K = DT.convexHull();
plot(P(:,1),P(:,2),'.');
hold on
plot(P(K,1),P(K,2),'-r');
```

It’s often more convenient to create a unique data set by removing duplicates prior to creating the delaunayTriangulation. Doing this eliminates the potential for confusion. This can be accomplished using the unique function as follows:

```
rng('default')
P = rand([25 2]);
P(18,:) = P(8,:)
P(16,:) = P(6,:)
P(12,:) = P(2,:)

[~, I, ~] = unique(P,'first','rows');
I = sort(I);
P = P(I,:);
DT = delaunayTriangulation(P)  % The point set is unique
```

See Also

More About

- "Spatial Searching" on page 7-53

Creating and Editing Delaunay Triangulations

This example shows how to create, edit, and query Delaunay triangulations using the `delaunayTriangulation` class. The Delaunay triangulation is the most widely used triangulation in scientific computing. The properties associated with the triangulation provide a basis for solving a variety of geometric problems. Construction of constrained Delaunay triangulations is also shown, together with an applications covering medial axis computation and mesh morphing.

Example One: Create and Plot 2-D Delaunay Triangulation

This example shows you how to compute a 2-D Delaunay triangulation and then plot the triangulation together with the vertex and triangle labels.

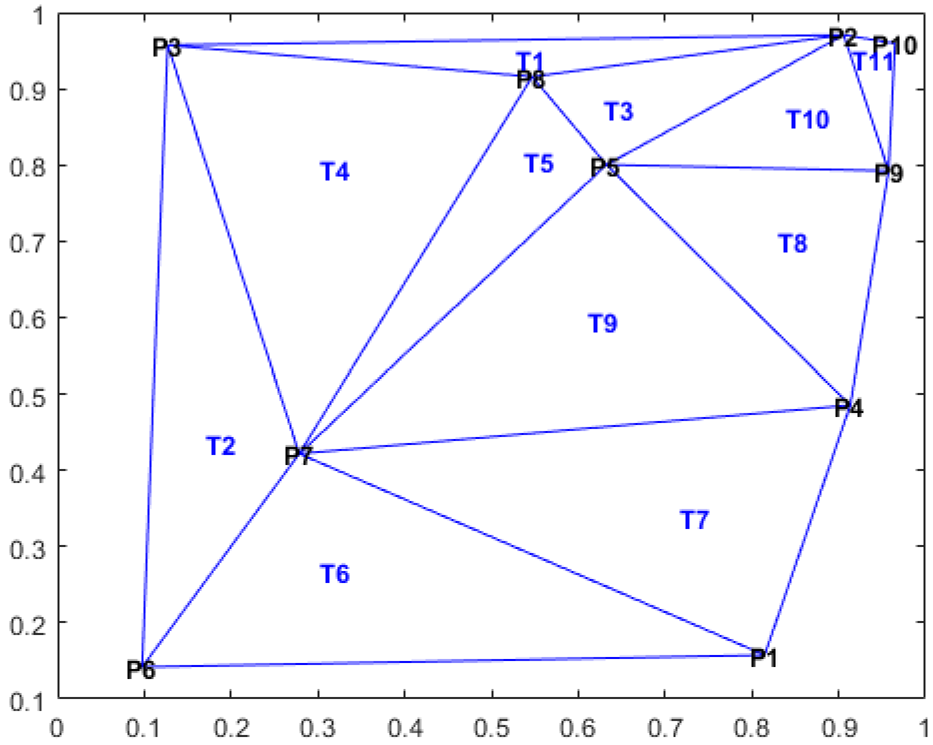
```
rng default
x = rand(10,1);
y = rand(10,1);
dt = delaunayTriangulation(x,y)
```

```
dt = 
  delaunayTriangulation with properties:

              Points: [10x2 double]
    ConnectivityList: [11x3 double]
         Constraints: []
```

```
triplot(dt)
```

Display the vertex and triangle labels on the plot.

```
hold on
vxlabels = arrayfun(@(n) {sprintf('P%d', n)}, (1:10)');
Hpl = text(x,y,vxlabels,'FontWeight','bold','HorizontalAlignment',...
   'center','BackgroundColor','none');
ic = incenter(dt);
numtri = size(dt,1);
trilabels = arrayfun(@(x) {sprintf('T%d',x)}, (1:numtri)');
Htl = text(ic(:,1),ic(:,2),trilabels,'FontWeight','bold', ...
   'HorizontalAlignment','center','Color','blue');
hold off
```



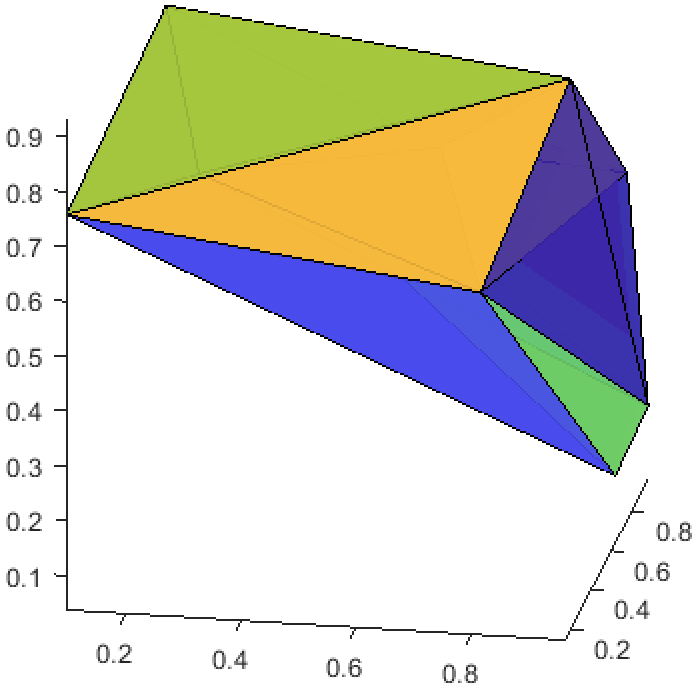
Example Two: Create and Plot 3-D Delaunay Triangulation

This example shows you how to compute and plot a 3-D Delaunay triangulation.

```
rng default
X = rand(10,3);
dt = delaunayTriangulation(X)
```

```
dt =
  delaunayTriangulation with properties:

              Points: [10x3 double]
    ConnectivityList: [18x4 double]
         Constraints: []
```

```
tetramesh(dt)
view([10 20])
```



To display large tetrahedral meshes, use the convexHull method to compute the boundary triangulation and plot it using trisurf. For example:

```
triboundary = convexHull(dt);
trisurf(triboundary, X(:,1), X(:,2), X(:,3),'FaceColor','cyan')
```

Example Three: Access Triangulation Data Structure

There are two ways to access the triangulation data structure. One way is via the Triangulation property, the other way is using indexing.

Create a 2-D Delaunay triangulation from 10 random points.

```
rng default
X = rand(10,2);
dt = delaunayTriangulation(X)
```

```
dt = 
  delaunayTriangulation with properties:

              Points: [10x2 double]
    ConnectivityList: [11x3 double]
         Constraints: []
```

One way to access the triangulation data structure is with the ConnectivityList property.

```
dt.ConnectivityList
```

```
ans = 11×3

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

Indexing is a shorthand way to query the triangulation. The syntax is `dt(i,j)`, where `j` is the `j`th vertex of the `i`th triangle. Standard indexing rules apply.

Query the triangulation data structure with indexing.

```
dt(:,:)
```

```
ans = 11×3

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

The second triangle is:

```
dt(2,:)
```

```
ans = 1×3

     6     7     3
```

The third vertex of the second triangle is:

```
dt(2,3)
```

```
ans = 3
```

The first three triangles are:

```
dt(1:3,:)
```

```
ans = 3×3

     8     2     3
     6     7     3
     5     2     8
```

Example Four: Edit Delaunay Triangulation to Insert or Remove Points

This example shows you how to use index-based subscripting to insert or remove points. It is more efficient to edit a `delaunayTriangulation` to make minor modifications as opposed to recreating a new `delaunayTriangulation` from scratch, this is especially true if the data set is large.

Construct a Delaunay triangulation from 10 random points within a unit square.

```
rng default
x = rand(10,1);
y = rand(10,1);
dt = delaunayTriangulation(x,y)
```

```
dt = 
  delaunayTriangulation with properties:

              Points: [10x2 double]
    ConnectivityList: [11x3 double]
         Constraints: []
```

Insert 5 additional random points.

```
dt.Points(end+(1:5),:) = rand(5,2)
```

```
dt = 
  delaunayTriangulation with properties:

              Points: [15x2 double]
    ConnectivityList: [20x3 double]
         Constraints: []
```

Replace the fifth point.

```
dt.Points(5,:) = [0 0]
```

```
dt = 
  delaunayTriangulation with properties:

              Points: [15x2 double]
    ConnectivityList: [20x3 double]
         Constraints: []
```

Remove the fourth point.

```
dt.Points(4,:) = []
```

```
dt = 
  delaunayTriangulation with properties:

              Points: [14x2 double]
    ConnectivityList: [18x3 double]
```

```
         Constraints: []
```

Example Five: Create Constrained Delaunay Triangulation

This example shows you how to create a constrained Delaunay triangulation and illustrates the effect of the constraints.

Create and plot a constrained Delaunay triangulation.

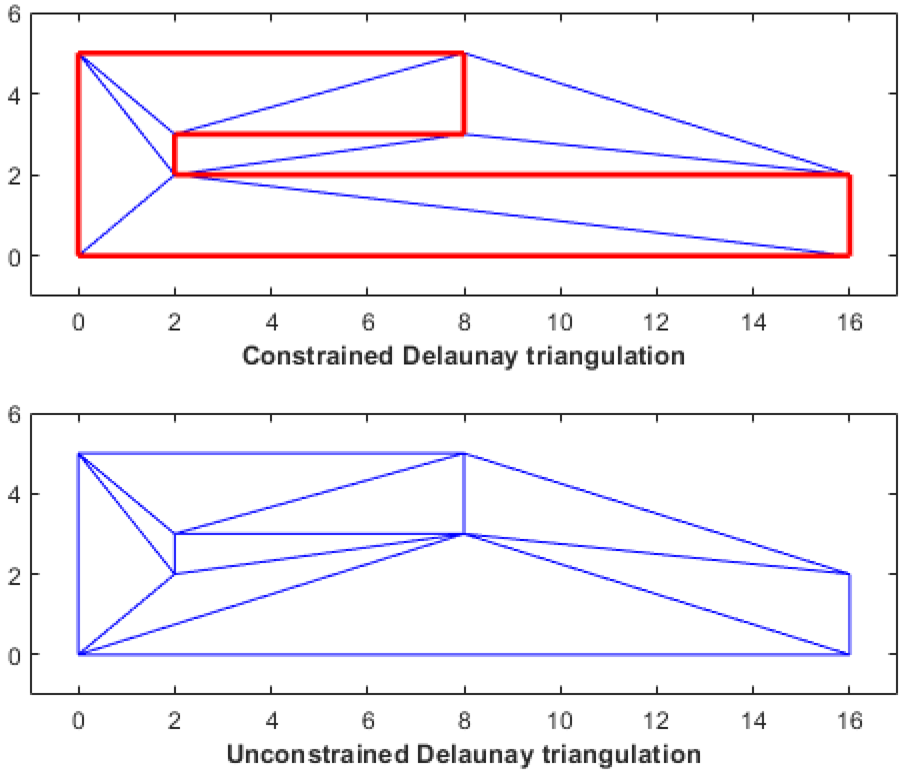
```
X = [0 0; 16 0; 16 2; 2 2; 2 3; 8 3; 8 5; 0 5];
C = [1 2; 2 3; 3 4; 4 5; 5 6; 6 7; 7 8; 8 1];
dt = delaunayTriangulation(X,C);
subplot(2,1,1)
triplot(dt)
axis([-1 17 -1 6])
xlabel('Constrained Delaunay triangulation','FontWeight','b')
```

Plot the constrained edges in red.

```
hold on
plot(X(C'),X(C'+size(X,1)),'-r','LineWidth',2)
hold off
```

Now delete the constraints and plot the unconstrained Delaunay triangulation.

```
dt.Constraints = [];
subplot(2,1,2)
triplot(dt)
axis([-1 17 -1 6])
xlabel('Unconstrained Delaunay triangulation','FontWeight','b')
```



Example Six: Create Constrained Delaunay Triangulation of Geographical Map

Load a map of the perimeter of the conterminous United States. Construct a constrained Delaunay triangulation representing the polygon. This triangulation spans a domain that is bounded by the convex hull of the set of points. Filter out the triangles that are within the domain of the polygon and plot them. Note: The data set contains duplicate data points; that is, two or more datapoints have the same location. The duplicate points are rejected and the `delaunayTriangulation` reformats the constraints accordingly.

```
clf
load usapolygon
```

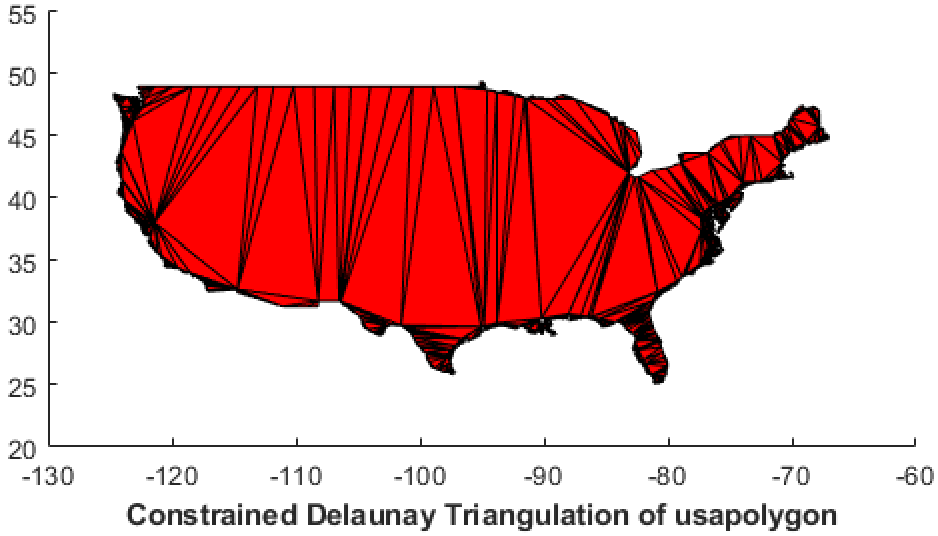
Define an edge constraint between two successive points that make up the polygonal boundary and create the Delaunay triangulation.

```
nump = numel(uslon);
C = [(1:(nump-1))' (2:nump)'; nump 1];
dt = delaunayTriangulation(uslon,uslat,C);
```

```
Warning: Duplicate data points have been detected and removed.
 The Triangulation indices and constraints are defined with respect to the unique set of points i

Warning: Intersecting edge constraints have been split, this may have added new points into the t
```

```
io = isInterior(dt);
patch('Faces',dt(io,:),'Vertices',dt.Points,'FaceColor','r')
axis equal
```

```
axis([-130 -60 20 55])
xlabel('Constrained Delaunay Triangulation of usapolygon','FontWeight','b')
```



Example Seven: Curve Reconstruction from Point Cloud

This example highlights the use of a Delaunay triangulation to reconstruct a polygonal boundary from a cloud of points. The reconstruction is based on the elegant Crust algorithm.

Reference: N. Amenta, M. Bern, and D. Eppstein. The crust and the beta-skeleton: combinatorial curve reconstruction. *Graphical Models and Image Processing*, 60:125-135, 1998.

Create a set of points representing the point cloud.

```
numpts = 192;
t = linspace( -pi, pi, numpts+1 )';
t(end) = [];
r = 0.1 + 5*sqrt( cos( 6*t ).^2 + (0.7).^2 );
x = r.*cos(t);
y = r.*sin(t);
ri = randperm(numpts);
x = x(ri);
y = y(ri);
```

Construct a Delaunay Triangulation of the point set.

```
dt = delaunayTriangulation(x,y);
tri = dt(:,:);
```

Insert the location of the Voronoi vertices into the existing triangulation.

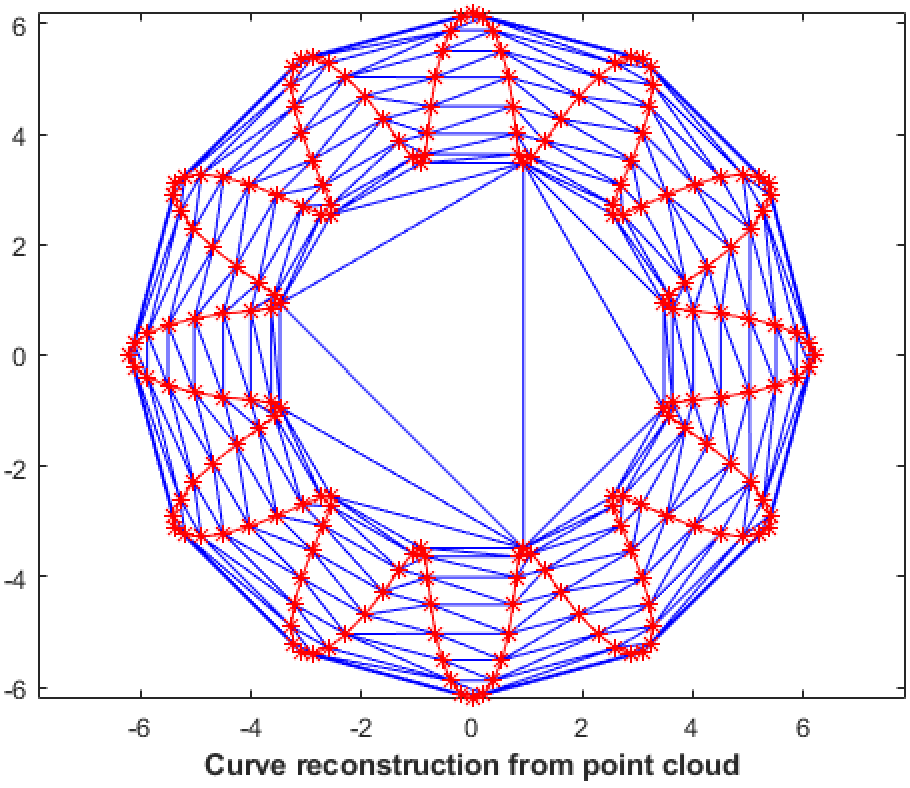
```
V = voronoiDiagram(dt);
```

Remove the infinite vertex and filter out duplicate points using `unique`.

```
V(1,:) = [];
numv = size(V,1);
dt.Points(end+(1:numv),:) = unique(V,'rows');
```

The Delaunay edges that connect pairs of sample points represent the boundary.

```
delEdges = edges(dt);
validx = delEdges(:,1) <= numpts;
validy = delEdges(:,2) <= numpts;
boundaryEdges = delEdges((validx & validy),:)';
xb = x(boundaryEdges);
yb = y(boundaryEdges);
clf
triplot(tri,x,y)
axis equal
hold on
plot(x,y,'*r')
plot(xb,yb,'-r')
xlabel('Curve reconstruction from point cloud','FontWeight','b')
hold off
```



Example Eight: Compute Approximate Medial Axis of Polygonal Domain

This example shows how to create an approximate Medial Axis of a polygonal domain using a constrained Delaunay triangulation. The *Medial Axis* of a polygon is defined by the locus of the center of a maximal disk within the polygon interior.

Construct a constrained Delaunay triangulation of a sample of points on the domain boundary.

```
load trimesh2d
dt = delaunayTriangulation(x,y,Constraints);
inside = isInterior(dt);
```

Construct a triangulation to represent the domain triangles.

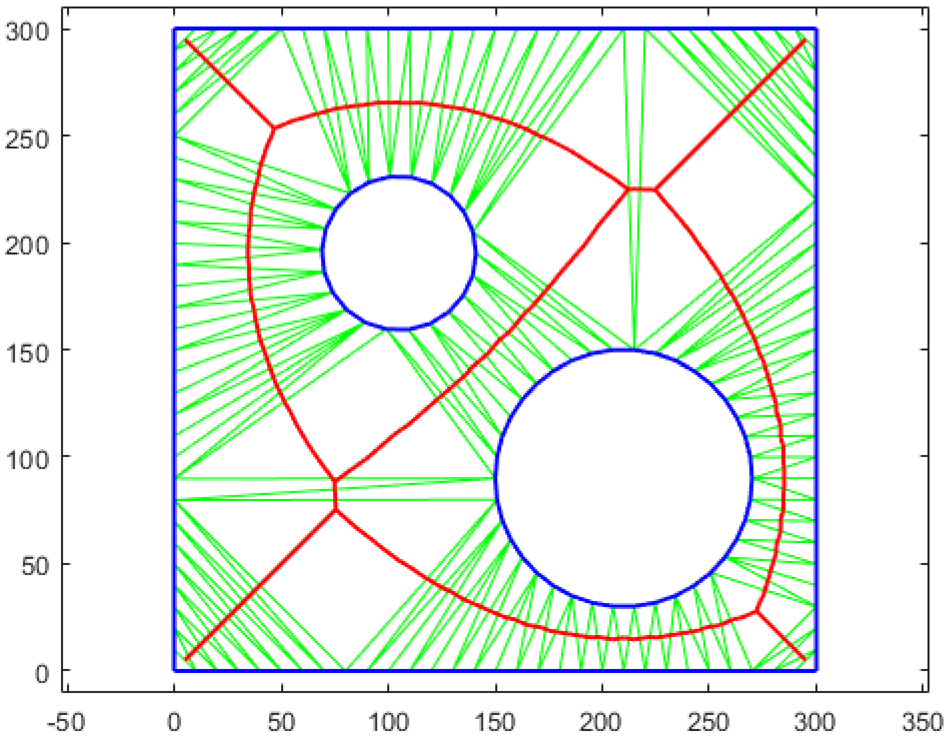
```
tr = triangulation(dt(inside,:),dt.Points);
```

Construct a set of edges that join the circumcenters of neighboring triangles. The additional logic constructs a unique set of such edges.

```
numt = size(tr,1);
T = (1:numt)';
neigh = neighbors(tr);
cc = circumcenter(tr);
xcc = cc(:,1);
ycc = cc(:,2);
idx1 = T < neigh(:,1);
idx2 = T < neigh(:,2);
idx3 = T < neigh(:,3);
neigh = [T(idx1) neigh(idx1,1); T(idx2) neigh(idx2,2); T(idx3) neigh(idx3,3)]';
```

Plot the domain triangles in green, the domain boundary in blue, and the medial axis in red.

```
clf
triplot(tr,'g')
hold on
plot(xcc(neigh), ycc(neigh), '-r','LineWidth',1.5)
axis([-10 310 -10 310])
axis equal
plot(x(Constraints'),y(Constraints'),'-b','LineWidth',1.5)
xlabel('Medial Axis of Polygonal Domain','FontWeight','b')
hold off
```



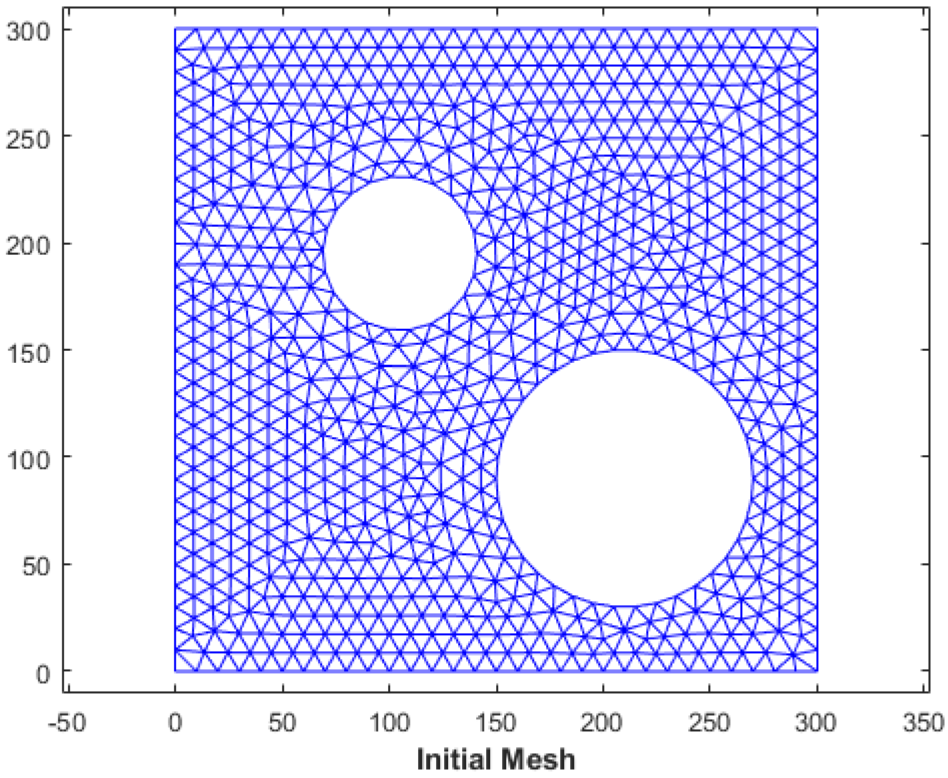
Medial Axis of Polygonal Domain

Example Nine: Morph 2-D Mesh to Modified Boundary

This example shows how to morph a mesh of a 2-D domain to accommodate a modification to the domain boundary.

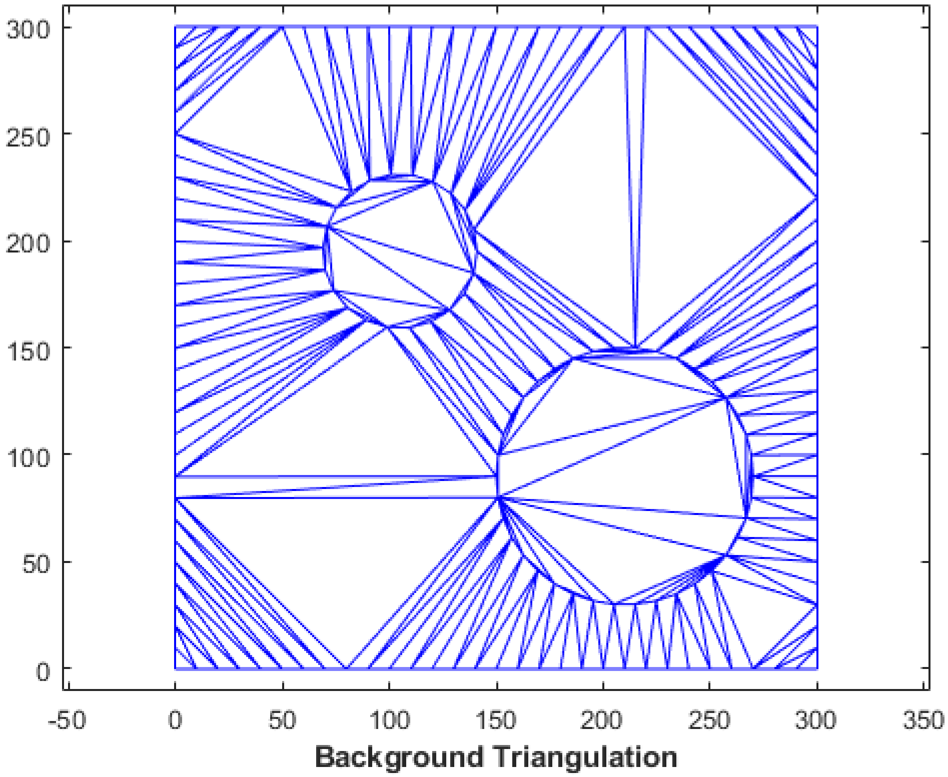
Step 1: Load the data. The mesh to be morphed is defined by `trife`, `xfe`, and `yfe`, which is a triangulation in face-vertex format.

```
load trimesh2d
clf
triplot(trife,xfe,yfe)
axis equal
axis([-10 310 -10 310])
axis equal
xlabel('Initial Mesh','FontWeight','b')
```



Step 2: Construct a background triangulation - a Constrained Delaunay triangulation of the set of points representing the mesh boundary. For each vertex of the mesh, compute a descriptor that defines its location with respect to the background triangulation. The descriptor is the enclosing triangle together with the barycentric coordinates with respect to that triangle.

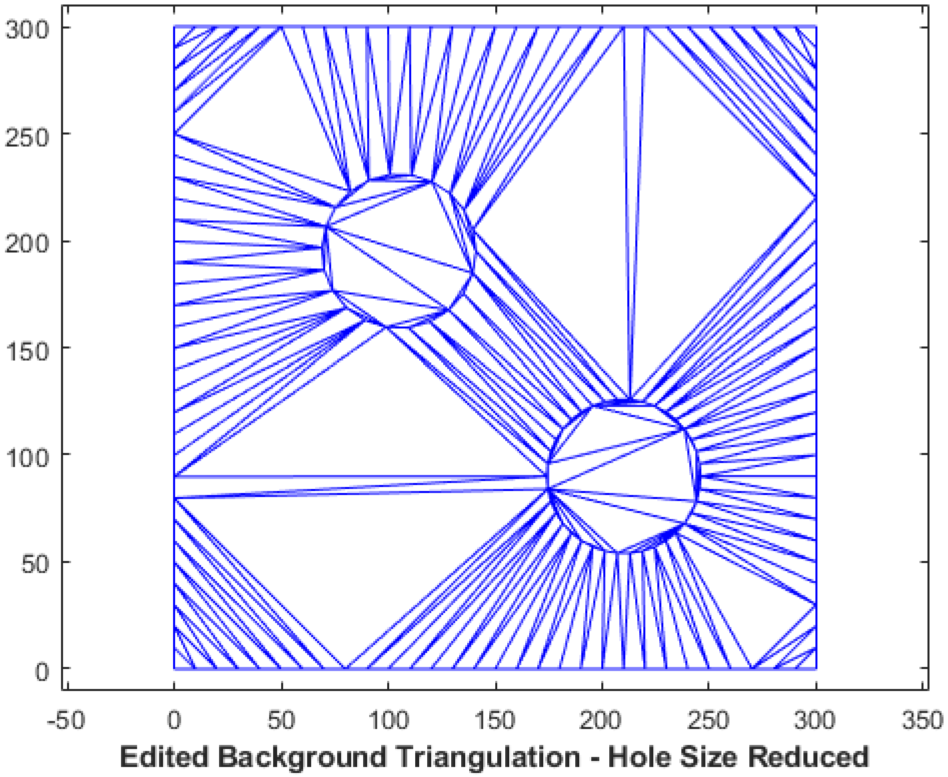
```
dt = delaunayTriangulation(x,y,Constraints);
clf
triplot(dt)
axis equal
axis([-10 310 -10 310])
axis equal
xlabel('Background Triangulation','FontWeight','b')
```



```
descriptors.tri = pointLocation(dt,xfe,yfe);
descriptors.baryCoords = cartesianToBarycentric(dt,descriptors.tri,[xfe yfe]);
```

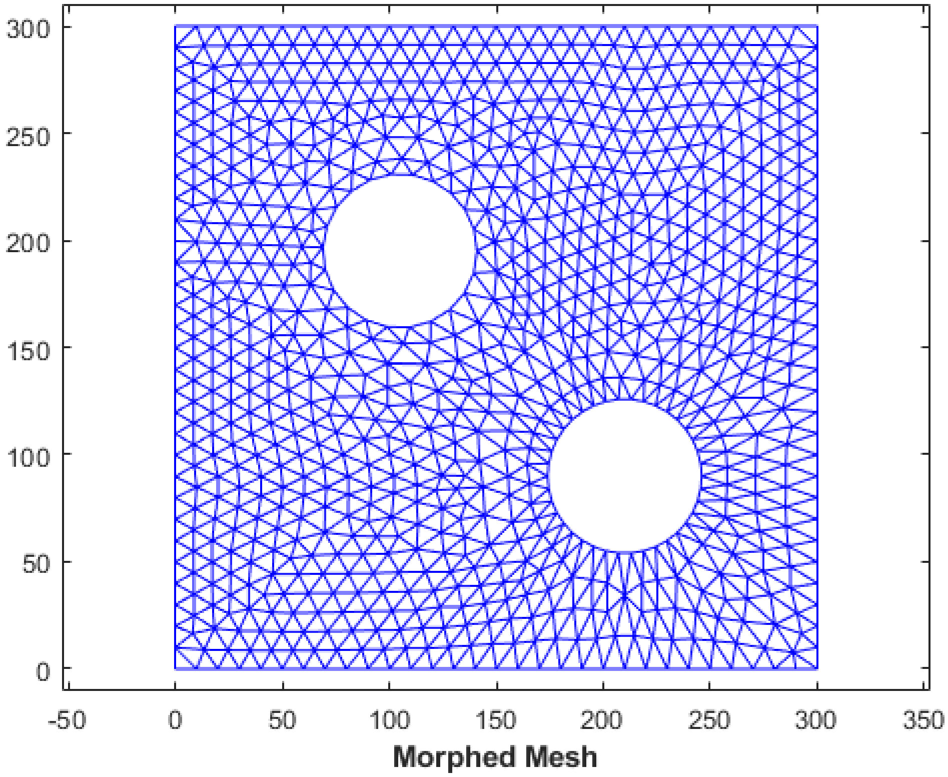
Step 3: Edit the background triangulation to incorporate the desired modification to the domain boundary.

```
cc1 = [210 90];
circ1 = (143:180)';
x(circ1) = (x(circ1)-cc1(1))*0.6 + cc1(1);
y(circ1) = (y(circ1)-cc1(2))*0.6 + cc1(2);
tr = triangulation(dt(:,:),x,y);
clf
triplot(tr)
axis([-10 310 -10 310])
axis equal
xlabel('Edited Background Triangulation - Hole Size Reduced','FontWeight','b')
```



Step 4: Convert the descriptors back to Cartesian coordinates using the deformed background triangulation as a basis for evaluation.

```
Xnew = barycentricToCartesian(tr,descriptors.tri,descriptors.baryCoords);
tr = triangulation(trife,Xnew);
clf
triplot(tr)
axis([-10 310 -10 310])
axis equal
xlabel('Morphed Mesh','FontWeight','b')
```


300
250
200
150
100
50
0
-50
0
50
100
150
200
250
300
350
Morphed Mesh

Spatial Searching

Introduction

MATLAB provides the necessary functions for performing a spatial search using either a Delaunay triangulation or a general triangulation. The search queries that MATLAB supports are:

- Nearest-neighbor search (sometimes called closest-point search or proximity search).
- Point-location search (sometimes called point-in-triangle search or point-in-simplex search, where a simplex is a triangle, tetrahedron or higher dimensional equivalent).

Given a set of points X and a query point q in Euclidean space, the nearest-neighbor search locates a point p in X that is closer to q than to any other point in X. Given a triangulation of X, the point-location search locates the triangle or tetrahedron that contains the query point q. Since these methods work for both Delaunay as well as general triangulations, you can use them even if a modification of the points violates the Delaunay criterion. You also can search a general triangulation represented in matrix format.

While MATLAB supports these search schemes in N dimensions, exact spatial searches usually become prohibitive as the number of dimensions extends beyond 3-D. You should consider approximate alternatives for large problems in up to 10 dimensions.

Nearest-Neighbor Search

There are a few ways to compute nearest-neighbors in MATLAB, depending on the dimensionality of the problem:

- For 2-D and 3-D searches, use the `nearestNeighbor` method provided by the `triangulation` class and inherited by the `delaunayTriangulation` class.
- For 4-D and higher, use the `delaunayn` function to construct the triangulation and the complementary `dsearchn` function to perform the search. While these N-D functions support 2-D and 3-D, they are not as general and efficient as the triangulation search methods.

This example shows how to perform a nearest-neighbor search in 2-D with `delaunayTriangulation`.

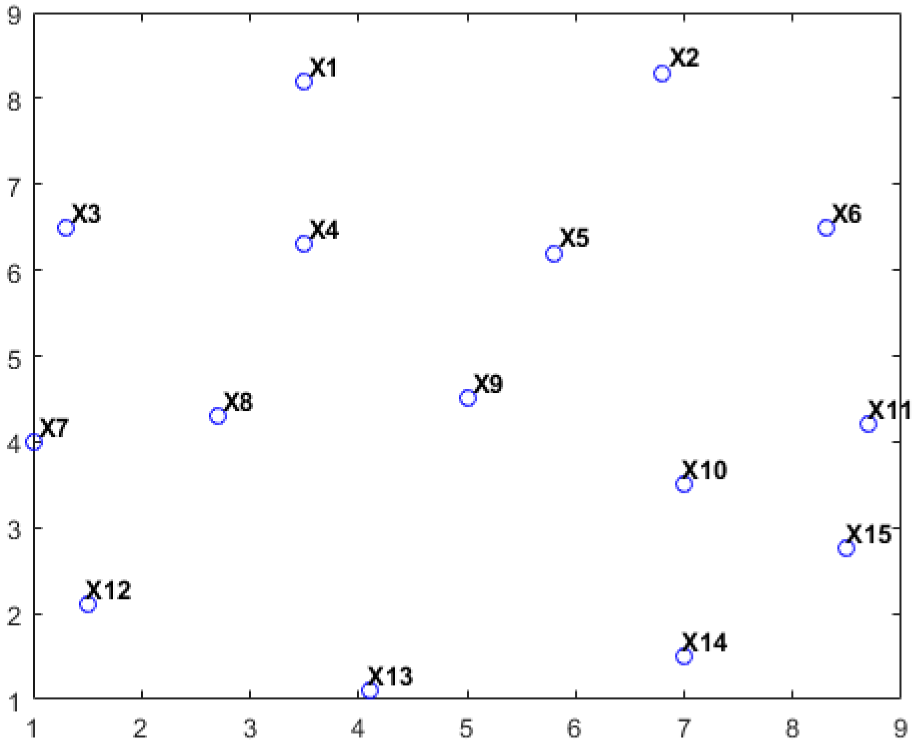
Begin by creating a random set of 15 points.

```
X = [3.5 8.2; 6.8 8.3; 1.3 6.5; 3.5 6.3; 5.8 6.2; 8.3 6.5;...
    1 4; 2.7 4.3; 5 4.5; 7 3.5; 8.7 4.2; 1.5 2.1; 4.1 1.1; ...
    7 1.5; 8.5 2.75];
```

Plot the points and add annotations to show the ID labels.

```
plot(X(:,1),X(:,2),'ob')
hold on
```

```
vxlabels = arrayfun(@(n) {sprintf('X%d', n)}, (1:15)');
Hpl = text(X(:,1)+0.2, X(:,2)+0.2, vxlabels, 'FontWeight', ...
  'bold', 'HorizontalAlignment','center', 'BackgroundColor', ...
  'none');
hold off
```



Create a Delaunay triangulation from the points.

```
dt = delaunayTriangulation(X);
```

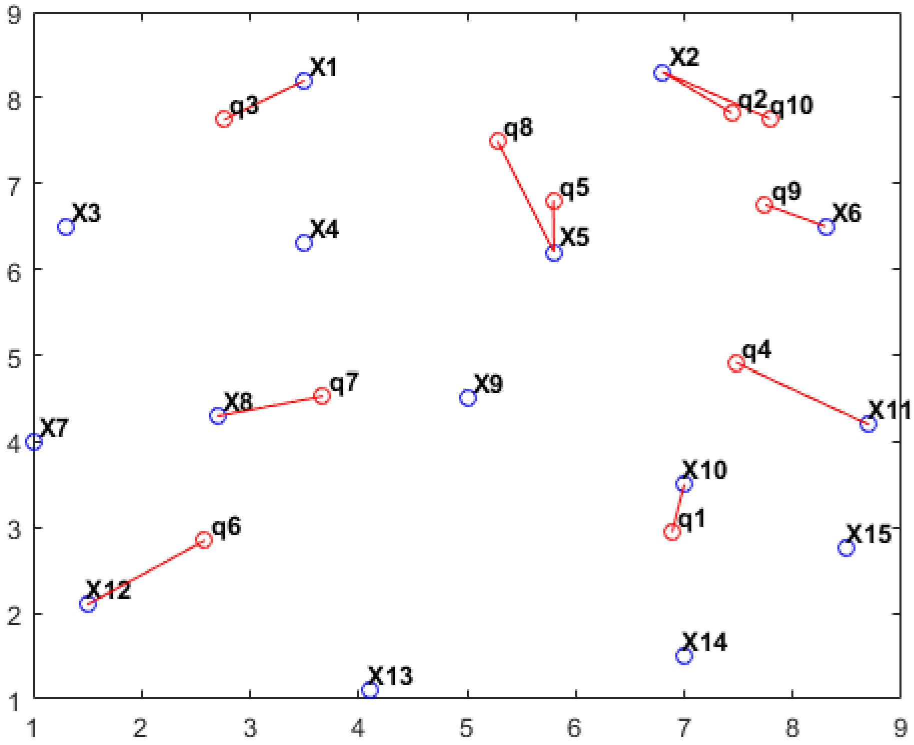
Create some query points and for each query point find the index of its corresponding nearest neighbor in X using the nearestNeighbor method.

```
numq = 10;
rng(0,'twister');
q = 2+rand(numq,2)*6;
xi = nearestNeighbor(dt, q);
```

Add the query points to the plot and add line segments joining the query points to their nearest neighbors.

```
xnn = X(xi,:);

hold on
plot(q(:,1),q(:,2),'or');
plot([xnn(:,1) q(:,1)]',[xnn(:,2) q(:,2)]','-r');

vxlabels = arrayfun(@(n) {sprintf('q%d', n)}, (1:numq)');
```

```
Hpl = text(q(:,1)+0.2, q(:,2)+0.2, vxlabels, 'FontWeight', ...
     'bold', 'HorizontalAlignment','center', ...
     'BackgroundColor','none');

hold off
```



Performing a nearest-neighbor search in 3-D is a direct extension of the 2-D example based on `delaunayTriangulation`.

For 4-D and higher, use the `delaunayn` and `dsearchn` functions as illustrated in the following example:

Create a random sample of points in 4-D and triangulate the points using `delaunayn`:

```
X = 20*rand(50,4) -10;
tri = delaunayn(X);
```

Create some query points and for each query point find the index of its corresponding nearest-neighbor in X using the `dsearchn` function:

```
q = rand(5,4);
xi = dsearchn(X,tri, q)
```

The `nearestNeighbor` method and the `dsearchn` function allow the Euclidean distance between the query point and its nearest-neighbor to be returned as an optional argument. In the 4-D example, you can compute the distances, `dnn`, as follows:

```
[xi,dnn] = dsearchn(X,tri, q)
```

Point-Location Search

A point-location search is a triangulation search algorithm that locates the simplex (triangle, tetrahedron, and so on) enclosing a query point. As in the case of the nearest-neighbor search, there are a few approaches to performing a point-location search in MATLAB, depending on the dimensionality of the problem:

- For 2-D and 3-D, use the class-based approach with the `pointLocation` method provided by the `triangulation` class and inherited by the `delaunayTriangulation` class.
- For 4-D and higher, use the `delaunayn` function to construct the triangulation and the complementary `tsearchn` function to perform the point-location search. Although supporting 2-D and 3-D, these N-D functions are not as general and efficient as the triangulation search methods.

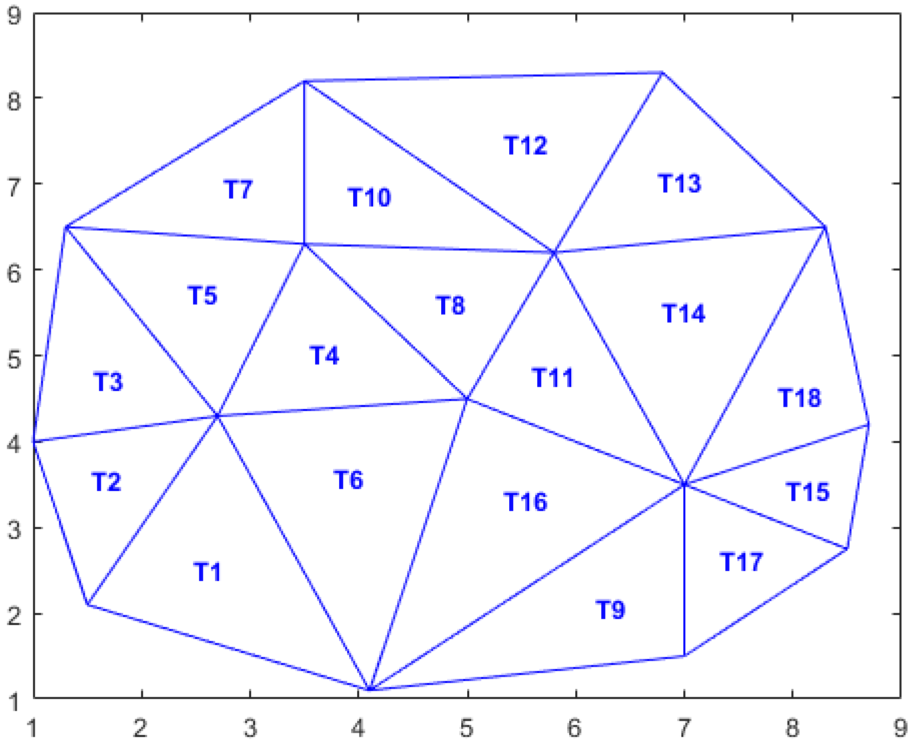
This example shows how to use the `delaunayTriangulation` class to perform a point location search in 2-D.

Begin with a set of 2-D points.

```
X = [3.5 8.2; 6.8 8.3; 1.3 6.5; 3.5 6.3; 5.8 6.2; ...
     8.3 6.5; 1 4; 2.7 4.3; 5 4.5; 7 3.5; 8.7 4.2; ...
     1.5 2.1; 4.1 1.1; 7 1.5; 8.5 2.75];
```

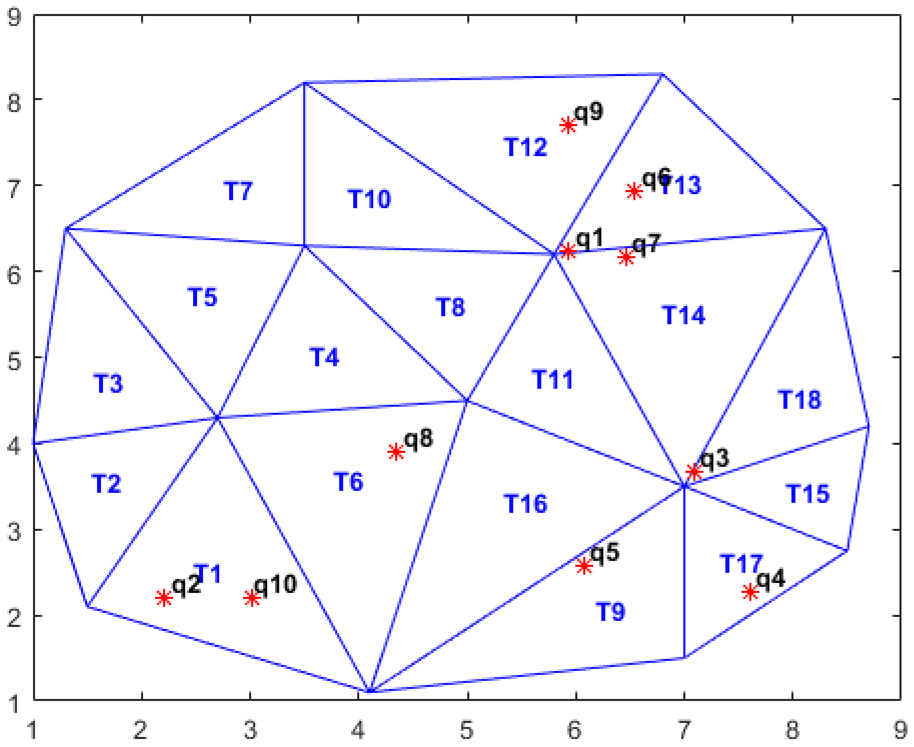
Create the triangulation and plot it showing the triangle ID labels at the incenters of the triangles.

```
 dt = delaunayTriangulation(X);
triplot(dt);

hold on
ic = incenter(dt);
numtri = size(dt,1);
trilabels = arrayfun(@(x) {sprintf('T%d', x)}, (1:numtri)');
Htl = text(ic(:,1), ic(:,2), trilabels, 'FontWeight', ...
      'bold', 'HorizontalAlignment', 'center', 'Color', ...
      'blue');
hold off
```



Now create some query points and add them to the plot. Then find the index of the corresponding enclosing triangles using the `pointLocation` method.

```
q = [5.9344    6.2363;
    2.2143    2.1910;
    7.0948    3.6615;
    7.6040    2.2770;
    6.0724    2.5828;
    6.5464    6.9407;
    6.4588    6.1690;
    4.3534    3.9026;
    5.9329    7.7013;
    3.0271    2.2067];

hold on;
plot(q(:,1),q(:,2),'*r');
vxlabels = arrayfun(@(n) {sprintf('q%d', n)}, (1:10)');
Hpl = text(q(:,1)+0.2, q(:,2)+0.2, vxlabels, 'FontWeight', ...
      'bold', 'HorizontalAlignment','center', ...
      'BackgroundColor', 'none');
hold off
```



```
ti = pointLocation(dt,q);
```

Performing a point-location search in 3-D is a direct extension of performing a point-location search in 2-D with `delaunayTriangulation`.

For 4-D and higher, use the `delaunayn` and `tsearchn` functions as illustrated in the following example:

Create a random sample of points in 4-D and triangulate them using `delaunayn`:

```
X = 20*rand(50,4) -10;
tri = delaunayn(X);
```

Create some query points and find the index of the corresponding enclosing simplices using the `tsearchn` function:

```
q = rand(5,4);
ti = tsearchn(X,tri,q)
```

The `pointLocation` method and the `tsearchn` function allow the corresponding barycentric coordinates to be returned as an optional argument. In the 4-D example, you can compute the barycentric coordinates as follows:

```
[ti,bc] = tsearchn(X,tri,q)
```

The barycentric coordinates are useful for performing linear interpolation. These coordinates provide you with weights that you can use to scale the values at each vertex of the enclosing simplex. See "Interpolating Scattered Data" on page 8-17 for further details.

See Also

`delaunay` | `delaunayTriangulation` | `delaunayn` | `dsearchn` | `nearestNeighbor` | `pointLocation` | `triangulation` | `triangulation` | `tsearchn`

Related Examples

- "Working with Delaunay Triangulations" on page 7-14
- "Triangulation Representations" on page 7-2
- "Interpolating Scattered Data" on page 8-17

Voronoi Diagrams

In this section...

The Voronoi diagram of a discrete set of points X decomposes the space around each point X(i) into a region of influence R{i}. This decomposition has the property that an arbitrary point P within the region R{i} is closer to point i than any other point. The region of influence is called a Voronoi region and the collection of all the Voronoi regions is the Voronoi diagram.

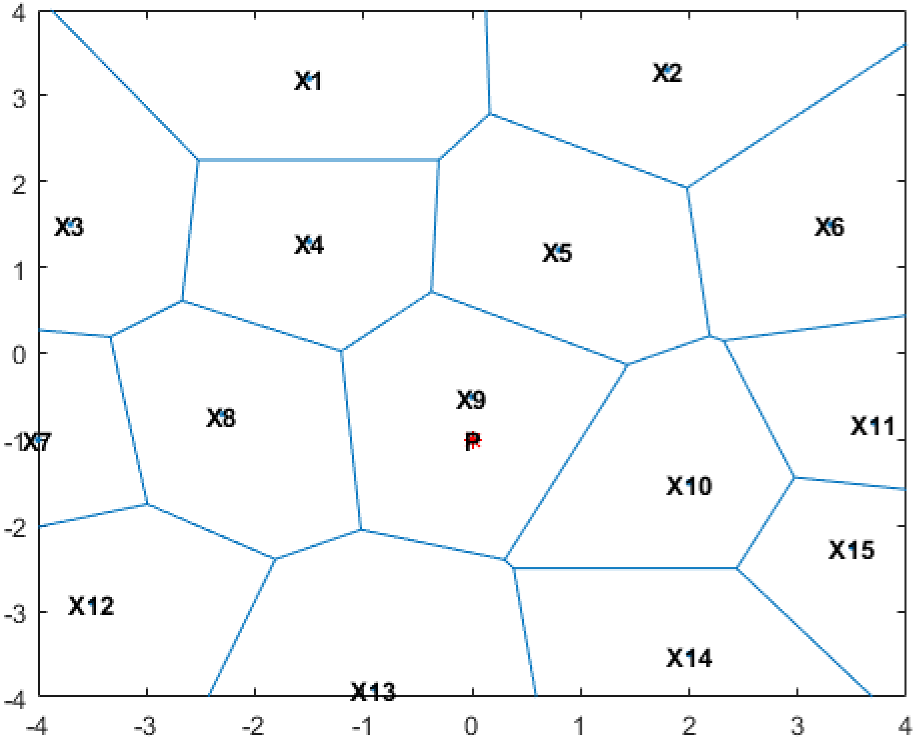
The Voronoi diagram is an N-D geometric construct, but most practical applications are in 2-D and 3-D space. The properties of the Voronoi diagram are best understood using an example.

Plot 2-D Voronoi Diagram and Delaunay Triangulation

This example shows the Voronoi diagram and the Delaunay triangulation on the same 2-D plot.

Use the 2-D voronoi function to plot the voronoi diagram for a set of points.

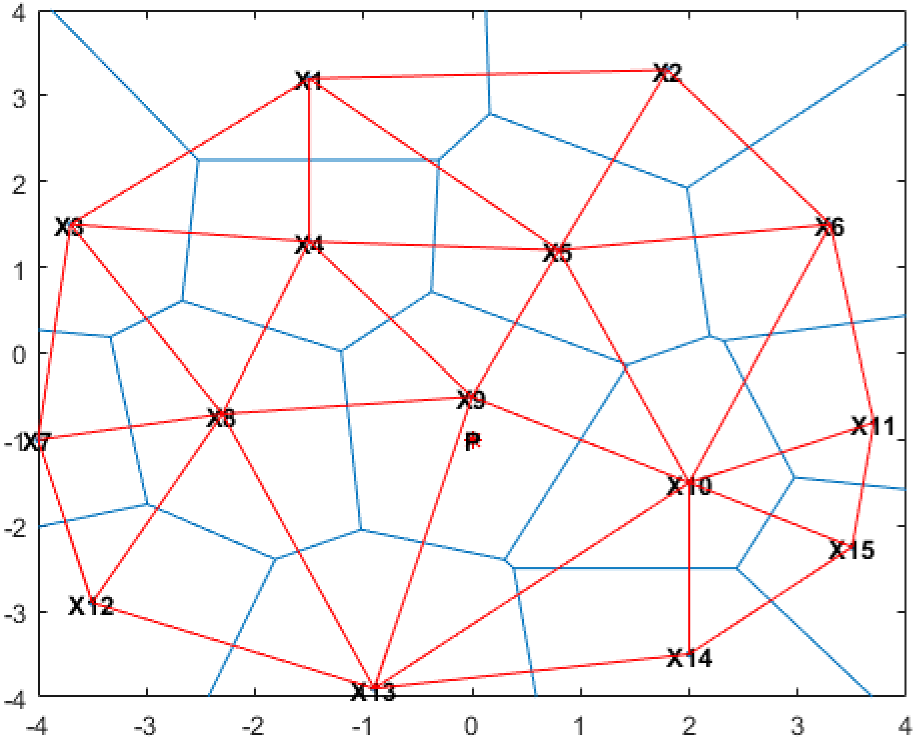
```
figure()
X = [-1.5 3.2; 1.8 3.3; -3.7 1.5; -1.5 1.3; ...
      0.8 1.2; 3.3 1.5; -4.0 -1.0;-2.3 -0.7; ...
      0 -0.5; 2.0 -1.5; 3.7 -0.8; -3.5 -2.9; ...
     -0.9 -3.9; 2.0 -3.5; 3.5 -2.25];

voronoi(X(:,1),X(:,2))

% Assign labels to the points.
nump = size(X,1);
plabels = arrayfun(@(n) {sprintf('X%d', n)}, (1:nump)');
hold on
Hpl = text(X(:,1), X(:,2), plabels, 'FontWeight', ...
      'bold', 'HorizontalAlignment','center', ...
      'BackgroundColor', 'none');

% Add a query point, P, at (0, -1.5).
P = [0 -1];
plot(P(1),P(2), '*r');
text(P(1), P(2), 'P', 'FontWeight', 'bold', ...
     'HorizontalAlignment','center', ...
     'BackgroundColor', 'none');
hold off
```



Observe that P is closer to X9 than to any other point in X, which is true for any point P within the region that bounds X9.

The Voronoi diagram of a set of points X is closely related to the Delaunay triangulation of X. To see this relationship, construct a Delaunay triangulation of the point set X and superimpose the triangulation plot on the Voronoi diagram.

```
dt = delaunayTriangulation(X);
hold on
triplot(dt,'-r');
hold off
```



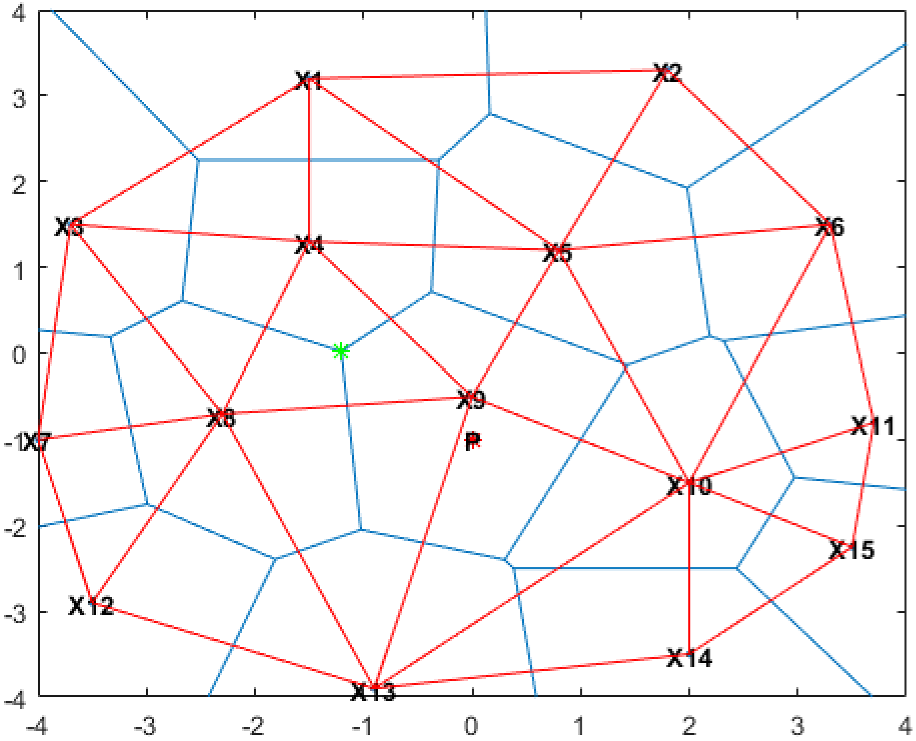
From the plot you can see that the Voronoi region associated with the point X9 is defined by the perpendicular bisectors of the Delaunay edges attached to X9. Also, the vertices of the Voronoi edges are located at the circumcenters of the Delaunay triangles. You can illustrate these associations by plotting the circumcenter of triangle {|X9|,|X4|,|X8|}.

To find the index of this triangle, query the triangulation. The triangle contains the location (-1, 0).

```
tidx = pointLocation(dt,-1,0);
```

Now, find the circumcenter of this triangle and plot it in green.

```
cc = circumcenter(dt,tidx);
hold on
plot(cc(1),cc(2),'*g');
hold off
```



The Delaunay triangulation and Voronoi diagram are geometric duals of each other. You can compute the Voronoi diagram from the Delaunay triangulation and vice versa.

Observe that the Voronoi regions associated with points on the convex hull are unbounded (for example, the Voronoi region associated with `X13`). The edges in this region "end" at infinity. The Voronoi edges that bisect Delaunay edges (`X13`, `X12`) and (`X13`, `X14`) extend to infinity. While the Voronoi diagram provides a nearest-neighbor decomposition of the space around each point in the set, it does not directly support nearest-neighbor queries. However, the geometric constructions used to compute the Voronoi diagram are also used to perform nearest-neighbor searches.

Computing the Voronoi Diagram

This example shows how to compute a 2–D and 3–D Voronoi diagram.

MATLAB provides functions to plot the Voronoi diagram in 2-D and to compute the topology of the Voronoi diagram in N-D. In practice, Voronoi computation is not practical in dimensions beyond 6-D for moderate to large data sets, due to the exponential growth in required memory.

The `voronoi` plot function plots the Voronoi diagram for a set of points in 2-D space. In MATLAB there are two ways to compute the topology of the Voronoi diagram of a point set:

- Using the function `voronoin`
- Using the `delaunayTriangulation` method, `voronoiDiagram`.

The voronoin function supports the computation of the Voronoi topology for discrete points in N-D (N ≥ 2). The voronoiDiagram method supports computation of the Voronoi topology for discrete points 2-D or 3-D.

The voronoiDiagram method is recommended for 2-D or 3-D topology computations as it is more robust and gives better performance for large data sets. This method supports incremental insertion and removal of points and complementary queries, such as nearest-neighbor point search.

The voronoin function and the voronoiDiagram method represent the topology of the Voronoi diagram using a matrix format. See “Triangulation Matrix Format” on page 7-3 for further details on this data structure.

Given a set of points, X, obtain the topology of the Voronoi diagram as follows:

- Using the voronoin function

```
[V,R] = voronoin(X)
```

- Using the voronoiDiagram method

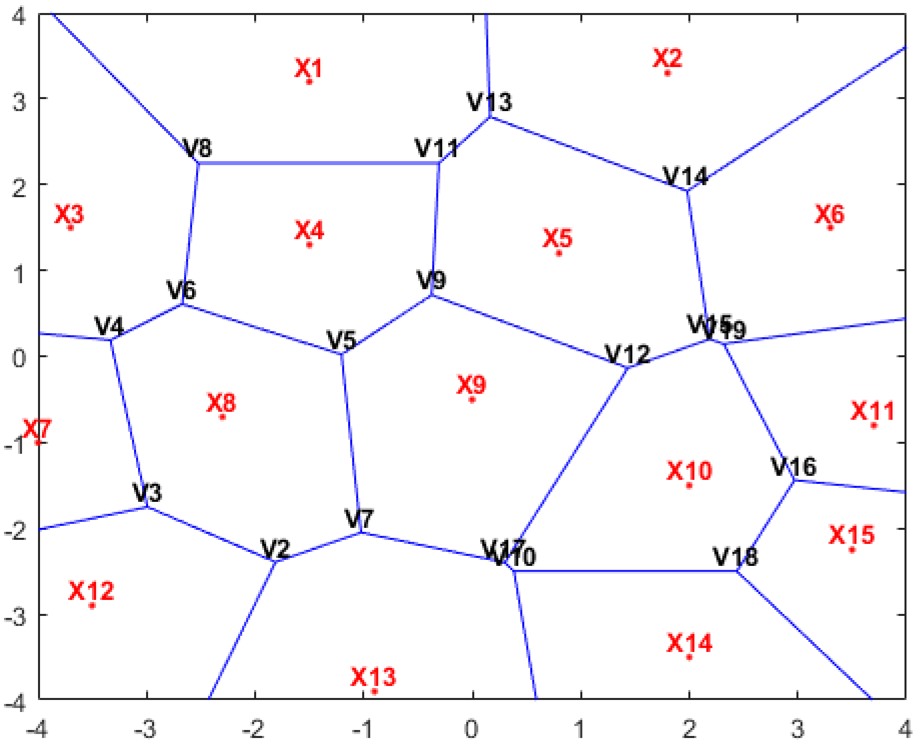
```
dt = delaunayTriangulation(X);

[V,R] = voronoiDiagram(dt)
```

V is a matrix representing the coordinates of the Voronoi vertices (the vertices are the end points of the Voronoi edges). By convention the first vertex in V is the infinite vertex. R is a vector cell array length size(X,1), representing the Voronoi region associated with each point. Hence, the Voronoi region associated with the point X(i) is R{i}.

Define and plot the voronoi diagram for a set of points

```
X = [-1.5 3.2; 1.8 3.3; -3.7 1.5; -1.5 1.3; 0.8 1.2; ...
      3.3 1.5; -4.0 -1.0; -2.3 -0.7; 0 -0.5; 2.0 -1.5; ...
      3.7 -0.8; -3.5 -2.9; -0.9 -3.9; 2.0 -3.5; 3.5 -2.25];
[VX,VY] = voronoi(X(:,1),X(:,2));
h = plot(VX,VY,'-b',X(:,1),X(:,2),'.r');
xlim([-4,4])
ylim([-4,4])

% Assign labels to the points X.
nump = size(X,1);
plabels = arrayfun(@(n) {sprintf('X%d', n)}, (1:nump)');
hold on
Hpl = text(X(:,1), X(:,2)+0.2, plabels, 'color', 'r', ...
      'FontWeight', 'bold', 'HorizontalAlignment',...
      'center', 'BackgroundColor', 'none');

% Compute the Voronoi diagram.
dt = delaunayTriangulation(X);
[V,R] = voronoiDiagram(dt);


% Assign labels to the Voronoi vertices V.
% By convention the first vertex is at infinity.
numv = size(V,1);
vlabels = arrayfun(@(n) {sprintf('V%d', n)}, (2:numv)');
hold on
```

```
Hpl = text(V(2:end,1), V(2:end,2)+.2, vlabels, ...
      'FontWeight', 'bold', 'HorizontalAlignment',...
      'center', 'BackgroundColor', 'none');
hold off
```



`R{9}` gives the indices of the Voronoi vertices associated with the point site `X9`.

```
R{9}
```

```
ans = 1×5

     5     7    17    12     9
```

The indices of the Voronoi vertices are the indices with respect to the `V` array.

Similarly, `R{4}` gives the indices of the Voronoi vertices associated with the point site `X4`.

```
R{4}
```

```
ans = 1×5

     5     9    11     8     6
```

In 3-D a Voronoi region is a convex polyhedron, the syntax for creating the Voronoi diagram is similar. However the geometry of the Voronoi region is more complex. The following example illustrates the creation of a 3-D Voronoi diagram and the plotting of a single region.

Create a sample of 25 points in 3-D space and compute the topology of the Voronoi diagram for this point set.

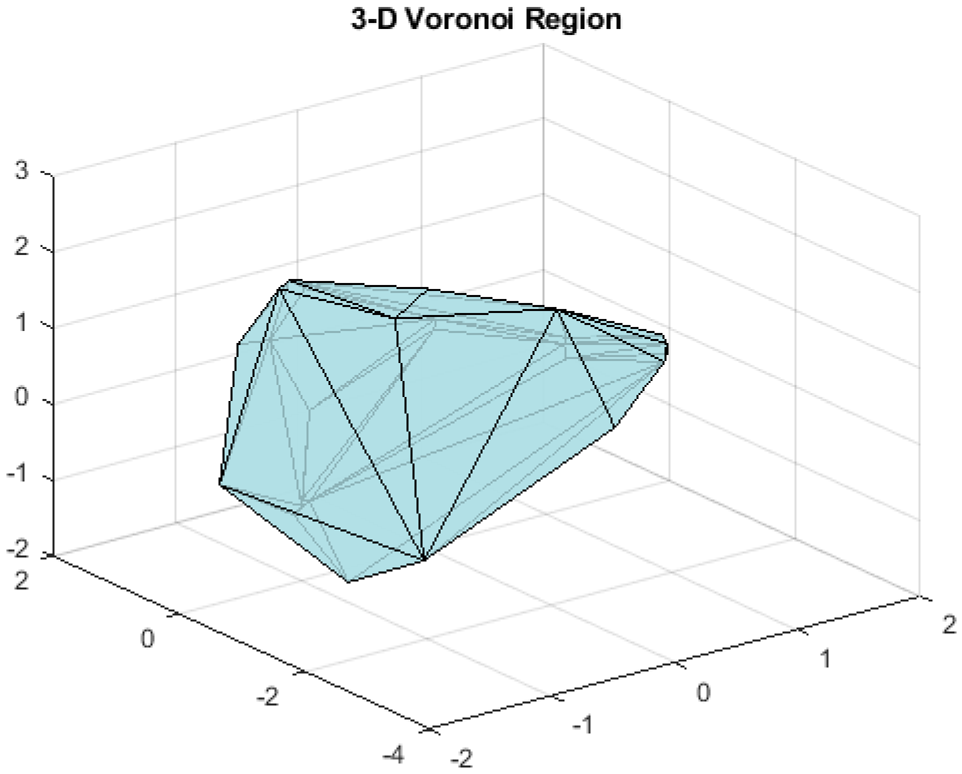
```
rng('default')
X = -3 + 6.*rand([25 3]);
dt = delaunayTriangulation(X);
```

Compute the topology of the Voronoi diagram.

```
[V,R] = voronoiDiagram(dt);
```

Find the point closest to the origin and plot the Voronoi region associated with this point.

```
tid = nearestNeighbor(dt,0,0,0);
XR10 = V(R{tid},:);
K = convhull(XR10);
defaultFaceColor  = [0.6875 0.8750 0.8984];
trisurf(K, XR10(:,1) ,XR10(:,2) ,XR10(:,3) , ...
        'FaceColor', defaultFaceColor, 'FaceAlpha',0.8)
title('3-D Voronoi Region')
```



See Also

More About

- “Spatial Searching” on page 7-53

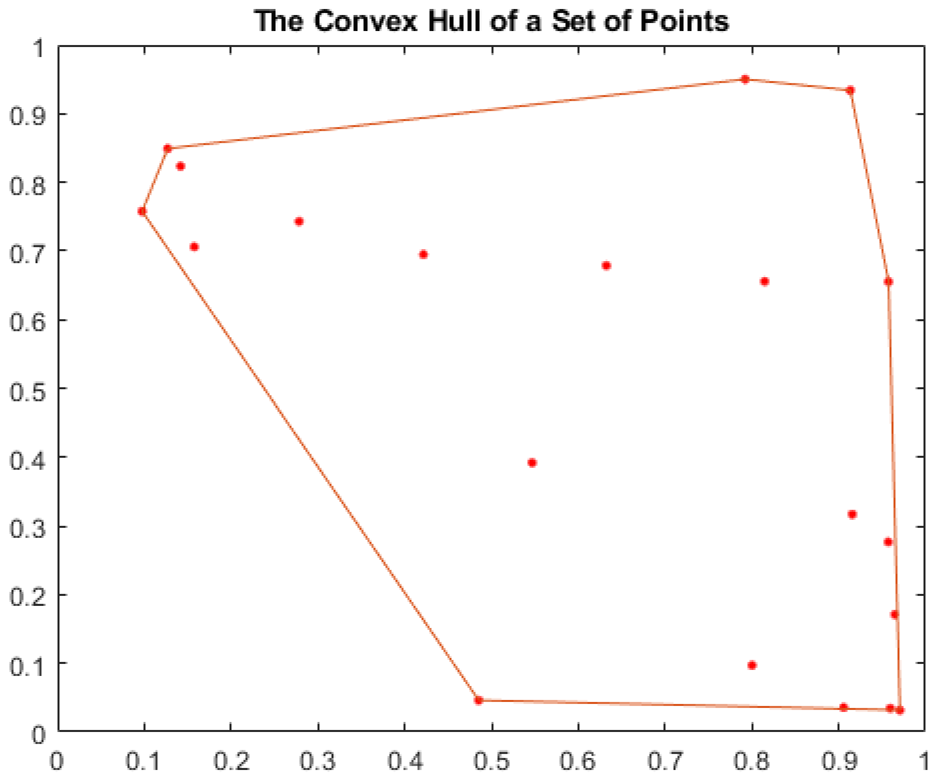
Types of Region Boundaries

In this section...

Convex Hulls vs. Nonconvex Polygons

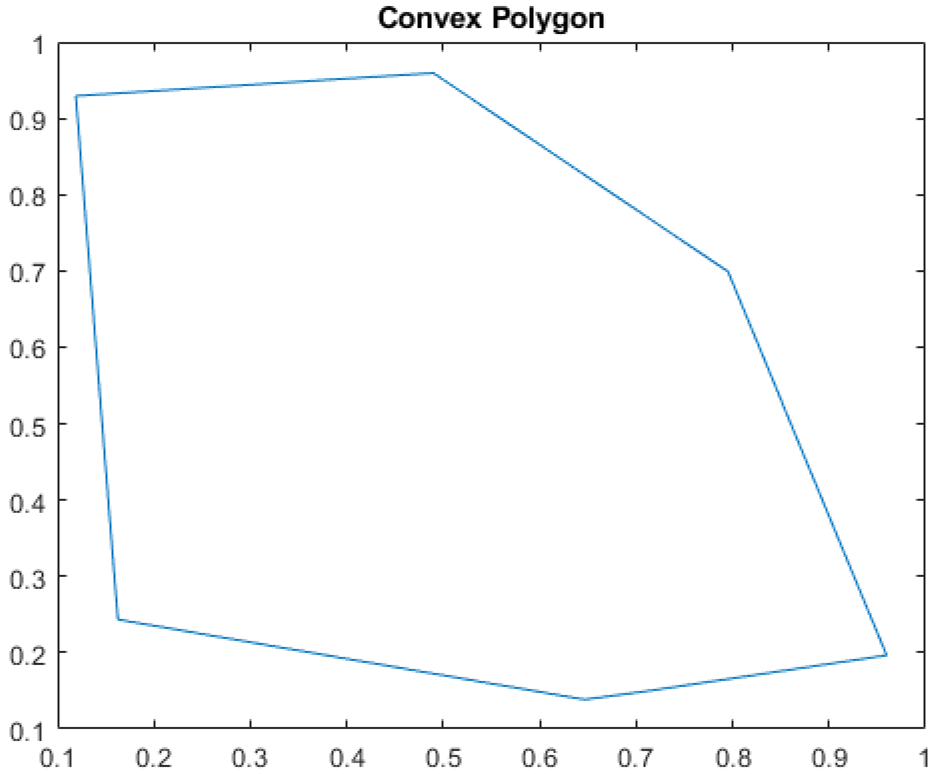
The convex hull of a set of points in N-D space is the smallest convex region enclosing all points in the set. If you think of a 2-D set of points as pegs in a peg board, the convex hull of that set would be formed by taking an elastic band and using it to enclose all the pegs.

```
rng('default')
x = rand(20,1);
y = rand(20,1);
plot(x,y,'r.','MarkerSize',10)
hold on
k = convhull(x,y);
plot(x(k),y(k))
title('The Convex Hull of a Set of Points')
hold off
```



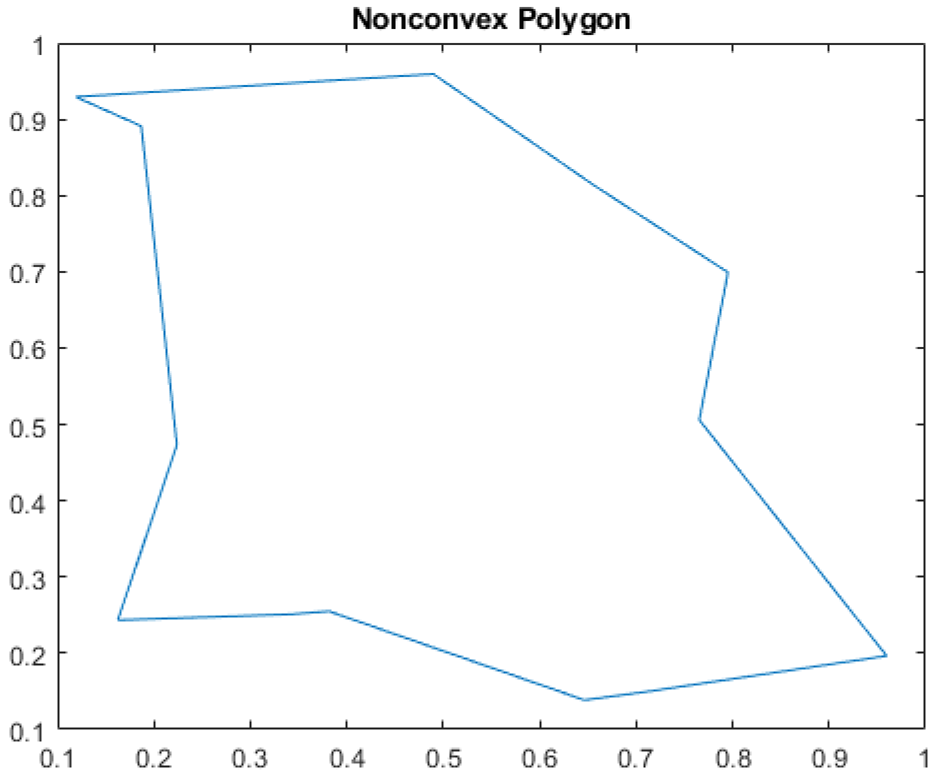
A convex polygon is a polygon that does not have concave vertices, for example:

```
x = rand(20,1);
y = rand(20,1);
k = convhull(x,y);
plot(x(k),y(k))
title('Convex Polygon')
```



You can also create a boundary of a point set that is nonconvex. If you shrink and tighten the convex hull from above, you can enclose all of the points in a nonconvex polygon with concave vertices:

```
k = boundary(x,y,0.9);
plot(x(k),y(k))
title('Nonconvex Polygon')
```



The convex hull has numerous applications. You can compute the upper bound on the area bounded by a discrete point set in the plane from the convex hull of the set. The convex hull simplifies the representation of more complex polygons or polyhedra. For instance, to determine whether two nonconvex bodies intersect, you could apply a series of fast rejection steps to avoid the penalty of a full intersection analysis:

- Check if the axis-aligned bounding boxes around each body intersect.
- If the bounding boxes intersect, you can compute the convex hull of each body and check intersection of the hulls.

If the convex hulls did not intersect, this would avoid the expense of a more comprehensive intersection test.

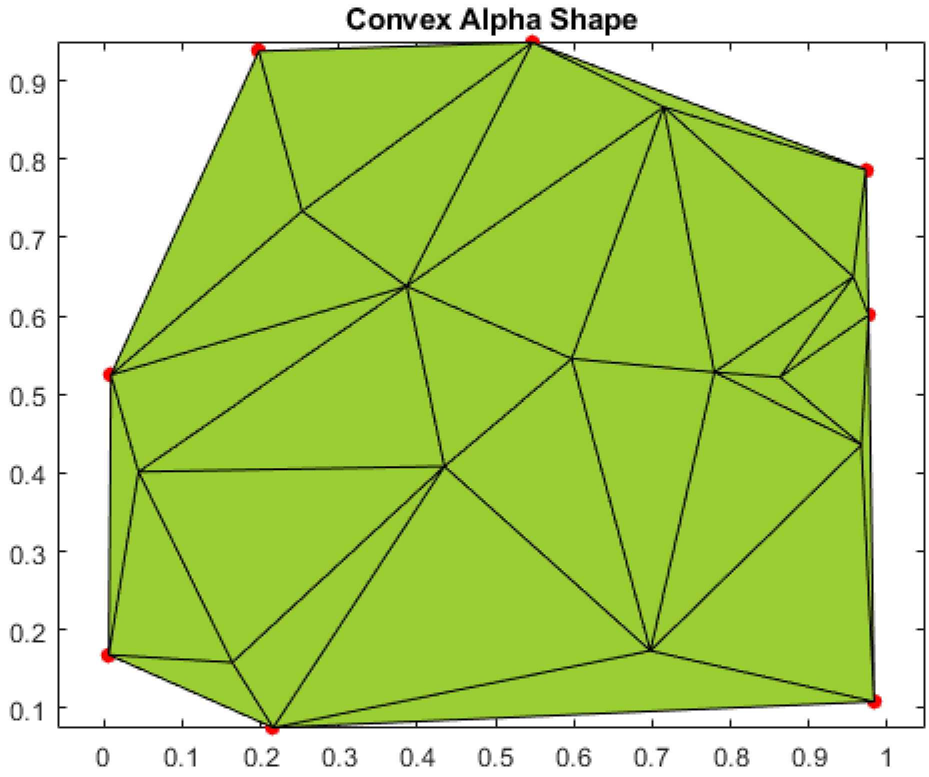
While convex hulls and nonconvex polygons are convenient ways to represent relatively simple boundaries, they are in fact specific instances of a more general geometric construct called the alpha shape.

Alpha Shapes

The alpha shape of a set of points is a *generalization* of the convex hull and a subgraph of the Delaunay triangulation. That is, the convex hull is just one type of alpha shape, and the full family of alpha shapes can be derived from the Delaunay triangulation of a given point set.

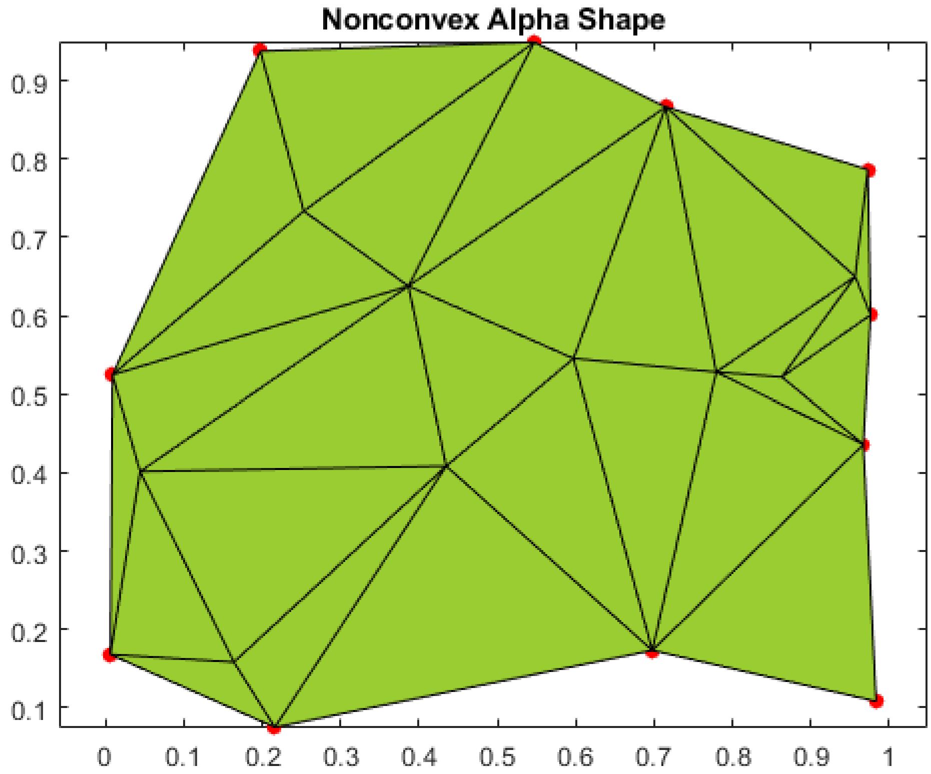
```
rng(4)
x = rand(20,1);
```

```
y = rand(20,1);
plot(x,y,'r.','MarkerSize',20)
hold on
shp = alphaShape(x,y,100);
plot(shp)
title('Convex Alpha Shape')
hold off
```



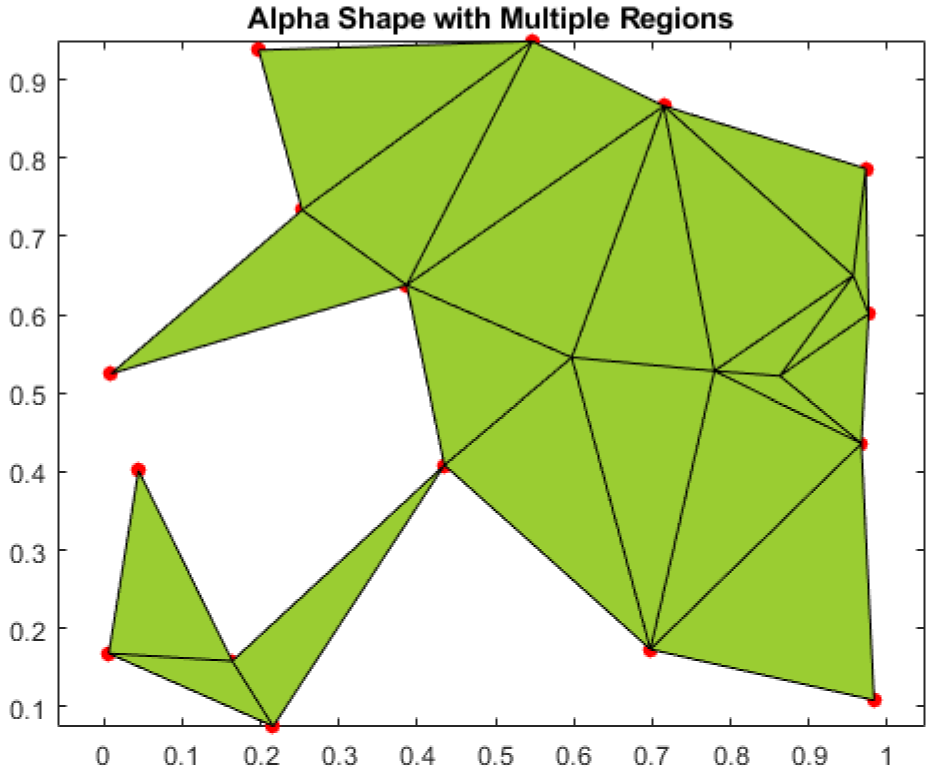
Unlike the convex hull, alpha shapes have a parameter that controls the level of detail, or how tightly the boundary fits around the point set. The parameter is called *alpha* or the *alpha radius*. Varying the alpha radius from 0 to `Inf` produces a set of different alpha shapes unique for that point set.

```
plot(x,y,'r.','MarkerSize',20)
hold on
shp = alphaShape(x,y,.5);
plot(shp)
title('Nonconvex Alpha Shape')
hold off
```



Varying the alpha radius can sometimes result in an alpha shape with multiple regions, which might or might not contain holes. However, the `alphaShape` function in MATLAB® always returns regularized alpha shapes, which prevents isolated or dangling points, edges, or faces.

```
plot(x,y,'r.','MarkerSize',20)
hold on
shp = alphaShape(x,y);
plot(shp)
title('Alpha Shape with Multiple Regions')
hold off
```



See Also

`alphaShape` | `convhull`

More About

- “Using the delaunayTriangulation Class” on page 7-20
- “Triangulation Matrix Format” on page 7-3

Computing the Convex Hull

In this section...

MATLAB provides several ways to compute the convex hull:

- Using the MATLAB functions `convhull` and `convhulln`
- Using the `convexHull` method provided by the `delaunayTriangulation` class
- Using the `alphaShape` function with an alpha radius of `Inf`.

The `convhull` function supports the computation of convex hulls in 2-D and 3-D. The `convhulln` function supports the computation of convex hulls in N-D (N ≥ 2). The `convhull` function is recommended for 2-D or 3-D computations due to better robustness and performance.

The `delaunayTriangulation` class supports 2-D or 3-D computation of the convex hull from the Delaunay triangulation. This computation is not as efficient as the dedicated `convhull` and `convhulln` functions. However, if you have a `delaunayTriangulation` of a point set and require the convex hull, the `convexHull` method can compute the convex hull more efficiently from the existing triangulation.

The `alphaShape` function also supports the 2-D or 3-D computation of the convex hull by setting the alpha radius input parameter to `Inf`. Like `delaunayTriangulation`, however, computing the convex hull using `alphaShape` is less efficient than using `convhull` or `convhulln` directly. The exception is when you are working with a previously created alpha shape object.

Computing the Convex Hull Using convhull and convhulln

The `convhull` and `convhulln` functions take a set of points and output the indices of the points that lie on the boundary of the convex hull. The point index-based representation of the convex hull supports plotting and convenient data access. The following examples illustrate the computation and representation of the convex hull.

The first example uses a 2-D point set from the seamount dataset as input to the convhull function.

Load the data.

```
load seamount
```

Compute the convex hull of the point set.

```
K = convhull(x,y);
```

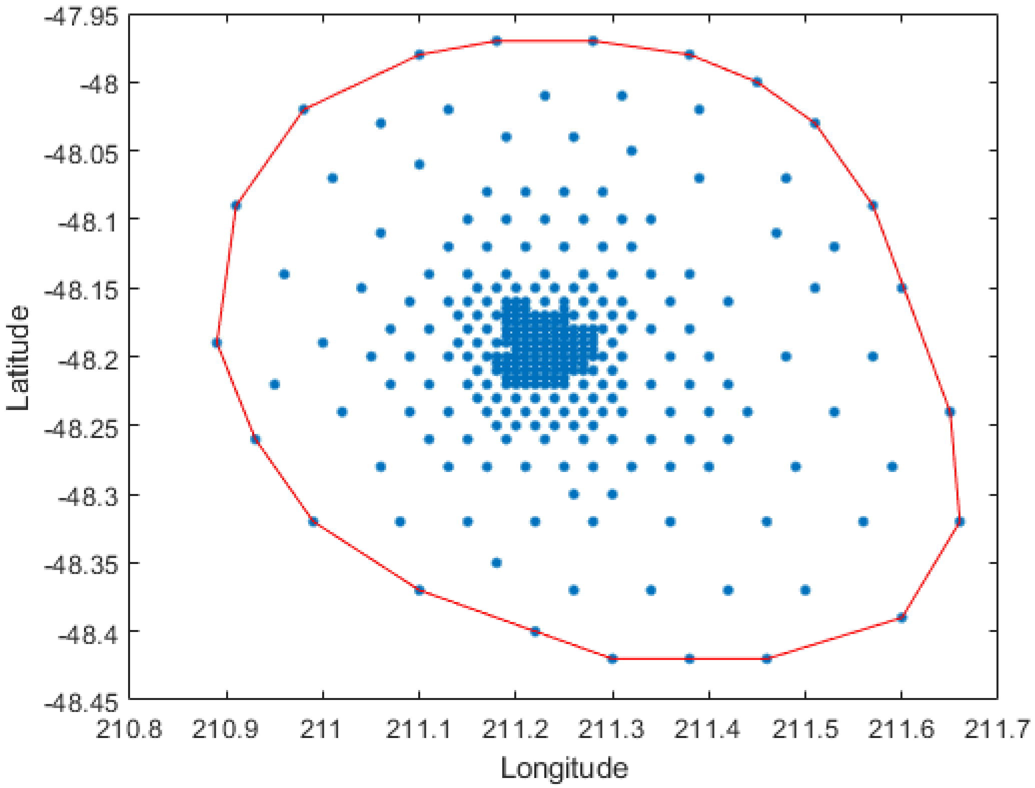
`K` represents the indices of the points arranged in a counter-clockwise cycle around the convex hull.

Plot the data and its convex hull.

```
plot(x,y,'.','markersize',12)
xlabel('Longitude')
ylabel('Latitude')
```

```
hold on
```

```
plot(x(K),y(K),'r')
```



Add point labels to the points on the convex hull to observe the structure of K.

```
[K,A] = convhull(x,y);
```

convhull can compute the convex hull of both 2-D and 3-D point sets. You can reuse the seamount dataset to illustrate the computation of the 3-D convex hull.

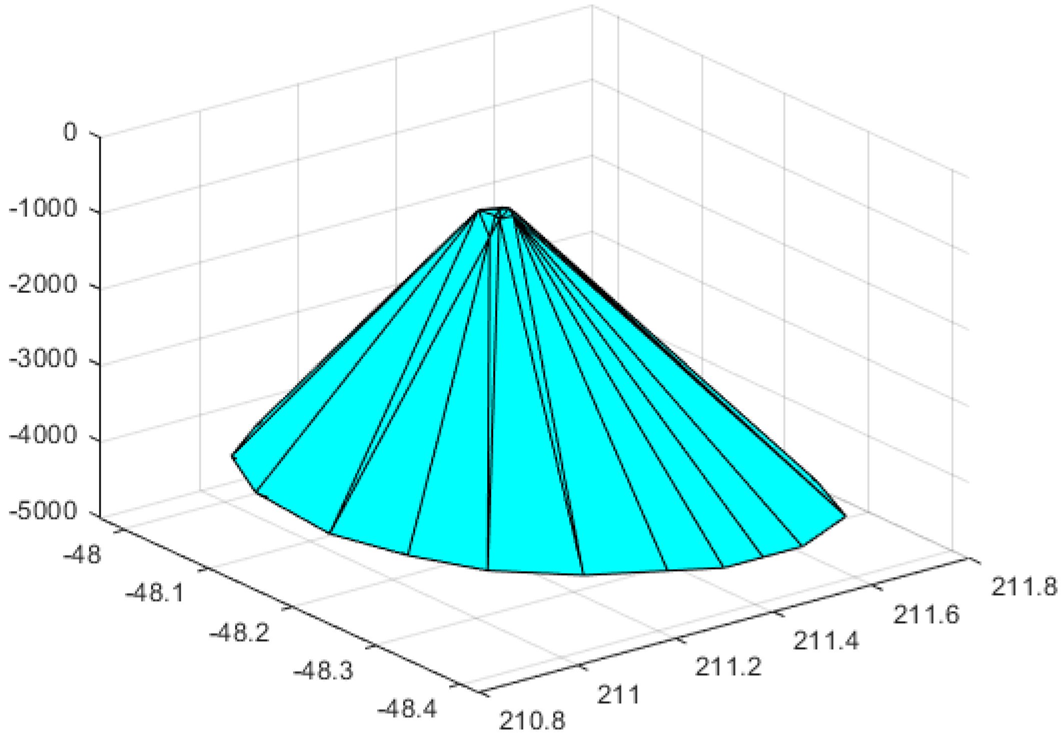
Include the seamount z-coordinate data elevations.

```
close(gcf)
K = convhull(x,y,z);
```

In 3-D the boundary of the convex hull, K, is represented by a triangulation. This is a set of triangular facets in matrix format that is indexed with respect to the point array. Each row of the matrix K represents a triangle.

Since the boundary of the convex hull is represented as a triangulation, you can use the triangulation plotting function trisurf.

```
trisurf(K,x,y,z,'Facecolor','cyan')
```

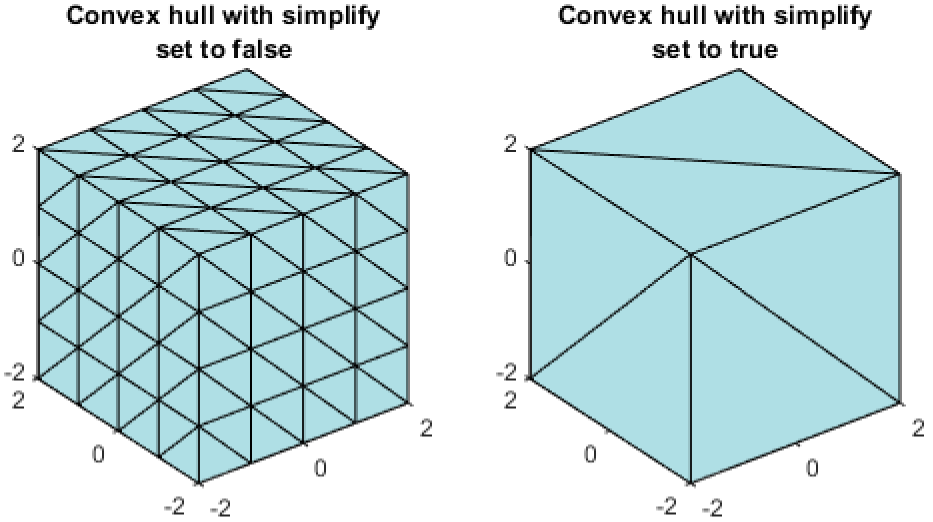


The volume bounded by the 3-D convex hull can optionally be returned by `convhull`, the syntax is as follows.

```
[K,V] = convhull(x,y,z);
```

The `convhull` function also provides the option of simplifying the representation of the convex hull by removing vertices that do not contribute to the area or volume. For example, if boundary facets of the convex hull are collinear or coplanar, you can merge them to give a more concise representation. The following example illustrates use of this option.

```
[x,y,z] = meshgrid(-2:1:2,-2:1:2,-2:1:2);
x = x(:);
y = y(:);
z = z(:);

K1 = convhull(x,y,z);
subplot(1,2,1)
defaultFaceColor  = [0.6875 0.8750 0.8984];
trisurf(K1,x,y,z,'Facecolor',defaultFaceColor)
axis equal
title(sprintf('Convex hull with simplify\nset to false'))

K2 = convhull(x,y,z,'simplify',true);
subplot(1,2,2)
trisurf(K2,x,y,z,'Facecolor',defaultFaceColor)
axis equal
title(sprintf('Convex hull with simplify\nset to true'))
```


Convex hull with simplify set to false
Convex hull with simplify set to true
2
0
-2
2
0
-2
-2
0
2

MATLAB provides the `convhulln` function to support the computation of convex hulls and hypervolumes in higher dimensions. Though `convhulln` supports N-D, problems in more than 10 dimensions present challenges due to the rapidly growing memory requirements.

The `convhull` function is superior to `convhulln` in 2-D and 3-D as it is more robust and gives better performance.

Convex Hull Computation Using the delaunayTriangulation Class

This example shows the relationship between a Delaunay triangulation of a set of points in 2-D and the convex hull of that set of points.

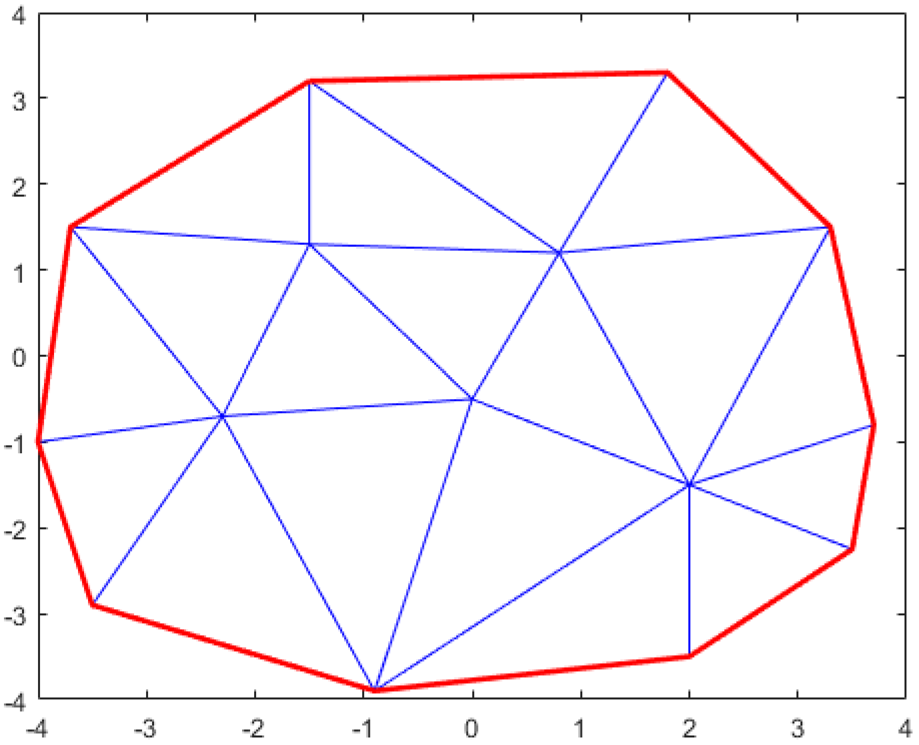
The `delaunayTriangulation` class supports computation of Delaunay triangulations in 2-D and 3-D space. This class also provides a `convexHull` method to derive the convex hull from the triangulation.

Create a Delaunay triangulation of a set of points in 2-D.

