MATLAB®

Mathematics

MATLAB®

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.
3 Apple Hill Drive
Natick, MA 01760-2098

Contents

Matrices and Arrays

1

Linear Algebra

2

3

Sparse Matrices

4

Graph and Network Algorithms

5

Functions of One Variable

6

<div align="right">

Computational Geometry

</div>

7

8

9

Function Handles

10

Ordinary Differential Equations (ODEs)

11

Boundary Value Problems (BVPs)

12

Partial Differential Equations (PDEs)

13

14

Delay Differential Equations (DDEs)

15

Numerical Integration

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, 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 1-26
- "Reshaping and Rearranging Arrays" on page 1-11

Removing Rows or Columns from a Matrix

The easiest way to remove a row or column of a matrix is setting 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 1-11
- "Array Indexing" on page 1-26

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 1-9
- "Array Indexing" on page 1-26

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.

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.
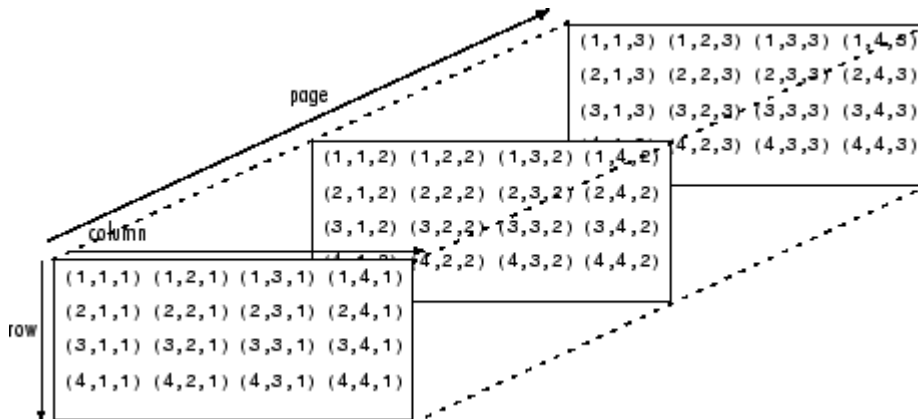
Creating Multidimensional Arrays

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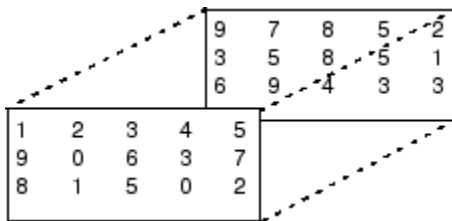
Reshaping a multidimensional array can be useful for performing certain operations or visualizing the data. Use the `reshape` function to rearrange the elements of the 3-D array into a 6-by-5 matrix.

```
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 1-26
- "Reshaping and Rearranging Arrays" on page 1-11

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 an element of a `datetime` vector.

```
t = datetime(2018,1:5,1)
```

t = 1x5 datetime array
```
01-Jan-2018    01-Feb-2018    01-Mar-2018    01-Apr-2018    01-May-2018
```

```
march1 = t(3)
```

march1 = datetime
```
01-Mar-2018
```

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 1-17.

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 array
    "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 a Table"
- "Access Data in a Structure Array"
- "Access Data in Cell Array"

Getting Information About a Matrix

In this section...
"Dimensions of the Matrix" on page 1-32
"Classes Used in the Matrix" on page 1-33
"Data Structures Used in the Matrix" on page 1-34

Dimensions of the Matrix

These functions return information about the shape and size of a matrix.

Function	Description
length	Return the length of the longest dimension. (The length of a matrix or array with any zero dimension is zero.)
ndims	Return the number of dimensions.
numel	Return the number of elements.
size	Return the length of each dimension.

The following examples show some simple ways to use these functions. Both use the 3-by-5 matrix A shown here:

```
A = 10*gallery('uniformdata',[5],0);
A(4:5, :) = []
A =
    9.5013    7.6210    6.1543    4.0571    0.5789
    2.3114    4.5647    7.9194    9.3547    3.5287
    6.0684    0.1850    9.2181    9.1690    8.1317
```

Example Using numel

Using the numel function, find the average of all values in matrix A:

```
sum(A(:))/numel(A)
ans =
    5.8909
```

Example Using ndims, numel, and size

Using ndims and size, go through the matrix and find those values that are between 5 and 7, inclusive:

```
if ndims(A) ~= 2
   return
end

[rows cols] = size(A);
for m = 1:rows
   for n = 1:cols
      x = A(m, n);
      if x >= 5 && x <= 7
         disp(sprintf('A(%d, %d) = %5.2f', m, n, A(m,n)))
      end
   end
end
```

The code returns the following:

```
A(1, 3) =  6.15
A(3, 1) =  6.07
```

Classes Used in the Matrix

These functions test elements of a matrix for a specific data type.

Function	Description
isa	Detect if input is of a given data type.
iscell	Determine if input is a cell array.
iscellstr	Determine if input is a cell array of strings.
ischar	Determine if input is a character array.
isfloat	Determine if input is a floating-point array.
isinteger	Determine if input is an integer array.
islogical	Determine if input is a logical array.
isnumeric	Determine if input is a numeric array.
isreal	Determine if input is an array of real numbers.

Function	Description
isstruct	Determine if input is a MATLAB structure array.

Example Using isnumeric and isreal

Pick out the real numeric elements from this vector:

```
A = [5+7i 8/7 4.23 39j pi 9-2i];

for m = 1:numel(A)
   if isnumeric(A(m)) && isreal(A(m))
      disp(A(m))
   end
end
```

The values returned are

```
    1.1429
    4.2300
    3.1416
```

Data Structures Used in the Matrix

These functions test elements of a matrix for a specific data structure.

Function	Description
isempty	Determine if input has any dimension with size zero.
isscalar	Determine if input is a 1-by-1 matrix.
issparse	Determine if input is a sparse matrix.
isvector	Determine if input is a 1-by-n or n-by-1 matrix.

Summary of Matrix and Array Functions

This section summarizes the principal functions used in creating and handling matrices. Most of these functions work on multidimensional arrays as well.

Functions to Create a Matrix

Function	Description
[a,b] or [a;b]	Create a matrix from specified elements, or concatenate matrices together.
accumarray	Construct a matrix using accumulation.
blkdiag	Construct a block diagonal matrix.
cat	Concatenate matrices along the specified dimension.
diag	Create a diagonal matrix from a vector.
horzcat	Concatenate matrices horizontally.
magic	Create a square matrix with rows, columns, and diagonals that add up