```
X = [-1.5 3.2; 1.8 3.3; -3.7 1.5; -1.5 1.3; 0.8 1.2; ...
      3.3 1.5; -4.0 -1.0; -2.3 -0.7; 0 -0.5; 2.0 -1.5; ...
      3.7 -0.8; -3.5 -2.9; -0.9 -3.9; 2.0 -3.5; 3.5 -2.25];

dt = delaunayTriangulation(X);
```

Plot the triangulation and highlight the edges that are shared only by a single triangle reveals the convex hull.

```
triplot(dt)

fe = freeBoundary(dt)';
hold on
plot(X(fe,1), X(fe,2), '-r', 'LineWidth',2)
hold off
```



In 3-D, the facets of the triangulation that are shared only by one tetrahedron represent the boundary of the convex hull.

The dedicated `convhull` function is generally more efficient than a computation based on the `convexHull` method. However, the triangulation based approach is appropriate if:

- You have a `delaunayTriangulation` of the point set already and the convex hull is also required.
- You need to add or remove points from the set incrementally and need to recompute the convex hull frequently after you have edited the points.

Convex Hull Computation Using alphaShape

This example shows how to compute the convex hull of a 2-D point set using the `alphaShape` function.

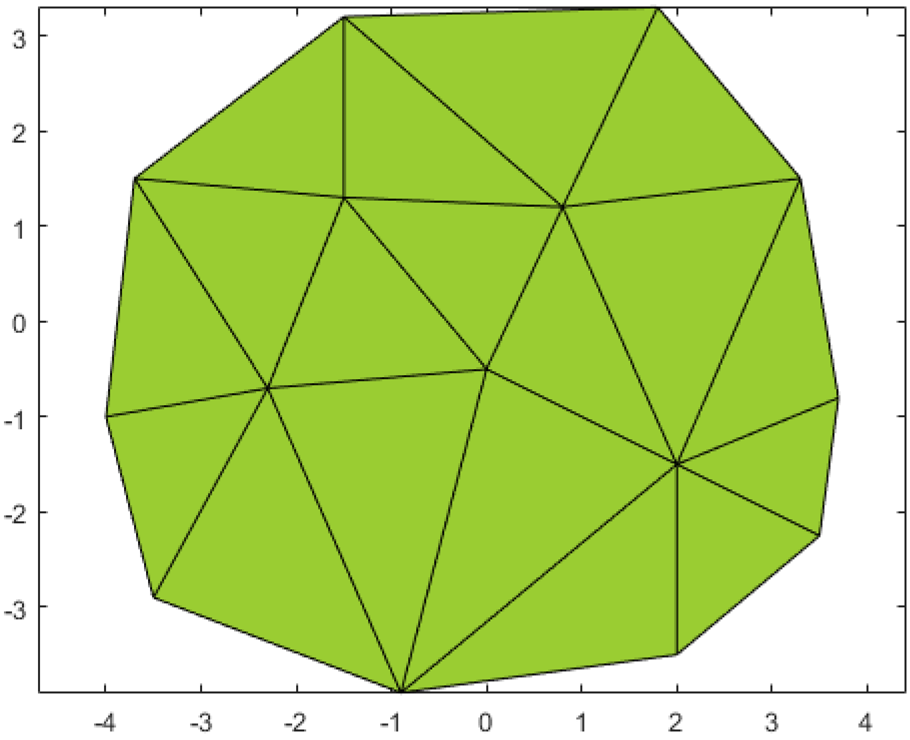
alphaShape computes a regularized alpha shape from a set of 2-D or 3-D points. You can specify the alpha radius, which determines how tightly or loosely the alpha shape envelops the point set. When the alpha radius is set to Inf, the resulting alpha shape is the convex hull of the point set.

Create a set of 2-D points.

```
X = [-1.5 3.2; 1.8 3.3; -3.7 1.5; -1.5 1.3; 0.8 1.2; ...
      3.3 1.5; -4.0 -1.0; -2.3 -0.7; 0 -0.5; 2.0 -1.5; ...
      3.7 -0.8; -3.5 -2.9; -0.9 -3.9; 2.0 -3.5; 3.5 -2.25];
```

Compute and plot the convex hull of the point set using an alpha shape with alpha radius equal to Inf.

```
shp = alphaShape(X,Inf);
plot(shp)
```



See Also

alphaShape | convexHull | convhull | convhulln | delaunayTriangulation

Related Examples

- “Working with Delaunay Triangulations” on page 7-14

Interpolation

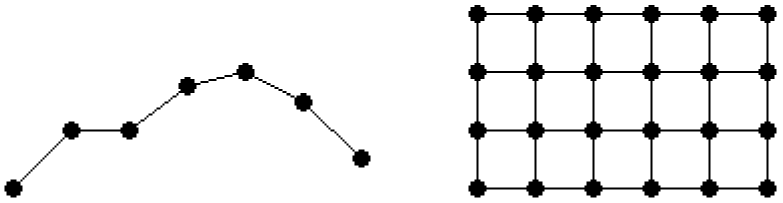
Gridded and Scattered Sample Data

Interpolation is a method to estimate the value of a function at a query location that lies within the domain of a set of sample data points. The function value is calculated based on the sample data points that are closest to the query point. MATLAB can perform two kinds of interpolation depending on the structure of the sample data. The sample data can form a grid, or can be scattered.

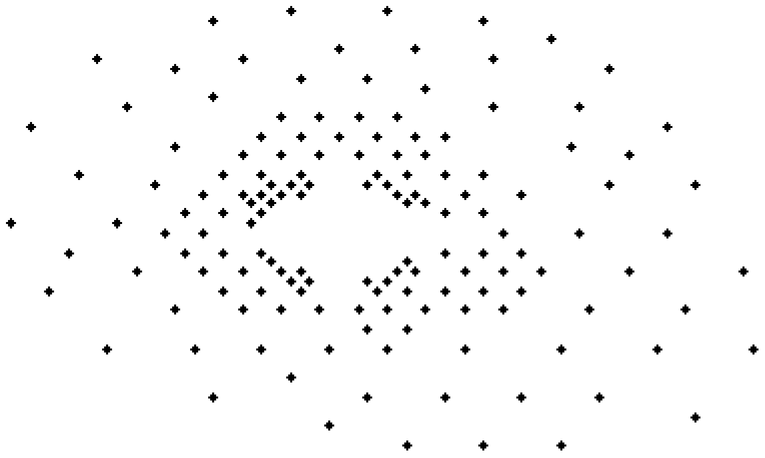
Gridded sample data makes interpolation more efficient, because the organized structure of the data makes it easy for MATLAB to find the sample data points closest to the query point. However, interpolating scattered data requires a “Delaunay Triangulation” of the data points, and this introduces an extra layer of computation. Therefore, if your data can be approximated as a grid, gridded interpolation provides substantial savings in computation time and memory usage compared to scattered interpolation.

The two approaches to interpolation are covered in the following topics:

- “Interpolating Gridded Data” on page 8-5 covers 1-D interpolation, and the N-D interpolation of sample data that is in axis-aligned grid format:

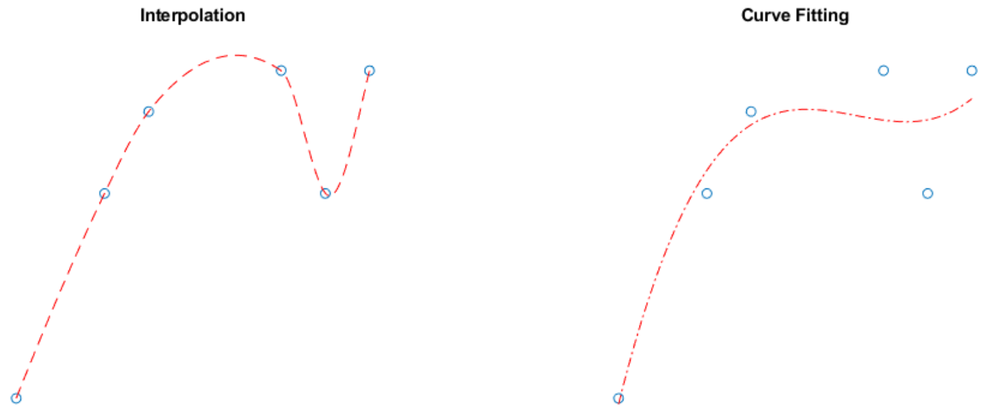


- “Interpolating Scattered Data” on page 8-17 covers the N-D interpolation of scattered data:



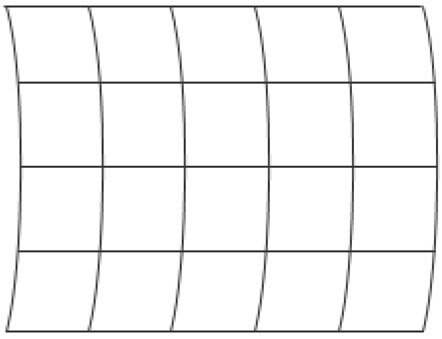
Interpolation versus Curve Fitting

The interpolation methods available in MATLAB create interpolating functions that pass through the sample data points. That is, if you query the interpolation function at a sample location, you get back the exact sample data value and not an approximation. By contrast, curve and surface fitting algorithms do not necessarily pass through the sample data points. For more information about curve fitting, see Curve Fitting Toolbox.

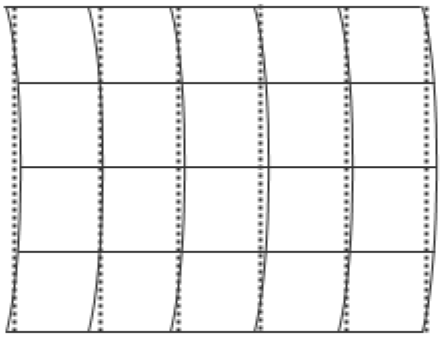


Grid Approximation Techniques

In some cases, you may need to approximate a grid for your data. For example, a grid can have points that lie along curved lines. A data set like this might occur if your data is longitude and latitude based:



With a curved grid, you are effectively dealing with a set of scattered data and must use more computationally expensive scattered interpolation functions to interpolate the values. However, although the input data cannot be gridded directly, it is sometimes feasible to approximate the curved grid with straight grid lines at appropriate intervals:



You can create an approximate grid by creating a set of grid vectors with appropriate spacing. Approximating a curved grid with straight lines allows you to get the performance benefits of grid-based interpolation, at the cost of slightly distorting the data. For more information about creating grid vectors, see “Grid Representations” on page 8-5.

See Also

`griddedInterpolant` | `scatteredInterpolant`

Related Examples

- “Interpolating Gridded Data” on page 8-5
- “Interpolating Scattered Data” on page 8-17

Interpolating Gridded Data

Gridded data consists of values or measurements at regularly spaced points that form a grid. Gridded data arises in many areas, such as meteorology, surveying, and medical imaging. In these areas, it is common to take measurements at regular spatial intervals, possibly over time. These ordered grids of data can range from 1-D (for simple time series) to 4-D (for measuring volumes over time) or higher. Some examples of gridded data are:

- 1-D: Stock prices over time
- 2-D: Temperature of a surface
- 3-D: MRI image of a brain
- 4-D: Ocean measurements in a volume of water over time

In all of these applications, grid-based interpolation efficiently extends the usefulness of the data to points where no measurement was taken. For example, if you have hourly price data for a stock, you can use interpolation to approximate the price every 15 minutes.

MATLAB Gridded Interpolation Functions

MATLAB provides several tools for grid-based interpolation:

Grid Creation Functions

The `meshgrid` and `ndgrid` functions create grids of various dimensionality. `meshgrid` can create 2-D or 3-D grids, while `ndgrid` can create grids with any number of dimensions. These functions return grids using different output formats. You can convert between these grid formats using the `pagetranspose` (*as of R2020b*) or `permute` functions to swap the first two dimensions of the grid.

Interpolation Functions

The `interp` family of functions includes `interp1`, `interp2`, `interp3`, and `interpn`. Each function is designed to interpolate data with a specific number of dimensions. `interp2` and `interp3` use grids in `meshgrid` format, while `interpn` uses grids in `ndgrid` format.

Interpolation Objects

`griddedInterpolant` objects support interpolation in any number of dimensions for data in `ndgrid` format. These objects also support multivalued interpolation (*as of R2021a*), where each grid point can have multiple values associated with it.

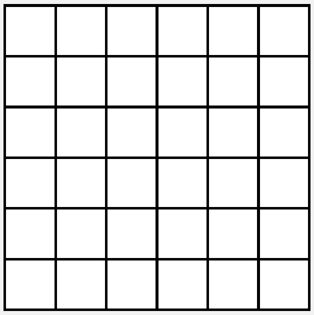
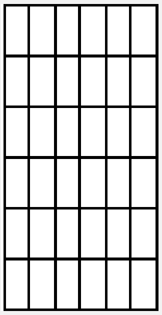
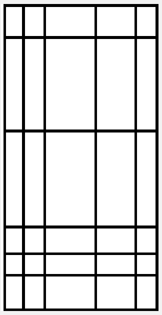
There are memory and performance benefits to using `griddedInterpolant` objects over the `interp` functions. `griddedInterpolant` offers substantial performance improvements for repeated queries of the interpolant object, whereas the `interp` functions perform a new calculation each time they are called. Also, `griddedInterpolant` stores the sample points in a memory-efficient format (as a compact grid on page 8-6) and is multithreaded to take advantage of multicore computer processors.

Grid Representations

MATLAB allows you to represent a grid in one of three representations: full grid, compact grid, or default grid. The default grid and compact grid are used primarily for convenience and improved efficiency, respectively.

Full Grid

A full grid is one in which all points are explicitly defined. The outputs of `ndgrid` and `meshgrid` define a full grid. You can create full grids that are uniform, in which points in each dimension have equal spacing, or nonuniform, in which the spacing varies in one or more of the dimensions. Uniform grids can have different spacing in each dimension, as long as the spacing is constant within each dimension.

Uniform	Uniform	Nonuniform
		

An example of a uniform full grid is:

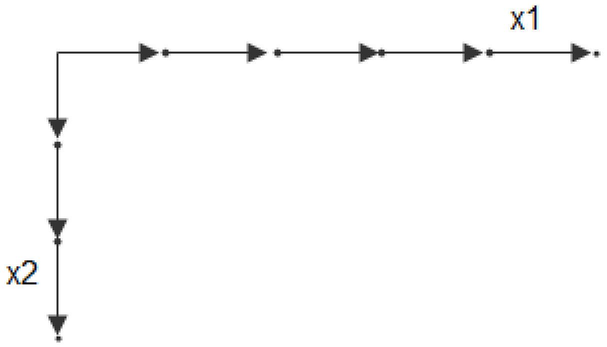
```
[X,Y] = meshgrid([1 2 3],[3 6 9 12])
```

```
X =

     1     2     3
     1     2     3
     1     2     3
     1     2     3


Y =

     3     3     3
     6     6     6
     9     9     9
    12    12    12
```

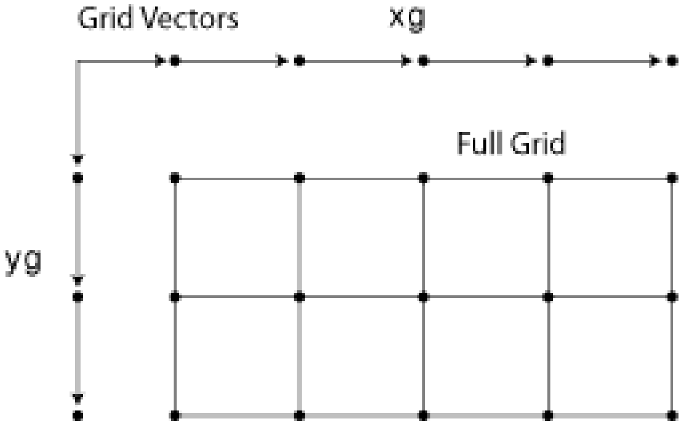
Compact Grid

Explicitly defining every point in a grid can consume a lot of memory when you are dealing with large grids. The compact grid representation is a way to dispense with the memory overhead of a full grid. The compact grid representation stores only grid vectors (one for each dimension) instead of the full grid. Together, the grid vectors implicitly define the grid. In fact, the inputs for `meshgrid` and `ndgrid` are grid vectors, and these functions replicate the grid vectors to form the full grid. The compact grid representation enables you to bypass grid creation and supply the grid vectors directly to the interpolation function.

For example, consider two vectors, `x1 = 1:3` and `x2 = 1:5`. You can think of these vectors as a set of coordinates in the `x1` direction and a set of coordinates in the `x2` direction, like so:



Each arrow points to a location. You can use these two vectors to define a set of grid points, where one set of coordinates is given by `x1` and the other set of coordinates is given by `x2`. When the grid vectors are replicated, they form two coordinate arrays that make up the full grid:



Your input grid vectors might be monotonic or nonmonotonic. Monotonic vectors contain values that either increase in that dimension or decrease in that dimension. Conversely, nonmonotonic vectors contain values that fluctuate. If the input grid vector is nonmonotonic, such as `[2 4 6 3 1]`, then `[X1,X2] = ndgrid([2 4 6 3 1])` outputs a nonmonotonic grid. Your grid vectors should be monotonic if you intend to pass the grid to other MATLAB functions. The `sort` function is useful to ensure monotonicity.

Default Grid

In some applications, only the values at the grid points are important and not the distances between grid points. For example, most MRI scans gather data that is uniformly spaced in all directions. In cases like this, you can allow the interpolation function to automatically generate a default grid representation to use with the data. To do this, leave out the grid inputs to the interpolation function. When you leave out the grid inputs, the function automatically considers the data to be on a unit-spaced grid. The function creates this unit-spaced grid while it executes, saving you the trouble of creating a grid yourself.

Example: Temperature Interpolation on 2-D Grid

Consider temperature data collected on a surface at regular 5 cm intervals, extending 20 cm in each direction. Use `meshgrid` to create the full grid.

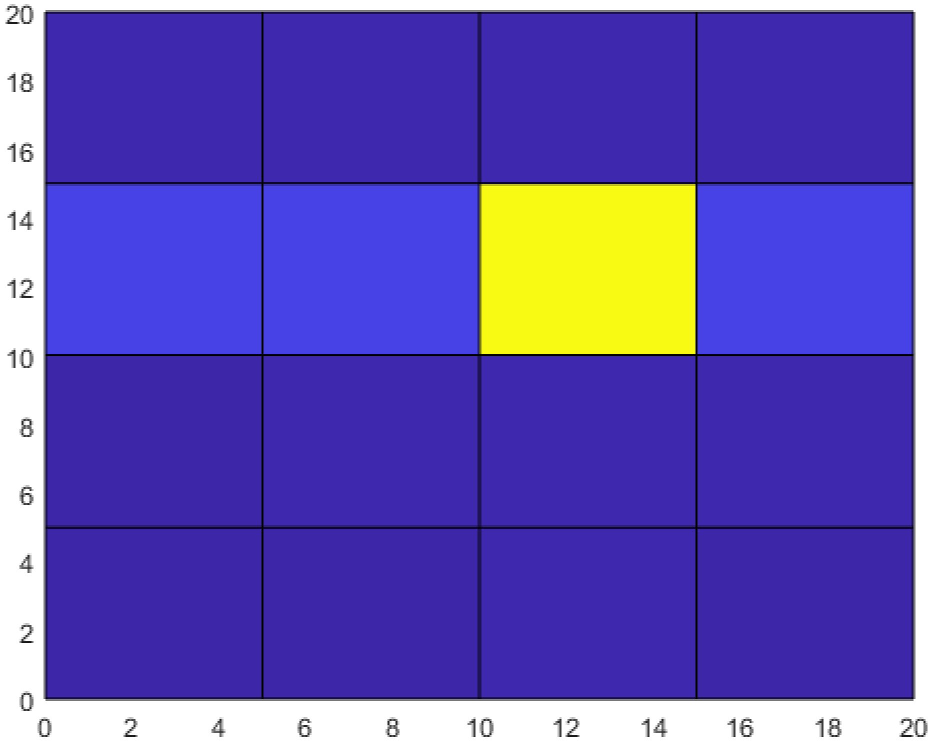
```
[X,Y] = meshgrid(0:5:20)
```

```
X =

     0     5    10    15    20
     0     5    10    15    20
     0     5    10    15    20
     0     5    10    15    20
     0     5    10    15    20


Y =

     0     0     0     0     0
     5     5     5     5     5
    10    10    10    10    10
    15    15    15    15    15
    20    20    20    20    20
```

The (*x*,*y*) coordinates of each grid point are represented as corresponding elements in the `X` and `Y` matrices. The first grid point is given by `[X(1) Y(1)]`, which is `[0 0]`, the next grid point is given by `[X(2) Y(2)]`, which is `[0 5]`, and so on.

Now, create a matrix to represent temperature measurements on the grid and then plot the data as a surface.

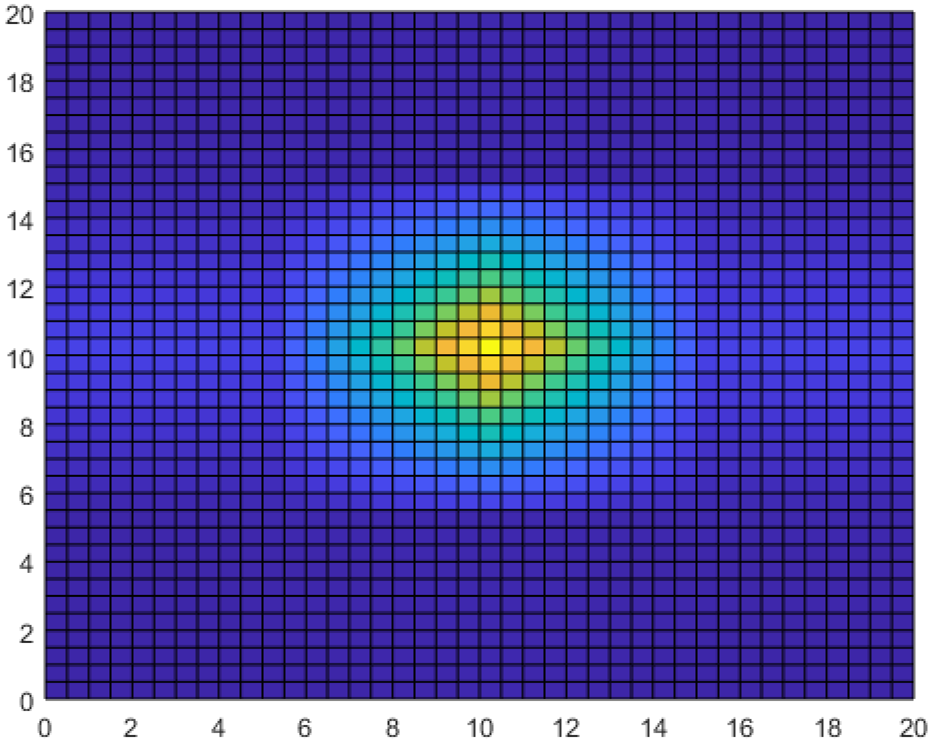
```
T = [1   1   10   1   1;
     1   10  10   10  10;
     100 100 1000 100 100;
     10  10  10   10  1;
     1   1   10   1   1];
surf(X,Y,T)
view(2)
```



Although the temperature at the center grid point is large, its location and influence on surrounding grid points is not apparent from the raw data.

To improve the resolution of the data by a factor of 10, use `interp2` to interpolate the temperature data onto a finer grid that uses 0.5 cm intervals. Use `meshgrid` again to create a finer grid represented by the matrices `Xq` and `Yq`. Then, use `interp2` with the original grid, the temperature data, and the new grid points, and plot the resulting data. By default, `interp2` uses linear interpolation in each dimension.

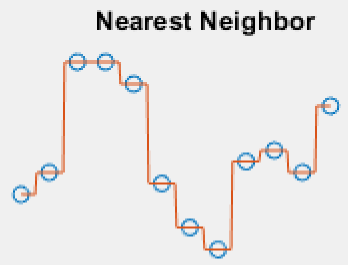
```
[Xq,Yq] = meshgrid(0:0.5:20);
Tq = interp2(X,Y,T,Xq,Yq);
surf(Xq,Yq,Tq)
view(2)
```

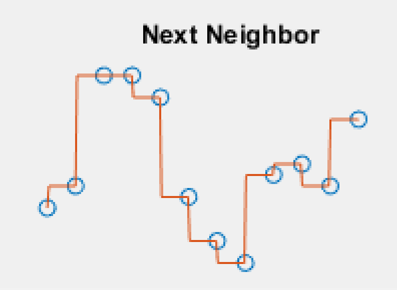
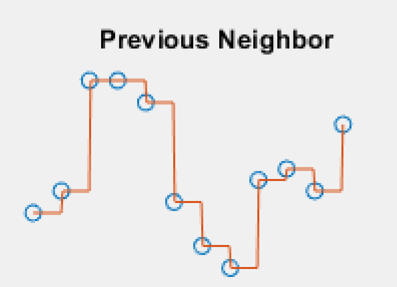
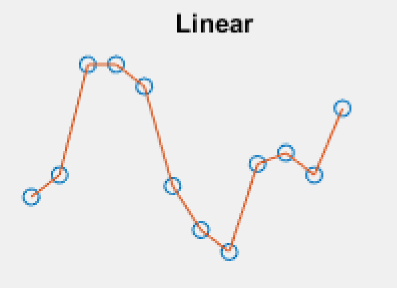
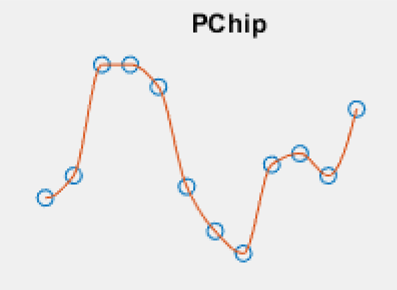


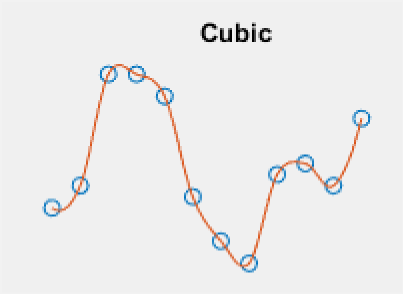
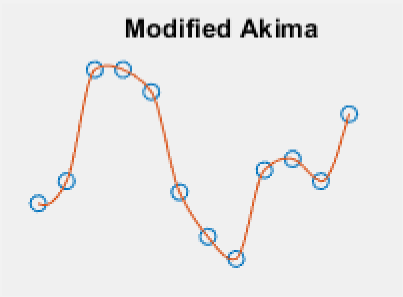
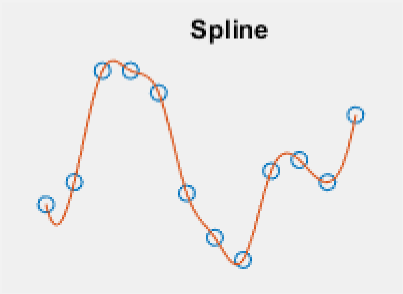
Interpolating the temperature data adds detail to the image and greatly improves the usefulness of the data within the area of measurements.

Gridded Interpolation Methods

The grid-based interpolation functions and objects in MATLAB offer several different methods for interpolation. When choosing an interpolation method, keep in mind that some require more memory or longer computation time than others. You may need to trade off these resources to achieve the desired smoothness in the result. The following table gives a preview of each interpolation method applied to the same 1-D data, and also provides an overview of the benefits, trade-offs, and requirements for each method.

Method	Description
	The interpolated value at a query point is the value at the nearest sample grid point. • Discontinuous • Modest memory requirements • Fastest computation time • Requires 2 grid points in each dimension

Method	Description
	The interpolated value at a query point is the value at the next sample grid point. • Discontinuous • Same memory requirements and computation time as nearest neighbor • Available for 1-D interpolation only • Requires at least 2 grid points
	The interpolated value at a query point is the value at the previous sample grid point. • Discontinuous • Same memory requirements and computation time as nearest neighbor • Available for 1-D interpolation only • Requires at least 2 grid points
	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension. • C0 continuous • Requires more memory and computation time than nearest neighbor • Requires at least 2 grid points in each dimension
	The interpolated value at a query point is based on a shape-preserving piece-wise cubic interpolation of the values at neighboring grid points. • C1 continuous • Requires more memory and computation time than linear • Available for 1-D interpolation only • Requires at least 4 grid points

Method	Description
	The interpolated value at a query point is based on cubic interpolation of the values at neighboring grid points in each respective dimension. • C1 continuous • Requires more memory and computation time than linear • Grid must have uniform spacing, though the spacing in each dimension does not have to be the same • Requires at least 4 points in each dimension
	The interpolated value at a query point is based on a piecewise function of polynomials with degree at most three evaluated using the values of neighboring grid points in each respective dimension. The Akima formula is modified to avoid overshoots. • C1 continuous • Similar memory requirements as spline • Requires more computation time than cubic, but typically less than spline • Requires at least 2 grid points in each dimension
	The interpolated value at a query point is based on a cubic interpolation of the values at neighboring grid points in each respective dimension. • C2 continuous • Requires more memory and computation time than cubic • Requires at least 4 points in each dimension

See Also

`griddedInterpolant` | `interp1` | `interp2` | `interp3` | `interpn`

Related Examples

- “Resample Image with Gridded Interpolation” on page 8-53
- “Interpolation of Multiple 1-D Value Sets” on page 8-13
- “Interpolation of 2-D Selections in 3-D Grids” on page 8-15

Interpolation of Multiple 1-D Value Sets

This example shows how to interpolate three 1-D data sets in a single pass using `griddedInterpolant`. This is a faster alternative to looping over your data sets.

Define the *x*-coordinates that are common to all value sets.

```
x = (1:5)';
```

Define the sets of sample points along the columns of matrix V.

```
V = [x, 2*x, 3*x]
```

```
V = 5×3

     1     2     3
     2     4     6
     3     6     9
     4     8    12
     5    10    15
```

Create the interpolant `F` by passing the sample points and sample values to `griddedInterpolant`. With this setup, `griddedInterpolant` interprets `V` as containing three different 1-D data sets defined at the same *x*-values.

```
F = griddedInterpolant(x,V);
```

Create a vector of query points with `0.5` spacing.

```
qx = 1:0.5:5;
```

Evaluate the interpolant at the *x*-coordinates for each value set.

```
Vq = F(qx)
```

```
Vq = 9×3

    1.0000    2.0000    3.0000
    1.5000    3.0000    4.5000
    2.0000    4.0000    6.0000
    2.5000    5.0000    7.5000
    3.0000    6.0000    9.0000
    3.5000    7.0000   10.5000
    4.0000    8.0000   12.0000
    4.5000    9.0000   13.5000
    5.0000   10.0000   15.0000
```

See Also

`griddedInterpolant`

Related Examples

- "Interpolating Gridded Data" on page 8-5

- “Resample Image with Gridded Interpolation” on page 8-53

Interpolation of 2-D Selections in 3-D Grids

This example shows how to reduce the dimensionality of the grid plane arrays in 3-D to solve a 2-D interpolation problem.

In some application areas, it might be necessary to interpolate a lower dimensional plane of a grid; for example, interpolating a plane of a 3-D grid. When you extract the grid plane from the 3-D grid, the resulting arrays might be in 3-D format. You can use the `squeeze` function to reduce the dimensionality of the grid plane arrays to solve the problem in 2-D.

Create a 3-D sample grid and corresponding values.

```
[X,Y,Z] = ndgrid(1:5);
V = X.^2 + Y.^2 +Z;
```

Select a 2-D sample from the grid. In this case, the third column of samples.

```
x = X(:,3,:);
z = Z(:,3,:);
v = V(:,3,:);
```

The 2-D plane occurs at `Y=3`, so the `Y` dimension has been fixed. `x`, `z`, and `v` are 5-by-1-by-5 arrays. You must reduce them to 2-D arrays before evaluating the interpolant.

Reduce `x`, `z`, and `v` down to 2-D arrays using the `squeeze` function.

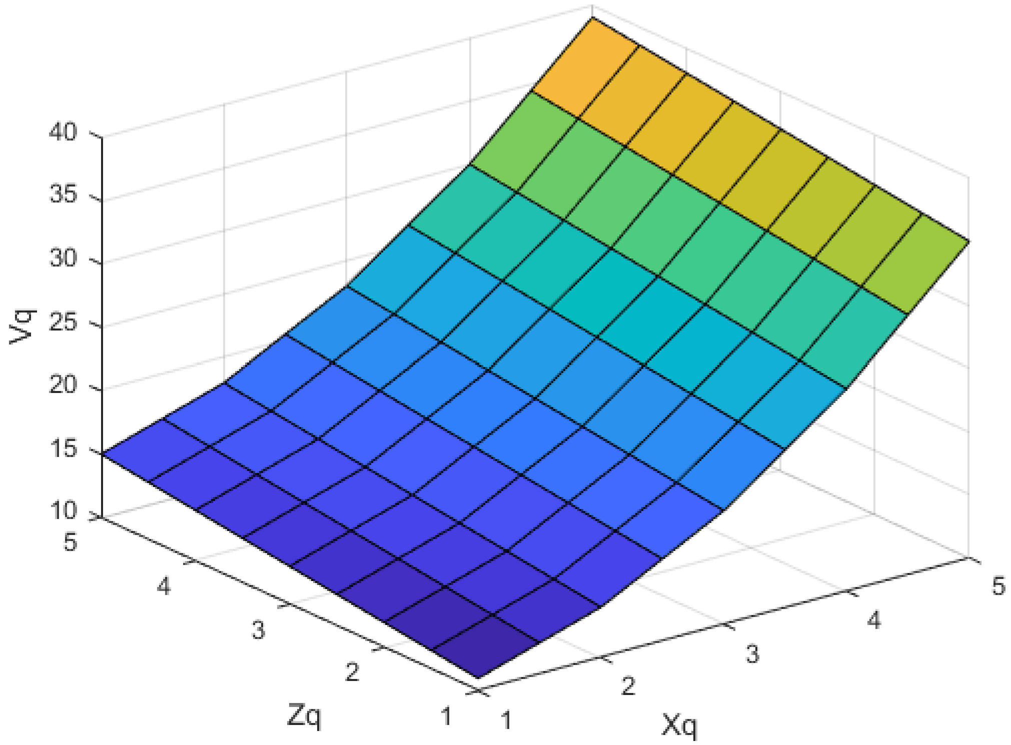
```
x = squeeze(x);
z = squeeze(z);
v = squeeze(v);
```

Interpolate the 2-D slice over a finer grid of query points.

```
[Xq,Zq] = ndgrid(1:0.5:5);
Vq = interpn(x,z,v,Xq,Zq);
```

Plot the results.

```
figure
surf(Xq,Zq,Vq);
xlabel('Xq');
ylabel('Zq');
zlabel('Vq');
```



See Also

`interpn` | `squeeze`

More About

- "Interpolating Gridded Data" on page 8-5
- "Interpolation of Multiple 1-D Value Sets" on page 8-13

Interpolating Scattered Data

In this section...

Scattered Data

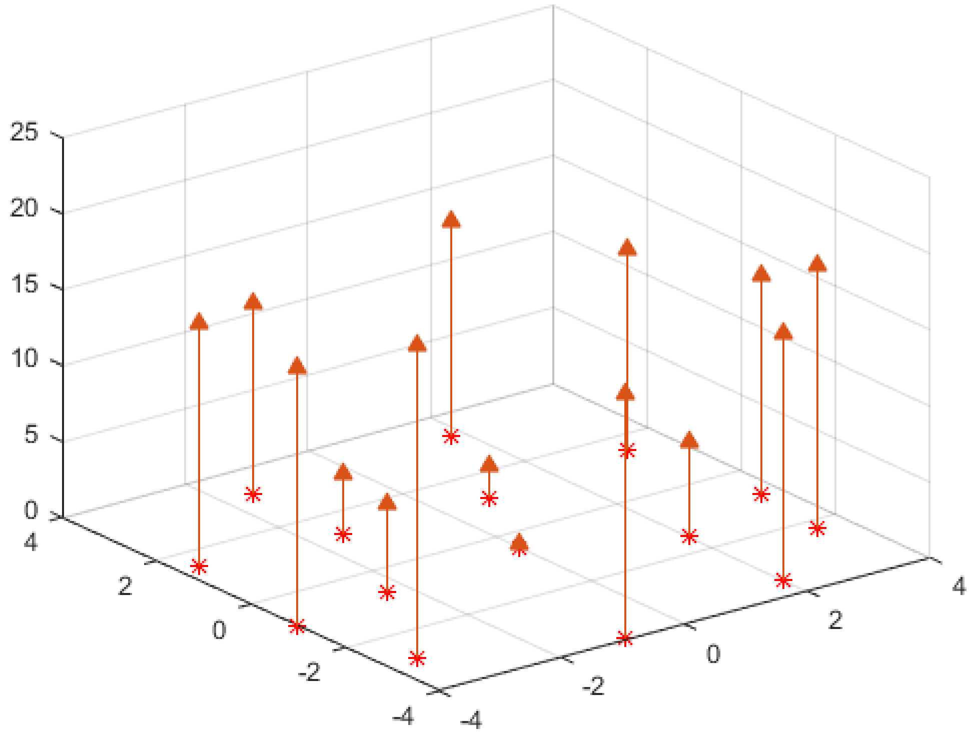
Scattered data consists of a set of points X and corresponding values V, where the points have no structure or order between their relative locations. There are various approaches to interpolating scattered data. One widely used approach uses a Delaunay triangulation of the points.

This example shows how to construct an interpolating surface by triangulating the points and lifting the vertices by a magnitude V into a dimension orthogonal to X.

There are variations on how you can apply this approach. In this example, the interpolation is broken down into separate steps; typically, the overall interpolation process is accomplished with one function call.

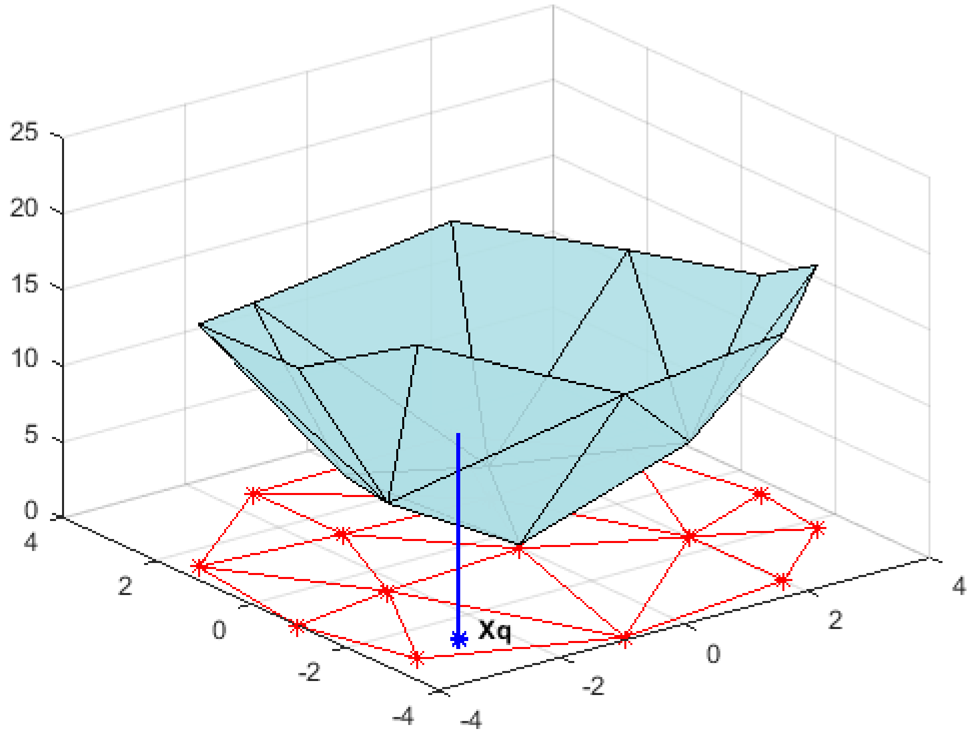
Create a scattered data set on the surface of a paraboloid.

```
X = [-1.5 3.2; 1.8 3.3; -3.7 1.5; -1.5 1.3; ...
      0.8 1.2; 3.3 1.5; -4.0 -1.0; -2.3 -0.7;
      0 -0.5; 2.0 -1.5; 3.7 -0.8; -3.5 -2.9; ...
      -0.9 -3.9; 2.0 -3.5; 3.5 -2.25];
V = X(:,1).^2 + X(:,2).^2;
hold on
plot3(X(:,1),X(:,2),zeros(15,1), '*r')
axis([-4, 4, -4, 4, 0, 25]);
grid
stem3(X(:,1),X(:,2),V,'^','fill')
hold off
view(322.5, 30);
```



Create a Delaunay triangulation, lift the vertices, and evaluate the interpolant at the query point Xq.

```
figure('Color', 'white')
t = delaunay(X(:,1),X(:,2));

hold on

trimesh(t,X(:,1),X(:,2), zeros(15,1), ...
    'EdgeColor','r', 'FaceColor','none')
defaultFaceColor  = [0.6875 0.8750 0.8984];
trisurf(t,X(:,1),X(:,2), V, 'FaceColor', ...
    defaultFaceColor, 'FaceAlpha',0.9);
plot3(X(:,1),X(:,2),zeros(15,1), '*r')
axis([-4, 4, -4, 4, 0, 25]);
grid
plot3(-2.6,-2.6,0,'*b','LineWidth', 1.6)
plot3([-2.6 -2.6]',[-2.6 -2.6]',[0 13.52]','-b','LineWidth',1.6)
hold off

view(322.5, 30);

text(-2.0, -2.6, 'Xq', 'FontWeight', 'bold', ...
'HorizontalAlignment','center', 'BackgroundColor', 'none');
```



This step generally involves traversing of the triangulation data structure to find the triangle that encloses the query point. Once you find the point, the subsequent steps to compute the value depend on the interpolation method. You could compute the nearest point in the neighborhood and use the value at that point (the nearest-neighbor interpolation method). You could also compute the weighted sum of values of the three vertices of the enclosing triangle (the linear interpolation method). These methods and their variants are covered in texts and references on scattered data interpolation.

Though the illustration highlights 2-D interpolation, you can apply this technique to higher dimensions. In more general terms, given a set of points `X` and corresponding values `V`, you can construct an interpolant of the form `V = F(X)`. You can evaluate the interpolant at a query point `Xq`, to give `Vq = F(Xq)`. This is a single-valued function; for any query point `Xq` within the convex hull of `X`, it will produce a unique value `Vq`. The sample data is assumed to respect this property in order to produce a satisfactory interpolation.

MATLAB provides two ways to perform triangulation-based scattered data interpolation:

- The functions `griddata` and `griddatan`
- The `scatteredInterpolant` class

The `griddata` function supports 2-D scattered data interpolation. The `griddatan` function supports scattered data interpolation in N-D; however, it is not practical in dimensions higher than 6-D for moderate to large point sets, due to the exponential growth in memory required by the underlying triangulation.

The scatteredInterpolant class supports scattered data interpolation in 2-D and 3-D space. Use of this class is encouraged as it is more efficient and readily adapts to a wider range of interpolation problems.

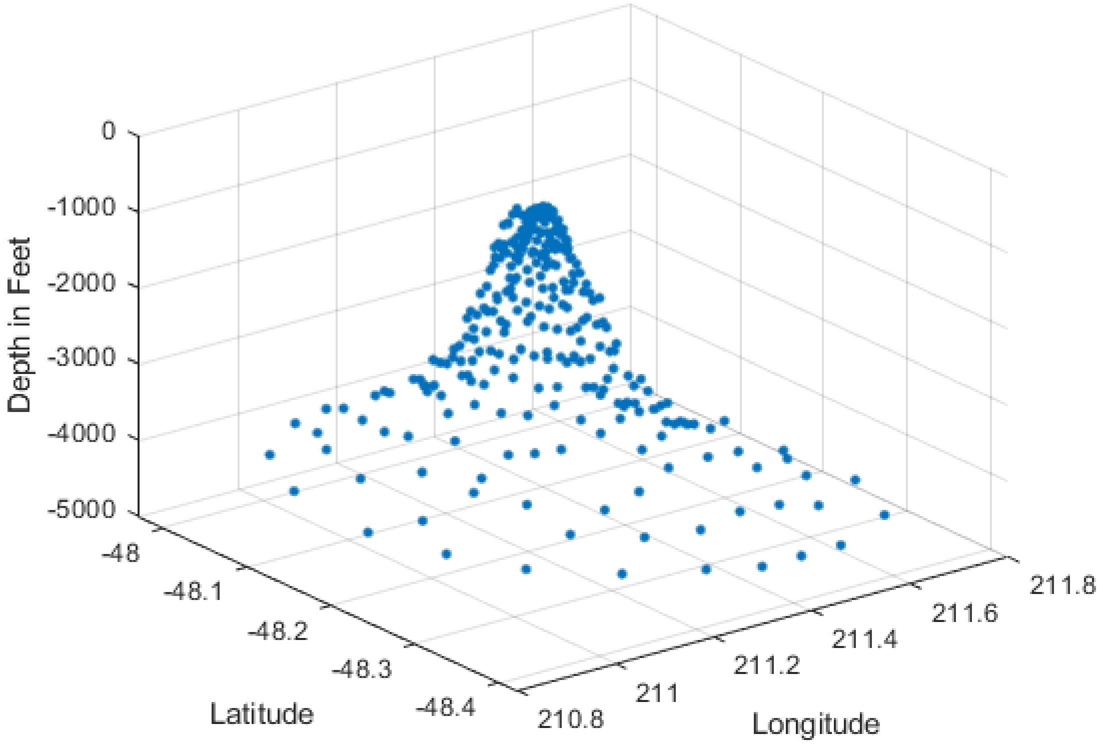
Interpolating Scattered Data Using griddata and griddatan

The griddata and griddatan functions take a set of sample points, X, corresponding values, V, and query points, Xq, and return the interpolated values, Vq. The calling syntax is similar for each function; the primary distinction is the 2-D / 3–D griddata function lets you define the points in terms of X, Y / X, Y, Z coordinates. These two functions interpolate scattered data at predefined grid-point locations; the intent is to produce gridded data, hence the name. Interpolation is more general in practice. You might want to query at arbitrary locations within the convex hull of the points.

This example shows how the griddata function interpolates scattered data at a set of grid points and uses this gridded data to create a contour plot.

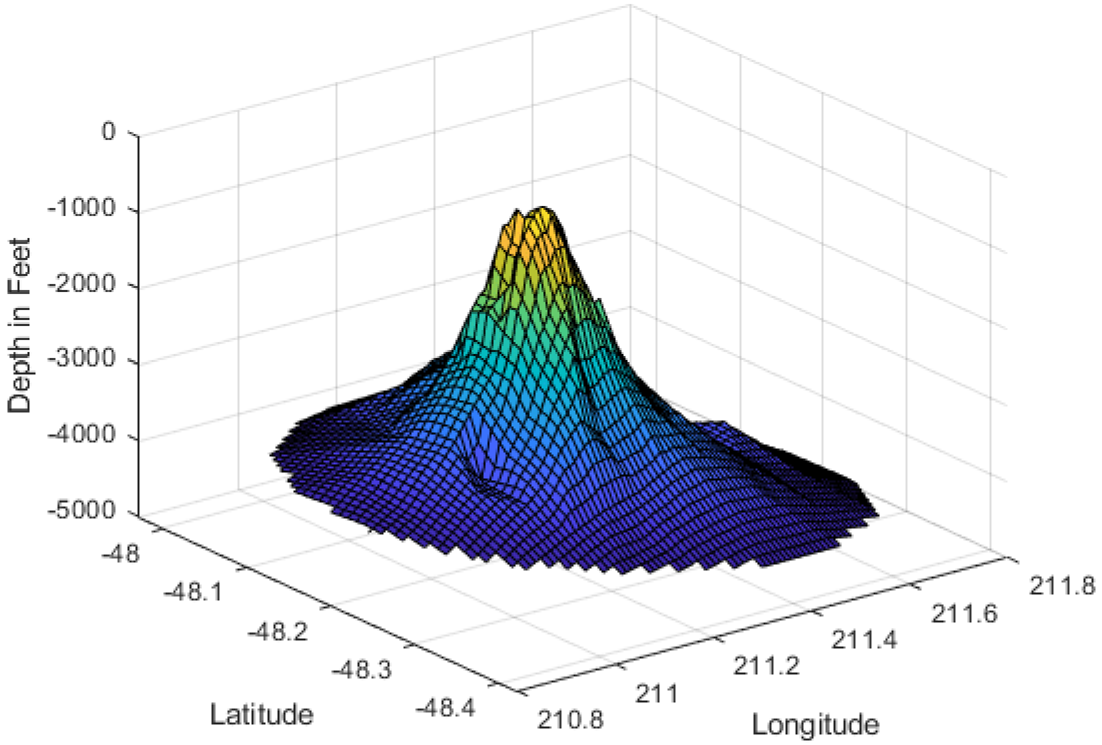
Plot the seamount data set (a *seamount* is an underwater mountain). The data set consists of a set of longitude (x) and latitude (y) locations, and corresponding seamount elevations (z) measured at those coordinates.

```
load seamount
plot3(x,y,z,'.','markersize',12)
xlabel('Longitude')
ylabel('Latitude')
zlabel('Depth in Feet')
grid on
```



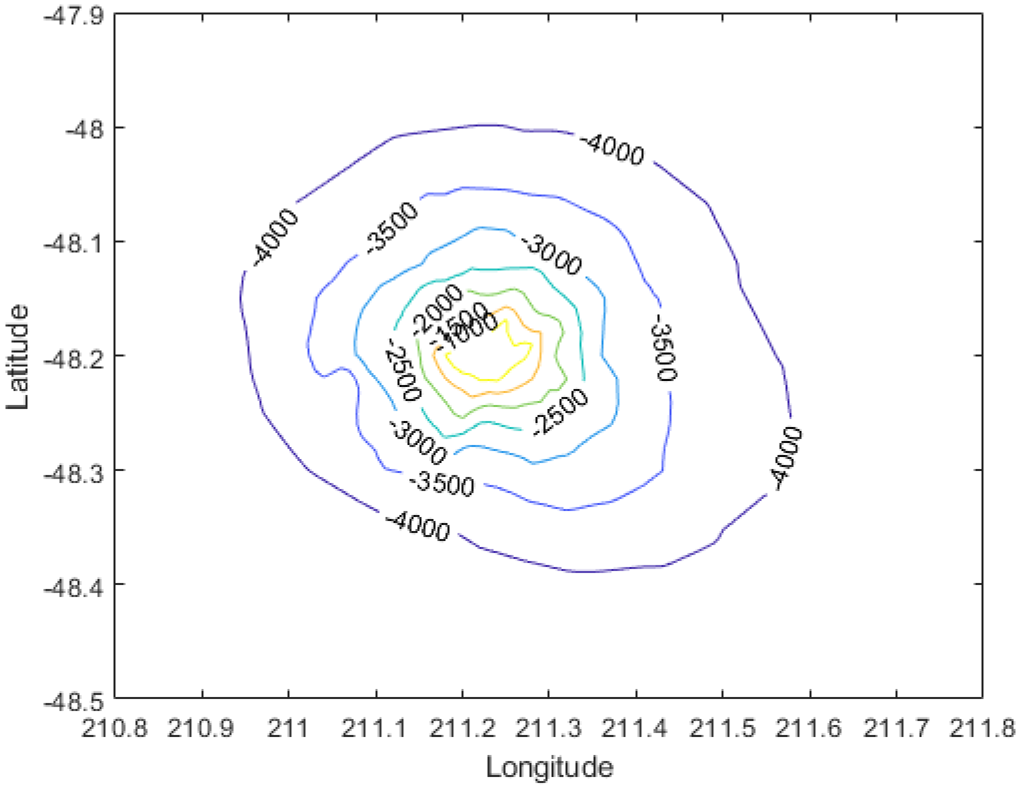
Use meshgrid to create a set of 2-D grid points in the longitude-latitude plane and then use griddata to interpolate the corresponding depth at those points.

```
figure
[xi,yi] = meshgrid(210.8:0.01:211.8, -48.5:0.01:-47.9);
zi = griddata(x,y,z,xi,yi);
surf(xi,yi,zi);
xlabel('Longitude')
ylabel('Latitude')
zlabel('Depth in Feet')
```



Now that the data is in a gridded format, compute and plot the contours.

```
figure
[c,h] = contour(xi,yi,zi);
clabel(c,h);
xlabel('Longitude')
ylabel('Latitude')
```



You can also use `griddata` to interpolate at arbitrary locations within the convex hull of the dataset. For example, the depth at coordinates (211.3, -48.2) is given by:

```
zi = griddata(x,y,z, 211.3, -48.2);
```

The underlying triangulation is computed each time the `griddata` function is called. This can impact performance if the same data set is interpolated repeatedly with different query points. The `scatteredInterpolant` class described in “Interpolating Scattered Data Using the scatteredInterpolant Class” on page 8-23 is more efficient in this respect.

MATLAB software also provides `griddatan` to support interpolation in higher dimensions. The calling syntax is similar to `griddata`.

scatteredInterpolant Class

The `griddata` function is useful when you need to interpolate to find the values at a set of predefined grid-point locations. In practice, interpolation problems are often more general, and the `scatteredInterpolant` class provides greater flexibility. The class has the following advantages:

- It produces an interpolating function that can be queried efficiently. That is, the underlying triangulation is created once and reused for subsequent queries.
- The interpolation method can be changed independently of the triangulation.
- The values at the data points can be changed independently of the triangulation.

- Data points can be incrementally added to the existing interpolant without triggering a complete recomputation. Data points can also be removed and moved efficiently, provided the number of points edited is small relative to the total number of sample points.
- It provides extrapolation functionality for approximating values at points that fall outside the convex hull. See “Extrapolating Scattered Data” on page 8-44 for more information.

`scatteredInterpolant` provides the following interpolation methods:

- `'nearest'` — Nearest-neighbor interpolation, where the interpolating surface is discontinuous.
- `'linear'` — Linear interpolation (default), where the interpolating surface is C0 continuous.
- `'natural'` — Natural-neighbor interpolation, where the interpolating surface is C1 continuous except at the sample points.

The `scatteredInterpolant` class supports scattered data interpolation in 2-D and 3-D space. You can create the interpolant by calling `scatteredInterpolant` and passing the point locations and corresponding values, and optionally the interpolation and extrapolation methods. See the `scatteredInterpolant` reference page for more information about the syntaxes you can use to create and evaluate a `scatteredInterpolant`.

Interpolating Scattered Data Using the scatteredInterpolant Class

This example shows how to use `scatteredInterpolant` to interpolate a scattered sampling of the `peaks` function.

Create the scattered data set.

```
rng default;
X = -3 + 6.*rand([250 2]);
V = peaks(X(:,1),X(:,2));
```

Create the interpolant.

```
F = scatteredInterpolant(X,V)
```

```
F = 
  scatteredInterpolant with properties:

                 Points: [250x2 double]
                 Values: [250x1 double]
                 Method: 'linear'
    ExtrapolationMethod: 'linear'
```

The `Points` property represents the coordinates of the data points, and the `Values` property represents the associated values. The `Method` property represents the interpolation method that performs the interpolation. The `ExtrapolationMethod` property represents the extrapolation method used when query points fall outside the convex hull.

You can access the properties of `F` in the same way you access the fields of a `struct`. For example, use `F.Points` to examine the coordinates of the data points.

Evaluate the interpolant.

`scatteredInterpolant` provides subscripted evaluation of the interpolant. It is evaluated the same way as a function. You can evaluate the interpolant as follows. In this case, the value at the query location is given by `Vq`. You can evaluate at a single query point:

```
Vq = F([1.5 1.25])
```

```
Vq = 1.4838
```

You can also pass individual coordinates:

```
Vq = F(1.5, 1.25)
```

```
Vq = 1.4838
```

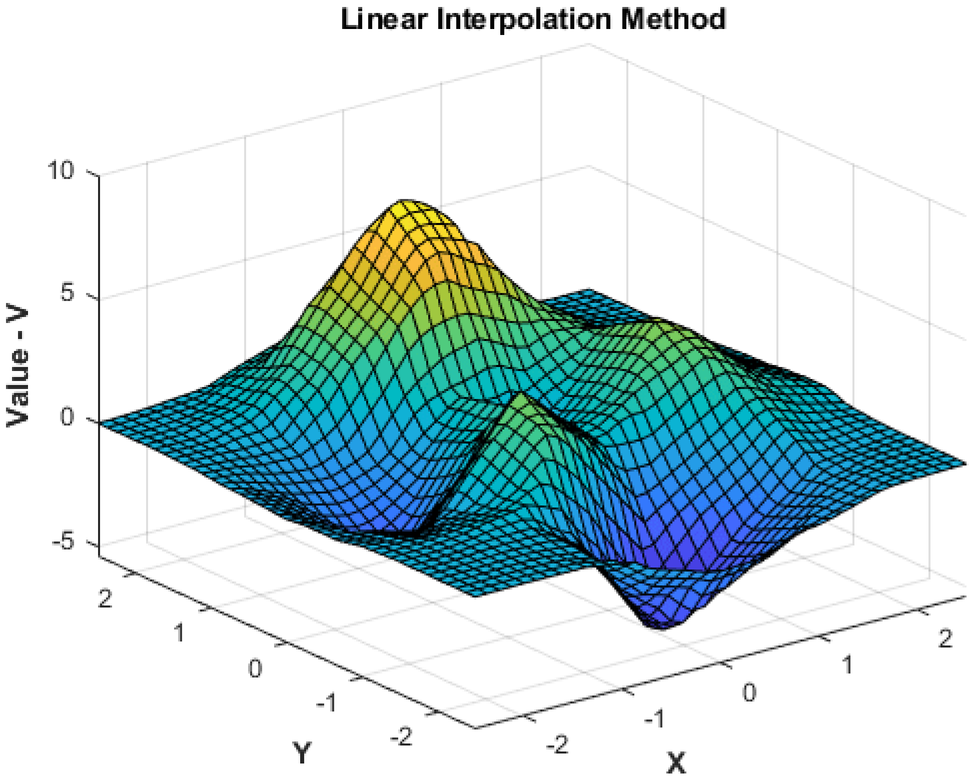
You can evaluate at a vector of point locations:

```
Xq = [0.5 0.25; 0.75 0.35; 1.25 0.85];
Vq = F(Xq)
```

```
Vq = 3×1

    0.4057
    1.2199
    2.1639
```

You can evaluate `F` at grid point locations and plot the result.

```
[Xq,Yq] = meshgrid(-2.5:0.125:2.5);
Vq = F(Xq,Yq);
surf(Xq,Yq,Vq);
xlabel('X','fontweight','b'), ylabel('Y','fontweight','b');
zlabel('Value - V','fontweight','b');
title('Linear Interpolation Method','fontweight','b');
```



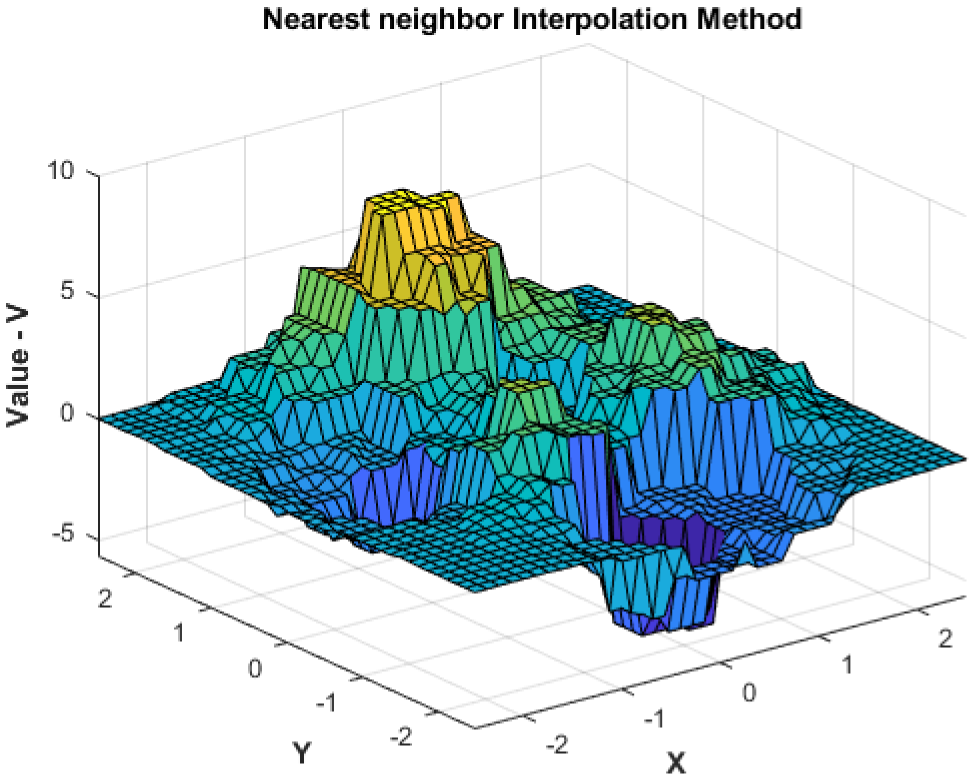
Change the interpolation method.

You can change the interpolation method on the fly. Set the method to `'nearest'`.

```
F.Method = 'nearest';
```

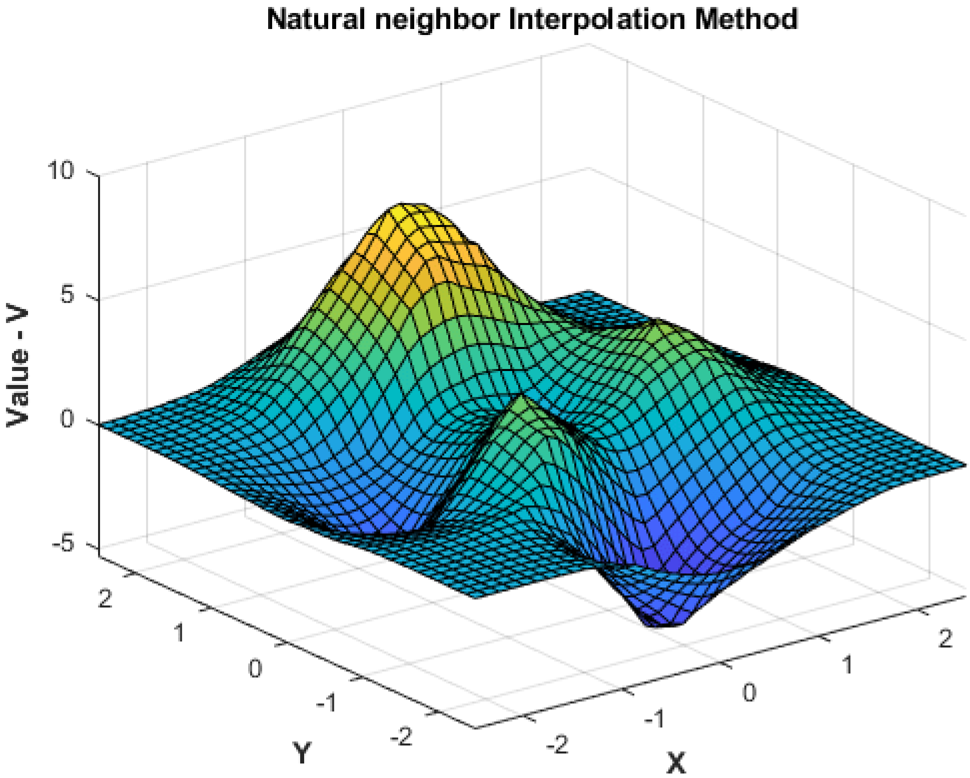
Reevaluate and plot the interpolant as before.

```
Vq = F(Xq,Yq);
figure
surf(Xq,Yq,Vq);
xlabel('X','fontweight','b'),ylabel('Y','fontweight','b')
zlabel('Value - V','fontweight','b')
title('Nearest neighbor Interpolation Method','fontweight','b');
```



Change the interpolation method to natural neighbor, reevaluate, and plot the results.

```
F.Method = 'natural';
Vq = F(Xq,Yq);
figure
surf(Xq,Yq,Vq);
xlabel('X','fontweight','b'),ylabel('Y','fontweight','b')
zlabel('Value - V','fontweight','b')
title('Natural neighbor Interpolation Method','fontweight','b');
```



Replace the values at the sample data locations.

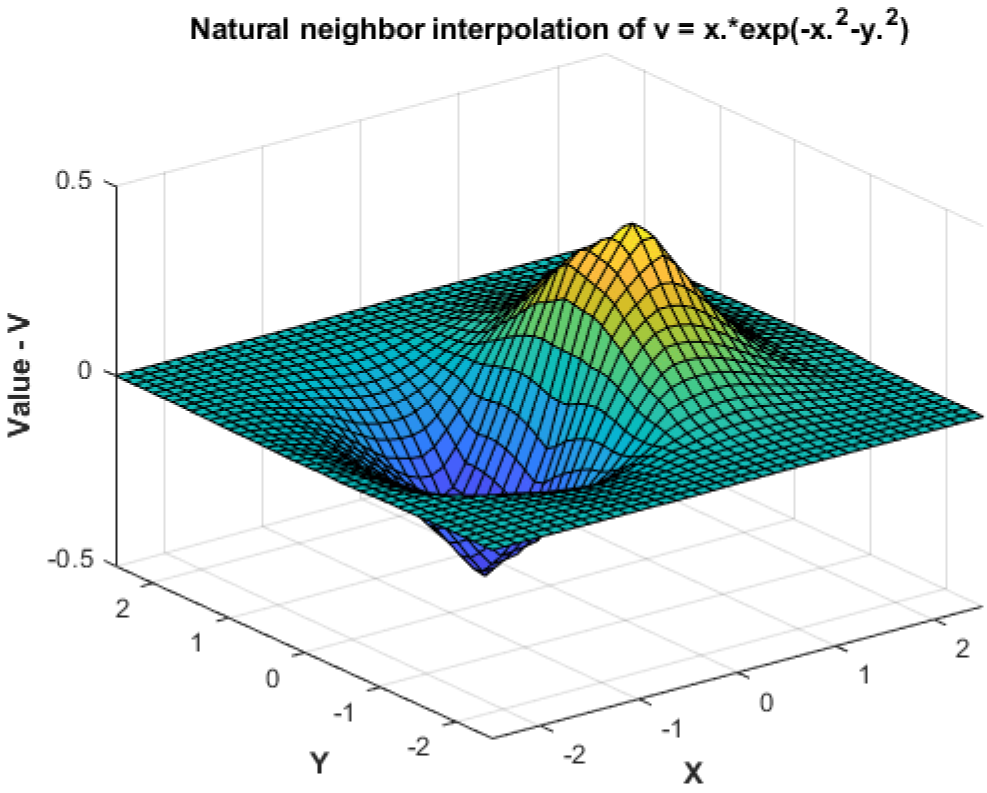
You can change the values V at the sample data locations, X, on the fly. This is useful in practice as some interpolation problems may have multiple sets of values at the same locations. For example, suppose you want to interpolate a 3-D velocity field that is defined by locations (x, y, z) and corresponding componentized velocity vectors (Vx, Vy, Vz). You can interpolate each of the velocity components by assigning them to the values property (V) in turn. This has important performance benefits, because it allows you to reuse the same interpolant without incurring the overhead of computing a new one each time.

The following steps show how to change the values in our example. You will compute the values using the expression, $v = xe^{-x^2 - y^2}$.

```
V = X(:,1).*exp(-X(:,1).^2-X(:,2).^2);
F.Values = V;
```

Evaluate the interpolant and plot the result.

```
Vq = F(Xq,Yq);
figure
surf(Xq,Yq,Vq);
xlabel('X','fontweight','b'), ylabel('Y','fontweight','b')
zlabel('Value - V','fontweight','b')
title('Natural neighbor interpolation of v = x.*exp(-x.^2-y.^2)')
```



Add additional point locations and values to the existing interpolant.

This performs an efficient update as opposed to a complete recomputation using the augmented data set.

When adding sample data, it is important to add both the point locations and the corresponding values.

Continuing the example, create new sample points as follows:

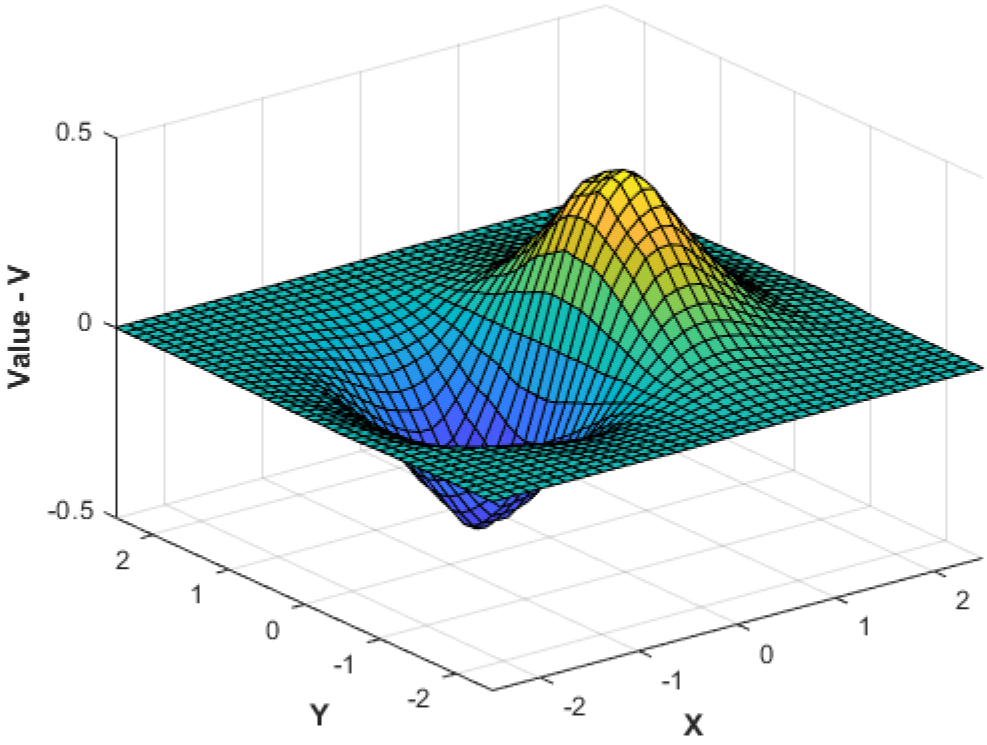
```
X = -1.5 + 3.*rand(100,2);
V = X(:,1).*exp(-X(:,1).^2-X(:,2).^2);
```

Add the new points and corresponding values to the triangulation.

```
F.Points(end+(1:100),:) = X;
F.Values(end+(1:100)) = V;
```

Evaluate the refined interpolant and plot the result.

```
Vq = F(Xq,Yq);
figure
surf(Xq,Yq,Vq);
xlabel('X','fontweight','b'), ylabel('Y','fontweight','b');
zlabel('Value - V','fontweight','b');
```



Remove data from the interpolant.

You can incrementally remove sample data points from the interpolant. You also can remove data points and corresponding values from the interpolant. This is useful for removing spurious outliers.

When removing sample data, it is important to remove both the point location and the corresponding value.

Remove the 25th point.

```
F.Points(25,:) = [];
F.Values(25) = [];
```

Remove points 5 to 15.

```
F.Points(5:15,:) = [];
F.Values(5:15) = [];
```

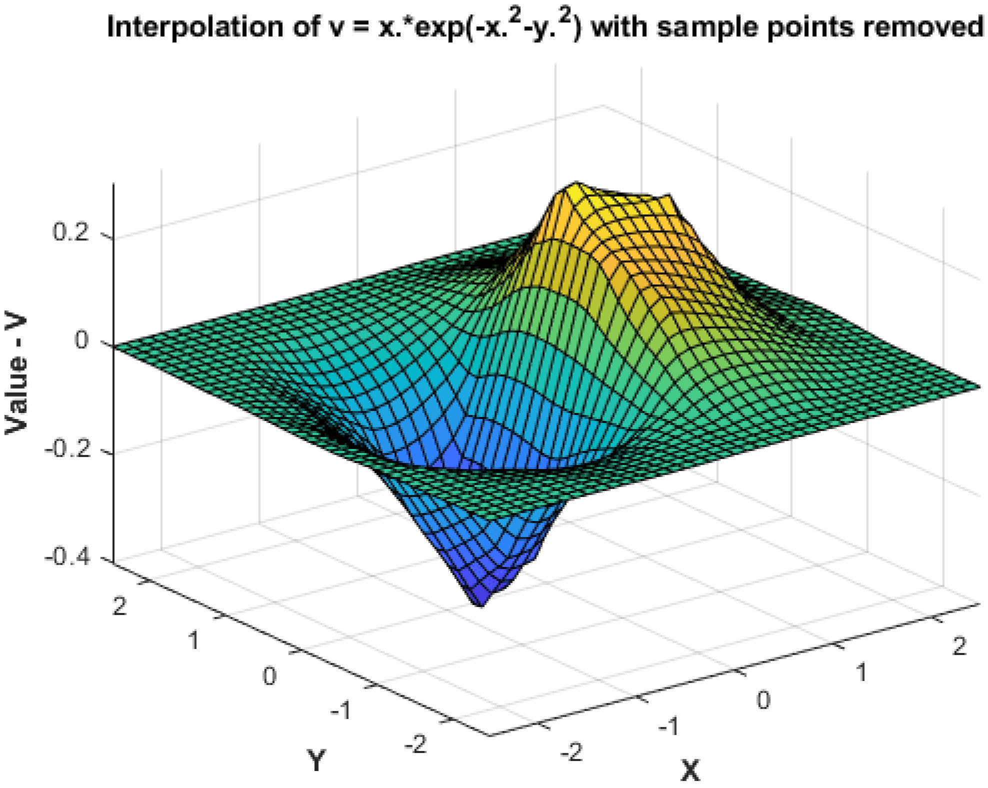
Retain 150 points and remove the rest.

```
F.Points(150:end,:) = [];
F.Values(150:end) = [];
```

This creates a coarser surface when you evaluate and plot:

```
Vq = F(Xq,Yq);
figure
surf(Xq,Yq,Vq);
```

```
xlabel('X','fontweight','b'), ylabel('Y','fontweight','b');
zlabel('Value - V','fontweight','b');
title('Interpolation of v = x.*exp(-x.^2-y.^2) with sample points removed')
```



Interpolation of Complex Scattered Data

This example shows how to interpolate scattered data when the value at each sample location is complex.

Create the sample data.

```
rng('default')
X = -3 + 6*rand([250 2]);
V = complex(X(:,1).*X(:,2), X(:,1).^2 + X(:,2).^2);
```

Create the interpolant.

```
F = scatteredInterpolant(X,V);
```

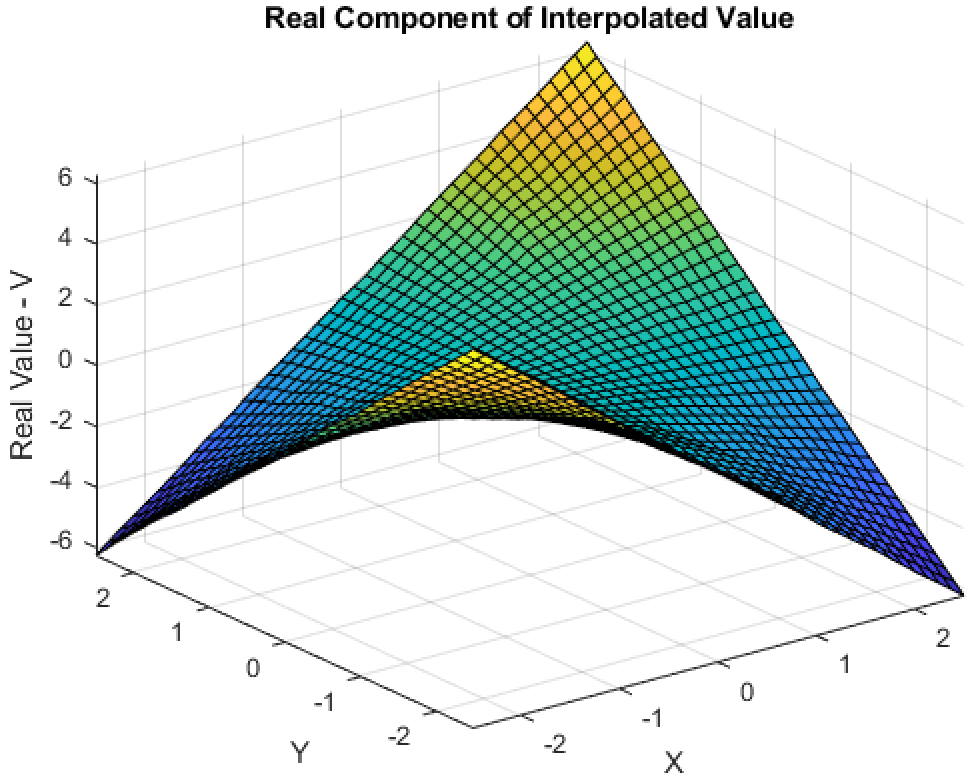
Create a grid of query points and evaluate the interpolant at the grid points.

```
[Xq,Yq] = meshgrid(-2.5:0.125:2.5);
Vq = F(Xq,Yq);
```

Plot the real component of `Vq`.

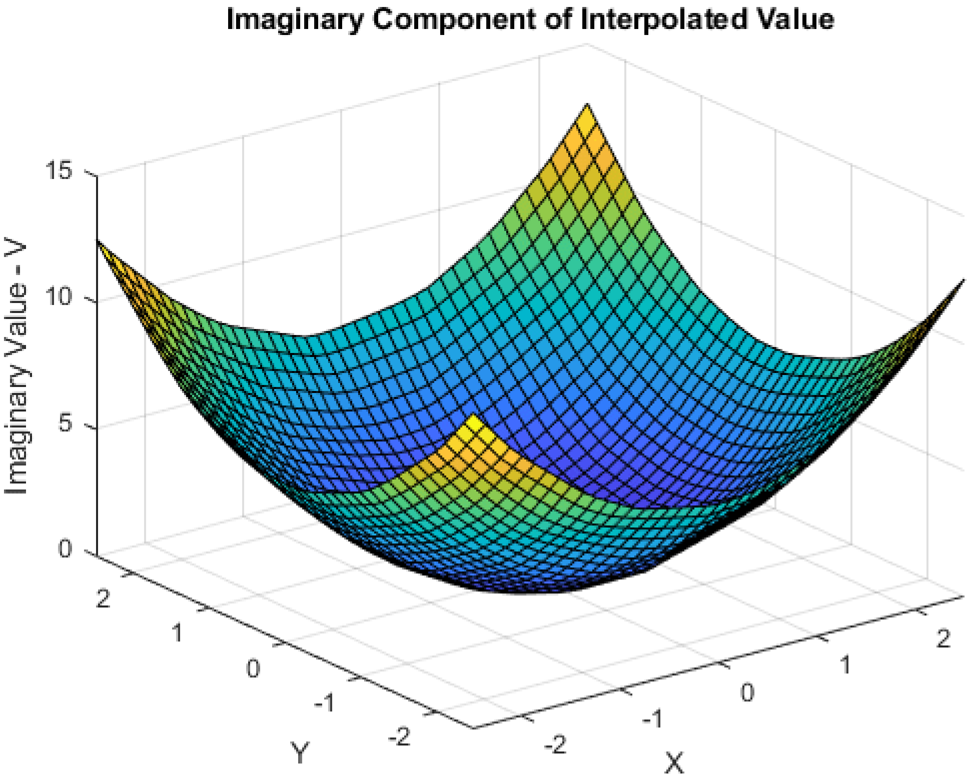
```
VqReal = real(Vq);
figure
```

```
surf(Xq,Yq,VqReal);
xlabel('X');
ylabel('Y');
zlabel('Real Value - V');
title('Real Component of Interpolated Value');
```



Plot the imaginary component of Vq.

```
VqImag = imag(Vq);
figure
surf(Xq,Yq,VqImag);
xlabel('X');
ylabel('Y');
zlabel('Imaginary Value - V');
title('Imaginary Component of Interpolated Value');
```



Addressing Problems in Scattered Data Interpolation

Many of the illustrative examples in the previous sections dealt with the interpolation of point sets that were sampled on smooth surfaces. In addition, the points were relatively uniformly spaced. For example, clusters of points were not separated by relatively large distances. In addition, the interpolant was evaluated well within the convex hull of the point locations.

When dealing with real-world interpolation problems the data may be more challenging. It may come from measuring equipment that is likely to produce inaccurate readings or outliers. The underlying data may not vary smoothly, the values may jump abruptly from point to point. This section provides you with some guidelines to identify and address problems with scattered data interpolation.

Input Data Containing NaNs

You should preprocess sample data that contains NaN values to remove the NaN values as this data cannot contribute to the interpolation. If a NaN is removed, the corresponding data values/ coordinates should also be removed to ensure consistency. If NaN values are present in the sample data, the constructor will error when called.

The following example illustrates how to remove NaNs.

Create some data and replace some entries with NaN:

```
x = rand(25,1)*4-2;
y = rand(25,1)*4-2;
```

```
V = x.^2 + y.^2;

x(5) = NaN; x(10) = NaN; y(12) = NaN; V(14) = NaN;
```

This code errors:

```
F = scatteredInterpolant(x,y,V);
```

Instead, find the sample point indices of the NaNs and then construct the interpolant:

```
nan_flags = isnan(x) | isnan(y) | isnan(V);

x(nan_flags) = [];
y(nan_flags) = [];
V(nan_flags) = [];

F = scatteredInterpolant(x,y,V);
```

The following example is similar if the point locations are in matrix form. First, create data and replace some entries with NaN values.

```
X = rand(25,2)*4-2;
V = X(:,1).^2 + X(:,2).^2;

X(5,1) = NaN; X(10,1) = NaN; X(12,2) = NaN; V(14) = NaN;
```

This code errors:

```
F = scatteredInterpolant(X,V);
```

Find the sample point indices of the NaN and then construct the interpolant:

```
nan_flags = isnan(X(:,1)) | isnan(X(:,2)) | isnan(V);

X(nan_flags,:) = [];
V(nan_flags) = [];

F = scatteredInterpolant(X,V);
```

Interpolant Outputs NaN Values

griddata and griddatan return NaN values when you query points outside the convex hull using the 'linear' or 'natural' methods. However, you can expect numeric results if you query the same points using the 'nearest' method. This is because the nearest neighbor to a query point exists both inside and outside the convex hull.

If you want to compute approximate values outside the convex hull, you should use scatteredInterpolant. See “Extrapolating Scattered Data” on page 8-44 for more information.

Handling Duplicate Point Locations

Input data is rarely “perfect” and your application could have to handle duplicate data point locations. Two or more data points at the same location in your data set can have different corresponding values. In this scenario, scatteredInterpolant merges the points and computes the average of the corresponding values. This example shows how scatteredInterpolant performs an interpolation on a data set with duplicate points.

1. Create some sample data that lies on a planar surface:

   ```
   x = rand(100,1)*6-3;
   y = rand(100,1)*6-3;

   V = x + y;
   ```
2. Introduce a duplicate point location by assigning the coordinates of point 50 to point 100:

   ```
   x(50) = x(100);
   y(50) = y(100);
   ```
3. Create the interpolant. Notice that F contains 99 unique data points:

   ```
   F = scatteredInterpolant(x,y,V)
   ```
4. Check the value associated with the 50th point:

   ```
   F.Values(50)
   ```

This value is the average of the original 50th and 100th value, as these two data points have the same location:

```
(V(50)+V(100))/2
```

In this scenario the interpolant resolves the ambiguity in a reasonable manner. However in some instances, data points can be close rather than coincident, and the values at those locations can be different.

In some interpolation problems, multiple sets of sample values might correspond to the same locations. For example, a set of values might be recorded at the same locations at different periods in time. For efficiency, you can interpolate one set of readings and then replace the values to interpolate the next set.

Always use consistent data management when replacing values in the presence of duplicate point locations. Suppose you have two sets of values associated with the 100 data point locations and you would like to interpolate each set in turn by replacing the values.

1. Consider two sets of values:

   ```
   V1 = x + 4*y;
   V2 = 3*x + 5*y
   ```
2. Create the interpolant. scatteredInterpolant merges the duplicate locations and the interpolant contains 99 unique sample points:

   ```
   F = scatteredInterpolant(x,y,V1)
   ```

   Replacing the values directly via F.Values = V2 means assigning 100 values to 99 samples. The context of the previous merge operation is lost; the number of sample locations will not match the number of sample values. The interpolant will require the inconsistency to be resolved to support queries.

In this more complex scenario, it is necessary to remove the duplicates prior to creating and editing the interpolant. Use the unique function to find the indices of the unique points. unique can also output arguments that identify the indices of the duplicate points.

```
[~, I, ~] = unique([x y],'first','rows');
I = sort(I);
x = x(I);
y = y(I);
```

```
V1 = V1(I);
V2 = V2(I);
F = scatteredInterpolant(x,y,V1)
```

Now you can use `F` to interpolate the first data set. Then you can replace the values to interpolate the second data set.

```
F.Values = V2;
```

Achieving Efficiency When Editing a scatteredInterpolant

`scatteredInterpolant` allows you to edit the properties representing the sample values (`F.Values`) and the interpolation method (`F.Method`). Since these properties are independent of the underlying triangulation, the edits can be performed efficiently. However, like working with a large array, you should take care not to accidentally create unnecessary copies when editing the data. Copies are made when more than one variable references an array and that array is then edited.

A copy is not made in the following:

```
A1 = magic(4)
A1(4,4) = 11
```

However, a copy is made in this scenario because the array is referenced by another variable. The arrays `A1` and `A2` can no longer share the same data once the edit is made:

```
A1 = magic(4)
A2 = A1
A1(4,4) = 32
```

Similarly, if you pass the array to a function and edit the array within that function, a deep copy may be made depending on how the data is managed. `scatteredInterpolant` contains data and it behaves like an array—in MATLAB language, it is called a value object. The MATLAB language is designed to give optimum performance when your application is structured into functions that reside in files. Prototyping at the command line may not yield the same level of performance.

The following example demonstrates this behavior, but it should be noted that performance gains in this example do not generalize to other functions in MATLAB.

This code does not produce optimal performance:

```
x = rand(1000000,1)*4-2;
y = rand(1000000,1)*4-2;
z = x.*exp(-x.^2-y.^2);
tic; F = scatteredInterpolant(x,y,z); toc
tic; F.Values = 2*z; toc
```

You can place the code in a function file to execute it more efficiently.

When MATLAB executes a program that is composed of functions that reside in files, it has a complete picture of the execution of the code; this allows MATLAB to optimize for performance. When you type the code at the command line, MATLAB cannot anticipate what you are going to type next, so it cannot perform the same level of optimization. Developing applications through the creation of reusable functions is general and recommended practice, and MATLAB will optimize the performance in this setting.

Interpolation Results Poor Near the Convex Hull

The Delaunay triangulation is well suited to scattered data interpolation problems because it has favorable geometric properties that produce good results. These properties are:

- The rejection of sliver-shaped triangles/tetrahedra in favor of more equilateral-shaped ones.
- The empty circumcircle property that implicitly defines a nearest-neighbor relation between the points.

The empty circumcircle property ensures the interpolated values are influenced by sample points in the neighborhood of the query location. Despite these qualities, in some situations the distribution of the data points may lead to poor results and this typically happens near the convex hull of the sample data set. When the interpolation produces unexpected results, a plot of the sample data and underlying triangulation can often provide insight into the problem.

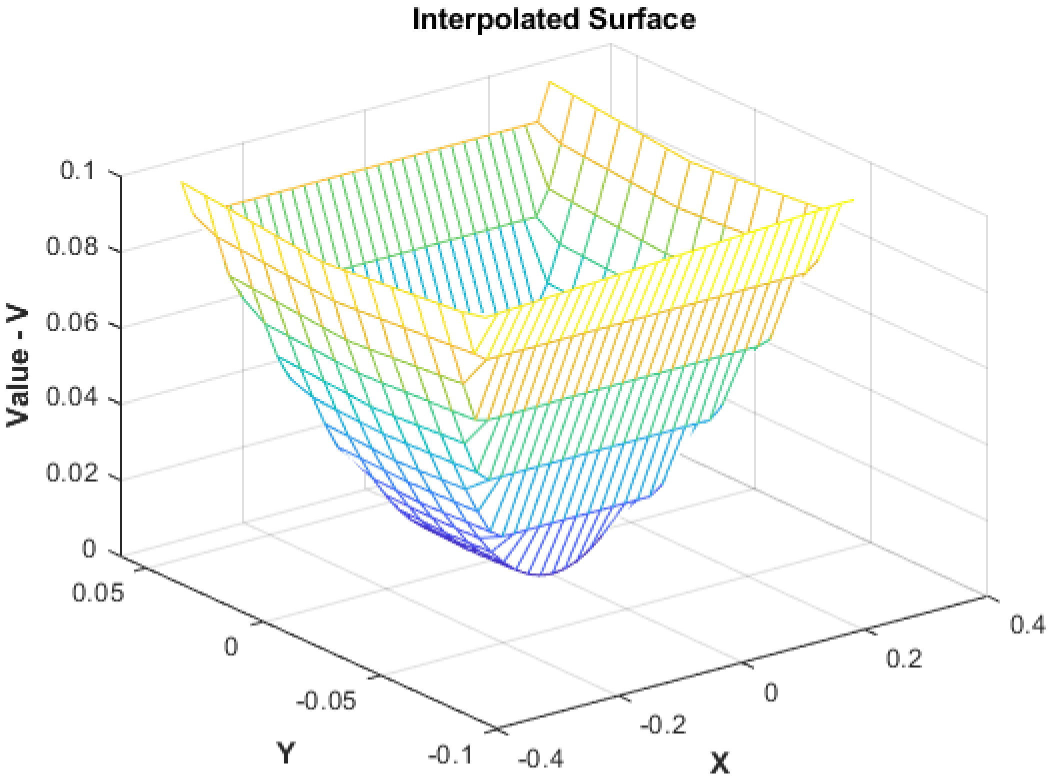
This example shows an interpolated surface that deteriorates near the boundary.

Create a sample data set that will exhibit problems near the boundary.

```
t = 0.4*pi:0.02:0.6*pi;
x1 = cos(t)';
y1 = sin(t)'-1.02;
x2 = x1;
y2 = y1*(-1);
x3 = linspace(-0.3,0.3,16)';
y3 = zeros(16,1);
x = [x1;x2;x3];
y = [y1;y2;y3];
```

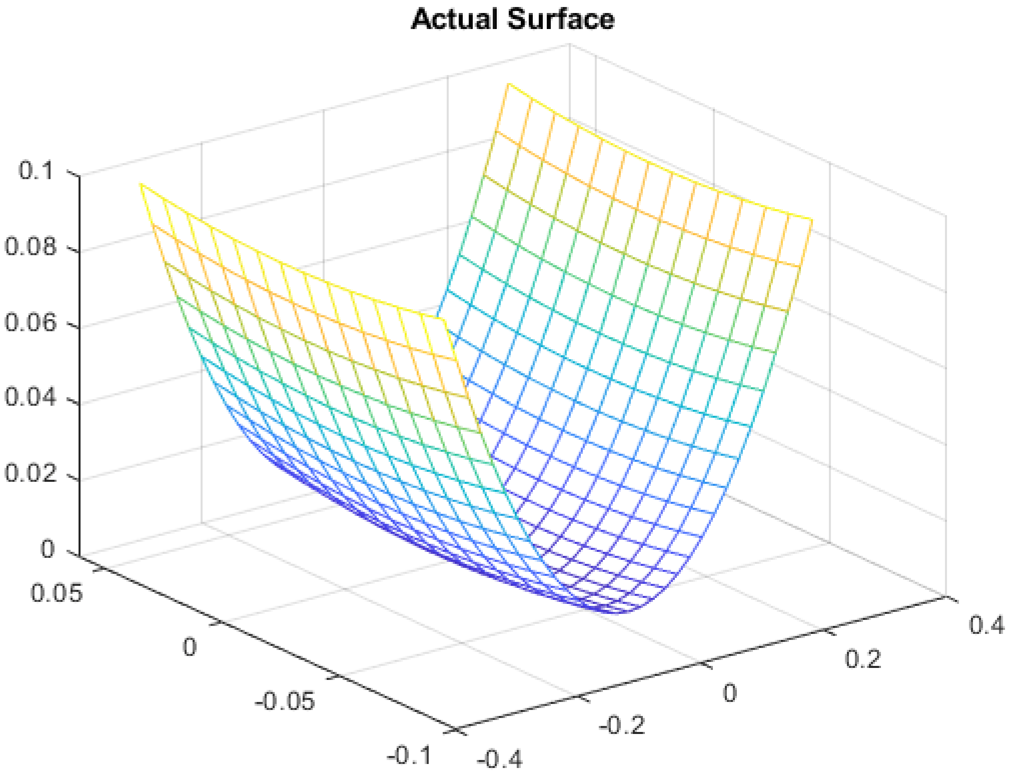
Now lift these sample points onto the surface $z = x^2 + y^2$ and interpolate the surface.

```
z = x.^2 + y.^2;
F = scatteredInterpolant(x,y,z);
[xi,yi] = meshgrid(-0.3:.02:0.3, -0.0688:0.01:0.0688);
zi = F(xi,yi);
mesh(xi,yi,zi)
xlabel('X','fontweight','b'), ylabel('Y','fontweight','b')
zlabel('Value - V','fontweight','b')
title('Interpolated Surface');
```



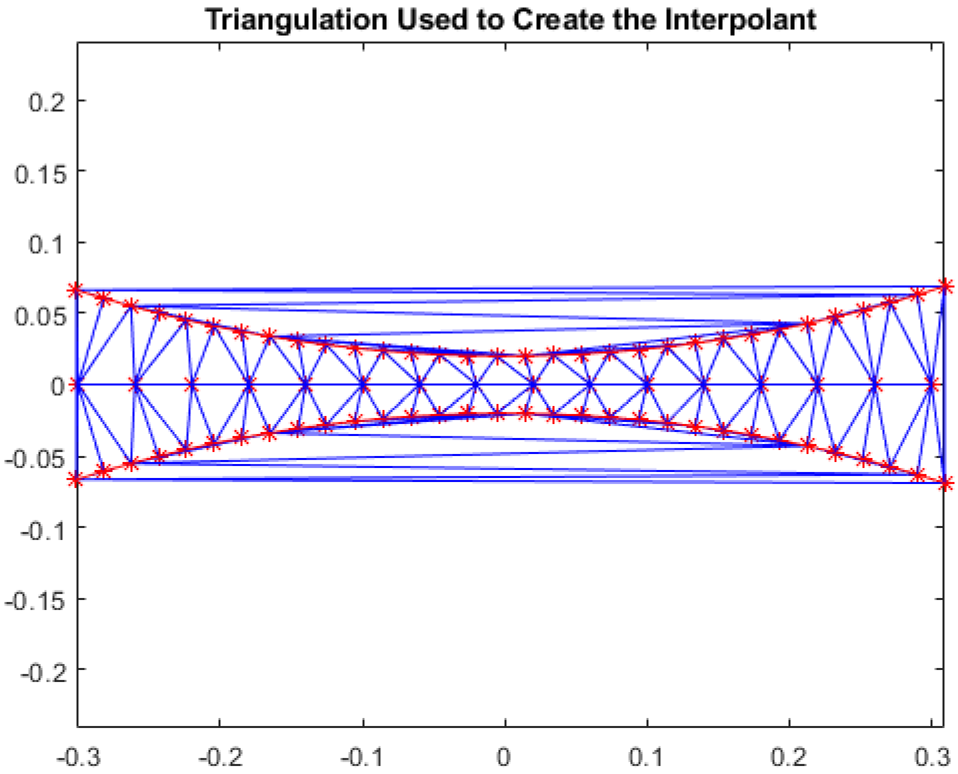
The actual surface is:

```
zi = xi.^2 + yi.^2;
figure
mesh(xi,yi,zi)
title('Actual Surface')
```



To understand why the interpolating surface deteriorates near the boundary, it is helpful to look at the underlying triangulation:

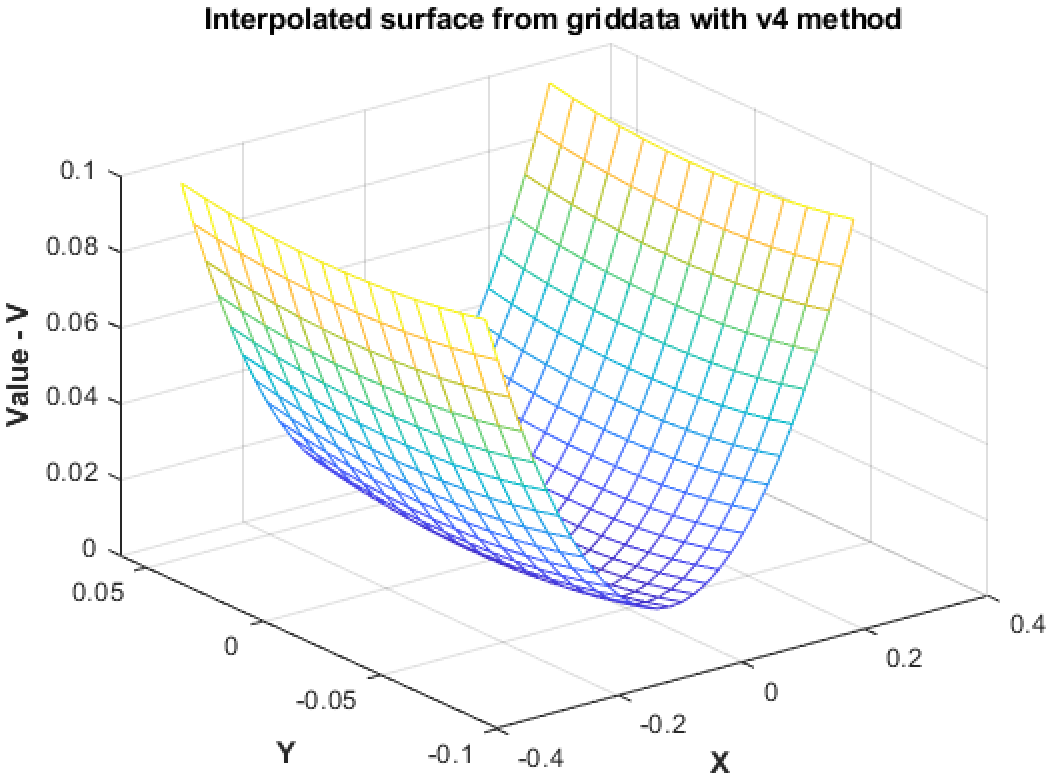
```
dt = delaunayTriangulation(x,y);
figure
plot(x,y,'*r')
axis equal
hold on
triplot(dt)
plot(x1,y1,'-r')
plot(x2,y2,'-r')
title('Triangulation Used to Create the Interpolant')
hold off
```



The triangles within the red boundaries are relatively well shaped; they are constructed from points that are in close proximity and the interpolation works well in this region. Outside the red boundary, the triangles are sliver-like and connect points that are remote from each other. There is not sufficient sampling to accurately capture the surface, so it is not surprising that the results in these regions are poor. In 3-D, visual inspection of the triangulation gets a bit trickier, but looking at the point distribution can often help illustrate potential problems.

The MATLAB® 4 `griddata` method, `'v4'`, is not triangulation-based and is not affected by deterioration of the interpolation surface near the boundary.

```
[xi,yi] = meshgrid(-0.3:.02:0.3, -0.0688:0.01:0.0688);
zi = griddata(x,y,z,xi,yi,'v4');
mesh(xi,yi,zi)
xlabel('X','fontweight','b'), ylabel('Y','fontweight','b')
zlabel('Value - V','fontweight','b')
title('Interpolated surface from griddata with v4 method','fontweight','b');
```



The interpolated surface from `griddata` using the `'v4'` method corresponds to the expected actual surface.

Interpolation Using a Specific Delaunay Triangulation

In this section...

Nearest-Neighbor Interpolation Using a delaunayTriangulation Query

This example shows how to perform nearest-neighbor interpolation on a scattered set of points using a specific Delaunay triangulation.

Create a `delaunayTriangulation` of a set of scattered points in 2-D.

```
rng('default')
P = -2.5 + 5*rand([50 2]);
DT = delaunayTriangulation(P)
```

```
DT = 
  delaunayTriangulation with properties:

              Points: [50x2 double]
    ConnectivityList: [84x3 double]
         Constraints: []
```

Sample a parabolic function, *V(x,y)*, at the points specified in `P`.

```
V = P(:,1).^2 + P(:,2).^2;
```

Define 10 random query points.

```
Pq = -2 + 4*rand([10 2]);
```

Perform nearest-neighbor interpolation on `V` using the triangulation, `DT`. Use `nearestNeighbor` to find the indices of the nearest-neighbor vertices, `vi`, for the set of query points, `Pq`. Then examine the specific values of `V` at the indices.

```
vi = nearestNeighbor(DT,Pq);
Vq = V(vi)
```

```
Vq = 10×1

    2.7208
    3.7792
    1.8394
    3.5086
    1.8394
    3.5086
    1.4258
    5.4053
    4.0670
    0.5586
```

Linear Interpolation Using a delaunayTriangulation Query

This example shows how to perform linear interpolation on a scattered set of points with a specific Delaunay triangulation.

You can use the `triangulation` method, `pointLocation`, to compute the enclosing triangle of a query point and the magnitudes of the vertex weights. The weights are called barycentric coordinates, and they represent a partition of unity. That is, the sum of the three weights equals 1. The interpolated value of a function, *V*, at a query point is the sum of the weighted values of *V* at the three vertices. That is, if the function has values, V1, V2, V3 at the three vertices, and the weights are B1, B2, B3, then the interpolated value is (V1)(B1) + (V2)(B2) + (V3)(B3).

Create a `delaunayTriangulation` of a set of scattered points in 2-D.

```
rng('default')
P = -2.5 + 5*rand([50 2]);
DT = delaunayTriangulation(P)
```

```
DT = 
  delaunayTriangulation with properties:

              Points: [50x2 double]
    ConnectivityList: [84x3 double]
         Constraints: []
```

Sample a parabolic function, *V(x,y)*, at the points in `P`.

```
V = P(:,1).^2 + P(:,2).^2;
```

Define 10 random query points.

```
Pq = -2 + 4*rand([10 2]);
```

Find the triangle that encloses each query point using the `pointLocation` method. In the code below, `ti` contains the IDs of the enclosing triangles and `bc` contains the barycentric coordinates associated with each triangle.

```
[ti,bc] = pointLocation(DT,Pq);
```

Find the values of *V(x,y)* at the vertices of each enclosing triangle.

```
triVals = V(DT(ti,:));
```

Calculate the sum of the weighted values of *V(x,y)* using the dot product.

```
Vq = dot(bc',triVals')'
```

```
Vq = 10×1

    2.2736
    4.2596
    2.1284
    3.5372
    4.6232
    2.1797
    1.2779
    4.7644
    3.6311
    1.2196
```

See Also

`delaunayTriangulation` | `nearestNeighbor` | `pointLocation`

More About

- "Interpolating Scattered Data" on page 8-17

Extrapolating Scattered Data

Factors That Affect the Accuracy of Extrapolation

`scatteredInterpolant` provides functionality for approximating values at points that fall outside the convex hull. The `'linear'` extrapolation method is based on a least-squares approximation of the gradient at the boundary of the convex hull. The values it returns for query points outside the convex hull are based on the values and gradients at the boundary. The quality of the solution depends on how well you've sampled your data. If your data is coarsely sampled, the quality of the extrapolation is poor.

In addition, the triangulation near the convex hull boundary can have sliver-like triangles. These triangles can compromise your extrapolation results in the same way that they can compromise interpolation results. See "Interpolation Results Poor Near the Convex Hull" on page 8-36 for more information.

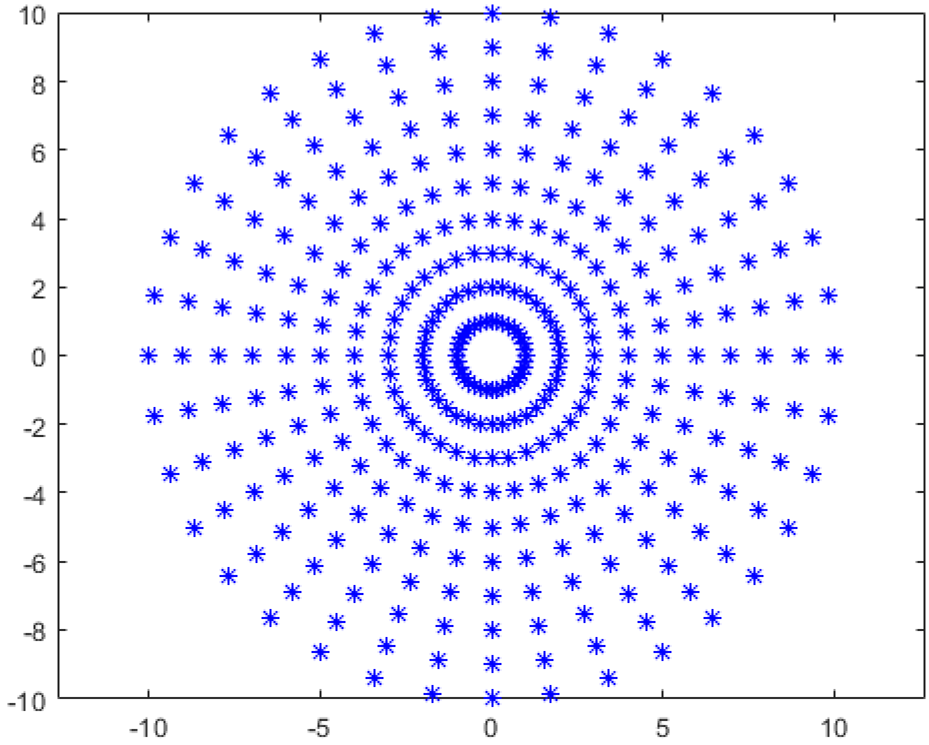
You should inspect your extrapolation results visually using your knowledge of the behavior outside the domain.

Compare Extrapolation of Coarsely and Finely Sampled Scattered Data

This example shows how to interpolate two different samplings of the same parabolic function. It also shows that a better distribution of sample points produces better extrapolation results.

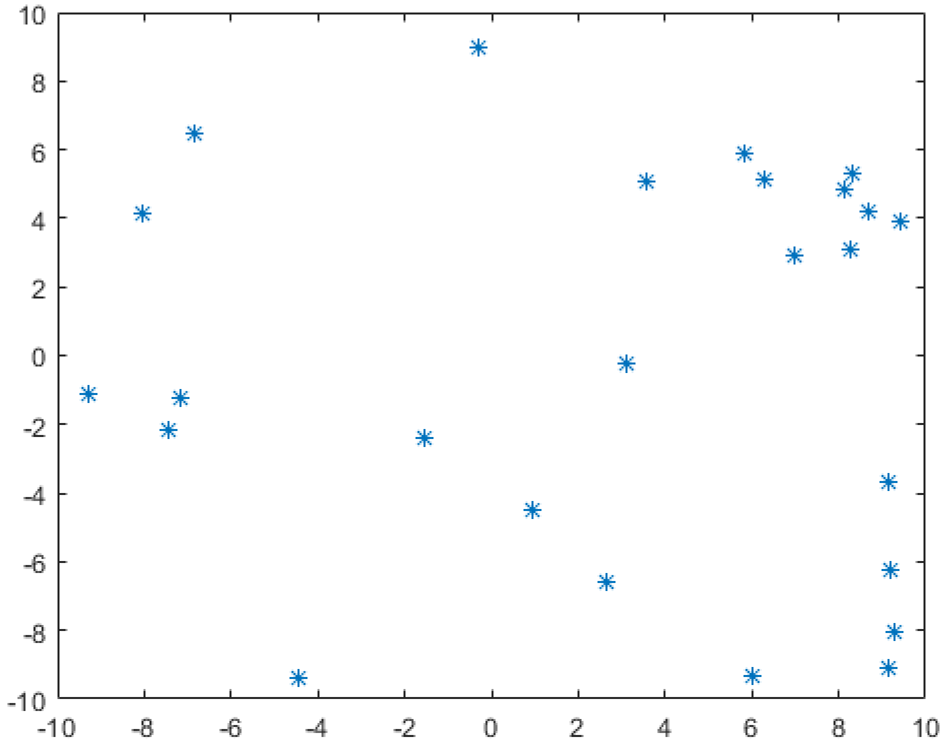
Create a radial distribution of points spaced 10 degrees apart around 10 concentric circles. Use `bsxfun` to compute the coordinates, $x = \cos\theta$ and $y = \sin\theta$.

```
theta = 0:10:350;
c = cosd(theta);
s = sind(theta);
r = 1:10;

x1 = bsxfun(@times,r.',c);
y1 = bsxfun(@times,r.',s);

figure
plot(x1,y1,'*b')
axis equal
```



Create a second, more coarsely distributed set of points. Use the `rand` function to create random samplings in the range, [-10, 10].

```
rng default;
x2 = -10 + 20*rand([25 1]);
y2 = -10 + 20*rand([25 1]);
figure
plot(x2,y2,'*')
```



Sample a parabolic function, `v(x,y)`, at both sets of points.

```
v1 = x1.^2 + y1.^2;
v2 = x2.^2 + y2.^2;
```

Create a `scatteredInterpolant` for each sampling of `v(x,y)`.

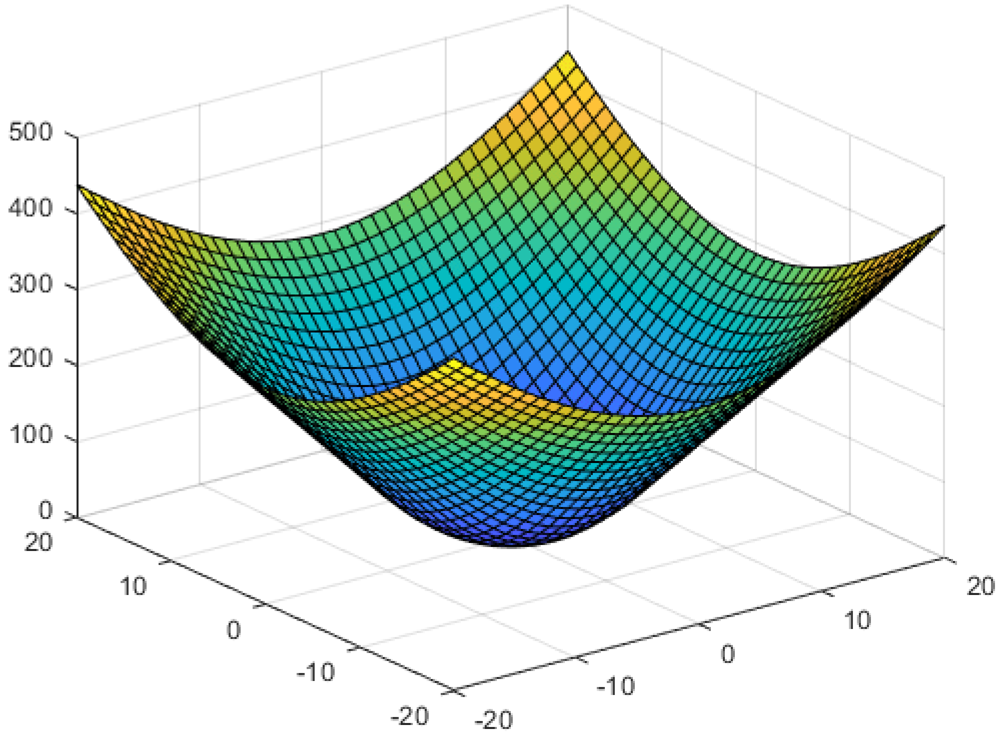
```
F1 = scatteredInterpolant(x1(:),y1(:),v1(:));
F2 = scatteredInterpolant(x2(:),y2(:),v2(:));
```

Create a grid of query points that extend beyond each domain.

```
[xq,yq] = ndgrid(-20:20);
```

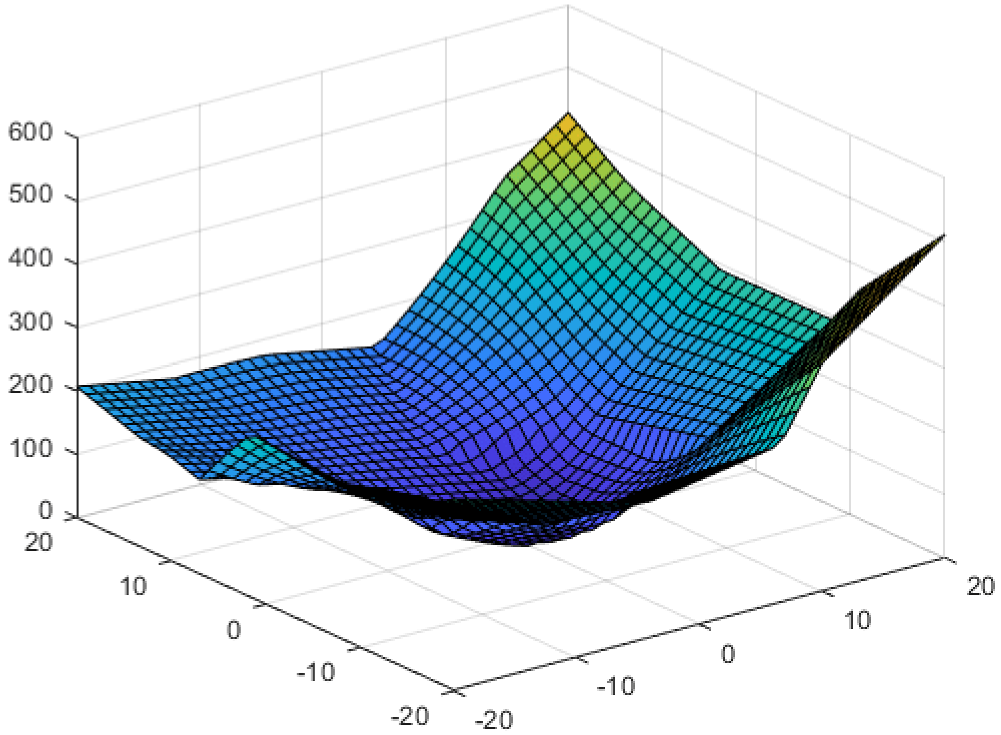
Evaluate `F1` and plot the results.

```
figure
vq1 = F1(xq,yq);
surf(xq,yq,vq1)
```



Evaluate F2 and plot the results.

```
figure
vq2 = F2(xq,yq);
surf(xq,yq,vq2)
```



The quality of the extrapolation is not as good for `F2` because of the coarse sampling of points in `v2`.

Extrapolation of 3-D Data

This example shows how to extrapolate a well sampled 3-D gridded dataset using `scatteredInterpolant`. The query points lie on a planar grid that is completely outside domain.

Create a 10-by-10-by-10 grid of sample points. The points in each dimension are in the range, [-10, 10].

```
[x,y,z] = ndgrid(-10:10);
```

Sample a function, *v(x,y,z)*, at the sample points.

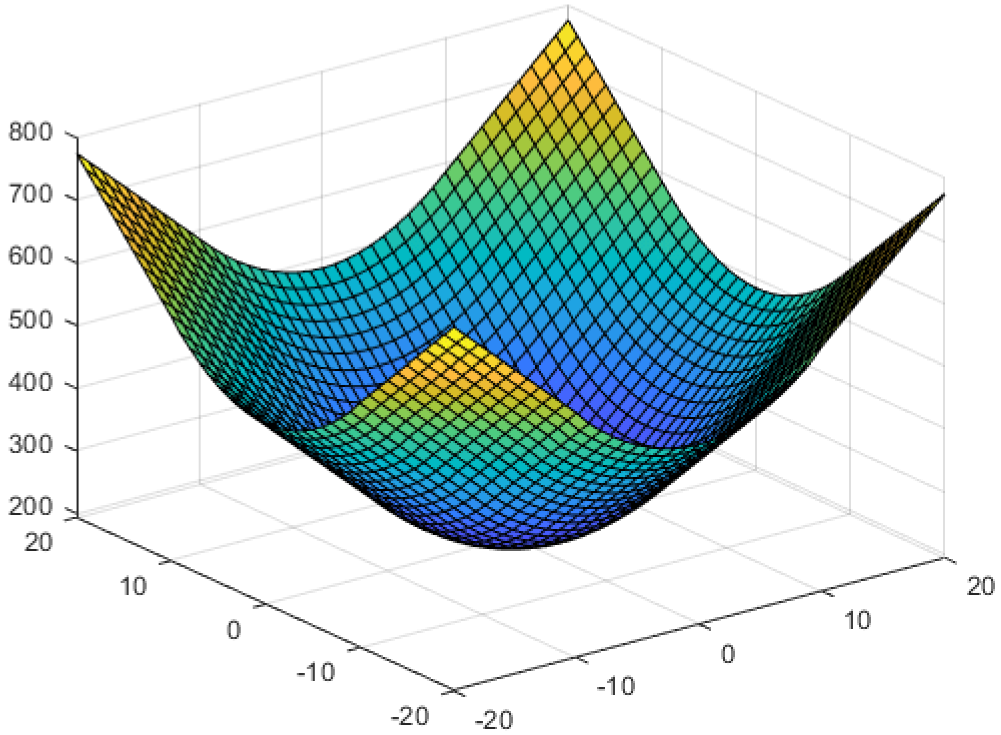
```
v = x.^2 + y.^2 + z.^2;
```

Create a `scatteredInterpolant`, specifying linear interpolation and extrapolation.

```
F = scatteredInterpolant(x(:),y(:),z(:),v(:),'linear','linear');
```

Evaluate the interpolant over an *x*-*y* grid spanning the range, [-20,20] at an elevation, *z* = 15.

```
[xq,yq,zq] = ndgrid(-20:20,-20:20,15);
vq = F(xq,yq,zq);
figure
surf(xq,yq,vq)
```



The extrapolation returned good results because the function is well sampled.

Normalize Data with Differing Magnitudes

This example shows how to use normalization to improve scattered data interpolation results with griddata. Normalization can improve the interpolation results in some cases, but in others it can compromise the accuracy of the solution. Whether to use normalization is a judgment made based on the nature of the data being interpolated.

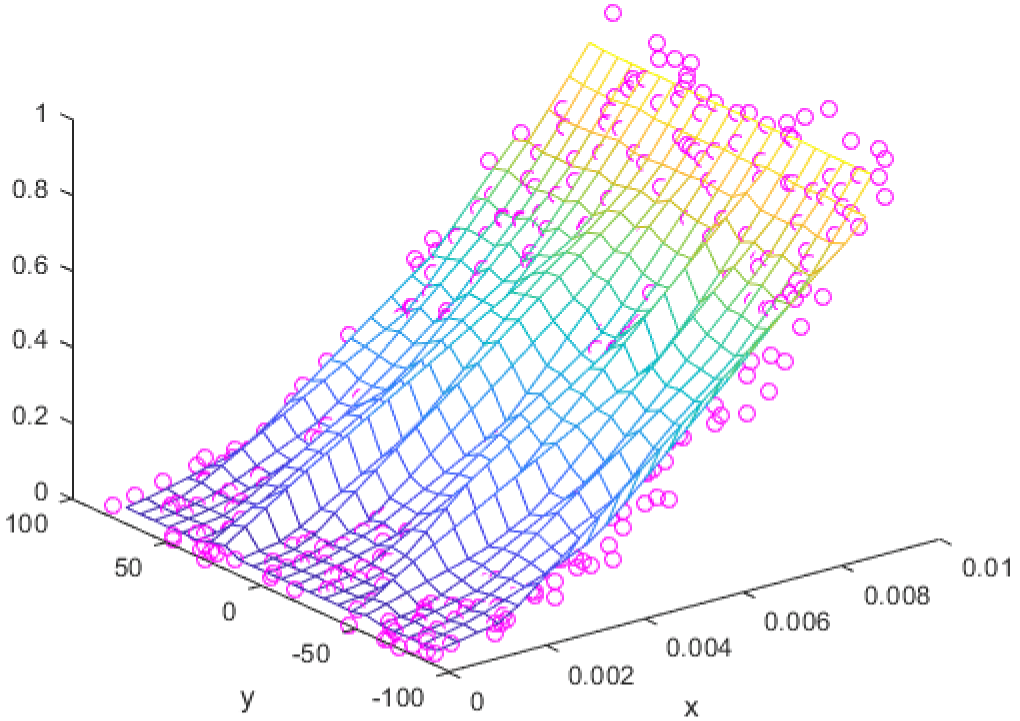
- **Benefits:** Normalizing your data can potentially improve the interpolation result when the independent variables have different units and substantially different scales. In this case, scaling the inputs to have similar magnitudes might improve the numerical aspects of the interpolation. An example where normalization would be beneficial is if x represents engine speed in RPMs from 500 to 3500, and y represents engine load from 0 to 1. The scales of x and y differ by a few orders of magnitude, and they have different units.
- **Cautions:** Use caution when normalizing your data if the independent variables have the same units, even if the scales of the variables are different. With data of the same units, normalization distorts the solution by adding a directional bias, which affects the underlying triangulation and ultimately compromises the accuracy of the interpolation. An example where normalization is erroneous is if both x and y represent locations and have units of meters. Scaling x and y unequally is not recommended because 10 m due East should be spatially the same as 10 m due North.

Create some sample data where the values in y are a few orders of magnitude larger than those in x. Assume that x and y have different units.

```
x = rand(1,500)/100;
y = 2.*(rand(1,500)-0.5).*90;
z = (x.*1e2).^2;
```

Use the sample data to construct a grid of query points. Interpolate the sample data on the grid and plot the results.

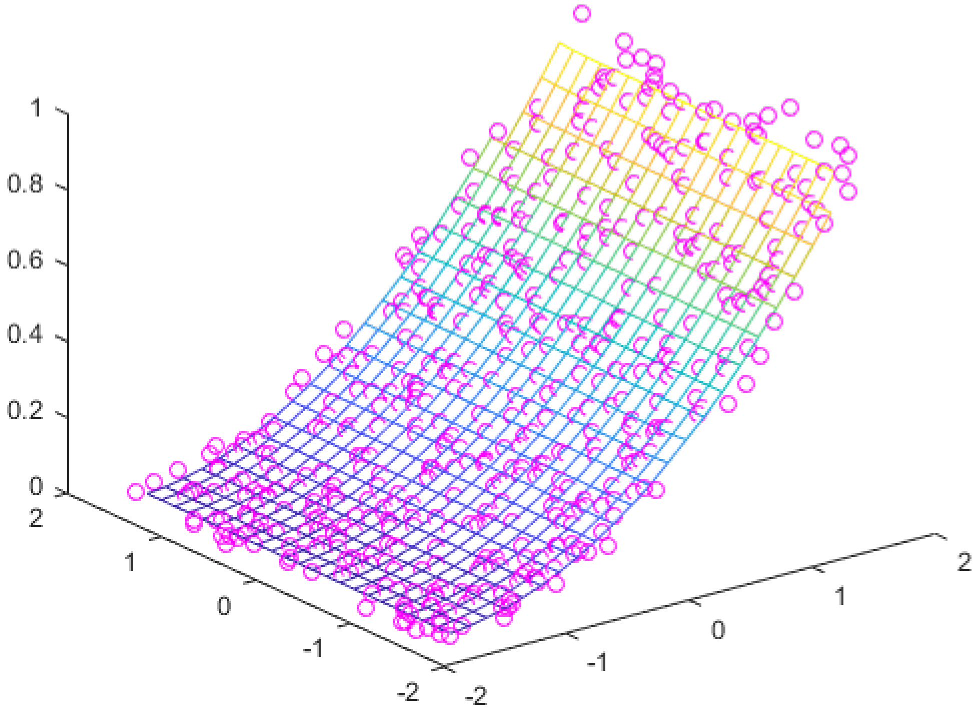
```
X = linspace(min(x),max(x),25);
Y = linspace(min(y),max(y),25);
[xq, yq] = meshgrid(X,Y);
zq = griddata(x,y,z,xq,yq);

plot3(x,y,z,'mo')
hold on
mesh(xq,yq,zq)
xlabel('x')
ylabel('y')
hold off
```



The result produced by `griddata` is not very smooth and seems to be noisy. The differing scales in the independent variables contribute to this, since a small change in the size of one variable can lead to a much larger change in the size of the other variable.

Since `x` and `y` have different units, normalizing them so that they have similar magnitudes should help produce better results. Normalize the sample points using *z*-scores and regenerate the interpolation using `griddata`.

```
% Normalize Sample Points
x = normalize(x);
y = normalize(y);

% Regenerate Grid
X = linspace(min(x),max(x),25);
Y = linspace(min(y),max(y),25);
[xq, yq] = meshgrid(X,Y);

% Interpolate and Plot
zq = griddata(x,y,z,xq,yq);
plot3(x,y,z,'mo')
hold on
mesh(xq,yq,zq)
```



In this case, normalizing the sample points permits `griddata` to compute a smoother solution.

See Also

`griddata` | `griddatan` | `scatteredInterpolant`

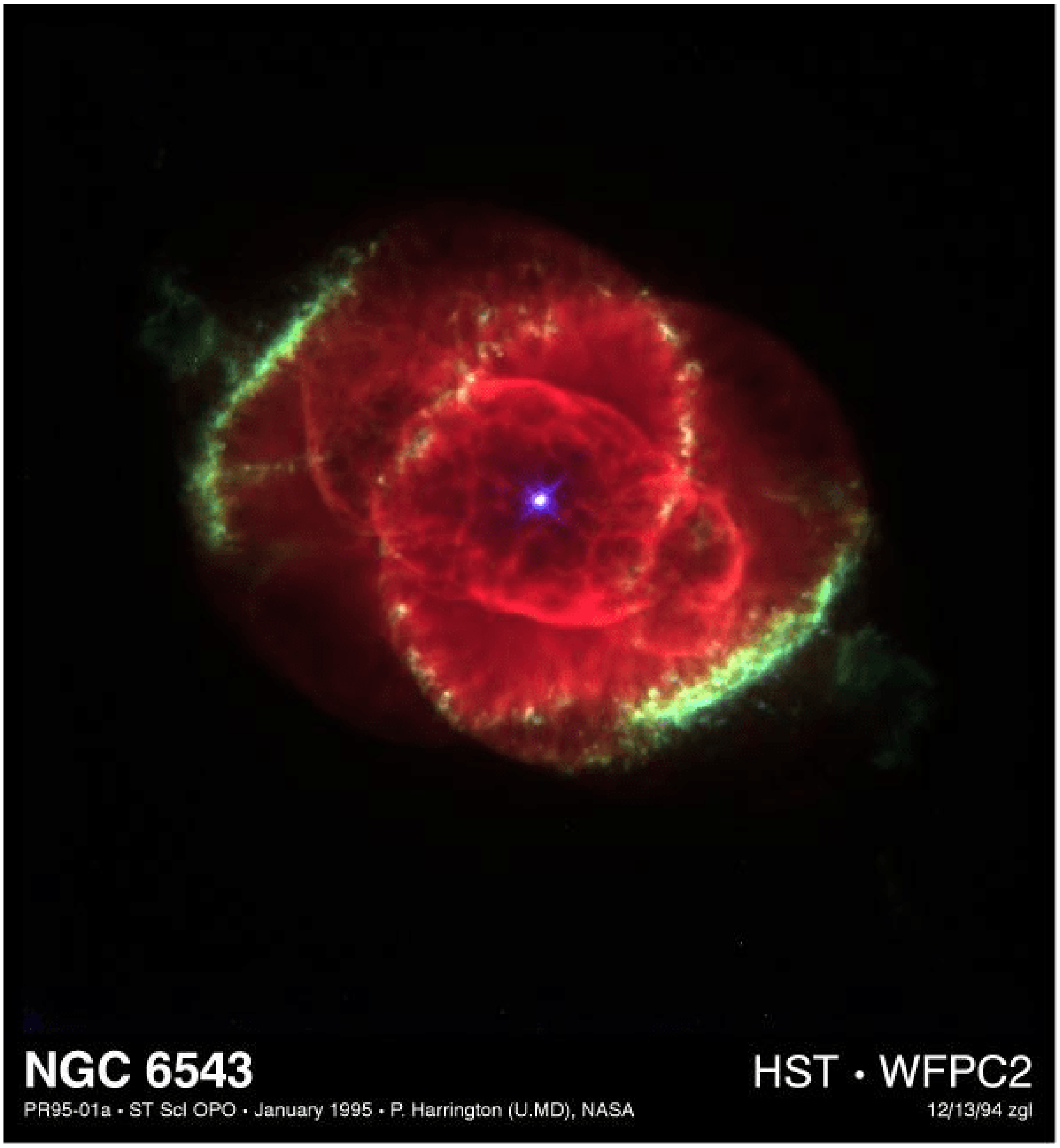
Resample Image with Gridded Interpolation

This example shows how to use `griddedInterpolant` to resample the pixels in an image. Resampling an image is useful for adjusting the resolution and size, and you also can use it to smooth out the pixels after zooming.

Load Image

Load and show the image `ngc6543a.jpg`, which is a Hubble Space Telescope image of the planetary nebulae NGC 6543. This image displays several interesting structures, such as concentric gas shells, jets of high-speed gas, and unusual knots of gas. The matrix `A` that represents the image is a 650-by-600-by-3 matrix of `uint8` integers.

```
A = imread('ngc6543a.jpg');
imshow(A)
```


NGC 6543
HST • WFPC2
PR95-01a • ST ScI OPO • January 1995 • P. Harrington (U.MD), NASA
12/13/94 zgl

Create Interpolant

Create a gridded interpolant object for the image. `griddedInterpolant` only works for double-precision and single-precision matrices, so convert the `uint8` matrix to `double`. To interpolate each RGB channel of the image, specify two grid vectors to describe the sample points in the first two dimensions. The grid vectors are grouped together as column vectors in a cell array `{xg1,xg2,...,xgN}`. With this formulation, `griddedInterpolant` treats the 3-D matrix as containing multiple 2-D data sets defined on the same grid.

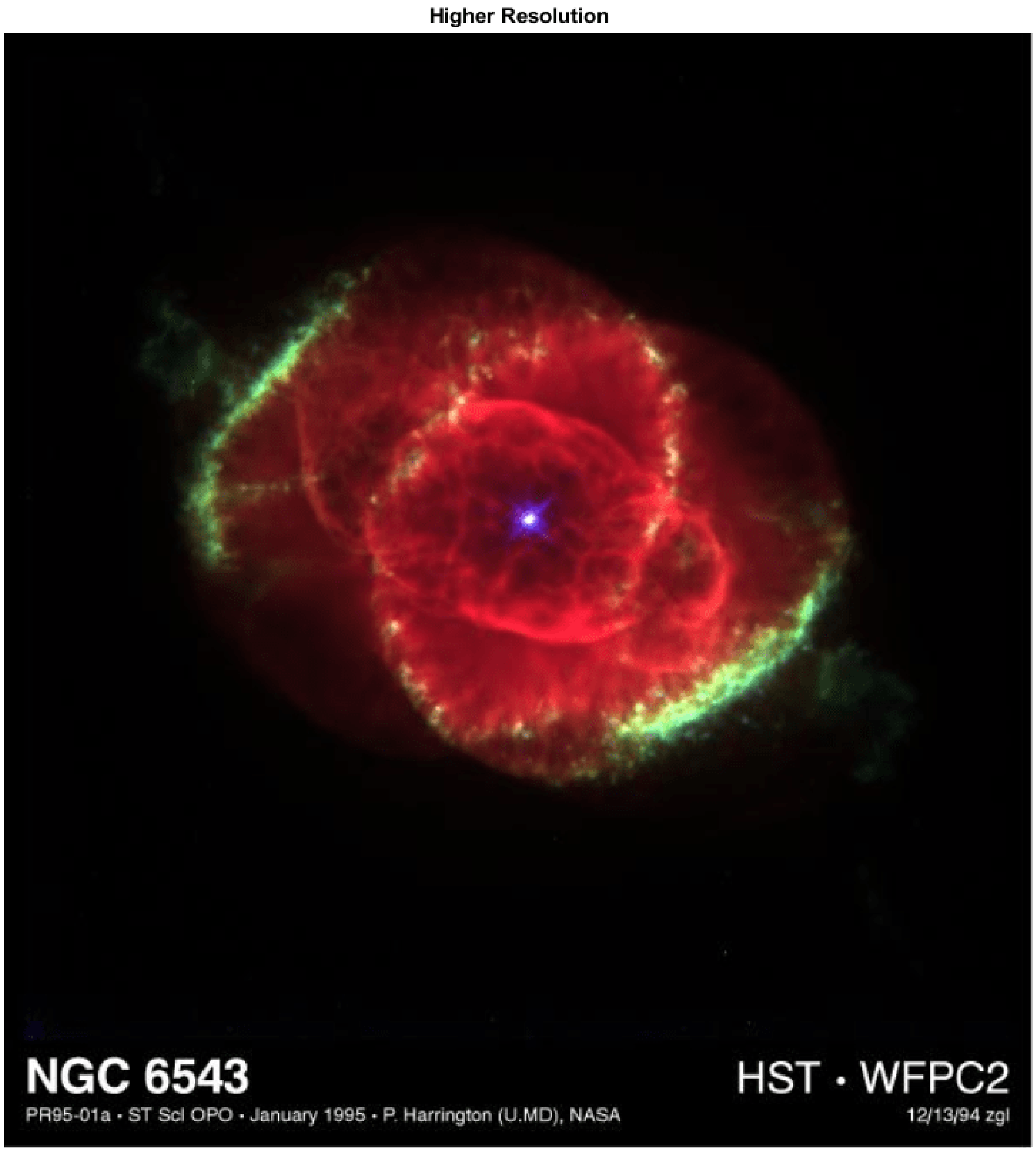
```
sz = size(A);
xg = 1:sz(1);
```

```
yg = 1:sz(2);
F = griddedInterpolant({xg,yg},double(A));
```

Resample Image Pixels

Use the sizes of the first two matrix dimensions to resample the image so that it is 120% the size. That is, for each 5 pixels in the original image, the interpolated image has 6 pixels. Evaluate the interpolant at the query points with the syntax `F({xq,yq})`. `griddedInterpolant` evaluates each page in the 3-D image at the query points.

```
xq = (0:5/6:sz(1))';
yq = (0:5/6:sz(2))';
vq = uint8(F({xq,yq}));
imshow(vq)
title('Higher Resolution')
```


Higher Resolution
NGC 6543
PR95-01a • ST ScI OPO • January 1995 • P. Harrington (U.MD), NASA
HST • WFPC2
12/13/94 zgl

Similarly, reduce the size of the image by querying the interpolant with 55% fewer points than the original image. While you can simply index into the original image matrix to produce lower resolution images, interpolation enables you to resample the image at noninteger pixel locations.

```
xq = (0:1.55:sz(1))';
yq = (0:1.55:sz(2))';
vq = uint8(F({xq,yq}));
figure
imshow(vq)
title('Lower Resolution')
```

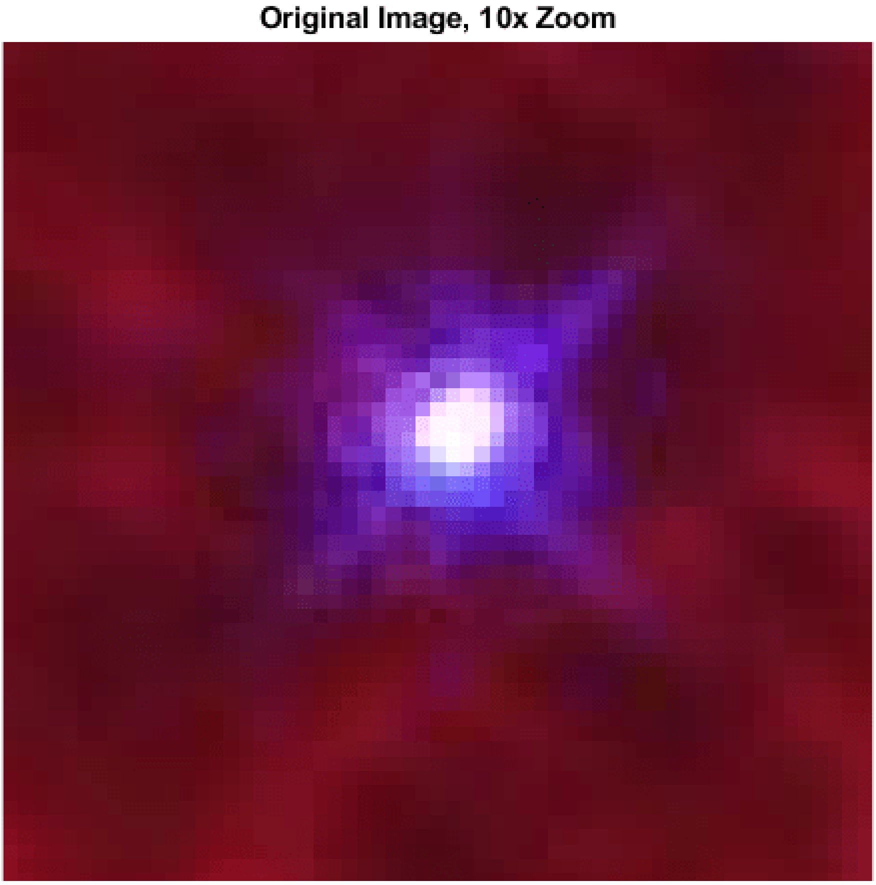
Lower Resolution
NGC 6543
HST · WFPC2
PR95-01a · ST ScI OPO · January 1995 · P. Harrington (U.MD), NASA
12/13/94 zgl

Smooth Out Zooming Artifacts

As you zoom in on an image, the pixels in the region of interest become larger and detail in the image is quickly lost. You can use image resampling to smooth out these zooming artifacts.

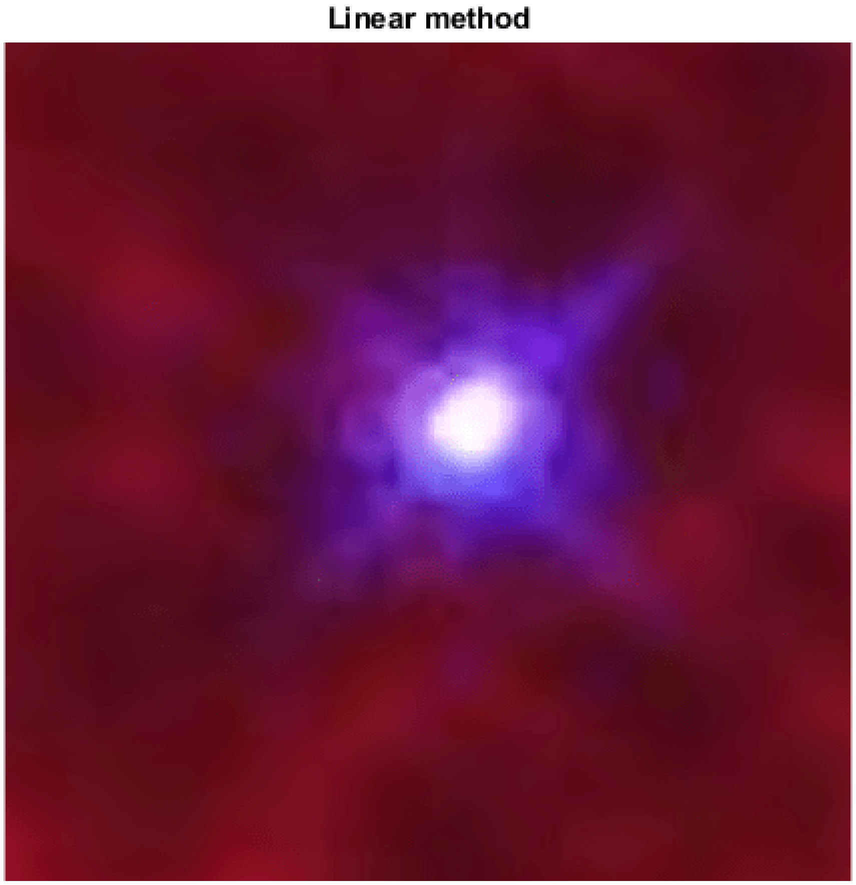
Zoom in on the bright spot in the center of the original image. (The indexing into A is to center this bright spot in the image so that subsequent zooming does not push it out of the frame.)

```
imshow(A(1:570,10:600,:),'InitialMagnification','fit')
zoom(10)
title('Original Image, 10x Zoom')
```

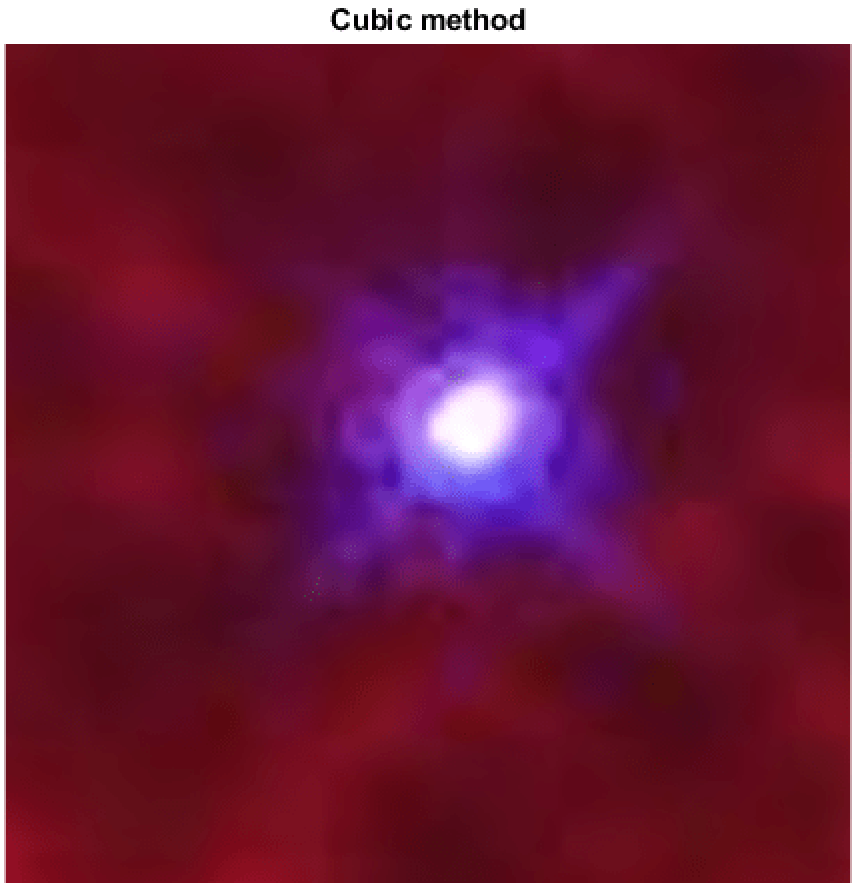


Query the interpolant F to reproduce this zoomed image (approximately) with 10x higher resolution. Compare the results from several different interpolation methods.

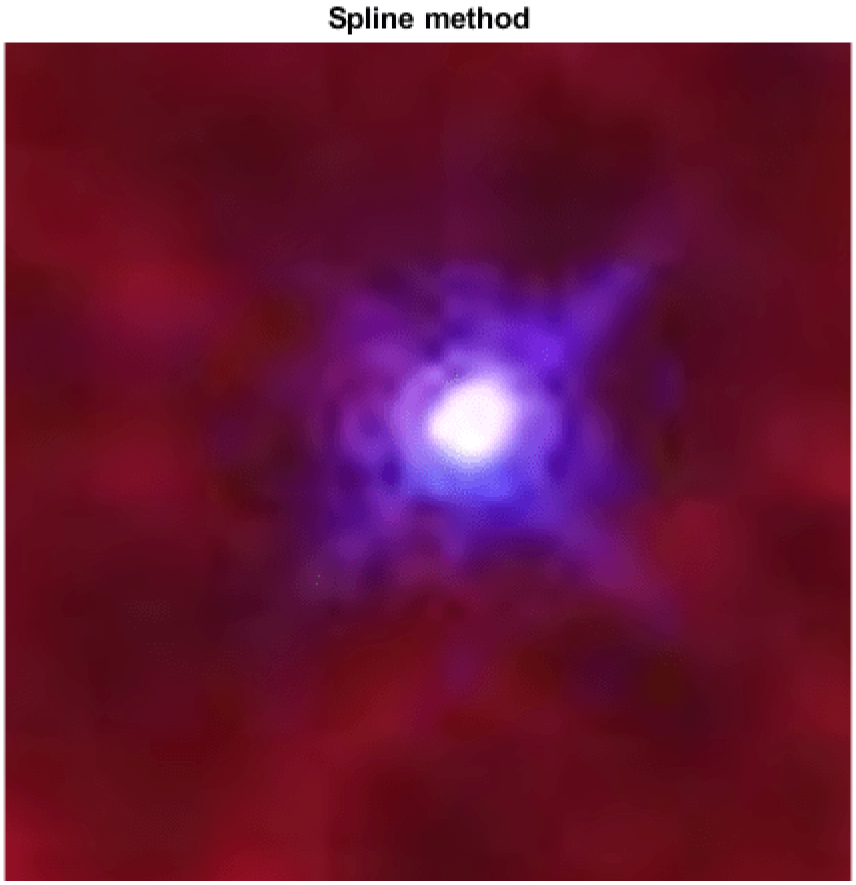
```
xq = (1:0.1:sz(1))';
yq = (1:0.1:sz(2))';
F.Method = 'linear';
vq = uint8(F({xq,yq}));
imshow(vq(1:5700,150:5900,:),'InitialMagnification','fit')
zoom(10)
title('Linear method')
```



```
F.Method = 'cubic';
vq = uint8(F({xq,yq}));
imshow(vq(1:5700,150:5900,:),'InitialMagnification','fit')
zoom(10)
title('Cubic method')
```



```
F.Method = 'spline';
vq = uint8(F({xq,yq}));
imshow(vq(1:5700,150:5900,:),'InitialMagnification','fit')
zoom(10)
title('Spline method')
```



See Also

`griddedInterpolant` | `imshow`

More About

- “Interpolating Gridded Data” on page 8-5

Optimization

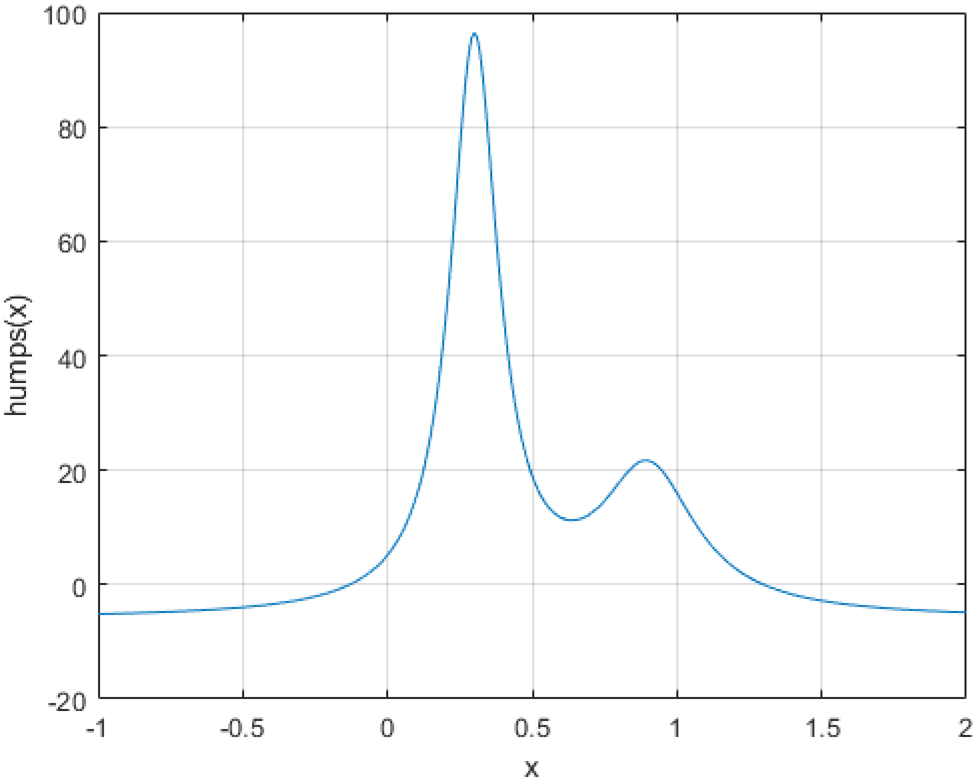
Optimizing Nonlinear Functions

In this section...

Minimizing Functions of One Variable

Given a mathematical function of a single variable, you can use the fminbnd function to find a local minimizer of the function in a given interval. For example, consider the humps.m function, which is provided with MATLAB®. The following figure shows the graph of humps.

```
x = -1:.01:2;
y = humps(x);
plot(x,y)
xlabel('x')
ylabel('humps(x)')
grid on
```



To find the minimum of the humps function in the range (0.3,1), use

```
x = fminbnd(@humps,0.3,1)
```

```
x = 0.6370
```

You can see details of the solution process by using `optimset` to create options with the `Display` option set to `'iter'`. Pass the resulting options to `fminbnd`.

```
options = optimset('Display','iter');
x = fminbnd(@humps,0.3,1,options)
```

```
 Func-count     x          f(x)         Procedure
    1        0.567376      12.9098        initial
    2        0.732624      13.7746        golden
    3        0.465248      25.1714        golden
    4        0.644416      11.2693        parabolic
    5          0.6413      11.2583        parabolic
    6        0.637618      11.2529        parabolic
    7        0.636985      11.2528        parabolic
    8        0.637019      11.2528        parabolic
    9        0.637052      11.2528        parabolic

Optimization terminated:
 the current x satisfies the termination criteria using OPTIONS.TolX of 1.000000e-04

x = 0.6370
```

The iterative display shows the current value of `x` and the function value at `f(x)` each time a function evaluation occurs. For `fminbnd`, one function evaluation corresponds to one iteration of the algorithm. The last column shows the procedure `fminbnd` uses at each iteration, a golden section search or a parabolic interpolation. For details, see “Optimization Solver Iterative Display” on page 9-13.

Minimizing Functions of Several Variables

The `fminsearch` function is similar to `fminbnd` except that it handles functions of many variables. Specify a starting vector x_0 rather than a starting interval. `fminsearch` attempts to return a vector *x* that is a local minimizer of the mathematical function near this starting vector.

To try `fminsearch`, create a function `three_var` of three variables, `x`, `y`, and `z`.

```
function b = three_var(v)
x = v(1);
y = v(2);
z = v(3);
b = x.^2 + 2.5*sin(y) - z^2*x^2*y^2;
```

Now find a minimum for this function using `x = -0.6`, `y = -1.2`, and `z = 0.135` as the starting values.

```
v = [-0.6,-1.2,0.135];
a = fminsearch(@three_var,v)

a =
    0.0000   -1.5708    0.1803
```

Maximizing Functions

The fminbnd and fminsearch solvers attempt to minimize an objective function. If you have a maximization problem, that is, a problem of the form

$$\max_x f(x),$$

then define $g(x) = -f(x)$, and minimize g.

For example, to find the maximum of tan(cos(x)) near $x = 5$, evaluate:

```
[x fval] = fminbnd(@(x)-tan(cos(x)),3,8)

x =
    6.2832

fval =
   -1.5574
```

The maximum is 1.5574 (the negative of the reported fval), and occurs at $x = 6.2832$. This answer is correct since, to five digits, the maximum is tan(1) = 1.5574, which occurs at $x = 2\pi = 6.2832$.

fminsearch Algorithm

fminsearch uses the Nelder-Mead simplex algorithm as described in Lagarias et al. [1]. This algorithm uses a simplex of $n + 1$ points for n-dimensional vectors x. The algorithm first makes a simplex around the initial guess x_0 by adding 5% of each component $x_0(i)$ to x_0. The algorithm uses these n vectors as elements of the simplex in addition to x_0. (The algorithm uses 0.00025 as component i if $x_0(i) = 0$.) Then, the algorithm modifies the simplex repeatedly according to the following procedure.

Note The keywords for the fminsearch iterative display appear in **bold** after the description of the step.

1. Let $x(i)$ denote the list of points in the current simplex, $i = 1,...,n + 1$.
2. Order the points in the simplex from lowest function value $f(x(1))$ to highest $f(x(n + 1))$. At each step in the iteration, the algorithm discards the current worst point $x(n + 1)$, and accepts another point into the simplex. [Or, in the case of step 7 below, it changes all n points with values above $f(x(1))$].
3. Generate the reflected point

   $$r = 2m - x(n + 1), \tag{9-1}$$

   where

   $$m = \Sigma x(i)/n,\ i = 1...n, \tag{9-2}$$

   and calculate $f(r)$.
4. If $f(x(1)) \le f(r) < f(x(n))$, accept r and terminate this iteration. **Reflect**
5. If $f(r) < f(x(1))$, calculate the expansion point s

$$s = m + 2(m - x(n + 1)), \tag{9-3}$$

and calculate $f(s)$.

a If $f(s) < f(r)$, accept s and terminate the iteration. **Expand**

b Otherwise, accept r and terminate the iteration. **Reflect**

6 If $f(r) \geq f(x(n))$, perform a contraction between m and either$x(n + 1)$ or r, depending on which has the lower objective function value.

a If $f(r) < f(x(n + 1))$ (that is, r is better than $x(n + 1)$), calculate

$$c = m + (r - m)/2 \tag{9-4}$$

and calculate $f(c)$. If $f(c) < f(r)$, accept c and terminate the iteration. **Contract outside**

Otherwise, continue with Step 7 (Shrink).

b If $f(r) \geq f(x(n + 1))$, calculate

$$cc = m + (x(n + 1) - m)/2 \tag{9-5}$$

and calculate $f(cc)$. If $f(cc) < f(x(n + 1))$, accept cc and terminate the iteration. **Contract inside**

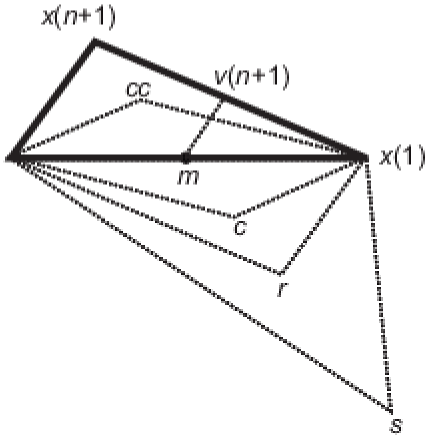
Otherwise, continue with Step 7 (Shrink).

7 Calculate the n points

$$v(i) = x(1) + (x(i) - x(1))/2 \tag{9-6}$$

and calculate $f(v(i))$, $i = 2,...,n + 1$. The simplex at the next iteration is $x(1)$, $v(2),...,v(n + 1)$. **Shrink**

The following figure shows the points that `fminsearch` can calculate in the procedure, along with each possible new simplex. The original simplex has a bold outline. The iterations proceed until they meet a stopping criterion.



Reference

[1] Lagarias, J. C., J. A. Reeds, M. H. Wright, and P. E. Wright. “Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions.” *SIAM Journal of Optimization*, Vol. 9, Number 1, 1998, pp. 112–147.

See Also

More About

- “Optimization Troubleshooting and Tips” on page 9-28
- “Nonlinear Optimization” (Optimization Toolbox)
- “Curve Fitting via Optimization” on page 9-7

Curve Fitting via Optimization

This example shows how to fit a nonlinear function to data. For this example, the nonlinear function is the standard exponential decay curve

$$y(t) = A\exp(-\lambda t),$$

where $y(t)$ is the response at time t, and A and λ are the parameters to fit. Fitting the curve means finding parameters A and λ that minimize the sum of squared errors

$$\sum_{i=1}^{n} (y_i - A\exp(-\lambda t_i))^2,$$

where the times are t_i and the responses are $y_i, i = 1, \ldots, n$. The sum of squared errors is the objective function.

Create Sample Data

Usually, you have data from measurements. For this example, create artificial data based on a model with $A = 40$ and $\lambda = 0.5$, with normally distributed pseudorandom errors.

```
rng default % for reproducibility
tdata = 0:0.1:10;
ydata = 40*exp(-0.5*tdata) + randn(size(tdata));
```

Write Objective Function

Write a function that accepts parameters `A` and `lambda` and data `tdata` and `ydata`, and returns the sum of squared errors for the model $y(t)$. Put all the variables to optimize (`A` and `lambda`) in a single vector variable (`x`). For more information, see “Minimizing Functions of Several Variables” on page 9-3.

```
type sseval
```

```
function sse = sseval(x,tdata,ydata)
A = x(1);
lambda = x(2);
sse = sum((ydata - A*exp(-lambda*tdata)).^2);
```

Save this objective function as a file named `sseval.m` on your MATLAB® path.

The `fminsearch` solver applies to functions of one variable, `x`. However, the `sseval` function has three variables. The extra variables `tdata` and `ydata` are not variables to optimize, but are data for the optimization. Define the objective function for `fminsearch` as a function of `x` alone:

```
fun = @(x)sseval(x,tdata,ydata);
```

For information about including extra parameters such as `tdata` and `ydata`, see “Parameterizing Functions” on page 10-2.

Find the Best Fitting Parameters

Start from a random positive set of parameters `x0`, and have `fminsearch` find the parameters that minimize the objective function.

```
x0 = rand(2,1);
bestx = fminsearch(fun,x0)
```

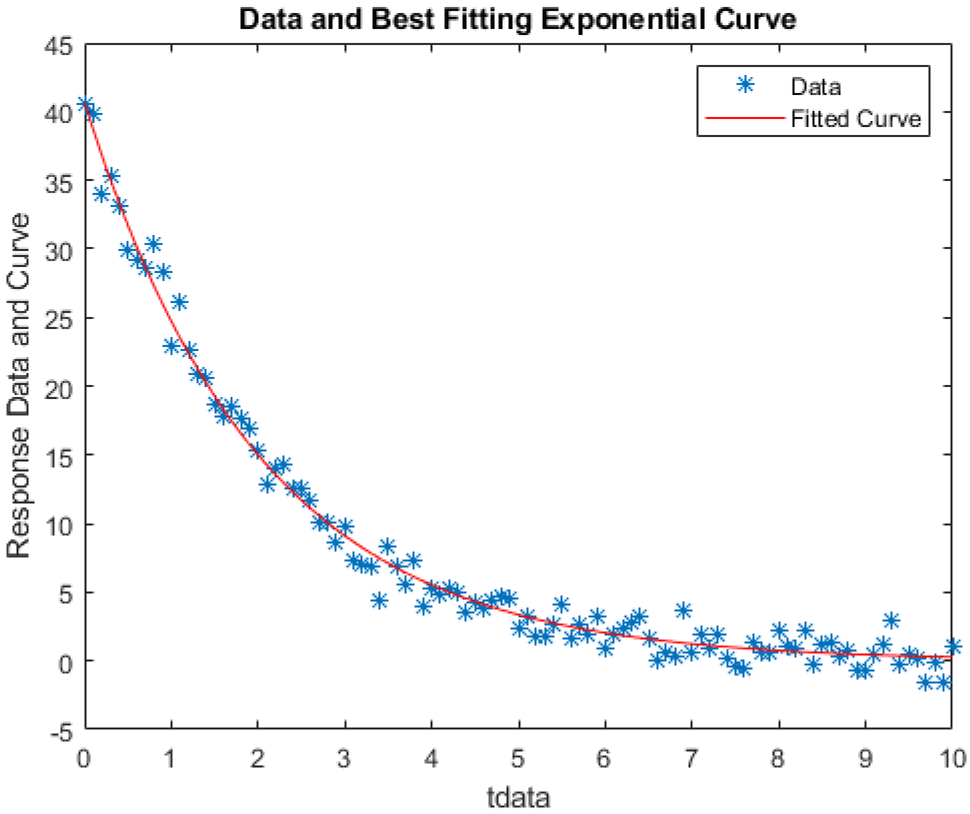
```
bestx = 2×1

   40.6877
    0.4984
```

The result `bestx` is reasonably near the parameters that generated the data, `A = 40` and `lambda = 0.5`.

Check the Fit Quality

To check the quality of the fit, plot the data and the resulting fitted response curve. Create the response curve from the returned parameters of your model.

```
A = bestx(1);
lambda = bestx(2);
yfit = A*exp(-lambda*tdata);
plot(tdata,ydata,'*');
hold on
plot(tdata,yfit,'r');
xlabel('tdata')
ylabel('Response Data and Curve')
title('Data and Best Fitting Exponential Curve')
legend('Data','Fitted Curve')
hold off
```



See Also

More About

- “Optimizing Nonlinear Functions” on page 9-2
- “Nonlinear Data-Fitting” (Optimization Toolbox)
- “Nonlinear Regression” (Statistics and Machine Learning Toolbox)

Set Optimization Options

In this section...

How to Set Options

You can specify optimization parameters using an `options` structure that you create using the `optimset` function. You then pass `options` as an input to the optimization function, for example, by calling `fminbnd` with the syntax

```
x = fminbnd(fun,x1,x2,options)
```

or `fminsearch` with the syntax

```
x = fminsearch(fun,x0,options)
```

For example, to display output from the algorithm at each iteration, set the `Display` option to `'iter'`:

```
options = optimset('Display','iter');
```

Options Table

Option	Description	Solvers
`Display`	A flag indicating whether intermediate steps appear on the screen. • `'notify'` (default) displays output only if the function does not converge. • `'iter'` displays intermediate steps (not available with `lsqnonneg`). See “Optimization Solver Iterative Display” on page 9-13. • `'off'` or `'none'` displays no intermediate steps. • `'final'` displays just the final output.	`fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`
`FunValCheck`	Check whether objective function values are valid. • `'on'` displays an error when the objective function or constraints return a value that is complex or `NaN`. • `'off'` (default) displays no error.	`fminbnd`, `fminsearch`, `fzero`
`MaxFunEvals`	The maximum number of function evaluations allowed. The default value is `500` for `fminbnd` and `200*length(x0)` for `fminsearch`.	`fminbnd`, `fminsearch`

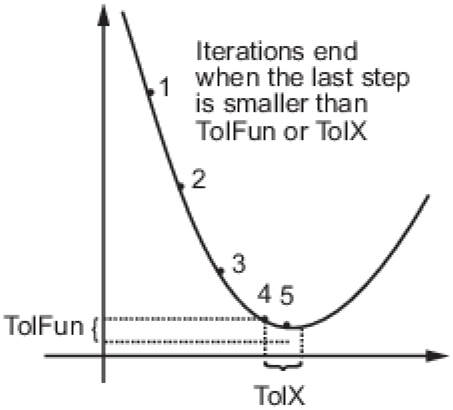
Option	Description	Solvers
MaxIter	The maximum number of iterations allowed. The default value is 500 for fminbnd and 200*length(x0) for fminsearch.	fminbnd, fminsearch
OutputFcn	Display information on the iterations of the solver. The default is [] (none). See "Optimization Solver Output Functions" on page 9-14.	fminbnd, fminsearch, fzero
PlotFcns	Plot information on the iterations of the solver. The default is [] (none). For available predefined functions, see "Optimization Solver Plot Functions" on page 9-20.	fminbnd, fminsearch, fzero
TolFun	The termination tolerance for the function value. The default value is 1.e-4. See "Tolerances and Stopping Criteria" on page 9-11.	fminsearch
TolX	The termination tolerance for *x*. The default value is 1.e-4, except for fzero, which has a default value of eps (= 2.2204e-16), and lsqnonneg, which has a default value of 10*eps*norm(c,1)*length(c). See "Tolerances and Stopping Criteria" on page 9-11.	fminbnd, fminsearch, fzero, lsqnonneg

Tolerances and Stopping Criteria

The number of iterations in an optimization depends on the stopping criteria for the solver. These criteria include several tolerances you can set. Generally, a tolerance is a threshold which, if crossed, stops the iterations of a solver.

Tip Generally, set the TolFun and TolX tolerances to well above eps, and usually above 1e-14. Setting small tolerances does not guarantee accurate results. Instead, a solver can fail to recognize when it has converged, and can continue futile iterations. A tolerance value smaller than eps effectively disables that stopping condition. This tip does not apply to fzero, which uses a default value of eps for TolX.

- TolX is a lower bound on the size of a step, meaning the norm of (x_i – x_{i+1}). If the solver attempts to take a step that is smaller than TolX, the iterations end. Solvers generally use TolX as a *relative* bound, meaning iterations end when $|(x_i – x_{i+1})|$ < TolX*(1 + $|x_i|$), or a similar relative measure.


Iterations end when the last step is smaller than TolFun or TolX
1
2
3
4 5
TolFun
TolX

- `TolFun` is a lower bound on the change in the value of the objective function during a step. If |$f(x_i) - f(x_{i+1})$| < `TolFun`, the iterations end. Solvers generally use `TolFun` as a *relative* bound, meaning iterations end when |$f(x_i) - f(x_{i+1})$| < `TolFun`(1 + |$f(x_i)$|), or a similar relative measure.
- `MaxIter` is a bound on the number of solver iterations. `MaxFunEvals` is a bound on the number of function evaluations.

Note Unlike other solvers, `fminsearch` stops when it satisfies *both* `TolFun` and `TolX`.

Output Structure

The `output` structure includes the number of function evaluations, the number of iterations, and the algorithm. The structure appears when you provide `fminbnd`, `fminsearch`, or `fzero` with a fourth output argument, as in

```
[x,fval,exitflag,output] = fminbnd(@humps,0.3,1);
```

The details of the `output` structure for each solver are on the function reference pages.

The `output` structure is not an option that you choose with `optimset`. It is an optional output for `fminbnd`, `fminsearch`, and `fzero`.

See Also

More About

- “Optimizing Nonlinear Functions” on page 9-2
- “Optimization Solver Output Functions” on page 9-14
- “Optimization Solver Plot Functions” on page 9-20

Optimization Solver Iterative Display

You obtain details of the steps solvers take by setting the `Display` option to `'iter'` with `optimset`. The displayed output contains headings and items from the following list.

Heading	Information Displayed	Solvers
`Iteration`	Iteration number, meaning the number of steps the algorithm has taken	`fminsearch`
`Func-count`	Cumulative number of function evaluations	`fminbnd`, `fminsearch`, `fzero`
`x`	Current point	`fminbnd`, `fzero`
`f(x)`	Current objective function value	`fminbnd`, `fzero`
`min f(x)`	Smallest objective function value found	`fminsearch`
`Procedure`	Algorithm used during the iteration	
	• `initial` • `golden` (golden section search) • `parabolic` (parabolic interpolation)	`fminbnd`
	• `initial simplex` • `expand` • `reflect` • `shrink` • `contract inside` • `contract outside` For details, see “fminsearch Algorithm” on page 9-4.	`fminsearch`
	• `initial` (initial point) • `search` (search for an interval containing a zero) • `bisection` • `interpolation` (linear interpolation or inverse quadratic interpolation)	`fzero`
`a`, `f(a)`, `b`, `f(b)`	Search points and their function values while looking for an interval with function values of opposite signs	`fzero`

See Also

More About

- “Set Optimization Options” on page 9-10
- “Optimization Solver Output Functions” on page 9-14
- “Optimization Solver Plot Functions” on page 9-20

Optimization Solver Output Functions

In this section...

What Is an Output Function?

An output function is a function that an optimization function calls at each iteration of its algorithm. Typically, you use an output function to generate graphical output, record the history of the data the algorithm generates, or halt the algorithm based on the data at the current iteration. You can create an output function as a function file, a local function, or a nested function.

You can use the `OutputFcn` option with the following MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`

Creating and Using an Output Function

The following is a simple example of an output function that plots the points generated by an optimization function.

```
function stop = outfun(x, optimValues, state)
stop = false;
hold on;
plot(x(1),x(2),'.');
drawnow
```

You can use this output function to plot the points generated by `fminsearch` in solving the optimization problem

$$\min_x f(x) = \min_x e^{x_1}\left(4x_1^2 + 2x_2^2 + x_1x_2 + 2x_2\right).$$

To do so,

1. Create a file containing the preceding code and save it as `outfun.m` in a folder on the MATLAB path.
2. Set the value of the `Outputfcn` field of the `options` structure to a function handle to `outfun`.

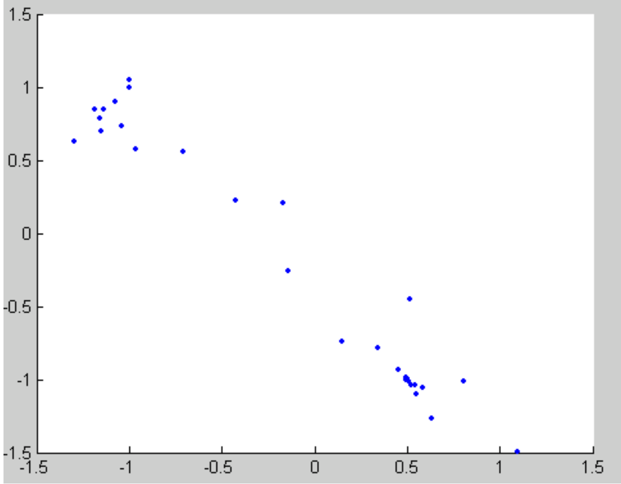
   ```
   options = optimset('OutputFcn', @outfun);
   ```
3. Enter the following commands:

```
hold on
objfun=@(x) exp(x(1))*(4*x(1)^2+2*x(2)^2+x(1)*x(2)+2*x(2));
[x fval] = fminsearch(objfun, [-1 1], options)
hold off
```

These commands return the solution

```
x =
    0.1290   -0.5323

fval =
   -0.5689
```

and display the following plot of the points generated by `fminsearch`:



Structure of the Output Function

The function definition line of the output function has the following form:

```
stop = outfun(x, optimValues, state)
```

where

- `stop` is a flag that is `true` or `false` depending on whether the optimization routine halts or continues. See “Stop Flag” on page 9-18.
- `x` is the point computed by the algorithm at the current iteration.
- `optimValues` is a structure containing data from the current iteration. “Fields in optimValues” on page 9-17 describes the structure in detail.
- `state` is the current state of the algorithm. “States of the Algorithm” on page 9-17 lists the possible values.

The optimization function passes the values of the input arguments to `outfun` at each iteration.

Example of a Nested Output Function

The example in “Creating and Using an Output Function” on page 9-14 does not require the output function to preserve data from one iteration to the next. When you do not need to save data between iterations, you can write the output function as a function file and call the optimization function directly from the command line. However, to have an output function to record data from one iteration to the next, write a single file that does the following:

- Contains the output function as a nested function—see “Nested Functions” in MATLAB Programming Fundamentals for more information.
- Calls the optimization function.

In the following example, the function file also contains the objective function as a local function. You can instead write the objective function as a separate file or as an anonymous function.

Nested functions have access to variables in the surrounding file. Therefore, this method enables the output function to preserve variables from one iteration to the next.

The following example uses an output function to record the `fminsearch` iterates in solving

$$\min_x f(x) = \min_x e^{x_1}\left(4x_1^2 + 2x_2^2 + x_1x_2 + 2x_2\right).$$

The output function returns the sequence of points as a matrix called `history`.

To run the example, do the following steps:

1. Open a new file in the MATLAB Editor.
2. Copy and paste the following code into the file.

```
function [x fval history] = myproblem(x0)
    history = [];
    options = optimset('OutputFcn', @myoutput);
    [x fval] = fminsearch(@objfun, x0,options);

    function stop = myoutput(x,optimvalues,state);
        stop = false;
        if isequal(state,'iter')
          history = [history; x];
        end
    end

    function z = objfun(x)
      z = exp(x(1))*(4*x(1)^2+2*x(2)^2+x(1)*x(2)+2*x(2));
    end
end
```

3. Save the file as `myproblem.m` in a folder on the MATLAB path.
4. At the MATLAB prompt, enter

```
[x fval history] = myproblem([-1 1]);
```

The function `fminsearch` returns `x`, the optimal point, and `fval`, the value of the objective function at x.

```
x,fval

x =
    0.1290   -0.5323

fval =
   -0.5689
```

In addition, the output function `myoutput` returns the matrix `history`, which contains the points generated by the algorithm at each iteration, to the MATLAB workspace. The first four rows of `history` are

```
history(1:4,:)

ans =

   -1.0000    1.0000
   -1.0000    1.0000
   -1.0750    0.9000
   -1.0125    0.8500
```

The final row of points in `history` is the same as the optimal point, `x`.

```
history(end,:)

ans =

    0.1290   -0.5323

objfun(history(end,:))

ans =

   -0.5689
```

Fields in optimValues

The following table lists the fields of the `optimValues` structure that are provided by the optimization functions `fminbnd`, `fminsearch`, and `fzero`.

The "Command-Line Display Headings" column of the table lists the headings that appear when you set the `Display` parameter of `options` to `'iter'`.

optimValues Field (optimValues.field)	Description	Command-Line Display Heading
`funccount`	Cumulative number of function evaluations	`Func-count`
`fval`	Function value at current point	`min f(x)`
`iteration`	Iteration number — starts at `0`	`Iteration`
`procedure`	Procedure messages	`Procedure`

States of the Algorithm

The following table lists the possible values for `state`:

State	Description
`'init'`	The algorithm is in the initial state before the first iteration.

State	Description
'interrupt'	The algorithm is performing an iteration. In this state, the output function can halt the current iteration of the optimization. You might want the output function to halt the iteration to improve the efficiency of the computations. When state is set to 'interrupt', the values of x and optimValues are the same as at the last call to the output function, in which state is set to 'iter'.
'iter'	The algorithm is at the end of an iteration.
'done'	The algorithm is in the final state after the last iteration.

The following code illustrates how the output function uses the value of state to decide which tasks to perform at the current iteration.

```
switch state
    case 'init'
          % Setup for plots or dialog boxes
    case 'iter'
          % Make updates to plots or dialog boxes as needed
    case 'interrupt'
          % Check conditions to see whether optimization
          % should quit
    case 'done'
          % Cleanup of plots, dialog boxes, or final plot
end
```

Stop Flag

The output argument stop is a flag that is true or false. The flag tells the optimization function whether the optimization halts (true) or continues (false). The following examples show typical ways to use the stop flag.

Stopping an Optimization Based on Data in optimValues

The output function can stop an optimization at any iteration based on the current data in optimValues. For example, the following code sets stop to true if the objective function value is less than 5:

```
function stop = myoutput(x, optimValues, state)
stop = false;
% Check if objective function is less than 5.
if optimValues.fval < 5
    stop = true;
end
```

Stopping an Optimization Based on Dialog Box Input

If you design a UI to perform optimizations, you can have the output function stop an optimization with, for example, a **Stop** button. The following code shows how to do this callback. The code assumes that the **Stop** button callback stores the value true in the optimstop field of a handles structure called hObject stored in appdata.

```
function stop = myoutput(x, optimValues, state)
stop = false;
```

```
% Check if user has requested to stop the optimization.
stop = getappdata(hObject,'optimstop');
```

See Also

More About

- “Optimization Solver Plot Functions” on page 9-20
- “Set Optimization Options” on page 9-10
- “Optimization Solver Iterative Display” on page 9-13

Optimization Solver Plot Functions

In this section...

What Is a Plot Function?

The `PlotFcns` field of the `options` structure specifies one or more functions that an optimization function calls at each iteration to plot various measures of progress. Pass a function handle or cell array of function handles. The structure of a plot function is the same as the structure of an output function. For more information on this structure, see “Optimization Solver Output Functions” on page 9-14.

You can use the `PlotFcns` option with the following MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`

The predefined plot functions for these optimization functions are:

- `@optimplotx` plots the current point
- `@optimplotfval` plots the function value
- `@optimplotfunccount` plots the function count (not available for `fzero`)

To view or modify a predefined plot function, open the function file in the MATLAB Editor. For example, to view the function file for plotting the current point, enter:

```
edit optimplotx.m
```

Example: Plot Function

View the progress of a minimization using `fminsearch` with the plot function `@optimplotfval`:

1 Write a file for the objective function. For this example, use:

```
function f = onehump(x)

r = x(1)^2 + x(2)^2;
s = exp(-r);
f = x(1)*s+r/20;
```

2 Set the options to use the plot function:

```
options = optimset('PlotFcns',@optimplotfval);
```

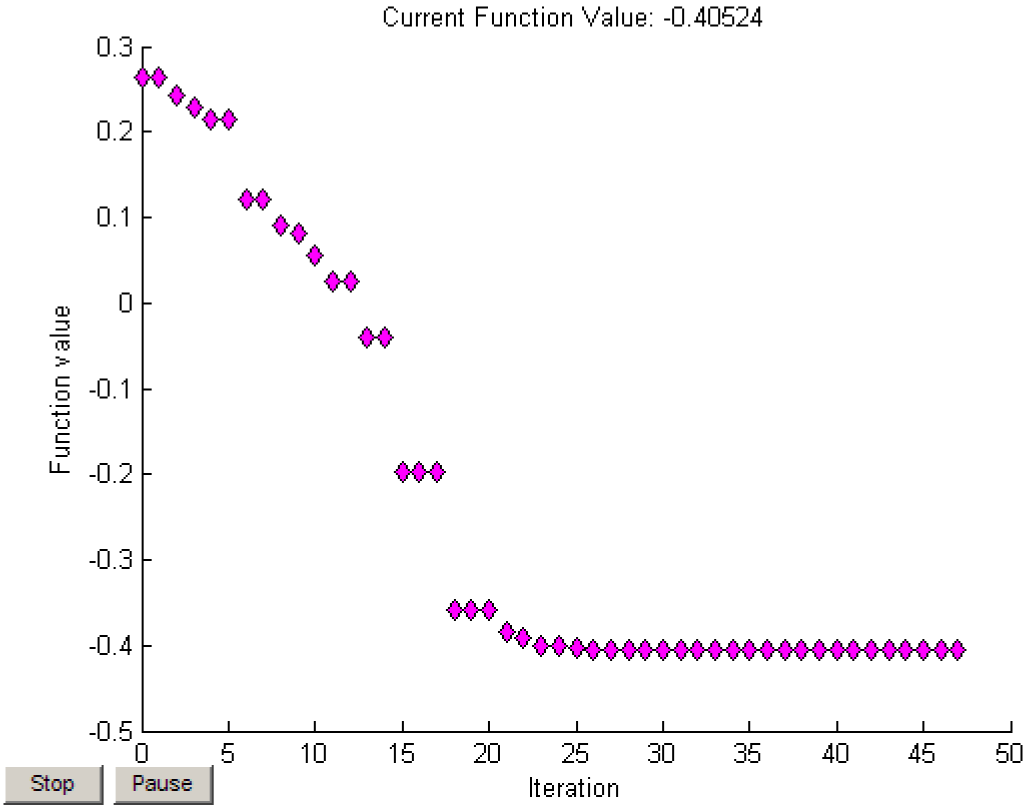
3 Call `fminsearch` starting from [2,1]:

```
[x ffinal] = fminsearch(@onehump,[2,1],options)
```

4 MATLAB returns the following:

```
x =
   -0.6691    0.0000
```

```
ffinal =
   -0.4052
```



See Also

More About

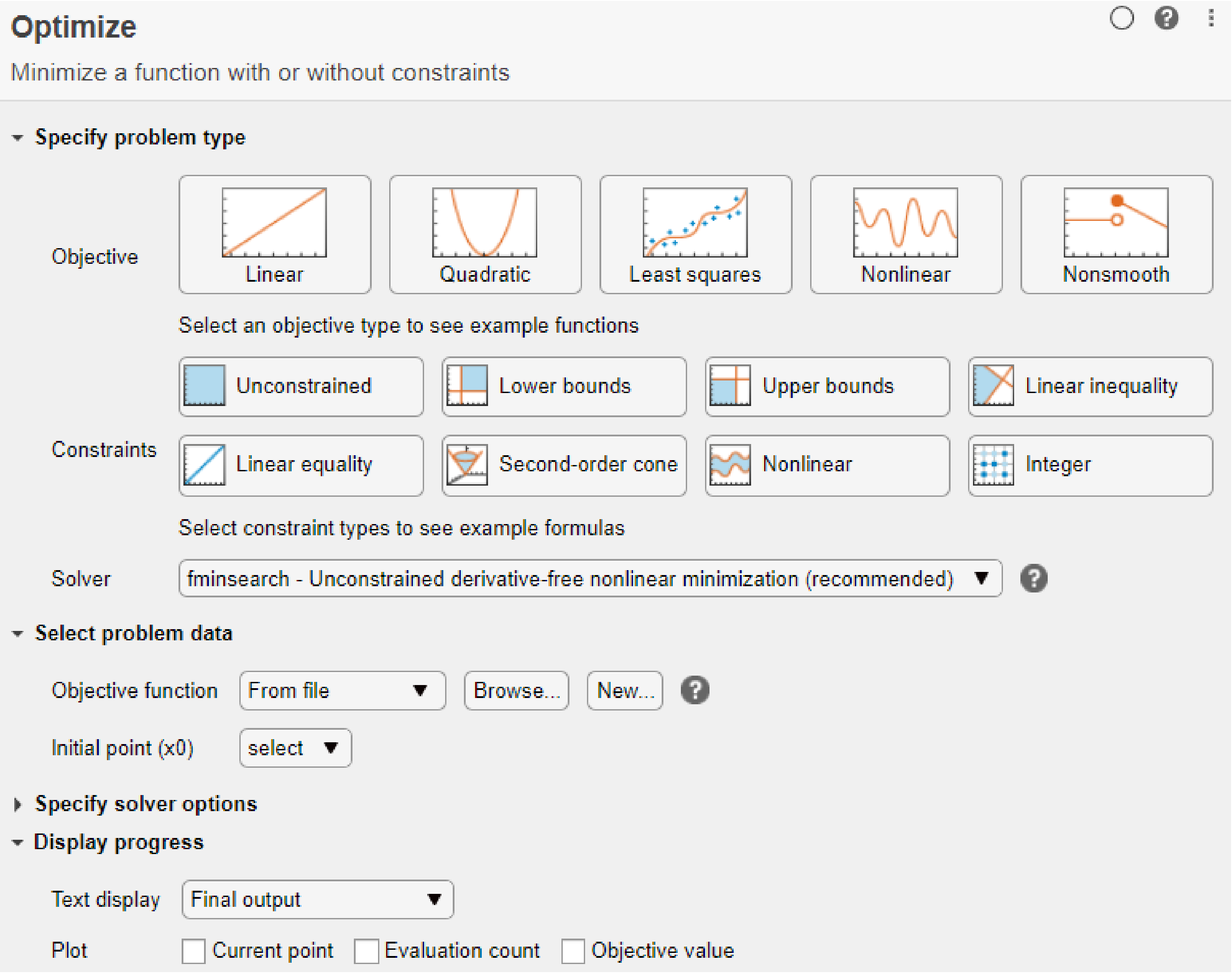
- “Optimization Solver Output Functions” on page 9-14
- “Set Optimization Options” on page 9-10
- “Optimization Solver Iterative Display” on page 9-13

Optimize Live Editor Task

In this section...

What Is the Optimize Live Editor Task?

The **Optimize** Live Editor task provides a visual interface for the `fminbnd`, `fminsearch`, `fzero`, and `lsqnonneg` solvers. To start the task, click the **New Live Script** button. Then click the **Insert** tab and select **Task > Optimize**.



Optimize

Minimize a function with or without constraints

▾ Specify problem type

Objective

Select an objective type to see example functions

Constraints

Select constraint types to see example formulas

Solver fminsearch - Unconstrained derivative-free nonlinear minimization (recommended) ▼

▾ Select problem data

Objective function From file ▼ Browse... New...

Initial point (x0) select ▼

▸ Specify solver options

▾ Display progress

Text display Final output ▼

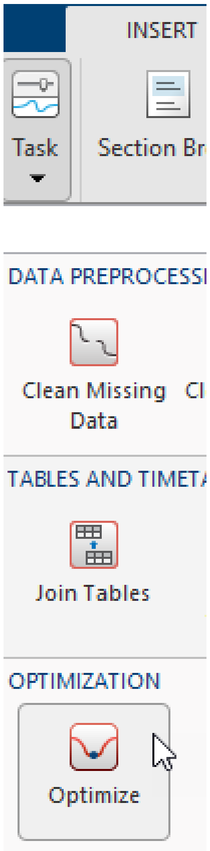
Plot □ Current point □ Evaluation count □ Objective value

Minimize a Nonlinear Function of Several Variables

This example shows how to minimize the function $f(x,y) = 100\left(y - x^2\right)^2 + (a - x)^2$ where the variable $a = \pi$ using the **Optimize** Live Editor task.

For a video describing a similar optimization problem, see How to Use the Optimize Live Editor Task.

1. On the **Home** tab, in the **File** section, click the **New Live Script** button.
2. Insert an **Optimize** Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.



3. For use in entering problem data, click the **Section Break** button. New sections appear above and below the task.
4. In the section above the **Optimize** task, enter the following code.

```
a = pi;
x0 = [-1 2];
```

5. To place these variables into the workspace, press **Ctrl + Enter**.
6. In the **Specify problem type** section of the task, click the **Objective > Nonlinear** button and the **Constraints > Unconstrained** button. The task shows that the recommended solver is `fminsearch`.

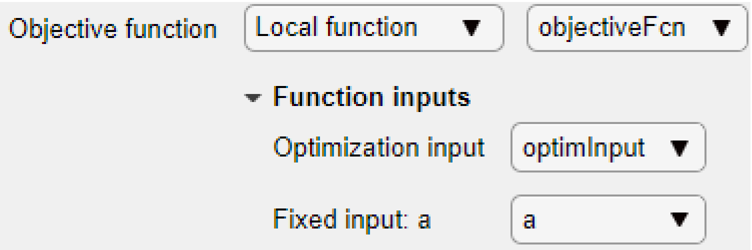
Note If you have Optimization Toolbox™, your recommended solver at this point is different. Choose `fminsearch` to proceed with the example.

7 In the **Select problem data** section, select **Objective function > Local function** and then click the **New** button. A function script appears in a new section below the task. Edit the resulting code to contain the following uncommented lines.

```
function f = objectiveFcn(optimInput,a)
x = optimInput(1);
y = optimInput(2);
f = 100*(y - x^2)^2 + (a - x)^2;
end
```

8 In the **Select problem data** section, select `objectiveFcn` as the local function.

9 In the **Select problem data** section, under **Function inputs**, select **Optimization input > optimInput** and **Fixed input: a > a**.



Objective function Local function ▼ objectiveFcn ▼

Function inputs

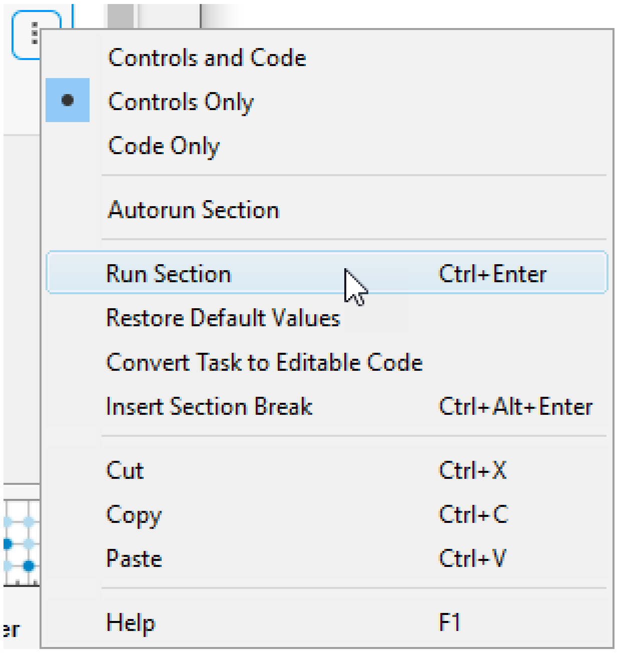
Optimization input optimInput ▼

Fixed input: a a ▼

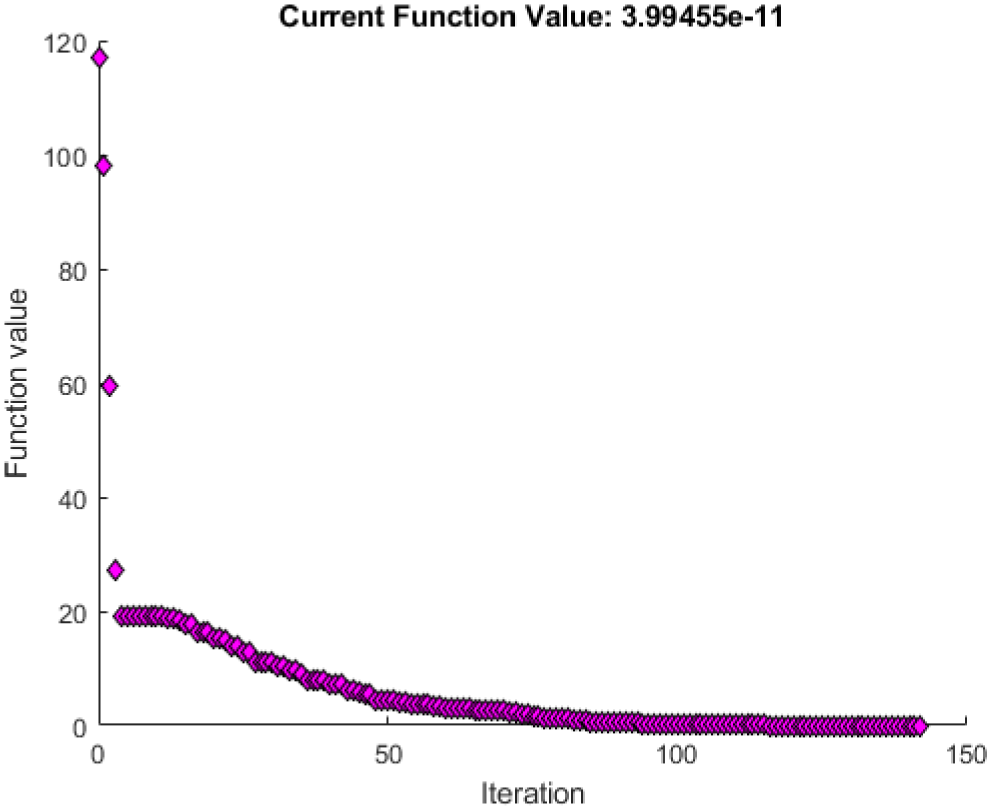
10 Select **Initial point (x0) > x0**.

11 In the **Display progress** section, select **Objective value** for the plot.

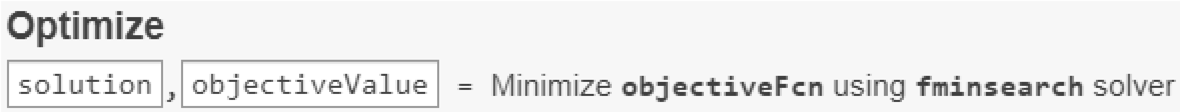
12 To run the solver, click the options button ⋮ at the top right of the task window, and select **Run Section**.



The following plot appears.



13 To view the solution point, look at the top of the **Optimize** task.



Optimize

solution , objectiveValue = Minimize **objectiveFcn** using **fminsearch** solver

The `solution` and `objectiveValue` variables are returned to the workspace. To view their values, insert a section break below the task and enter this code.

```
disp(solution)
disp(objectiveValue)
```

14 Run the section by pressing **Ctrl+Enter**.

```
disp(solution)

    3.1416    9.8696

disp(objectiveValue)

  3.9946e-11
```

Solve a Scalar Equation

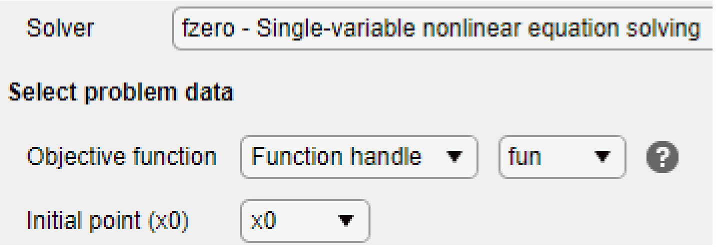
This example shows how to use the **Optimize** Live Editor task to find the point *x* where cos(*x*) = *x*.

1 On the **Home** tab, in the **File** section, click the **New Live Script** button. Enter these lines of code in the live script.

```
fun = @(x)cos(x) - x;
x0 = 0;
```

The first line defines the anonymous function `fun`, which takes the value 0 at the point *x* where cos(*x*) = *x*. The second line defines the initial point `x0` = 0, where `fzero` begins its search for a solution.

2. Put these variables in the MATLAB workspace by pressing **Ctrl+Enter**.
3. Insert an **Optimize** Live Editor task. Click the **Insert** tab and then, in the **Code** section, select **Task > Optimize**.
4. In the **Specify problem type** section of the task, select **Solver > fzero**.
5. In the **Select problem data** section, select **Objective function > Function handle** and then select `fun`. Select **Initial point (x0) > x0**.



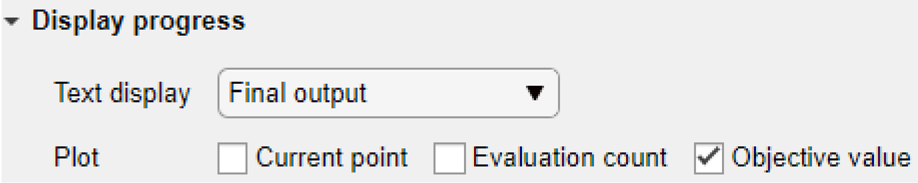
Solver fzero - Single-variable nonlinear equation solving

Select problem data

Objective function Function handle ▼ fun ▼

Initial point (x0) x0 ▼

6. In the Display progress section, select **Objective value** for the plot.

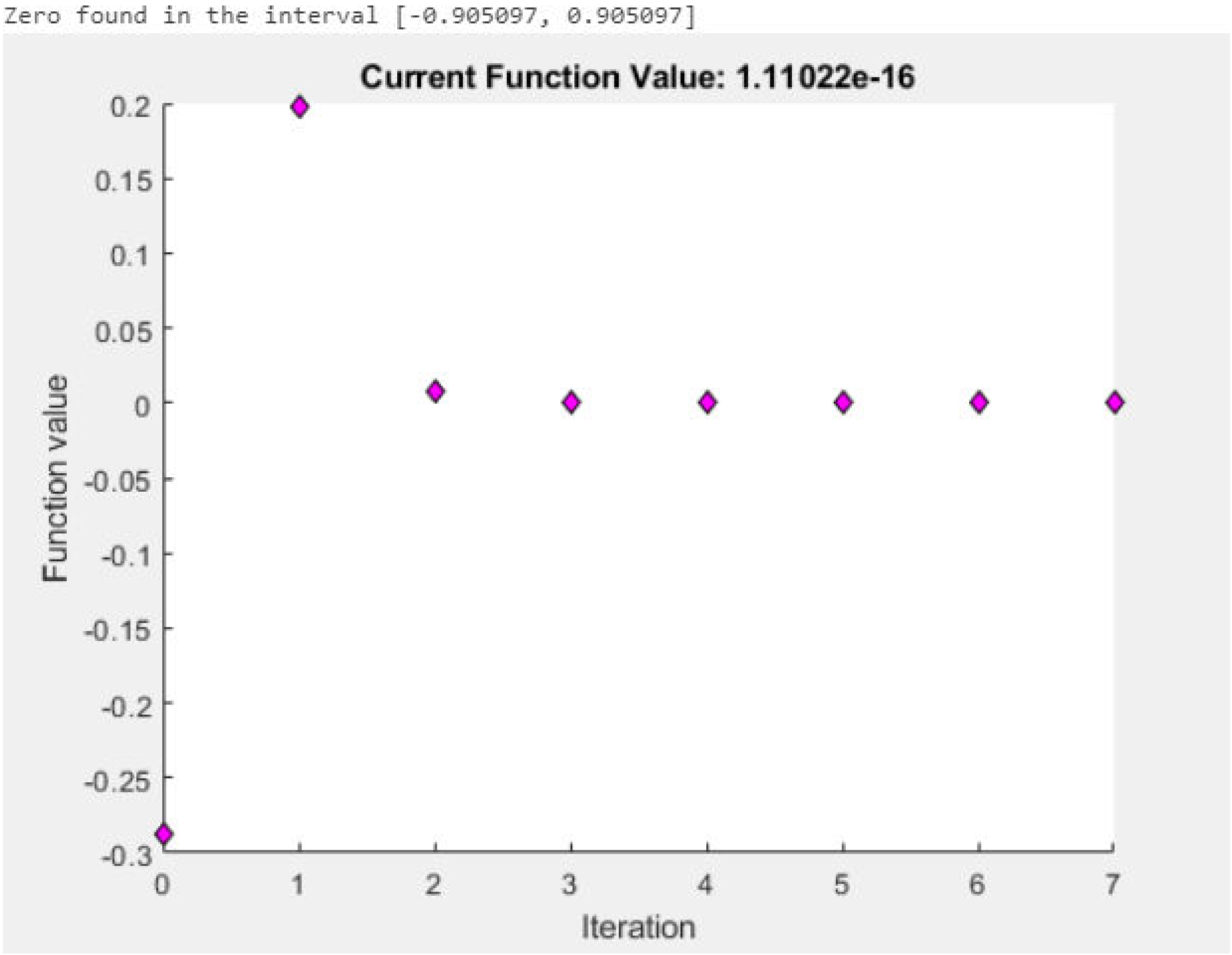


Display progress

Text display Final output ▼

Plot □ Current point □ Evaluation count ☑ Objective value

7. Run the solver by pressing **Ctrl+Enter**.



```
Zero found in the interval [-0.905097, 0.905097]
```

8 To see the solution value, insert a new section below the task by clicking the **Section Break** button on the **Insert** tab. In the new section, enter `solution` and press **Ctrl+Enter**.

```
solution
```

```
solution = 0.7391
```

See Also

Optimize | `fminsearch` | `fzero`

More About

- “Optimization”
- “Add Interactive Tasks to a Live Script”
- How to Use the Optimize Live Editor Task

Optimization Troubleshooting and Tips

This table describes typical optimization problems and provides recommendations for handling them.

Problem	Recommendation
The solution found by `fminbnd` or `fminsearch` is not a global minimum. A global minimum has the smallest objective function value among all points in the search space.	There is no guarantee that a solution is a global minimum unless your problem is continuous and has only one minimum. To search for a global minimum, start the optimization from multiple starting points (or intervals, in the case of `fminbnd`).
It is impossible to evaluate the objective function `f(x)` at some points `x`. Such points are called infeasible.	Modify your function to return a large positive value for `f(x)` at infeasible points `x`.
The minimization routine appears to enter an infinite loop or returns a solution that is not a minimum (or not a zero, in the case of `fzero`).	Perhaps your objective function returns `NaN` or complex values. Solvers expect only real objective function values. Any other values can cause unexpected results. To determine whether there are `NaN` or complex values, set `options = optimset('FunValCheck','on')` and call the optimization function with `options` as an input argument. If an objective function returns a `NaN` or complex value, this setting causes the solver to throw an error.
The solver takes a long time.	Most optimization problems benefit from good starting points. Try random starting points in a region that might be close to a solution, based on your problem characteristics. Sometimes you can solve a complicated problem using an evolutionary approach. First, solve problems with a smaller number of independent variables. Use solutions from these simpler problems as starting points for more complicated problems by using an appropriate mapping. Also, you can sometimes speed the solution by using simpler objective functions and less stringent stopping conditions in the initial stages of an optimization problem.
It is unclear what the solver is doing.	To see what the solver is doing as it iterates: • Monitor the iterations using a plot function such as `@optimplotfval`. Use `optimset` to set the `PlotFcns` option. See “Optimization Solver Plot Functions” on page 9-20. • Set the `Display` option to display information at the command line. See “Optimization Solver Iterative Display” on page 9-13.

Problem	Recommendation
`fminsearch` fails to reach a solution.	`fminsearch` can fail to reach a solution for various reasons. • Poor scaling. If your problem is not adequately centered and scaled, the solver can fail to converge correctly. Try to have each coordinate give roughly the same effect on the objective function, and ensure that the scale of each coordinate near a possible solution is not too large or small. To do so, edit the objective function and add or multiply appropriate constants for each coordinate. • Inappropriate stopping criteria. If the value you specify for `TolFun` or `TolX` is too small, `fminsearch` can fail to realize when it has reached a solution. If the value is too large, `fminsearch` can halt far from a solution. • Poor initial point. Try starting `fminsearch` from various initial points. • Insufficient iterations. If the solver runs out of iterations or gets stuck, try restarting `fminsearch` from its final point to reach a better solution. Sometimes `fminsearch` reaches a better solution when you increase the value of the `MaxFunEvals` and `MaxIter` options from their default of `200*length(x0)`.

See Also

More About

- “Optimizing Nonlinear Functions” on page 9-2

10

Function Handles

Parameterizing Functions

In this section...

Overview

This topic explains how to store or access extra parameters for mathematical functions that you pass to MATLAB function functions, such as `fzero` or `integral`.

MATLAB function functions evaluate mathematical expressions over a range of values. They are called function functions because they are functions that accept a function handle (a pointer to a function) as an input. Each of these functions expects that your objective function has a specific number of input variables. For example, `fzero` and `integral` accept handles to functions that have exactly one input variable.

Suppose you want to find the zero of the cubic polynomial $x^3 + bx + c$ for different values of the coefficients *b* and *c*. Although you could create a function that accepts three input variables (*x*, *b*, and *c*), you cannot pass a function handle that requires all three of those inputs to `fzero`. However, you can take advantage of properties of anonymous or nested functions to define values for additional inputs.

Parameterizing Using Nested Functions

One approach for defining parameters is to use a nested function—a function completely contained within another function in a program file. For this example, create a file named `findzero.m` that contains a parent function `findzero` and a nested function `poly`:

```
function y = findzero(b,c,x0)

y = fzero(@poly,x0);

   function y = poly(x)
   y = x^3 + b*x + c;
   end
end
```

The nested function defines the cubic polynomial with one input variable, `x`. The parent function accepts the parameters `b` and `c` as input values. The reason to nest `poly` within `findzero` is that nested functions share the workspace of their parent functions. Therefore, the `poly` function can access the values of `b` and `c` that you pass to `findzero`.

To find a zero of the polynomial with `b = 2` and `c = 3.5`, using the starting point `x0 = 0`, you can call `findzero` from the command line:

```
x = findzero(2,3.5,0)
```

```
x =
   -1.0945
```

Parameterizing Using Anonymous Functions

Another approach for accessing extra parameters is to use an anonymous function. Anonymous functions are functions that you can define in a single command, without creating a separate program file. They can use any variables that are available in the current workspace.

For example, create a handle to an anonymous function that describes the cubic polynomial, and find the zero:

```
b = 2;
c = 3.5;
cubicpoly = @(x) x^3 + b*x + c;
x = fzero(cubicpoly,0)
```

```
x =
   -1.0945
```

Variable `cubicpoly` is a function handle for an anonymous function that has one input, `x`. Inputs for anonymous functions appear in parentheses immediately following the `@` symbol that creates the function handle. Because `b` and `c` are in the workspace when you create `cubicpoly`, the anonymous function does not require inputs for those coefficients.

You do not need to create an intermediate variable, `cubicpoly`, for the anonymous function. Instead, you can include the entire definition of the function handle within the call to `fzero`:

```
b = 2;
c = 3.5;
x = fzero(@(x) x^3 + b*x + c,0)
```

```
x =
   -1.0945
```

You also can use anonymous functions to call more complicated objective functions that you define in a function file. For example, suppose you have a file named `cubicpoly.m` with this function definition:

```
function y = cubicpoly(x,b,c)
y = x^3 + b*x + c;
end
```

At the command line, define `b` and `c`, and then call `fzero` with an anonymous function that invokes `cubicpoly`:

```
b = 2;
c = 3.5;
x = fzero(@(x) cubicpoly(x,b,c),0)
```

```
x =
   -1.0945
```

Note To change the values of the parameters, you must create a new anonymous function. For example:

```
b = 10;
c = 25;
x = fzero(@(x) x^3 + b*x + c,0);
```

See Also

More About

- “Create Function Handle”
- “Nested Functions”
- “Anonymous Functions”

Ordinary Differential Equations (ODEs)

Choose an ODE Solver

Ordinary Differential Equations

An *ordinary differential equation* (ODE) contains one or more derivatives of a dependent variable, y, with respect to a single independent variable, t, usually referred to as time. The notation used here for representing derivatives of y with respect to t is y' for a first derivative, y'' for a second derivative, and so on. The *order* of the ODE is equal to the highest-order derivative of y that appears in the equation.

For example, this is a second order ODE:

$$y'' = 9y$$

In an *initial value problem*, the ODE is solved by starting from an initial state. Using the initial condition, y_0, as well as a period of time over which the answer is to be obtained, $\left(t_0, t_f\right)$, the solution is obtained iteratively. At each step the solver applies a particular algorithm to the results of previous steps. At the first such step, the initial condition provides the necessary information that allows the integration to proceed. The final result is that the ODE solver returns a vector of time steps $t = \left[t_0, t_1, t_2, ..., t_f\right]$ as well as the corresponding solution at each step $y = \left[y_0, y_1, y_2, ..., y_f\right]$.

Types of ODEs

The ODE solvers in MATLAB solve these types of first-order ODEs:

- Explicit ODEs of the form $y' = f(t, y)$.
- Linearly implicit ODEs of the form $M(t, y)y' = f(t, y)$, where $M(t, y)$ is a nonsingular mass matrix. The mass matrix can be time- or state-dependent, or it can be a constant matrix. Linearly implicit ODEs involve linear combinations of the first derivative of y, which are encoded in the mass matrix.

  Linearly implicit ODEs can always be transformed to an explicit form, $y' = M^{-1}(t, y)f(t, y)$. However, specifying the mass matrix directly to the ODE solver avoids this transformation, which is inconvenient and can be computationally expensive.
- If some components of y' are missing, then the equations are called *differential algebraic equations*, or DAEs, and the system of DAEs contains some *algebraic variables*. Algebraic variables are dependent variables whose derivatives do not appear in the equations. A system of DAEs can be rewritten as an equivalent system of first-order ODEs by taking derivatives of the equations to eliminate the algebraic variables. The number of derivatives needed to rewrite a DAE as an ODE is called the differential index. The `ode15s` and `ode23t` solvers can solve index-1 DAEs.
- Fully implicit ODEs of the form $f(t, y, y') = 0$. Fully implicit ODEs cannot be rewritten in an explicit form, and might also contain some algebraic variables. The `ode15i` solver is designed for fully implicit problems, including index-1 DAEs.

You can supply additional information to the solver for some types of problems by using the `odeset` function to create an options structure.

Systems of ODEs

You can specify any number of coupled ODE equations to solve, and in principle the number of equations is only limited by available computer memory. If the system of equations has *n* equations,

$$\begin{pmatrix} y'_1 \\ y'_2 \\ \vdots \\ y'_n \end{pmatrix} = \begin{pmatrix} f_1(t, y_1, y_2, ..., y_n) \\ f_2(t, y_1, y_2, ..., y_n) \\ \vdots \\ f_n(t, y_1, y_2, ..., y_n) \end{pmatrix},$$

then the function that encodes the equations returns a vector with *n* elements, corresponding to the values for $y'_1,\ y'_2,\ \ldots,\ y'_n$. For example, consider the system of two equations

$$\begin{cases} y'_1 = y_2 \\ y'_2 = y_1 y_2 - 2\ . \end{cases}$$

A function that encodes these equations is

```
function dy = myODE(t,y)
dy(1) = y(2);
dy(2) = y(1)*y(2)-2;
```

Higher-Order ODEs

The MATLAB ODE solvers only solve first-order equations. You must rewrite higher-order ODEs as an equivalent system of first-order equations using the generic substitutions

$$\begin{aligned} y_1 &= y \\ y_2 &= y' \\ y_3 &= y'' \\ &\vdots \\ y_n &= y^{(n-1)}. \end{aligned}$$

The result of these substitutions is a system of *n* first-order equations

$$\begin{cases} y'_1 = y_2 \\ y'_2 = y_3 \\ \quad\vdots \\ y'_n = f(t, y_1, y_2, ..., y_n). \end{cases}$$

For example, consider the third-order ODE

$$y''' - y''y + 1 = 0.$$

Using the substitutions

$$\begin{aligned} y_1 &= y \\ y_2 &= y' \\ y_3 &= y'' \end{aligned}$$

results in the equivalent first-order system

$$\begin{cases} y'_1 = y_2 \\ y'_2 = y_3 \\ y'_3 = y_1\, y_3 - 1. \end{cases}$$

The code for this system of equations is then

```
function dydt = f(t,y)
dydt(1) = y(2);
dydt(2) = y(3);
dydt(3) = y(1)*y(3)-1;
```

Complex ODEs

Consider the complex ODE equation

$$y' = f(t, y)\,,$$

where $y = y_1 + iy_2$. To solve it, separate the real and imaginary parts into different solution components, then recombine the results at the end. Conceptually, this looks like

$$yv = [\mathrm{Real}(y) \quad \mathrm{Imag}(y)]$$
$$fv = [\mathrm{Real}(f(t, y)) \quad \mathrm{Imag}(f(t, y))]\ .$$

For example, if the ODE is $y' = yt + 2i$, then you can represent the equation using a function file.

```
function f = complexf(t,y)
% Define function that takes and returns complex values
f = y.*t + 2*i;
```

Then, the code to separate the real and imaginary parts is

```
function fv = imaginaryODE(t,yv)
% Construct y from the real and imaginary components
y = yv(1) + i*yv(2);

% Evaluate the function
yp = complexf(t,y);

% Return real and imaginary in separate components
fv = [real(yp); imag(yp)];
```

When you run a solver to obtain the solution, the initial condition `y0` is also separated into real and imaginary parts to provide an initial condition for each solution component.

```
y0 = 1+i;
yv0 = [real(y0); imag(y0)];
tspan = [0 2];
[t,yv] = ode45(@imaginaryODE, tspan, yv0);
```

Once you obtain the solution, combine the real and imaginary components together to obtain the final result.

```
y = yv(:,1) + i*yv(:,2);
```

Basic Solver Selection

ode45 performs well with most ODE problems and should generally be your first choice of solver. However, ode23 and ode113 can be more efficient than ode45 for problems with looser or tighter accuracy requirements.

Some ODE problems exhibit *stiffness*, or difficulty in evaluation. Stiffness is a term that defies a precise definition, but in general, stiffness occurs when there is a difference in scaling somewhere in the problem. For example, if an ODE has two solution components that vary on drastically different time scales, then the equation might be stiff. You can identify a problem as stiff if nonstiff solvers (such as ode45) are unable to solve the problem or are extremely slow. If you observe that a nonstiff solver is very slow, try using a stiff solver such as ode15s instead. When using a stiff solver, you can improve reliability and efficiency by supplying the Jacobian matrix or its sparsity pattern.

This table provides general guidelines on when to use each of the different solvers.

Solver	Problem Type	Accuracy	When to Use
ode45	Nonstiff	Medium	Most of the time. ode45 should be the first solver you try.
ode23		Low	ode23 can be more efficient than ode45 at problems with crude tolerances, or in the presence of moderate stiffness.
ode113		Low to High	ode113 can be more efficient than ode45 at problems with stringent error tolerances, or when the ODE function is expensive to evaluate.
ode15s	Stiff	Low to Medium	Try ode15s when ode45 fails or is inefficient and you suspect that the problem is stiff. Also use ode15s when solving differential algebraic equations (DAEs).

Solver	Problem Type	Accuracy	When to Use
`ode23s`		Low	`ode23s` can be more efficient than `ode15s` at problems with crude error tolerances. It can solve some stiff problems for which `ode15s` is not effective. `ode23s` computes the Jacobian in each step, so it is beneficial to provide the Jacobian via `odeset` to maximize efficiency and accuracy. If there is a mass matrix, it must be constant.
`ode23t`		Low	Use `ode23t` if the problem is only moderately stiff and you need a solution without numerical damping. `ode23t` can solve differential algebraic equations (DAEs).
`ode23tb`		Low	Like `ode23s`, the `ode23tb` solver might be more efficient than `ode15s` at problems with crude error tolerances.
`ode15i`	Fully implicit	Low	Use `ode15i` for fully implicit problems *f(t,y,y') = 0* and for differential algebraic equations (DAEs) of index 1.

For details and further recommendations about when to use each solver, see [5].

Summary of ODE Examples and Files

There are several example files available that serve as excellent starting points for most ODE problems. To run the **Differential Equations Examples** app, which lets you easily explore and run examples, type

```
odeexamples
```

To open an individual example file for editing, type

```
edit exampleFileName.m
```

To run an example, type

```
exampleFileName
```

This table contains a list of the available ODE and DAE example files as well as the solvers and options they use. Links are included for the subset of examples that are also published directly in the documentation.

Example File	Solver Used	Options Specified	Description	Documentation Link
`amp1dae`	`ode23t`	• `'Mass'`	Stiff DAE — electrical circuit with constant, singular mass matrix	"Solve Stiff Transistor Differential Algebraic Equation" on page 11-74
`ballode`	`ode23`	• `'Events'` • `'OutputFcn'` • `'OutputSel'` • `'Refine'` • `'InitialStep'` • `'MaxStep'`	Simple event location — bouncing ball	"ODE Event Location" on page 11-12
`batonode`	`ode45`	• `'Mass'`	ODE with time- and state-dependent mass matrix — motion of a baton	"Solve Equations of Motion for Baton Thrown into Air" on page 11-55
`brussode`	`ode15s`	• `'JPattern'` • `'Vectorized'`	Stiff large problem — diffusion in a chemical reaction (the Brusselator)	"Solve Stiff ODEs" on page 11-22
`burgersode`	`ode15s`	• `'Mass'` • `'MStateDependence'` • `'JPattern'` • `'MvPattern'` • `'RelTol'` • `'AbsTol'`	ODE with strongly state-dependent mass matrix — Burgers' equation solved using a moving mesh technique	"Solve ODE with Strongly State-Dependent Mass Matrix" on page 11-61
`fem1ode`	`ode15s`	• `'Mass'` • `'MStateDependence'` • `'Jacobian'`	Stiff problem with a time-dependent mass matrix — finite element method	—
`fem2ode`	`ode23s`	• `'Mass'`	Stiff problem with a constant mass matrix — finite element method	—
`hb1ode`	`ode15s`	—	Stiff ODE problem solved on a very long interval — Robertson chemical reaction	—

Example File	Solver Used	Options Specified	Description	Documentation Link
`hb1dae`	`ode15s`	• `'Mass'` • `'RelTol'` • `'AbsTol'` • `'Vectorized'`	Stiff, linearly implicit DAE from a conservation law — Robertson chemical reaction	“Solve Robertson Problem as Semi-Explicit Differential Algebraic Equations (DAEs)”
`ihb1dae`	`ode15i`	• `'RelTol'` • `'AbsTol'` • `'Jacobian'`	Stiff, fully implicit DAE — Robertson chemical reaction	“Solve Robertson Problem as Implicit Differential Algebraic Equations (DAEs)”
`iburgersode`	`ode15i`	• `'RelTol'` • `'AbsTol'` • `'Jacobian'` • `'JPattern'`	Implicit ODE system — Burgers’ equation	—
`kneeode`	`ode15s`	• `'NonNegative'`	The “knee problem” with nonnegativity constraints	“Nonnegative ODE Solution” on page 11-34
`orbitode`	`ode45`	• `'RelTol'` • `'AbsTol'` • `'Events'` • `'OutputFcn'`	Advanced event location — restricted three body problem	“ODE Event Location” on page 11-12
`rigidode`	`ode45`	—	Nonstiff problem — Euler equations of a rigid body without external forces	“Solve Nonstiff ODEs” on page 11-18
`vdpode`	`ode15s`	• `'Jacobian'`	Parameterizable van der Pol equation (stiff for large μ)	“Solve Stiff ODEs” on page 11-22

References

[1] Shampine, L. F. and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.

[2] Forsythe, G., M. Malcolm, and C. Moler, *Computer Methods for Mathematical Computations*, Prentice-Hall, New Jersey, 1977.

[3] Kahaner, D., C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, New Jersey, 1989.

[4] Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[5] Shampine, L. F. and M. W. Reichelt, “The MATLAB ODE Suite,” *SIAM Journal on Scientific Computing*, Vol. 18, 1997, pp. 1–22.

[6] Shampine, L. F., Gladwell, I. and S. Thompson, *Solving ODEs with MATLAB*, Cambridge University Press, Cambridge UK, 2003.

See Also

`odeset` | `odextend`

More About

- “Solve Nonstiff ODEs” on page 11-18
- “Solve Stiff ODEs” on page 11-22
- “Solve Differential Algebraic Equations (DAEs)” on page 11-29

External Websites

- Ordinary Differential Equations

Summary of ODE Options

Solving ODEs frequently requires fine-tuning parameters, adjusting error tolerances, or passing additional information to the solver. This topic shows how to specify options, and which differential equation solvers each option is compatible with.

Options Syntax

Use the `odeset` function to create an options structure that you then pass to the solver as the fourth input argument. For example, to adjust the relative and absolute error tolerances:

```
opts = odeset('RelTol',1e-2,'AbsTol',1e-5);
[t,y] = ode45(@odefun,tspan,y0,opts);
```

If you use the command `odeset` with no inputs, then MATLAB displays a list of the possible values for each option, with default values indicated by curly braces `{}`.

The `odeget` function queries the value of an option in an existing structure, which you can use to dynamically change option values based on conditions. For example, this code detects whether `Stats` is set to `'on'`, and changes the value if necessary:

```
if isempty(odeget(opts,'Stats'))
  odeset(opts,'Stats','on')
end
```

Compatibility of Options with Each Solver

Some options in `odeset` are generic and compatible with any solver, while others are solver-specific. This table summarizes the compatibility of each option with the different solvers.

Option	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb	ode15i
RelTol AbsTol NormControl	✓	✓	✓	✓	✓	✓	✓	✓
OutputFcn OutputSel Refine Stats	✓	✓	✓	✓	✓	✓	✓	✓
NonNegative	✓	✓	✓	✓*	—	✓*	✓*	—
Events	✓	✓	✓	✓	✓	✓	✓	✓**
MaxStep InitialStep	✓	✓	✓	✓	✓	✓	✓	✓

Option	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23tb	ode15i
Jacobian JPattern Vectorized	—	—	—	✓	✓	✓	✓	✓
Mass MStateDependence MvPattern MassSingular	✓ ✓ — —	✓ ✓ — —	✓ ✓ — —	✓ ✓ ✓ ✓	✓ — — —	✓ ✓ ✓ ✓	✓ ✓ ✓ —	— — — —
InitialSlope	—	—	—	✓	—	✓	—	—
MaxOrder BDF	—	—	—	✓	—	—	—	✓ —

* Use the `NonNegative` parameter with `ode15s`, `ode23t`, and `ode23tb` only for those problems in which there is no mass matrix.

** The events function for `ode15i` must accept a third input argument for `yp`.

Usage Examples

MATLAB includes several example files that show how to use various options. For example, type `edit ballode` to see an example that uses `'Events'` to specify an events function, or `edit batonode` to see an example that uses `'Mass'` to specify a mass matrix. For a complete summary of example files and which options they use, see “Summary of ODE Examples and Files” on page 11-6.

See Also

`odeget` | `odeset`

More About

- “Choose an ODE Solver” on page 11-2
- “ODE Event Location” on page 11-12
- “Nonnegative ODE Solution” on page 11-34

ODE Event Location

In this section...

What is Event Location?

Part of the difficulty in solving some systems of ODEs is determining an appropriate time to stop the solution. The final time in the interval of integration might be defined by a specific event and not by a number. An example is an apple falling from a tree. The ODE solver should stop once the apple hits the ground, but you might not know when that event would occur beforehand. Similarly, some problems involve events that do not terminate the solution. An example is a moon orbiting a planet. In this case you might not want to stop the integration early, but you still want to detect each time the moon completes one period around the larger body.

Use event functions to detect when certain events occur during the solution of an ODE. Event functions take an expression that you specify, and detect an event when that expression is equal to zero. They can also signal the ODE solver to halt integration when they detect an event.

Writing an Event Function

Use the `'Events'` option of the `odeset` function to specify an event function. The event function must have the general form

```
[value,isterminal,direction] = myEventsFcn(t,y)
```

In the case of `ode15i`, the event function must also accept a third input argument for `yp`.

The output arguments `value`, `isterminal`, and `direction` are vectors whose `i`th element corresponds to the `i`th event:

- `value(i)` is a mathematical expression describing the `i`th event. An event occurs when `value(i)` is equal to zero.
- `isterminal(i) = 1` if the integration is to terminate when the `i`th event occurs. Otherwise, it is `0`.
- `direction(i) = 0` if all zeros are to be located (the default). A value of `+1` locates only zeros where the event function is increasing, and `-1` locates only zeros where the event function is decreasing. Specify `direction = []` to use the default value of `0` for all events.

Again, consider the case of an apple falling from a tree. The ODE that represents the falling body is

$$y'' = -1 + y'^2,$$

with the initial conditions $y(0) = 1$ and $y'(0) = 0$. You can use an event function to determine when $y(t) = 0$, which is when the apple hits the ground. For this problem, an event function that detects when the apple hits the ground is

```
function [position,isterminal,direction] = appleEventsFcn(t,y)
position = y(1); % The value that we want to be zero
isterminal = 1;  % Halt integration
direction = 0;   % The zero can be approached from either direction
```

Event Information

If you specify an events function, then call the ODE solver with three extra output arguments, as

```
[t,y,te,ye,ie] = odeXY(odefun,tspan,y0,options)
```

The three additional outputs returned by the solver correspond to the detected events:

- `te` is a column vector of the times at which events occurred.
- `ye` contains the solution value at each of the event times in `te`.
- `ie` contains indices into the vector returned by the event function. The values indicate which event the solver detected.

Alternatively, you can call the solver with a single output, as

```
sol = odeXY(odefun,tspan,y0,options)
```

In this case, the event information is stored in the structure as `sol.te`, `sol.ye`, and `sol.ie`.

Limitations

The root-finding mechanism employed by the ODE solver in conjunction with the event function has these limitations:

- If a terminal event occurs during the first step of the integration, then the solver registers the event as nonterminal and continues integrating.
- If more than one terminal event occurs during the first step, then only the first event registers and the solver continues integrating.
- Zeros are determined by sign crossings between steps. Therefore, zeros of functions with an even number of crossings between steps can be missed.

If the solver steps past events, try reducing `RelTol` and `AbsTol` to improve accuracy. Alternatively, set `MaxStep` to place an upper bound on the step size. Adjusting `tspan` does not change the steps taken by the solver.

Simple Event Location: A Bouncing Ball

This example shows how to write a simple event function for use with an ODE solver. The example file `ballode` models the motion of a bouncing ball. The events function halts the integration each time the ball bounces, and the integration then restarts with new initial conditions. As the ball bounces, the integration stops and restarts several times.

The equations for the bouncing ball are

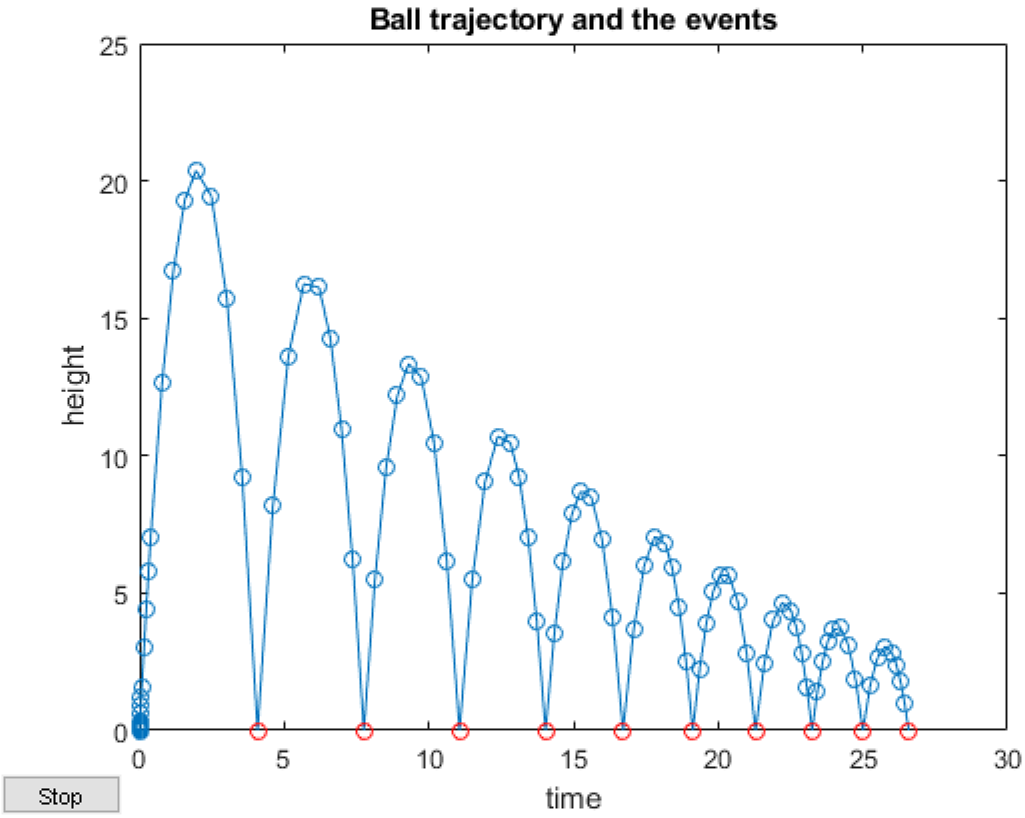
$$\begin{aligned} y_1' &= y_2 \\ y_2' &= -9.8. \end{aligned}$$

A ball bounce occurs when the height of the ball $y_1(t)$ is equal to zero after decreasing. An events function that codes this behavior is

```
function [value,isterminal,direction] = bounceEvents(t,y)
value = y(1);     % Detect height = 0
isterminal = 1;   % Stop the integration
direction = -1;   % Negative direction only
```

Type `ballode` to run the example and illustrate the use of an events function to simulate the bouncing of a ball.

```
ballode
```



Advanced Event Location: Restricted Three Body Problem

This example shows how to use the directional components of an event function. The example file `orbitode` simulates a restricted three body problem where one body is orbiting two much larger bodies. The events function determines the points in the orbit where the orbiting body is closest and farthest away. Since the value of the events function is the same at the closest and farthest points of the orbit, the direction of zero crossing is what distinguishes them.

The equations for the restricted three body problem are

$$\begin{aligned}
y'_1 &= y_3 \\
y'_2 &= y_4 \\
y'_3 &= 2y_4 + y_1 - \frac{\mu^*(y_1+\mu)}{r_1^3} - \frac{\mu(y_1-\mu^*}{r_2^3} \\
y'_4 &= -2y_3 + y_2 - \frac{\mu^* y_2}{r_1^3} - \frac{\mu y_2}{r_2^3},
\end{aligned}$$

where

$$\begin{aligned}
\mu &= 1/82.45 \\
\mu^* &= 1 - \mu \\
r_1 &= \sqrt{(y_1+\mu)^2 + y_2^2} \\
r_2 &= \sqrt{(y_1-\mu^*)^2 + y_2^2}.
\end{aligned}$$

The first two solution components are coordinates of the body of infinitesimal mass, so plotting one against the other gives the orbit of the body.

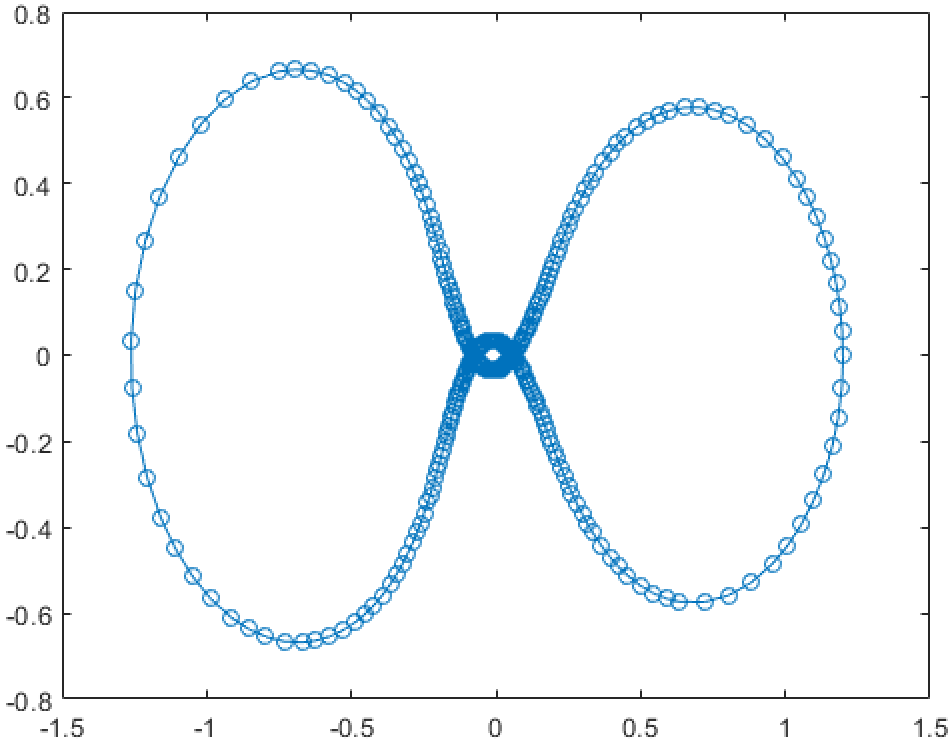
The events function nested in `orbitode.m` searches for two events. One event locates the point of maximum distance from the starting point, and the other locates the point where the spaceship returns to the starting point. The events are located accurately, even though the step sizes used by the integrator are not determined by the locations of the events. In this example, the ability to specify the direction of the zero crossing is critical. Both the point of return to the starting point and the point of maximum distance from the starting point have the same event values, and the direction of the crossing is used to distinguish them. An events function that codes this behavior is

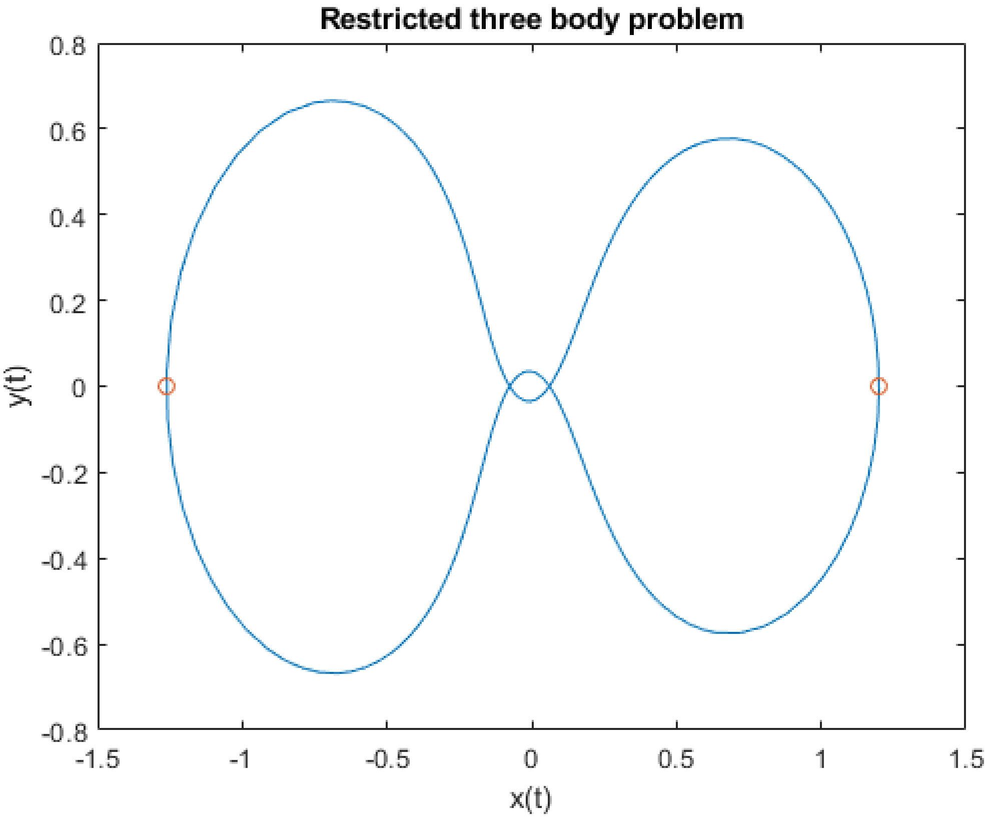
```
function [value,isterminal,direction] = orbitEvents(t,y)
% dDSQdt is the derivative of the equation for current distance. Local
% minimum/maximum occurs when this value is zero.
dDSQdt = 2 * ((y(1:2)-y0(1:2))' * y(3:4));
value = [dDSQdt; dDSQdt];
isterminal = [1;  0];         % stop at local minimum
direction  = [1; -1];         % [local minimum, local maximum]
end
```

Type `orbitode` to run the example.

```
orbitode
```

```
This is an example of event location where the ability to
specify the direction of the zero crossing is critical.  Both
the point of return to the initial point and the point of
maximum distance have the same event function value, and the
direction of the crossing is used to distinguish them.

Calling ODE45 with event functions active...

Note that the step sizes used by the integrator are NOT
determined by the location of the events, and the events are
still located accurately.
```


0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1.5
-1
-0.5
0
0.5
1
1.5



See Also

`odeget` | `odeset`

More About

- “Choose an ODE Solver” on page 11-2
- “Summary of ODE Options” on page 11-10
- “Parameterizing Functions” on page 10-2

Solve Nonstiff ODEs

This page contains two examples of solving nonstiff ordinary differential equations using `ode45`. MATLAB® has three solvers for nonstiff ODEs.

- `ode45`
- `ode23`
- `ode113`

For most nonstiff problems, `ode45` performs best. However, `ode23` is recommended for problems that permit a slightly cruder error tolerance or in the presence of moderate stiffness. Likewise, `ode113` can be more efficient than `ode45` for problems with stringent error tolerances.

If the nonstiff solvers take a long time to solve the problem or consistently fail the integration, then the problem might be stiff. See "Solve Stiff ODEs" on page 11-22 for more information.

Example: Nonstiff van der Pol Equation

The van der Pol equation is a second order ODE

$$y_1'' - \mu\left(1 - y_1^2\right)y_1' + y_1 = 0,$$

where $\mu > 0$ is a scalar parameter. Rewrite this equation as a system of first-order ODEs by making the substitution $y_1' = y_2$. The resulting system of first-order ODEs is

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1. \end{aligned}$$

The system of ODEs must be coded into a function file that the ODE solver can use. The general functional signature of an ODE function is

```
dydt = odefun(t,y)
```

That is, the function must accept both `t` and `y` as inputs, even if it does not use `t` for any computations.

The function file `vdp1.m` codes the van der Pol equation using $\mu = 1$. The variables y_1 and y_2 are represented by `y(1)` and `y(2)`, and the two-element column vector `dydt` contains the expressions for y_1' and y_2'.

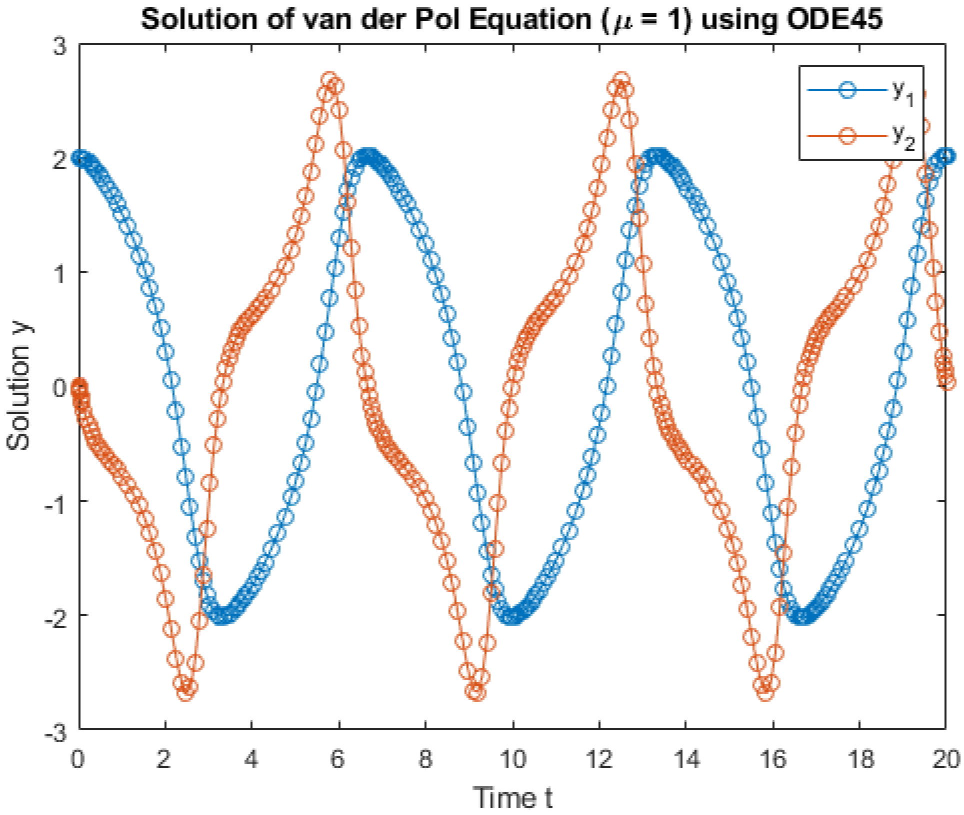
```
function dydt = vdp1(t,y)
%VDP1  Evaluate the van der Pol ODEs for mu = 1
%
%   See also ODE113, ODE23, ODE45.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Solve the ODE using the `ode45` function on the time interval `[0 20]` with initial values `[2 0]`. The output is a column vector of time points `t` and a solution array `y`. Each row in `y` corresponds to a time returned in the corresponding row of `t`. The first column of `y` corresponds to y_1, and the second column to y_2.

```
[t,y] = ode45(@vdp1,[0 20],[2; 0]);
```

Plot the solutions for y_1 and y_2 against `t`.

```
plot(t,y(:,1),'-o',t,y(:,2),'-o')
title('Solution of van der Pol Equation (\mu = 1) using ODE45');
xlabel('Time t');
ylabel('Solution y');
legend('y_1','y_2')
```



The `vdpode` function solves the same problem, but it accepts a user-specified value for μ. The van der Pol equations become stiff as μ increases. For example, with the value $\mu = 1000$ you need to use a stiff solver such as `ode15s` to solve the system.

Example: Nonstiff Euler Equations

The Euler equations for a rigid body without external forces are a standard test problem for ODE solvers intended for nonstiff problems.

The equations are

$$\begin{aligned} y_1' &= y_2 y_3 \\ y_2' &= -y_1 y_3 \\ y_3' &= -0.51 y_1 y_2 . \end{aligned}$$

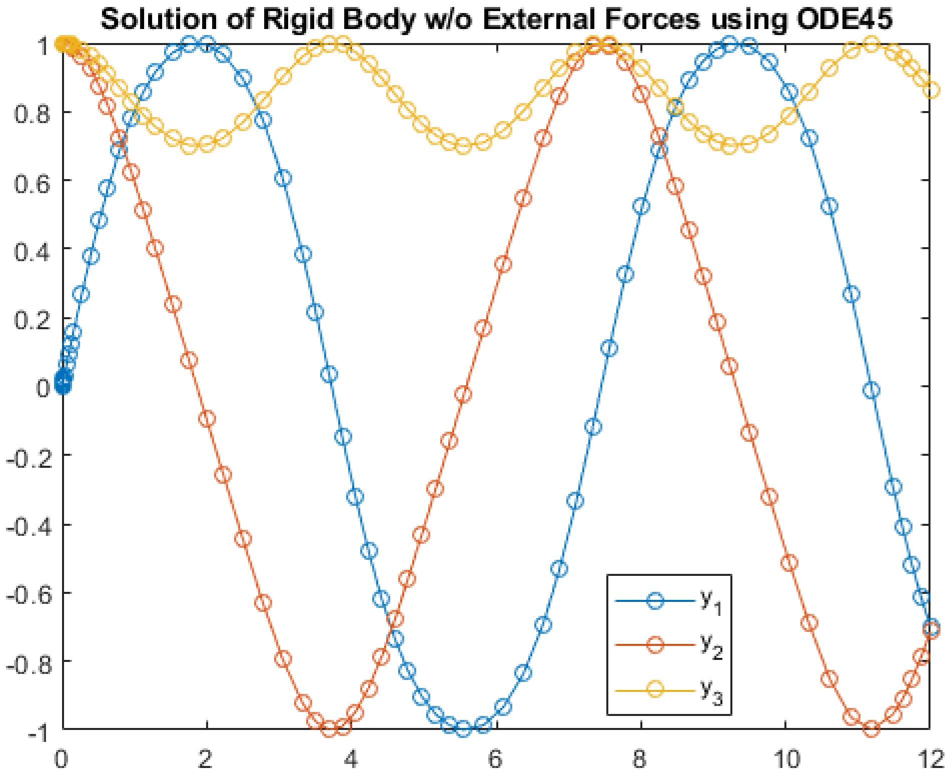
The function file `rigidode` defines and solves this first-order system of equations over the time interval `[0 12]`, using the vector of initial conditions `[0; 1; 1]` corresponding to the initial values of y_1, y_2, and y_3. The local function `f(t,y)` encodes the system of equations.

`rigidode` calls `ode45` with no output arguments, so the solver uses the default output function `odeplot` to automatically plot the solution points after each step.

```
function rigidode
%RIGIDODE  Euler equations of a rigid body without external forces.
%   A standard test problem for non-stiff solvers proposed by Krogh.  The
%   analytical solutions are Jacobian elliptic functions, accessible in
%   MATLAB.  The interval here is about 1.5 periods; it is that for which
%   solutions are plotted on p. 243 of Shampine and Gordon.
%
%   L. F. Shampine and M. K. Gordon, Computer Solution of Ordinary
%   Differential Equations, W.H. Freeman & Co., 1975.
%
%   See also ODE45, ODE23, ODE113, FUNCTION_HANDLE.

%   Mark W. Reichelt and Lawrence F. Shampine, 3-23-94, 4-19-94
%   Copyright 1984-2014 The MathWorks, Inc.

tspan = [0 12];
y0 = [0; 1; 1];

% solve the problem using ODE45
figure;
ode45(@f,tspan,y0);

% --------------------------------------------------------------------------

function dydt = f(t,y)
dydt = [    y(2)*y(3)
   -y(1)*y(3)
   -0.51*y(1)*y(2) ];
```

Solve the nonstiff Euler equations by calling the `rigidode` function.

```
rigidode
title('Solution of Rigid Body w/o External Forces using ODE45')
legend('y_1','y_2','y_3','Location','Best')
```



See Also

ode113 | ode23 | ode45

More About

- “Choose an ODE Solver” on page 11-2
- “Parameterizing Functions” on page 10-2

Solve Stiff ODEs

This page contains two examples of solving stiff ordinary differential equations using ode15s. MATLAB® has four solvers designed for stiff ODEs.

- ode15s
- ode23s
- ode23t
- ode23tb

For most stiff problems, ode15s performs best. However, ode23s, ode23t, and ode23tb can be more efficient if the problem permits a crude error tolerance.

What is a Stiff ODE?

For some ODE problems, the step size taken by the solver is forced down to an unreasonably small level in comparison to the interval of integration, even in a region where the solution curve is smooth. These step sizes can be so small that traversing a short time interval might require millions of evaluations. This can lead to the solver failing the integration, but even if it succeeds it will take a very long time to do so.

Equations that cause this behavior in ODE solvers are said to be *stiff*. The problem that stiff ODEs pose is that explicit solvers (such as ode45) are untenably slow in achieving a solution. This is why ode45 is classified as a *nonstiff solver* along with ode23 and ode113.

Solvers that are designed for stiff ODEs, known as *stiff solvers*, typically do more work per step. The pay-off is that they are able to take much larger steps, and have improved numerical stability compared to the nonstiff solvers.

Solver Options

For stiff problems, specifying the Jacobian matrix using odeset is particularly important. Stiff solvers use the Jacobian matrix $\partial f_i / \partial y_j$ to estimate the local behavior of the ODE as the integration proceeds, so supplying the Jacobian matrix (or, for large sparse systems, its sparsity pattern) is critical for efficiency and reliability. Use the Jacobian, JPattern, or Vectorized options of odeset to specify information about the Jacobian. If you do not supply the Jacobian then the solver estimates it numerically using finite differences.

See odeset for a complete listing of other solver options.

Example: Stiff van der Pol Equation

The van der Pol equation is a second order ODE

$$y_1'' - \mu \left(1 - y_1^2\right) y_1' + y_1 = 0,$$

where $\mu > 0$ is a scalar parameter. When $\mu = 1$, the resulting system of ODEs is nonstiff and easily solved using ode45. However, if you increase μ to 1000, then the solution changes dramatically and exhibits oscillation on a much longer time scale. Approximating the solution of the initial value problem becomes more difficult. Because this particular problem is stiff, a solver intended for nonstiff problems, such as ode45, is too inefficient to be practical. Use a stiff solver such as ode15s for this problem instead.

Rewrite the van der Pol equation as a system of first-order ODEs by making the substitution $y_1' = y_2$. The resulting system of first-order ODEs is

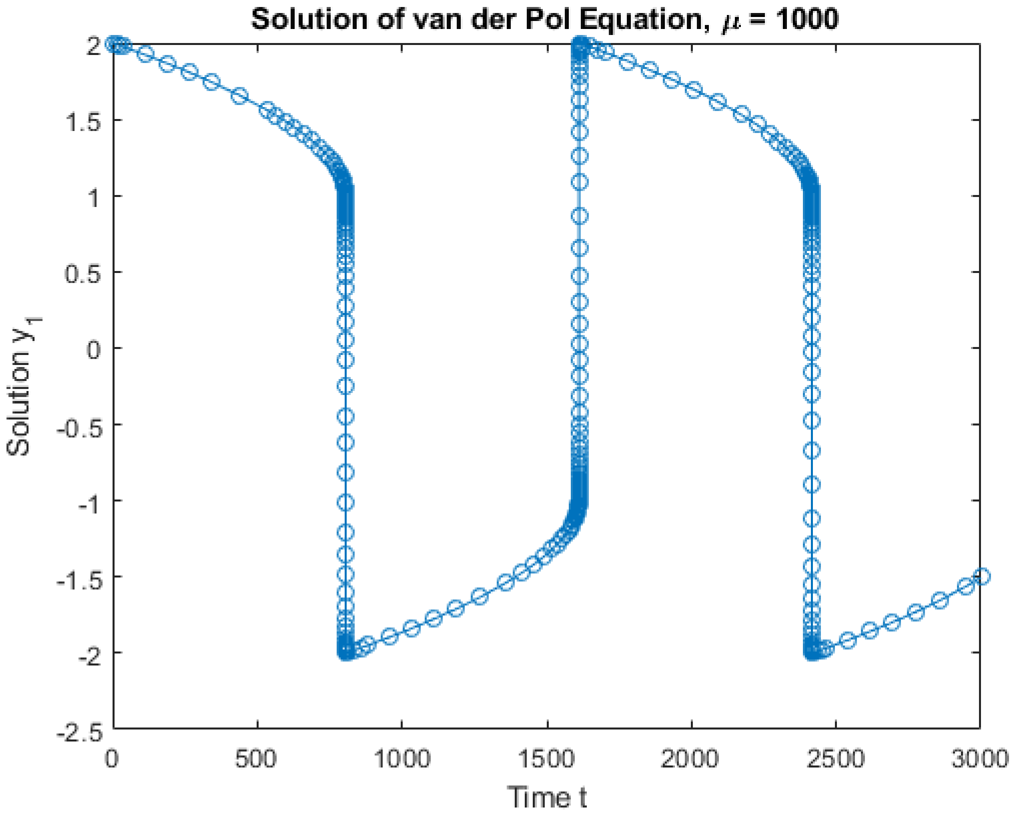
$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1. \end{aligned}$$

The `vdp1000` function evaluates the van der Pol equation using $\mu = 1000$.

```
function dydt = vdp1000(t,y)
%VDP1000  Evaluate the van der Pol ODEs for mu = 1000.
%
%   See also ODE15S, ODE23S, ODE23T, ODE23TB.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

Use the `ode15s` function to solve the problem with an initial conditions vector of `[2; 0]`, over a time interval of `[0 3000]`. For scaling reasons, plot only the first component of the solution.

```
[t,y] = ode15s(@vdp1000,[0 3000],[2; 0]);
plot(t,y(:,1),'-o');
title('Solution of van der Pol Equation, \mu = 1000');
xlabel('Time t');
ylabel('Solution y_1');
```



The `vdpode` function also solves the same problem, but it accepts a user-specified value for μ. The equations become increasingly stiff as μ increases.

Example: Sparse Brusselator System

The classic Brusselator system of equations is potentially large, stiff, and sparse. The Brusselator system models diffusion in a chemical reaction, and is represented by a system of equations involving u, v, u', and v'.

$$\begin{aligned} u_i' &= 1 + u_i^2 v_i - 4u_i + \alpha\,(N+1)^2\,(u_{i-1} - 2_i + u_{i+1}) \\ v_i' &= 3u_i - u_i^2 v_i + \alpha\,(N+1)^2\,(v_{i-1} - 2v_i + v_{i+1}) \end{aligned}$$

The function file `brussode` solves this set of equations on the time interval `[0,10]` with $\alpha = 1/50$. The initial conditions are

$$\begin{aligned} u_j(0) &= 1 + \sin(2\pi x_j) \\ v_j(0) &= 3, \end{aligned}$$

where $x_j = i/N + 1$ for $i = 1, \ldots, N$. Therefore, there are $2N$ equations in the system, but the Jacobian $\partial f/\partial y$ is a banded matrix with a constant width of 5 if the equations are ordered as $u_1, v_1, u_2, v_2, \ldots$. As N increases, the problem becomes increasingly stiff, and the Jacobian becomes increasingly sparse.

The function call `brussode(N)`, for $N \geq 2$, specifies a value for `N` in the system of equations, corresponding to the number of grid points. By default, `brussode` uses $N = 20$.

`brussode` contains a few subfunctions:

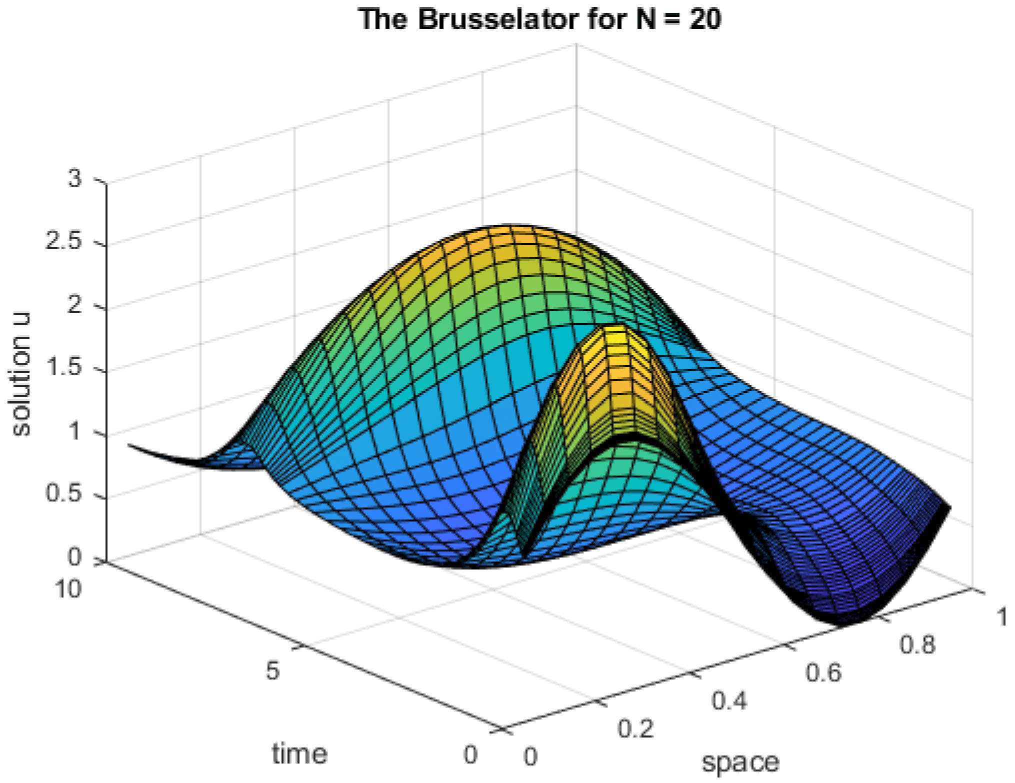
- The nested function `f(t,y)` encodes the system of equations for the Brusselator problem, returning a vector.
- The local function `jpattern(N)` returns a sparse matrix of 1s and 0s showing the locations of nonzeros in the Jacobian. This matrix is assigned to the `JPattern` field of the options structure. The ODE solver uses this sparsity pattern to generate the Jacobian numerically as a sparse matrix. Supplying this sparsity pattern in the problem significantly reduces the number of function evaluations required to generate the 2N-by-2N Jacobian, from 2N evaluations to just 4.

```
function brussode(N)
%BRUSSODE  Stiff problem modelling a chemical reaction (the Brusselator).
%   The parameter N >= 2 is used to specify the number of grid points; the
%   resulting system consists of 2N equations. By default, N is 20.  The
%   problem becomes increasingly stiff and increasingly sparse as N is
%   increased.  The Jacobian for this problem is a sparse constant matrix
%   (banded with bandwidth 5).
%
%   The property 'JPattern' is used to provide the solver with a sparse
%   matrix of 1's and 0's showing the locations of nonzeros in the Jacobian
%   df/dy.  By default, the stiff solvers of the ODE Suite generate Jacobians
%   numerically as full matrices.  However, when a sparsity pattern is
%   provided, the solver uses it to generate the Jacobian numerically as a
%   sparse matrix.  Providing a sparsity pattern can significantly reduce the
%   number of function evaluations required to generate the Jacobian and can
%   accelerate integration.  For the BRUSSODE problem, only 4 evaluations of
%   the function are needed to compute the 2N x 2N Jacobian matrix.
%
%   Setting the 'Vectorized' property indicates the function f is
%   vectorized.
%
%   E. Hairer and G. Wanner, Solving Ordinary Differential Equations II,
%   Stiff and Differential-Algebraic Problems, Springer-Verlag, Berlin,
%   1991, pp. 5-8.
%
%   See also ODE15S, ODE23S, ODE23T, ODE23TB, ODESET, FUNCTION_HANDLE.

%   Mark W. Reichelt and Lawrence F. Shampine, 8-30-94
%   Copyright 1984-2014 The MathWorks, Inc.

% Problem parameter, shared with the nested function.
if nargin<1
   N = 20;
end

tspan = [0; 10];
y0 = [1+sin((2*pi/(N+1))*(1:N)); repmat(3,1,N)];

options = odeset('Vectorized','on','JPattern',jpattern(N));

[t,y] = ode15s(@f,tspan,y0,options);

u = y(:,1:2:end);
```

```
x = (1:N)/(N+1);
figure;
surf(x,t,u);
view(-40,30);
xlabel('space');
ylabel('time');
zlabel('solution u');
title(['The Brusselator for N = ' num2str(N)]);

% -------------------------------------------------------------------------
% Nested function -- N is provided by the outer function.
%

   function dydt = f(t,y)
      % Derivative function
      c = 0.02 * (N+1)^2;
      dydt = zeros(2*N,size(y,2));      % preallocate dy/dt

      % Evaluate the 2 components of the function at one edge of the grid
      % (with edge conditions).
      i = 1;
      dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + c*(1-2*y(i,:)+y(i+2,:));
      dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + c*(3-2*y(i+1,:)+y(i+3,:));

      % Evaluate the 2 components of the function at all interior grid points.
      i = 3:2:2*N-3;
      dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
         c*(y(i-2,:)-2*y(i,:)+y(i+2,:));
      dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
         c*(y(i-1,:)-2*y(i+1,:)+y(i+3,:));

      % Evaluate the 2 components of the function at the other edge of the grid
      % (with edge conditions).
      i = 2*N-1;
      dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + c*(y(i-2,:)-2*y(i,:)+1);
      dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + c*(y(i-1,:)-2*y(i+1,:)+3);
   end
% -------------------------------------------------------------------------

end  % brussode

% ---------------------------------------------------------------------------
% Subfunction -- the sparsity pattern
%

function S = jpattern(N)
% Jacobian sparsity pattern
B = ones(2*N,5);
B(2:2:2*N,2) = zeros(N,1);
B(1:2:2*N-1,4) = zeros(N,1);
S = spdiags(B,-2:2,2*N,2*N);
end
% ---------------------------------------------------------------------------
```

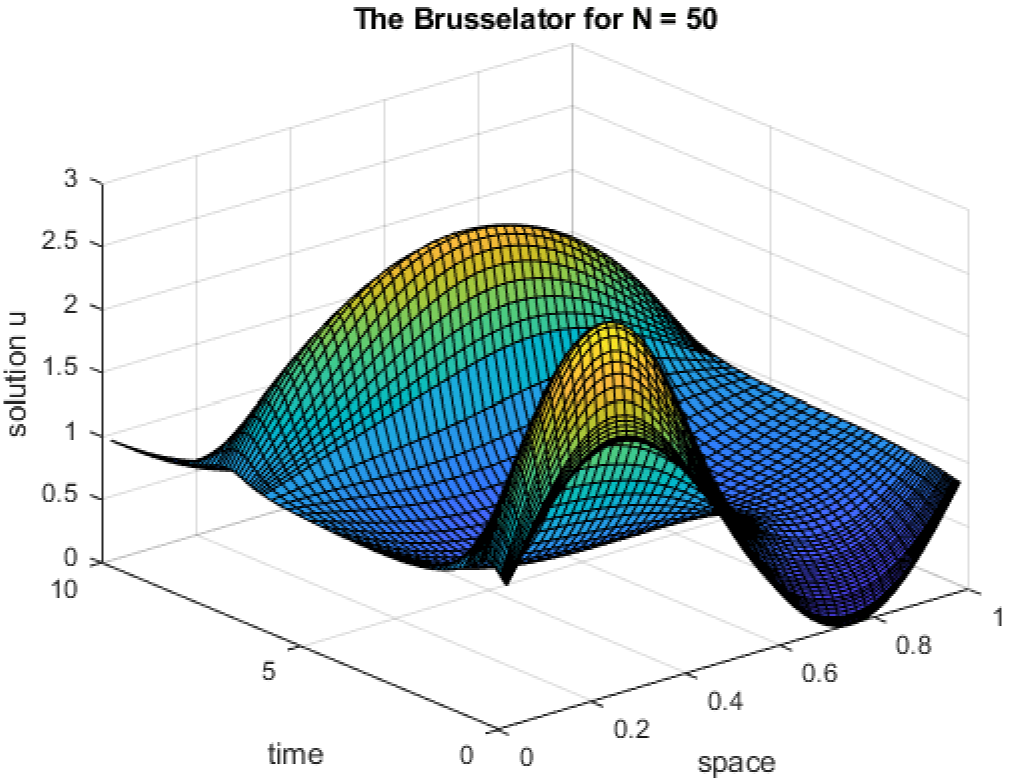
Solve the Brusselator system for $N = 20$ by running the function brussode.

```
brussode
```



Solve the system for $N = 50$ by specifying an input to `brussode`.

```
brussode(50)
```



See Also

`ode15s` | `ode23s` | `ode23t` | `ode23tb`

More About

- "Choose an ODE Solver" on page 11-2
- "Summary of ODE Options" on page 11-10
- "Parameterizing Functions" on page 10-2

Solve Differential Algebraic Equations (DAEs)

In this section...

What is a Differential Algebraic Equation?

Differential algebraic equations are a type of differential equation where one or more derivatives of dependent variables are not present in the equations. Variables that appear in the equations without their derivative are called *algebraic*, and the presence of algebraic variables means that you cannot write down the equations in the explicit form $y' = f(t, y)$. Instead, you can solve DAEs with these forms:

- The `ode15s` and `ode23t` solvers can solve index-1 linearly implicit problems with a singular mass matrix $M(t, y)y' = f(t, y)$, including semi-explicit DAEs of the form

$$\begin{aligned} y' &= f(t, y, z) \\ 0 &= g(t, y, z) \ . \end{aligned}$$

  In this form, the presence of algebraic variables leads to a singular mass matrix, since there are one or more zeros on the main diagonal.

$$My' = \begin{pmatrix} y'_1 & 0 & \cdots & 0 \\ 0 & y'_2 & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & 0 \end{pmatrix} .$$

  By default, solvers automatically test the singularity of the mass matrix to detect DAE systems. If you know about singularity ahead of time then you can set the `MassSingular` option of `odeset` to `'yes'`. With DAEs, you can also provide the solver with a guess of the initial conditions for y'_0 using the `InitialSlope` property of `odeset`. This is in addition to specifying the usual initial conditions for y_0 in the call to the solver.

- The `ode15i` solver can solve more general DAEs in the fully implicit form

$$f(t, y, y') = 0 \ .$$

  In the fully implicit form, the presence of algebraic variables leads to a singular Jacobian matrix. This is because at least one of the columns in the matrix is guaranteed to contain all zeros, since the derivative of that variable does not appear in the equations.

$$J = \partial f / \partial y' = \begin{pmatrix} \frac{\partial f_1}{\partial y'_1} & \cdots & \frac{\partial f_1}{\partial y'_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial y'_1} & \cdots & \frac{\partial f_m}{\partial y'_n} \end{pmatrix}$$

The `ode15i` solver requires that you specify initial conditions for both y'_0 and y_0. Also, unlike the other ODE solvers, `ode15i` requires the function encoding the equations to accept an extra input: `odefun(t,y,yp)`.

DAEs arise in a wide variety of systems because physical conservation laws often have forms like $x + y + z = 0$. If `x`, `x'`, `y`, and `y'` are defined explicitly in the equations, then this conservation equation is sufficient to solve for `z` without having an expression for `z'`.

Consistent Initial Conditions

When you are solving a DAE, you can specify initial conditions for both y'_0 and y_0. The `ode15i` solver requires both initial conditions to be specified as input arguments. For the `ode15s` and `ode23t` solvers, the initial condition for y'_0 is optional (but can be specified using the `InitialSlope` option of `odeset`). In both cases, it is possible that the initial conditions you specify do not agree with the equations you are trying to solve. Initial conditions that conflict with one another are called *inconsistent*. The treatment of the initial conditions varies by solver:

- `ode15s` and `ode23t` — If you do not specify an initial condition for y'_0, then the solver automatically computes consistent initial conditions based on the initial condition you provide for y_0. If you specify an inconsistent initial condition for y'_0, then the solver treats the values as guesses, attempts to compute consistent values close to the guesses, and continues on to solve the problem.
- `ode15i` — The initial conditions you supply to the solver must be consistent, and `ode15i` does not check the supplied values for consistency. The helper function `decic` computes consistent initial conditions for this purpose.

Differential Index

DAEs are characterized by their *differential index*, which is a measure of their singularity. By differentiating equations you can eliminate algebraic variables, and if you do this enough times then the equations take the form of a system of explicit ODEs. The differential index of a system of DAEs is the number of derivatives you must take to express the system as an equivalent system of explicit ODEs. Thus, ODEs have a differential index of 0.

An example of an index-1 DAE is

$$y(t) = k(t)\ .$$

For this equation, you can take a single derivative to obtain the explicit ODE form

$$y' = k'(t)\ .$$

An example of an index-2 DAE is

$$\begin{aligned} y'_1 &= y_2 \\ 0 &= k(t) - y_1\ . \end{aligned}$$

These equations require two derivatives to be rewritten in the explicit ODE form

$$\begin{aligned} y'_1 &= k'(t) \\ y'_2 &= k''(t)\ . \end{aligned}$$

The ode15s and ode23t solvers only solve DAEs of index 1. If the index of your equations is 2 or higher, then you need to rewrite the equations as an equivalent system of index-1 DAEs. It is always possible to take derivatives and rewrite a DAE system as an equivalent system of index-1 DAEs. Be aware that if you replace algebraic equations with their derivatives, then you might have removed some constraints. If the equations no longer include the original constraints, then the numerical solution can drift.

If you have Symbolic Math Toolbox, then see “Solve Differential Algebraic Equations (DAEs)” (Symbolic Math Toolbox) for more information.

Imposing Nonnegativity

Most of the options in odeset on page 11-10 work as expected with the DAE solvers ode15s, ode23t, and ode15i. However, one notable exception is with the use of the NonNegative on page 11-34 option. The NonNegative option does not support implicit solvers (ode15s, ode23t, ode23tb) applied to problems with a mass matrix. Therefore, you cannot use this option to impose nonnegativity constraints on a DAE problem, which necessarily has a singular mass matrix. For more details, see [1] on page 11-33.

Solve Robertson Problem as Semi-Explicit Differential Algebraic Equations (DAEs)

This example reformulates a system of ODEs as a system of differential algebraic equations (DAEs). The Robertson problem found in hb1ode.m is a classic test problem for programs that solve stiff ODEs. The system of equations is

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= 0.04y_1 - 10^4 y_2 y_3 - (3 \times 10^7)y_2^2 \\ y_3' &= (3 \times 10^7)y_2^2. \end{aligned}$$

hb1ode solves this system of ODEs to steady state with the initial conditions $y_1 = 1$, $y_2 = 0$, and $y_3 = 0$. But the equations also satisfy a linear conservation law,

$$y_1' + y_2' + y_3' = 0.$$

In terms of the solution and initial conditions, the conservation law is

$$y_1 + y_2 + y_3 = 1.$$

The system of equations can be rewritten as a system of DAEs by using the conservation law to determine the state of y_3. This reformulates the problem as the DAE system

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= 0.04y_1 - 10^4 y_2 y_3 - (3 \times 10^7)y_2^2 \\ 0 &= y_1 + y_2 + y_3 - 1. \end{aligned}$$

The differential index of this system is 1, since only a single derivative of y_3 is required to make this a system of ODEs. Therefore, no further transformations are required before solving the system.

The function robertsdae encodes this DAE system. Save robertsdae.m in your current folder to run the example.

```
function out = robertsdae(t,y)
out = [-0.04*y(1) + 1e4*y(2).*y(3)
   0.04*y(1) - 1e4*y(2).*y(3) - 3e7*y(2).^2
   y(1) + y(2) + y(3) - 1 ];
```

The full example code for this formulation of the Robertson problem is available in hb1dae.m.

Solve the DAE system using `ode15s`. Consistent initial conditions for `y0` are obvious based on the conservation law. Use `odeset` to set the options:

- Use a constant mass matrix to represent the left hand side of the system of equations.

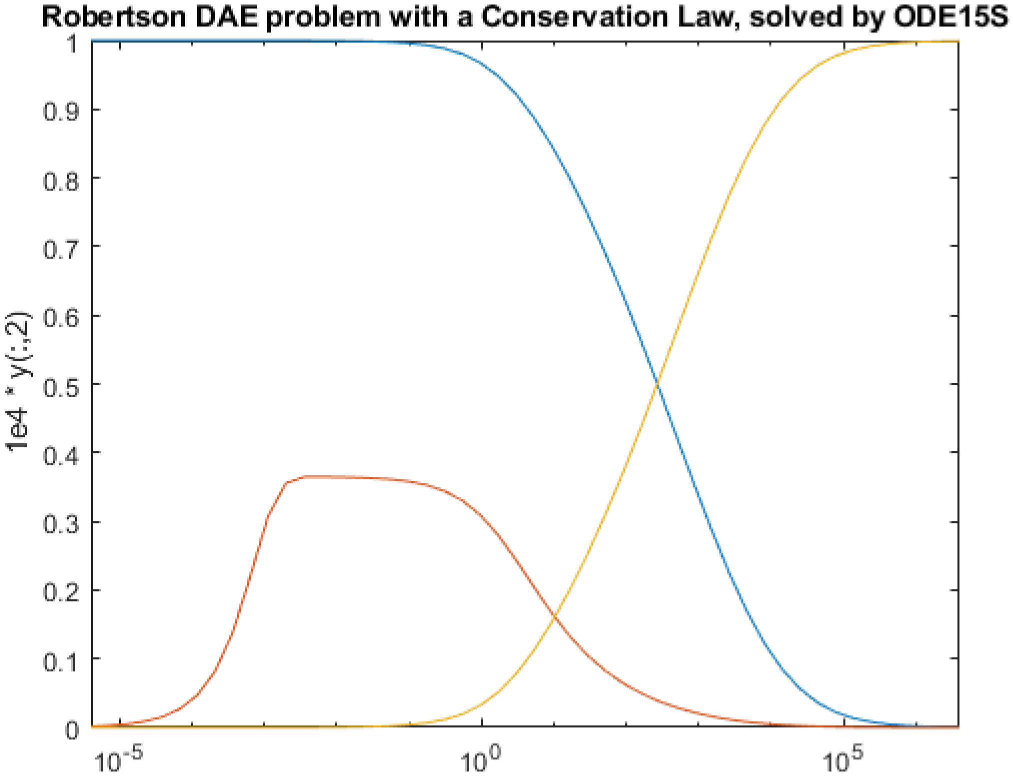
$$\begin{pmatrix} y_1' \\ y_2' \\ 0 \end{pmatrix} = My' \rightarrow M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

- Set the relative error tolerance to `1e-4`.
- Use an absolute tolerance of `1e-10` for the second solution component, since the scale varies dramatically from the other components.
- Leave the `'MassSingular'` option at its default value `'maybe'` to test the automatic detection of a DAE.

```
y0 = [1; 0; 0];
tspan = [0 4*logspace(-6,6)];
M = [1 0 0; 0 1 0; 0 0 0];
options = odeset('Mass',M,'RelTol',1e-4,'AbsTol',[1e-6 1e-10 1e-6]);
[t,y] = ode15s(@robertsdae,tspan,y0,options);
```

Plot the solution.

```
y(:,2) = 1e4*y(:,2);
semilogx(t,y);
ylabel('1e4 * y(:,2)');
title('Robertson DAE problem with a Conservation Law, solved by ODE15S');
```



References

[1] Shampine, L.F., S. Thompson, J.A. Kierzenka, and G.D. Byrne. "Non-negative solutions of ODEs." *Applied Mathematics and Computation*. Vol. 170, 2005, pp. 556-569.

See Also

`ode15i` | `ode15s` | `ode23t` | `odeset`

More About

- “Choose an ODE Solver” on page 11-2
- “Summary of ODE Options” on page 11-10
- “Equation Solving” (Symbolic Math Toolbox)

External Websites

- Solving Index-1 DAEs in MATLAB and Simulink

Nonnegative ODE Solution

This topic shows how to constrain the solution of an ODE to be nonnegative. Imposing nonnegativity is not always trivial, but sometimes it is necessary due to the physical interpretation of the equations or due to the nature of the solution. You should only impose this constraint on the solution when necessary, such as in cases where the integration fails without it, or where the solution would be inapplicable.

If certain components of the solution must be nonnegative, then use `odeset` to set the `NonNegative` option for the indices of these components. This option is not available for `ode23s`, `ode15i`, or for implicit solvers (`ode15s`, `ode23t`, `ode23tb`) applied to problems with a mass matrix. In particular, you cannot impose nonnegativity constraints on a DAE problem, which necessarily has a singular mass matrix.

Example: Absolute Value Function

Consider the initial value problem

$$y' = -|y|,$$

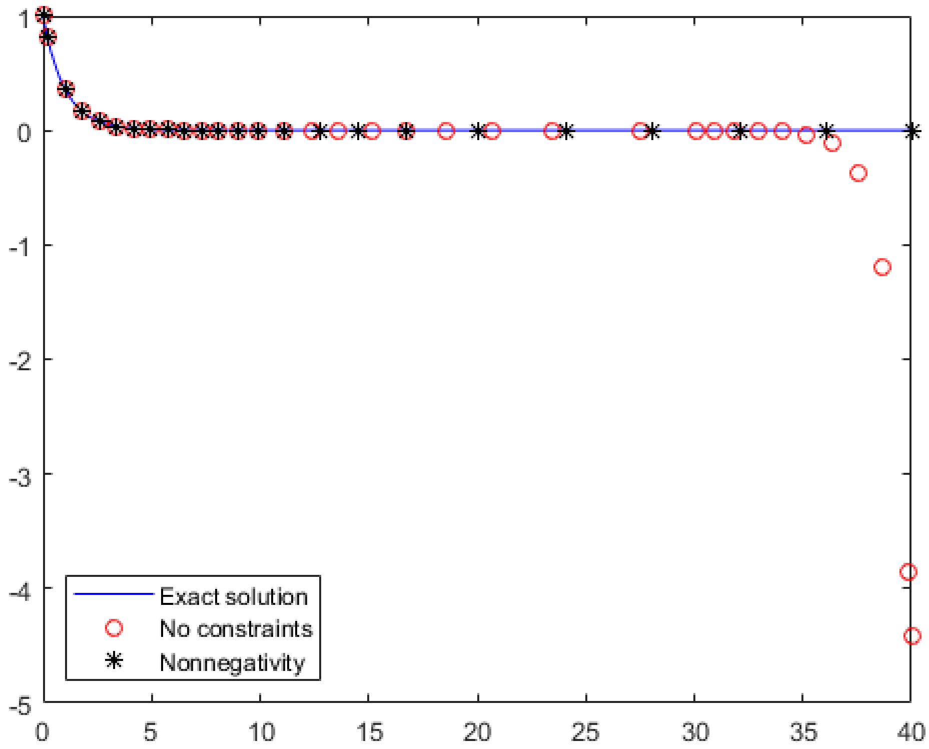
solved on the interval $[0, 40]$ with the initial condition $y(0) = 1$. The solution of this ODE decays to zero. If the solver produces a negative solution value, then it begins to track the solution of the ODE through this value, and the computation eventually fails as the calculated solution diverges to $-\infty$. Using the `NonNegative` option prevents this integration failure.

Compare the analytic solution of $y(t) = e^{-t}$ to a solution of the ODE using `ode45` with no extra options, and to one with the `NonNegative` option set.

```
ode = @(t,y) -abs(y);

% Standard solution with |ode45|
options1 = odeset('Refine',1);
[t0,y0] = ode45(ode,[0 40],1,options1);

% Solution with nonnegative constraint
options2 = odeset(options1,'NonNegative',1);
[t1,y1] = ode45(ode,[0 40],1,options2);

% Analytic solution
t = linspace(0,40,1000);
y = exp(-t);
```

Plot the three solutions for comparison. Imposing nonnegativity is crucial to keep the solution from veering off toward $-\infty$.

```
plot(t,y,'b-',t0,y0,'ro',t1,y1,'k*');
legend('Exact solution','No constraints','Nonnegativity', ...
       'Location','SouthWest')
```



Example: The Knee Problem

Another example of a problem that requires a nonnegative solution is the *knee problem* coded in the example file `kneeode`. The equation is

$$\epsilon y' = (1 - x)y - y^2,$$

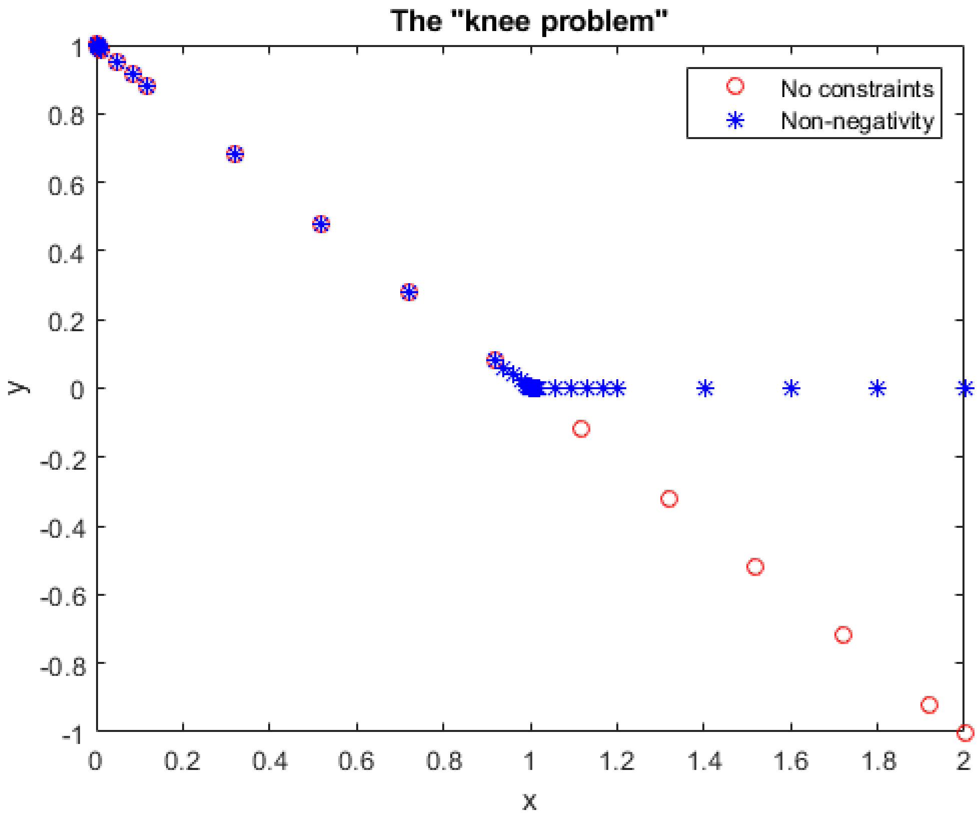
solved on the interval $[0, 2]$ with the initial condition $y(0) = 1$. The parameter ϵ generally is taken to satisfy $0 < \epsilon \ll 1$, and this problem uses $\epsilon = 1 \times 10^{-6}$. The solution to this ODE approaches $y = 1 - x$ for $x < 1$ and $y = 0$ for $x > 1$. However, computing the numerical solution with default tolerances shows that the solution follows the $y = 1 - x$ isocline for the whole interval of integration. Imposing nonnegativity constraints results in the correct solution.

Solve the knee problem with and without nonnegativity constraints.

```
epsilon = 1e-6;
y0 = 1;
xspan = [0 2];
odefcn = @(x,y,epsilon) ((1-x)*y - y^2)/epsilon;

% Solve without imposing constraints
[x1,y1] = ode15s(@(x,y) odefcn(x,y,epsilon), xspan, y0);

% Impose a nonnegativity constraint
options = odeset('NonNegative',1);
[x2,y2] = ode15s(@(x,y) odefcn(x,y,epsilon), xspan, y0, options);
```

Plot the solutions for comparison.

```
plot(x1,y1,'ro',x2,y2,'b*')
axis([0,2,-1,1])
title('The "knee problem"')
legend('No constraints','Non-negativity')
xlabel('x')
ylabel('y')
```



References

[1] Shampine, L.F., S. Thompson, J.A. Kierzenka, and G.D. Byrne, "Non-negative solutions of ODEs," *Applied Mathematics and Computation* Vol. 170, 2005, pp. 556-569.

See Also

`odeset`

More About

- “Choose an ODE Solver” on page 11-2
- “Summary of ODE Options” on page 11-10

Troubleshoot Common ODE Problems

Error Tolerances

Question or Problem	Answer
How do I choose the error thresholds `RelTol` and `AbsTol`?	`RelTol`, the relative accuracy tolerance, controls the number of correct digits in the computed answer. `AbsTol`, the absolute error tolerance, controls the difference between the computed answer and the true solution. At each step, the error `e` in component `i` of the solution satisfies `|e(i)| ≤ max(RelTol*abs(y(i)),AbsTol(i))` Roughly speaking, this means that you want `RelTol` correct digits in all solution components, excluding those smaller than the threshold `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you might have to specify a value for `AbsTol(i)` that is small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.
I want answers that are correct to the precision of the computer. Why can I not simply set `RelTol` to `eps`?	You can get close to machine precision, but not that close. The solvers do not allow `RelTol` near `eps` because they try to approximate a continuous function. At tolerances comparable to `eps`, the machine arithmetic causes all functions to look discontinuous.
How do I tell the solver that I do not care about getting an accurate answer for one of the solution components?	You can increase the absolute error tolerance `AbsTol` for this solution component. If the tolerance is bigger than the solution component, this specifies that no digits in the component need to be correct. The solver might have to get some correct digits in this component to compute other components accurately, but it generally handles this automatically.

Problem Scale

Question or Problem	Answer
How large a problem can I solve with the ODE suite?	The primary constraints are memory and time. At each time step, the solvers for nonstiff problems allocate vectors of length `n`, where `n` is the number of equations in the system. The solvers for stiff problems allocate vectors of length `n` but also allocate an `n`-by-`n` Jacobian matrix. For these solvers, it might be advantageous to specify the Jacobian sparsity pattern using the `JPattern` option of `odeset`. If the problem is nonstiff, or if you are using the `JPattern` option, it might be possible to solve a problem with thousands of unknowns. In this case, however, storage of the result can be problematic. Ask the solver to evaluate the solution at specific points only, or call the solver with no output arguments and use an output function to monitor the solution.
I am solving a very large system, but only care about a few of the components of `y`. Is there any way to avoid storing all of the elements?	Yes. The `OutputFcn` option is designed specifically for this purpose. When you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history. Instead, the solver calls `OutputFcn(t,y,flag)` at each time step. To keep the history of specific elements, write an output function that stores or plots only the elements you care about.
What is the startup cost of the integration and how can I reduce it?	The biggest startup cost occurs as the solver attempts to find a step size appropriate to the scale of the problem. If you happen to know an appropriate step size, use the `InitialStep` option. For example, if you repeatedly call the integrator in an event location loop, the last step that was taken before the event is probably scaled correctly for the next integration. Type `edit ballode` to see an example.
The first step size that the integrator takes is too large, and it misses important behavior.	You can specify the first step size with the `InitialStep` option. The integrator tries this step size, then reduces it if necessary.

Solution Components

Question or Problem	Answer
The solution does not look like what I expected.	If your expectations are correct, then reduce the error tolerances from their default values. A smaller relative error tolerance is needed to accurately solve problems integrated over “long” intervals, as well as problems that are moderately unstable. Check whether there are solution components that stay smaller than their absolute error tolerance for some time. If so, you are not requiring any correct digits in these components. This might be acceptable for these components, but failing to compute them accurately might degrade the accuracy of other components that depend on them.
My plots are not smooth enough.	Increase the value of `Refine` from its default of `4` in `ode45` or `1` in the other solvers. The bigger the value of `Refine`, the more output points the solver generates. Execution speed is not affected much by the value of `Refine`.
I am plotting the solution as it is computed and it looks fine, but the code gets stuck at some point.	First verify that the ODE function is smooth near the point where the code gets stuck. If it is not, then the solver must take small steps to deal with this. It might help to break the integration interval into pieces over which the ODE function is smooth. If the function is smooth and the code is taking extremely small steps, you are probably trying to solve a stiff problem with a solver not intended for that purpose. Switch to using one of the stiff solvers `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.
What if I have the final and not the initial value?	All the solvers of the ODE suite allow you to solve backward or forward in time. The syntax for the solvers is `[t,y] = ode45(odefun,[t0 tf],y0);` and the syntax accepts `t0 > tf`.

Question or Problem	Answer
My integration proceeds very slowly, using too many time steps.	First, check that `tspan` is not too long. Remember that the solver uses as many time points as necessary to produce a smooth solution. If the ODE function changes on a time scale that is very short compared to `tspan`, then the solver uses a lot of time steps. Long-time integration is a hard problem. Break `tspan` into smaller pieces. If the ODE function does not change noticeably on the `tspan` interval, it could be that your problem is stiff. Try using one of the stiff solvers `ode15s`, `ode23s`, `ode23t`, or `ode23tb`. Finally, make sure that the ODE function is written in an efficient way. The solvers evaluate the derivatives in the ODE function many times. The cost of numerical integration depends critically on the expense of evaluating the ODE function. Rather than recomputing complicated constant parameters at each evaluation, store them in global variables, or calculate them once and pass them to nested functions.
I know that the solution undergoes a radical change at time `t`, where `t0 <= t <= tf` , but the integrator steps past without “seeing” it.	If you know that there is a sharp change at time `t`, try breaking the `tspan` interval into two pieces, `[t0 t]` and `[t tf]`, and call the integrator twice or continue the integration using `odextend`. If the differential equation has periodic coefficients or solutions, ensure the solver does not step over periods by restricting the maximum step size to the length of the period.

Problem Type

Can the solvers handle partial differential equations (PDEs) that have been discretized by the method of lines?	Yes, because the discretization produces a system of ODEs. Depending on the discretization, you might have a form involving mass matrices, which the ODE solvers provide for. Often the system is stiff. This is to be expected if the PDE is parabolic, or when there are phenomena that happen on very different time scales such as a chemical reaction in a fluid flow. In such cases, use one of the four stiff solvers `ode15s`, `ode23s`, `ode23t`, or `ode23tb`. If there are many equations, use the `JPattern` option to specify the Jacobian sparsity pattern. This can make the difference between success and failure as it prevents the computation from being too expensive. Type `edit burgersode` to see an example that uses `JPattern`. If the system is not stiff, or not very stiff, then `ode23` and `ode45` are more efficient than the stiff solvers `ode15s`, `ode23s`, `ode23t`, and `ode23tb`. You can solve parabolic-elliptic partial differential equations in 1-D directly with the MATLAB PDE solver `pdepe`.
Can I integrate a set of sampled data?	Not directly. Instead, represent the data as a function by interpolation or some other scheme for fitting data. The smoothness of this function is critical. A piecewise polynomial fit such as a spline can look smooth to the eye, but rough to a solver; the solver takes small steps where the derivatives of the fit have jumps. Either use a smooth function to represent the data or use one of the lower-order solvers (`ode23`, `ode23s`, `ode23t`, `ode23tb`) that is less sensitive to smoothness. See “ODE with Time-Dependent Terms” for an example.

See Also

`deval` | `odeget` | `odeset` | `odextend`

More About

- “Choose an ODE Solver” on page 11-2
- “Summary of ODE Options” on page 11-10
- “ODE Event Location” on page 11-12
- “Nonnegative ODE Solution” on page 11-34

Differential Equations

This example shows how to use MATLAB® to formulate and solve several different types of differential equations. MATLAB offers several numerical algorithms to solve a wide variety of differential equations:

- Initial value problems
- Boundary value problems
- Delay differential equations
- Partial differential equations

Initial Value Problem

`vanderpoldemo` is a function that defines the van der Pol equation

$$\frac{d^2y}{dt^2} - \mu\left(1 - y^2\right)\frac{dy}{dt} + y = 0\,.$$

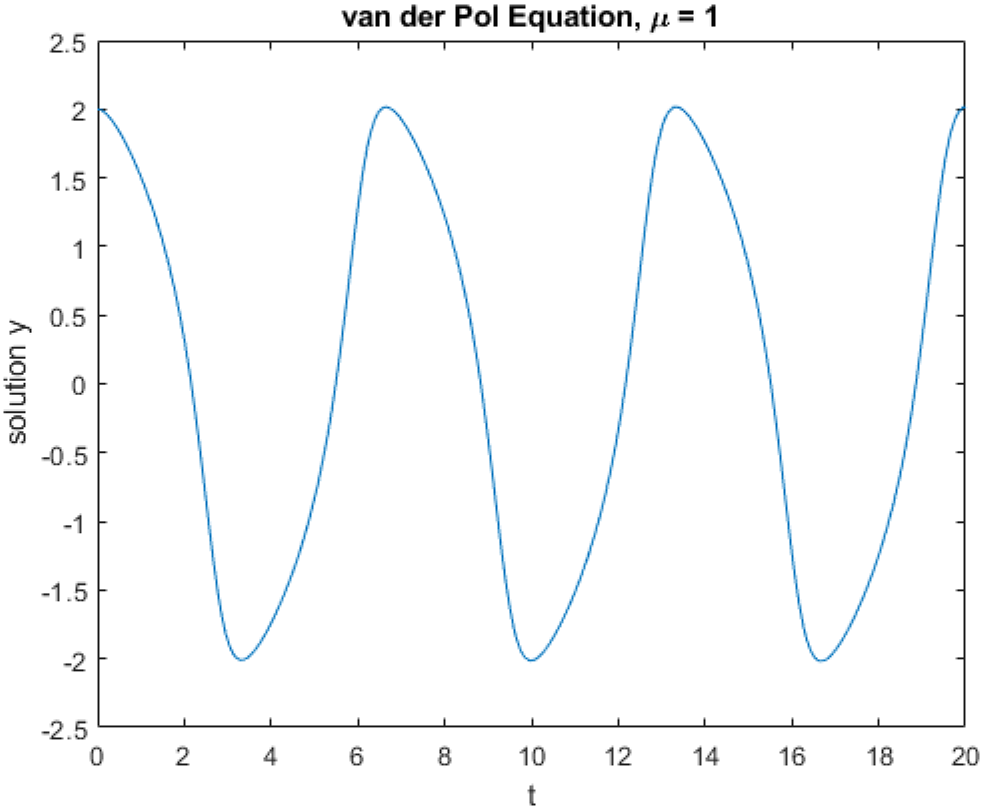
```
type vanderpoldemo
```

```
function dydt = vanderpoldemo(t,y,Mu)
%VANDERPOLDEMO Defines the van der Pol equation for ODEDEMO.

% Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); Mu*(1-y(1)^2)*y(2)-y(1)];
```

The equation is written as a system of two first-order ordinary differential equations (ODEs). These equations are evaluated for different values of the parameter μ. For faster integration, you should choose an appropriate solver based on the value of μ.

For $\mu = 1$, any of the MATLAB ODE solvers can solve the van der Pol equation efficiently. The `ode45` solver is one such example. The equation is solved in the domain $[0, 20]$ with the initial conditions $y(0) = 2$ and $\left.\frac{\mathrm{dy}}{\mathrm{dt}}\right|_{t\,=\,0} = 0$.

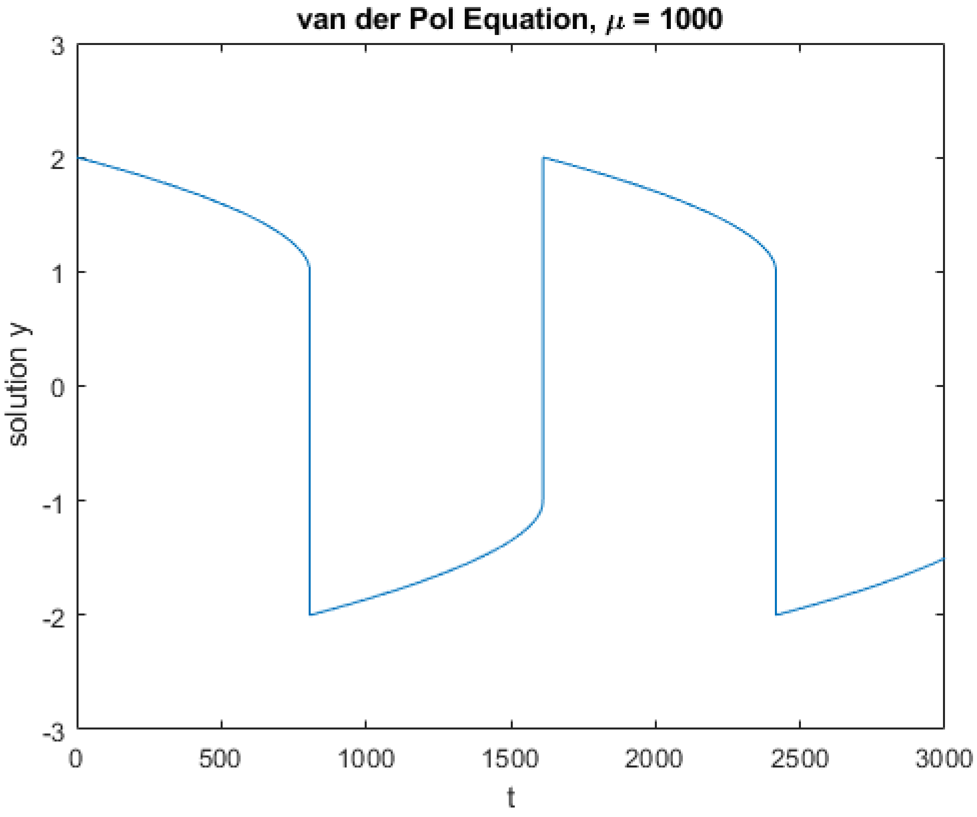
```
tspan = [0 20];
y0 = [2; 0];
Mu = 1;
ode = @(t,y) vanderpoldemo(t,y,Mu);
[t,y] = ode45(ode, tspan, y0);

% Plot solution
plot(t,y(:,1))
xlabel('t')
ylabel('solution y')
title('van der Pol Equation, \mu = 1')
```



For larger magnitudes of μ, the problem becomes *stiff*. This label is for problems that resist attempts to be evaluated with ordinary techniques. Instead, special numerical methods are needed for fast integration. The `ode15s`, `ode23s`, `ode23t`, and `ode23tb` functions can solve stiff problems efficiently.

This solution to the van der Pol equation for $\mu = 1000$ uses `ode15s` with the same initial conditions. You need to stretch out the time span drastically to $[0, 3000]$ to be able to see the periodic movement of the solution.

```
tspan = [0, 3000];
y0 = [2; 0];
Mu = 1000;
ode = @(t,y) vanderpoldemo(t,y,Mu);
[t,y] = ode15s(ode, tspan, y0);

plot(t,y(:,1))
title('van der Pol Equation, \mu = 1000')
axis([0 3000 -3 3])
xlabel('t')
ylabel('solution y')
```



Boundary Value Problems

`bvp4c` and `bvp5c` solve boundary value problems for ordinary differential equations.

The example function `twoode` has a differential equation written as a system of two first-order ODEs. The differential equation is

$$\frac{d^2y}{dt^2} + |y| = 0 .$$

```
type twoode
```

```
function dydx = twoode(x,y)
%TWOODE  Evaluate the differential equations for TWOBVP.
%
%   See also TWOBC, TWOBVP.

%   Lawrence F. Shampine and Jacek Kierzenka
%   Copyright 1984-2014 The MathWorks, Inc.

dydx = [ y(2); -abs(y(1)) ];
```

The function `twobc` has the boundary conditions for the problem: $y(0) = 0$ and $y(4) = -2$.

```
type twobc
```

```
function res = twobc(ya,yb)
%TWOBC  Evaluate the residual in the boundary conditions for TWOBVP.
```

```
%
%   See also TWOODE, TWOBVP.

%   Lawrence F. Shampine and Jacek Kierzenka
%   Copyright 1984-2014 The MathWorks, Inc.

res = [ ya(1); yb(1) + 2 ];
```

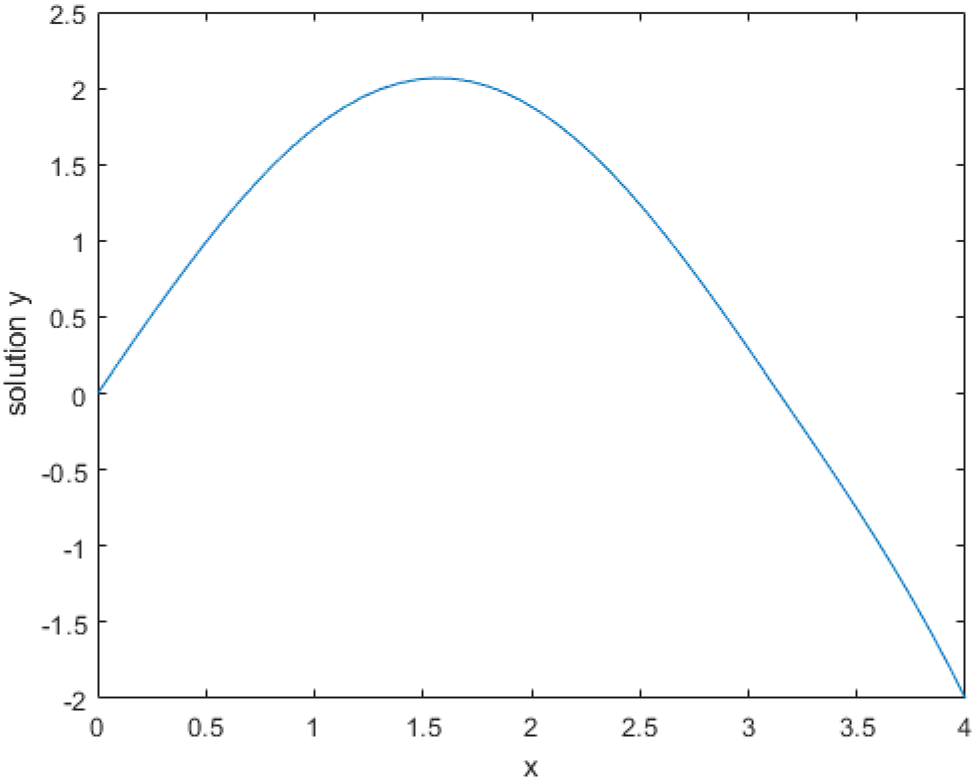
Prior to calling bvp4c, you have to provide a guess for the solution you want represented at a mesh. The solver then adapts the mesh as it refines the solution.

The bvpinit function assembles the initial guess in a form you can pass to the solver bvp4c. For a mesh of [0 1 2 3 4] and constant guesses of $y(x) = 1$ and $y'(x) = 0$, the call to bvpinit is:

```
solinit = bvpinit([0 1 2 3 4],[1; 0]);
```

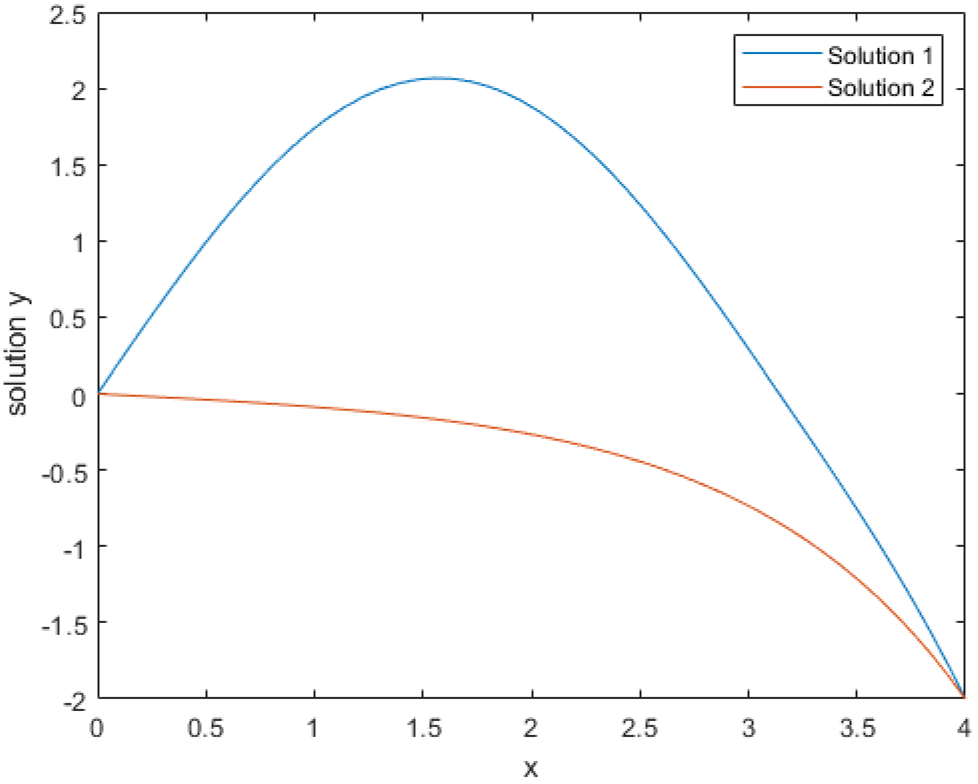
With this initial guess, you can solve the problem with bvp4c. Evaluate the solution returned by bvp4c at some points using deval, and then plot the resulting values.

```
sol = bvp4c(@twoode, @twobc, solinit);

xint = linspace(0, 4, 50);
yint = deval(sol, xint);
plot(xint, yint(1,:));
xlabel('x')
ylabel('solution y')
hold on
```



This particular boundary value problem has exactly two solutions. You can obtain the other solution by changing the initial guesses to $y(x) = -1$ and $y'(x) = 0$.

```
solinit = bvpinit([0 1 2 3 4],[-1; 0]);
sol = bvp4c(@twoode,@twobc,solinit);

xint = linspace(0,4,50);
yint = deval(sol,xint);
plot(xint,yint(1,:));
legend('Solution 1','Solution 2')
hold off
```



Delay Differential Equations

dde23, ddesd, and ddensd solve delay differential equations with various delays. The examples ddex1, ddex2, ddex3, ddex4, and ddex5 form a mini tutorial on using these solvers.

The ddex1 example shows how to solve the system of differential equations

$$y_1'(t) = y_1(t-1)$$
$$y_2'(t) = y_1(t-1) + y_2(t-0.2)$$
$$y_3'(t) = y_2(t).$$

You can represent these equations with the anonymous function

```
ddex1fun = @(t,y,Z) [Z(1,1); Z(1,1)+Z(2,2); y(2)];
```

The history of the problem (for $t \leq 0$) is constant:

$$y_1(t) = 1$$
$$y_2(t) = 1$$
$$y_3(t) = 1 .$$

You can represent the history as a vector of ones.

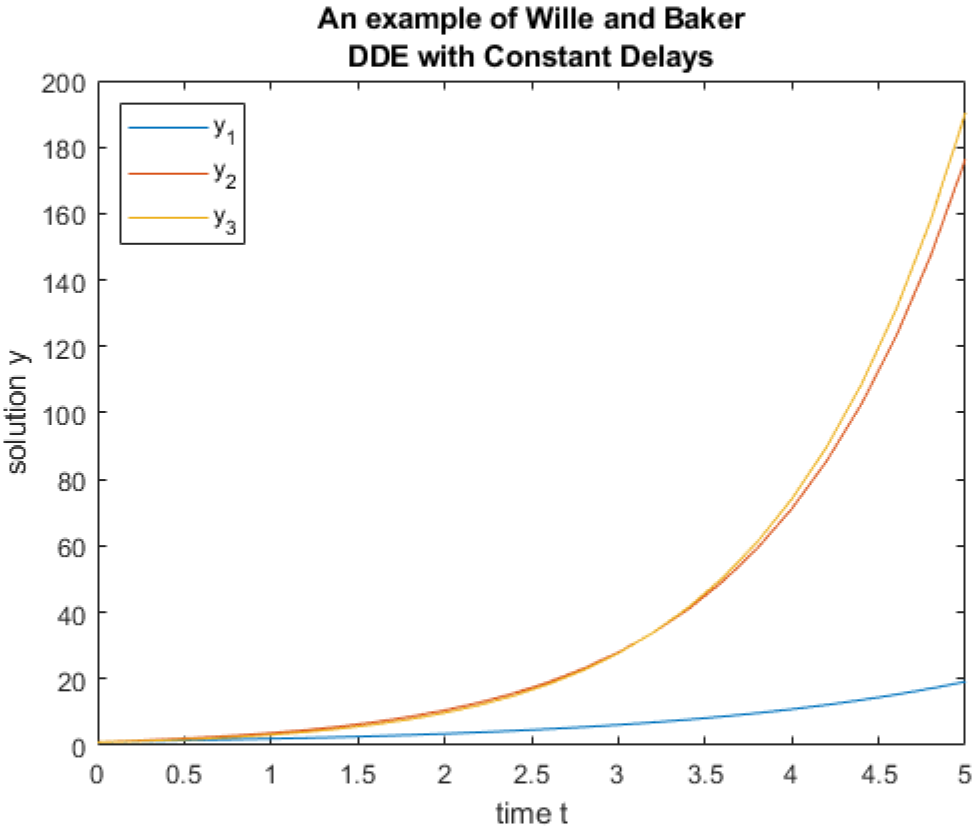
```
ddex1hist = ones(3,1);
```

A two-element vector represents the delays in the system of equations.

```
lags = [1 0.2];
```

Pass the function, delays, solution history, and interval of integration $[0, 5]$ to the solver as inputs. The solver produces a continuous solution over the whole interval of integration that is suitable for plotting.

```
sol = dde23(ddex1fun, lags, ddex1hist, [0 5]);

plot(sol.x,sol.y);
title({'An example of Wille and Baker', 'DDE with Constant Delays'});
xlabel('time t');
ylabel('solution y');
legend('y_1','y_2','y_3','Location','NorthWest');
```



Partial Differential Equations

pdepe solves partial differential equations in one space variable and time. The examples pdex1, pdex2, pdex3, pdex4, and pdex5 form a mini tutorial on using pdepe.

This example problem uses the functions pdex1pde, pdex1ic, and pdex1bc.

pdex1pde defines the differential equation

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\frac{\partial u}{\partial x}\right).$$

```
type pdex1pde
```

```
function [c,f,s] = pdex1pde(x,t,u,DuDx)
%PDEX1PDE  Evaluate the differential equations components for the PDEX1 problem.
%
%   See also PDEPE, PDEX1.

%   Lawrence F. Shampine and Jacek Kierzenka
%   Copyright 1984-2014 The MathWorks, Inc.

c = pi^2;
f = DuDx;
s = 0;
```

pdex1ic sets up the initial condition

$$u(x,0) = \sin \pi x \,.$$

```
type pdex1ic
```

```
function u0 = pdex1ic(x)
%PDEX1IC  Evaluate the initial conditions for the problem coded in PDEX1.
%
%   See also PDEPE, PDEX1.

%   Lawrence F. Shampine and Jacek Kierzenka
%   Copyright 1984-2014 The MathWorks, Inc.

u0 = sin(pi*x);
```

pdex1bc sets up the boundary conditions

$$u(0,t) = 0,$$

$$\pi e^{-t} + \frac{\partial}{\partial x} u(1,t) = 0 \,.$$

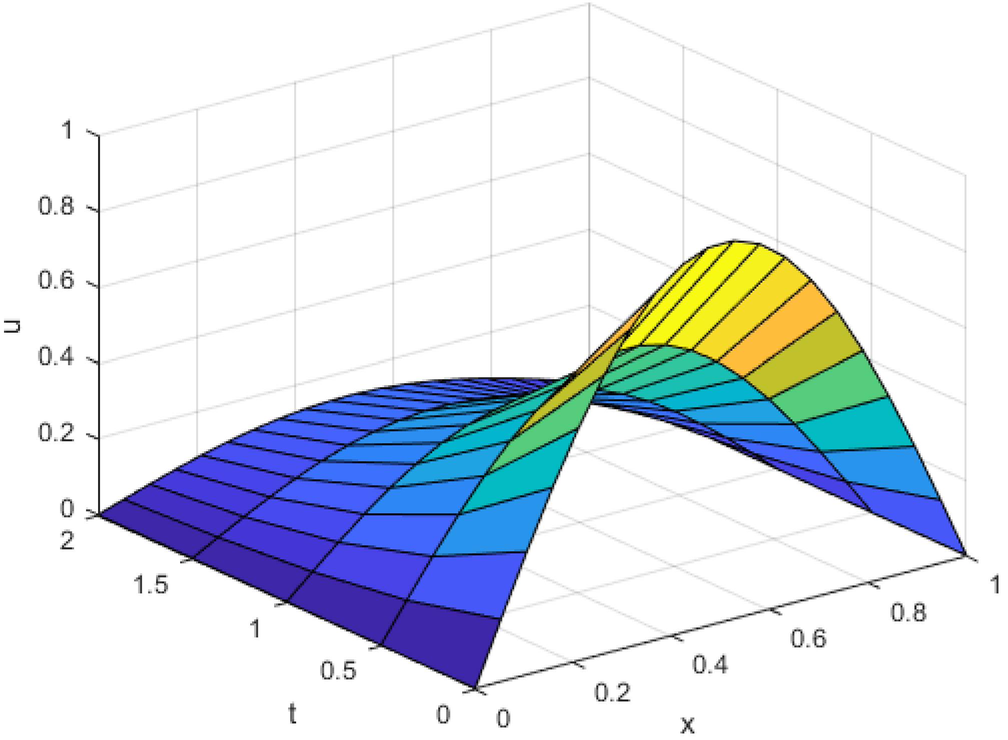
```
type pdex1bc
```

```
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
%PDEX1BC  Evaluate the boundary conditions for the problem coded in PDEX1.
%
%   See also PDEPE, PDEX1.

%   Lawrence F. Shampine and Jacek Kierzenka
%   Copyright 1984-2014 The MathWorks, Inc.
```

```
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
```

pdepe requires the spatial discretization x and a vector of times t (at which you want a snapshot of the solution). Solve the problem using a mesh of 20 nodes and request the solution at five values of t. Extract and plot the first component of the solution.

```
x = linspace(0,1,20);
t = [0 0.5 1 1.5 2];
sol = pdepe(0,@pdex1pde,@pdex1ic,@pdex1bc,x,t);

u1 = sol(:,:,1);

surf(x,t,u1);
xlabel('x');
ylabel('t');
zlabel('u');
```



See Also

bvp4c | ode45 | pdepe

More About

- “Choose an ODE Solver” on page 11-2
- “Solving Boundary Value Problems” on page 12-2
- “Solving Partial Differential Equations” on page 13-2

Solve Predator-Prey Equations

This example shows how to solve a differential equation representing a predator/prey model using both `ode23` and `ode45`. These functions are for the numerical solution of ordinary differential equations using variable step size Runge-Kutta integration methods. `ode23` uses a simple 2nd and 3rd order pair of formulas for medium accuracy and `ode45` uses a 4th and 5th order pair for higher accuracy.

Consider the pair of first-order ordinary differential equations known as the **Lotka-Volterra equations**, or **predator-prey model**:

$$\frac{\mathrm{dx}}{\mathrm{dt}} = x - \alpha \mathrm{xy}$$

$$\frac{\mathrm{dy}}{\mathrm{dt}} = -y + \beta \mathrm{xy}\,.$$

The variables x and y measure the sizes of the prey and predator populations, respectively. The quadratic cross term accounts for the interactions between the species. The prey population increases when no predators are present, and the predator population decreases when prey are scarce.

Code Equations

To simulate the system, create a function that returns a column vector of state derivatives, given state and time values. The two variables x and y can be represented in MATLAB as the first two values in a vector `y`. Similarly, the derivatives are the first two values in a vector `yp`. The function must accept values for `t` and `y` and return the values produced by the equations in `yp`.

```
yp(1) = (1 - alpha*y(2))*y(1)
```

```
yp(2) = (-1 + beta*y(1))*y(2)
```

In this example, the equations are contained in a file called `lotka.m`. This file uses parameter values of $\alpha = 0.01$ and $\beta = 0.02$.

```
type lotka
```

```
function yp = lotka(t,y)
%LOTKA  Lotka-Volterra predator-prey model.

%   Copyright 1984-2014 The MathWorks, Inc.

yp = diag([1 - .01*y(2), -1 + .02*y(1)])*y;
```

Simulate System

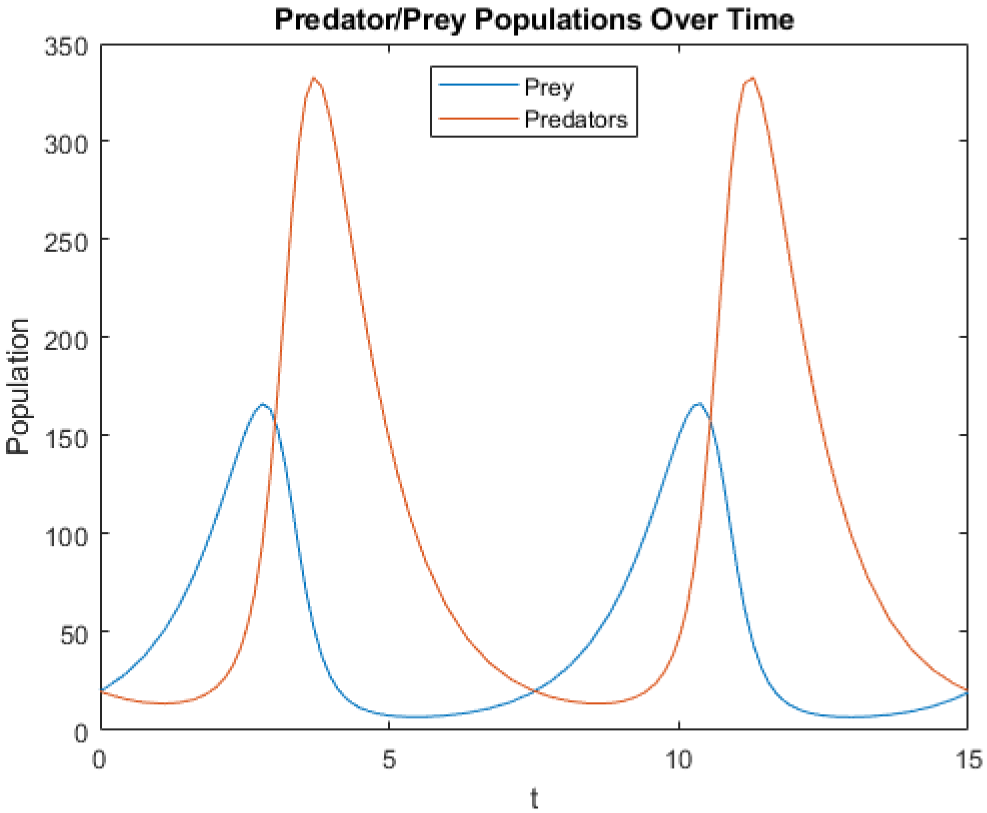
Use `ode23` to solve the differential equation defined in `lotka` over the interval $0 < t < 15$. Use an initial condition of $x(0) = y(0) = 20$ so that the populations of predators and prey are equal.

```
t0 = 0;
tfinal = 15;
y0 = [20; 20];
[t,y] = ode23(@lotka,[t0 tfinal],y0);
```

Plot Results

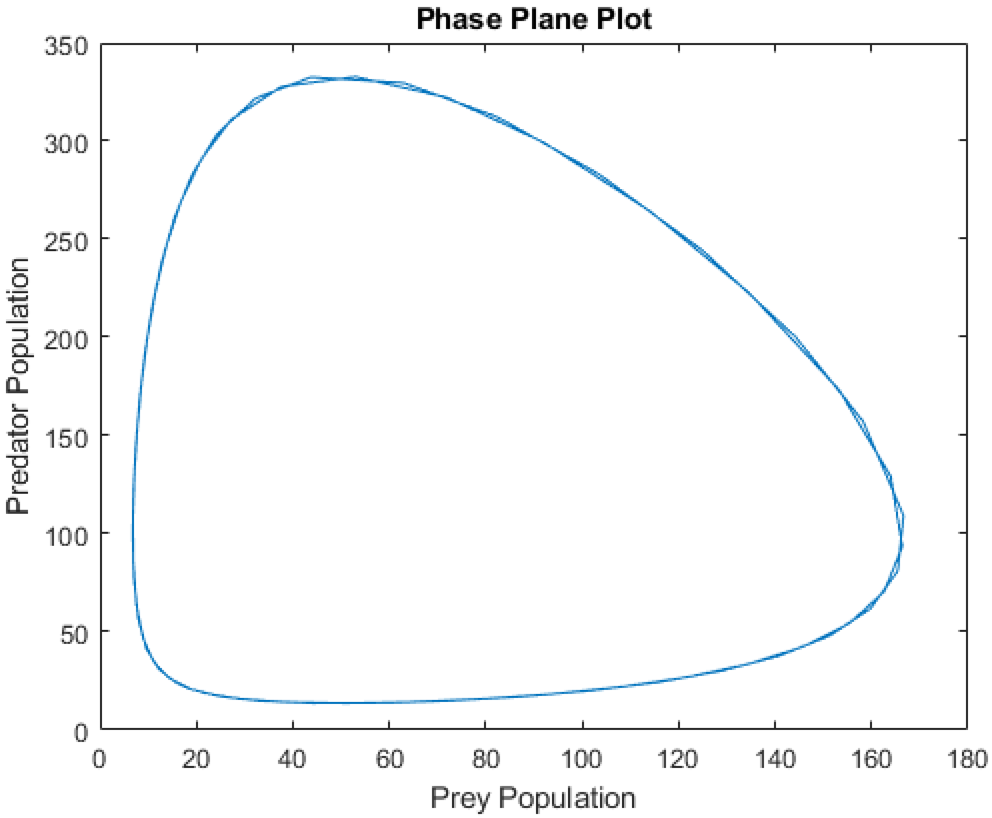
Plot the resulting populations against time.

```
plot(t,y)
title('Predator/Prey Populations Over Time')
xlabel('t')
ylabel('Population')
legend('Prey','Predators','Location','North')
```



Now plot the populations against each other. The resulting phase plane plot makes the cyclic relationship between the populations very clear.

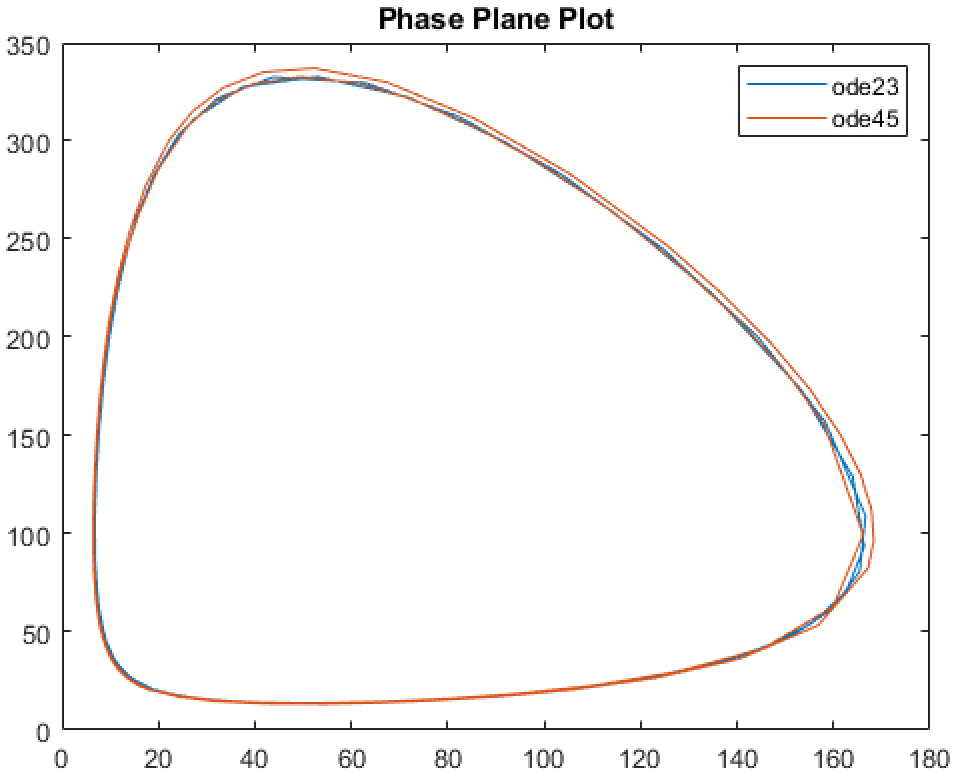
```
plot(y(:,1),y(:,2))
title('Phase Plane Plot')
xlabel('Prey Population')
ylabel('Predator Population')
```



Compare Results of Different Solvers

Solve the system a second time using `ode45`, instead of `ode23`. The `ode45` solver takes longer for each step, but it also takes larger steps. Nevertheless, the output of `ode45` is smooth because by default the solver uses a continuous extension formula to produce output at four equally spaced time points in the span of each step taken. (You can adjust the number of points with the `'Refine'` option.) Plot both solutions for comparison.

```
[T,Y] = ode45(@lotka,[t0 tfinal],y0);

plot(y(:,1),y(:,2),'-',Y(:,1),Y(:,2),'-');
title('Phase Plane Plot')
legend('ode23','ode45')
```



The results show that solving differential equations using different numerical methods can produce slightly different answers.

See Also

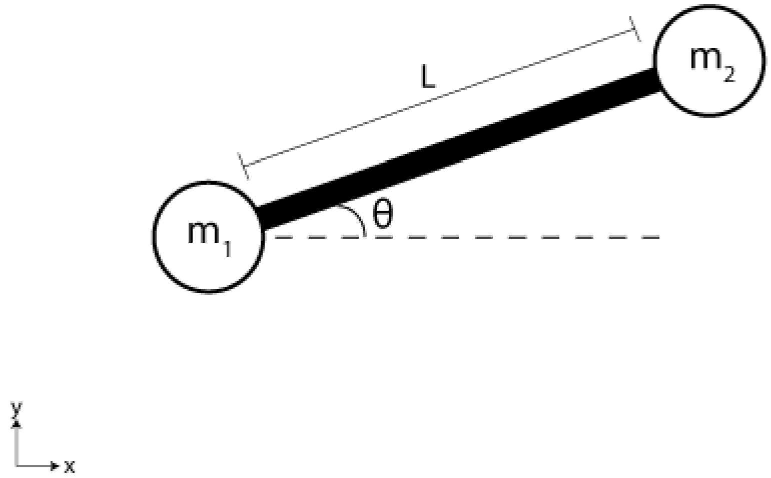
ode23 | ode45

More About

- “Choose an ODE Solver” on page 11-2
- “Solve Nonstiff ODEs” on page 11-18

Solve Equations of Motion for Baton Thrown into Air

This example solves a system of ordinary differential equations that model the dynamics of a baton thrown into the air [1]. The baton is modeled as two particles with masses m_1 and m_2 connected by a rod of length L. The baton is thrown into the air and subsequently moves in the vertical xy-plane subject to the force due to gravity. The rod forms an angle θ with the horizontal and the coordinates of the first mass are (x, y). With this formulation, the coordinates of the second mass are $(x + L\cos\theta, y + L\sin\theta)$.



The equations of motion for the system are obtained by applying Lagrange's equations for each of the three coordinates, x, y, and θ:

$$(m_1 + m_2)\ddot{x} - m_2 L\ddot{\theta}\sin\theta - m_2 L\dot{\theta}^2\cos\theta = 0,$$

$$(m_1 + m_2)\ddot{y} - m_2 L\ddot{\theta}\cos\theta - m_2 L\dot{\theta}^2\sin\theta + (m_1 + m_2)g = 0,$$

$$L^2\ddot{\theta} - L\ddot{x}\sin\theta + L\ddot{y}\cos\theta + g\,L\cos\theta = 0\,.$$

To solve this system of ODEs in MATLAB®, code the equations into a function before calling the solver `ode45`. You can either include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equations

The `ode45` solver requires the equations to be written in the form $\dot{q} = f(t, q)$, where $\dot{q}$ is the first derivative of each coordinate. In this problem, the solution vector has six components: x, y, the angle θ, and their first derivatives:

$$q = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \end{bmatrix} = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ \theta \\ \dot{\theta} \end{bmatrix}.$$

With this notation, you can rewrite the equations of motion entirely in terms of the elements of q:

$$\begin{aligned} &(m_1 + m_2)\dot{q}_2 - m_2 L\,\dot{q}_6 \sin q_5 = m_2 L\, q_6^2 \cos q_5, \\ &(m_1 + m_2)\dot{q}_4 - m_2 L\,\dot{q}_6 \cos q_5 = m_2 L\, q_6^2 \sin q_5 - (m_1 + m_2)g, \\ &L^2 \dot{q}_6 - L\,\dot{q}_2 \sin q_5 + L\,\dot{q}_4 \cos q_5 = -g\,L \cos q_5\,. \end{aligned}$$

Unfortunately, the equations of motion do not fit into the form $\dot{q} = f(t, q)$ required by the solver, since there are several terms on the left with first derivatives. When this occurs, you must use a mass matrix to represent the left side of the equation.

With matrix notation, you can rewrite the equations of motion as a system of six equations using a mass matrix in the form $M(t, q)\,\dot{q} = f(t, q)$. The mass matrix expresses the linear combinations of first derivatives on the left side of the equation with a matrix-vector product. In this form, the system of equations becomes:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & m_1 + m_2 & 0 & 0 & 0 & -m_2 L \sin q_5 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & m_1 + m_2 & 0 & m_2 L \cos q_5 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & -L \sin q_5 & 0 & L \cos q_5 & 0 & L^2 \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \\ \dot{q}_5 \\ \dot{q}_6 \end{bmatrix} = \begin{bmatrix} q_2 \\ m_2 L\, q_6^2 \cos q_5 \\ q_4 \\ m_2 L\, q_6^2 \sin q_5 - (m_1 + m_2)g \\ q_6 \\ -g\,L \cos q_5 \end{bmatrix}$$

From this expression, you can write a function that calculates the nonzero elements of the mass matrix. The function takes three inputs: t and the solution vector q are required (you must specify these inputs even if they are not used in the body of the function), and P is an optional extra input used to pass in parameter values. To pass the parameters for this problem to the function, create `P` as a structure that holds the parameter values and then use the technique described in “Parameterizing Functions” on page 10-2 to pass the structure to the function as an extra input.

```
function M = mass(t,q,P)
% Extract parameters
m1 = P.m1;
m2 = P.m2;
L = P.L;
g = P.g;

% Mass matrix function
M = zeros(6,6);
M(1,1) = 1;
M(2,2) = m1 + m2;
M(2,6) = -m2*L*sin(q(5));
```

```
M(3,3) = 1;
M(4,4) = m1 + m2;
M(4,6) = m2*L*cos(q(5));
M(5,5) = 1;
M(6,2) = -L*sin(q(5));
M(6,4) = L*cos(q(5));
M(6,6) = L^2;
end
```

Next, you can write a function for the right side of each of the equations in the system $M(t, q)\,\dot{q} = f(t, q)$. Like the mass matrix function, this function takes two required inputs for t and q, and one optional input for parameter values P.

```
function dydt = f(t,q,P)
% Extract parameters
m1 = P.m1;
m2 = P.m2;
L = P.L;
g = P.g;

% Equation to solve
dydt = [q(2)
        m2*L*q(6)^2*cos(q(5))
        q(4)
        m2*L*q(6)^2*sin(q(5))-(m1+m2)*g
        q(6)
        -g*L*cos(q(5))];
end
```

Solve System of Equations

First, create a structure P of parameter values for m_1, m_2, g, and L by setting appropriately named fields in a structure. The structure P is passed to the mass matrix and ODE functions as an extra input.

```
P.m1 = 0.1;
P.m2 = 0.1;
P.L = 1;
P.g = 9.81
```

```
P = struct with fields:
    m1: 0.1000
    m2: 0.1000
     L: 1
     g: 9.8100
```

Create a vector with 25 points between 0 and 4 for the time span of the integration. When you specify a vector of times, ode45 returns interpolated solutions at the requested times.

```
tspan = linspace(0,4,25);
```

Set the initial conditions of the problem. Since the baton is thrown upward at an angle, use nonzero values for the initial velocities, $\dot{x}_0 = 4$, $\dot{y}_0 = 20$, and $\dot{\theta}_0 = 2$. For the initial positions, the baton begins in an upright position, so $\theta_0 = -\pi/2$, $x_0 = 0$, and $y_0 = L$.

```
y0 = [0; 4; P.L; 20; -pi/2; 2];
```

Use odeset to create an options structure that references the mass matrix function. Since the solver expects the mass matrix function to have only two inputs, use an anonymous function to pass in the parameters P from the workspace.

```
opts = odeset('Mass',@(t,q) mass(t,q,P));
```

Finally, solve the system of equations using ode45 with these inputs:

- An anonymous function for the equations @(t,q) f(t,q,P)
- The vector tspan of times where the solution is requested
- The vector y0 of initial conditions
- The options structure opts that references the mass matrix

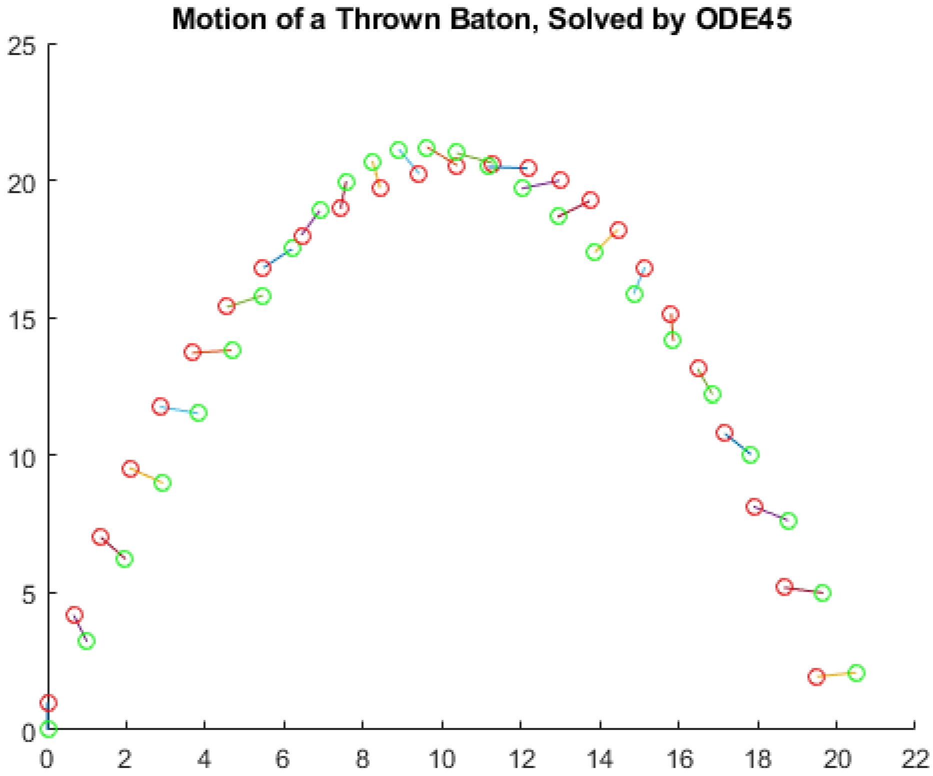
```
[t,q] = ode45(@(t,q) f(t,q,P),tspan,y0,opts);
```

Plot Results

The outputs from ode45 contain the solutions of the equations of motion at each requested time step. To examine the results, plot the motion of the baton over time.

Loop through each row of the solution, and at each time step, plot the position of the baton. Color each end of the baton differently so that you can see its rotation over time.

```
figure
title('Motion of a Thrown Baton, Solved by ODE45');
axis([0 22 0 25])
hold on
for j = 1:length(t)
   theta = q(j,5);
   X = q(j,1);
   Y = q(j,3);
   xvals = [X X+P.L*cos(theta)];
   yvals = [Y Y+P.L*sin(theta)];
   plot(xvals,yvals,xvals(1),yvals(1),'ro',xvals(2),yvals(2),'go')
end
hold off
```



References

[1] Wells, Dare A. *Schaum's Outline of Theory and Problems of Lagrangian Dynamics: With a Treatment of Euler's Equations of Motion, Hamilton's Equations and Hamilton's Principle*. Schaum's Outline Series. New York: Schaum Pub. Co, 1967.

Local Functions

Listed here are the local helper functions that the ODE solver calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydt = f(t,q,P) % Equations to solve
% Extract parameters
m1 = P.m1;
m2 = P.m2;
L = P.L;
g = P.g;

% RHS of system of equations
dydt = [q(2)
        m2*L*q(6)^2*cos(q(5))
        q(4)
        m2*L*q(6)^2*sin(q(5))-(m1+m2)*g
        q(6)
        -g*L*cos(q(5))];
end
%-------------------------------------------------
```

```
function M = mass(t,q,P) % Mass matrix function
% Extract parameters
m1 = P.m1;
m2 = P.m2;
L = P.L;
g = P.g;

% Set nonzero elements in mass matrix
M = zeros(6,6);
M(1,1) = 1;
M(2,2) = m1 + m2;
M(2,6) = -m2*L*sin(q(5));
M(3,3) = 1;
M(4,4) = m1 + m2;
M(4,6) = m2*L*cos(q(5));
M(5,5) = 1;
M(6,2) = -L*sin(q(5));
M(6,4) = L*cos(q(5));
M(6,6) = L^2;
end
```

See Also

ode45

More About

- “Choose an ODE Solver” on page 11-2
- “Solve Nonstiff ODEs” on page 11-18

Solve ODE with Strongly State-Dependent Mass Matrix

This example shows how to solve Burgers' equation using a moving mesh technique [1]. The problem includes a mass matrix, and options are specified to account for the strong state dependence and sparsity of the mass matrix, making the solution process more efficient.

Problem Setup

Burgers' equation is a convection-diffusion equation given by the PDE

$$\frac{\partial u}{\partial t} = \epsilon\frac{\partial^2 u}{\partial x^2} - \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right), \quad 0 < x < 1, \quad t > 0, \;\; \epsilon = 1\ \times 10^{-4}.$$

Applying a coordinate transformation (Eq. 18 in [1]) leads to an extra term on the left-hand side:

$$\frac{\partial u}{\partial t} - \frac{\partial u}{\partial x}\frac{\partial x}{\partial t} = \epsilon\frac{\partial^2 u}{\partial x^2} - \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right).$$

Converting the PDE into an ODE of one variable is accomplished by using finite differences to approximate the partial derivatives taken with respect to x. If the finite differences are written as Δ, then the PDE can be rewritten as an ODE that only contains derivatives taken with respect to t:

$$\frac{du}{dt} - \Delta u\frac{dx}{dt} = \epsilon\Delta^2 u - \Delta\left(\frac{u^2}{2}\right).$$

In this form, you can use an ODE solver such as `ode15s` to solve for u and x over time.

For this example, the problem is formulated on a *moving* mesh of N points, and the moving mesh technique described in [1] positions the mesh points at each time step so that they are concentrated in areas of change. The boundary and initial conditions are

$$u(0, t) = u(1, t) = 0,$$

$$u(x, 0) = \sin(2\pi x) + \frac{1}{2}\sin(\pi x).$$

For a given initial mesh of N points, there are $2N$ equations to solve: N equations corresponding to Burgers' equation, and N equations determining the movement of each mesh point. So, the final system of equations is:

$$\frac{du_1}{dt} - \Delta u_1\frac{dx_1}{dt} = \epsilon\Delta^2 u_1 - \Delta\left(\frac{{u_1}^2}{2}\right),$$

$$\vdots$$

$$\frac{du_N}{dt} - \Delta u_N\frac{dx_N}{dt} = \epsilon\Delta^2 u_N - \Delta\left(\frac{{u_N}^2}{2}\right),$$

$$\frac{d^2\dot{x}_1}{dt^2} = \frac{1}{\tau}\frac{d}{dt}\left(B(x_1, t)\frac{dx_1}{dt}\right),$$

$$\vdots$$

$$\frac{d^2\dot{x}_N}{dt^2} = \frac{1}{\tau}\frac{d}{dt}\left(B(x_N, t)\frac{dx_N}{dt}\right).$$

The terms for the moving mesh correspond to MMPDE6 in [1]. The parameter τ represents a timescale for forcing the mesh toward equidistribution. The term $B(x, t)$ is a monitor function given by Eq. 21 in [1]:

$$B(x, t) = \sqrt{1 + \left(\frac{du_i}{dx_i}\right)^2}\,.$$

The approach used in this example to solve Burgers' equation with moving mesh points demonstrates several techniques:

- The system of equations is expressed using a mass matrix formulation, $M\,y' = f(t, y)$. The mass matrix is provided to the `ode15s` solver as a function.
- The derivative function not only includes the equations for Burgers' equation, but also a set of equations governing the moving mesh selection.
- The sparsity patterns of the Jacobian dF/dy and the derivative of the mass matrix multiplied with a vector d(Mv)/dy are supplied to the solver as functions. Supplying these sparsity patterns helps the solver operate more efficiently.
- Finite differences are used to approximate several partial derivatives.

To solve this equation in MATLAB®, write a derivative function, a mass matrix function, a function for the sparsity pattern of the Jacobian dF/dy, and a function for the sparsity pattern of d(Mv)/dy. You can either include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Mass Matrix

The left side of the system of equations involves linear combinations of first derivatives, so a mass matrix is required to represent all of the terms. Set the left side of the system of equations equal to $M\,y'$ to extract the form of the mass matrix. The mass matrix is composed of four blocks, each of which is a square matrix of order N:

$$\begin{bmatrix} \frac{\partial u_1}{\partial t} - \frac{\partial u_1}{\partial x_1}\frac{\partial x_1}{\partial t} \\ \vdots \\ \frac{\partial u_N}{\partial t} - \frac{\partial u_N}{\partial x_N}\frac{\partial x_N}{\partial t} \\ \frac{\partial^2 \dot{x}_1}{\partial t^2} \\ \vdots \\ \frac{\partial^2 \dot{x}_N}{\partial t^2} \end{bmatrix} = M\,y' = \begin{bmatrix} M_1 & M_2 \\ M_3 & M_4 \end{bmatrix} \begin{bmatrix} \dot{u}_1 \\ \vdots \\ \dot{u}_N \\ \dot{x}_1 \\ \vdots \\ \dot{x}_N \end{bmatrix}.$$

This formulation shows that M_1 and M_2 form the left side of Burgers' equations (the first N equations in the system), while M_3 and M_4 form the left side of the mesh equations (the last N equations in the system). The block matrices are:

$$M_1 = I_N,$$

$$M_2 = -\frac{\partial u_i}{\partial x_i} I_N,$$

$$M_3 = 0_N,$$

$$M_4 = \frac{\partial^2}{\partial t^2} I_N\,.$$

I_N is the $N \times N$ identity matrix. The partial derivatives in M_2 are estimated using finite differences, while the partial derivative in M_4 uses a Laplacian matrix. Notice that M_3 contains only zeros because none of the equations for the mesh movement depend on $\dot{u}$.

Now you can write a function that computes the mass matrix. The function must accept two inputs for time t and the solution vector y. Since the solution vector y contains half $\dot{u}$ components and half $\dot{x}$ components, the function extracts these first. Then, the function forms all of the block matrices (taking the boundary values of the problem into account) and assembles the mass matrix using the four blocks.

```
function M = mass(t,y)
% Extract the components of y for the solution u and mesh x
N = length(y)/2;
u = y(1:N);
x = y(N+1:end);

% Boundary values of solution u and mesh x
u0 = 0;
uNP1 = 0;
x0 = 0;
xNP1 = 1;

% M1 and M2 are the portions of the mass matrix for Burgers' equation.
% The derivative du/dx is approximated with finite differences, using
% single-sided differences on the edges and centered differences in between.
M1 = speye(N);
M2 = sparse(N,N);
M2(1,1) = - (u(2) - u0)/(x(2) - x0);
for i = 2:N-1
    M2(i,i) = - (u(i+1) - u(i-1))/(x(i+1) - x(i-1));
end
M2(N,N) = - (uNP1 - u(N-1))/(xNP1 - x(N-1));

% M3 and M4 define the equations for mesh point evolution, corresponding to
% MMPDE6 in the reference paper. Since the mesh functions only involve d/dt(dx/dt),
% the M3 portion of the mass matrix is all zeros. The second derivative in M4 is
% approximated using a finite difference Laplacian matrix.
M3 = sparse(N,N);
e = ones(N,1);
M4 = spdiags([e -2*e e],-1:1,N,N);

% Assemble mass matrix
M = [M1 M2
     M3 M4];
end
```

Note: All functions are included as local functions at the end of the example.

Code Derivative Function

The derivative function for this problem returns a vector with $2N$ elements. The first N elements correspond to Burgers' equations, while the last N elements are for the moving mesh equations. The function `movingMeshODE` goes through these steps to evaluate the right-hand sides of all the equations in the system:

1. Evaluate Burgers' equations using finite differences (first N elements).
2. Evaluate monitor function (last N elements).
3. Apply spatial smoothing to monitor function and evaluate moving mesh equations.

The first N equations in the derivative function encode the right side of Burgers' equations. Burgers' equations can be considered as a differential operator involving spatial derivatives of the form:

$$f(u) = \epsilon \frac{\partial^2 u}{\partial x^2} - \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right).$$

The reference paper [1] describes the process of approximating the differential operator f using centered finite differences by

$$f_i = \epsilon \left[\frac{\left(\frac{u_{i+1} - u_i}{x_{i+1} - x_i}\right) - \left(\frac{u_i - u_{i-1}}{x_i - x_{i-1}}\right)}{\frac{1}{2}(x_{i+1} - x_{i-1})} \right] - \frac{1}{2}\left(\frac{u_{i+1}^2 - u_{i-1}^2}{x_{i+1} - x_{i-1}}\right).$$

On the edges of the mesh (for which $i = 1$ and $i = N$), only single-sided differences are used instead. This example uses $\epsilon = 1 \times 10^{-4}$.

The equations governing the mesh (comprising the last N equations in the derivative function) are

$$\frac{\partial^2 \dot{x}}{\partial t^2} = \frac{1}{\tau} \frac{\partial}{\partial t}\left(B(x,t)\frac{\partial x}{\partial t}\right).$$

Just as with Burgers' equations, you can use finite differences to approximate the monitor function $B(x,t)$:

$$B(x,t) = \sqrt{1 + \left(\frac{\partial u_i}{\partial x_i}\right)^2} = \sqrt{1 + \left(\frac{u_{i+1} - u_{i-1}}{x_{i+1} - x_{i-1}}\right)^2}.$$

Once the monitor function is evaluated, spatial smoothing is applied (Equations 14 and 15 in [1]). This example uses $\gamma = 2$ and $p = 2$ for the spatial smoothing parameters.

The function encoding the system of equations is

```
function g = movingMeshODE(t,y)
% Extract the components of y for the solution u and mesh x
N = length(y)/2;
u = y(1:N);
x = y(N+1:end);

% Boundary values of solution u and mesh x
u0 = 0;
uNP1 = 0;
x0 = 0;
```

```
xNP1 = 1;

% Preallocate g vector of derivative values.
g = zeros(2*N,1);

% Use centered finite differences to approximate the RHS of Burgers'
% equations (with single-sided differences on the edges). The first N
% elements in g correspond to Burgers' equations.
for i = 2:N-1
    delx = x(i+1) - x(i-1);
    g(i) = 1e-4*((u(i+1) - u(i))/(x(i+1) - x(i)) - ...
        (u(i) - u(i-1))/(x(i) - x(i-1)))/(0.5*delx) ...
        - 0.5*(u(i+1)^2 - u(i-1)^2)/delx;
end
delx = x(2) - x0;
g(1) = 1e-4*((u(2) - u(1))/(x(2) - x(1)) - (u(1) - u0)/(x(1) - x0))/(0.5*delx) ...
    - 0.5*(u(2)^2 - u0^2)/delx;
delx = xNP1 - x(N-1);
g(N) = 1e-4*((uNP1 - u(N))/(xNP1 - x(N)) - ...
    (u(N) - u(N-1))/(x(N) - x(N-1)))/delx - ...
    0.5*(uNP1^2 - u(N-1)^2)/delx;

% Evaluate the monitor function values (Eq. 21 in reference paper), used in
% RHS of mesh equations. Centered finite differences are used for interior
% points, and single-sided differences are used on the edges.
M = zeros(N,1);
for i = 2:N-1
    M(i) = sqrt(1 + ((u(i+1) - u(i-1))/(x(i+1) - x(i-1)))^2);
end
M0 = sqrt(1 + ((u(1) - u0)/(x(1) - x0))^2);
M(1) = sqrt(1 + ((u(2) - u0)/(x(2) - x0))^2);
M(N) = sqrt(1 + ((uNP1 - u(N-1))/(xNP1 - x(N-1)))^2);
MNP1 = sqrt(1 + ((uNP1 - u(N))/(xNP1 - x(N)))^2);

% Apply spatial smoothing (Eqns. 14 and 15) with gamma = 2, p = 2.
SM = zeros(N,1);
for i = 3:N-2
    SM(i) = sqrt((4*M(i-2)^2 + 6*M(i-1)^2 + 9*M(i)^2 + ...
        6*M(i+1)^2 + 4*M(i+2)^2)/29);
end
SM0 = sqrt((9*M0^2 + 6*M(1)^2 + 4*M(2)^2)/19);
SM(1) = sqrt((6*M0^2 + 9*M(1)^2 + 6*M(2)^2 + 4*M(3)^2)/25);
SM(2) = sqrt((4*M0^2 + 6*M(1)^2 + 9*M(2)^2 + 6*M(3)^2 + 4*M(4)^2)/29);
SM(N-1) = sqrt((4*M(N-3)^2 + 6*M(N-2)^2 + 9*M(N-1)^2 + 6*M(N)^2 + 4*MNP1^2)/29);
SM(N) = sqrt((4*M(N-2)^2 + 6*M(N-1)^2 + 9*M(N)^2 + 6*MNP1^2)/25);
SMNP1 = sqrt((4*M(N-1)^2 + 6*M(N)^2 + 9*MNP1^2)/19);
for i = 2:N-1
    g(i+N) = (SM(i+1) + SM(i))*(x(i+1) - x(i)) - ...
        (SM(i) + SM(i-1))*(x(i) - x(i-1));
end
g(1+N) = (SM(2) + SM(1))*(x(2) - x(1)) - (SM(1) + SM0)*(x(1) - x0);
g(N+N) = (SMNP1 + SM(N))*(xNP1 - x(N)) - (SM(N) + SM(N-1))*(x(N) - x(N-1));

% Form final discrete approximation for Eq. 12 in reference paper, the equation governing
% the mesh points.
tau = 1e-3;
g(1+N:end) = - g(1+N:end)/(2*tau);
end
```

Code Functions for Sparsity Patterns

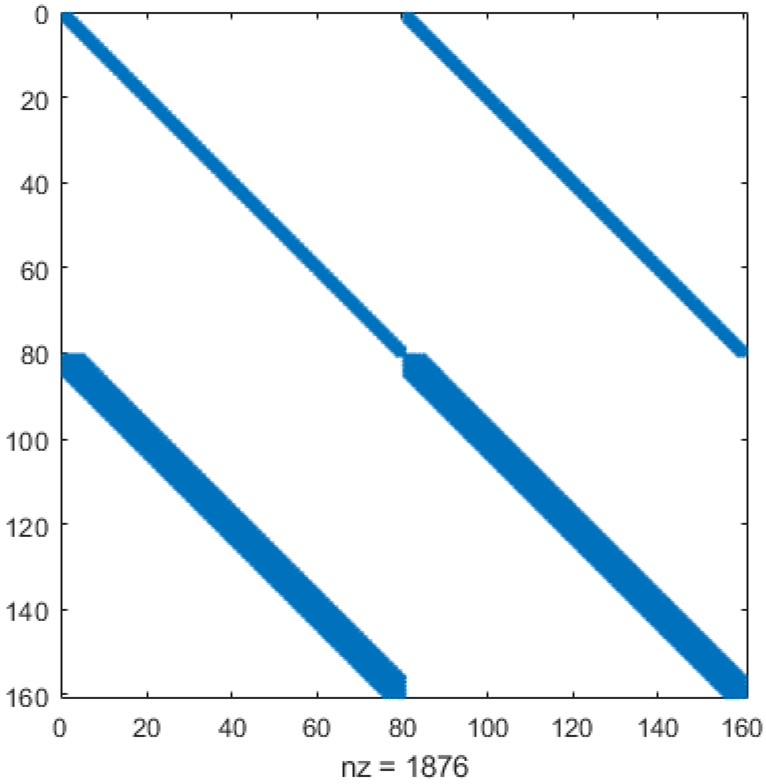
The Jacobian dF/dy for the derivative function is a $2N \times 2N$ matrix containing all of the partial derivatives of the derivative function, `movingMeshODE`. `ode15s` estimates the Jacobian using finite differences when the matrix is not supplied in the options structure. You can supply the sparsity pattern of the Jacobian to help `ode15s` calculate it more quickly.

The function for the sparsity pattern of the Jacobian is

```
function out = JPat(N)
S1 = spdiags(ones(N,3),-1:1,N,N);
S2 = spdiags(ones(N,9),-4:4,N,N);
out = [S1 S1
       S2 S2];
end
```

Plot the sparsity pattern of dF/dy for $N = 80$ using `spy`.

```
spy(JPat(80))
```



Another way to make the calculation more efficient is to provide the sparsity pattern of d(Mv)/dy. You can find this sparsity pattern by examining which terms of u_i and x_i are present in the finite differences calculated in the mass matrix function.

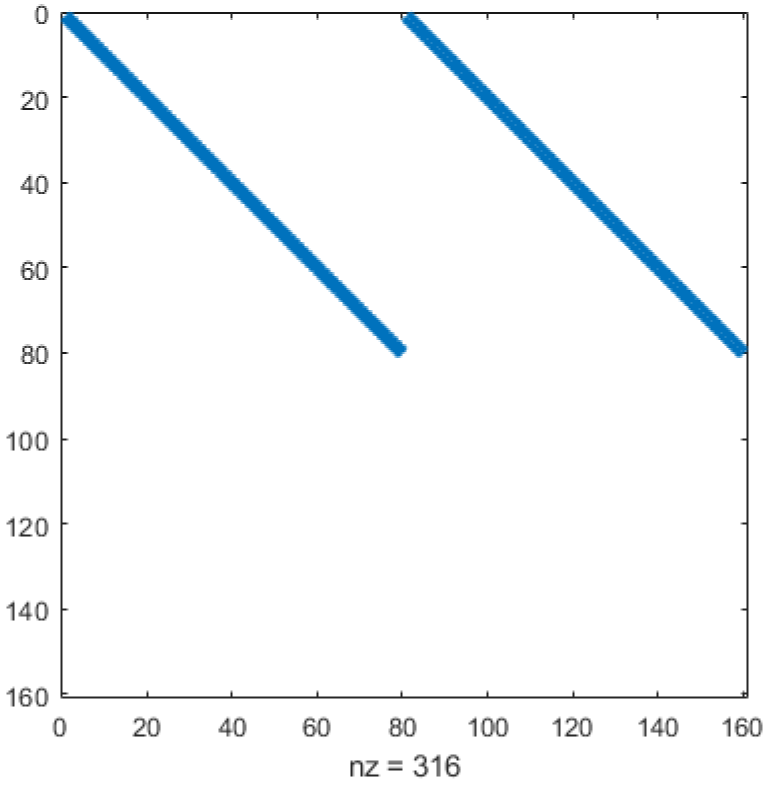
The function for the sparsity pattern of d(Mv)/dy is

```
function S = MvPat(N)
S = sparse(2*N,2*N);
```

```
S(1,2) = 1;
S(1,2+N) = 1;
for i = 2:N-1
    S(i,i-1) = 1;
    S(i,i+1) = 1;
    S(i,i-1+N) = 1;
    S(i,i+1+N) = 1;
end
S(N,N-1) = 1;
S(N,N-1+N) = 1;
end
```

Plot the sparsity pattern of $d(\mathrm{Mv})/\mathrm{dy}$ for $N = 80$ using `spy`.

```
spy(MvPat(80))
```



Solve System of Equations

Solve the system with the value $N = 80$. For the initial conditions, initialize x with a uniform grid and evaluate $u(x, 0)$ on the grid.

```
N = 80;
h = 1/(N+1);
xinit = h*(1:N);
uinit = sin(2*pi*xinit) + 0.5*sin(pi*xinit);
y0 = [uinit xinit];
```

Use `odeset` to create an options structure that sets several values:

- A function handle for the mass matrix
- The state-dependence of the mass matrix, which for this problem is `'strong'` since the mass matrix is a function of both *t* and *y*
- A function handle that calculates the Jacobian sparsity pattern
- A function handle that calculates the sparsity pattern of the derivative of the mass matrix multiplied by a vector
- The absolute and relative error tolerances

```
opts = odeset('Mass',@mass,'MStateDependence','strong','JPattern',JPat(N),...
   'MvPattern',MvPat(N),'RelTol',1e-5,'AbsTol',1e-4);
```

Finally, call `ode15s` to solve the system on the interval [0, 1] using the `movingMeshODE` derivative function, the time span, the initial conditions, and the options structure.

```
tspan = [0 1];
sol = ode15s(@movingMeshODE,tspan,y0,opts);
```

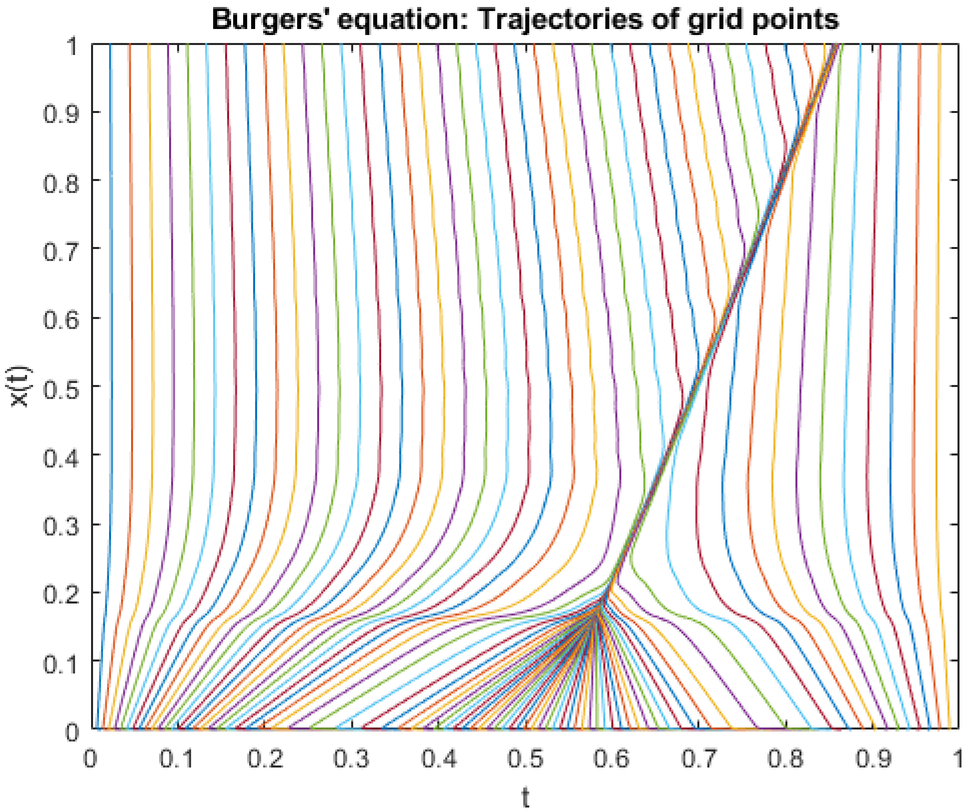
Plot Results

The result of the integration is a structure `sol` that contains the time steps *t*, the mesh points $x(t)$, and the solution $u(x,t)$. Extract these values from the structure.

```
t = sol.x;
x = sol.y(N+1:end,:);
u = sol.y(1:N,:);
```

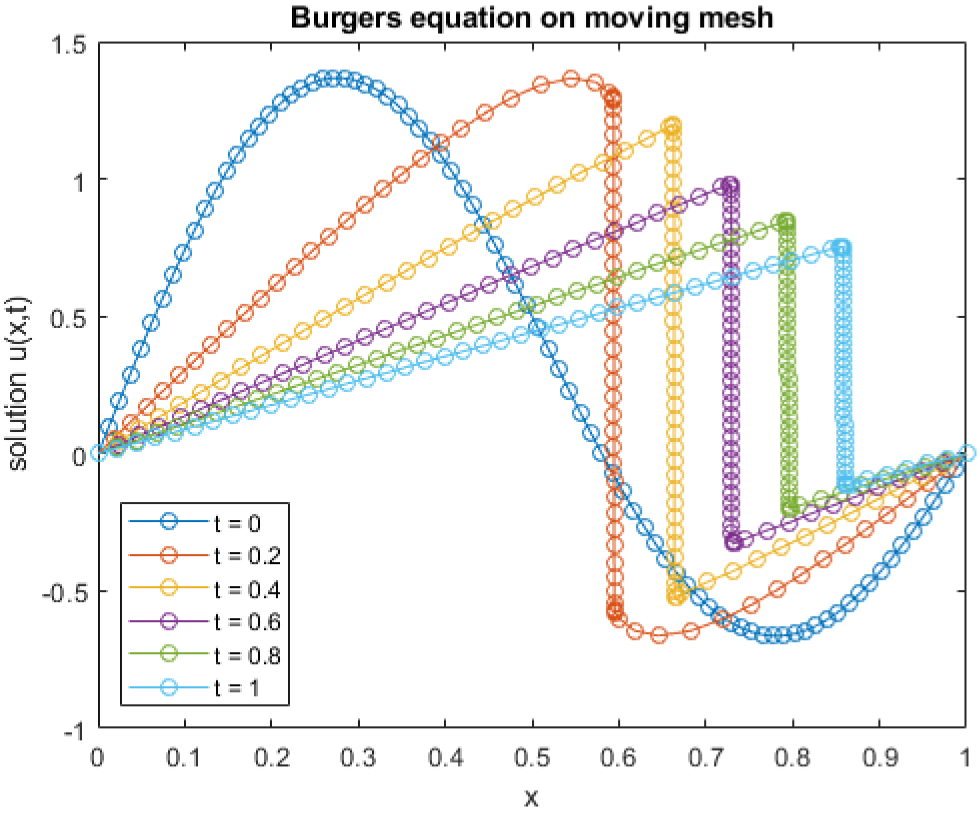
Plot the movement of the mesh points over time. The plot shows that the mesh points retain a reasonably even spacing over time (due to the monitor function), but they are able to cluster near the discontinuity in the solution as it moves.

```
plot(x,t)
xlabel('t')
ylabel('x(t)')
title('Burgers'' equation: Trajectories of grid points')
```



Now, sample $u(x, t)$ at a few values of t and plot the evolution of the solution over time. The mesh points at the ends of the interval are fixed, so `x(0) = 0` and `x(N+1) = 1`. The boundary values are `u(t,0) = 0` and `u(t,1) = 0`, which you must add to the known values computed for the figure.

```
tint = 0:0.2:1;
yint = deval(sol,tint);
figure
labels = {};
for j = 1:length(tint)
   solution = [0; yint(1:N,j); 0];
   location = [0; yint(N+1:end,j); 1];
   labels{j} = ['t = ' num2str(tint(j))];
   plot(location,solution,'-o')
   hold on
end
xlabel('x')
ylabel('solution u(x,t)')
legend(labels{:},'Location','SouthWest')
title('Burgers equation on moving mesh')
hold off
```



The plot shows that $u(x, 0)$ is a smooth wave that develops a steep gradient over time as it moves towards $x = 1$. The mesh points track the movement of the discontinuity so that extra evaluation points are in the appropriate position in each time step.

References

[1] Huang, Weizhang, et al. “Moving Mesh Methods Based on Moving Mesh Partial Differential Equations.” *Journal of Computational Physics*, vol. 113, no. 2, Aug. 1994, pp. 279–90. https://doi.org/10.1006/jcph.1994.1135.

Local Functions

Listed here are the local helper functions that the solver `ode15s` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function g = movingMeshODE(t,y)
% Extract the components of y for the solution u and mesh x
N = length(y)/2;
u = y(1:N);
x = y(N+1:end);

% Boundary values of solution u and mesh x
u0 = 0;
uNP1 = 0;
x0 = 0;
xNP1 = 1;
```

```
% Preallocate g vector of derivative values.
g = zeros(2*N,1);

% Use centered finite differences to approximate the RHS of Burgers'
% equations (with single-sided differences on the edges). The first N
% elements in g correspond to Burgers' equations.
for i = 2:N-1
    delx = x(i+1) - x(i-1);
    g(i) = 1e-4*((u(i+1) - u(i))/(x(i+1) - x(i)) - ...
        (u(i) - u(i-1))/(x(i) - x(i-1)))/(0.5*delx) ...
        - 0.5*(u(i+1)^2 - u(i-1)^2)/delx;
end
delx = x(2) - x0;
g(1) = 1e-4*((u(2) - u(1))/(x(2) - x(1)) - (u(1) - u0)/(x(1) - x0))/(0.5*delx) ...
    - 0.5*(u(2)^2 - u0^2)/delx;
delx = xNP1 - x(N-1);
g(N) = 1e-4*((uNP1 - u(N))/(xNP1 - x(N)) - ...
    (u(N) - u(N-1))/(x(N) - x(N-1)))/delx - ...
    0.5*(uNP1^2 - u(N-1)^2)/delx;

% Evaluate the monitor function values (Eq. 21 in reference paper), used in
% RHS of mesh equations. Centered finite differences are used for interior
% points, and single-sided differences are used on the edges.
M = zeros(N,1);
for i = 2:N-1
    M(i) = sqrt(1 + ((u(i+1) - u(i-1))/(x(i+1) - x(i-1)))^2);
end
M0 = sqrt(1 + ((u(1) - u0)/(x(1) - x0))^2);
M(1) = sqrt(1 + ((u(2) - u0)/(x(2) - x0))^2);
M(N) = sqrt(1 + ((uNP1 - u(N-1))/(xNP1 - x(N-1)))^2);
MNP1 = sqrt(1 + ((uNP1 - u(N))/(xNP1 - x(N)))^2);

% Apply spatial smoothing (Eqns. 14 and 15) with gamma = 2, p = 2.
SM = zeros(N,1);
for i = 3:N-2
    SM(i) = sqrt((4*M(i-2)^2 + 6*M(i-1)^2 + 9*M(i)^2 + ...
        6*M(i+1)^2 + 4*M(i+2)^2)/29);
end
SM0 = sqrt((9*M0^2 + 6*M(1)^2 + 4*M(2)^2)/19);
SM(1) = sqrt((6*M0^2 + 9*M(1)^2 + 6*M(2)^2 + 4*M(3)^2)/25);
SM(2) = sqrt((4*M0^2 + 6*M(1)^2 + 9*M(2)^2 + 6*M(3)^2 + 4*M(4)^2)/29);
SM(N-1) = sqrt((4*M(N-3)^2 + 6*M(N-2)^2 + 9*M(N-1)^2 + 6*M(N)^2 + 4*MNP1^2)/29);
SM(N) = sqrt((4*M(N-2)^2 + 6*M(N-1)^2 + 9*M(N)^2 + 6*MNP1^2)/25);
SMNP1 = sqrt((4*M(N-1)^2 + 6*M(N)^2 + 9*MNP1^2)/19);
for i = 2:N-1
    g(i+N) = (SM(i+1) + SM(i))*(x(i+1) - x(i)) - ...
        (SM(i) + SM(i-1))*(x(i) - x(i-1));
end
g(1+N) = (SM(2) + SM(1))*(x(2) - x(1)) - (SM(1) + SM0)*(x(1) - x0);
g(N+N) = (SMNP1 + SM(N))*(xNP1 - x(N)) - (SM(N) + SM(N-1))*(x(N) - x(N-1));

% Form final discrete approximation for Eq. 12 in reference paper, the equation governing
% the mesh points.
tau = 1e-3;
g(1+N:end) = - g(1+N:end)/(2*tau);
end

% ----------------------------------------------------------------------
```

```
function M = mass(t,y)
% Extract the components of y for the solution u and mesh x
N = length(y)/2;
u = y(1:N);
x = y(N+1:end);

% Boundary values of solution u and mesh x
u0 = 0;
uNP1 = 0;
x0 = 0;
xNP1 = 1;

% M1 and M2 are the portions of the mass matrix for Burgers' equation.
% The derivative du/dx is approximated with finite differences, using
% single-sided differences on the edges and centered differences in between.
M1 = speye(N);
M2 = sparse(N,N);
M2(1,1) = - (u(2) - u0)/(x(2) - x0);
for i = 2:N-1
    M2(i,i) = - (u(i+1) - u(i-1))/(x(i+1) - x(i-1));
end
M2(N,N) = - (uNP1 - u(N-1))/(xNP1 - x(N-1));

% M3 and M4 define the equations for mesh point evolution, corresponding to
% MMPDE6 in the reference paper. Since the mesh functions only involve d/dt(dx/dt),
% the M3 portion of the mass matrix is all zeros. The second derivative in M4 is
% approximated using a finite difference Laplacian matrix.
M3 = sparse(N,N);
e = ones(N,1);
M4 = spdiags([e -2*e e],-1:1,N,N);

% Assemble mass matrix
M = [M1 M2
     M3 M4];
end

% ------------------------------------------------------------------------
function out = JPat(N)  % Jacobian sparsity pattern
S1 = spdiags(ones(N,3),-1:1,N,N);
S2 = spdiags(ones(N,9),-4:4,N,N);
out = [S1 S1
    S2 S2];
end

% ------------------------------------------------------------------------
function S = MvPat(N)  % Sparsity pattern for the derivative of the Mass matrix times a vector
S = sparse(2*N,2*N);
S(1,2) = 1;
S(1,2+N) = 1;
for i = 2:N-1
    S(i,i-1) = 1;
    S(i,i+1) = 1;
    S(i,i-1+N) = 1;
    S(i,i+1+N) = 1;
end
S(N,N-1) = 1;
S(N,N-1+N) = 1;
```

```
end
% ------------------------------------------------------------------------
```

See Also

`ode15s` | `odeset`

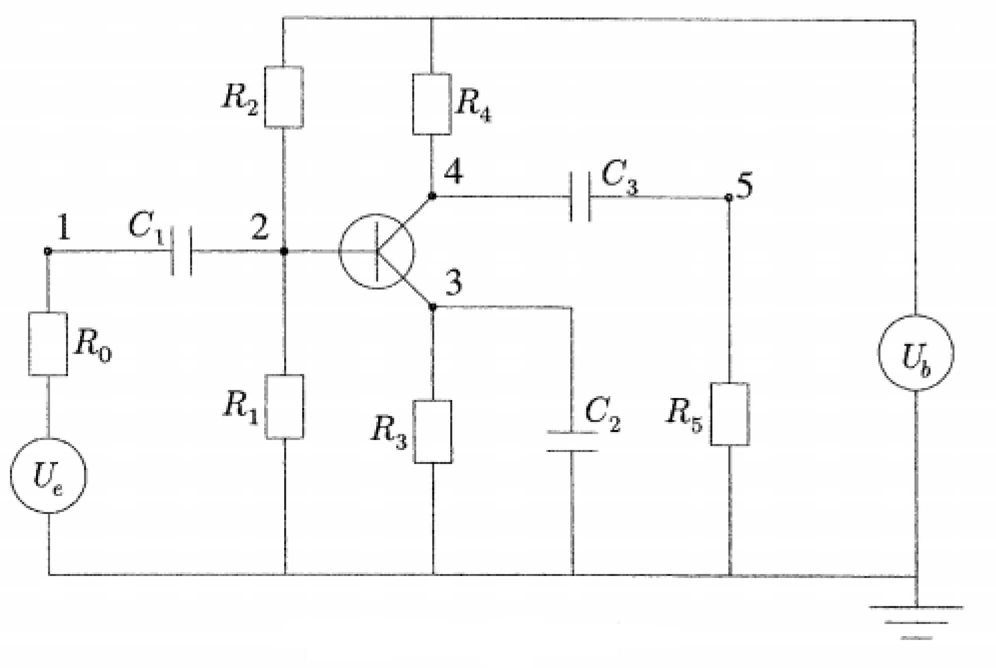
More About

- "Choose an ODE Solver" on page 11-2
- "Summary of ODE Options" on page 11-10
- "Solve Equations of Motion for Baton Thrown into Air" on page 11-55

Solve Stiff Transistor Differential Algebraic Equation

This example shows how to use `ode23t` to solve a stiff differential algebraic equation (DAE) that describes an electrical circuit [1]. The one-transistor amplifier problem coded in the example file amp1dae.m can be rewritten in semi-explicit form, but this example solves it in its original form $Mu' = \phi(u)$. The problem includes a constant, singular mass matrix M.

The transistor amplifier circuit contains six resistors, three capacitors, and a transistor.



- The initial voltage signal is $U_e(t) = 0.4\sin(200\pi t)$.
- The operating voltage is $U_b = 6$.
- The voltages at the nodes are given by $U_i(t)\ (i = 1, 2, 3, 4, 5)$.
- The values of the resistors $R_i\ (i = 1, 2, 3, 4, 5, 6)$ are constant, and the current through each resistor satisfies $I = U/R$.
- The values of the capacitors $C_i\ (i = 1, 2, 3)$ are constant, and the current through each capacitor satisfies $I = C \cdot \mathrm{dU/dt}$.

The goal is to solve for the output voltage through node 5, $U_5(t)$.

To solve this equation in MATLAB®, you need to code the equations, code a mass matrix, and set the initial conditions and interval of integration before calling the solver `ode23t`. You can either include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Mass Matrix

Using Kirchoff's law to equalize the current through each node (1 through 5), you can obtain a system of five equations describing the circuit:

$$\begin{aligned}
&\text{node 1}: \ \frac{U_e(t)}{R_0} - \frac{U_1}{R_0} + C_1(U_2{}' - U_1{}') = 0, \\
&\text{node 2}: \ \frac{U_b}{R_2} - U_2\left(\frac{1}{R_1} + \frac{1}{R_2}\right) + C_1(U_1{}' - U_2{}') - 0.01f(U_2 - U_3) = 0, \\
&\text{node 3}: \ f(U_2 - U_3) - \frac{U_3}{R_3} - C_2U_3{}' = 0, \\
&\text{node 4}: \ \frac{U_b}{R_4} - \frac{U_4}{R_4} + C_3(U_5{}' - U_4{}') - 0.99f(U_2 - U_3) = 0, \\
&\text{node 5}: \ -\frac{U_5}{R_5} + C_3(U_4{}' - U_5{}') = 0\,.
\end{aligned}$$

The mass matrix of this system, found by collecting the derivative terms on the left side of the equations, has the form

$$M = \begin{pmatrix} -c_1 & c_1 & 0 & 0 & 0 \\ c_1 & -c_1 & 0 & 0 & 0 \\ 0 & 0 & -c_2 & 0 & 0 \\ 0 & 0 & 0 & -c_3 & c_3 \\ 0 & 0 & 0 & c_3 & -c_3 \end{pmatrix},$$

where $c_k = k \times 10^{-6}$ for $k = 1, 2, 3$.

Create a mass matrix with the appropriate constants c_k, and then use the `odeset` function to specify the mass matrix. Even though it is apparent that the mass matrix is singular, leave the `'MassSingular'` option at its default value of `'maybe'` to test the automatic detection of a DAE problem by the solver.

```
c = 1e-6 * (1:3);
M = zeros(5,5);
M(1,1) = -c(1);
M(1,2) =  c(1);
M(2,1) =  c(1);
M(2,2) = -c(1);
M(3,3) = -c(2);
M(4,4) = -c(3);
M(4,5) =  c(3);
M(5,4) =  c(3);
M(5,5) = -c(3);
opts = odeset('Mass',M);
```

Code Equations

The function `transampdae` contains the system of equations for this example. The function defines values for all of the voltages and constant parameters. The derivatives gathered on the left side of the equations are coded in the mass matrix, and `transampdae` codes the right side of the equations.

```
function dudt = transampdae(t,u)
% Define voltages and parameters
Ue = @(t) 0.4*sin(200*pi*t);
Ub = 6;
R0 = 1000;
```

```
R15 = 9000;
alpha = 0.99;
beta = 1e-6;
Uf = 0.026;

% Define system of equations
f23 = beta*(exp((u(2) - u(3))/Uf) - 1);
dudt = [ -(Ue(t) - u(1))/R0
    -(Ub/R15 - u(2)*2/R15 - (1-alpha)*f23)
    -(f23 - u(3)/R15)
    -((Ub - u(4))/R15 - alpha*f23)
    (u(5)/R15) ];
end
```

Note: This function is included as a local function at the end of the example.

Code Initial Conditions

Set the initial conditions. This example uses the consistent initial conditions for the current through each node computed in [1].

```
Ub = 6;
u0(1) = 0;
u0(2) = Ub/2;
u0(3) = Ub/2;
u0(4) = Ub;
u0(5) = 0;
```

Solve System of Equations

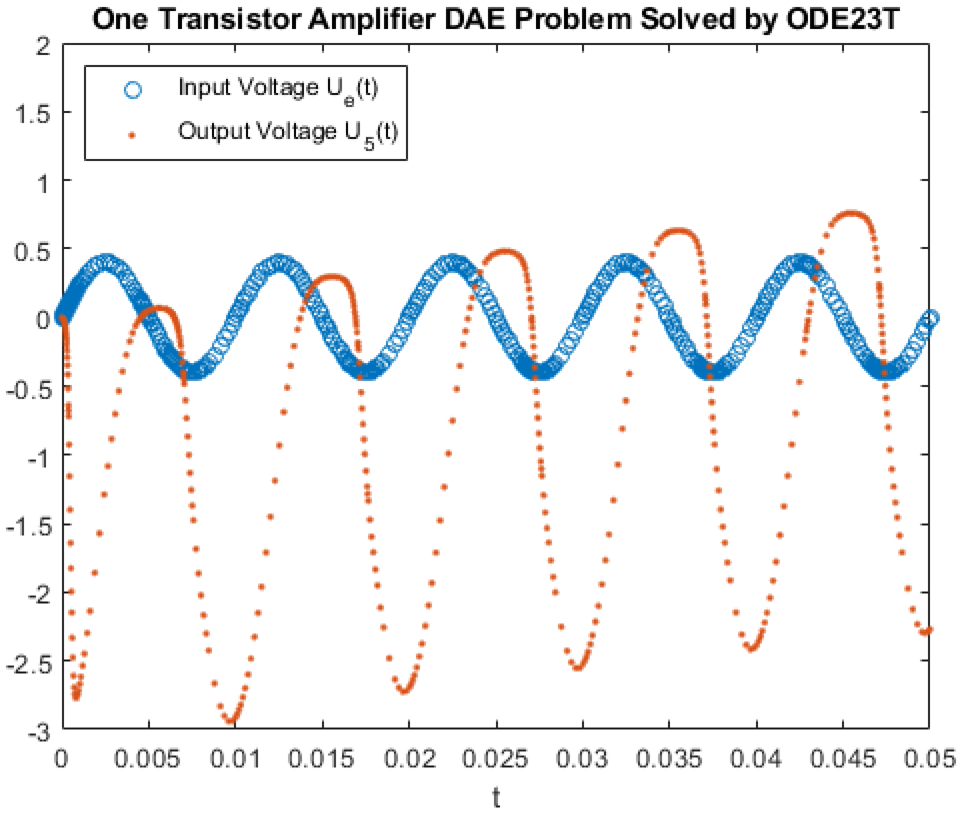
Solve the DAE system over the time interval `[0 0.05]` using `ode23t`.

```
tspan = [0 0.05];
[t,u] = ode23t(@transampdae,tspan,u0,opts);
```

Plot Results

Plot the initial voltage $U_e(t)$ and output voltage $U_5(t)$.

```
Ue = @(t) 0.4*sin(200*pi*t);
plot(t,Ue(t),'o',t,u(:,5),'.')
axis([0 0.05 -3 2]);
legend('Input Voltage U_e(t)','Output Voltage U_5(t)','Location','NorthWest');
title('One Transistor Amplifier DAE Problem Solved by ODE23T');
xlabel('t');
```



References

[1] Hairer, E., and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer Berlin Heidelberg, 1991, p. 377.

Local Functions

Listed here is the local helper function that the ODE solver `ode23t` calls to calculate the solution. Alternatively, you can save this function as its own file in a directory on the MATLAB path.

```
function dudt = transampdae(t,u)
% Define voltages and parameters
Ue = @(t) 0.4*sin(200*pi*t);
Ub = 6;
R0 = 1000;
R15 = 9000;
alpha = 0.99;
beta = 1e-6;
Uf = 0.026;

% Define system of equations
f23 = beta*(exp((u(2) - u(3))/Uf) - 1);
dudt = [ -(Ue(t) - u(1))/R0
    -(Ub/R15 - u(2)*2/R15 - (1-alpha)*f23)
    -(f23 - u(3)/R15)
    -((Ub - u(4))/R15 - alpha*f23)
    (u(5)/R15) ];
end
```

See Also

ode15s | ode23t

More About

- “Choose an ODE Solver” on page 11-2
- “Summary of ODE Options” on page 11-10
- “Solve Differential Algebraic Equations (DAEs)” on page 11-29
- “Solve ODE with Strongly State-Dependent Mass Matrix” on page 11-61

Solve System of ODEs with Multiple Initial Conditions

This example compares two techniques to solve a system of ordinary differential equations with multiple sets of initial conditions. The techniques are:

- Use a `for`-loop to perform several simulations, one for each set of initial conditions. This technique is simple to use but does not offer the best performance for large systems.
- Vectorize the ODE function to solve the system of equations for all sets of initial conditions simultaneously. This technique is the faster method for large systems but requires rewriting the ODE function so that it reshapes the inputs properly.

The equations used to demonstrate these techniques are the well-known Lotka-Volterra equations, which are first-order nonlinear differential equations that describe the populations of predators and prey.

Problem Description

The Lotka-Volterra equations are a system of two first-order, nonlinear ODEs that describe the populations of predators and prey in a biological system. Over time, the populations of the predators and prey change according to the equations

$$\frac{\mathrm{dx}}{\mathrm{dt}} = \alpha x - \beta \mathrm{xy},$$

$$\frac{\mathrm{dy}}{\mathrm{dt}} = \delta \mathrm{xy} - \gamma y\,.$$

The variables in these equations are

- x is the population size of the prey
- y is the population size of the predators
- t is time
- α, β, δ, and γ are constant parameters that describe the interactions between the two species. This example uses the parameter values $\alpha = \gamma = 1$, $\beta = 0.01$, and $\delta = 0.02$.

For this problem, the initial values for x and y are the initial population sizes. Solving the equations then provides information about how the populations change over time as the species interact.

Solve Equations with One Initial Condition

To solve the Lotka-Volterra equations in MATLAB, write a function that encodes the equations, specify a time interval for the integration, and specify the initial conditions. Then you can use one of the ODE solvers, such as `ode45`, to simulate the system over time.

A function that encodes the equations is

```
function dpdt = lotkaODE(t,p)
% LOTKA Lotka-Volterra predator-prey model
delta = 0.02;
beta = 0.01;

dpdt = [p(1) .* (1 - beta*p(2));
        p(2) .* (-1 + delta*p(1))];
end
```

(This function is included as a local function at the end of the example.)

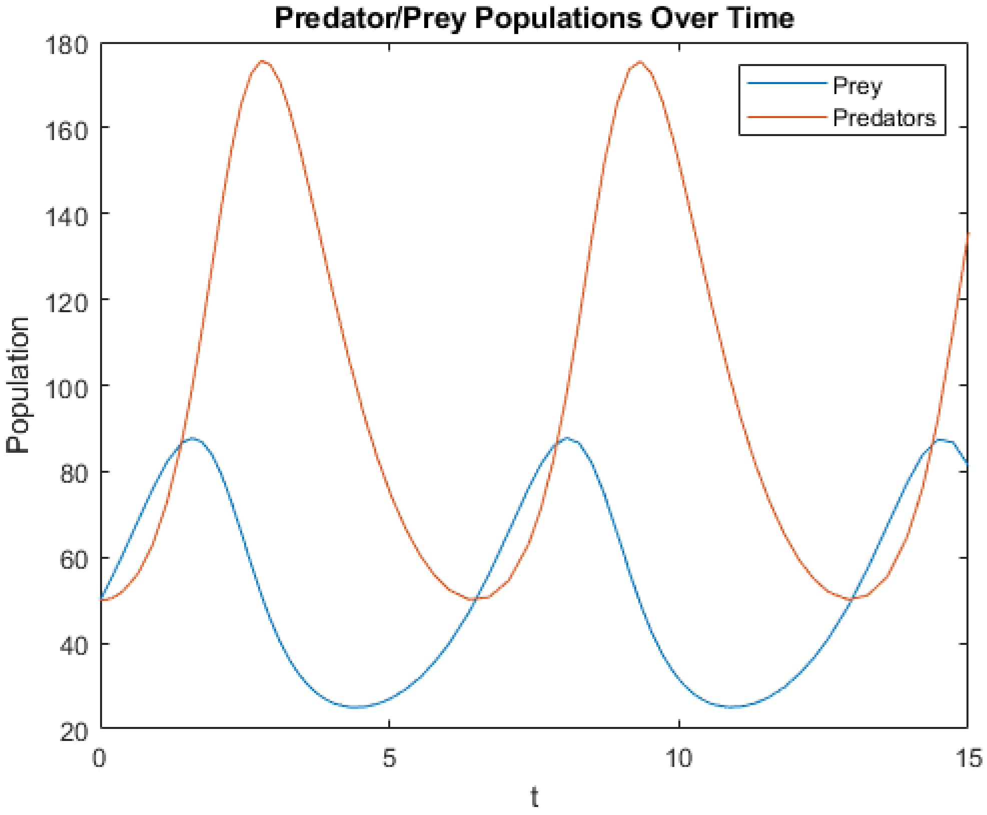
Since there are two equations in the system, `dpdt` is a vector with one element for each equation. Also, the solution vector `p` has one element for each solution component: `p(1)` represents *x* in the original equations, and `p(2)` represents *y* in the original equations.

Next, specify the time interval for integration as $[0, 15]$ and set the initial population sizes for *x* and *y* to 50.

```
t0 = 0;
tfinal = 15;
p0 = [50; 50];
```

Solve the system with `ode45` by specifying the ODE function, the time span, and the initial conditions. Plot the resulting populations versus time.

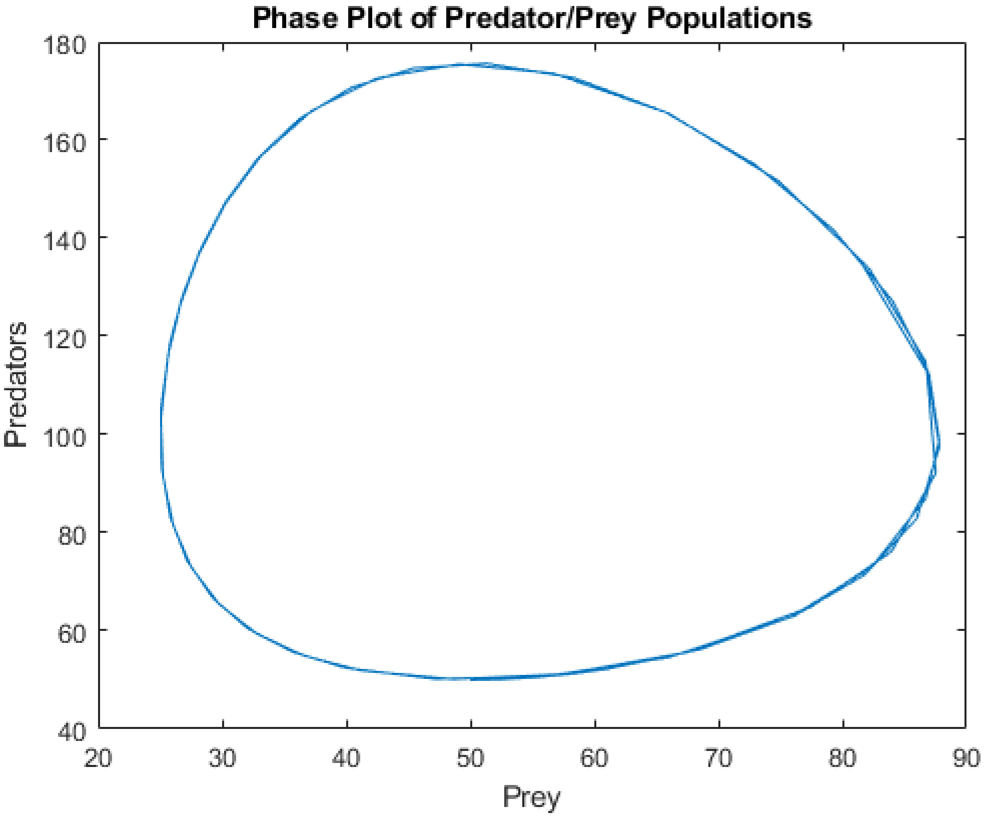
```
[t,p] = ode45(@lotkaODE,[t0 tfinal],p0);
plot(t,p)
title('Predator/Prey Populations Over Time')
xlabel('t')
ylabel('Population')
legend('Prey','Predators')
```



Since the solutions exhibit periodicity, plot the solutions against each other in a phase plot.

```
plot(p(:,1),p(:,2))
title('Phase Plot of Predator/Prey Populations')
```

```
xlabel('Prey')
ylabel('Predators')
```



The resulting plots show the solution for the given initial population sizes. To solve the equations for different initial population sizes, change the values in `p0` and rerun the simulation. However, this method only solves the equations for one initial condition at a time. The next two sections describe techniques to solve for many different initial conditions.

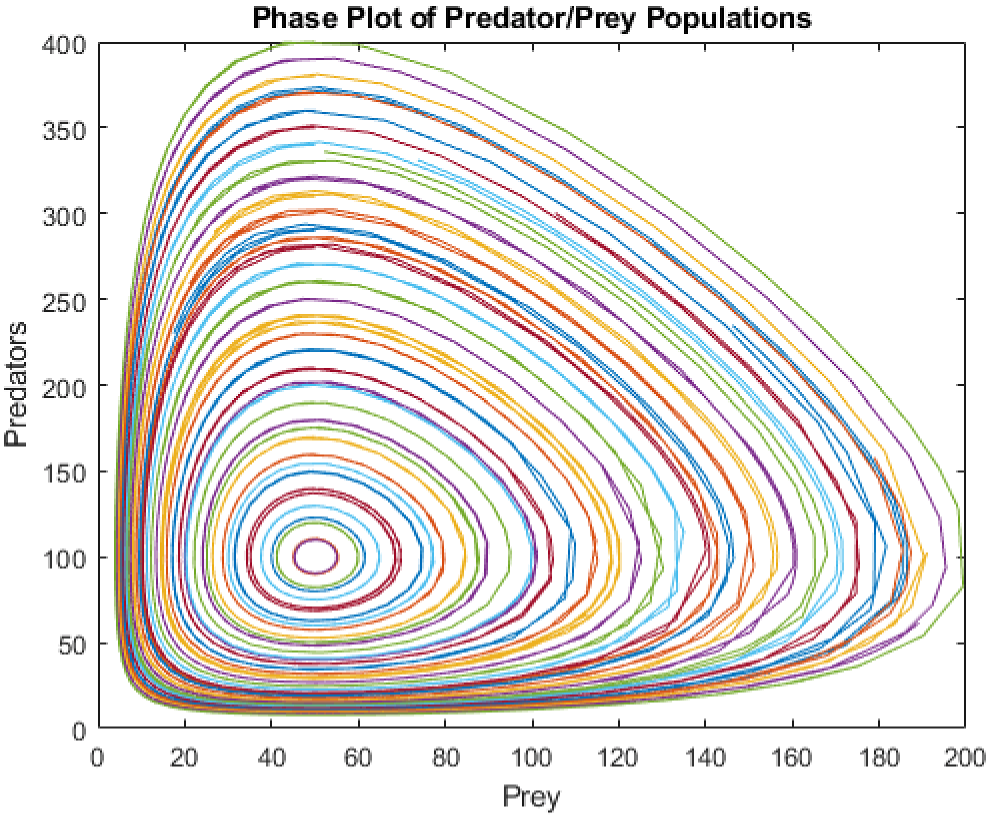
Method 1: Compute Multiple Initial Conditions with `for`-loop

The simplest way to solve a system of ODEs for multiple initial conditions is with a `for`-loop. This technique uses the same ODE function as the single initial condition technique, but the `for`-loop automates the solution process.

For example, you can hold the initial population size for *x* constant at 50, and use the `for`-loop to vary the initial population size for *y* between 10 and 400. Create a vector of population sizes for `y0`, and then loop over the values to solve the equations for each set of initial conditions. Plot a phase plot with the results from all iterations.

```
y0 = 10:10:400;
for k = 1:length(y0)
    [t,p] = ode45(@lotkaODE,[t0 tfinal],[50 y0(k)]);
    plot(p(:,1),p(:,2))
    hold on
end
title('Phase Plot of Predator/Prey Populations')
xlabel('Prey')
```

```
ylabel('Predators')
hold off
```



The phase plot shows all of the computed solutions for the different sets of initial conditions.

Method 2: Compute Multiple Initial Conditions with Vectorized ODE Function

Another method to solve a system of ODEs for multiple initial conditions is to rewrite the ODE function so that all of the equations are solved simultaneously. The steps to do this are:

- Provide all of the initial conditions to `ode45` as a matrix. The size of the matrix is `s`-by-`n`, where `s` is the number of solution components and `n` is the number of initial conditions being solved for. Each column in the matrix then represents one complete set of initial conditions for the system.
- The ODE function must accept an extra input parameter for `n`, the number of initial conditions.
- Inside the ODE function, the solver passes the solution components p as a column vector. The ODE function must reshape the vector into a matrix with size `s`-by-n. Each row of the matrix then contains all of the initial conditions for each variable.
- The ODE function must solve the equations in a vectorized format, so that the expression accepts vectors for the solution components. In other words, `f(t,[y1 y2 y3 ...])` must return `[f(t,y1) f(t,y2) f(t,y3) ...]`.
- Finally, the ODE function must reshape its output back into a vector so that the ODE solver receives a vector back from each function call.

If you follow these steps, then the ODE solver can solve the system of equations using a vector for the solution components, while the ODE function reshapes the vector into a matrix and solves each solution component for all of the initial conditions. The result is that you can solve the system for all of the initial conditions in one simulation.

To implement this method for the Lotka-Volterra system, start by finding the number of initial conditions n, and then form a matrix of initial conditions.

```
n = length(y0);
p0_all = [50*ones(n,1) y0(:)]';
```

Next, rewrite the ODE function so that it accepts n as an input. Use n to reshape the solution vector into a matrix, then solve the vectorized system and reshape the output back into a vector. A modified ODE function that performs these tasks is

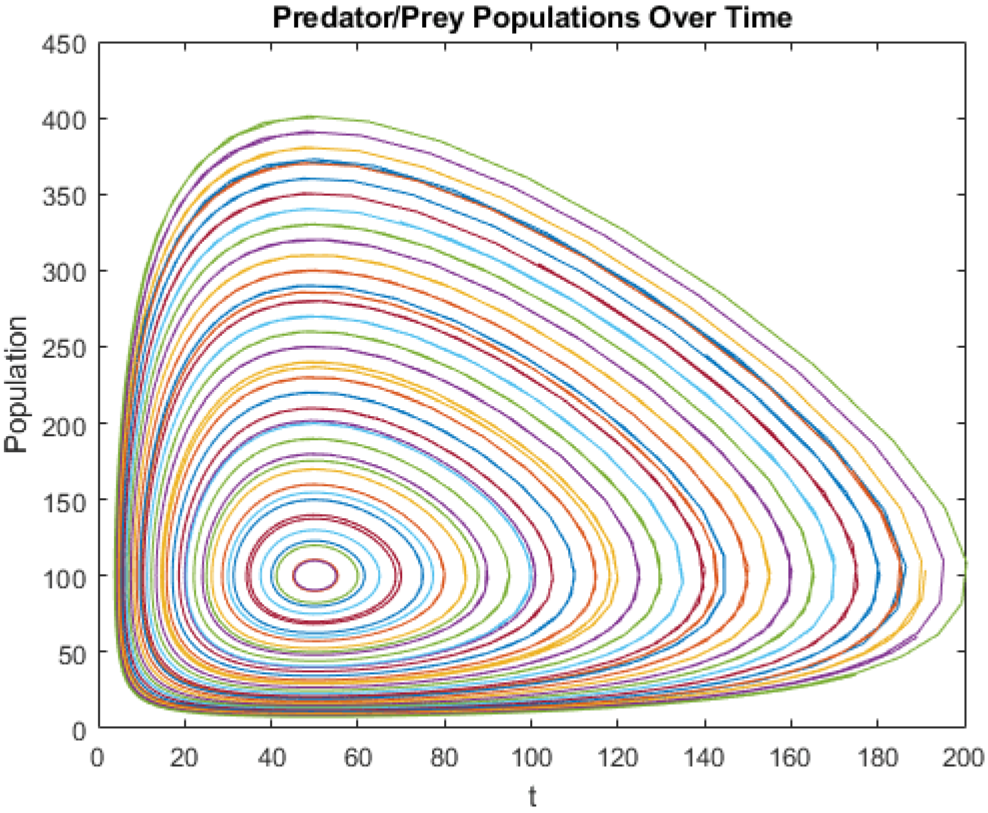
```
function dpdt = lotkasystem(t,p,n)
%LOTKA  Lotka-Volterra predator-prey model for system of inputs p.
delta = 0.02;
beta = 0.01;

% Change the size of p to be: Number of equations-by-number of initial
% conditions.
p = reshape(p,[],n);

% Write equations in vectorized form.
dpdt = [p(1,:) .* (1 - beta*p(2,:));
        p(2,:) .* (-1 + delta*p(1,:))];

% Linearize output.
dpdt = dpdt(:);
end
```

Solve the system of equations for all of the initial conditions using ode45. Since ode45 requires the ODE function to accept two inputs, use an anonymous function to pass in the value of n from the workspace to lotkasystem.

```
[t,p] = ode45(@(t,p) lotkasystem(t,p,n),[t0 tfinal],p0_all);
```

Reshape the output vector into a matrix with size (numTimeSteps*s)-by-n. Each column of the output p(:,k) contains the solutions for one set of initial conditions. Plot a phase plot of the solution components.

```
p = reshape(p,[],n);
nt = length(t);
for k = 1:n
    plot(p(1:nt,k),p(nt+1:end,k))
    hold on
end
title('Predator/Prey Populations Over Time')
xlabel('t')
ylabel('Population')
hold off
```



The results are comparable to those obtained by the `for`-loop technique. However, there are some properties of the vectorized solution technique that you should keep in mind:

- The calculated solutions can be slightly different than those computed from a single initial input. The difference arises because the ODE solver applies norm checks to the entire system to calculate the size of the time steps, so the time-stepping behavior of the solution is slightly different. The change in time steps generally does not affect the accuracy of the solution, but rather which times the solution is evaluated at.
- For stiff ODE solvers (`ode15s`, `ode23s`, `ode23t`, `ode23tb`) that automatically evaluate the numerical Jacobian of the system, specifying the block diagonal sparsity pattern of the Jacobian using the `JPattern` option of `odeset` can improve the efficiency of the calculation. The block diagonal form of the Jacobian arises from the input reshaping performed in the rewritten ODE function.

Compare Timing Results

Time each of the previous methods using `timeit`. The timing for solving the equations with one set of initial conditions is included as a baseline number to see how the methods scale.

```
% Time one IC
baseline = timeit(@() ode45(@lotkaODE,[t0 tfinal],p0),2);

% Time for-loop
for k = 1:length(y0)
    loop_timing(k) = timeit(@() ode45(@lotkaODE,[t0 tfinal],[50 y0(k)]),2);
end
```

```
loop_timing = sum(loop_timing);

% Time vectorized fcn
vectorized_timing = timeit(@() ode45(@(t,p) lotkasystem(t,p,n),[t0 tfinal],p0_all),2);
```

Create a table with the timing results. Multiply all of the results by 1e3 to express the times in milliseconds. Include a column with the time per solution, which divides each time by the number of initial conditions being solved for.

```
TimingTable = table(1e3.*[baseline; loop_timing; vectorized_timing], 1e3.*[baseline; loop_timing/
    'VariableNames',{'TotalTime (ms)','TimePerSolution (ms)'},'RowNames',{'One IC','Multi ICs: Fo
```

```
TimingTable=3×2 table
                            TotalTime (ms)    TimePerSolution (ms)
                            ______________    ____________________

    One IC                      1.2302               1.2302
    Multi ICs: For-loop         26.753              0.66884
    Mult ICs: Vectorized        1.9208             0.048019
```

The `TimePerSolution` column shows that the vectorized technique is the fastest of the three methods.

Local Functions

Listed here are the local functions that `ode45` calls to calculate the solutions.

```
function dpdt = lotkaODE(t,p)
% LOTKA Lotka-Volterra predator-prey model
delta = 0.02;
beta = 0.01;

dpdt = [p(1) .* (1 - beta*p(2));
        p(2) .* (-1 + delta*p(1))];
end
%------------------------------------------------------------------
function dpdt = lotkasystem(t,p,n)
%LOTKA  Lotka-Volterra predator-prey model for system of inputs p.
delta = 0.02;
beta = 0.01;

% Change the size of p to be: Number of equations-by-number of initial
% conditions.
p = reshape(p,[],n);

% Write equations in vectorized form.
dpdt = [p(1,:) .* (1 - beta*p(2,:));
        p(2,:) .* (-1 + delta*p(1,:))];

% Linearize output.
dpdt = dpdt(:);
end
```

See Also

ode45 | odeset

More About

- “Choose an ODE Solver” on page 11-2
- “Solve Predator-Prey Equations” on page 11-51

Boundary Value Problems (BVPs)

Solving Boundary Value Problems

In a *boundary value problem* (BVP), the goal is to find a solution to an ordinary differential equation (ODE) that also satisfies certain specified *boundary conditions*. The boundary conditions specify a relationship between the values of the solution at two or more locations in the interval of integration. In the simplest case, the boundary conditions apply at the beginning and end (or boundaries) of the interval.

The MATLAB BVP solvers `bvp4c` and `bvp5c` are designed to handle systems of ODEs of the form

$$y' = f(x, y)$$

where:

- x is the independent variable
- y is the dependent variable
- y' represents the derivative of y with respect to x, also written as dy/dx

Boundary Conditions

In the simplest case of a *two-point BVP*, the solution to the ODE is sought on an interval $[a, b]$, and must satisfy the boundary conditions

$$g(y(a), y(b)) = 0\ .$$

To specify the boundary conditions for a given BVP, you must:

- Write a function of the form `res = bcfun(ya,yb)`, or use the form `res = bcfun(ya,yb,p)` if there are unknown parameters involved. You supply this function to the solver as the second input argument. The function returns `res`, which is the residual value of the solution at the boundary point. For example, if $y(a) = 1$ and $y(b) = 0$, then the boundary condition function is

```
function res = bcfun(ya,yb)
res = [ya(1)-1
       yb(1)];
end
```

- In the initial guess for the solution, the first and last points in the mesh specify the points at which the boundary conditions are enforced. For the above boundary conditions, you can specify `bvpinit(linspace(a,b,5),yinit)` to enforce the boundary conditions at a and b.

The BVP solvers in MATLAB also can accommodate other types of problems that have:

- Unknown parameters p
- Singularities in the solutions
- Multipoint conditions (internal boundaries that separate the integration interval into several regions)

In the case of *multipoint boundary conditions*, the boundary conditions apply at more than two points in the interval of integration. For example, the solution might be required to be zero at the beginning, middle, and end of the interval. See `bvpinit` for details on how to specify multiple boundary conditions.

Initial Guess of Solution

Unlike initial value problems, a boundary value problem can have:

- No solution
- A finite number of solutions
- Infinitely many solutions

An important part of the process of solving a BVP is providing a guess for the required solution. The quality of this guess can be critical for the solver performance and even for a successful computation.

Use the `bvpinit` function to create a structure for the initial guess of the solution. The solvers `bvp4c` and `bvp5c` accept this structure as the third input argument.

Creating a good initial guess for the solution is more an art than a science. However, some general guidelines include:

- Have the initial guess satisfy the boundary conditions, since the solution is required to satisfy them as well. If the problem contains unknown parameters, then the initial guess for the parameters also should satisfy the boundary conditions.
- Try to incorporate as much information about the physical problem or expected solution into the initial guess as possible. For example, if the solution is supposed to oscillate or have a certain number of sign changes, then the initial guess should as well.
- Consider the placement of the mesh points (the x-coordinates of the initial guess of the solution). The BVP solvers adapt these points during the solution process, so you do not need to specify too many mesh points. Best practice is to specify a few mesh points placed near where the solution changes rapidly.
- If there is a known, simpler solution on a smaller interval, then use it as an initial guess on a larger interval. Often you can solve a problem as a series of relatively simpler problems, a practice called *continuation*. With continuation, a series of simple problems are connected by using the solution of one problem as the initial guess to solve the next problem.

Finding Unknown Parameters

Often BVPs involve unknown parameters p that have to be determined as part of solving the problem. The ODE and boundary conditions become

$$y' = f(x, y, p)$$
$$g(y(a), y(b), p) = 0$$

In this case, the boundary conditions must suffice to determine the values of the parameters p.

You must provide the solver with an initial guess for any unknown parameters. When you call `bvpinit` to create the structure `solinit`, specify the initial guess as a vector in the third input argument `parameters`.

```
solinit = bvpinit(x,v,parameters)
```

Additionally, the functions `odefun` and `bcfun` that encode the ODE equations and boundary conditions must each have a third argument.

```
dydx = odefun(x,y,parameters)
res = bcfun(ya,yb,parameters)
```

While solving the differential equations, the solver adjusts the value of the unknown parameters to satisfy the boundary conditions. The solver returns the final values of these unknown parameters in `sol.parameters`.

Singular BVPs

`bvp4c` and `bvp5c` can solve a class of singular BVPs of the form

$$\begin{aligned} y' &= \frac{1}{x}Sy + f(x,y), \\ 0 &= g(y(0), y(b))\,. \end{aligned}$$

The solvers can also accommodate unknown parameters for problems of the form

$$\begin{aligned} y' &= \frac{1}{x}Sy + f(x,y,p), \\ 0 &= g(y(0), y(b), p)\,. \end{aligned}$$

Singular problems must be posed on an interval [0,*b*] with $b > 0$. Use `bvpset` to pass the constant matrix *S* to the solver as the value of the `'SingularTerm'` option. Boundary conditions at $x = 0$ must be consistent with the necessary condition for a smooth solution, $Sy(0) = 0$. The initial guess of the solution also should satisfy this condition.

When you solve a singular BVP, the solver requires that your function `odefun(x,y)` return only the value of the $f(x, y)$ term in the equation. The term involving *S* is handled by the solver separately using the `'SingularTerm'` option.

BVP Solver Selection

MATLAB includes the solvers `bvp4c` and `bvp5c` to solve BVPs. In most cases you can use the solvers interchangeably. The main difference between the solvers is that `bvp4c` implements a fourth-order formula, while `bvp5c` implements a fifth-order formula.

The `bvp5c` function is used exactly like `bvp4c`, with the exception of the meaning of error tolerances between the two solvers. If $S(x)$ approximates the solution *y(x)*, `bvp4c` controls the residual $|S'(x) - f(x,S(x))|$. This approach indirectly controls the true error $|y(x) - S(x)|$. Use `bvp5c` to control the true error directly.

Solver	Description
bvp4c	bvp4c is a finite difference code that implements the 3-stage Lobatto IIIa formula. This is a collocation formula, and the collocation polynomial provides a C^1-continuous solution that is fourth-order accurate uniformly in the interval of integration. Mesh selection and error control are based on the residual of the continuous solution. The collocation technique uses a mesh of points to divide the interval of integration into subintervals. The solver determines a numerical solution by solving a global system of algebraic equations resulting from the boundary conditions and the collocation conditions imposed on all the subintervals. The solver then estimates the error of the numerical solution on each subinterval. If the solution does not satisfy the tolerance criteria, then the solver adapts the mesh and repeats the process. You *must* provide the points of the initial mesh, as well as an initial approximation of the solution at the mesh points.
bvp5c	bvp5c is a finite difference code that implements the four-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a C^1-continuous solution that is fifth-order accurate uniformly in [a,b]. The formula is implemented as an implicit Runge-Kutta formula. bvp5c solves the algebraic equations directly, whereas bvp4c uses analytical condensation. bvp4c handles unknown parameters directly, while bvp5c augments the system with trivial differential equations for unknown parameters.

Evaluating the Solution

The collocation methods implemented in bvp4c and bvp5c produce C^1-continuous solutions over the interval of integration [*a*,*b*]. You can evaluate the approximate solution, *S*(*x*), at any point in [*a*,*b*] using the helper function deval and the structure sol returned by the solver. For example, to evaluate the solution sol at the mesh of points xint, use the command

```
Sxint = deval(sol,xint)
```

The deval function is vectorized. For a vector xint, the ith column of Sxint approximates the solution y(xint(i)).

BVP Examples and Files

Several available example files serve as excellent starting points for most common BVP problems. To easily explore and run examples, simply use the **Differential Equations Examples** app. To run this app, type

```
odeexamples
```

To open an individual example file for editing, type

```
edit exampleFileName.m
```

To run an example, type

```
exampleFileName
```

This table contains a list of the available BVP example files, as well as the solvers and the options they use.

Example File	Solver Used	Options Specified	Description	Example Link
emdenbvp	bvp4c or bvp5c	• 'SingularTerm'	Emden's equation, a singular BVP	"Solve BVP with Singular Term" on page 12-25
fsbvp	bvp4c or bvp5c	—	Falkner-Skan BVP on an infinite interval	"Verify BVP Consistency Using Continuation" on page 12-20
mat4bvp	bvp4c or bvp5c	—	Fourth eigenfunction of Mathieu's equation	"Solve BVP with Unknown Parameter" on page 12-11
rcbvp	bvp4c and bvp5c	• 'FJacobian' • 'AbsTol' • 'RelTol' • 'Stats'	Example comparing the errors controlled by bvp4c and bvp5c	"Compare bvp4c and bvp5c Solvers" (bvp4c) "Compare bvp4c and bvp5c Solvers" (bvp5c)
shockbvp	bvp4c or bvp5c	• 'FJacobian' • 'BCJacobian' • 'Vectorized'	Solution with a shock layer near x = 0	"Solve BVP Using Continuation" on page 12-15
twobvp	bvp4c	—	BVP with exactly two solutions	"Solve BVP with Two Solutions" on page 12-8
threebvp	bvp4c or bvp5c	—	Three-point boundary value problem	"Solve BVP with Multiple Boundary Conditions" on page 12-29

References

[1] Ascher, U., R. Mattheij, and R. Russell. “*Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*.” Philadelphia, PA: SIAM, 1995, p. 372.

[2] Shampine, L.F., and J. Kierzenka. "A BVP Solver based on residual control and the MATLAB PSE." *ACM Trans. Math. Softw.* Vol. 27, Number 3, 2001, pp. 299–316.

[3] Shampine, L.F., M.W. Reichelt, and J. Kierzenka. "Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c." *MATLAB File Exchange*, 2004.

[4] Shampine, L.F., and J. Kierzenka. "A BVP Solver that Controls Residual and Error." *J. Numer. Anal. Ind. Appl. Math.* Vol. 3(1-2), 2008, pp. 27–41.

See Also

`bvp4c` | `bvp5c` | `bvpinit` | `bvpset` | `ode45` | `pdepe`

Solve BVP with Two Solutions

This example uses `bvp4c` with two different initial guesses to find both solutions to a BVP problem.

Consider the differential equation

$$y'' + e^y = 0.$$

This equation is subject to the boundary conditions

$$y(0) = y(1) = 0.$$

To solve this equation in MATLAB, you need to code the equation and boundary conditions, then generate a suitable initial guess for the solution before calling the boundary value problem solver `bvp4c`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Create a function to code the equation. This function should have the signature `dydx = bvpfun(x,y)` or `dydx = bvpfun(x,y,parameters)`, where:

- `x` is the independent variable.
- `y` is the solution (dependent variable).
- `parameters` is a vector of unknown parameter values (optional).

These inputs are automatically passed to the function by the solver, but the variable names determine how you code the equations. In this case, you can rewrite the second-order equation as a system of first-order equations

$$y_1' = y_2,$$

$$y_2' = -e^{y_1}.$$

The function encoding these equations is

```
function dydx = bvpfun(x,y)
dydx = [y(2)
        -exp(y(1))];
end
```

Code Boundary Conditions

For two-point boundary value conditions like the ones in this problem, the boundary conditions function should have the signature `res = bcfun(ya,yb)` or `res = bcfun(ya,yb,parameters)`, depending on whether unknown parameters are involved. `ya` and `yb` are column vectors that the solver automatically passes to the function, and `bcfun` returns the residual in the boundary conditions.

For the boundary conditions $y(0) = y(1) = 0$, the `bcfun` function specifies that the residual value is zero at both boundaries. These residual values are enforced at the first and last points of the mesh that you specify to `bvpinit` in your initial guess. The initial mesh in this problem should have `x(1) = 0` and `x(end) = 1`.

```
function res = bcfun(ya,yb)
res = [ya(1)
       yb(1)];
end
```

Form Initial Guess

Call bvpinit to generate an initial guess of the solution. The mesh for x does not need to have a lot of points, but the first point must be 0. Then the last point must be 1 so that the boundary conditions are properly specified. Use an initial guess for y where the first component is slightly positive and the second component is zero.

```
xmesh = linspace(0,1,5);
solinit = bvpinit(xmesh, [0.1 0]);
```

Solve Equation

Solve the BVP using the bvp4c solver.

```
sol1 = bvp4c(@bvpfun, @bcfun, solinit);
```

Use Different Initial Guess

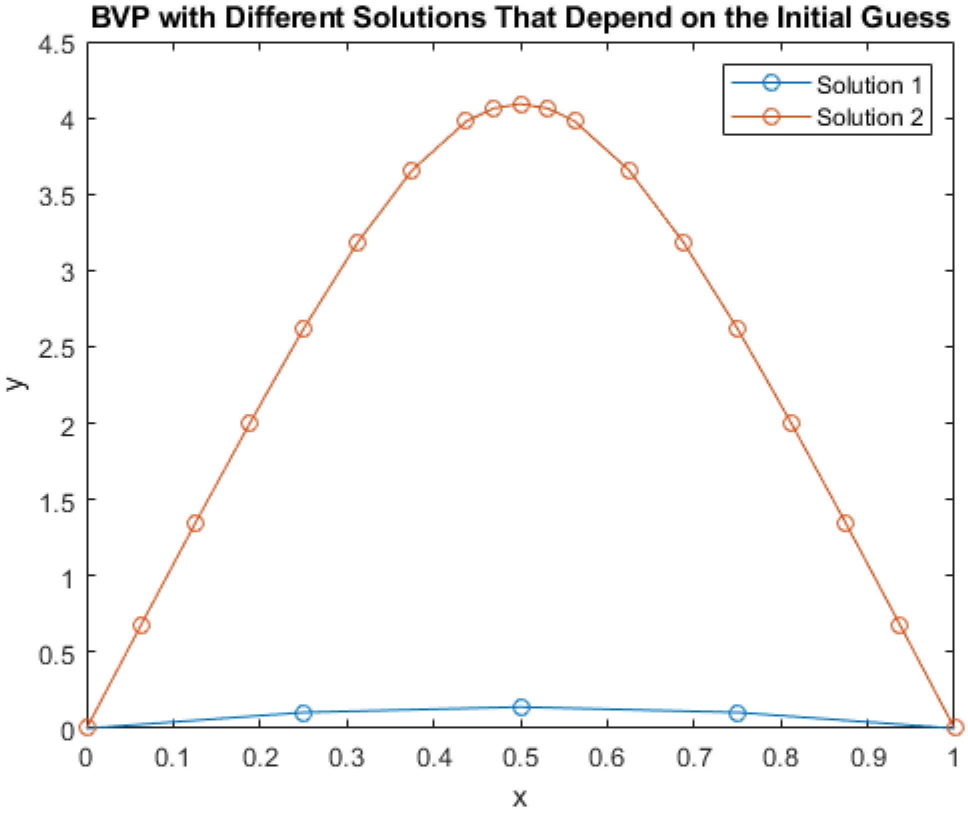
Solve the BVP a second time using a different initial guess for the solution.

```
solinit = bvpinit(xmesh, [3 0]);
sol2 = bvp4c(@bvpfun, @bcfun, solinit);
```

Compare Solutions

Plot the solutions that bvp4c calculates for the different initial conditions. Both solutions satisfy the stated boundary conditions, but have different behaviors inbetween. Since the solution is not always unique, the different behaviors show the importance of giving a good initial guess for the solution.

```
plot(sol1.x,sol1.y(1,:),'-o',sol2.x,sol2.y(1,:),'-o')
title('BVP with Different Solutions That Depend on the Initial Guess')
xlabel('x')
ylabel('y')
legend('Solution 1','Solution 2')
```



Local Functions

Listed here are the local helper functions that the BVP solver `bvp4c` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydx = bvpfun(x,y) % equation being solved
dydx = [y(2)
        -exp(y(1))];
end
%-------------------------------------------
function res = bcfun(ya,yb) % boundary conditions
res = [ya(1)
       yb(1)];
end
%-------------------------------------------
```

See Also

`bvp4c` | `bvp5c` | `bvpinit`

More About

- “Solving Boundary Value Problems” on page 12-2

Solve BVP with Unknown Parameter

This example shows how to use bvp4c to solve a boundary value problem with an unknown parameter.

Mathieu's equation is defined on the interval $[0, \pi]$ by

$$y'' + (\lambda - 2q\cos(2x))y = 0.$$

When the parameter $q = 5$, the boundary conditions are

$$y'(0) = 0,$$

$$y'(\pi) = 0.$$

However, this only determines $y(x)$ up to a constant multiple, so a third condition is required to specify a particular solution,

$$y(0) = 1.$$

To solve this system of equations in MATLAB, you need to code the equations, boundary conditions, and initial guess before calling the boundary value problem solver bvp4c. You can either include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Create a function to code the equations. This function should have the signature `dydx = mat4ode(x,y,lambda)`, where:

- x is the independent variable.
- y is the dependent variable.
- lambda is an unknown parameter representing an eigenvalue.

You can write Mathieu's equation as a first-order system using the substitutions $y_1 = y$ and $y_2 = y'$,

$$y_1' = y_2,$$

$$y_2' = -(\lambda - 2q\cos(2x))y_1.$$

The corresponding function is then

```
function dydx = mat4ode(x,y,lambda) % equation being solved
dydx = [y(2)
      -(lambda - 2*q*cos(2*x))*y(1)];
end
```

Note: All functions are included as local functions at the end of the example.

Code Boundary Conditions

Now, write a function that returns the residual value of the boundary conditions at the boundary points. This function should have the signature `res = mat4bc(ya,yb,lambda)`, where:

- `ya` is the value of the boundary condition at the beginning of the interval $[a, b]$.
- `yb` is the value of the boundary condition at the end of the interval $[a, b]$.
- `lambda` is an unknown parameter representing an eigenvalue.

This problem has three boundary conditions in the interval $[0, \pi]$. To calculate the residual values, you need to put the boundary conditions into the form $g(x, y) = 0$. In this form the boundary conditions are

$y'(0) = 0,$

$y'(\pi) = 0,$

$y(0) - 1 = 0.$

The corresponding function is then

```
function res = mat4bc(ya,yb,lambda) % boundary conditions
res = [ya(2)
       yb(2)
       ya(1)-1];
end
```

Create Initial Guess

Lastly, create an initial guess of the solution. You must provide an initial guess for both solution components $y_1 = y(x)$ and $y_2 = y'(x)$, as well as the unknown parameter λ. Only eigenvalues and eigenfunctions that are close to the initial guesses can be computed. To increase the likelihood that the computed eigenfunction corresponds to the fourth eigenvalue, you should choose an initial guess that has the correct qualitative behavior.

For this problem, a cosine function makes for a good initial guess since it satisfies the three boundary conditions. Code the initial guess for y using a function that returns the guess for y_1 and y_2.

```
function yinit = mat4init(x) % initial guess function
yinit = [cos(4*x)
        -4*sin(4*x)];
end
```

Call `bvpinit` using a mesh of 10 points in the interval $[0, \pi]$, the initial guess function, and a guess of 15 for the value of λ.

```
lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
```

Solve Equation

Call `bvp4c` with the ODE function, boundary condition function, and initial guess.

```
sol = bvp4c(@mat4ode, @mat4bc, solinit);
```

Value of Parameter

Print the value of the unknown parameter λ found by `bvp4c`. This value is the fourth eigenvalue ($q = 5$) of Mathieu's equation.

```
fprintf('Fourth eigenvalue is approximately %7.3f.\n',...
            sol.parameters)
```

```
Fourth eigenvalue is approximately  17.097.
```

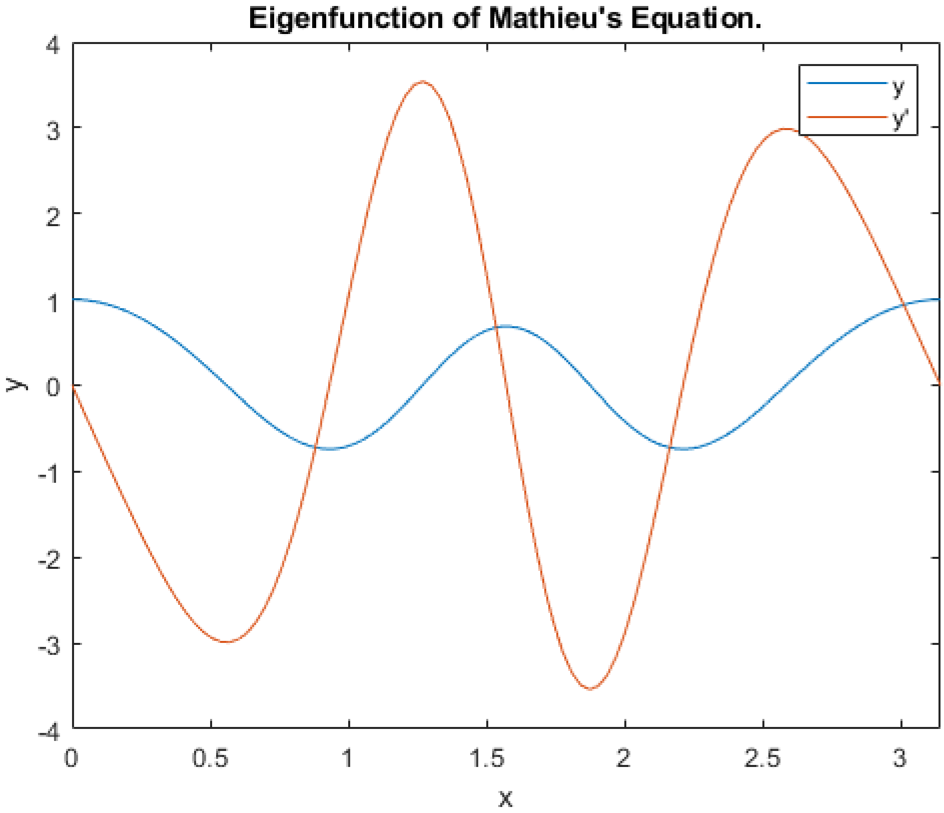
Plot Solution

Use `deval` to evaluate the solution computed by `bvp4c` at 100 points in the interval $[0, \pi]$.

```
xint = linspace(0,pi);
Sxint = deval(sol,xint);
```

Plot both solution components. The plot shows the eigenfunction (and its derivative) associated with the fourth eigenvalue $\lambda_4 = 17.097$.

```
plot(xint,Sxint)
axis([0 pi -4 4])
title('Eigenfunction of Mathieu''s Equation.')
xlabel('x')
ylabel('y')
legend('y','y''')
```



Local Functions

Listed here are the local helper functions that the BVP solver `bvp4c` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydx = mat4ode(x,y,lambda) % equation being solved
q = 5;
dydx = [y(2)
      -(lambda - 2*q*cos(2*x))*y(1)];
```

```
end
%-------------------------------------------
function res = mat4bc(ya,yb,lambda) % boundary conditions
res = [ya(2)
       yb(2)
       ya(1)-1];
end
%-------------------------------------------
function yinit = mat4init(x) % initial guess function
yinit = [cos(4*x)
        -4*sin(4*x)];
end
%-------------------------------------------
```

See Also

bvp4c | bvp5c | bvpinit

More About

- “Solving Boundary Value Problems” on page 12-2

Solve BVP Using Continuation

This example shows how to solve a numerically difficult boundary value problem using continuation, which effectively breaks the problem up into a sequence of simpler problems.

For $0 < e \ll 1$, consider the differential equation

$$ey'' + \mathrm{x}y' = -e\pi^2\cos(\pi x) - \pi \mathrm{x}\sin(\pi x).$$

The problem is posed on the interval $[-1, 1]$ and is subject to the boundary conditions

$$y(-1) = -2,$$

$$y(1) = 0.$$

When $e = 10^{-4}$, the solution to the equation undergoes a rapid transition near $x = 0$, so it is difficult to solve numerically. Instead, this example uses continuation to iterate through several values of e until $e = 10^{-4}$. The intermediate solutions are each used as the initial guess for the next problem.

To solve this system of equations in MATLAB, you need to code the equations, boundary conditions, and initial guess before calling the boundary value problem solver `bvp4c`. You either can include the required functions as local functions at the end of a file (as done here), or you can save them as separate, named files in a directory on the MATLAB path.

Code Equation

Using the substitutions $y_1 = y$ and $y_2 = y'$, you can rewrite the equation as the system of first-order equations

$$y_1' = y_2,$$

$$y_2' = -\frac{x}{e}y' - \pi^2\cos(\pi x) - \frac{\pi x}{e}\sin(\pi x).$$

Write a function to code the equations with the signature `dydx = shockode(x,y)`, where:

- `x` is the independent variable.
- `y` is the dependent variable.
- `dydx(1)` gives the equation for y_1', and `dydx(2)` gives the equation for y_2'.

Make the function vectorized, so that `shockode([x1 x2 ...],[y1 y2 ...])` returns `[shockode(x1,y1) shockode(x2,y2) ...]`. This approach improves solver performance.

The corresponding function is

```
function dydx = shockode(x,y)
pix = pi*x;
dydx = [y(2,:)
       -x/e.*y(2,:) - pi^2*cos(pix) - pix/e.*sin(pix)];
end
```

Note: All functions are included as local functions at the end of the example.

Code Boundary Conditions

The BVP solver requires the boundary conditions to be in the form $g(y(a),y(b)) = 0$. In this form the boundary conditions are:

$y(-1) + 2 = 0,$

$y(1) = 0.$

Write a function to code the boundary conditions with the signature `res = shockbc(ya,yb)`, where:

- `ya` is the value of the boundary condition at the beginning of the interval $[a,b]$.
- `yb` is the value of the boundary condition at the end of the interval $[a,b]$.

The corresponding function is

```
function res = shockbc(ya,yb) % boundary conditions
res = [ya(1)+2
       yb(1)];
end
```

Code Jacobians

The analytical Jacobians for the ODE function and boundary conditions can be calculated easily in this problem. Providing the Jacobians makes the solver more efficient, since the solver no longer needs to approximate them with finite differences.

For the ODE function, the Jacobian matrix is

$$J_{\text{ODE}} = \frac{\partial f}{\partial y} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -\frac{x}{e} \end{bmatrix}.$$

The corresponding function is

```
function jac = shockjac(x,y,e)
jac = [0   1
       0  -x/e];
end
```

Similarly, for the boundary conditions, the Jacobian matrices are

$$J_{y(a)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix},\ J_{y(b)} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}.$$

The corresponding function is

```
function [dBCdya,dBCdyb] = shockbcjac(ya,yb)
dBCdya = [1 0; 0 0];
dBCdyb = [0 0; 1 0];
end
```

Form Initial Guess

Use a constant guess for the solution on a mesh of five points in $[-1,1]$.

```
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
```

Solve Equation

If you try to solve the equation directly using $e = 10^{-4}$, then the solver is not able to overcome the poor conditioning of the problem near the $x = 0$ transition point. Instead, to obtain the solution for $e = 10^{-4}$, this example uses continuation by solving a sequence of problems for 10^{-2}, 10^{-3}, and 10^{-4}. The output from the solver in each iteration acts as the guess for the solution in the next iteration (this is why the variable for the initial guess from `bvpinit` is `sol`, and the output from the solver is also named `sol`).

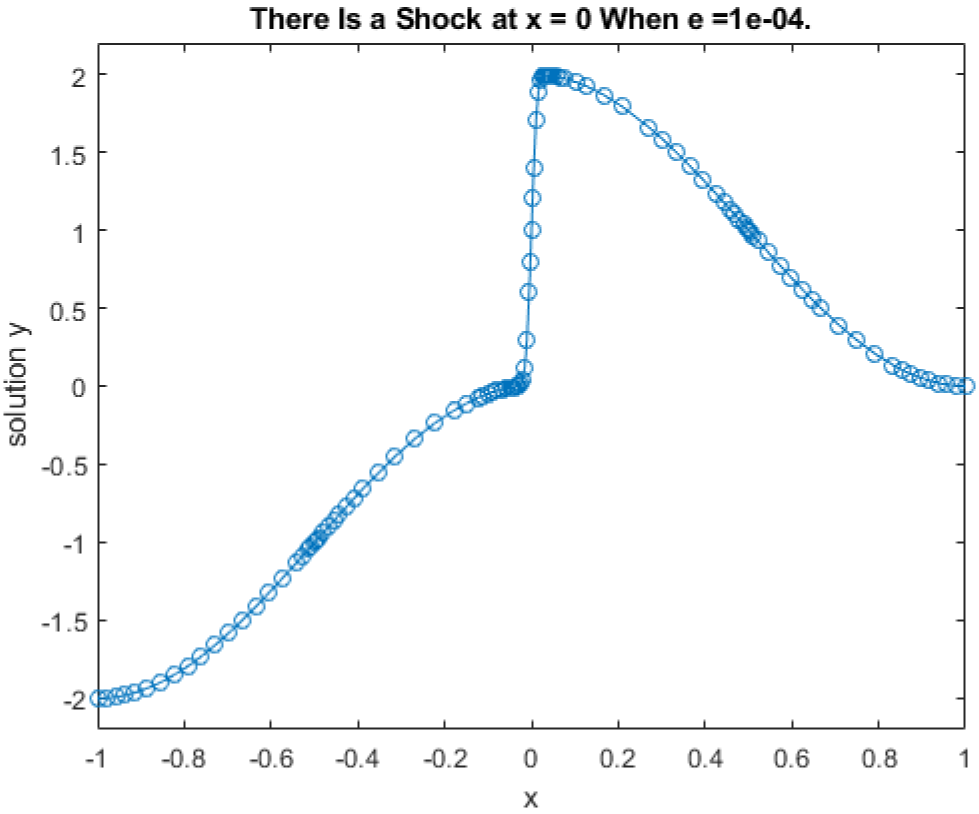
Since the value of the Jacobian depends on the value of e, set the options in the loop specifying the `shockjac` and `shockbcjac` functions for the Jacobians. Also, turn vectorization on since `shockode` is coded to handle vectors of values.

```
e = 0.1;
for i = 2:4
   e = e/10;
   options = bvpset('FJacobian',@(x,y) shockjac(x,y,e),'BCJacobian',@shockbcjac,'Vectorized','on
   sol = bvp4c(@(x,y) shockode(x,y,e),@shockbc, sol, options);
end
```

Plot Solution

Plot the results from `bvp4c` for the mesh x and solution $y(x)$. With continuation, the solver is able to handle the discontinuity at $x = 0$.

```
plot(sol.x,sol.y(1,:),'-o');
axis([-1 1 -2.2 2.2]);
title(['There Is a Shock at x = 0 When e =' sprintf('%.e',e) '.']);
xlabel('x');
ylabel('solution y');
```



Local Functions

Listed here are the local functions that the BVP solver `bvp4c` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydx = shockode(x,y,e) % equation to solve
pix = pi*x;
dydx = [y(2,:)
       -x/e.*y(2,:) - pi^2*cos(pix) - pix/e.*sin(pix)];
end
%-------------------------------------------
function res = shockbc(ya,yb) % boundary conditions
res = [ya(1)+2
       yb(1)];
end
%-------------------------------------------
function jac = shockjac(x,y,e) % jacobian of shockode
jac = [0   1
       0  -x/e];
end
%-------------------------------------------
function [dBCdya,dBCdyb] = shockbcjac(ya,yb) % jacobian of shockbc
dBCdya = [1 0; 0 0];
dBCdyb = [0 0; 1 0];
```

```
end
%-------------------------------------------
```

See Also

`bvp4c` | `bvpinit` | `bvpset`

More About

- “Solving Boundary Value Problems” on page 12-2
- “Verify BVP Consistency Using Continuation” on page 12-20

Verify BVP Consistency Using Continuation

This example shows how to use continuation to gradually extend a BVP solution to larger intervals.

Falkner-Skan boundary value problems [1] arise from similarity solutions of a viscous, incompressible, laminar flow over a flat plate. An example equation is

$$f''' + f\,f'' + \beta\left(1 - f'^{\,2}\right) = 0.$$

The problem is posed on the infinite interval $[0,\ \infty\ ]$ with $\beta = 0.5$, subject to the boundary conditions

$$f(0) = 0,$$

$$f'(0) = 0,$$

$$f'(\ \infty\ ) = 1.$$

The BVP cannot be solved on the infinite interval, and it is impractical to solve the BVP in a very large finite interval. Instead, this example solves a sequence of problems posed on the smaller interval $[0, a]$ to verify that the solution has consistent behavior as $a \to\ \infty$. This practice of breaking the problem up into simpler problems, with the solution of each problem feeding back in as the initial guess for the next problem, is called *continuation*.

To solve this system of equations in MATLAB, you need to code the equations, boundary conditions, and options before calling the boundary value problem solver `bvp4c`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Create a function to code the equations. This function should have the signature `dfdx = fsode(x,f)`, where:

- x is the independent variable.
- f is the dependent variable.

You can rewrite the third-order equation as a system of first-order equations using the substitutions $f_1 = f$, $f_2 = f'$, and $f_3 = f''$. The equations become

$$f_1{}' = f_2,$$

$$f_2{}' = f_3,$$

$$f_3{}' = -f_1f_3 - \beta\left(1 - f_2^2\right).$$

The corresponding function is

```
function dfdeta = fsode(x,f)
b = 0.5;
dfdeta = [ f(2)
           f(3)
          -f(1)*f(3) - b*(1 - f(2)^2) ];
end
```

Note: All functions are included as local functions at the end of the example.

Code Boundary Conditions

Now, write a function that returns the residual value of the boundary conditions at the boundary points. This function should have the signature `res = fsbc(f0,finf)`, where:

- `f0` is the value of the boundary condition at the beginning of the interval.
- `finf` is the value of the boundary condition at the end of the interval.

To calculate the residual values, you need to put the boundary conditions in the form $g(x,y)=0$. In this form, the boundary conditions are

$$f(0)=0,$$

$$f'(0)=0,$$

$$f'(\infty)-1=0.$$

The corresponding function is

```
function res = fsbc(f0,finf)
res = [f0(1)
       f0(2)
       finf(2) - 1];
end
```

Create Initial Guess

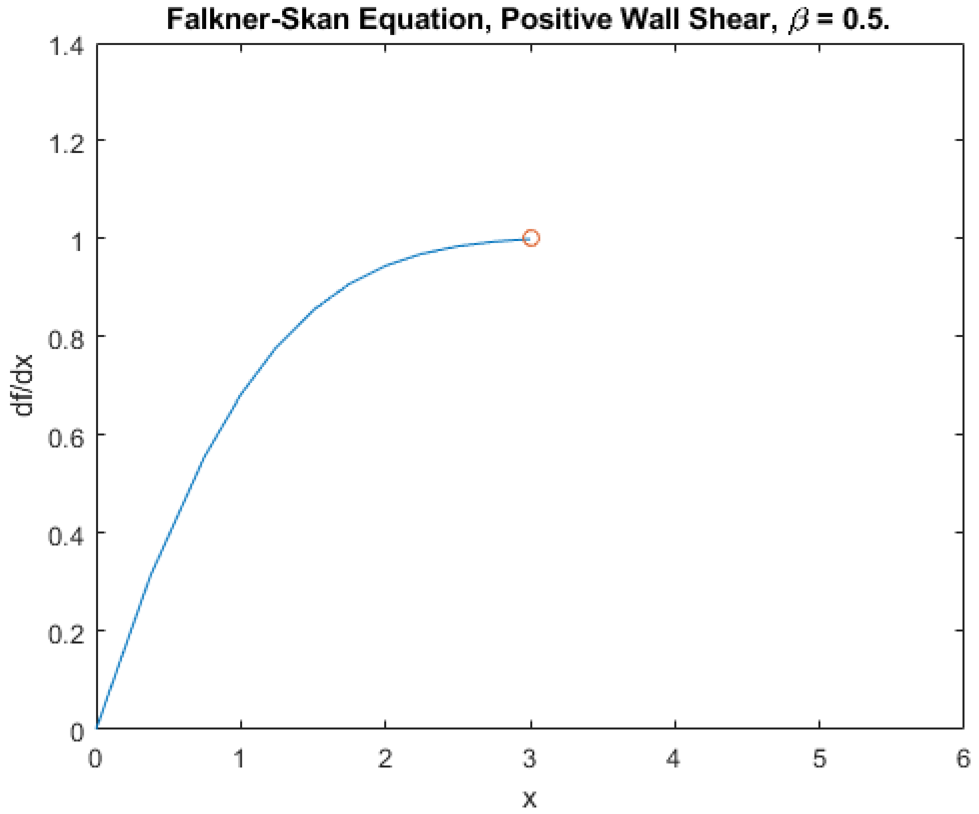
Lastly, you must provide an initial guess for the solution. A crude mesh of five points and a constant guess that satisfies the boundary conditions are good enough to get convergence on the interval $[0,3]$. The variable `infinity` denotes the right-hand limit of the interval of integration. As the value of `infinity` increases on subsequent iterations from 3 to its maximum value of 6, the solution from each previous iteration acts as the initial guess for the next iteration.

```
infinity = 3;
maxinfinity = 6;
solinit = bvpinit(linspace(0,infinity,5),[0 0 1]);
```

Solve Equation and Plot Solutions

Solve the problem in the initial interval [0, 3]. Plot the solution, and compare the value of $f''(0)$ to the analytic value [1].

```
sol = bvp4c(@fsode,@fsbc,solinit);
x = sol.x;
f = sol.y;
plot(x,f(2,:),x(end),f(2,end),'o');
axis([0 maxinfinity 0 1.4]);
title('Falkner-Skan Equation, Positive Wall Shear, \beta = 0.5.')
xlabel('x')
ylabel('df/dx')
hold on
```



```
fprintf('Cebeci & Keller report that f''''(0) = 0.92768.\n')
```

```
Cebeci & Keller report that f''(0) = 0.92768.
```

```
fprintf('Value computed using infinity = %g is %7.5f.\n', ...
         infinity,f(3,1))
```

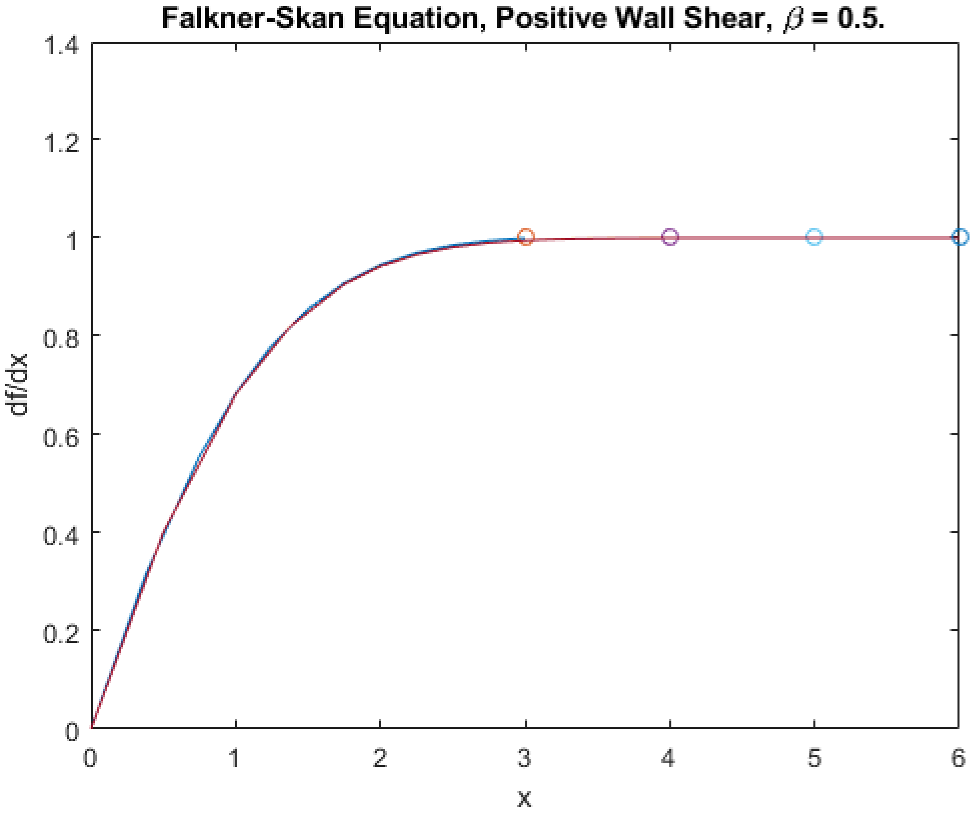
```
Value computed using infinity = 3 is 0.92915.
```

Now, solve the problem on progressively larger intervals by increasing the value of infinity for each iteration. The bvpinit function extrapolates each solution to the new interval to act as the initial guess for the next value of infinity. Each iteration prints the calculated value of $f''(0)$ and superimposes the plot of the solution over the previous solutions. When infinity = 6, the consistent behavior of the solution becomes evident and the value of $f''(0)$ is very close to the predicted value.

```
for Bnew = infinity+1:maxinfinity
  solinit = bvpinit(sol,[0 Bnew]); % Extend solution to new interval
  sol = bvp4c(@fsode,@fsbc,solinit);
  x = sol.x;
  f = sol.y;

  fprintf('Value computed using infinity = %g is %7.5f.\n', ...
           Bnew,f(3,1))
  plot(x,f(2,:),x(end),f(2,end),'o');
  drawnow
end
```

```
Value computed using infinity = 4 is 0.92774.
Value computed using infinity = 5 is 0.92770.
Value computed using infinity = 6 is 0.92770.
```

```
hold off
```



Local Functions

Listed here are the local helper functions that the BVP solver `bvp4c` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dfdeta = fsode(x,f) % equation being solved
dfdeta = [ f(2)
           f(3)
          -f(1)*f(3) - 0.5*(1 - f(2)^2) ];
end
%-------------------------------------------
function res = fsbc(f0,finf) % boundary conditions
res = [f0(1)
       f0(2)
       finf(2) - 1];
end
%-------------------------------------------
```

References

[1] Cebeci, T. and H. B. Keller. "Shooting and Parallel Shooting Methods for Solving the Falkner-Skan Boundary-layer Equation." *J. Comp. Phys.*, Vol. 7, 1971, pp. 289-300.

See Also

`bvp4c` | `bvp5c` | `bvpinit`

More About

- “Solving Boundary Value Problems” on page 12-2
- “Solve BVP Using Continuation” on page 12-15

Solve BVP with Singular Term

This example shows how to solve Emden's equation, which is a boundary value problem with a singular term that arises in modeling a spherical body of gas.

After reducing the PDE of the model using symmetry, the equation becomes a second-order ODE defined on the interval $[0, 1]$,

$$y'' + \frac{2}{x}y' + y^5 = 0.$$

At $x = 0$, the $(2/x)$ term is singular, but symmetry implies the boundary condition $y'(0) = 0$. With this boundary condition, the term $(2/x)y'$ is well defined as $x \to 0$. For the boundary condition $y(1) = \sqrt{3}/2$, the BVP has an analytical solution that you can compare to a numeric solution calculated in MATLAB®,

$$y(x) = \left[\sqrt{1 + \frac{x^2}{3}}\right]^{-1}.$$

The BVP solver `bvp4c` can solve singular BVPs that have the form

$$y' = S\frac{y}{x} + f(x, y).$$

The matrix S must be constant and the boundary conditions at $x = 0$ must be consistent with the necessary condition $S \cdot y(0) = 0$. Use the `'SingularTerm'` option of `bvpset` to pass the `S` matrix to the solver.

You can rewrite Emden's equation as a system of first-order equations using $y_1 = y$ and $y_2 = y'$ as

$$y_1' = y_2,$$

$$y_2' = -\frac{2}{x}y_2 - y_1^5.$$

The boundary conditions become

$$y_2(0) = 0,$$

$$y_1(1) = \sqrt{3}/2.$$

In the required matrix form, the system is

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \frac{1}{x}\begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} y_2 \\ -y_1^5 \end{bmatrix}.$$

In matrix form it is clear that $S = \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix}$ and $f(x, y) = \begin{bmatrix} y_2 \\ -y_1^5 \end{bmatrix}$.

To solve this system of equations in MATLAB, you need to code the equations, boundary conditions, and options before calling the boundary value problem solver `bvp4c`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Create a function to code the equations for $f(x,y)$. This function should have the signature `dydx = emdenode(x,y)`, where:

- `x` is the independent variable.
- `y` is the dependent variable.
- `dydx(1)` gives the equation for y_1', and `dydx(2)` the equation for y_2'.

These inputs are automatically passed to the function by the solver, but the variable names determine how you code the equations. In this case:

```
function dydx = emdenode(x,y)
dydx = [y(2)
       -y(1)^5];
end
```

The term that contains `S` is passed to the solver using options, so that term is not included in the function.

Code Boundary Conditions

Now, write a function that returns the residual value of the boundary conditions at the boundary points. This function should have the signature `res = emdenbc(ya,yb)`, where:

- `ya` is the value of the boundary condition at the beginning of the interval.
- `yb` is the value of the boundary condition at the end of the interval.

For this problem, one of the boundary conditions is for y_1, and the other is for y_2. To calculate the residual values, you need to put the boundary conditions into the form $g(x,y) = 0$.

In this form the boundary conditions are

$$y_2(0) = 0,$$

$$y_1(1) - \sqrt{3}/2 = 0.$$

The corresponding function is then

```
function res = emdenbc(ya,yb)
res = [ya(2)
       yb(1) - sqrt(3)/2];
end
```

Create Initial Guess

Lastly, create an initial guess of the solution. For this problem, use a constant initial guess that satisfies the boundary conditions, and a simple mesh of five points between 0 and 1. Using many mesh points is unnecessary since the BVP solver adapts these points during the solution process.

$$y_1 = \sqrt{3}/2,$$

$y_2 = 0$.

```
guess = [sqrt(3)/2; 0];
xmesh = linspace(0,1,5);
solinit = bvpinit(xmesh, guess);
```

Solve Equation

Create a matrix for `S` and pass it to `bvpset` as the value of the `'SingularTerm'` option. Finally, call `bvp4c` with the ODE function, boundary condition function, initial guess, and option structure.

```
S = [0 0; 0 -2];
options = bvpset('SingularTerm',S);
sol = bvp4c(@emdenode, @emdenbc, solinit, options);
```

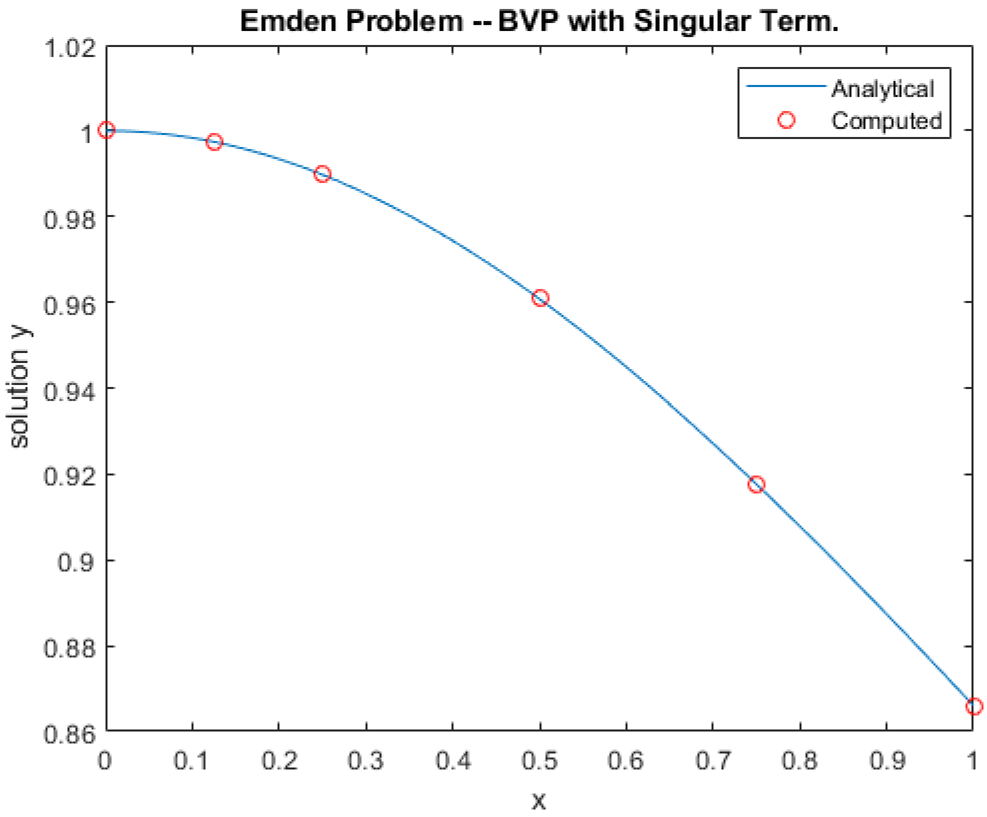
Plot Solution

Calculate the value of the analytical solution in [0, 1].

```
x = linspace(0,1);
truy = 1 ./ sqrt(1 + (x.^2)/3);
```

Plot the analytical solution and the solution calculated by `bvp4c` for comparison.

```
plot(x,truy,sol.x,sol.y(1,:),'ro');
title('Emden Problem -- BVP with Singular Term.')
legend('Analytical','Computed');
xlabel('x');
ylabel('solution y');
```



Local Functions

Listed here are the local helper functions that the BVP solver `bvp4c` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydx = emdenode(x,y) % equation being solved
dydx = [y(2)
       -y(1)^5];
end
%-------------------------------------------
function res = emdenbc(ya,yb) % boundary conditions
res = [ya(2)
       yb(1) - sqrt(3)/2];
end
%-------------------------------------------
```

See Also

`bvp4c` | `bvp5c` | `bvpinit` | `bvpset`

More About

- “Solving Boundary Value Problems” on page 12-2

Solve BVP with Multiple Boundary Conditions

This example shows how to solve a multipoint boundary value problem, where the solution of interest satisfies conditions inside the interval of integration.

For x in $[0, \lambda]$, consider the equations

$$v' = \frac{C - 1}{n},$$

$$C' = \frac{\mathrm{vC} - \min(x, 1)}{\eta}.$$

The known parameters of the problem are n, κ, $\lambda > 1$, and $\eta = \frac{\lambda^2}{n \cdot \kappa^2}$.

The term $\min(x, 1)$ in the equation for $C'(x)$ is not smooth at $x = 1$, so the problem cannot be solved directly. Instead, you can break the problem into two: one set in the interval $[0, 1]$, and the other set in the interval $[1, \lambda]$. The connection between the two regions is that the solutions must be continuous at $x = 1$. The solution must also satisfy the boundary conditions

$$v(0) = 0,$$

$$C(\lambda) = 1.$$

The equations for each region are

Region 1: $0 \le x \le 1$

$$v' = \frac{C - 1}{n},$$

$$C' = \frac{\mathrm{vC} - x}{\eta}.$$

Region 2: $1 \le x \le \lambda$

$$v' = \frac{C - 1}{n},$$

$$C' = \frac{\mathrm{vC} - 1}{\eta}.$$

The interface point $x = 1$ is included in both regions. At this point, the solver produces both *left* and *right* solutions, which must be equal to ensure continuity of the solution.

To solve this system of equations in MATLAB, you need to code the equations, boundary conditions, and initial guess before calling the boundary value problem solver `bvp5c`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equations

The equations for $v'(x)$ and $C'(x)$ depend on the region being solved. For multipoint boundary value problems the derivative function must accept a third input argument `region`, which is used to identify the region where the derivative is being evaluated. The solver numbers the regions from left to right, starting with 1.

Create a function to code the equations with the signature `dydx = f(x,y,region,p)`, where:

- `x` is the independent variable.
- `y` is the dependent variable.
- `dydx(1)` gives the equation for $v'(x)$, and `dydx(2)` the equation for $C'(x)$.
- `region` is the number of the region where the derivative is being computed (in this two-region problem, `region` is `1` or `2`).
- `p` is a vector containing the values of the constant parameters $[n, \kappa, \lambda, \eta]$.

Use a switch statement to return different equations depending on the region being solved. The function is

```
function dydx = f(x,y,region,p) % equations being solved
n = p(1);
eta = p(4);

dydx = zeros(2,1);
dydx(1) = (y(2) - 1)/n;

switch region
    case 1    % x in [0 1]
        dydx(2) = (y(1)*y(2) - x)/eta;
    case 2    % x in [1 lambda]
        dydx(2) = (y(1)*y(2) - 1)/eta;
end
end
```

Note: All functions are included as local functions at the end of the example.

Code Boundary Conditions

Solving two first-order differential equations in two regions requires four boundary conditions. Two of these conditions come from the original problem:

$$v(0) = 0,$$

$$C(\lambda) - 1 = 0.$$

The other two conditions enforce the continuity of the left and right solutions at the interface point $x = 1$:

$$v_L(1) - v_R(1) = 0,$$

$$C_L(1) - C_R(1) = 0.$$

For multipoint BVPs, the arguments of the boundary conditions function `YL` and `YR` become matrices. In particular, the `k`th column `YL(:,k)` is the solution at the left boundary of the `k`th region. Similarly, `YR(:,k)` is the solution at the right boundary of the `k`th region.

In this problem, $y(0)$ is approximated by `YL(:,1)`, while $y(\lambda)$ is approximated by `YR(:,end)`. Continuity of the solution at $x = 1$ requires that `YR(:,1) = YL(:,2)`.

The function that encodes the residual value of the four boundary conditions is

```
function res = bc(YL,YR)
res = [YL(1,1)              % v(0) = 0
```

```
       YR(1,1) - YL(1,2)      % Continuity of v(x) at x=1
       YR(2,1) - YL(2,2)      % Continuity of C(x) at x=1
       YR(2,end) - 1];        % C(lambda) = 1
end
```

Form Initial Guess

For multipoint BVPs, the boundary conditions are automatically applied at the beginning and end of the interval of integration. However, you must specify double entries in `xmesh` for the other interface points. A simple guess that satisfies the boundary conditions is the constant guess `y = [1; 1]`.

```
xc = 1;
xmesh = [0 0.25 0.5 0.75 xc xc 1.25 1.5 1.75 2];
yinit = [1; 1];
sol = bvpinit(xmesh,yinit);
```

Solve Equation

Define the values of the constant parameters and put them in the vector `p`. Provide the function to `bvp5c` with the syntax `@(x,y,r) f(x,y,r,p)` to provide the vector of parameters.

Calculate the solution for several values of κ, using each solution as the initial guess for the next. For each value of κ, calculate the value of the osmolarity $\mathrm{Os} = \frac{1}{v(\lambda)}$. For each iteration of the loop, compare the computed value with the approximate analytical solution.

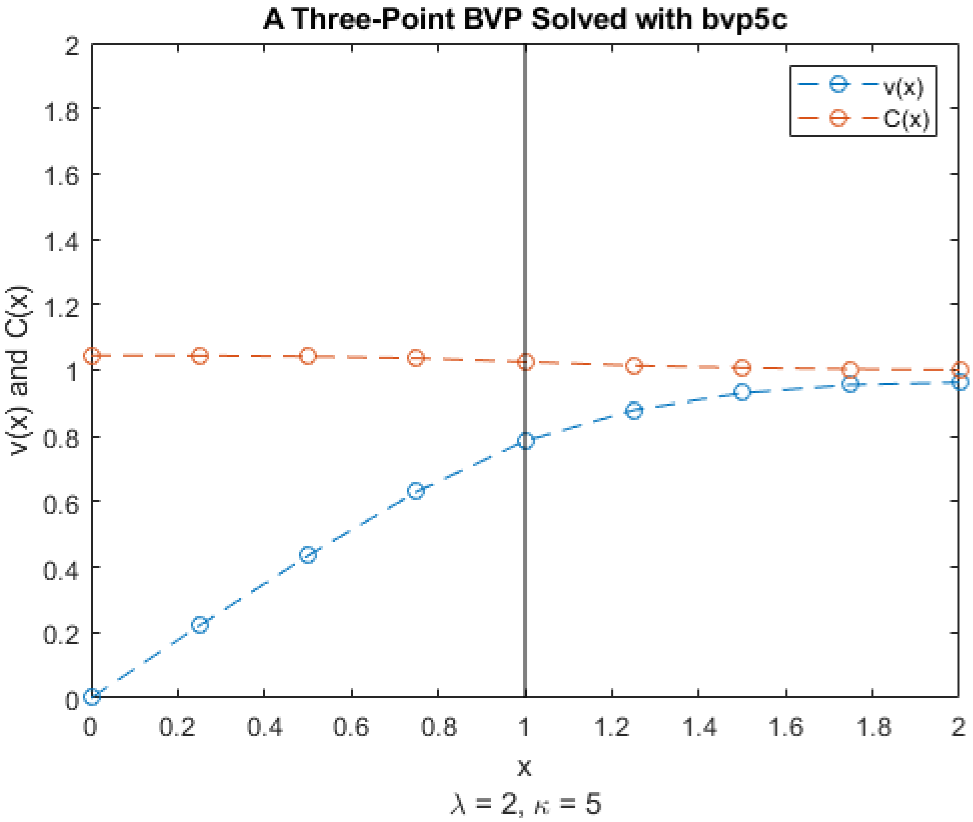
```
lambda = 2;
n = 5e-2;
for kappa = 2:5
   eta = lambda^2/(n*kappa^2);
   p = [n kappa lambda eta];
   sol = bvp5c(@(x,y,r) f(x,y,r,p), @bc, sol);
   K2 = lambda*sinh(kappa/lambda)/(kappa*cosh(kappa));
   approx = 1/(1 - K2);
   computed = 1/sol.y(1,end);
   fprintf('  %2i    %10.3f     %10.3f \n',kappa,computed,approx);
end
```

```
   2         1.462          1.454
   3         1.172          1.164
   4         1.078          1.071
   5         1.039          1.034
```

Plot Solution

Plot the solution components for $v(x)$ and $C(x)$ and a vertical line at the interface point $x = 1$. The displayed solution for $\kappa = 5$ results from the final iteration of the loop.

```
plot(sol.x,sol.y(1,:),'--o',sol.x,sol.y(2,:),'--o')
line([1 1], [0 2], 'Color', 'k')
legend('v(x)','C(x)')
title('A Three-Point BVP Solved with bvp5c')
xlabel({'x', '\lambda = 2, \kappa = 5'})
ylabel('v(x) and C(x)')
```



Local Functions

Listed here are the local helper functions that the BVP solver `bvp5c` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydx = f(x,y,region,p) % equations being solved
n = p(1);
eta = p(4);

dydx = zeros(2,1);
dydx(1) = (y(2) - 1)/n;

switch region
    case 1    % x in [0 1]
        dydx(2) = (y(1)*y(2) - x)/eta;
    case 2    % x in [1 lambda]
        dydx(2) = (y(1)*y(2) - 1)/eta;
end
end
%-------------------------------------------
function res = bc(YL,YR) % boundary conditions
res = [YL(1,1)               % v(0) = 0
       YR(1,1) - YL(1,2)     % Continuity of v(x) at x=1
       YR(2,1) - YL(2,2)     % Continuity of C(x) at x=1
       YR(2,end) - 1];       % C(lambda) = 1
```

```
end
%-------------------------------------------
```

See Also

bvp4c | bvp5c | bvpinit

More About

- “Solving Boundary Value Problems” on page 12-2

Partial Differential Equations (PDEs)

Solving Partial Differential Equations

In a partial differential equation (PDE), the function being solved for depends on several variables, and the differential equation can include partial derivatives taken with respect to each of the variables. Partial differential equations are useful for modelling waves, heat flow, fluid dispersion, and other phenomena with spatial behavior that changes over time.

What Types of PDEs Can You Solve with MATLAB?

The MATLAB PDE solver pdepe solves initial-boundary value problems for systems of PDEs in one spatial variable x and time t. You can think of these as ODEs of one variable that also change with respect to time.

pdepe uses an informal classification for the 1-D equations it solves:

- Equations with a time derivative are parabolic. An example is the heat equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$.
- Equations without a time derivative are elliptic. An example is the Laplace equation $\frac{\partial^2 u}{\partial x^2} = 0$.

pdepe requires at least one parabolic equation in the system. In other words, at least one equation in the system must include a time derivative.

pdepe also solves certain 2-D and 3-D problems that reduce to 1-D problems due to angular symmetry (see the argument description for the symmetry constant m for more information).

Partial Differential Equation Toolbox extends this functionality to generalized problems in 2-D and 3-D with Dirichlet and Neumann boundary conditions.

Solving 1-D PDEs

A 1-D PDE includes a function $u(x,t)$ that depends on time t and one spatial variable x. The MATLAB PDE solver pdepe solves systems of 1-D parabolic and elliptic PDEs of the form

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

The equation has the properties:

- The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$.
- The spatial interval $[a, b]$ must be finite.
- m can be 0, 1, or 2, corresponding to *slab*, *cylindrical*, or *spherical* symmetry, respectively. If $m > 0$, then $a \ge 0$ must also hold.
- The coefficient $f\left(x,t,u,\frac{\partial u}{\partial x}\right)$ is a flux term and $s\left(x,t,u,\frac{\partial u}{\partial x}\right)$ is a source term.
- The flux term must depend on the partial derivative $\partial u/\partial x$.

The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c\left(x,t,u,\frac{\partial u}{\partial x}\right)$. The diagonal elements of this matrix are either zero or positive. An element that is zero corresponds to an elliptic equation, and any other element corresponds to a parabolic equation. There must be at least one parabolic equation. An element of c that corresponds to a parabolic equation can vanish at isolated values of x if they are mesh points (points where the solution is evaluated). Discontinuities in c and s due to material interfaces are permitted provided that a mesh point is placed at each interface.

Solution Process

To solve PDEs with `pdepe`, you must define the equation coefficients for c, f, and s, the initial conditions, the behavior of the solution at the boundaries, and a mesh of points to evaluate the solution on. The function call `sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)` uses this information to calculate a solution on the specified mesh:

- `m` is the symmetry constant.
- `pdefun` defines the equations being solved.
- `icfun` defines the initial conditions.
- `bcfun` defines the boundary conditions.
- `xmesh` is a vector of spatial values for x.
- `tspan` is a vector of time values for t.

Together, the `xmesh` and `tspan` vectors form a 2-D grid that `pdepe` evaluates the solution on.

Equations

You must express the PDEs in the standard form expected by `pdepe`. Written in this form, you can read off the values of the coefficients c, f, and s.

In MATLAB you can code the equations with a function of the form

```
function [c,f,s] = pdefun(x,t,u,dudx)
c = 1;
f = dudx;
s = 0;
end
```

In this case `pdefun` defines the equation $\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$. If there are multiple equations, then c, f, and s are vectors with each element corresponding to one equation.

Initial Conditions

At the initial time $t = t_0$, for all x, the solution components satisfy initial conditions of the form

$$u(x,t_0) = u_0(x)\,.$$

In MATLAB you can code the initial conditions with a function of the form

```
function u0 = icfun(x)
u0 = 1;
end
```

In this case `u0 = 1` defines an initial condition of $u_0(x,t_0) = 1$. If there are multiple equations, then `u0` is a vector with each element defining the initial condition of one equation.

Boundary Conditions

At the boundary $x = a$ or $x = b$, for all t, the solution components satisfy boundary conditions of the form

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0.$$

$q(x,t)$ is a diagonal matrix with elements that are either zero or never zero. Note that the boundary conditions are expressed in terms of the flux f, rather than the partial derivative of u with respect to x. Also, of the two coefficients $p(x,t,u)$ and $q(x,t)$, only p can depend on u.

In MATLAB you can code the boundary conditions with a function of the form

```
function [pL,qL,pR,qR] = bcfun(xL,uL,xR,uR,t)
pL = uL;
qL = 0;
pR = uR - 1;
qR = 0;
end
```

pL and qL are the coefficients for the left boundary, while pR and qR are the coefficients for the right boundary. In this case bcfun defines the boundary conditions

$$u_L(x_L,t) = 0$$

$$u_R(x_R,t) = 1$$

If there are multiple equations, then the outputs pL, qL, pR, and qR are vectors with each element defining the boundary condition of one equation.

Integration Options

The default integration properties in the MATLAB PDE solver are selected to handle common problems. In some cases, you can improve solver performance by overriding these default values. To do this, use odeset to create an options structure. Then, pass the structure to pdepe as the last input argument:

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
```

Of the options for the underlying ODE solver ode15s, only those shown in the following table are available for pdepe.

Category	Option Name
Error control	RelTol, AbsTol, NormControl
Step-size	InitialStep, MaxStep
Event logging	Events

Evaluating the Solution

After you solve an equation with pdepe, MATLAB returns the solution as a 3-D array sol, where sol(i,j,k) contains the kth component of the solution evaluated at t(i) and x(j). In general, you can extract the kth solution component with the command u = sol(:,:,k).

The time mesh you specify is used purely for output purposes, and does not affect the internal time steps taken by the solver. However, the spatial mesh you specify can affect the quality and speed of the solution. After solving an equation, you can use `pdeval` to evaluate the solution structure returned by `pdepe` with a different spatial mesh.

Example: The Heat Equation

An example of a parabolic PDE is the heat equation in one dimension:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}\,.$$

This equation describes the dissipation of heat for $0 \le x \le L$ and $t \ge 0$. The goal is to solve for the temperature $u(x,t)$. The temperature is initially a nonzero constant, so the initial condition is

$$u(x,0) = T_0\,.$$

Also, the temperature is zero at the left boundary, and nonzero at the right boundary, so the boundary conditions are

$$\begin{aligned} u(0,t) &= 0, \\ u(L,t) &= 1\,. \end{aligned}$$

To solve this equation in MATLAB, you need to code the equation, initial conditions, and boundary conditions, then select a suitable solution mesh before calling the solver `pdepe`. You either can include the required functions as local functions at the end of a file (as in this example), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Before you can code the equation, you need to make sure that it is in the form that the `pdepe` solver expects:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

In this form, the heat equation is

$$1 \cdot \frac{\partial u}{\partial t} = x^0 \frac{\partial}{\partial x}\left(x^0 \frac{\partial u}{\partial x}\right) + 0\,.$$

So the values of the coefficients are as follows:

- $m = 0$
- $c = 1$
- $f = \frac{\partial u}{\partial x}$
- $s = 0$

The value of m is passed as an argument to `pdepe`, while the other coefficients are encoded in a function for the equation, which is

```
function [c,f,s] = heatpde(x,t,u,dudx)
c = 1;
f = dudx;
```

```
s = 0;
end
```

(Note: All functions are included as local functions at the end of the example.)

Code Initial Condition

The initial condition function for the heat equation assigns a constant value for u_0. This function must accept an input for x, even if it is unused.

```
function u0 = heatic(x)
u0 = 0.5;
end
```

Code Boundary Conditions

The standard form for the boundary conditions expected by the pdepe solver is

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0\,.$$

Written in this form, the boundary conditions for this problem are

$$u(0,t) + (0 \cdot f) = 0,$$

$$(u(L,t) - 1) + (0 \cdot f) = 0\,.$$

So the values for p and q are

- $p_L = u_L, \qquad q_L = 0\,.$
- $p_R = u_R - 1, \quad q_R = 0.$

The corresponding function is then

```
function [pl,ql,pr,qr] = heatbc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = ur - 1;
qr = 0;
end
```

Select Solution Mesh

Use a spatial mesh of 20 points and a time mesh of 30 points. Since the solution rapidly reaches a steady state, the time points near $t = 0$ are more closely spaced together to capture this behavior in the output.

```
L = 1;
x = linspace(0,L,20);
t = [linspace(0,0.05,20), linspace(0.5,5,10)];
```

Solve Equation

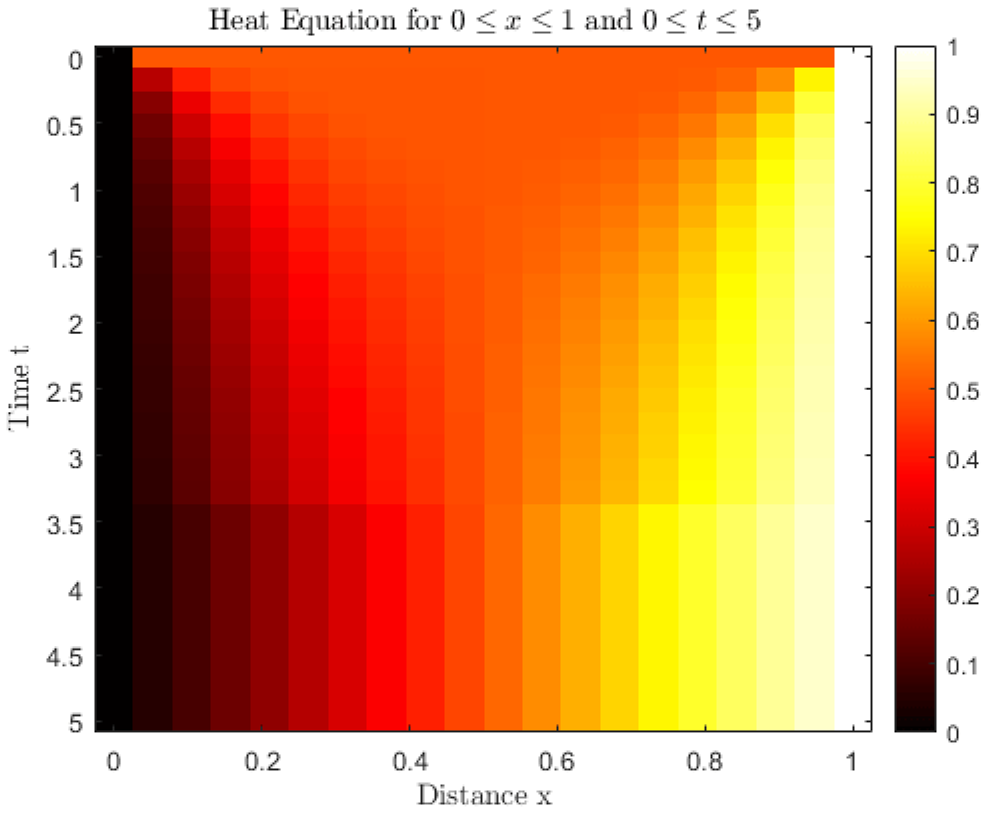
Finally, solve the equation using the symmetry m, the PDE equation, the initial condition, the boundary conditions, and the meshes for x and t.

```
m = 0;
sol = pdepe(m,@heatpde,@heatic,@heatbc,x,t);
```

Plot Solution

Use `imagesc` to visualize the solution matrix.

```
colormap hot
imagesc(x,t,sol)
colorbar
xlabel('Distance x','interpreter','latex')
ylabel('Time t','interpreter','latex')
title('Heat Equation for $0 \le x \le 1$ and $0 \le t \le 5$','interpreter','latex')
```



Local Functions

```
function [c,f,s] = heatpde(x,t,u,dudx)
c = 1;
f = dudx;
s = 0;
end
function u0 = heatic(x)
u0 = 0.5;
end
function [pl,ql,pr,qr] = heatbc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = ur - 1;
```

```
qr = 0;
end
```

PDE Examples and Files

Several available example files serve as excellent starting points for most common 1-D PDE problems. To explore and run examples, use the Differential Equations Examples app. To run this app, type

```
odeexamples
```

To open an individual file for editing, type

```
edit exampleFileName.m
```

To run an example, type

```
exampleFileName
```

This table contains a list of the available PDE example files.

Example File	Description	Example Link
pdex1	Simple PDE that illustrates the formulation, computation, and plotting of the solution.	"Solve Single PDE" on page 13-9
pdex2	Problem that involves discontinuities.	"Solve PDE with Discontinuity" on page 13-16
pdex3	Problem that requires computing values of the partial derivative.	"Solve PDE and Compute Partial Derivatives" on page 13-23
pdex4	System of two PDEs whose solution has boundary layers at both ends of the interval and changes rapidly for small *t*.	"Solve System of PDEs" on page 13-32
pdex5	System of PDEs with step functions as initial conditions.	"Solve System of PDEs with Initial Condition Step Functions" on page 13-39

References

[1] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp. 1–32.

See Also

bvp4c | ode45 | odeset | pdepe | pdeval

More About

- "Solve Single PDE" on page 13-9
- "Solve System of PDEs" on page 13-32

Solve Single PDE

This example shows how to formulate, compute, and plot the solution to a single PDE.

Consider the partial differential equation

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} .$$

The equation is defined on the interval $0 \leq x \leq 1$ for times $t \geq 0$. At $t = 0$, the solution satisfies the initial condition

$$u(x,0) = \sin(\pi x) .$$

Also, at $x = 0$ and $x = 1$, the solution satisfies the boundary conditions

$$u(0,t) = 0,$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0 .$$

To solve this equation in MATLAB, you need to code the equation, the initial conditions, and the boundary conditions, then select a suitable solution mesh before calling the solver `pdepe`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Before you can code the equation, you need to rewrite it in a form that the `pdepe` solver expects. The standard form that `pdepe` expects is

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

Written in this form, the PDE becomes

$$\pi^2\frac{\partial u}{\partial t} = x^0\frac{\partial}{\partial x}\left(x^0\frac{\partial u}{\partial x}\right) + 0.$$

With the equation in the proper form you can read off the relevant terms:

$$m = 0$$

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right) = \pi^2$$

$$f\left(x,t,u,\frac{\partial u}{\partial x}\right) = \frac{\partial u}{\partial x}$$

$$s\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0$$

Now you can create a function to code the equation. The function should have the signature `[c,f,s] = pdex1pde(x,t,u,dudx)`:

- x is the independent spatial variable.
- t is the independent time variable.
- u is the dependent variable being differentiated with respect to x and t.
- dudx is the partial spatial derivative $\partial u/\partial x$.
- The outputs c, f, and s correspond to coefficients in the standard PDE equation form expected by pdepe. These coefficients are coded in terms of the input variables x, t, u, and dudx.

As a result, the equation in this example can be represented by the function:

```
function [c,f,s] = pdex1pde(x,t,u,dudx)
c = pi^2;
f = dudx;
s = 0;
end
```

(Note: All functions are included as local functions at the end of the example.)

Code Initial Condition

Next, write a function that returns the initial condition. The initial condition is applied at the first time value tspan(1). The function should have the signature u0 = pdex1ic(x).

The corresponding function is

```
function u0 = pdex1ic(x)
u0 = sin(pi*x);
end
```

Code Boundary Conditions

Now, write a function that evaluates the boundary conditions. For problems posed on the interval $a \le x \le b$, the boundary conditions apply for all t and either $x = a$ or $x = b$. The standard form for the boundary conditions expected by the solver is

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0 .$$

Rewrite the boundary conditions in this standard form and read off the coefficient values.

For $x = 0$, the equation is

$$u(0,t) = 0 \rightarrow u + 0 \cdot \frac{\partial u}{\partial x} = 0 .$$

The coefficients are:

- $p(0,t,u) = u$
- $q(0,t) = 0$

For $x = 1$, the equation is

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0 \rightarrow \pi e^{-t} + 1 \cdot \frac{\partial u}{\partial x}(1,t) = 0 .$$

The coefficients are:

- $p(1,t,u) = \pi e^{-t}$
- $q(1,t) = 1$

Since the boundary condition function is expressed in terms of $f\left(x,t,u,\frac{\partial u}{\partial x}\right)$, and this term is already defined in the main PDE function, you do not need to specify this piece of the equation in the boundary condition function. You need only specify the values of $p(x,t,u)$ and $q(x,t)$ at each boundary.

The boundary function should use the function signature `[pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)`:

- The inputs `xl` and `ul` correspond to u and x for the left boundary.
- The inputs `xr` and `ur` correspond to u and x for the right boundary.
- `t` is the independent time variable.
- The outputs `pl` and `ql` correspond to $p(x,t,u)$ and $q(x,t)$ for the left boundary ($x = 0$ for this problem).
- The outputs `pr` and `qr` correspond to $p(x,t,u)$ and $q(x,t)$ for the right boundary ($x = 1$ for this problem).

The boundary conditions in this example are represented by the function:

```
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
end
```

Select Solution Mesh

Before solving the equation you need to specify the mesh points (t,x) at which you want `pdepe` to evaluate the solution. Specify the points as vectors `t` and `x`. The vectors `t` and `x` play different roles in the solver. In particular, the cost and accuracy of the solution depend strongly on the length of the vector `x`. However, the computation is much less sensitive to the values in the vector `t`.

For this problem, use a mesh with 20 equally spaced points in the spatial interval [0,1] and five values of `t` from the time interval [0,2].

```
x = linspace(0,1,20);
t = linspace(0,2,5);
```

Solve Equation

Finally, solve the equation using the symmetry `m`, the PDE equation, the initial conditions, the boundary conditions, and the meshes for `x` and `t`.

```
m = 0;
sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
```

`pdepe` returns the solution in a 3-D array `sol`, where `sol(i,j,k)` approximates the `k`th component of the solution u_k evaluated at `t(i)` and `x(j)`. The size of `sol` is `length(t)`-by-`length(x)`-by-`length(u0)`, since `u0` specifies an initial condition for each solution component. For this problem, `u` has only one component, so `sol` is a 5-by-20 matrix, but in general you can extract the `k`th solution component with the command `u = sol(:,:,k)`.

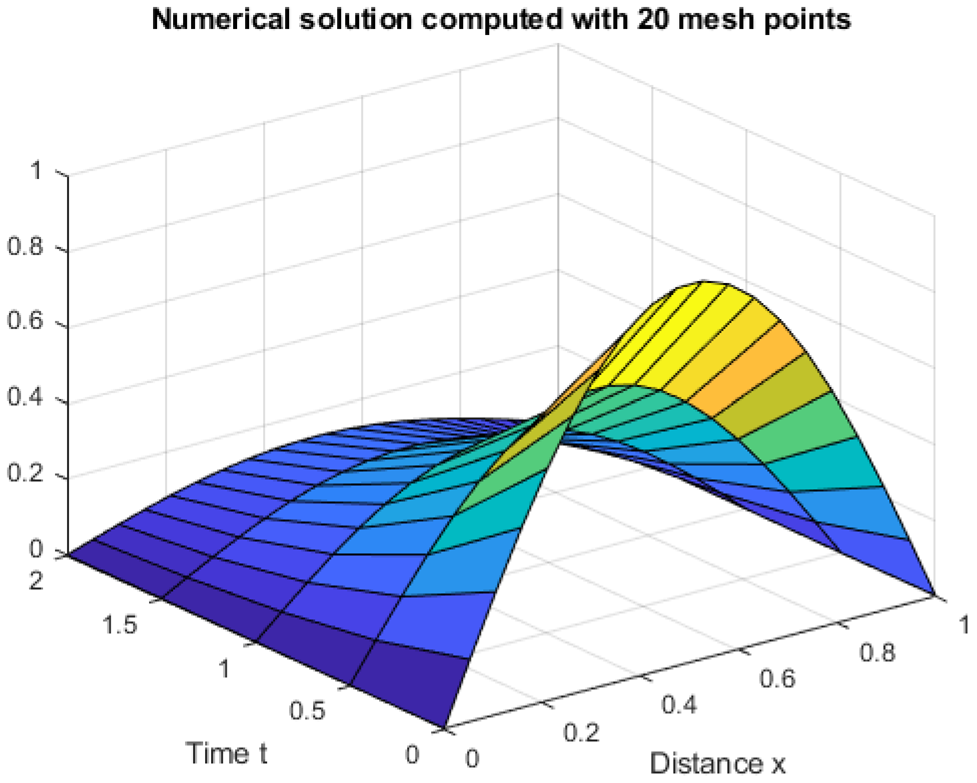
Extract the first solution component from `sol`.

```
u = sol(:,:,1);
```

Plot Solution

Create a surface plot of the solution.

```
surf(x,t,u)
title('Numerical solution computed with 20 mesh points')
xlabel('Distance x')
ylabel('Time t')
```

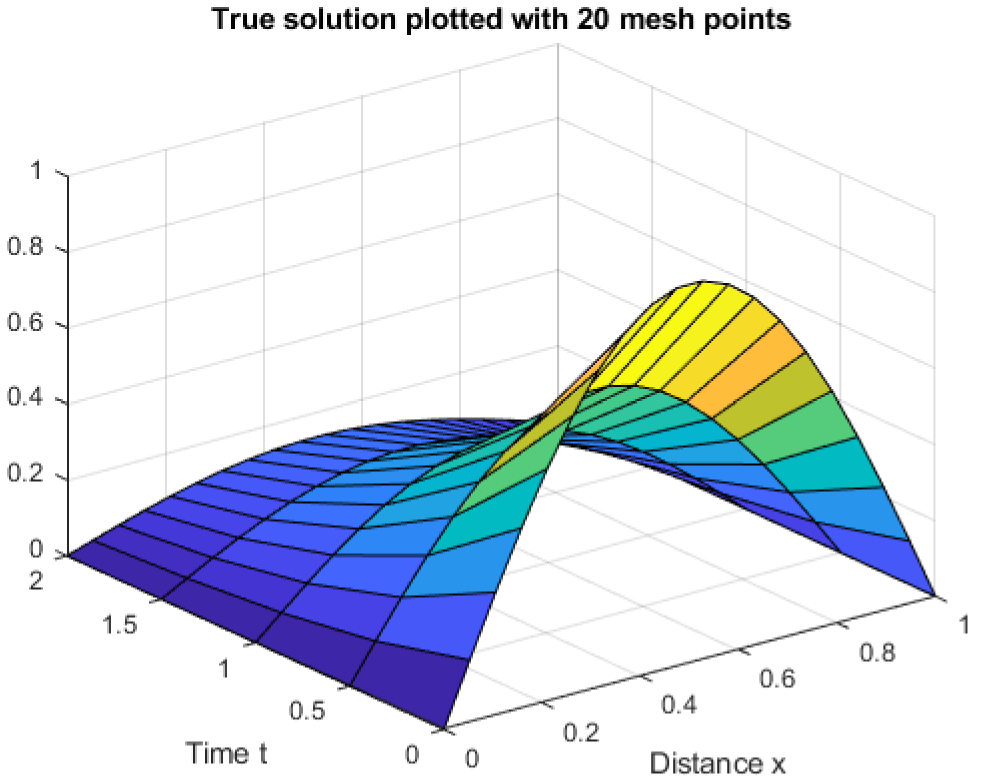


The initial condition and boundary conditions of this problem were chosen so that there would be an analytical solution, given by

$$u(x, t) = e^{-t}\sin(\pi x) .$$

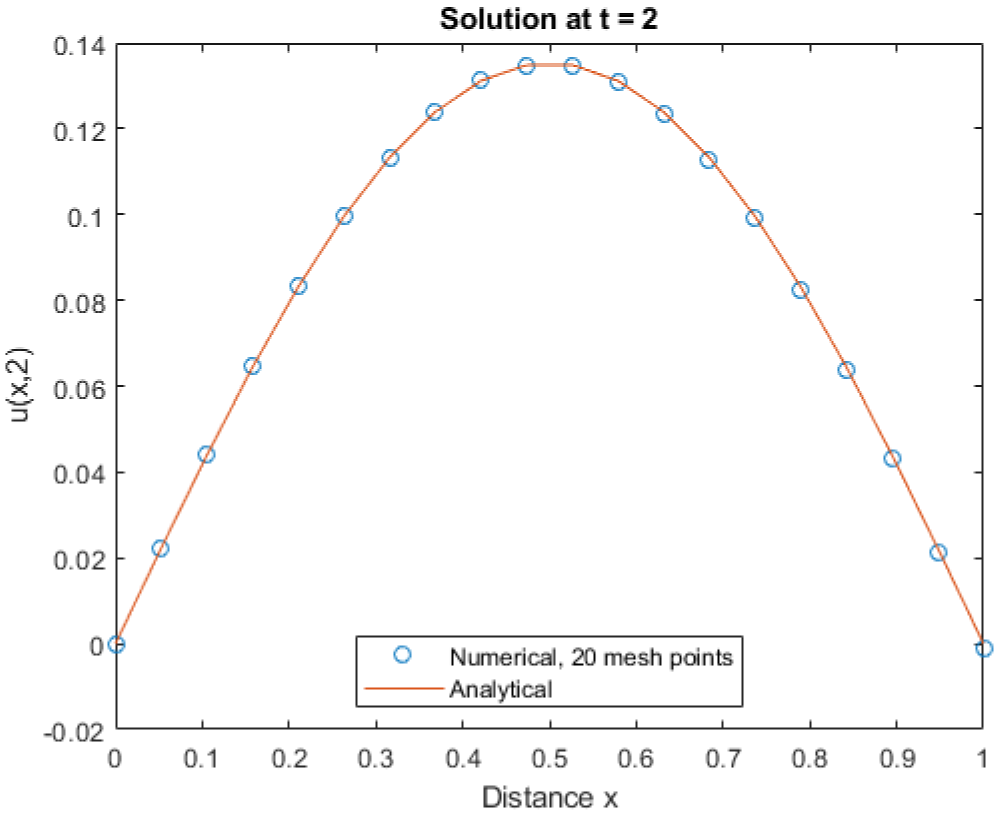
Plot the analytical solution with the same mesh points.

```
surf(x,t,exp(-t)'*sin(pi*x))
title('True solution plotted with 20 mesh points')
xlabel('Distance x')
ylabel('Time t')
```



Now, compare the numerical and analytical solutions at t_f, the final value of t. In this example $t_f = 2$.

```
plot(x,u(end,:),'o',x,exp(-t(end))*sin(pi*x))
title('Solution at t = 2')
legend('Numerical, 20 mesh points','Analytical','Location','South')
xlabel('Distance x')
ylabel('u(x,2)')
```



Local Functions

Listed here are the local helper functions that the PDE solver pdepe calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function [c,f,s] = pdex1pde(x,t,u,dudx) % Equation to solve
c = pi^2;
f = dudx;
s = 0;
end
%----------------------------------------------
function u0 = pdex1ic(x) % Initial conditions
u0 = sin(pi*x);
end
%----------------------------------------------
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t) % Boundary conditions
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
end
%----------------------------------------------
```

See Also

pdepe

More About

- “Solving Partial Differential Equations” on page 13-2
- “Solve PDE with Discontinuity” on page 13-16

Solve PDE with Discontinuity

This example shows how to solve a PDE that interfaces with a material. The material interface creates a discontinuity in the problem at $x = 0 . 5$, and the initial condition has a discontinuity at the right boundary $x = 1$.

Consider the piecewise PDE

$$\begin{cases} \dfrac{\partial u}{\partial t} = x^{-2}\dfrac{\partial}{\partial x}\left(x^2\, 5\dfrac{\partial u}{\partial x}\right) - 1000e^u \ (0 \le x \le 0 . 5) \\ \quad \dfrac{\partial u}{\partial t} = x^{-2}\dfrac{\partial}{\partial x}\left(x^2\,\dfrac{\partial u}{\partial x}\right) - e^u \qquad (0 . 5 \le x \le 1) \end{cases}$$

The initial conditions are

$$u(x, 0) = 0 \ (0 \le x < 1),$$
$$u(1, 0) = 1 \ (x = 1) .$$

The boundary conditions are

$$\frac{\partial u}{\partial x} = 0 \ (x = 0),$$
$$u(1, t) = 1 \ (x = 1) .$$

To solve this equation in MATLAB, you need to code the equation, the initial conditions, and the boundary conditions, then select a suitable solution mesh before calling the solver `pdepe`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Before you can code the equation, you need to make sure that it is in a form that the `pdepe` solver expects. The standard form that `pdepe` expects is

$$c\left(x, t, u, \tfrac{\partial u}{\partial x}\right)\tfrac{\partial u}{\partial t} = x^{-m}\tfrac{\partial}{\partial x}\left(x^m f\left(x, t, u, \tfrac{\partial u}{\partial x}\right)\right) + s\left(x, t, u, \tfrac{\partial u}{\partial x}\right).$$

In this case, the PDE is in the proper form, so you can read off the values of the coefficients.

$$\begin{cases} \tfrac{\partial u}{\partial t} = x^{-2}\tfrac{\partial}{\partial x}\left(x^2\, 5\tfrac{\partial u}{\partial x}\right) - 1000e^u \ (0 \le x \le 0 . 5) \\ \quad \tfrac{\partial u}{\partial t} = x^{-2}\tfrac{\partial}{\partial x}\left(x^2\,\tfrac{\partial u}{\partial x}\right) - e^u \qquad (0 . 5 \le x \le 1) \end{cases}$$

The values for the flux term $f\left(x, t, u, \tfrac{\partial u}{\partial x}\right)$ and source term $s\left(x . t, u, \tfrac{\partial u}{\partial x}\right)$ change depending on the value of x. The coefficients are:

$$m = 2$$

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right) = 1$$

$$\begin{cases} f\left(x, t, u, \dfrac{\partial u}{\partial x}\right) = 5\dfrac{\partial u}{\partial x} \ (0 \le x \le 0 . 5) \\ f\left(x, t, u, \dfrac{\partial u}{\partial x}\right) = \dfrac{\partial u}{\partial x} \ \ (0 . 5 \le x \le 1) \end{cases}$$

$$\begin{cases} s\left(x,t,u,\frac{\partial u}{\partial x}\right) = -1000e^{u} \ (0 \le x \le 0.5) \\ s\left(x,t,u,\frac{\partial u}{\partial x}\right) = -e^{u} \qquad (0.5 \le x \le 1) \end{cases}$$

Now you can create a function to code the equation. The function should have the signature `[c,f,s] = pdex2pde(x,t,u,dudx)`:

- `x` is the independent spatial variable.
- `t` is the independent time variable.
- `u` is the dependent variable being differentiated with respect to `x` and `t`.
- `dudx` is the partial spatial derivative $\partial u / \partial x$.
- The outputs `c`, `f`, and `s` correspond to coefficients in the standard PDE equation form expected by `pdepe`. These coefficients are coded in terms of the input variables `x`, `t`, `u`, and `dudx`.

As a result, the equation in this example can be represented by the function:

```
function [c,f,s] = pdex2pde(x,t,u,dudx)
c = 1;
if x <= 0.5
    f = 5*dudx;
    s = -1000*exp(u);
else
    f = dudx;
    s = -exp(u);
end
end
```

(Note: All functions are included as local functions at the end of the example.)

Code Initial Conditions

Next, write a function that returns the initial conditions. The initial condition is applied at the first time value and provides the value of $u(x,t_0)$ for any value of x. Use the function signature `u0 = pdex2ic(x)` to write the function.

The initial conditions are

$$\begin{aligned} &u(x,0) = 0 \qquad (0 \le x < 1), \\ &u(1,0) = 1 \qquad (x = 1)\,. \end{aligned}$$

The corresponding function is

```
function u0 = pdex2ic(x)
if x < 1
    u0 = 0;
else
    u0 = 1;
end
end
```

Code Boundary Conditions

Now, write a function that evaluates the boundary conditions. For problems posed on the interval $a \le x \le b$, the boundary conditions apply for all t and either $x = a$ or $x = b$. The standard form for the boundary conditions expected by the solver is

$$p(x,t,u) + q(x,t) f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0 \,.$$

Since this example has spherical symmetry ($m = 2$), the `pdepe` solver automatically enforces the left boundary condition to bound the solution at the origin, and ignores any conditions specified for the left boundary in the boundary function. So for the left boundary condition, you can specify $p_L = q_L = 0$. For the right boundary condition, you can rewrite the boundary condition in the standard form and read off the coefficient values for p_R and q_R.

For $x = 1$, the equation is $u(1,t) = 1 \rightarrow (u-1) + 0 \cdot \frac{\partial u}{\partial x} = 0$. The coefficients are:

- $p_R(1,t,u) = u - 1$
- $q_R(1,t) = 0$

The boundary function should use the function signature `[pl,ql,pr,qr] = pdex2bc(xl,ul,xr,ur,t)`:

- The inputs `xl` and `ul` correspond to u and x for the left boundary.
- The inputs `xr` and `ur` correspond to u and x for the right boundary.
- `t` is the independent time variable.
- The outputs `pl` and `ql` correspond to $p_L(x,t,u)$ and $q_L(x,t)$ for the left boundary ($x = 0$ for this problem).
- The outputs `pr` and `qr` correspond to $p_R(x,t,u)$ and $q_R(x,t)$ for the right boundary ($x = 1$ for this problem).

The boundary conditions in this example are represented by the function:

```
function [pl,ql,pr,qr] = pdex2bc(xl,ul,xr,ur,t)
pl = 0;
ql = 0;
pr = ur - 1;
qr = 0;
end
```

Select Solution Mesh

The spatial mesh should include several values near $x = 0.5$ to account for the discontinuous interface, as well as points near $x = 1$ because of the inconsistent initial value ($u(1,0) = 1$) and boundary value ($u(1,t) = 0$) at that point. The solution changes rapidly for small t, so use a time step that can resolve this sharp change.

```
x = [0 0.1 0.2 0.3 0.4 0.45 0.475 0.5 0.525 0.55 0.6 0.7 0.8 0.9 0.95 0.975 0.99 1];
t = [0 0.001 0.005 0.01 0.05 0.1 0.5 1];
```

Solve Equation

Finally, solve the equation using the symmetry m, the PDE equation, the initial conditions, the boundary conditions, and the meshes for x and t.

```
m = 2;
sol = pdepe(m,@pdex2pde,@pdex2ic,@pdex2bc,x,t);
```

pdepe returns the solution in a 3-D array sol, where sol(i,j,k) approximates the kth component of the solution u_k evaluated at t(i) and x(j). The size of sol is length(t)-by-length(x)-by-length(u0), since u0 specifies an initial condition for each solution component. For this problem, u has only one component, so sol is a 8-by-18 matrix, but in general you can extract the kth solution component with the command u = sol(:,:,k).

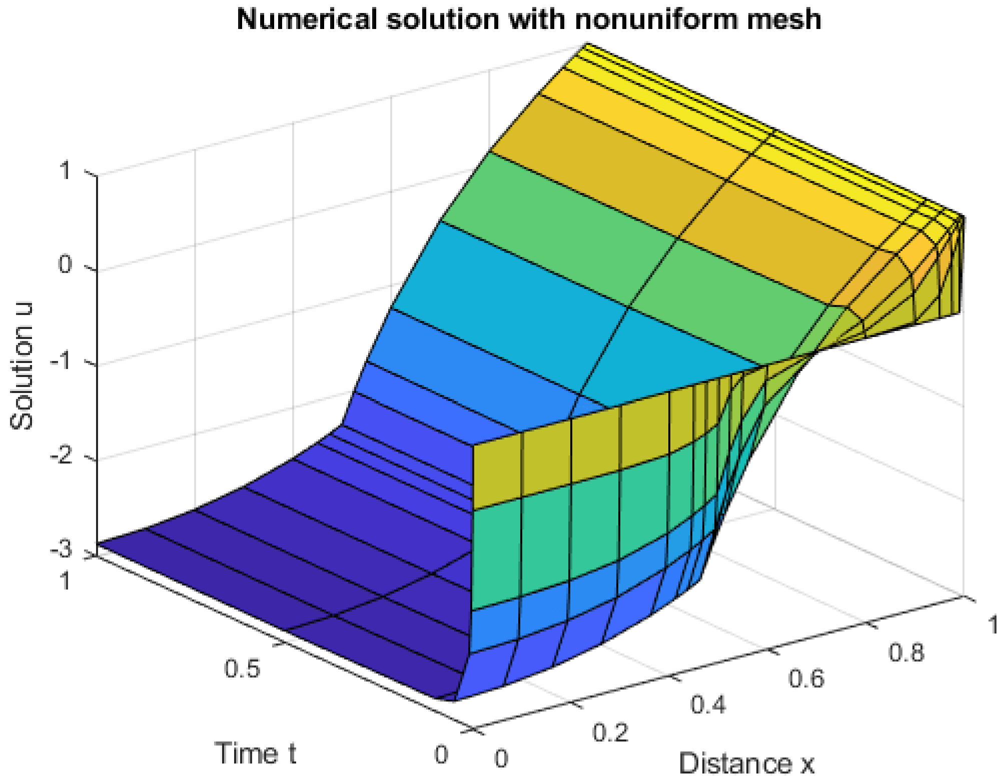
Extract the first solution component from sol.

```
u = sol(:,:,1);
```

Plot Solution

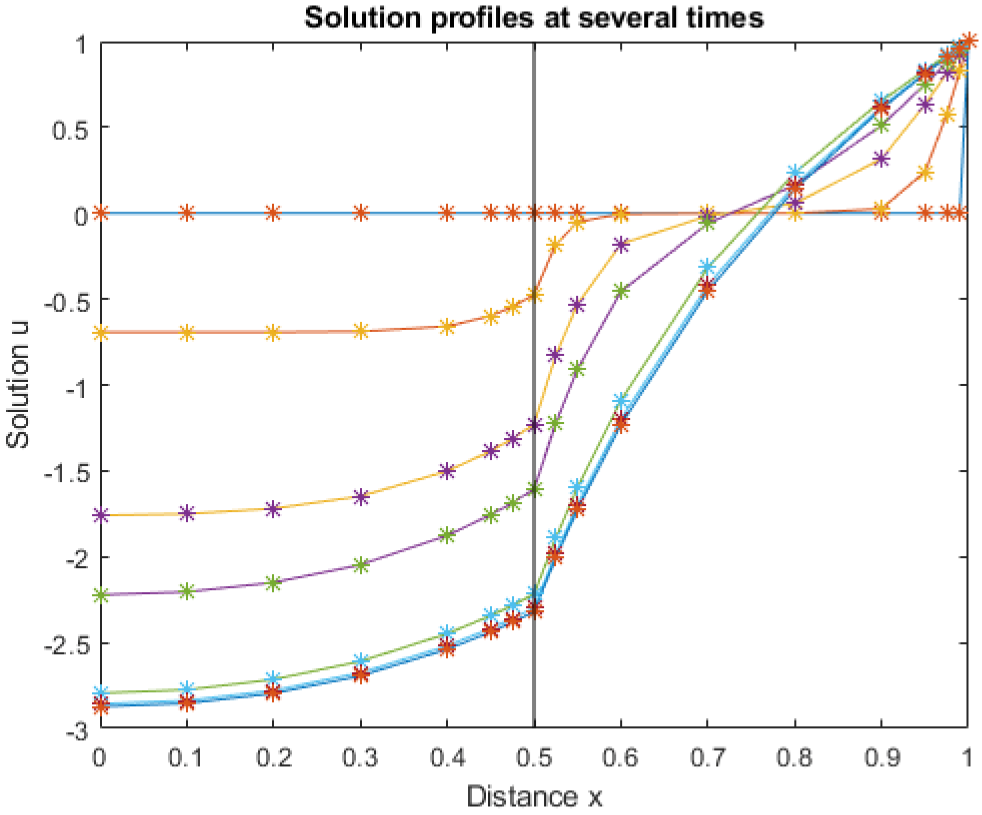
Create a surface plot of the solution u plotted at the selected mesh points for x and t. Since $m = 2$ the problem is posed in a spherical geometry with spherical symmetry, so the solution only changes in the radial x direction.

```
surf(x,t,u)
title('Numerical solution with nonuniform mesh')
xlabel('Distance x')
ylabel('Time t')
zlabel('Solution u')
```



Now, plot just x and u to get a side view of the contours in the surface plot. Add a line at $x = 0.5$ to highlight the effect of the material interface.

```
plot(x,u,x,u,'*')
line([0.5 0.5], [-3 1], 'Color', 'k')
xlabel('Distance x')
ylabel('Solution u')
title('Solution profiles at several times')
```



Local Functions

Listed here are the local helper functions that the PDE solver pdepe calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function [c,f,s] = pdex2pde(x,t,u,dudx) % Equation to solve
c = 1;
if x <= 0.5
    f = 5*dudx;
    s = -1000*exp(u);
else
    f = dudx;
    s = -exp(u);
end
end
%-----------------------------------------------
function u0 = pdex2ic(x) %Initial conditions
if x < 1
    u0 = 0;
else
    u0 = 1;
end
end
%-----------------------------------------------
function [pl,ql,pr,qr] = pdex2bc(xl,ul,xr,ur,t) % Boundary conditions
pl = 0;
ql = 0;
```

```
pr = ur - 1;
qr = 0;
end
%----------------------------------------------
```

See Also

pdepe

More About

- “Solving Partial Differential Equations” on page 13-2
- “Solve PDE and Compute Partial Derivatives” on page 13-23

Solve PDE and Compute Partial Derivatives

This example shows how to solve a transistor partial differential equation (PDE) and use the results to obtain partial derivatives that are part of solving a larger problem.

Consider the PDE

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} - \frac{D\eta}{L}\frac{\partial u}{\partial x}.$$

This equation arises in transistor theory [1], and $u(x,t)$ is a function describing the concentration of excess charge carriers (or *holes*) in the base of a PNP transistor. D and η are physical constants. The equation holds on the interval $0 \le x \le L$ for times $t \ge 0$.

The initial condition includes a constant K and is given by

$$u(x,0) = \frac{K\,L}{D}\left(\frac{1 - e^{-\eta(1 - x/L)}}{\eta}\right).$$

The problem has boundary conditions given by

$$u(0,t) = u(L,t) = 0\,.$$

For fixed x, the solution to the equation $u(x,t)$ describes the collapse of excess charge as $t \to \infty$. This collapse produces a current, called the *emitter discharge current*, which has another constant I_p*:*

$$I(t) = \left[\frac{I_p D}{K}\frac{\partial}{\partial x}u(x,t)\right]_{x\,=\,0}.$$

The equation is valid for $t > 0$ due to the inconsistency in the boundary values at $x = 0$ for $t = 0$ and $t > 0$. Since the PDE has a closed-form series solution for $u(x,t)$, you can calculate the emitter discharge current analytically as well as numerically, and compare the results.

To solve this problem in MATLAB, you need to code the PDE equation, initial conditions, and boundary conditions, then select a suitable solution mesh before calling the solver `pdepe`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Define Physical Constants

To keep track of the physical constants, create a structure array with fields for each one. When you later define functions for the equations, the initial condition, and the boundary conditions, you can pass in this structure as an extra argument so that the functions have access to the constants.

```
C.L = 1;
C.D = 0.1;
C.eta = 10;
C.K = 1;
C.Ip = 1;
```

Code Equation

Before you can code the equation, you need to make sure that it is in the form that the `pdepe` solver expects:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

In this form, the PDE is

$$\frac{\partial u}{\partial t} = x^0\,\frac{\partial}{\partial x}\left(x^0\,D\frac{\partial u}{\partial x}\right) - \frac{D\eta}{L}\frac{\partial u}{\partial x}.$$

So the values of the coefficients in the equation are

- $m = 0$ (Cartesian coordinates with no angular symmetry)
- $c\left(x,t,u,\frac{\partial u}{\partial x}\right) = 1$
- $f\left(x,t,u,\frac{\partial u}{\partial x}\right) = D\frac{\partial u}{\partial x}$
- $s\left(x,t,u,\frac{\partial u}{\partial x}\right) = -\frac{D\eta}{L}\frac{\partial u}{\partial x}$

Now you can create a function to code the equation. The function should have the signature `[c,f,s] = transistorPDE(x,t,u,dudx,C)`:

- `x` is the independent spatial variable.
- `t` is the independent time variable.
- `u` is the dependent variable being differentiated with respect to `x` and `t`.
- `dudx` is the partial spatial derivative $\partial u/\partial x$.
- `C` is an extra input containing the physical constants.
- The outputs `c`, `f`, and `s` correspond to coefficients in the standard PDE equation form expected by `pdepe`.

As a result, the equation in this example can be represented by the function:

```
function [c,f,s] = transistorPDE(x,t,u,dudx,C)
D = C.D;
eta = C.eta;
L = C.L;

c = 1;
f = D*dudx;
s = -(D*eta/L)*dudx;
end
```

(Note: All functions are included as local functions at the end of the example.)

Code Initial Condition

Next, write a function that returns the initial condition. The initial condition is applied at the first time value, and provides the value of $u(x,t_0)$ for any value of x. Use the function signature `u0 = transistorIC(x,C)` to write the function.

The initial condition is

$$u(x,0) = \frac{K\,L}{D}\left(\frac{1-e^{-\eta(1-x/L)}}{\eta}\right).$$

The corresponding function is

```
function u0 = transistorIC(x,C)
K = C.K;
L = C.L;
D = C.D;
eta = C.eta;

u0 = (K*L/D)*(1 - exp(-eta*(1 - x/L)))/eta;
end
```

Code Boundary Conditions

Now, write a function that evaluates the boundary conditions $u(0,t) = u(1,t) = 0$. For problems posed on the interval $a \le x \le b$, the boundary conditions apply for all t and either $x = a$ or $x = b$. The standard form for the boundary conditions expected by the solver is

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0 .$$

Written in this form, the boundary conditions for this problem are

- For $x = 0$, the equation is $u + 0 \cdot d\frac{\partial u}{\partial x} = 0$. The coefficients are:

- $p_L(x,t,u) = u,$
- $q_L(x,t) = 0 .$

- Likewise for $x = 1$, the equation is $u + 0 \cdot d\frac{\partial u}{\partial x} = 0$. The coefficients are:

- $p_R(x,t,u) = u,$
- $q_R(x,t) = 0 .$

The boundary function should use the function signature `[pl,ql,pr,qr] = transistorBC(xl,ul,xr,ur,t)`:

- The inputs `xl` and `ul` correspond to x and u for the left boundary.
- The inputs `xr` and `ur` correspond to x and u for the right boundary.
- `t` is the independent time variable.
- The outputs `pl` and `ql` correspond to $p_L(x,t,u)$ and $q_L(x,t)$ for the left boundary ($x = 0$ for this problem).
- The outputs `pr` and `qr` correspond to $p_R(x,t,u)$ and $q_R(x,t)$ for the right boundary ($x = 1$ for this problem).

The boundary conditions in this example are represented by the function:

```
function [pl,ql,pr,qr] = transistorBC(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = ur;
qr = 0;
end
```

Select Solution Mesh

The solution mesh defines the values of x and t where the solution is evaluated by the solver. Since the solution to this problem changes rapidly, use a relatively fine mesh of 50 spatial points in the interval $0 \leq x \leq L$ and 50 time points in the interval $0 \leq t \leq 1$.

```
x = linspace(0,C.L,50);
t = linspace(0,1,50);
```

Solve Equation

Finally, solve the equation using the symmetry m, the PDE equation, the initial condition, the boundary conditions, and the meshes for x and t. Since `pdepe` expects the PDE function to use four inputs and the initial condition function to use one input, create function handles that pass in the structure of physical constants as an extra input.

```
m = 0;
eqn = @(x,t,u,dudx) transistorPDE(x,t,u,dudx,C);
ic = @(x) transistorIC(x,C);
sol = pdepe(m,eqn,ic,@transistorBC,x,t);
```

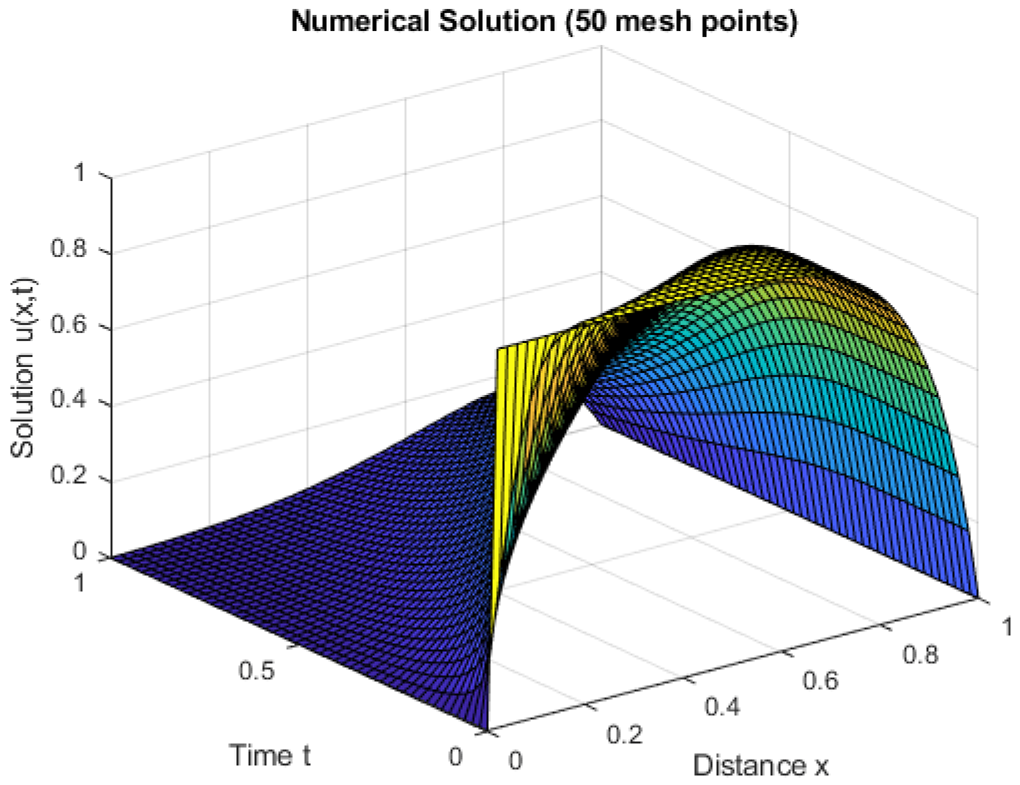
`pdepe` returns the solution in a 3-D array `sol`, where `sol(i,j,k)` approximates the `k`th component of the solution $u_k$evaluated at `t(i)` and `x(j)`. For this problem `u` has only one component, but in general you can extract the `k`th solution component with the command `u = sol(:,:,k)`.

```
u = sol(:,:,1);
```

Plot Solution

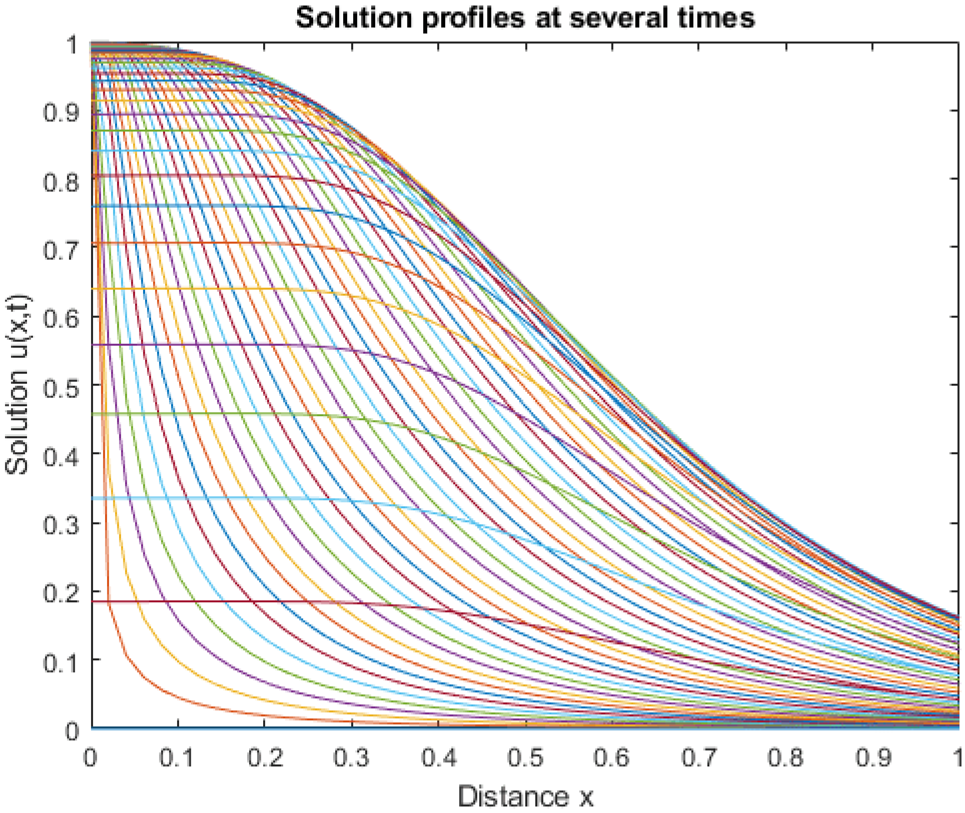
Create a surface plot of the solution u plotted at the selected mesh points for x and t.

```
surf(x,t,u)
title('Numerical Solution (50 mesh points)')
xlabel('Distance x')
ylabel('Time t')
zlabel('Solution u(x,t)')
```



Now, plot just *x* and *u* to get a side view of the contours in the surface plot.

```
plot(x,u)
xlabel('Distance x')
ylabel('Solution u(x,t)')
title('Solution profiles at several times')
```



Compute Emitter Discharge Current

Using a series solution for $u(x, t)$, the emitter discharge current can be expressed as the infinite series [1]:

$$I(t) = 2\pi^2 I_p\left(\frac{1-e^{-\eta}}{\eta}\right)\sum_{n=1}^{\infty}\frac{n^2}{n^2\pi^2+\eta^2/4}e^{-\frac{dt}{L^2}\left(n^2\pi^2+\eta^2/4\right)}.$$

Write a function to calculate the analytic solution for $I(t)$ using 40 terms in the series. The only variable is time, but specify a second input to the function for the structure of constants.

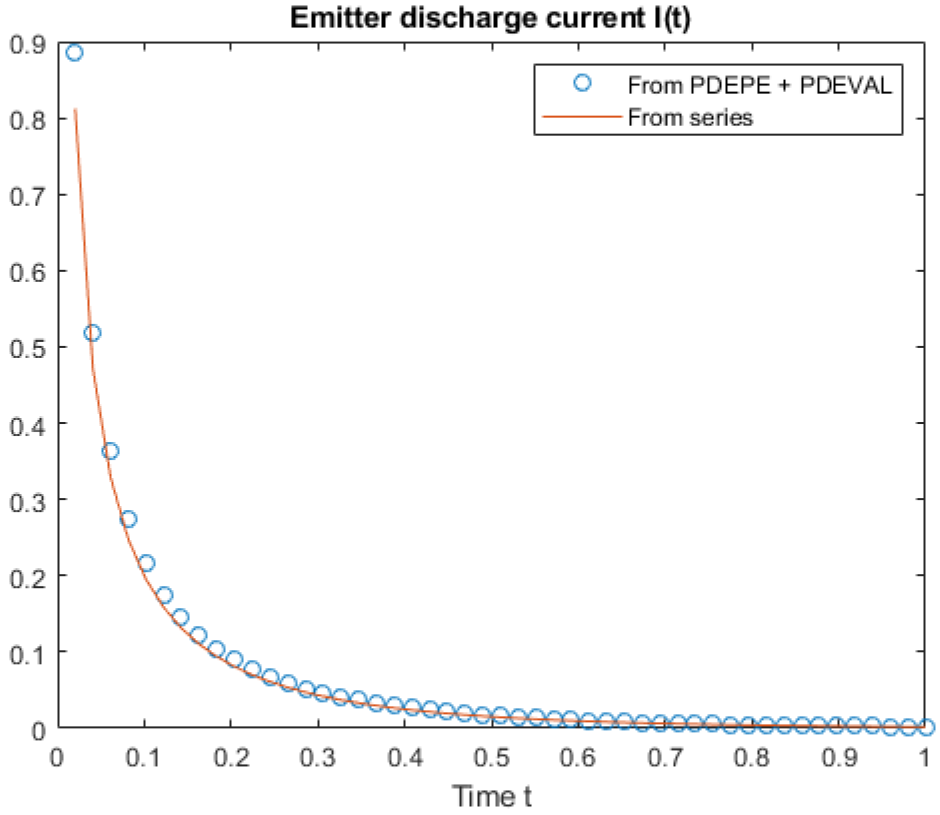
```
function It = serex3(t,C) % Approximate I(t) by series expansion.
Ip = C.Ip;
eta = C.eta;
D = C.D;
L = C.L;

It = 0;
for n = 1:40 % Use 40 terms
  m = (n*pi)^2 + 0.25*eta^2;
  It = It + ((n*pi)^2 / m)* exp(-(D/L^2)*m*t);
end
It = 2*Ip*((1 - exp(-eta))/eta)*It;
end
```

Using the numeric solution for $u(x, t)$ as computed by `pdepe`, you can also calculate the numeric approximation for $I(t)$ at $x = 0$ with

$$I(t) = \left[\frac{I_p D}{K} \frac{\partial}{\partial x} u(x,t)\right]_{x\,=\,0}.$$

Calculate the analytic and numeric solutions for $I(t)$ and plot the results. Use pdeval to compute the value of $\partial u / \partial x$ at $x = 0$.

```
nt = length(t);
I = zeros(1,nt);
seriesI = zeros(1,nt);
iok = 2:nt;
for j = iok
   % At time t(j), compute du/dx at x = 0.
   [~,I(j)] = pdeval(m,x,u(j,:),0);
   seriesI(j) = serex3(t(j),C);
end
% Numeric solution has form I(t) = (I_p*D/K)*du(0,t)/dx
I = (C.Ip*C.D/C.K)*I;

plot(t(iok),I(iok),'o',t(iok),seriesI(iok))
legend('From PDEPE + PDEVAL','From series')
title('Emitter discharge current I(t)')
xlabel('Time t')
```



The results match reasonably well. You can further improve the numeric result from pdepe by using a finer solution mesh.

Local Functions

Listed here are the local helper functions that the PDE solver pdepe calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function [c,f,s] = transistorPDE(x,t,u,dudx,C) % Equation to solve
D = C.D;
eta = C.eta;
L = C.L;

c = 1;
f = D*dudx;
s = -(D*eta/L)*dudx;
end
% ----------------------------------------------------
function u0 = transistorIC(x,C) % Initial condition
K = C.K;
L = C.L;
D = C.D;
eta = C.eta;

u0 = (K*L/D)*(1 - exp(-eta*(1 - x/L)))/eta;
end
% ----------------------------------------------------
function [pl,ql,pr,qr] = transistorBC(xl,ul,xr,ur,t) % Boundary conditions
pl = ul;
ql = 0;
pr = ur;
qr = 0;
end
% ----------------------------------------------------
function It = serex3(t,C) % Approximate I(t) by series expansion.
Ip = C.Ip;
eta = C.eta;
D = C.D;
L = C.L;

It = 0;
for n = 1:40 % Use 40 terms
  m = (n*pi)^2 + 0.25*eta^2;
  It = It + ((n*pi)^2 / m)* exp(-(D/L^2)*m*t);
end
It = 2*Ip*((1 - exp(-eta))/eta)*It;
end
% ----------------------------------------------------
```

References

[1] Zachmanoglou, E.C. and D.L. Thoe. *Introduction to Partial Differential Equations with Applications*. Dover, New York, 1986.

See Also

pdepe

More About

- “Solving Partial Differential Equations” on page 13-2
- “Solve System of PDEs” on page 13-32

Solve System of PDEs

This example shows how to formulate, compute, and plot the solution to a system of two partial differential equations.

Consider the system of PDEs

$$\frac{\partial u_1}{\partial t} = 0.024\frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2),$$

$$\frac{\partial u_2}{\partial t} = 0.170\frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2).$$

(The function $F(y) = e^{5.73y} - e^{-11.46y}$ is used as a shorthand.)

The equation holds on the interval $0 \le x \le 1$ for times $t \ge 0$. The initial conditions are

$$u_1(x, 0) = 1,$$

$$u_2(x, 0) = 0.$$

The boundary conditions are

$$\frac{\partial}{\partial x}u_1(0, t) = 0,$$
$$u_2(0, t) = 0,$$
$$\frac{\partial}{\partial x}u_2(1, t) = 0,$$
$$u_1(1, t) = 1.$$

To solve this equation in MATLAB, you need to code the equation, the initial conditions, and the boundary conditions, then select a suitable solution mesh before calling the solver `pdepe`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Before you can code the equation, you need to make sure that it is in the form that the `pdepe` solver expects:

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x, t, u, \frac{\partial u}{\partial x}\right)\right) + s\left(x, t, u, \frac{\partial u}{\partial x}\right).$$

In this form, the PDE coefficients are matrix-valued and the equation becomes

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\frac{\partial}{\partial t}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix} 0.024\frac{\partial u_1}{\partial x} \\ 0.170\frac{\partial u_2}{\partial x} \end{bmatrix} + \begin{bmatrix} -F(u_1 - u_2) \\ F(u_1 - u_2) \end{bmatrix}.$$

So the values of the coefficients in the equation are

$$m = 0$$

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{(diagonal values only)}$$

$$f\left(x, t, u, \frac{\partial u}{\partial x}\right) = \begin{bmatrix} 0.024\dfrac{\partial u_1}{\partial x} \\ 0.170\dfrac{\partial u_2}{\partial x} \end{bmatrix}$$

$$s\left(x, t, u, \frac{\partial u}{\partial x}\right) = \begin{bmatrix} -F(u_1 - u_2) \\ F(u_1 - u_2) \end{bmatrix}$$

Now you can create a function to code the equation. The function should have the signature `[c,f,s] = pdefun(x,t,u,dudx)`:

- `x` is the independent spatial variable.
- `t` is the independent time variable.
- `u` is the dependent variable being differentiated with respect to `x` and `t`. It is a two-element vector where `u(1)` is $u_1(x, t)$ and `u(2)` is $u_2(x, t)$.
- `dudx` is the partial spatial derivative $\partial u / \partial x$. It is a two-element vector where `dudx(1)` is $\partial u_1 / \partial x$ and `dudx(2)` is $\partial u_2 / \partial x$.
- The outputs `c`, `f`, and `s` correspond to coefficients in the standard PDE equation form expected by `pdepe`.

As a result, the equations in this example can be represented by the function:

```
function [c,f,s] = pdefun(x,t,u,dudx)
c = [1; 1];
f = [0.024; 0.17] .* dudx;
y = u(1) - u(2);
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
end
```

(Note: All functions are included as local functions at the end of the example.)

Code Initial Conditions

Next, write a function that returns the initial condition. The initial condition is applied at the first time value and provides the value of $u(x, t_0)$ for any value of x. The number of initial conditions must equal the number of equations, so for this problem there are two initial conditions. Use the function signature `u0 = pdeic(x)` to write the function.

The initial conditions are

$$u_1(x, 0) = 1,$$

$$u_2(x, 0) = 0 .$$

The corresponding function is

```
function u0 = pdeic(x)
u0 = [1; 0];
end
```

Code Boundary Conditions

Now, write a function that evaluates the boundary conditions

$$\begin{aligned}\frac{\partial}{\partial x}u_1(0,t) &= 0,\\ u_2(0,t) &= 0,\\ \frac{\partial}{\partial x}u_2(1,t) &= 0,\\ u_1(1,t) &= 1\,.\end{aligned}$$

For problems posed on the interval $a \le x \le b$, the boundary conditions apply for all t and either $x = a$ or $x = b$. The standard form for the boundary conditions expected by the solver is

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0\,.$$

Written in this form, the boundary conditions for the partial derivatives of u need to be expressed in terms of the flux $f\left(x,t,u,\frac{\partial u}{\partial x}\right)$. So the boundary conditions for this problem are

For $x = 0$, the equation is

$$\begin{bmatrix}0\\ u_2\end{bmatrix} + \begin{bmatrix}1\\ 0\end{bmatrix}\cdot\begin{bmatrix}0.024\frac{\partial u_1}{\partial x}\\ 0.170\frac{\partial u_2}{\partial x}\end{bmatrix} = 0\,.$$

The coefficients are:

$$\begin{aligned}p_L(x,t,u) &= \begin{bmatrix}0\\ u_2\end{bmatrix},\\ q_L(x,t) &= \begin{bmatrix}1\\ 0\end{bmatrix}.\end{aligned}$$

Likewise, for $x = 1$ the equation is

$$\begin{bmatrix}u_1 - 1\\ 0\end{bmatrix} + \begin{bmatrix}0\\ 1\end{bmatrix}\cdot\begin{bmatrix}0.024\frac{\partial u_1}{\partial x}\\ 0.170\frac{\partial u_2}{\partial x}\end{bmatrix} = 0\,.$$

The coefficients are:

$$\begin{aligned}p_R(x,t,u) &= \begin{bmatrix}u_1 - 1\\ 0\end{bmatrix},\\ q_R(x,t) &= \begin{bmatrix}0\\ 1\end{bmatrix}.\end{aligned}$$

The boundary function should use the function signature `[pl,ql,pr,qr] = pdebc(xl,ul,xr,ur,t)`:

- The inputs `xl` and `ul` correspond to u and x for the left boundary.
- The inputs `xr` and `ur` correspond to u and x for the right boundary.
- `t` is the independent time variable.
- The outputs `pl` and `ql` correspond to $p_L(x, t, u)$ and $q_L(x, t)$ for the left boundary ($x = 0$ for this problem).
- The outputs `pr` and `qr` correspond to $p_R(x, t, u)$ and $q_R(x, t)$ for the right boundary ($x = 1$ for this problem).

The boundary conditions in this example are represented by the function:

```
function [pl,ql,pr,qr] = pdebc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];
end
```

Select Solution Mesh

The solution to this problem changes rapidly when t is small. Although `pdepe` selects a time step that is appropriate to resolve the sharp changes, to see the behavior in the output plots you need to select appropriate output times. For the spatial mesh, there are boundary layers in the solution at both ends of $0 \leq x \leq 1$, so you need to specify mesh points there to resolve the sharp changes.

```
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];
```

Solve Equation

Finally, solve the equation using the symmetry m, the PDE equation, the initial conditions, the boundary conditions, and the meshes for x and t.

```
m = 0;
sol = pdepe(m,@pdefun,@pdeic,@pdebc,x,t);
```

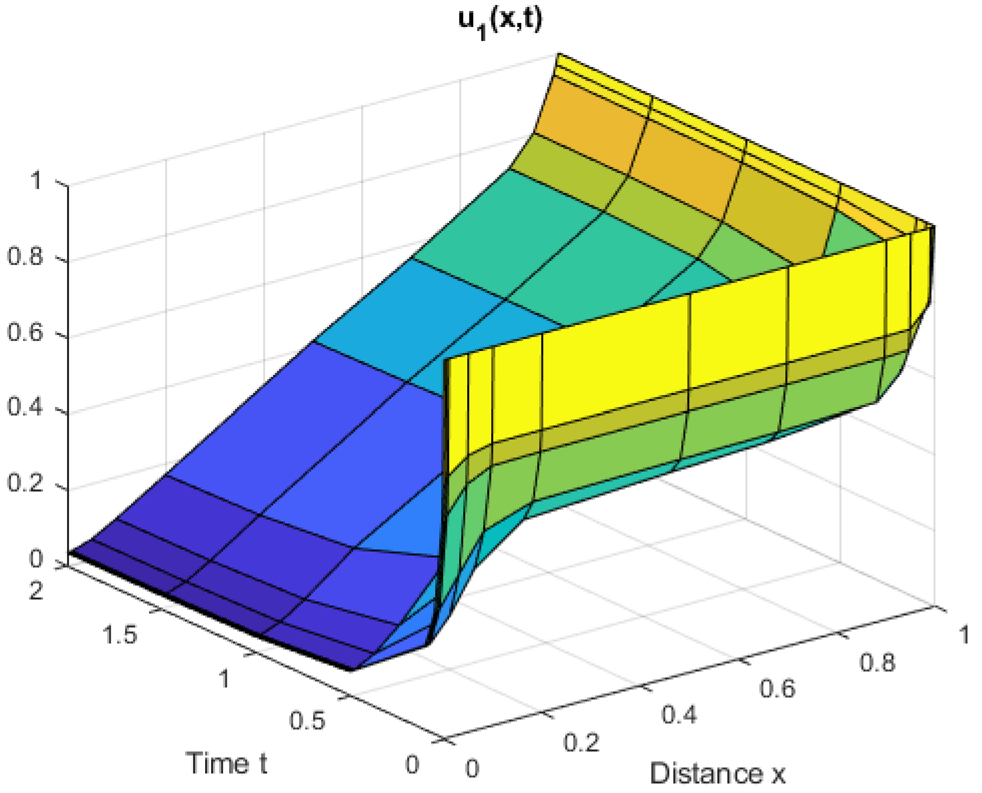
`pdepe` returns the solution in a 3-D array `sol`, where `sol(i,j,k)` approximates the `k`th component of the solution u_k evaluated at `t(i)` and `x(j)`. Extract each solution component into a separate variable.

```
u1 = sol(:,:,1);
u2 = sol(:,:,2);
```

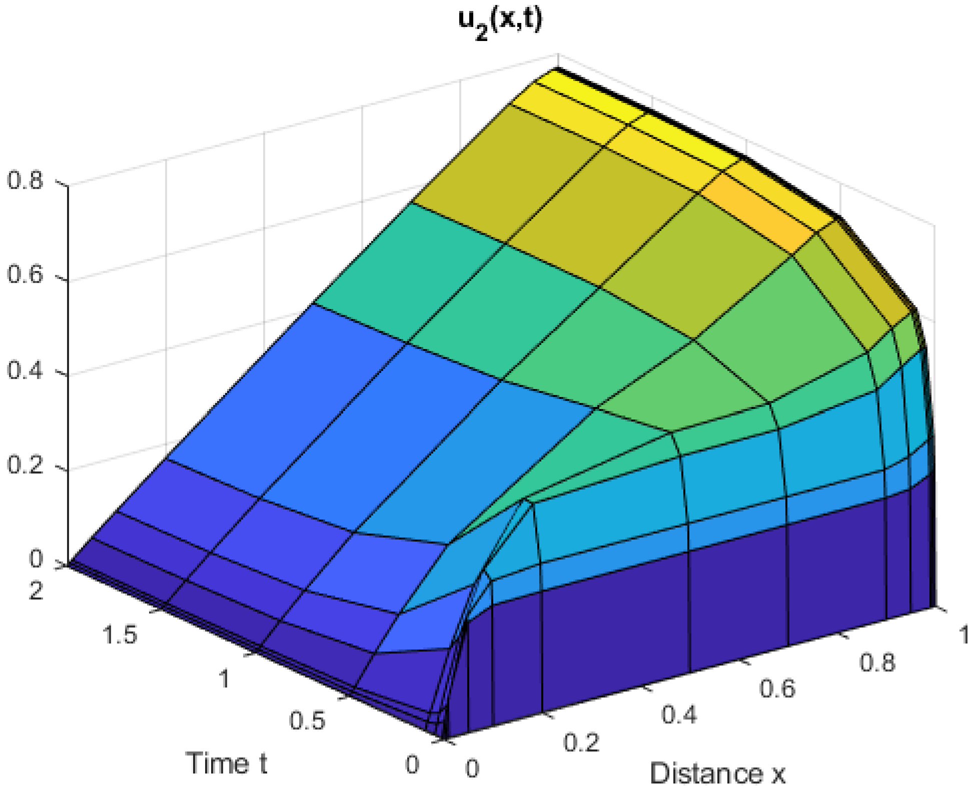
Plot Solution

Create surface plots of the solutions for u_1 and u_2 plotted at the selected mesh points for x and t.

```
surf(x,t,u1)
title('u_1(x,t)')
xlabel('Distance x')
ylabel('Time t')
```



```
surf(x,t,u2)
title('u_2(x,t)')
xlabel('Distance x')
ylabel('Time t')
```



Local Functions

Listed here are the local helper functions that the PDE solver pdepe calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function [c,f,s] = pdefun(x,t,u,dudx) % Equation to solve
c = [1; 1];
f = [0.024; 0.17] .* dudx;
y = u(1) - u(2);
F = exp(5.73*y)-exp(-11.47*y);
s = [-F; F];
end
% ---------------------------------------------
function u0 = pdeic(x) % Initial Conditions
u0 = [1; 0];
end
% ---------------------------------------------
function [pl,ql,pr,qr] = pdebc(xl,ul,xr,ur,t) % Boundary Conditions
pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];
end
% ---------------------------------------------
```

See Also

pdepe

More About

- “Solving Partial Differential Equations” on page 13-2
- “Solve Single PDE” on page 13-9
- “Solve System of PDEs with Initial Condition Step Functions” on page 13-39

Solve System of PDEs with Initial Condition Step Functions

This example shows how to solve a system of partial differential equations that uses step functions in the initial conditions.

Consider the PDEs

$$\frac{\partial n}{\partial t} = \frac{\partial}{\partial x}\left[d\frac{\partial n}{\partial x} - a\,n\frac{\partial c}{\partial x}\right] + S\,r\,n(N - n),$$
$$\frac{\partial c}{\partial t} = \frac{\partial^2 c}{\partial x^2} + S\left(\frac{n}{n+1} - c\right).$$

The equations involve the constant parameters d, a, S, r, and N, and are defined for $0 \le x \le 1$ and $t \ge 0$. These equations arise in a mathematical model of the first steps of tumor-related angiogenesis [1]. $n(x,t)$ represents the cell density of endothelial cells, and $c(x,t)$ the concentration of a protein they release in response to the tumor.

This problem has a constant, steady state when

$$\begin{bmatrix} n_0 \\ c_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}.$$

However, a stability analysis predicts evolution of the system to an inhomogeneous solution [1]. So step functions are used as the initial conditions to perturb the steady state and stimulate evolution of the system.

The boundary conditions require that both solution components have zero flux at $x = 0$ and $x = 1$.

$$\frac{\partial}{\partial x}n(0,t) = \frac{\partial}{\partial x}n(1,t) = 0,$$
$$\frac{\partial}{\partial x}c(0,t) = \frac{\partial}{\partial x}c(1,t) = 0.$$

To solve this system of equations in MATLAB, you need to code the equations, initial conditions, and boundary conditions, then select a suitable solution mesh before calling the solver `pdepe`. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Equation

Before you can code the equation, you need to make sure that it is in the form that the `pdepe` solver expects:

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

Since there are two equations in the system of PDEs, the system of PDEs can be rewritten as

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\frac{\partial}{\partial t}\begin{bmatrix} n \\ c \end{bmatrix} = \frac{\partial}{\partial x}\begin{bmatrix} d\frac{\partial n}{\partial x} - a\,n\frac{\partial c}{\partial x} \\ \frac{\partial c}{\partial x} \end{bmatrix} + \begin{bmatrix} S\,r\,n(N-n) \\ S\left(\frac{n}{n+1} - c\right) \end{bmatrix}.$$

The values of the coefficients in the equation are then

$$m = 0$$

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ (diagonal values only)}$$

$$f\left(x, t, u, \frac{\partial u}{\partial x}\right) = \begin{bmatrix} d\frac{\partial n}{\partial x} - a\, n\frac{\partial c}{\partial x} \\ \frac{\partial c}{\partial x} \end{bmatrix}$$

$$s\left(x, t, u, \frac{\partial u}{\partial x}\right) = \begin{bmatrix} S\, r\, n(N - n) \\ S\left(\frac{n}{n + 1} - c\right) \end{bmatrix}$$

Now you can create a function to code the equation. The function should have the signature `[c,f,s] = angiopde(x,t,u,dudx)`:

- `x` is the independent spatial variable.
- `t` is the independent time variable.
- `u` is the dependent variable being differentiated with respect to `x` and `t`. It is a two-element vector where `u(1)` is $n(x,t)$ and `u(2)` is $c(x,t)$.
- `dudx` is the partial spatial derivative $\partial u/\partial x$. It is a two-element vector where `dudx(1)` is $\partial n/\partial x$ and `dudx(2)` is $\partial c/\partial x$.
- The outputs `c`, `f`, and `s` correspond to coefficients in the standard PDE equation form expected by `pdepe`.

As a result, the equations in this example can be represented by the function:

```
function [c,f,s] = angiopde(x,t,u,dudx)
d = 1e-3;
a = 3.8;
S = 3;
r = 0.88;
N = 1;

c = [1; 1];
f = [d*dudx(1) - a*u(1)*dudx(2)
     dudx(2)];
s = [S*r*u(1)*(N - u(1));
     S*(u(1)/(u(1) + 1) - u(2))];
end
```

(Note: All functions are included as local functions at the end of the example.)

Code Initial Conditions

Next, write a function that returns the initial condition. The initial condition is applied at the first time value and provides the value of $n(x,t_0)$ and $c(x,t_0)$ for any value of x. Use the function signature `u0 = angioic(x)` to write the function.

This problem has a constant, steady state when

$$\begin{bmatrix} n_0 \\ c_0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0.5 \end{bmatrix}.$$

However, a stability analysis predicts evolution of the system to an inhomogenous solution [1]. So, step functions are used as the initial conditions to perturb the steady state and stimulate evolution of the system.

$$u(x,0) = \begin{bmatrix} n_0 \\ c_0 \end{bmatrix},$$

$$u(x,0) = \begin{cases} 1.05u_1 & 0.3 \le x \le 0.6 \\ 1.0005u_2 & 0.3 \le x \le 0.6 \end{cases}$$

The corresponding function is

```
function u0 = angioic(x)
u0 = [1; 0.5];
if x >= 0.3 && x <= 0.6
  u0(1) = 1.05 * u0(1);
  u0(2) = 1.0005 * u0(2);
end
end
```

Code Boundary Conditions

Now, write a function that evaluates the boundary conditions

$$\frac{\partial}{\partial x} n(0,t) = \frac{\partial}{\partial x} n(1,t) = 0,$$

$$\frac{\partial}{\partial x} c(0,t) = \frac{\partial}{\partial x} c(1,t) = 0.$$

For problems posed on the interval $a \le x \le b$, the boundary conditions apply for all t and either $x = a$ or $x = b$. The standard form for the boundary conditions expected by the solver is

$$p(x,t,u) + q(x,t) f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0.$$

For $x = 0$, the boundary condition equation is

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} d\frac{\partial n}{\partial x} - a\, n\frac{\partial c}{\partial x} \\ \frac{\partial c}{\partial x} \end{bmatrix} = 0.$$

So the coefficients are:

- $p_L(x,t,u) = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$
- $q_L(x,t) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$

For $x = 1$ the boundary conditions are the same, so $p_R(x,t,u) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ and $q_R(x,t) = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

The boundary function should use the function signature `[pl,ql,pr,qr] = angiobc(xl,ul,xr,ur,t)`, where:

- The inputs `xl` and `ul` correspond to u and x for the left boundary.
- The inputs `xr` and `ur` correspond to u and x for the right boundary.
- `t` is the independent time variable.
- The outputs `pl` and `ql` correspond to $p_L(x,t,u)$ and $q_L(x,t)$ for the left boundary ($x = 0$ for this problem).
- The outputs `pr` and `qr` correspond to $p_R(x,t,u)$ and $q_R(x,t)$ for the right boundary ($x = 1$ for this problem).

The boundary conditions in this example are represented by the function:

```
function [pl,ql,pr,qr] = angiobc(xl,ul,xr,ur,t)
pl = [0; 0];
ql = [1; 1];
pr = pl;
qr = ql;
end
```

Select Solution Mesh

A long time interval is required to see the limiting behavior of the equations, so use 10 points in the interval $0 \le t \le 200$. Also, the limit distribution of $c(x,t)$ varies by only about 0.1% over the interval $0 \le x \le 1$, so a relatively fine spatial mesh with 50 points is appropriate.

```
x = linspace(0,1,50);
t = linspace(0,200,10);
```

Solve Equation

Finally, solve the equation using the symmetry m, the PDE equation, the initial condition, the boundary conditions, and the meshes for x and t.

```
m = 0;
sol = pdepe(m,@angiopde,@angioic,@angiobc,x,t);
```

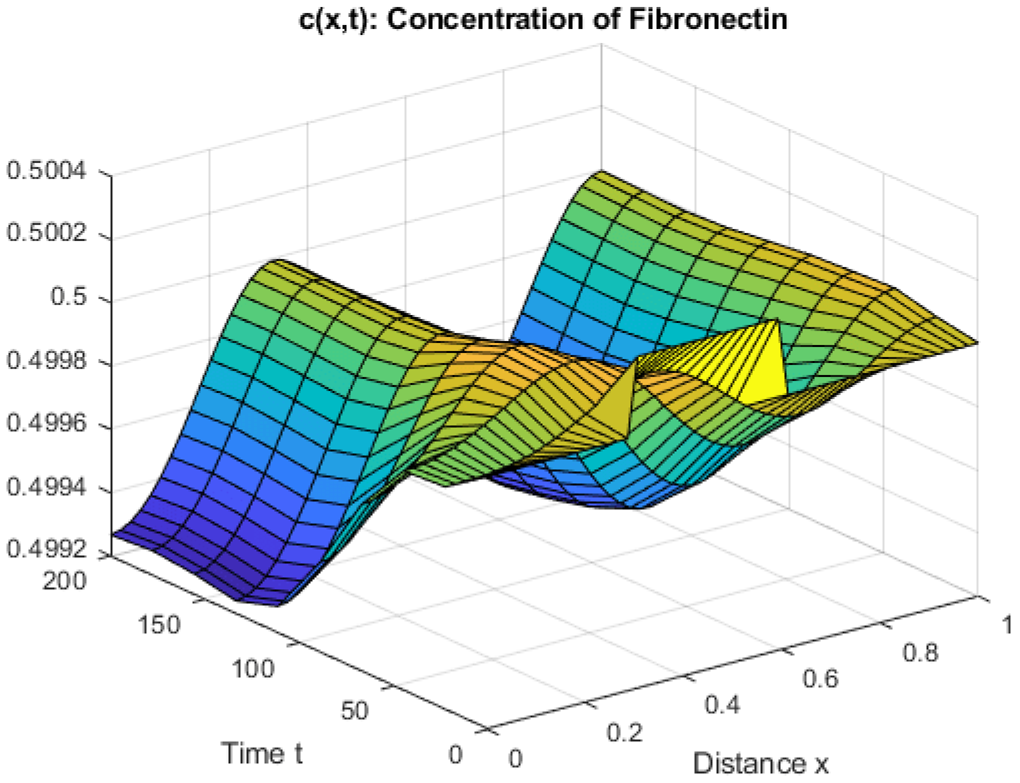
`pdepe` returns the solution in a 3-D array `sol`, where `sol(i,j,k)` approximates the `k`th component of the solution u_k evaluated at `t(i)` and `x(j)`. Extract the solution components into separate variables.

```
n = sol(:,:,1);
c = sol(:,:,2);
```

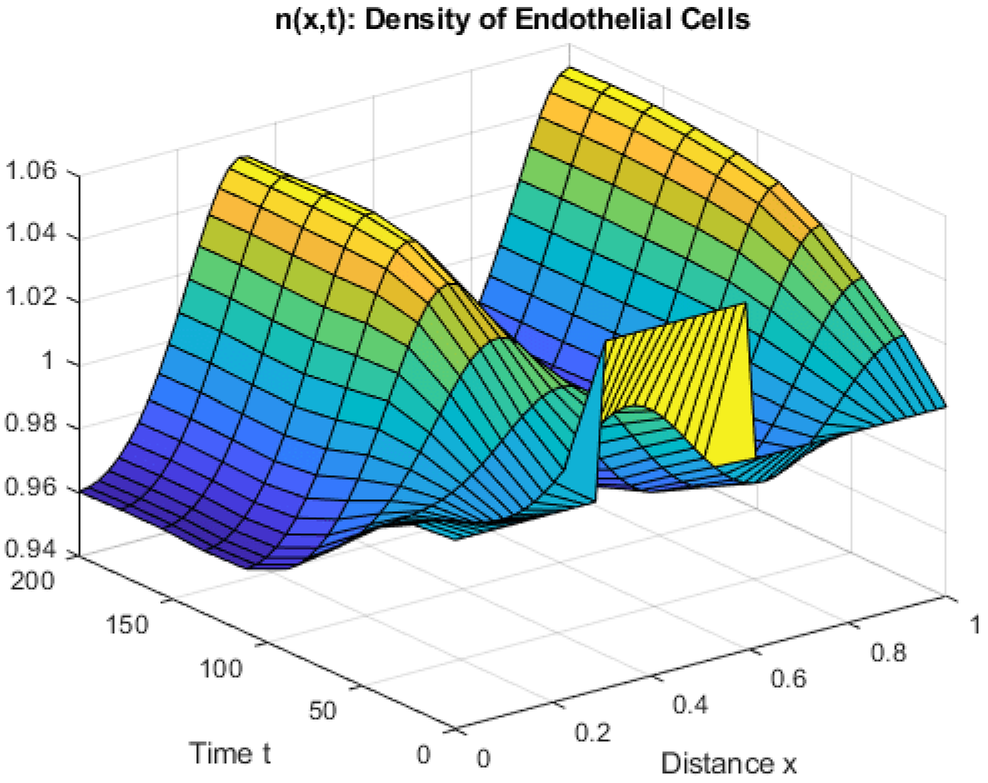
Plot Solution

Create a surface plot of the solution components n and c plotted at the selected mesh points for x and t.

```
surf(x,t,c)
title('c(x,t): Concentration of Fibronectin')
xlabel('Distance x')
ylabel('Time t')
```

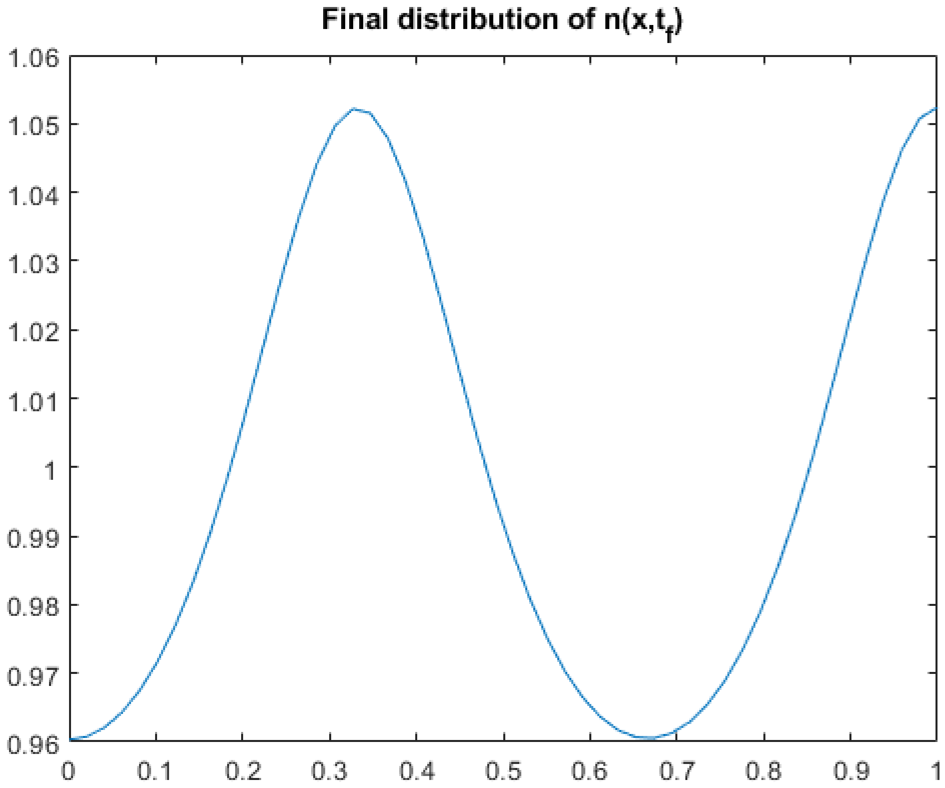


```
surf(x,t,n)
title('n(x,t): Density of Endothelial Cells')
xlabel('Distance x')
ylabel('Time t')
```

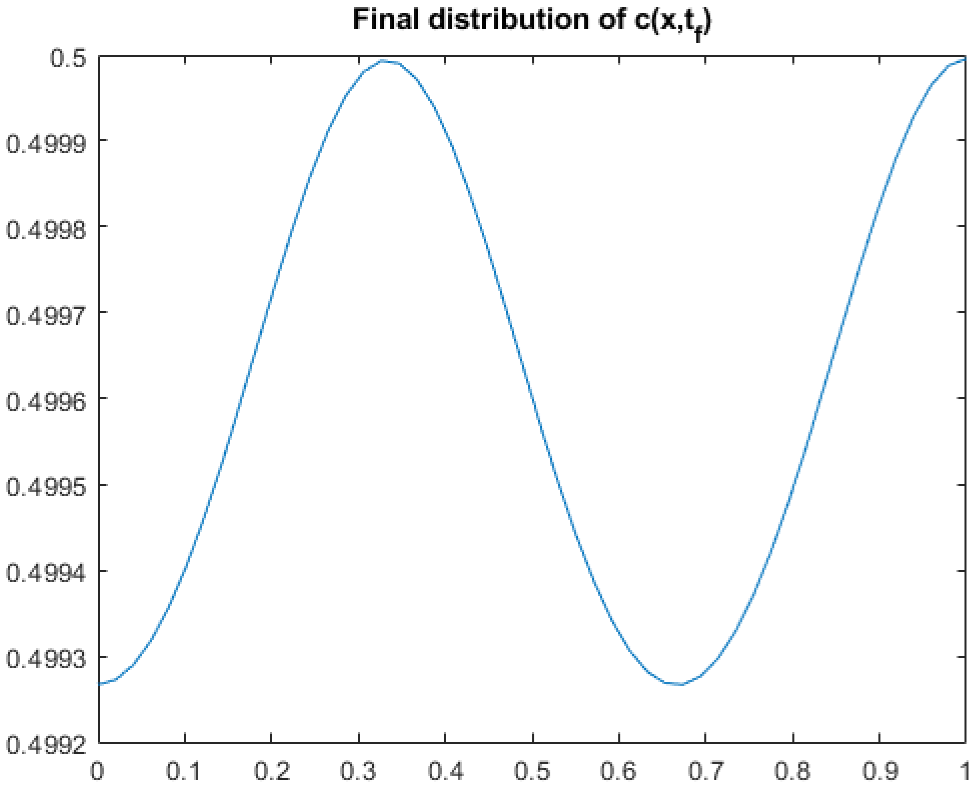


Now plot just the final distributions of the solutions at $t_f = 200$. These plots correspond to Figures 3 and 4 in [1].

```
plot(x,n(end,:))
title('Final distribution of n(x,t_f)')
```



```
plot(x,c(end,:))
title('Final distribution of c(x,t_f)')
```



References

[1] Humphreys, M.E. and M.A.J. Chaplain. "A mathematical model of the first steps of tumour-related angiogenesis: Capillary sprout formation and secondary branching." *IMA Journal of Mathematics Applied in Medicine & Biology*, 13 (1996), pp. 73-98.

Local Functions

Listed here are the local helper functions that the PDE solver pdepe calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function [c,f,s] = angiopde(x,t,u,dudx) % Equation to solve
d = 1e-3;
a = 3.8;
S = 3;
r = 0.88;
N = 1;

c = [1; 1];
f = [d*dudx(1) - a*u(1)*dudx(2)
     dudx(2)];
s = [S*r*u(1)*(N - u(1));
     S*(u(1)/(u(1) + 1) - u(2))];
end
% ---------------------------------------------
function u0 = angioic(x) % Initial Conditions
u0 = [1; 0.5];
```

```
if x >= 0.3 && x <= 0.6
  u0(1) = 1.05 * u0(1);
  u0(2) = 1.0005 * u0(2);
end
end
% --------------------------------------------
function [pl,ql,pr,qr] = angiobc(xl,ul,xr,ur,t) % Boundary Conditions
pl = [0; 0];
ql = [1; 1];
pr = pl;
qr = ql;
end
% --------------------------------------------
```

See Also

pdepe

More About

- “Solving Partial Differential Equations” on page 13-2
- “Solve Single PDE” on page 13-9
- “Solve System of PDEs” on page 13-32

Delay Differential Equations (DDEs)

Solving Delay Differential Equations

Delay differential equations (DDEs) are ordinary differential equations that relate the solution at the current time to the solution at past times. This delay can be constant, time-dependent, state-dependent, or derivative-dependent. In order for the integration to begin, you generally must provide a solution history so that the solution is accessible to the solver for times before the initial integration point.

Constant Delay DDEs

A system of differential equations with constant delays has the form:

$$y'(t) = f(t, y(t), y(t - \tau_1), \ldots, y(t - \tau_k)) .$$

Here, t is the independent variable, y is a column vector of dependent variables, and y' represents the first derivative of y with respect to t. The delays, $\tau_1,\ldots,\tau_k$, are positive constants.

The `dde23` function solves DDEs with constant delays with history $y(t) = S(t)$ for $t < t_0$.

The solutions of DDEs are generally continuous, but they have discontinuities in their derivatives. The `dde23` function tracks discontinuities in low-order derivatives. It integrates the differential equations with the same explicit Runge-Kutta (2,3) pair and interpolant used by `ode23`. The Runge-Kutta formulas are implicit for step sizes bigger than the delays. When $y(t)$ is smooth enough to justify steps this big, the implicit formulas are evaluated by a predictor-corrector iteration.

Time-Dependent and State-Dependent DDEs

Constant time delays are a special case of the more general DDE form:

$$y'(t) = f\big(t, y(t), y(dy_1), \ldots, y\big(dy_p\big)\big) .$$

Time-dependent and state-dependent DDEs involve delays $dy_1,\ldots, dy_k$ that can depend on both time t and state y. The delays $dy_j(t, y)$ must satisfy $dy_j(t, y) \leq t$ on the interval $[t_0, t_f]$ with $t_0 < t_f$.

The `ddesd` function finds the solution, $y(t)$, for time-dependent and state-dependent DDEs with history $y(t) = S(t)$ for $t < t_0$. The `ddesd` function integrates with the classic four-stage, fourth-order explicit Runge-Kutta method, and it controls the size of the residual of a natural interpolant. It uses iteration to take steps that are longer than the delays.

DDEs of Neutral Type

Delay differential equations of neutral type involve delays in y' as well as y:

$$y'(t) = f\big(t, y(t), y(dy_1), \ldots, y\big(dy_p\big), y'(dyp_1), \ldots, y'\big(dyp_q\big)\big) .$$

The delays in the solution must satisfy $dy_i(t,y) \leq t$. The delays in the first derivative must satisfy $dyp_j(t,y) < t$ so that y' does not appear on both sides of the equation.

The `ddensd` function solves DDEs of neutral type by approximating them with DDEs of the form given for time-dependent and state-dependent delays:

$$y'(t) = f\big(t, y(t), y(dy_1), \ldots, y\big(dy_p\big)\big) .$$

For more information, see Shampine [1].

Evaluating the Solution at Specific Points

Use the `deval` function and the output from any of the DDE solvers to evaluate the solution at specific points in the interval of integration. For example, `y = deval(sol, 0.5*(sol.x(1) + sol.x(end)))` evaluates the solution at the midpoint of the interval of integration.

History and Initial Values

When you solve a DDE, you approximate the solution on an interval $[t_0,t_f]$ with $t_0 < t_f$. The DDEs show how $y\,'(t)$ depends on values of the solution (and possibly its derivative) at times prior to t. For example, with constant delays $y\,'(t_0)$ depends on $y(t_0 - \tau_1),\ldots,y(t_0 - \tau_k)$ for positive constants τ_j. Because of this, a solution on $[t_0, t_k]$ depends on values it has at $t \le t_0$. You must define these values with a history function, $y(t) = S(t)$ for $t < t_0$.

Discontinuities in DDEs

If your problem has discontinuities, it is best to communicate them to the solver using an options structure. To do this, use the `ddeset` function to create an `options` structure containing the discontinuities in your problem.

There are three properties in the `options` structure that you can use to specify discontinuities; `InitialY`, `Jumps`, and `Events`. The property you choose depends on the location and nature of the discontinuities.

Nature of Discontinuity	Property	Comments
At the initial value $t = t_0$	`InitialY`	Generally the initial value $y(t_0)$ is the value $S(t_0)$ returned by the history function, meaning the solution is continuous at the initial point. If this is not the case, supply a different initial value using the `InitialY` property.
In the history, i.e., the solution at $t < t_0$, or in the equation coefficients for $t > t_0$	`Jumps`	Provide the known locations t of the discontinuities in a vector as the value of the `Jumps` property. Applies only to `dde23`.

Nature of Discontinuity	Property	Comments
State-dependent	`Events`	`dde23`, `ddesd`, and `ddensd` use the events function you supply to locate these discontinuities. When the solver finds such a discontinuity, restart the integration to continue. Specify the solution structure for the current integration as the history for the new integration. The solver extends each element of the solution structure after each restart so that the final structure provides the solution for the whole interval of integration. If the new problem involves a change in the solution, use the `InitialY` property to specify the initial value for the new integration.

Propagation of Discontinuities

Generally, the first derivative of the solution has a jump at the initial point. This is because the first derivative of the history function, $S(t)$, generally does not satisfy the DDE at this point. A discontinuity in any derivative of $y(t)$ propagates into the future at spacings of $\tau_1,\ldots, \tau_k$ when the delays are constant. If the delays are not constant, the propagation of discontinuities is more complicated. For neutral DDEs of the forms in “Constant Delay DDEs” on page 14-2 and “Time-Dependent and State-Dependent DDEs” on page 14-2, the discontinuity appears in the next higher order derivative each time it is propagated. In this sense, the solution gets smoother as the integration proceeds. Solutions of neutral DDEs of the form given in “DDEs of Neutral Type” on page 14-2 are qualitatively different. The discontinuity in the solution does not propagate to a derivative of higher order. In particular, the typical jump in $y'(t)$ at t_0 propagates as jumps in $y'(t)$ throughout $[t_0, t_f]$.

DDE Examples and Files

Several available example files serve as excellent starting points for most common DDE problems. To easily explore and run examples, simply use the **Differential Equations Examples** app. To run this app, type

```
odeexamples
```

To open an individual example file for editing, type

```
edit exampleFileName.m
```

To run an example, type

```
exampleFileName
```

This table contains a list of the available DDE example files, as well as the solvers and the options they use.

Example File	Solver Used	Options Specified	Description	Example Link
`ddex1`	`dde23`	—	DDE with constant history	“DDE with Constant Delays” on page 14-6
`ddex2`	`dde23`	• `'Jumps'`	DDE with a discontinuity	“Cardiovascular Model DDE with Discontinuities” on page 14-14
`ddex3`	`ddesd`	—	DDE with state-dependent delays	“DDE with State-Dependent Delays” on page 14-10
`ddex4`	`ddensd`	—	Neutral DDE with two delays	“DDE of Neutral Type” on page 14-19
`ddex5`	`ddensd`	—	Neutral DDE with initial value	“Initial Value DDE of Neutral Type” on page 14-23

References

[1] Shampine, L.F. “Dissipative Approximations to Neutral DDEs.” *Applied Mathematics & Computation*, Vol. 203, 2008, pp. 641–648.

See Also

`dde23` | `ddensd` | `ddesd` | `ddeset`

More About

- “DDE with Constant Delays” on page 14-6
- “DDE with State-Dependent Delays” on page 14-10

DDE with Constant Delays

This example shows how to use dde23 to solve a system of DDEs (delay differential equations) with constant delays.

The system of equations is

$$
\begin{aligned}
y_1'(t) &= y_1(t-1)\\
y_2'(t) &= y_1(t-1) + y_2(t-0.2)\\
y_3'(t) &= y_2(t)\,.
\end{aligned}
$$

The history function for $t \le 0$ is constant, $y_1(t) = y_2(t) = y_3(t) = 1$.

The time delays in the equations are only present in y terms, and the delays themselves are constants, so the equations form a system of *constant delay* equations.

To solve this system of equations in MATLAB, you need to code the equations, delays, and history before calling the delay differential equation solver dde23, which is meant for systems with constant delays. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Delays

First, create a vector to define the delays in the system of equations. This system has two different delays:

- A delay of 1 in the first component $y_1(t-1)$.
- A delay of 0.2 in the second component $y_2(t-0.2)$.

dde23 accepts a vector argument for the delays, where each element is the constant delay for one component.

```
lags = [1 0.2];
```

Code Equation

Now, create a function to code the equations. This function should have the signature dydt = ddefun(t,y,Z), where:

- t is time (independent variable).
- y is the solution (dependent variable).
- Z(:,j) approximates the delay $y(t-\tau_j)$, where the constant delay τ_j is given by lags(j).

These inputs are automatically passed to the function by the solver, but the variable names determine how you code the equations. In this case:

- Z(:,1) → $y_1(t-1)$
- Z(:,2) → $y_2(t-0.2)$

```
function dydt = ddefun(t,y,Z)
  ylag1 = Z(:,1);
  ylag2 = Z(:,2);
  dydt = [ylag1(1);
          ylag1(1)+ylag2(2);
          y(2)];
end
```

Note: All functions are included as local functions at the end of the example.

Code Solution History

Next, create a function to define the solution history. The solution history is the solution for times $t \le t_0$.

```
function s = history(t)
  s = ones(3,1);
end
```

Solve Equation

Finally, define the interval of integration $\left[t_0\ t_f\right]$ and solve the DDE using the `dde23` solver.

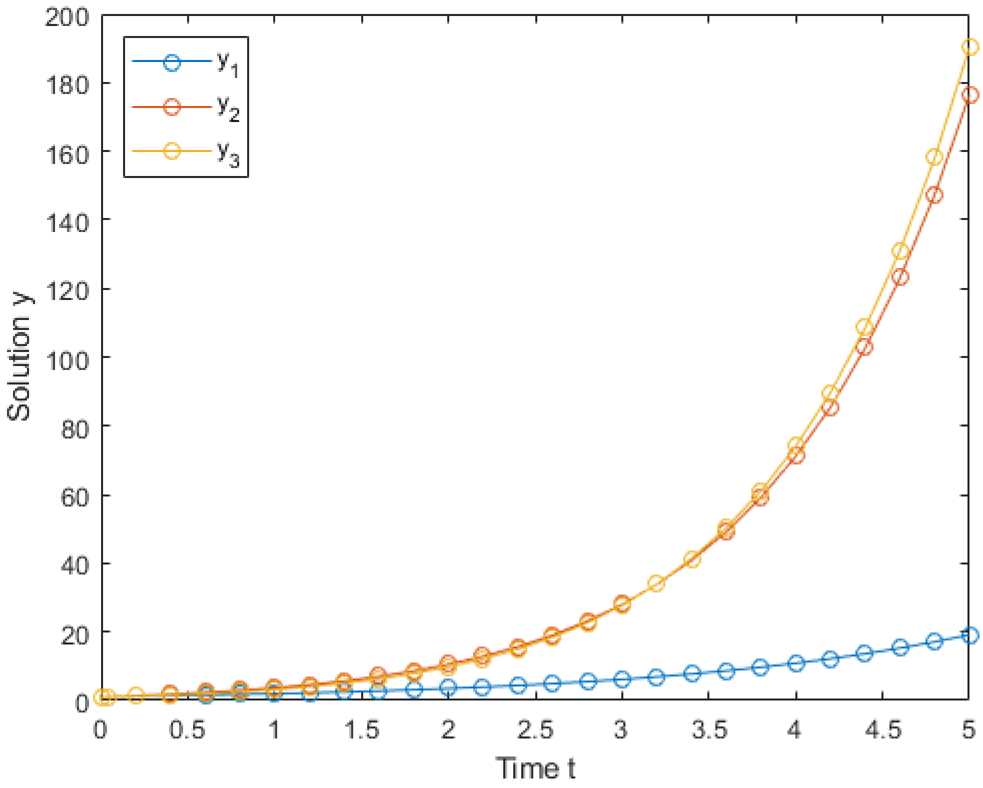
```
tspan = [0 5];
sol = dde23(@ddefun, lags, @history, tspan);
```

Plot Solution

The solution structure `sol` has the fields `sol.x` and `sol.y` that contain the internal time steps taken by the solver and corresponding solutions at those times. (If you need the solution at specific points, you can use `deval` to evaluate the solution at the specific points.)

Plot the three solution components against time.

```
plot(sol.x,sol.y,'-o')
xlabel('Time t');
ylabel('Solution y');
legend('y_1','y_2','y_3','Location','NorthWest');
```



Local Functions

Listed here are the local helper functions that the DDE solver dde23 calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydt = ddefun(t,y,Z) % equation being solved
  ylag1 = Z(:,1);
  ylag2 = Z(:,2);

  dydt = [ylag1(1);
          ylag1(1)+ylag2(2);
          y(2)];
end
%-------------------------------------------
function s = history(t) % history function for t <= 0
  s = ones(3,1);
end
%-------------------------------------------
```

See Also

dde23 | ddensd | ddesd | deval

More About

- “Solving Delay Differential Equations” on page 14-2

- “DDE with State-Dependent Delays” on page 14-10

DDE with State-Dependent Delays

This example shows how to use `ddesd` to solve a system of DDEs (delay differential equations) with state-dependent delays. This system of DDEs was used as a test problem by Enright and Hayashi [1].

The system of equations is

$$y_1'(t) = y_2(t),$$

$$y_2'(t) = -y_2\left(e^{1-y_2(t)}\right)\cdot y_2(t)^2\cdot e^{1-y_2(t)}.$$

The history functions for $t \le 0.1$ are the analytical solutions

$$y_1(t) = \log(t),$$

$$y_2(t) = \frac{1}{t}.$$

The time delays in the equations are only present in y terms. The delays depend only on the state of the second component $y_2(t)$, so the equations form a system of *state-dependent delay* equations.

To solve this system of equations in MATLAB, you need to code the equations, delays, and history before calling the delay differential equation solver `ddesd`, which is meant for systems with state-dependent delays. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Code Delays

First, write a function to define the time delays in the system. The only delay present in this system of equations is in the term$-y_2\left(e^{1-y_2(t)}\right)$.

```
function d = dely(t,y)
d = exp(1 - y(2));
end
```

Note: All functions are included as local functions at the end of the example.

Code Equation

Now, create a function to code the equations. This function should have the signature `dydt = ddefun(t,y,Z)`, where:

- `t` is time (independent variable).
- `y` is the solution (dependent variable).
- `Z(n,j)` approximates the delays $y_n(d(j))$, where the delay $d(j)$ is given by component `j` of `dely(t,y)`.

These inputs are automatically passed to the function by the solver, but the variable names determine how you code the equations. In this case:

- `Z(2,1)` $\rightarrow y_2\left(e^{1-y_2(t)}\right)$

```
function dydt = ddefun(t,y,Z)
dydt = [y(2);
       -Z(2,1)*y(2)^2*exp(1 - y(2))];
end
```

Code Solution History

Next, create a function to define the solution history. The solution history is the solution for times $t \leq t_0$.

```
function v = history(t) % history function for t < t0
v = [log(t);
     1./t];
end
```

Solve Equation

Finally, define the interval of integration $\left[t_0\ t_f\right]$ and solve the DDE using the `ddesd` solver.

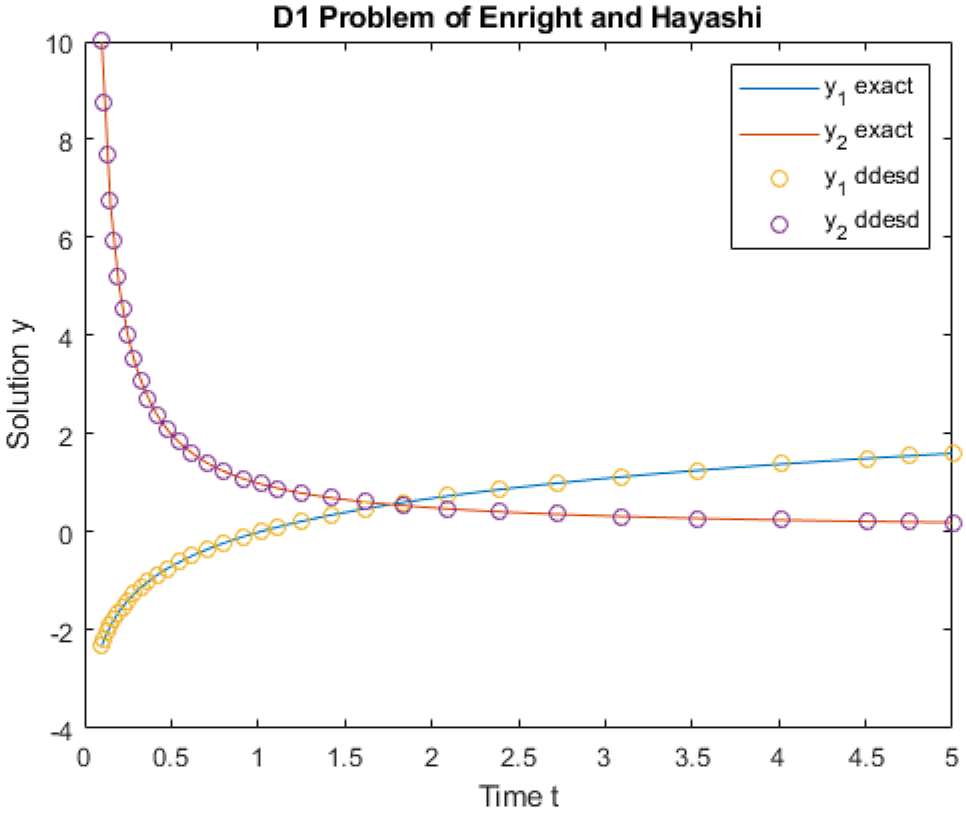
```
tspan = [0.1 5];
sol = ddesd(@ddefun, @dely, @history, tspan);
```

Plot Solution

The solution structure `sol` has the fields `sol.x` and `sol.y` that contain the internal time steps taken by the solver and corresponding solutions at those times. (If you need the solution at specific points, you can use `deval` to evaluate the solution at the specific points.)

Plot the two solution components against time using the history function to calculate the analytical solution within the integration interval for comparison.

```
ta = linspace(0.1,5);
ya = history(ta);

plot(ta,ya,sol.x,sol.y,'o')
legend('y_1 exact','y_2 exact','y_1 ddesd','y_2 ddesd')
xlabel('Time t')
ylabel('Solution y')
title('D1 Problem of Enright and Hayashi')
```



Local Functions

Listed here are the local helper functions that the DDE solver `ddesd` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydt = ddefun(t,y,Z) % equation being solved
dydt = [y(2);
       -Z(2,1).*y(2)^2.*exp(1 - y(2))];
end
%-------------------------------------------
function d = dely(t,y) % delay for y
d = exp(1 - y(2));
end
%-------------------------------------------
function v = history(t) % history function for t < t0
v = [log(t);
     1./t];
end
%-------------------------------------------
```

References

[1] Enright, W.H. and H. Hayashi. “The Evaluation of Numerical Software for Delay Differential Equations.” In *Proceedings of the IFIP TC2/WG2.5 working conference on Quality of numerical*

software: assessment and enhancement. (R.F. Boisvert, ed.). London, UK: Chapman & Hall, Ltd., pp. 179-193.

See Also

`dde23` | `ddensd` | `ddesd` | `deval`

More About

- “Solving Delay Differential Equations” on page 14-2
- “Cardiovascular Model DDE with Discontinuities” on page 14-14

Cardiovascular Model DDE with Discontinuities

This example shows how to use `dde23` to solve a cardiovascular model that has a discontinuous derivative. The example was originally presented by Ottesen [1].

The system of equations is

$$\dot{P}_a(t) = -\frac{1}{c_a R} P_a(t) + \frac{1}{c_a R} P_v(t) + \frac{1}{c_a} V_{\text{str}}\left(P_a^{\tau}(t)\right) H(t)$$

$$\dot{P}_v(t) = \frac{1}{c_v R} P_a(t) - \left(\frac{1}{c_v R} + \frac{1}{c_v r}\right) P_v(t)$$

$$\dot{H}(t) = \frac{\alpha_H T_s}{1 + \gamma_H T_p} - \beta_H T_p .$$

The terms for T_s and T_p are variations of the same equation with and without time delay. P_a^{τ} and P_a represent the mean arterial pressure with and without time delay, respectively.

$$T_s = \frac{1}{1 + \left(\frac{P_a^{\tau}}{\alpha_s}\right)^{\beta_s}}$$

$$T_p = \frac{1}{1 + \left(\frac{P_a}{\alpha_p}\right)^{-\beta_p}} .$$

This problem has a number of physical parameters:

- Arterial compliance $c_a = 1.55\,\text{ml/mmHg}$
- Venous compliance $c_v = 519\,\text{ml/mmHg}$
- Peripheral resistance $R = 1.05(0.84)\text{mmHg}\,s/\text{ml}$
- Venous outflow resistance $r = 0.068\,\text{mmHg}\,s/\text{ml}$
- Stroke volume $V_{\text{str}} = 67.9(77.9)\,\text{ml}$
- Typical mean arterial pressure $P_0 = 93\,\text{mmHg}$
- $\alpha_0 = \alpha_s = \alpha_p = 93(121)\,\text{mmHg}$
- $\alpha_H = 0.84\,\text{sec}^{-2}$
- $\beta_0 = \beta_s = \beta_p = 7$
- $\beta_H = 1.17$
- $\gamma_H = 0$

The system is heavily influenced by peripheral pressure, which decreases exponentially from $R = 1.05$ to $R = 0.84$ beginning at $t = 600$. As a result, the system has a discontinuity in a low-order derivative at $t = 600$.

The constant solution history is defined in terms of the physical parameters

$$P_a = P_0, \qquad P_v(t) = \frac{1}{1+\frac{R}{r}}P_0, \qquad H(t) = \frac{1}{\mathrm{RV}_{\mathrm{str}}}\frac{1}{1+\frac{r}{R}}P_0 \,.$$

To solve this system of equations in MATLAB, you need to code the equations, parameters, delays, and history before calling the delay differential equation solver `dde23`, which is meant for systems with constant time delays. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate, named files in a directory on the MATLAB path.

Define Physical Parameters

First, define the physical parameters of the problem as fields in a structure.

```
p.ca     = 1.55;
p.cv     = 519;
p.R      = 1.05;
p.r      = 0.068;
p.Vstr   = 67.9;
p.alpha0 = 93;
p.alphas = 93;
p.alphap = 93;
p.alphaH = 0.84;
p.beta0  = 7;
p.betas  = 7;
p.betap  = 7;
p.betaH  = 1.17;
p.gammaH = 0;
```

Code Delay

Next, create a variable `tau` to represent the constant time delay τ in the equations for the terms $P_a^{\tau}(t) = P_a(t-\tau)$.

```
tau = 4;
```

Code Equations

Now, create a function to code the equations. This function should have the signature `dydt = ddefun(t,y,Z,p)`, where:

- `t` is time (independent variable).
- `y` is the solution (dependent variable).
- `Z(n,j)` approximates the delays $y_n(d(j))$, where the delay $d(j)$ is given by component `j` of `dely(t,y)`.
- `p` is an optional fourth input used to pass in the values of the parameters.

The first three inputs are automatically passed to the function by the solver, and the variable names determine how you code the equations. The structure of parameters `p` is passed to the function when you call the solver. In this case the delays are represented with:

- `Z(:,1)` $\to P_a(t-\tau)$

```
function dydt = ddefun(t,y,Z,p)
    if t <= 600
      p.R = 1.05;
    else
```

```
      p.R = 0.21 * exp(600-t) + 0.84;
    end
    ylag = Z(:,1);
    Patau = ylag(1);
    Paoft = y(1);
    Pvoft = y(2);
    Hoft  = y(3);

    dPadt = - (1 / (p.ca * p.R)) * Paoft ...
            + (1/(p.ca * p.R)) * Pvoft ...
            + (1/p.ca) * p.Vstr * Hoft;

    dPvdt = (1 / (p.cv * p.R)) * Paoft...
            - ( 1 / (p.cv * p.R)...
            + 1 / (p.cv * p.r) ) * Pvoft;

    Ts = 1 / ( 1 + (Patau / p.alphas)^p.betas );
    Tp = 1 / ( 1 + (p.alphap / Paoft)^p.betap );

    dHdt = (p.alphaH * Ts) / (1 + p.gammaH * Tp) ...
           - p.betaH * Tp;

    dydt = [dPadt; dPvdt; dHdt];
end
```

Note: All functions are included as local functions at the end of the example.

Code Solution History

Next, create a vector to define the constant solution history for the three components P_a, P_v, and H. The solution history is the solution for times $t \le t_0$.

```
P0 = 93;
Paval = P0;
Pvval = (1 / (1 + p.R/p.r)) * P0;
Hval = (1 / (p.R * p.Vstr)) * (1 / (1 + p.r/p.R)) * P0;
history = [Paval; Pvval; Hval];
```

Solve Equation

Use `ddeset` to specify the presence of the discontinuity at $t = 600$. Finally, define the interval of integration $\left[t_0\ t_f\right]$ and solve the DDE using the `dde23` solver. Specify `ddefun` using an anonymous function to pass in the structure of parameters, `p`.

```
options = ddeset('Jumps',600);
tspan = [0 1000];
sol = dde23(@(t,y,Z) ddefun(t,y,Z,p), tau, history, tspan, options);
```

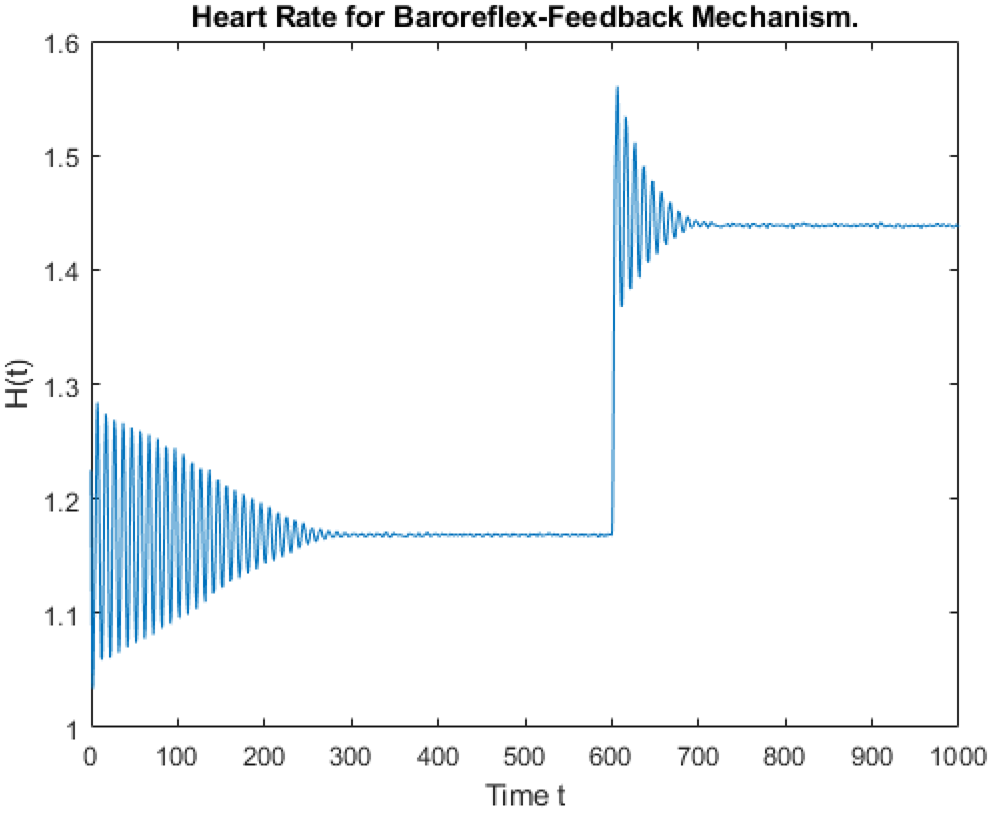
Plot Solution

The solution structure `sol` has the fields `sol.x` and `sol.y` that contain the internal time steps taken by the solver and corresponding solutions at those times. (If you need the solution at specific points, you can use `deval` to evaluate the solution at the specific points.)

Plot the third solution component (heart rate) against time.

```
plot(sol.x,sol.y(3,:))
title('Heart Rate for Baroreflex-Feedback Mechanism.')
```

```
xlabel('Time t')
ylabel('H(t)')
```



Local Functions

Listed here are the local helper functions that the DDE solver dde23 calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function dydt = ddefun(t,y,Z,p) % equation being solved
    if t <= 600
      p.R = 1.05;
    else
      p.R = 0.21 * exp(600-t) + 0.84;
    end
    ylag = Z(:,1);
    Patau = ylag(1);
    Paoft = y(1);
    Pvoft = y(2);
    Hoft  = y(3);

    dPadt = - (1 / (p.ca * p.R)) * Paoft ...
            + (1/(p.ca * p.R)) * Pvoft ...
            + (1/p.ca) * p.Vstr * Hoft;

    dPvdt = (1 / (p.cv * p.R)) * Paoft...
            - ( 1 / (p.cv * p.R)...
            + 1 / (p.cv * p.r) ) * Pvoft;
    Ts = 1 / ( 1 + (Patau / p.alphas)^p.betas );
    Tp = 1 / ( 1 + (p.alphap / Paoft)^p.betap );

    dHdt = (p.alphaH * Ts) / (1 + p.gammaH * Tp) ...
           - p.betaH * Tp;

    dydt = [dPadt; dPvdt; dHdt];
end
```

References

[1] Ottesen, J. T. “Modelling of the Baroreflex-Feedback Mechanism with Time-Delay.” *J. Math. Biol.* Vol. 36, Number 1, 1997, pp. 41–63.

See Also

dde23 | ddensd | ddesd | deval

More About

- “Solving Delay Differential Equations” on page 14-2
- “DDE of Neutral Type” on page 14-19

DDE of Neutral Type

This example shows how to use `ddensd` to solve a neutral DDE (delay differential equation), where delays appear in derivative terms. The problem was originally presented by Paul [1].

The equation is

$$y'(t) = 1 + y(t) - 2y\left(\frac{t}{2}\right)^2 - y'(t - \pi).$$

The history function is$y(t) = \cos(t)$ for $t \le 0$.

Since the equation has time delays in a y' term, the equation is called a *neutral DDE*. If the time delays are only present in y terms, then the equation would be a constant or state-dependent DDE, depending on what form the time delays have.

To solve this equation in MATLAB, you need to code the equation, delays, and history before calling the delay differential equation solver `ddensd`. You either can include these as local functions at the end of a file (as done here), or save them as separate files in a directory on the MATLAB path.

Code Delays

First, write functions to define the delays in the equation. The first term in the equation with a delay is $y\left(\frac{t}{2}\right)$.

```
function dy = dely(t,y)
    dy = t/2;
end
```

The other term in the equation with a delay is $y'(t - \pi)$.

```
function dyp = delyp(t,y)
    dyp = t-pi;
end
```

In this example, only one delay for y and one delay for y' are present. If there were more delays, then you can add them in these same function files, so that the functions return vectors instead of scalars.

Note: All functions are included as local functions at the end of the example.

Code Equation

Now, create a function to code the equation. This function should have the signature `yp = ddefun(t,y,ydel,ypdel)`, where:

- `t` is time (independent variable).
- `y` is the solution (dependent variable).
- `ydel` contains the delays for y.
- `ypdel` contains the delays for $y' = \frac{\mathrm{dy}}{\mathrm{dt}}$.

These inputs are automatically passed to the function by the solver, but the variable names determine how you code the equation. In this case:

- `ydel` → $y\left(\frac{t}{2}\right)$
- `ypdel` → $y'(t - \pi)$

```
function yp = ddefun(t,y,ydel,ypdel)
    yp = 1 + y - 2*ydel^2 - ypdel;
end
```

Code Solution History

Next, create a function to define the solution history. The solution history is the solution for times $t \le t_0$.

```
function y = history(t)
    y = cos(t);
end
```

Solve Equation

Finally, define the interval of integration $\left[t_0 \;\; t_f\right]$ and solve the DDE using the `ddensd` solver.

```
tspan = [0 pi];
sol = ddensd(@ddefun, @dely, @delyp, @history, [0,pi]);
```

Plot Solution

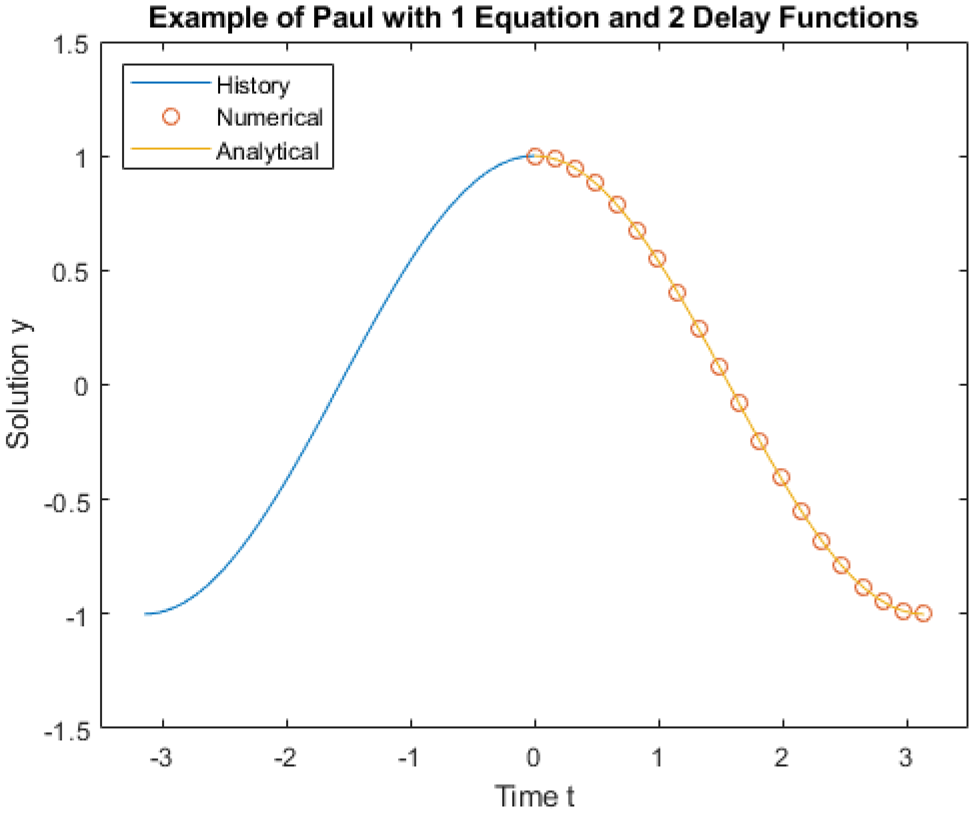
The solution structure `sol` has the fields `sol.x` and `sol.y` that contain the internal time steps taken by the solver and corresponding solutions at those times. However, you can use `deval` to evaluate the solution at the specific points.

Evaluate the solution at 20 equally spaced points between `0` and `pi`.

```
tn = linspace(0,pi,20);
yn = deval(sol,tn);
```

Plot the calculated solution and history against the analytical solution.

```
th = linspace(-pi,0);
yh = history(th);
ta = linspace(0,pi);
ya = cos(ta);

plot(th,yh,tn,yn,'o',ta,ya)
legend('History','Numerical','Analytical','Location','NorthWest')
xlabel('Time t')
ylabel('Solution y')
title('Example of Paul with 1 Equation and 2 Delay Functions')
axis([-3.5 3.5 -1.5 1.5])
```



Local Functions

Listed here are the local helper functions that the DDE solver `ddensd` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function yp = ddefun(t,y,ydel,ypdel) % equation being solved
    yp = 1 + y - 2*ydel^2 - ypdel;
end
%-------------------------------------------
function dy = dely(t,y) % delay for y
    dy = t/2;
end
%-------------------------------------------
function dyp = delyp(t,y) % delay for y'
    dyp = t-pi;
end
%-------------------------------------------
function y = history(t) % history function for t < 0
    y = cos(t);
end
%-------------------------------------------
```

References

[1] Paul, C.A.H. “A Test Set of Functional Differential Equations.” *Numerical Analysis Reports*. No. 243. Manchester, UK: Math Department, University of Manchester, 1994.

See Also

`dde23` | `ddensd` | `ddesd` | `deval`

More About

- “Solving Delay Differential Equations” on page 14-2
- “Initial Value DDE of Neutral Type” on page 14-23

Initial Value DDE of Neutral Type

This example shows how to use `ddensd` to solve a system of initial value DDEs (delay differential equations) with time-dependent delays. The example was originally presented by Jackiewicz [1].

The equation is

$$y'(t) = 2\cos(2t)y\left(\frac{t}{2}\right)^{2\cos(t)} + \log\left(y'\left(\frac{t}{2}\right)\right) - \log(2\cos(t)) - \sin(t)\,.$$

This equation is an *initial value* DDE because the time delays are zero at t_0. Therefore, a solution history is unnecessary to calculate a solution, only the initial values are needed:

$$y(0) = 1,$$

$$y'(0) = s\,.$$

s is the solution of $2 + \log(s) - \log(2) = 0$. The values of s that satisfy this equation are $s_1 = 2$ and $s_2 = 0.4063757399599599$.

Since the time delays in the equations are present in a y' term, this equation is called a *neutral DDE*.

To solve this equation in MATLAB, you need to code the equation and delays before calling the delay differential equation solver `ddensd`, which is the solver for neutral equations. You either can include the required functions as local functions at the end of a file (as done here), or save them as separate files in a directory on the MATLAB path.

Code Delays

First, write an anonymous function to define the delays in the equation. Since both y and y' have delays of the form $\frac{t}{2}$, only one function definition is required. This delay function is later passed to the solver twice, once to indicate the delay for y and once for y'.

```
delay = @(t,y) t/2;
```

Code Equation

Now, create a function to code the equation. This function should have the signature `yp = ddefun(t,y,ydel,ypdel)`, where:

- `t` is time (independent variable).
- `y` is the solution (dependent variable).
- `ydel` contains the delays for y.
- `ypdel` contains the delays for $y' = \frac{\mathrm{dy}}{\mathrm{dt}}$.

These inputs are automatically passed to the function by the solver, but the variable names determine how you code the equation. In this case:

- `ydel`$\rightarrow y\left(\frac{t}{2}\right)$
- `ypdel`$\rightarrow y'\left(\frac{t}{2}\right)$

```
function yp = ddefun(t,y,ydel,ypdel)
   yp = 2*cos(2*t)*ydel^(2*cos(t)) + log(ypdel) - log(2*cos(t)) - sin(t);
end
```

Note: All functions are included as local functions at the end of the example.

Solve Equation

Finally, define the interval of integration $[t_0 \ t_f]$ and the initial values, and then solve the DDE using the `ddensd` solver. Pass the initial values to the solver by specifying them in a cell array in the fourth input argument.

```
tspan = [0 0.1];
y0 = 1;
s1 = 2;
sol1 = ddensd(@ddefun, delay, delay, {y0,s1}, tspan);
```

Solve the equation a second time, this time using the alternate value of s for the initial condition.

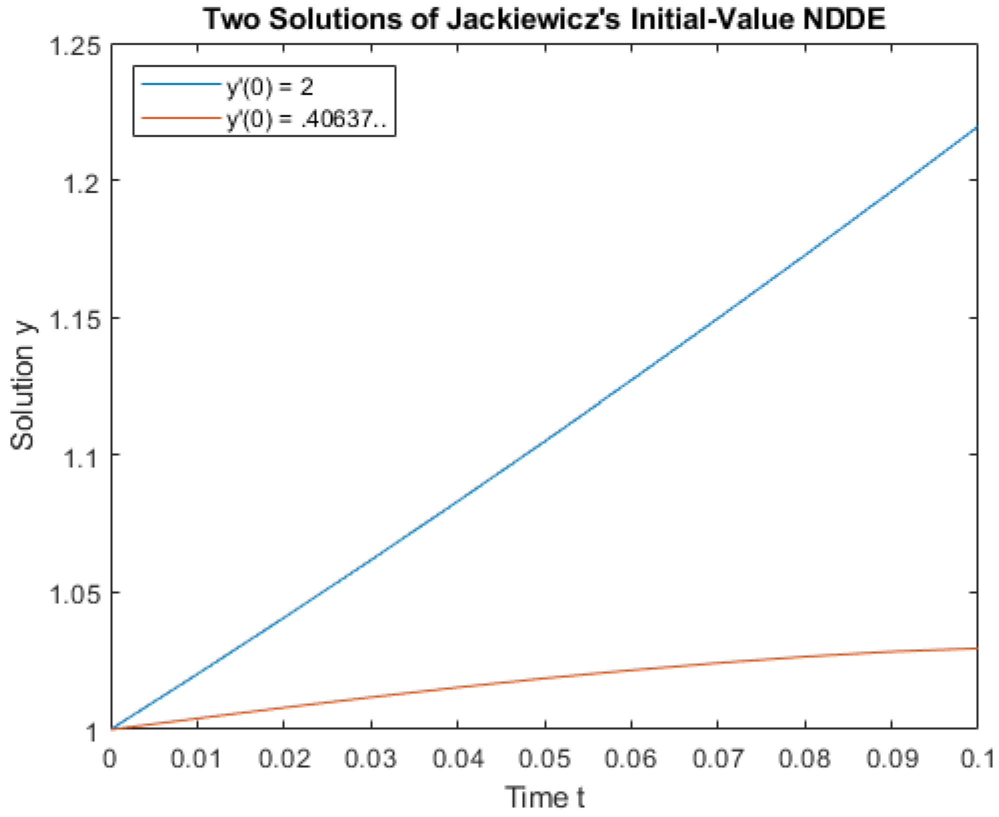
```
s2 = 0.4063757399599599;
sol2 = ddensd(@ddefun, delay, delay, {y0,s2}, tspan);
```

Plot Solution

The solution structures `sol1` and `sol2` have the fields `x` and `y` that contain the internal time steps taken by the solver and corresponding solutions at those times. However, you can use `deval` to evaluate the solution at the specific points.

Plot the two solutions to compare results.

```
plot(sol1.x,sol1.y,sol2.x,sol2.y);
legend('y''(0) = 2','y''(0) = .40637..','Location','NorthWest');
xlabel('Time t');
ylabel('Solution y');
title('Two Solutions of Jackiewicz''s Initial-Value NDDE');
```



Local Functions

Listed here are the local helper functions that the DDE solver `ddensd` calls to calculate the solution. Alternatively, you can save these functions as their own files in a directory on the MATLAB path.

```
function yp = ddefun(t,y,ydel,ypdel)
   yp = 2*cos(2*t)*ydel^(2*cos(t)) + log(ypdel) - log(2*cos(t)) - sin(t);
end
```

References

[1] Jackiewicz, Z. “One step Methods of any Order for Neutral Functional Differential Equations.” *SIAM Journal on Numerical Analysis*. Vol. 21, Number 3. 1984. pp. 486–511.

See Also

`dde23` | `ddensd` | `ddesd` | `deval`

More About

- “Solving Delay Differential Equations” on page 14-2
- “DDE of Neutral Type” on page 14-19

Numerical Integration

Integration to Find Arc Length

This example shows how to parametrize a curve and compute the arc length using `integral`.

Consider the curve parameterized by the equations

$x(t) = \sin(2t),\ \ y(t) = \cos(t),\ \ z(t) = t,$

where $t \epsilon [0,3\pi]$.

Create a three-dimensional plot of this curve.

```
t = 0:0.1:3*pi;
plot3(sin(2*t),cos(t),t)
```

The arc length formula says the length of the curve is the integral of the norm of the derivatives of the parameterized equations.

$$\int_0^{3\pi} \sqrt{4\cos^2(2t) + \sin^2(t) + 1}\ dt\,.$$

Define the integrand as an anonymous function.

```
f = @(t) sqrt(4*cos(2*t).^2 + sin(t).^2 + 1);
```

Integrate this function with a call to `integral`.

```
len = integral(f,0,3*pi)
```

```
len =
  17.2220
```

The length of this curve is about `17.2`.

See Also

`integral`

More About

- “Create Function Handle”
- “Singularity on Interior of Integration Domain” on page 15-5
- “Integration of Numeric Data” on page 15-8

Complex Line Integrals

This example shows how to calculate complex line integrals using the `'Waypoints'` option of the `integral` function. In MATLAB®, you use the `'Waypoints'` option to define a sequence of straight line paths from the first limit of integration to the first waypoint, from the first waypoint to the second, and so forth, and finally from the last waypoint to the second limit of integration.

Define the Integrand with an Anonymous Function

Integrate

$$\oint_C \frac{e^z}{z}\,dz$$

where C is a closed contour that encloses the simple pole of e^z/z at the origin.

Define the integrand with an anonymous function.

```
fun = @(z) exp(z)./z;
```

Integrate Without Using Waypoints

You can evaluate contour integrals of complex-valued functions with a parameterization. In general, a contour is specified, and then differentiated and used to parameterize the original integrand. In this case, specify the contour as the unit circle, but in all cases, the result is independent of the contour chosen.

```
g = @(theta) cos(theta) + 1i*sin(theta);
gprime = @(theta) -sin(theta) + 1i*cos(theta);
q1 = integral(@(t) fun(g(t)).*gprime(t),0,2*pi)
```

```
q1 = -0.0000 + 6.2832i
```

This method of parameterizing, although reliable, can be difficult and time consuming since a derivative must be calculated before the integration is performed. Even for simple functions, you need to write several lines of code to obtain the correct result. Since the result is the same with any closed contour that encloses the pole (in this case, the origin), instead you can use the `'Waypoints'` option of `integral` to construct a square or triangular path that encloses the pole.

Integrate Along a Contour That Encloses No Poles

If any limit of integration or element of the waypoints vector is complex, then `integral` performs the integration over a sequence of straight line paths in the complex plane. The natural direction around a contour is counterclockwise; specifying a clockwise contour is akin to multiplying by `-1`. Specify the contour in such a way that it encloses a single functional singularity. If you specify a contour that encloses no poles, then Cauchy's integral theorem guarantees that the value of the closed-loop integral is zero.

To see this, integrate `fun` around a square contour away from the origin. Use equal limits of integration to form a closed contour.

```
C = [2+i 2+2i 1+2i];
q = integral(fun,1+i,1+i,'Waypoints',C)
```

```
q = -3.3307e-16 + 6.6613e-16i
```

The result is on the order of `eps` and effectively zero.

Integrate Along a Contour with a Pole in the Interior

Specify a square contour that completely encloses the pole at the origin, and then integrate.

```
C = [1+i -1+i -1-i 1-i];
q2 = integral(fun,1,1,'Waypoints',C)
```

```
q2 = -0.0000 + 6.2832i
```

This result agrees with the `q1` calculated above, but uses much simpler code.

The exact answer for this problem is $2\pi i$.

```
2*pi*i
```

```
ans = 0.0000 + 6.2832i
```

See Also

`integral`

More About

- “Create Function Handle”
- “Singularity on Interior of Integration Domain” on page 15-5
- “Integration of Numeric Data” on page 15-8

Singularity on Interior of Integration Domain

This example shows how to split the integration domain to place a singularity on the boundary.

Define the Integrand with an Anonymous Function

The integrand of the complex-valued integral

$$\int_{-1}^{1}\int_{-1}^{1}\frac{1}{\sqrt{x+y}}\,dx\,dy$$

has a singularity when `x = y = 0` and is, in general, singular on the line `y = -x`.

Define this integrand with an anonymous function.

```
fun = @(x,y) ((x+y).^(-1/2));
```

Integrate Over a Square

Integrate `fun` over a square domain specified by $-1 \le x \le 1$ and $-1 \le y \le 1$.

```
format long
q = integral2(fun,-1,1,-1,1)
```

```
Warning: Non-finite result. The integration was unsuccessful. Singularity likely.

q =
                NaN +               NaNi
```

If there are singular values in the interior of the integration region, the integration fails to converge and returns a warning.

Split the Integration Domain into Two Triangles

You can redefine the integral by splitting the integration domain into complementary pieces and adding the smaller integrations together. Avoid integration errors and warnings by placing singularities on the boundary of the domain. In this case, you can split the square integration region into two triangles along the singular line `y = -x` and add the results.

```
q1 = integral2(fun,-1,1,-1,@(x)-x);
q2 = integral2(fun,-1,1,@(x)-x,1);
q = q1 + q2
```

```
q =
  3.771236166328258 - 3.771236166328255i
```

The integration succeeds when the singular values are on the boundary.

The exact value of this integral is

$$\frac{8\sqrt{2}}{3}(1-i)$$

```
8/3*sqrt(2)*(1-i)
```

```
ans =
  3.771236166328253 - 3.771236166328253i
```

See Also

integral | integral2 | integral3

More About

- “Create Function Handle”
- “Complex Line Integrals” on page 15-3
- “Integration of Numeric Data” on page 15-8

Analytic Solution to Integral of Polynomial

This example shows how to use the `polyint` function to integrate polynomial expressions analytically. Use this function to evaluate indefinite integral expressions of polynomials.

Define the Problem

Consider the real-valued indefinite integral,

$$\int \left(4x^5 - 2x^3 + x + 4\right) dx$$

The integrand is a polynomial, and the analytic solution is

$$\frac{2}{3}x^6 - \frac{1}{2}x^4 + \frac{1}{2}x^2 + 4x + k$$

where k is the constant of integration. Since the limits of integration are unspecified, the `integral` function family is not well-suited to solving this problem.

Express the Polynomial with a Vector

Create a vector whose elements represent the coefficients for each descending power of x.

```
p = [4 0 -2 0 1 4];
```

Integrate the Polynomial Analytically

Integrate the polynomial analytically using the `polyint` function. Specify the constant of integration with the second input argument.

```
k = 2;
I = polyint(p,k)
```

```
I = 1×7

    0.6667         0   -0.5000         0    0.5000    4.0000    2.0000
```

The output is a vector of coefficients for descending powers of x. This result matches the analytic solution above, but has a constant of integration `k = 2`.

See Also

`polyint` | `polyval`

More About

- “Singularity on Interior of Integration Domain” on page 15-5
- “Integration of Numeric Data” on page 15-8

Integration of Numeric Data

This example shows how to integrate a set of discrete velocity data numerically to approximate the distance traveled. The `integral` family only accepts function handles as inputs, so those functions cannot be used with discrete data sets. Use `trapz` or `cumtrapz` when a functional expression is not available for integration.

View the Velocity Data

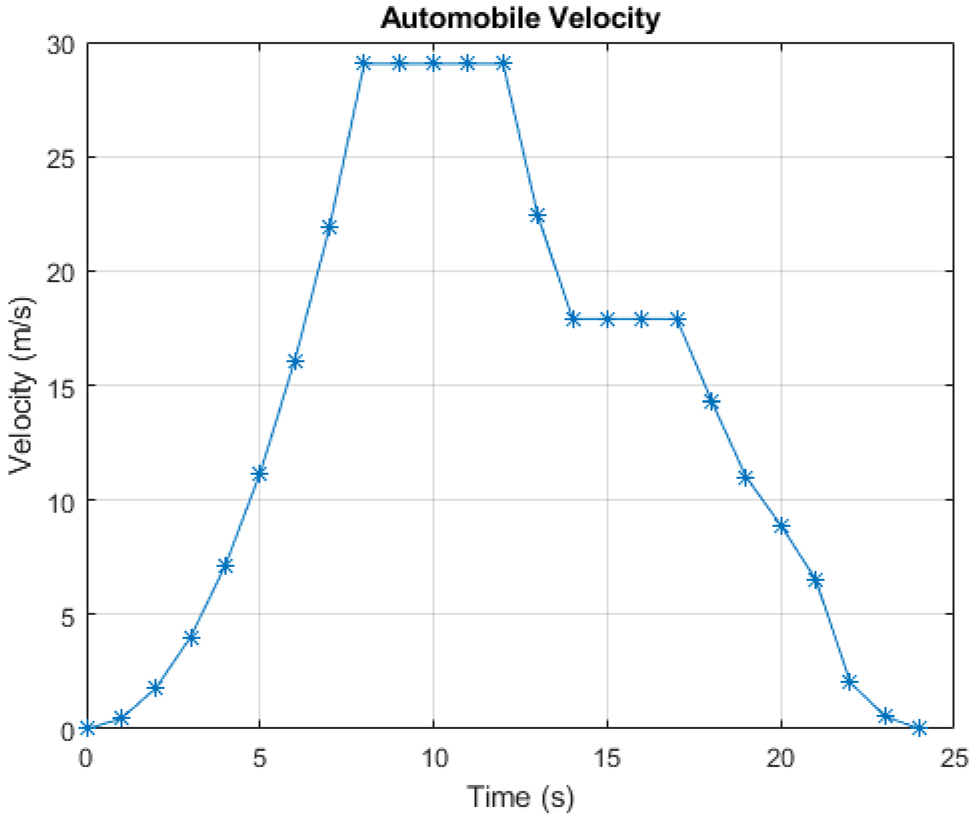
Consider the following velocity data and corresponding time data.

```
vel = [0 .45 1.79 4.02 7.15 11.18 16.09 21.90 29.05 29.05 ...
29.05 29.05 29.05 22.42 17.9 17.9 17.9 17.9 14.34 11.01 ...
8.9 6.54 2.03 0.55 0];
time = 0:24;
```

This data represents the velocity of an automobile (in m/s) taken at 1 s intervals over 24 s.

Plot the velocity data points and connect each point with a straight line.

```
figure
plot(time,vel,'-*')
grid on
title('Automobile Velocity')
xlabel('Time (s)')
ylabel('Velocity (m/s)')
```

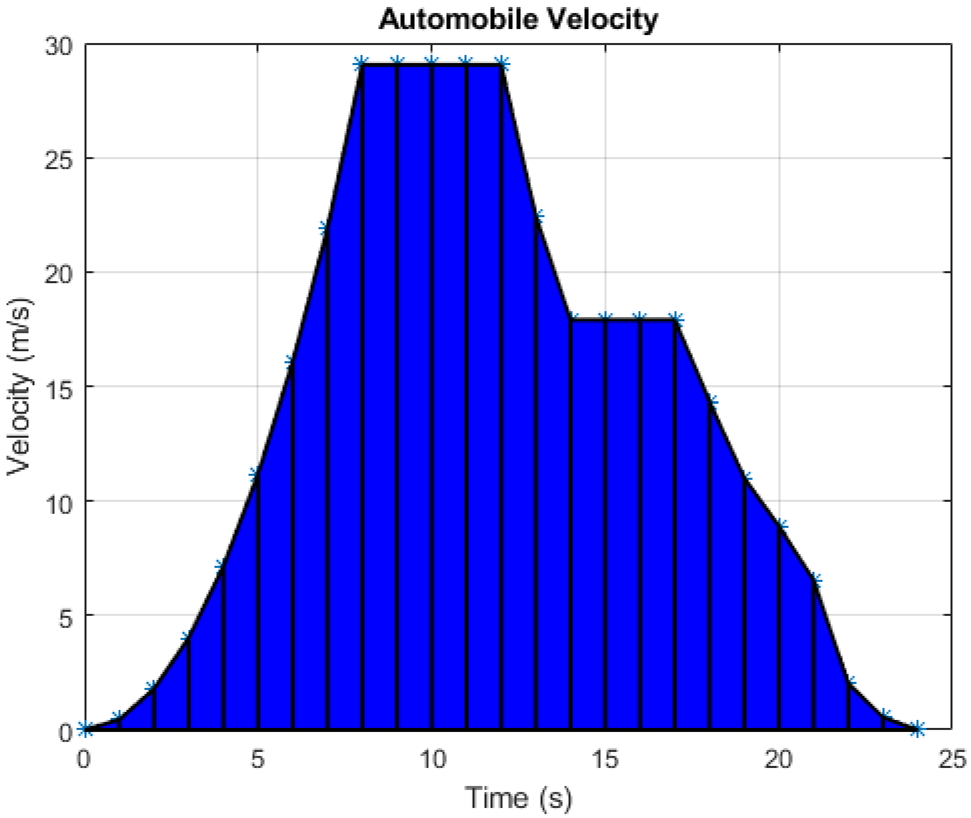


The slope is positive during periods of acceleration, zero during periods of constant velocity, and negative during periods of deceleration. At time `t = 0`, the vehicle is at rest with `vel(1) = 0` m/s. The vehicle accelerates until reaching a maximum velocity at `t = 8` s of `vel(9) = 29.05` m/s and maintains this velocity for 4 s. It then decelerates to `vel(14) = 17.9` m/s for 3 s and eventually back down to rest. Since this velocity curve has multiple discontinuities, a single continuous function cannot describe it.

Calculate the Total Distance Traveled

`trapz` performs discrete integration by using the data points to create trapezoids, so it is well suited to handling data sets with discontinuities. This method assumes linear behavior between the data points, and accuracy may be reduced when the behavior between data points is nonlinear. To illustrate, you can draw trapezoids onto the graph using the data points as vertices.

```
xverts = [time(1:end-1); time(1:end-1); time(2:end); time(2:end)];
yverts = [zeros(1,24); vel(1:end-1); vel(2:end); zeros(1,24)];
p = patch(xverts,yverts,'b','LineWidth',1.5);
```



`trapz` calculates the area under a set of discrete data by breaking the region into trapezoids. The function then adds the area of each trapezoid to compute the total area.

Calculate the total distance traveled by the automobile (corresponding to the shaded area) by integrating the velocity data numerically using `trapz`. By default, the spacing between points is assumed to be `1` if you use the syntax `trapz(Y)`. However, you can specify a different uniform or nonuniform spacing `X` with the syntax `trapz(X,Y)`. In this case, the spacing between readings in the `time` vector is `1`, so it is acceptable to use the default spacing.

```
distance = trapz(vel)
```

```
distance = 345.2200
```

The distance traveled by the automobile in `t = 24` s is about 345.22 m.

Plot Cumulative Distance Traveled

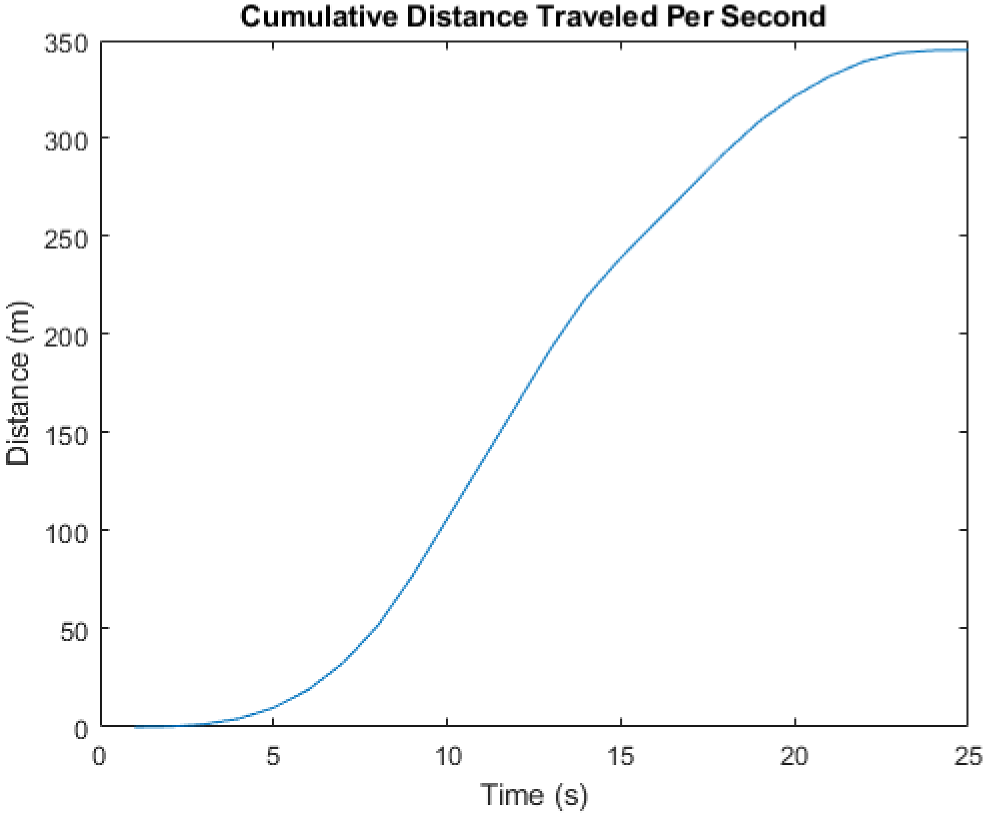
The `cumtrapz` function is closely related to `trapz`. While `trapz` returns only the final integration value, `cumtrapz` also returns intermediate values in a vector.

Calculate the cumulative distance traveled and plot the result.

```
cdistance = cumtrapz(vel);
T = table(time',cdistance','VariableNames',{'Time','CumulativeDistance'})
```

```
T=25×2 table
    Time    CumulativeDistance
    ____    __________________

      0                0      
      1            0.225      
      2            1.345      
      3             4.25      
      4            9.835      
      5               19      
      6           32.635      
      7            51.63      
      8           77.105      
      9           106.15      
     10            135.2      
     11           164.25      
     12           193.31      
     13           219.04      
     14            239.2      
     15            257.1      
      ⋮
```

```
plot(cdistance)
title('Cumulative Distance Traveled Per Second')
xlabel('Time (s)')
ylabel('Distance (m)')
```



See Also

`cumtrapz` | `integral` | `trapz`

More About

- "Singularity on Interior of Integration Domain" on page 15-5
- "Analytic Solution to Integral of Polynomial" on page 15-7

Calculate Tangent Plane to Surface

This example shows how to approximate gradients of a function by finite differences. It then shows how to plot a tangent plane to a point on the surface by using these approximated gradients.

Create the function $f(x, y) = x^2 + y^2$ using a function handle.

```
f = @(x,y) x.^2 + y.^2;
```

Approximate the partial derivatives of $f(x, y)$ with respect to x and y by using the `gradient` function. Choose a finite difference length that is the same as the mesh size.

```
[xx,yy] = meshgrid(-5:0.25:5);
[fx,fy] = gradient(f(xx,yy),0.25);
```

The tangent plane to a point on the surface, $P = (x_0, y_0, f(x_0, y_0))$, is given by

$$z = f(x_0, y_0) + \frac{\partial f(x_0, y_0)}{\partial x}(x - x_0) + \frac{\partial f(x_0, y_0)}{\partial y}(y - y_0)\,.$$

The `fx` and `fy` matrices are approximations to the partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$. The point of interest in this example, where the tangent plane meets the functional surface, is `(x0,y0) = (1,2)`. The function value at this point of interest is `f(1,2) = 5`.

To approximate the tangent plane `z` you need to find the value of the derivatives at the point of interest. Obtain the index of that point, and find the approximate derivatives there.

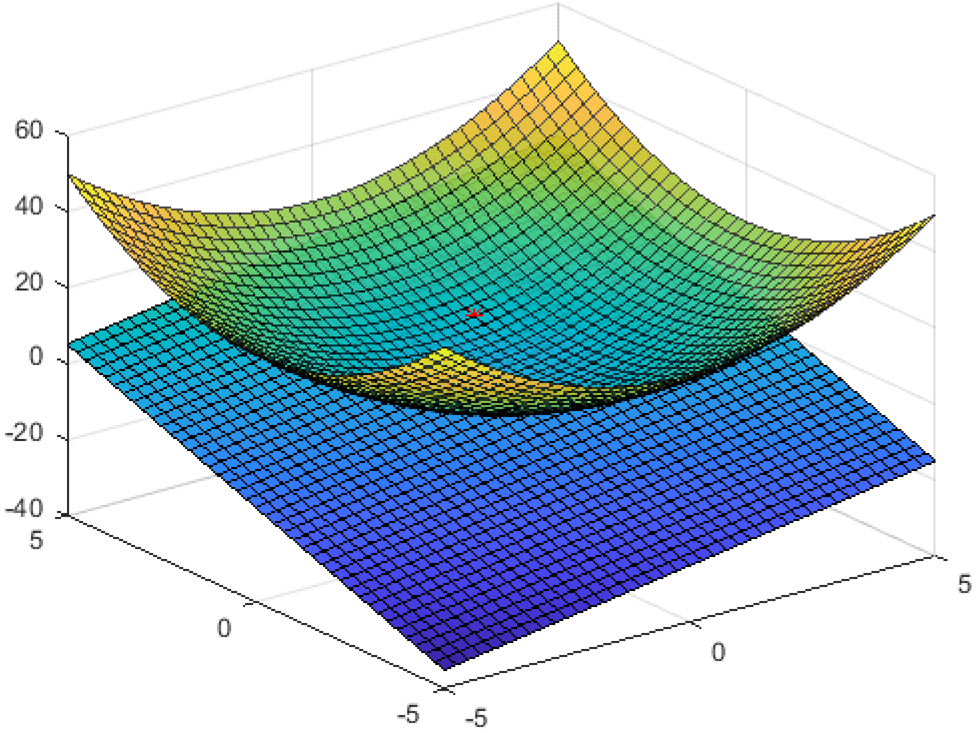
```
x0 = 1;
y0 = 2;
t = (xx == x0) & (yy == y0);
indt = find(t);
fx0 = fx(indt);
fy0 = fy(indt);
```

Create a function handle with the equation of the tangent plane `z`.

```
z = @(x,y) f(x0,y0) + fx0*(x-x0) + fy0*(y-y0);
```

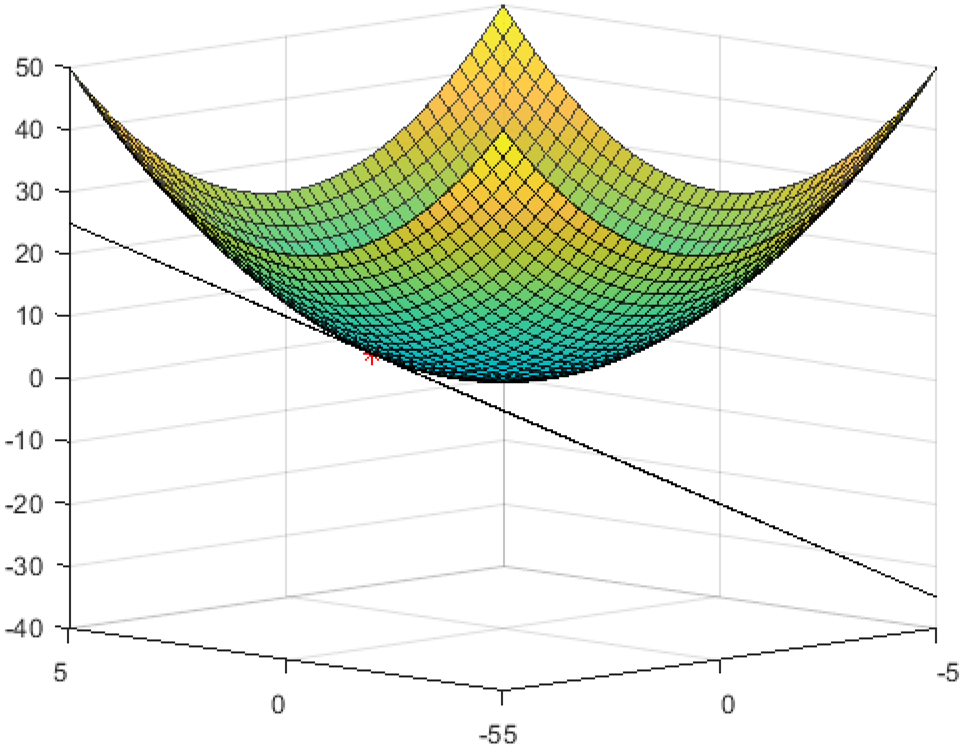
Plot the original function $f(x, y)$, the point `P`, and a piece of plane `z` that is tangent to the function at `P`.

```
surf(xx,yy,f(xx,yy),'EdgeAlpha',0.7,'FaceAlpha',0.9)
hold on
surf(xx,yy,z(xx,yy))
plot3(1,2,f(1,2),'r*')
```



View a side profile.

```
view(-135,9)
```



See Also

`gradient`

More About

- “Create Function Handle”

Fourier Transforms

Fourier Transforms

The Fourier transform is a mathematical formula that relates a signal sampled in time or space to the same signal sampled in frequency. In signal processing, the Fourier transform can reveal important characteristics of a signal, namely, its frequency components.

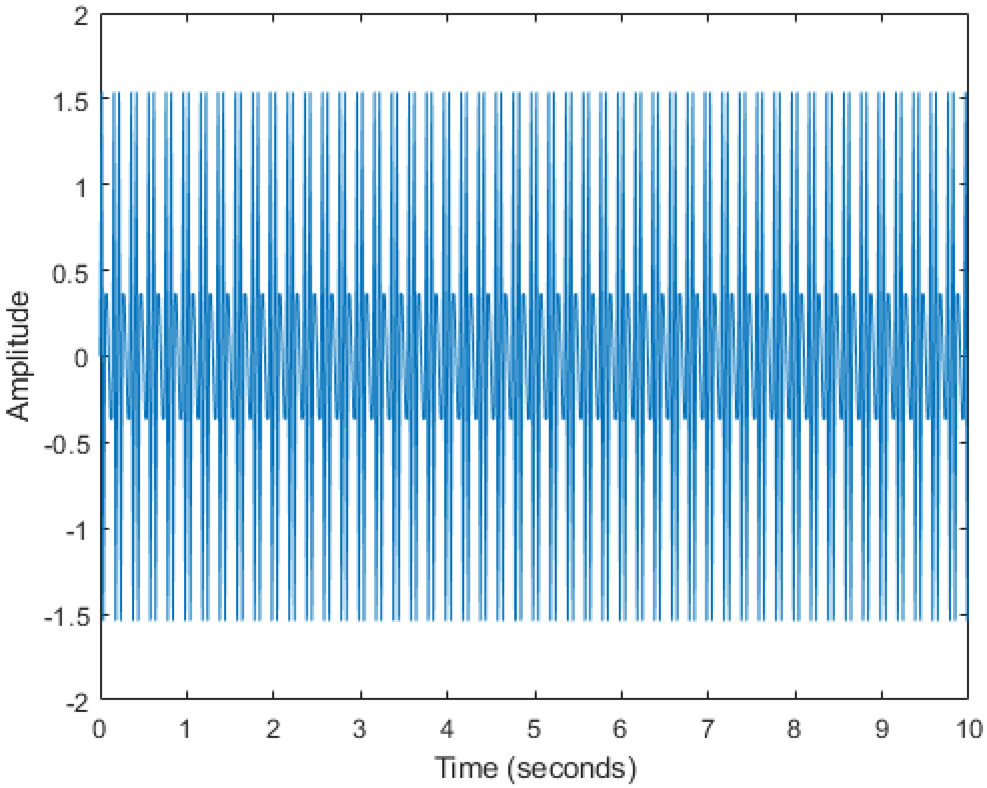
The Fourier transform is defined for a vector x with n uniformly sampled points by

$$y_{k+1} = \sum_{j=0}^{n-1} \omega^{jk} x_{j+1} .$$

$\omega = e^{-2\pi i/n}$ is one of n complex roots of unity where i is the imaginary unit. For x and y, the indices j and k range from 0 to $n - 1$.

The `fft` function in MATLAB® uses a fast Fourier transform algorithm to compute the Fourier transform of data. Consider a sinusoidal signal `x` that is a function of time `t` with frequency components of 15 Hz and 20 Hz. Use a time vector sampled in increments of $\frac{1}{50}$ of a second over a period of 10 seconds.

```
Ts = 1/50;
t = 0:Ts:10-Ts;
x = sin(2*pi*15*t) + sin(2*pi*20*t);
plot(t,x)
xlabel('Time (seconds)')
ylabel('Amplitude')
```

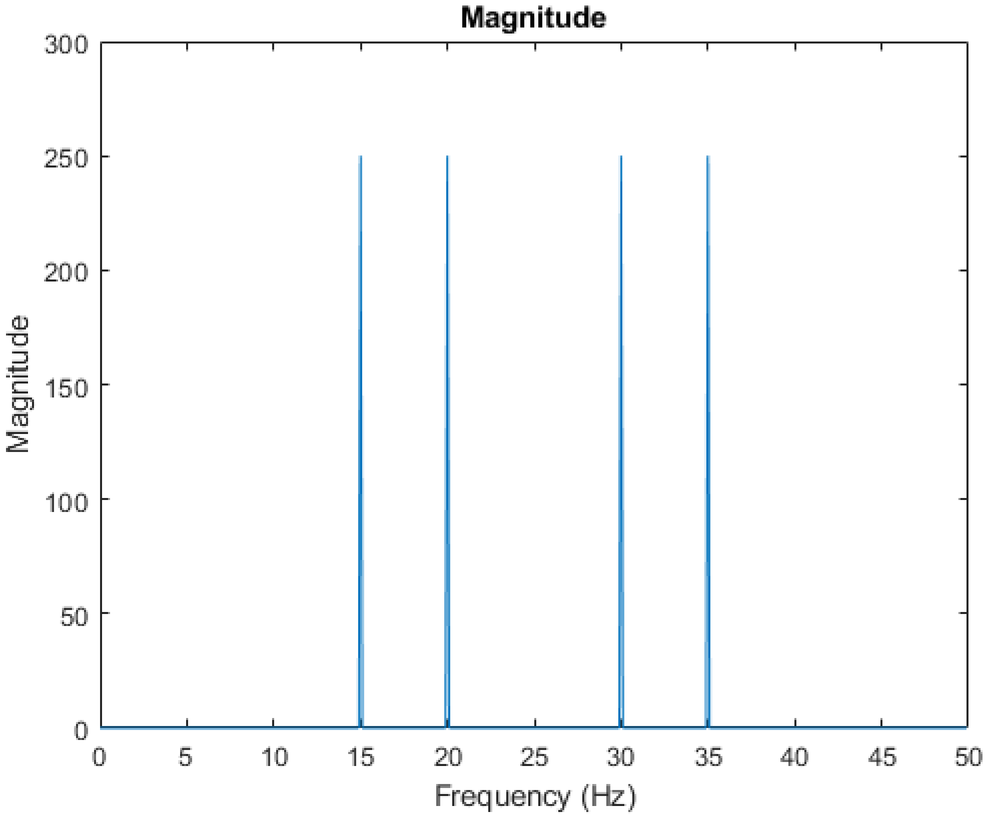


Compute the Fourier transform of the signal, and create the vector f that corresponds to the signal's sampling in frequency space.

```
y = fft(x);
fs = 1/Ts;
f = (0:length(y)-1)*fs/length(y);
```

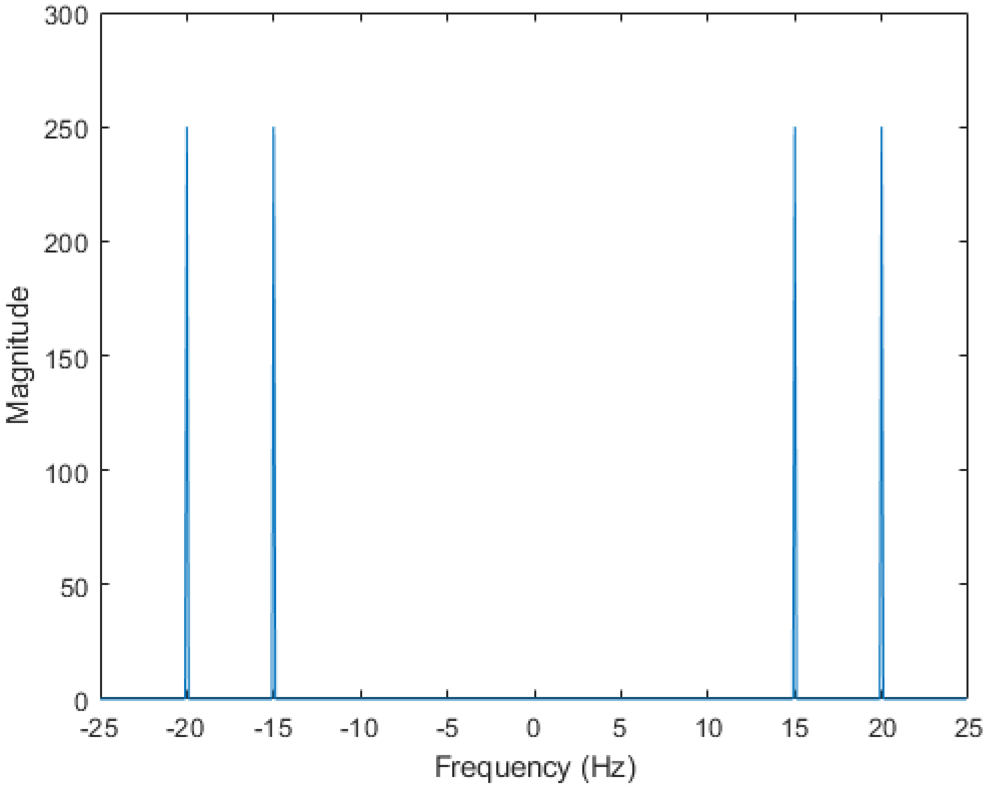
When you plot the magnitude of the signal as a function of frequency, the spikes in magnitude correspond to the signal's frequency components of 15 Hz and 20 Hz.

```
plot(f,abs(y))
xlabel('Frequency (Hz)')
ylabel('Magnitude')
title('Magnitude')
```



The transform also produces a mirror copy of the spikes, which correspond to the signal's negative frequencies. To better visualize this periodicity, you can use the fftshift function, which performs a zero-centered, circular shift on the transform.

```
n = length(x);
fshift = (-n/2:n/2-1)*(fs/n);
yshift = fftshift(y);
plot(fshift,abs(yshift))
xlabel('Frequency (Hz)')
ylabel('Magnitude')
```



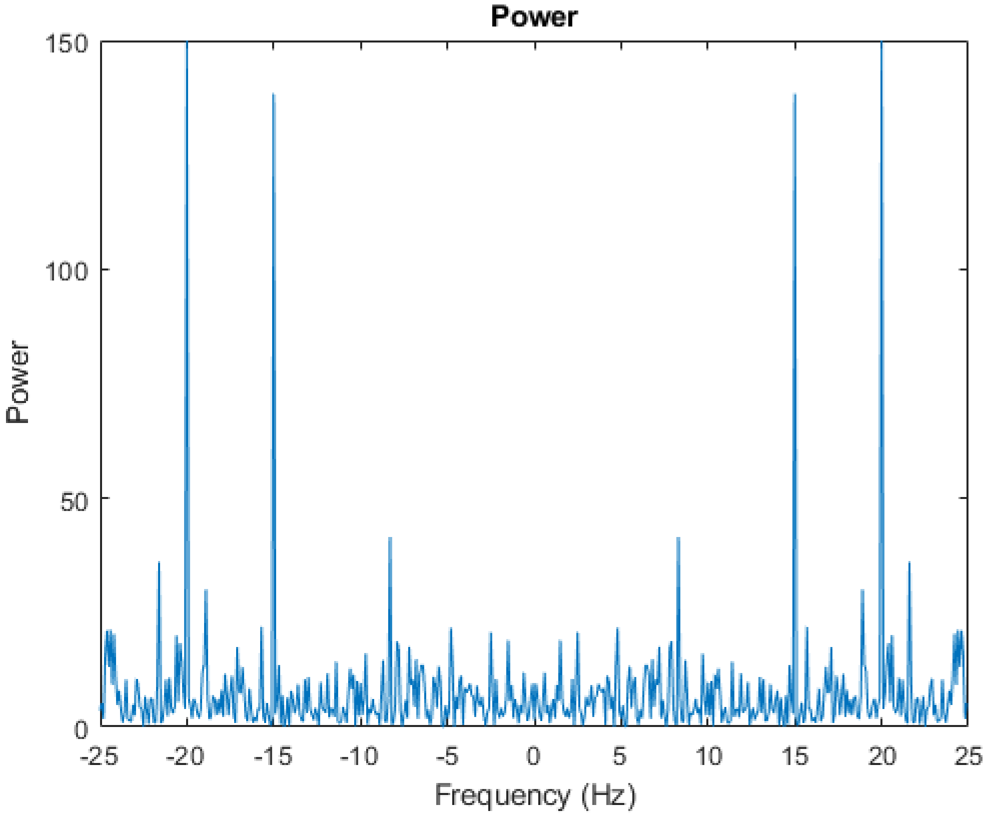
Noisy Signals

In scientific applications, signals are often corrupted with random noise, disguising their frequency components. The Fourier transform can process out random noise and reveal the frequencies. For example, create a new signal, `xnoise`, by injecting Gaussian noise into the original signal, `x`.

```
rng('default')
xnoise = x + 2.5*randn(size(t));
```

Signal power as a function of frequency is a common metric used in signal processing. Power is the squared magnitude of a signal's Fourier transform, normalized by the number of frequency samples. Compute and plot the power spectrum of the noisy signal centered at the zero frequency. Despite noise, you can still make out the signal's frequencies due to the spikes in power.

```
ynoise = fft(xnoise);
ynoiseshift = fftshift(ynoise);
power = abs(ynoiseshift).^2/n;
plot(fshift,power)
title('Power')
xlabel('Frequency (Hz)')
ylabel('Power')
```

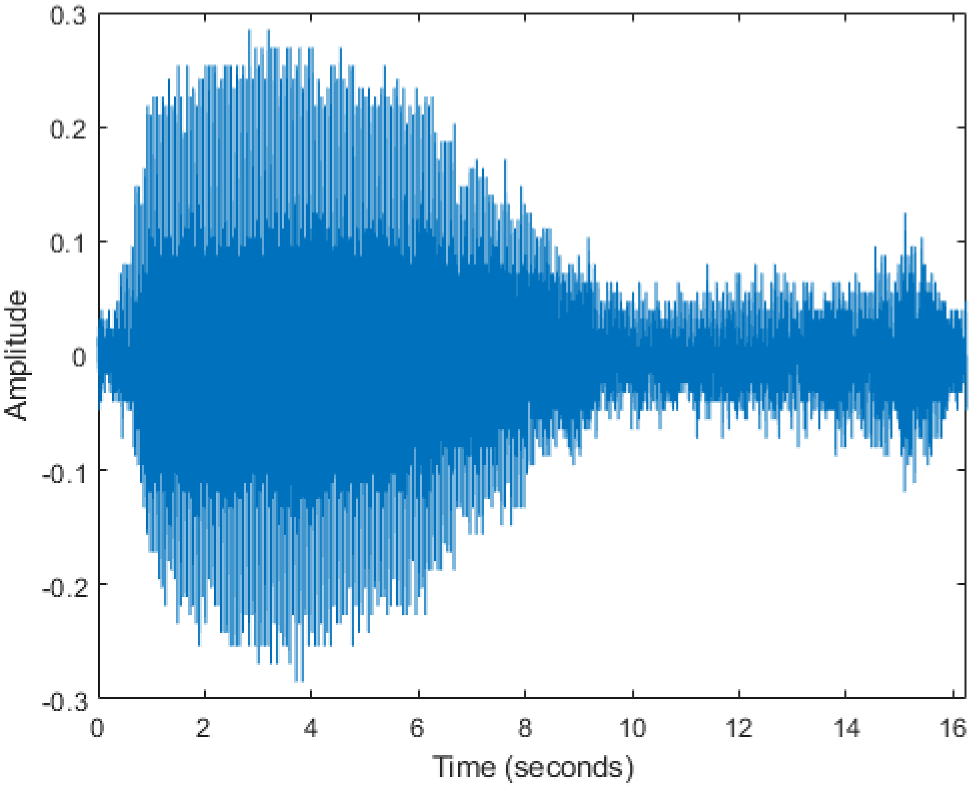


Computational Efficiency

Using the Fourier transform formula directly to compute each of the *n* elements of *y* requires on the order of n^2 floating-point operations. The fast Fourier transform algorithm requires only on the order of *n*log*n* operations to compute. This computational efficiency is a big advantage when processing data that has millions of data points. Many specialized implementations of the fast Fourier transform algorithm are even more efficient when *n* is a power of 2.

Consider audio data collected from underwater microphones off the coast of California. This data can be found in a library maintained by the Cornell University Bioacoustics Research Program. Load and format a subset of the data in `bluewhale.au`, which contains a Pacific blue whale vocalization. Because blue whale calls are low-frequency sounds, they are barely audible to humans. The time scale in the data is compressed by a factor of 10 to raise the pitch and make the call more clearly audible. You can use the command `sound(x,fs)` to listen to the entire audio file.

```
whaleFile = 'bluewhale.au';
[x,fs] = audioread(whaleFile);
whaleMoan = x(2.45e4:3.10e4);
t = 10*(0:1/fs:(length(whaleMoan)-1)/fs);

plot(t,whaleMoan)
xlabel('Time (seconds)')
ylabel('Amplitude')
xlim([0 t(end)])
```

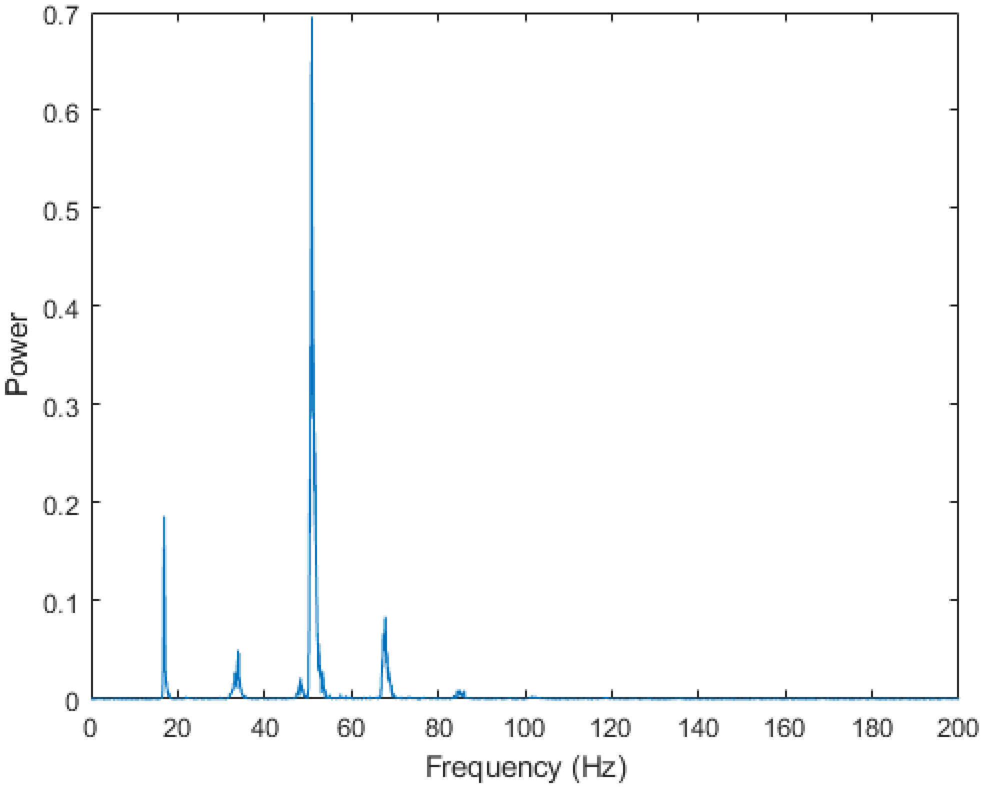


Specify a new signal length that is the next power of 2 greater than the original length. Then, use `fft` to compute the Fourier transform using the new signal length. `fft` automatically pads the data with zeros to increase the sample size. This padding can make the transform computation significantly faster, particularly for sample sizes with large prime factors.

```
m = length(whaleMoan);
n = pow2(nextpow2(m));
y = fft(whaleMoan,n);
```

Plot the power spectrum of the signal. The plot indicates that the moan consists of a fundamental frequency around 17 Hz and a sequence of harmonics, where the second harmonic is emphasized.

```
f = (0:n-1)*(fs/n)/10; % frequency vector
power = abs(y).^2/n;   % power spectrum
plot(f(1:floor(n/2)),power(1:floor(n/2)))
xlabel('Frequency (Hz)')
ylabel('Power')
```



See Also

`ifft` | `fft2` | `fftn` | `fftw` | `fft` | `fftshift` | `nextpow2`

Related Examples

- "2-D Fourier Transforms" on page 16-17

Basic Spectral Analysis

The Fourier transform is a tool for performing frequency and power spectrum analysis of time-domain signals.

Spectral Analysis Quantities

Spectral analysis studies the frequency spectrum contained in discrete, uniformly sampled data. The Fourier transform is a tool that reveals frequency components of a time- or space-based signal by representing it in frequency space. The following table lists common quantities used to characterize and interpret signal properties. To learn more about the Fourier transform, see “Fourier Transforms” on page 16-2.

Quantity	Description
`x`	Sampled data
`n = length(x)`	Number of samples
`fs`	Sample frequency (samples per unit time or space)
`dt = 1/fs`	Time or space increment per sample
`t = (0:n-1)/fs`	Time or space range for data
`y = fft(x)`	Discrete Fourier transform of data (DFT)
`abs(y)`	Amplitude of the DFT
`(abs(y).^2)/n`	Power of the DFT
`fs/n`	Frequency increment
`f = (0:n-1)*(fs/n)`	Frequency range
`fs/2`	Nyquist frequency (midpoint of frequency range)

Noisy Signal

The Fourier transform can compute the frequency components of a signal that is corrupted by random noise.

Create a signal with component frequencies at 15 Hz and 40 Hz, and inject random Gaussian noise.

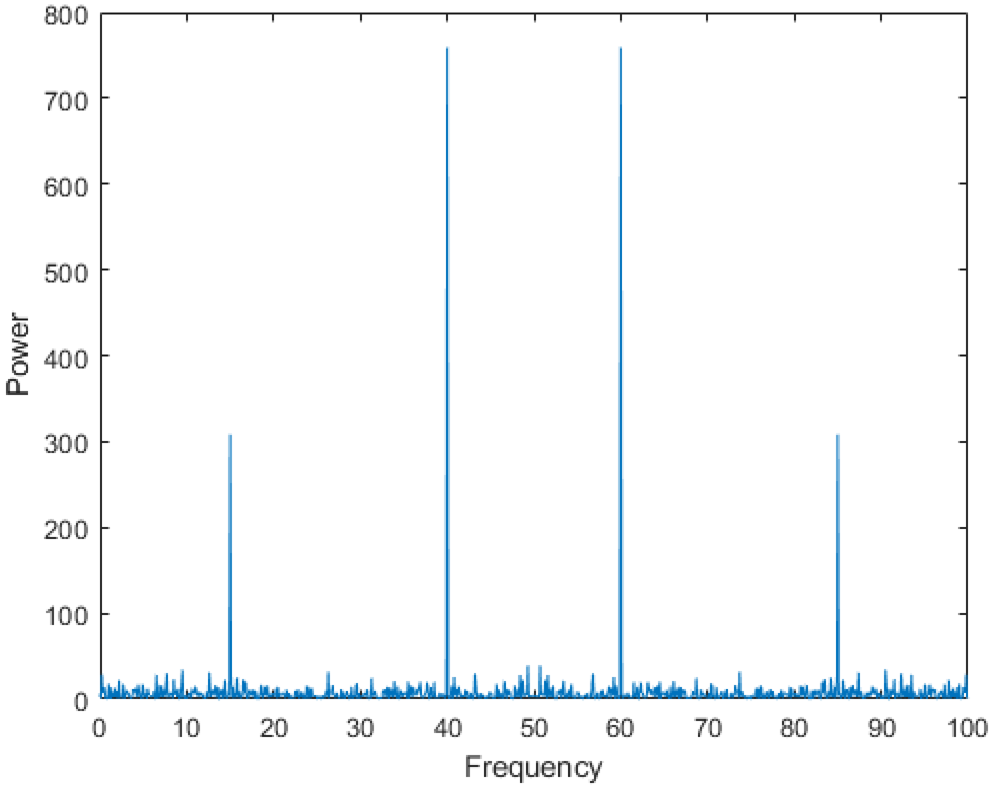
```
rng('default')
fs = 100;                               % sample frequency (Hz)
t = 0:1/fs:10-1/fs;                     % 10 second span time vector
x = (1.3)*sin(2*pi*15*t) ...            % 15 Hz component
  + (1.7)*sin(2*pi*40*(t-2)) ...        % 40 Hz component
  + 2.5*randn(size(t));                 % Gaussian noise;
```

The Fourier transform of the signal identifies its frequency components. In MATLAB®, the `fft` function computes the Fourier transform using a fast Fourier transform algorithm. Use `fft` to compute the discrete Fourier transform of the signal.

```
y = fft(x);
```

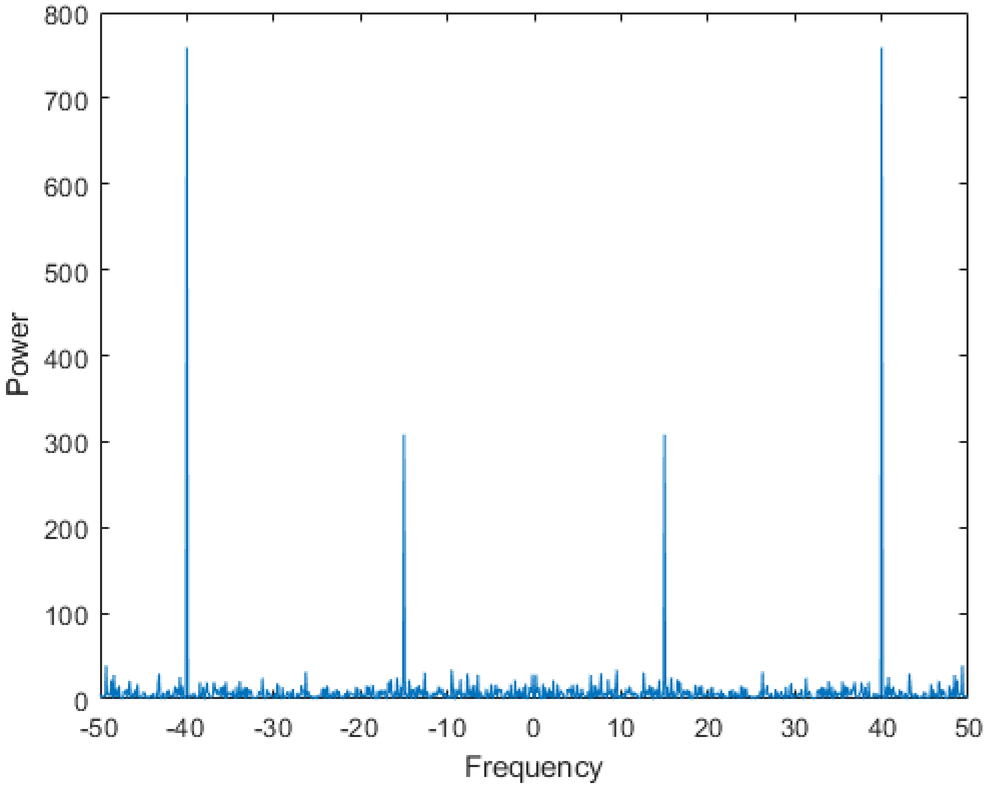
Plot the power spectrum as a function of frequency. While noise disguises a signal's frequency components in time-based space, the Fourier transform reveals them as spikes in power.

```
n = length(x);          % number of samples
f = (0:n-1)*(fs/n);     % frequency range
power = abs(y).^2/n;    % power of the DFT

plot(f,power)
xlabel('Frequency')
ylabel('Power')
```



In many applications, it is more convenient to view the power spectrum centered at 0 frequency because it better represents the signal's periodicity. Use the `fftshift` function to perform a circular shift on `y`, and plot the 0-centered power.

```
y0 = fftshift(y);         % shift y values
f0 = (-n/2:n/2-1)*(fs/n); % 0-centered frequency range
power0 = abs(y0).^2/n;    % 0-centered power

plot(f0,power0)
xlabel('Frequency')
ylabel('Power')
```



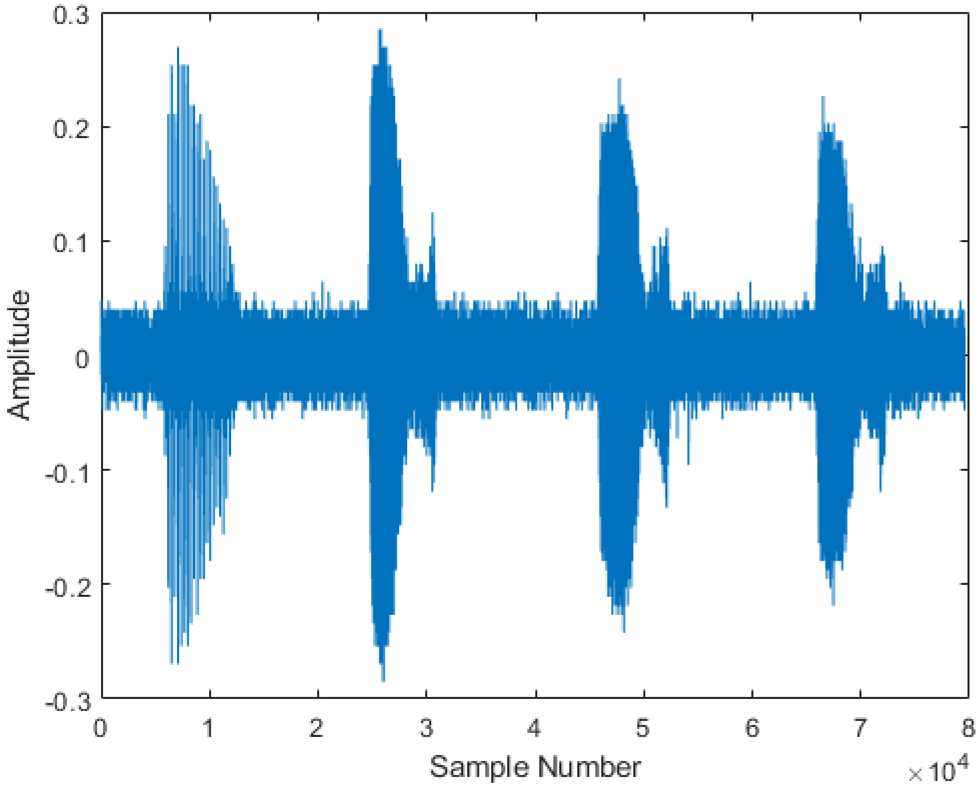
Audio Signal

You can use the Fourier transform to analyze the frequency spectrum of audio data.

The file `bluewhale.au` contains audio data from a Pacific blue whale vocalization recorded by underwater microphones off the coast of California. The file is from the library of animal vocalizations maintained by the Cornell University Bioacoustics Research Program.

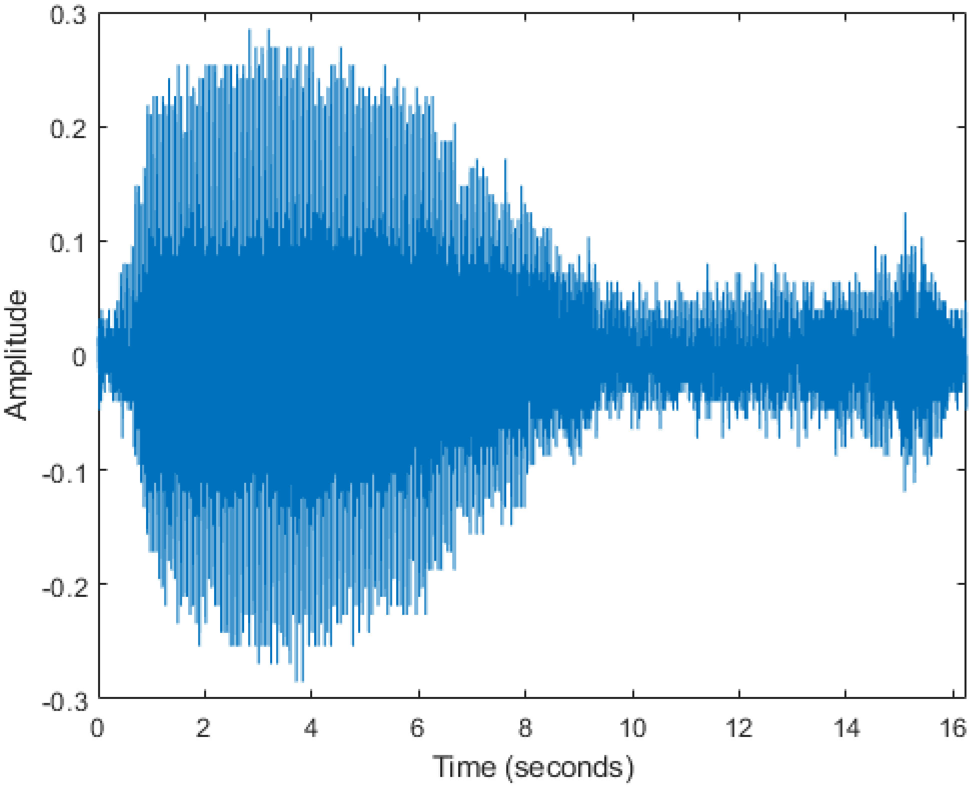
Because blue whale calls are so low, they are barely audible to humans. The time scale in the data is compressed by a factor of 10 to raise the pitch and make the call more clearly audible. Read and plot the audio data. You can use the command `sound(x,fs)` to listen to the audio.

```
whaleFile = 'bluewhale.au';
[x,fs] = audioread(whaleFile);

plot(x)
xlabel('Sample Number')
ylabel('Amplitude')
```



The first sound is a "trill" followed by three "moans". This example analyzes a single moan. Specify new data that approximately consists of the first moan, and correct the time data to account for the factor-of-10 speed-up. Plot the truncated signal as a function of time.

```
moan = x(2.45e4:3.10e4);
t = 10*(0:1/fs:(length(moan)-1)/fs);

plot(t,moan)
xlabel('Time (seconds)')
ylabel('Amplitude')
xlim([0 t(end)])
```

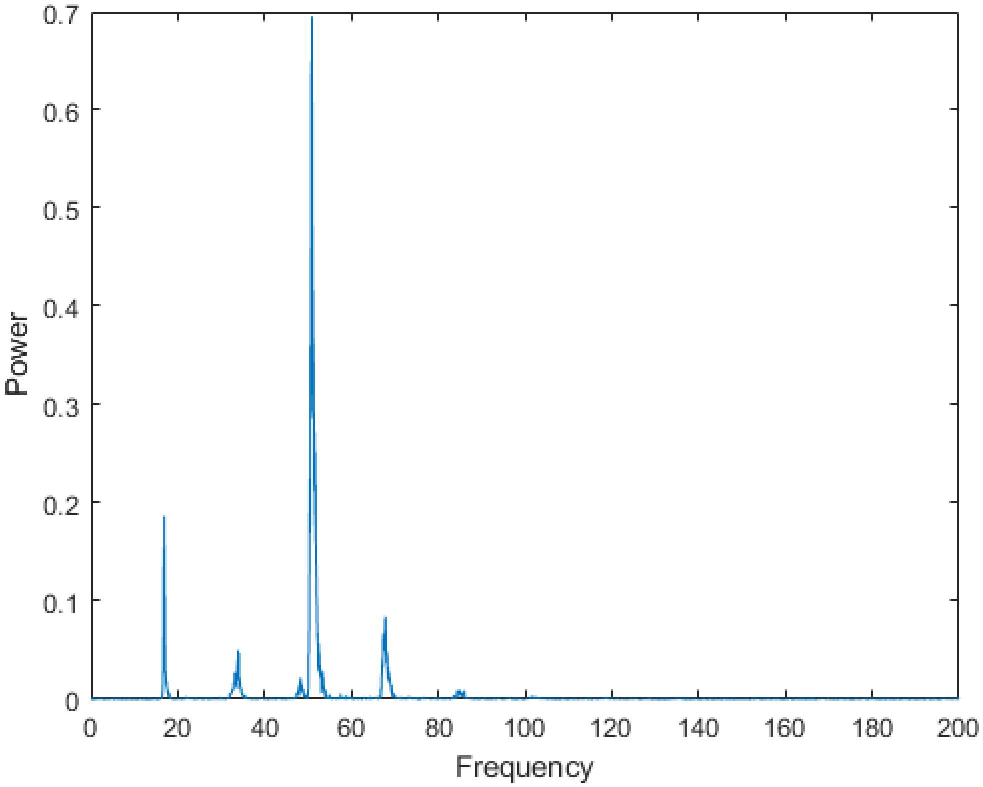


The Fourier transform of the data identifies frequency components of the audio signal. In some applications that process large amounts of data with `fft`, it is common to resize the input so that the number of samples is a power of 2. This can make the transform computation significantly faster, particularly for sample sizes with large prime factors. Specify a new signal length `n` that is a power of 2, and use the `fft` function to compute the discrete Fourier transform of the signal. `fft` automatically pads the original data with zeros to increase the sample size.

```
m = length(moan);       % original sample length
n = pow2(nextpow2(m));  % transform length
y = fft(moan,n);        % DFT of signal
```

Adjust the frequency range due to the speed-up factor, and compute and plot the power spectrum of the signal. The plot indicates that the moan consists of a fundamental frequency around 17 Hz and a sequence of harmonics, where the second harmonic is emphasized.

```
f = (0:n-1)*(fs/n)/10;
power = abs(y).^2/n;

plot(f(1:floor(n/2)),power(1:floor(n/2)))
xlabel('Frequency')
ylabel('Power')
```



See Also

`ifft` | `fft2` | `fftn` | `fft` | `fftshift` | `nextpow2`

Related Examples

- "Fourier Transforms" on page 16-2
- "2-D Fourier Transforms" on page 16-17

Polynomial Interpolation Using FFT

Use the fast Fourier transform (FFT) to estimate the coefficients of a trigonometric polynomial that interpolates a set of data.

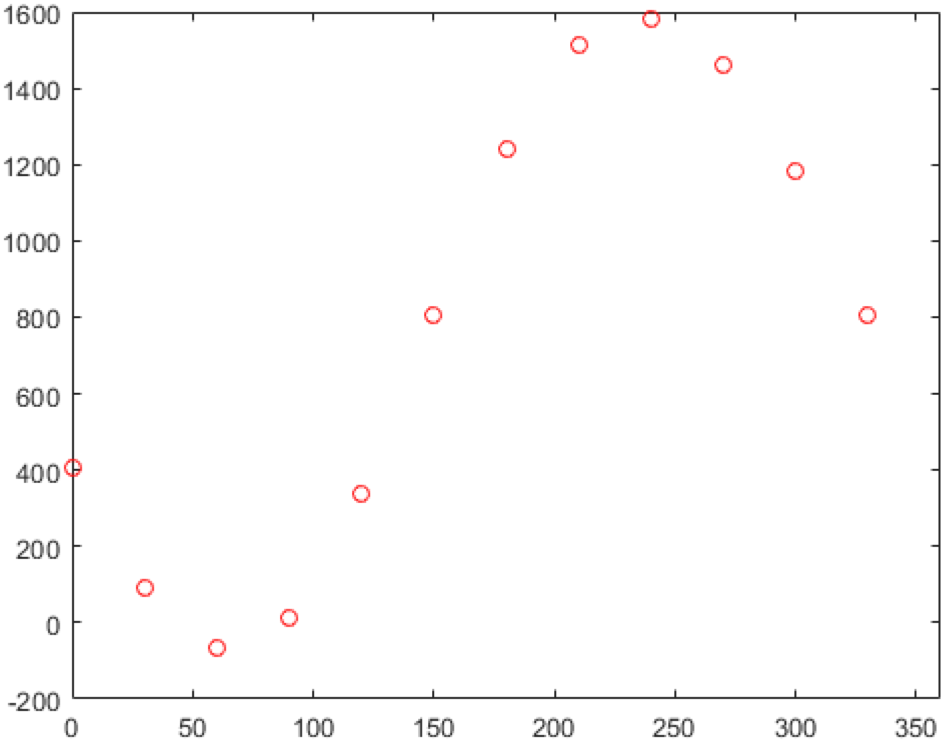
FFT in Mathematics

The FFT algorithm is associated with applications in signal processing, but it can also be used more generally as a fast computational tool in mathematics. For example, coefficients c_i of an nth degree polynomial $c_1x^n + c_2x^{n-1} + \ldots + c_nx + c_{n+1}$ that interpolates a set of data are commonly computed by solving a straightforward system of linear equations. While studying asteroid orbits in the early 19th century, Carl Friedrich Gauss discovered a mathematical shortcut for computing the coefficients of a polynomial interpolant by splitting the problem up into smaller subproblems and combining the results. His method was equivalent to estimating the discrete Fourier transform of his data.

Interpolate Asteroid Data

In a paper by Gauss, he describes an approach to estimating the orbit of the Pallas asteroid. He starts with the following twelve 2-D positional data points `x` and `y`.

```
x = 0:30:330;
y = [408 89 -66 10 338 807 1238 1511 1583 1462 1183 804];
plot(x,y,'ro')
xlim([0 360])
```



Gauss models the asteroid's orbit with a trigonometric polynomial of the following form.

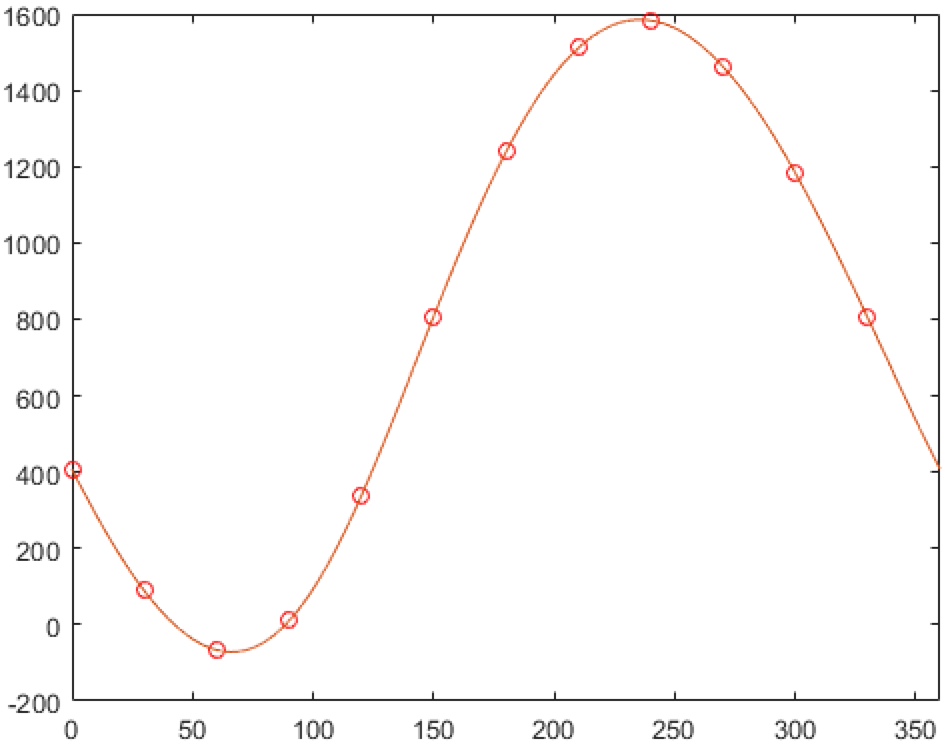
$$
\begin{aligned}
y = a_0 \; + \; & a_1\cos(2\pi(x/360)) + b_1\sin(2\pi(x/360)) \\
& a_2\cos(2\pi(2x/360)) + b_2\sin(2\pi(2x/360)) \\
& \ldots \\
& a_5\cos(2\pi(5x/360)) + b_5\sin(2\pi(5x/360)) \\
& a_6\cos(2\pi(6x/360))
\end{aligned}
$$

Use `fft` to compute the coefficients of the polynomial.

```
m = length(y);
n = floor((m+1)/2);
z = fft(y)/m;

a0 = z(1);
an = 2*real(z(2:n));
a6 = z(n+1);
bn = -2*imag(z(2:n));
```

Plot the interpolating polynomial over the original data points.

```
hold on
px = 0:0.01:360;
k = 1:length(an);
py = a0 + an*cos(2*pi*k'*px/360) ...
        + bn*sin(2*pi*k'*px/360) ...
        + a6*cos(2*pi*6*px/360);

plot(px,py)
```



References

[1] Briggs, W. and V.E. Henson. *The DFT: An Owner's Manual for the Discrete Fourier Transform*. Philadelphia: SIAM, 1995.

[2] Gauss, C. F. “Theoria interpolationis methodo nova tractata.” *Carl Friedrich Gauss Werke*. Band 3. Göttingen: Königlichen Gesellschaft der Wissenschaften, 1866.

[3] Heideman M., D. Johnson, and C. Burrus. “Gauss and the History of the Fast Fourier Transform.” *Arch. Hist. Exact Sciences*. Vol. 34. 1985, pp. 265–277.

[4] Goldstine, H. H. *A History of Numerical Analysis from the 16th through the 19th Century*. Berlin: Springer-Verlag, 1977.

See Also

`fft`

Related Examples

- “Fourier Transforms” on page 16-2

2-D Fourier Transforms

The fft2 function transforms 2-D data into frequency space. For example, you can transform a 2-D optical mask to reveal its diffraction pattern.

Two-Dimensional Fourier Transform

The following formula defines the discrete Fourier transform Y of an m-by-n matrix X.

$$Y_{p+1,q+1} = \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} \omega_m^{jp} \omega_n^{kq} X_{j+1,k+1}$$

ω_m and ω_n are complex roots of unity defined by the following equations.

$$\omega_m = e^{-2\pi i/m}$$

$$\omega_n = e^{-2\pi i/n}$$

i is the imaginary unit, p and j are indices that run from 0 to m–1, and q and k are indices that run from 0 to n–1. The indices for X and Y are shifted by 1 in this formula to reflect matrix indices in MATLAB.

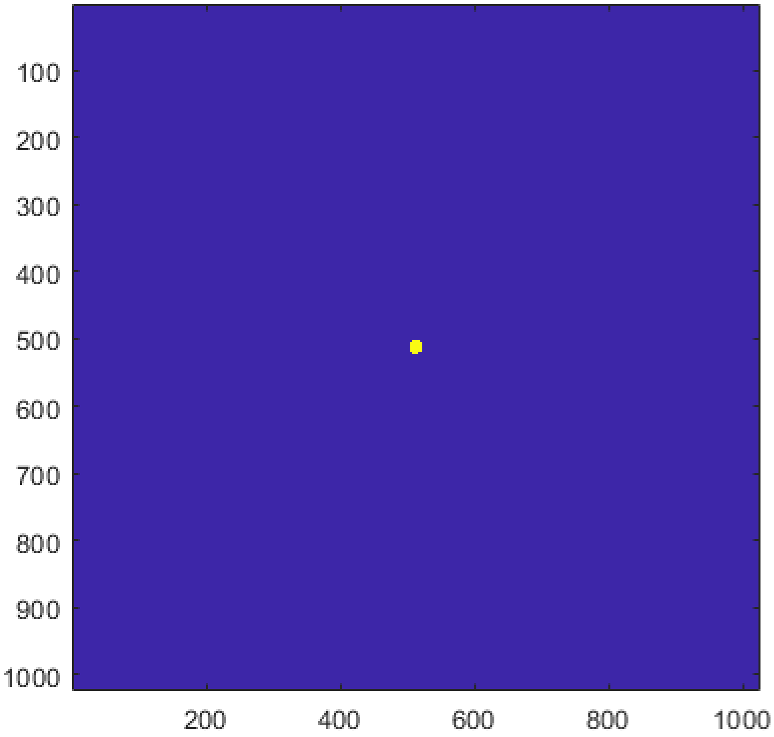
Computing the 2-D Fourier transform of X is equivalent to first computing the 1-D transform of each column of X, and then taking the 1-D transform of each row of the result. In other words, the command fft2(X) is equivalent to Y = fft(fft(X).').'.

2-D Diffraction Pattern

In optics, the Fourier transform can be used to describe the diffraction pattern produced by a plane wave incident on an optical mask with a small aperture [1]. This example uses the fft2 function on an optical mask to compute its diffraction pattern.

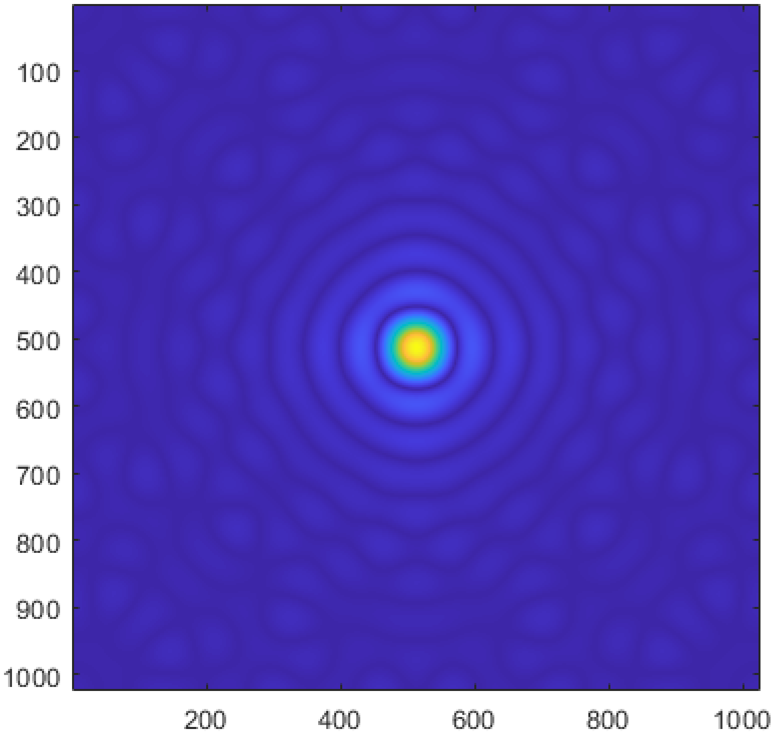
Create a logical array that defines an optical mask with a small, circular aperture.

```
n = 2^10;                 % size of mask
M = zeros(n);
I = 1:n;
x = I-n/2;                % mask x-coordinates
y = n/2-I;                % mask y-coordinates
[X,Y] = meshgrid(x,y);    % create 2-D mask grid
R = 10;                   % aperture radius
A = (X.^2 + Y.^2 <= R^2); % circular aperture of radius R
M(A) = 1;                 % set mask elements inside aperture to 1
imagesc(M)                % plot mask
axis image
```



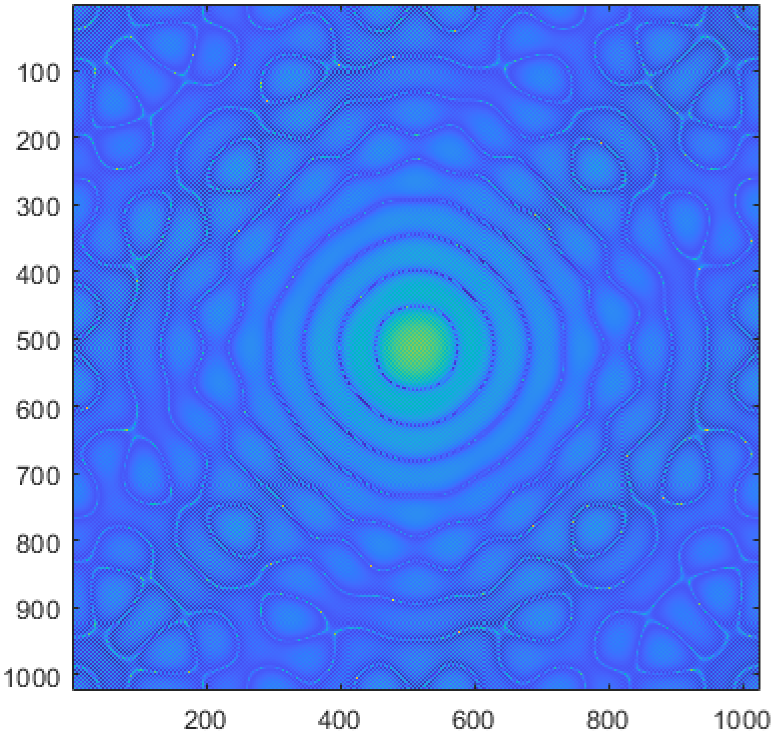
Use `fft2` to compute the 2-D Fourier transform of the mask, and use the `fftshift` function to rearrange the output so that the zero-frequency component is at the center. Plot the resulting diffraction pattern frequencies. Blue indicates small amplitudes and yellow indicates large amplitudes.

```
DP = fftshift(fft2(M));
imagesc(abs(DP))
axis image
```



To enhance the details of regions with small amplitudes, plot the 2-D logarithm of the diffraction pattern. Very small amplitudes are affected by numerical round-off error, and the rectangular grid causes radial asymmetry.

```
imagesc(abs(log2(DP)))
axis image
```



References

[1] Fowles, G. R. *Introduction to Modern Optics*. New York: Dover, 1989.

See Also

`fft` | `fft2` | `fftn` | `fftshift` | `ifft2`

Related Examples

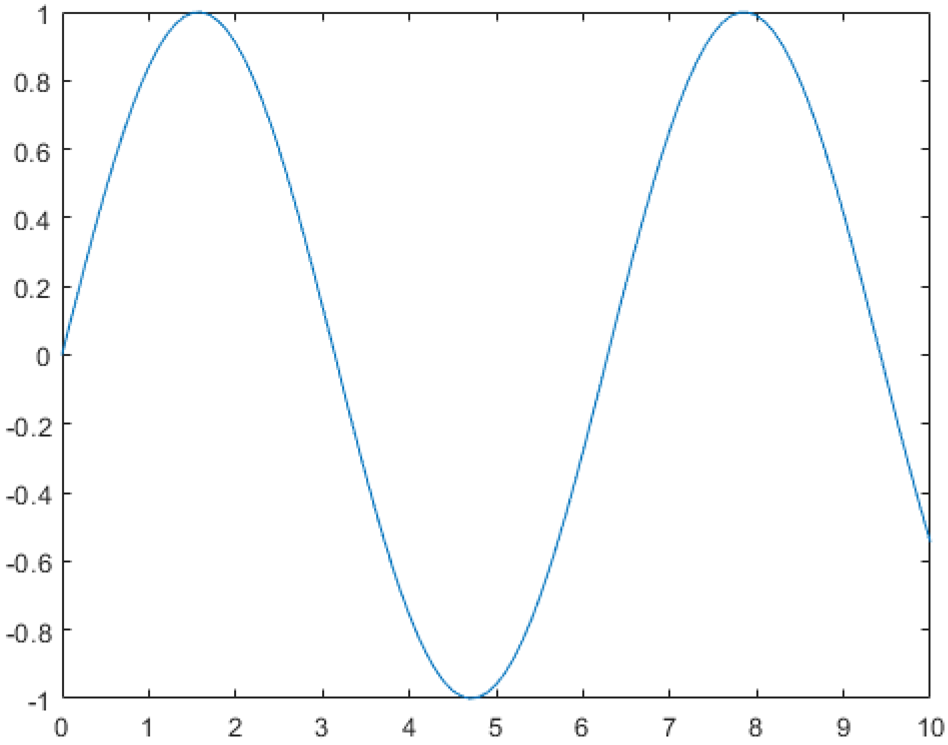
- “Fourier Transforms” on page 16-2

Square Wave from Sine Waves

This example shows how the Fourier series expansion for a square wave is made up of a sum of odd harmonics.

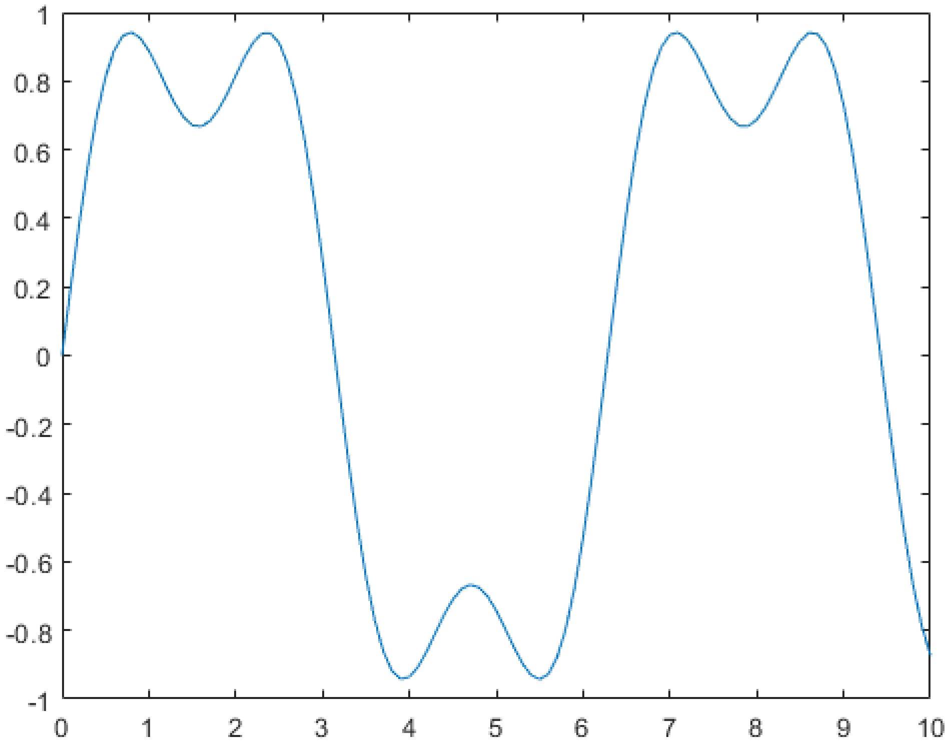
Start by forming a time vector running from 0 to 10 in steps of 0.1, and take the sine of all the points. Plot this fundamental frequency.

```
t = 0:.1:10;
y = sin(t);
plot(t,y);
```



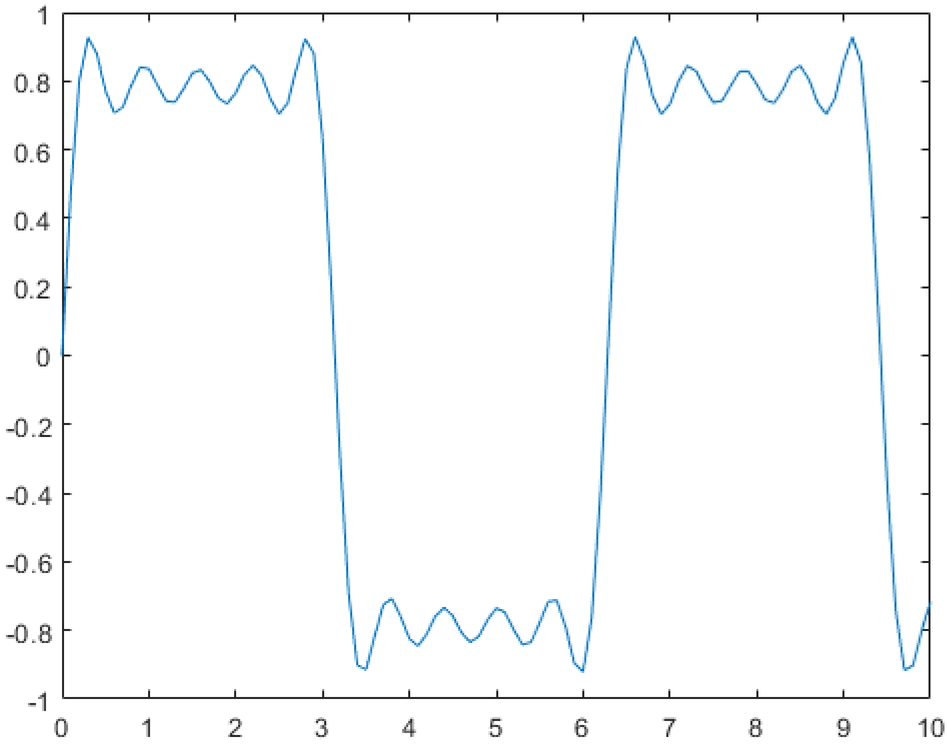
Next add the third harmonic to the fundamental, and plot it.

```
y = sin(t) + sin(3*t)/3;
plot(t,y);
```



Now use the first, third, fifth, seventh, and ninth harmonics.

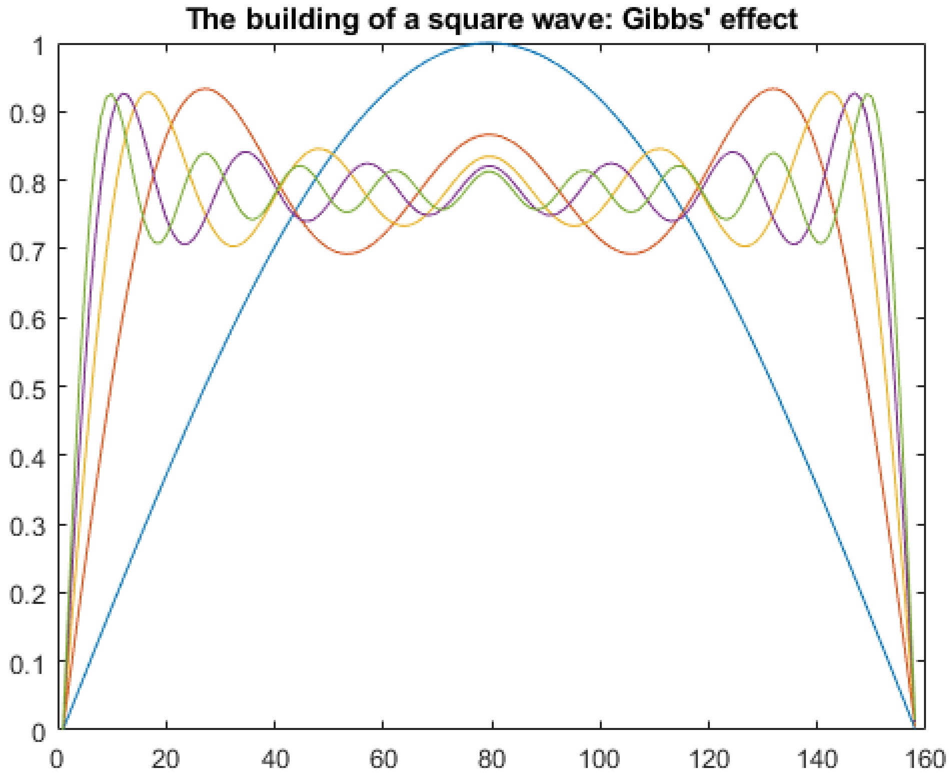
```
y = sin(t) + sin(3*t)/3 + sin(5*t)/5 + sin(7*t)/7 + sin(9*t)/9;
plot(t,y);
```



For a finale, go from the fundamental all the way to the 19th harmonic, creating vectors of successively more harmonics, and saving all intermediate steps as the rows of a matrix.

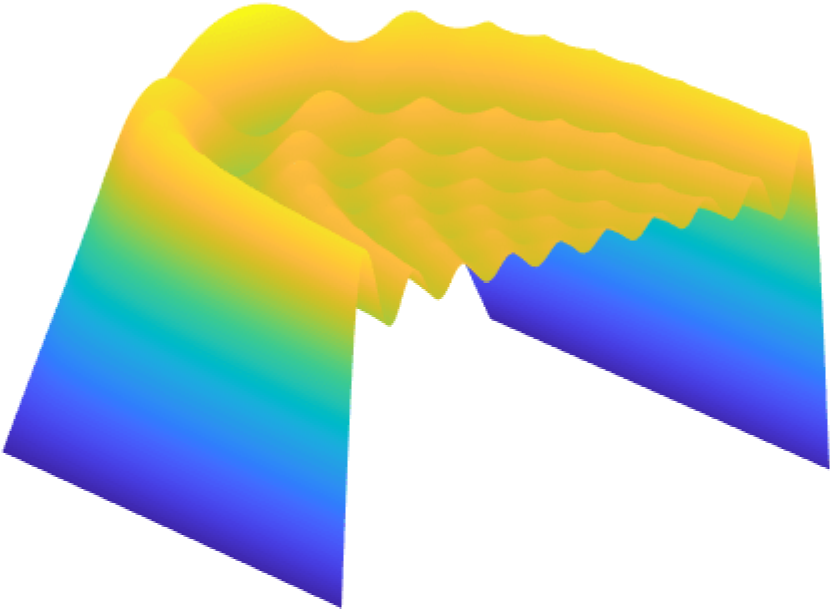
Plot the vectors on the same figure to show the evolution of the square wave. Note that the Gibbs effect says it will never quite get there.

```
t = 0:.02:3.14;
y = zeros(10,length(t));
x = zeros(size(t));
for k = 1:2:19
   x = x + sin(k*t)/k;
   y((k+1)/2,:) = x;
end
plot(y(1:2:9,:)')
title('The building of a square wave: Gibbs'' effect')
```



Here is a 3-D surface representing the gradual transformation of a sine wave into a square wave.

```
surf(y);
shading interp
axis off ij
```

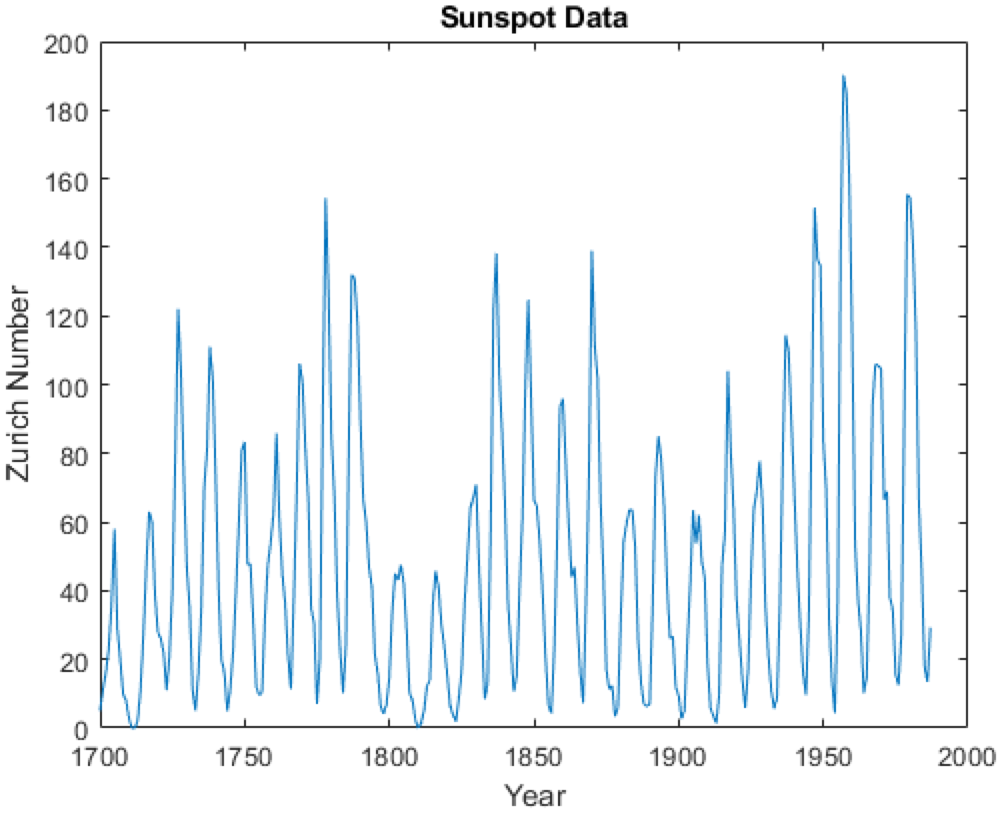


Analyzing Cyclical Data with FFT

You can use the Fourier transform to analyze variations in data, such as an event in nature over a period time.

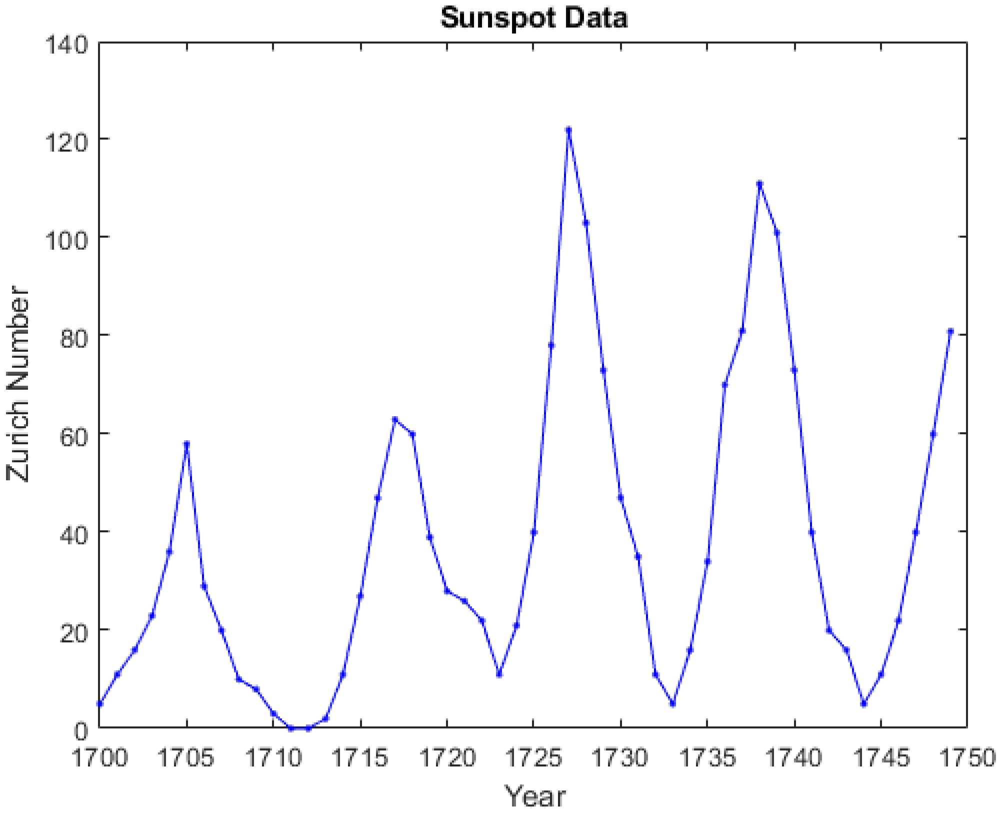
For almost 300 years, astronomers have tabulated the number and size of sunspots using the Zurich sunspot relative number. Plot the Zurich number over approximately the years 1700 to 2000.

```
load sunspot.dat
year = sunspot(:,1);
relNums = sunspot(:,2);
plot(year,relNums)
xlabel('Year')
ylabel('Zurich Number')
title('Sunspot Data')
```



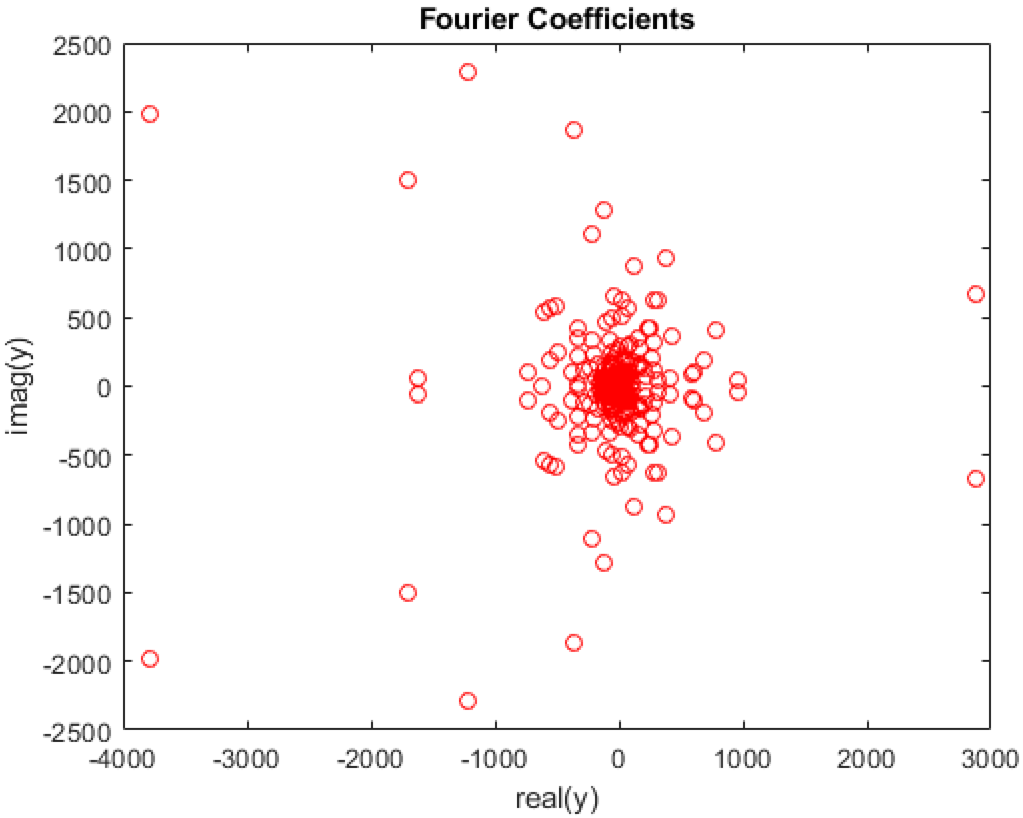
To take a closer look at the cyclical nature of sunspot activity, plot the first 50 years of data.

```
plot(year(1:50),relNums(1:50),'b.-');
xlabel('Year')
ylabel('Zurich Number')
title('Sunspot Data')
```



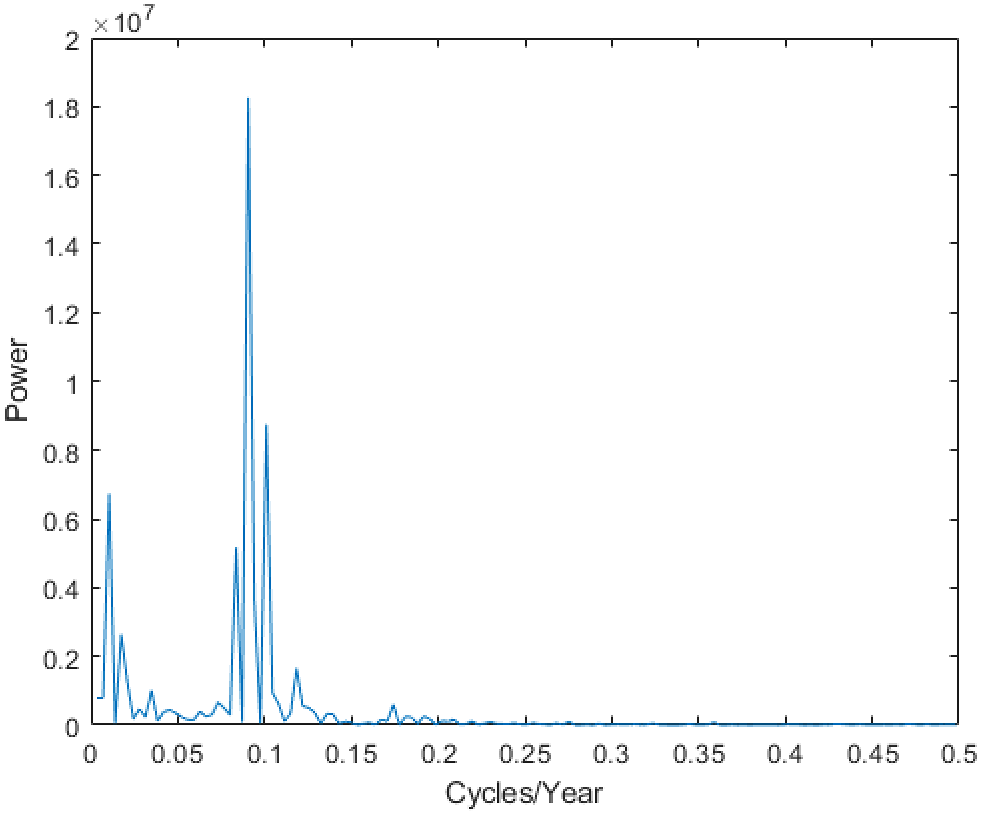
The Fourier transform is a fundamental tool in signal processing that identifies frequency components in data. Using the `fft` function, take the Fourier transform of the Zurich data. Remove the first element of the output, which stores the sum of the data. Plot the remainder of the output, which contains a mirror image of complex Fourier coefficients about the real axis.

```
y = fft(relNums);
y(1) = [];
plot(y,'ro')
xlabel('real(y)')
ylabel('imag(y)')
title('Fourier Coefficients')
```



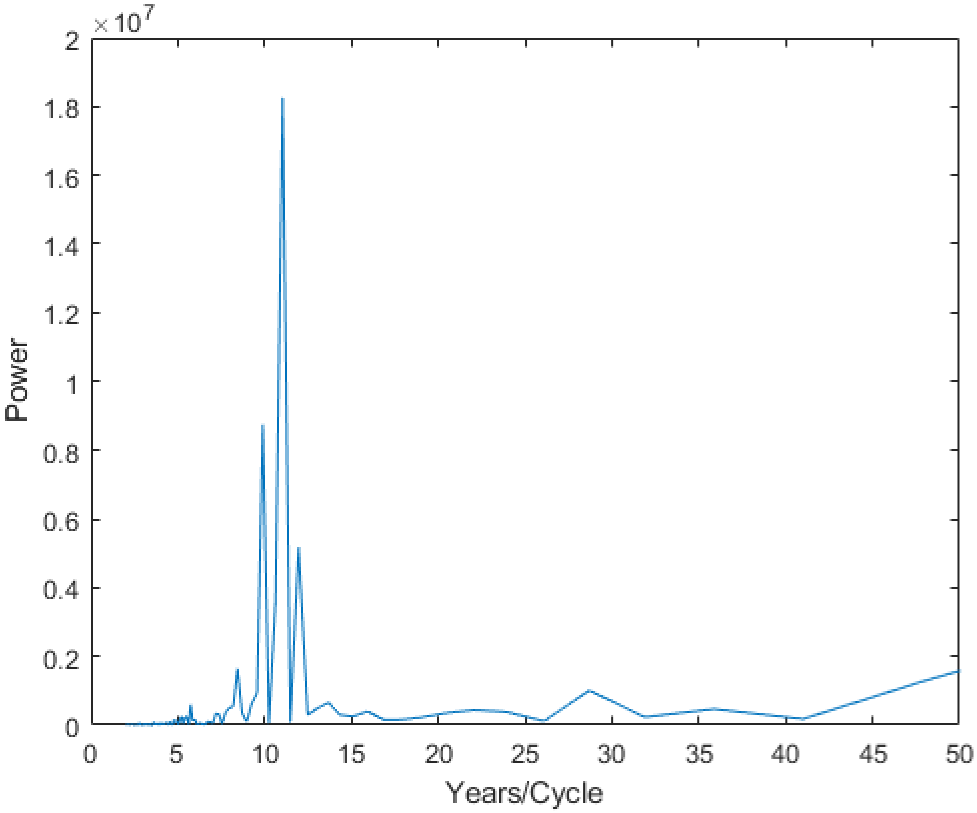
Fourier coefficients on their own are difficult to interpret. A more meaningful measure of the coefficients is their magnitude squared, which is a measure of power. Since half of the coefficients are repeated in magnitude, you only need to compute the power on one half of the coefficients. Plot the power spectrum as a function of frequency, measured in cycles per year.

```
n = length(y);
power = abs(y(1:floor(n/2))).^2; % power of first half of transform data
maxfreq = 1/2;                   % maximum frequency
freq = (1:n/2)/(n/2)*maxfreq;    % equally spaced frequency grid
plot(freq,power)
xlabel('Cycles/Year')
ylabel('Power')
```



Maximum sunspot activity happens less frequently than once per year. For a view of the cyclical activity that is easier to interpret, plot power as a function of period, measured in years per cycle. The plot reveals that sunspot activity peaks about once every 11 years.

```
period = 1./freq;
plot(period,power);
xlim([0 50]); %zoom in on max power
xlabel('Years/Cycle')
ylabel('Power')
```



See Also

`fft` | `fft2` | `fftw`

Related Examples

- “Fourier Transforms” on page 16-2
- “2-D Fourier Transforms” on page 16-17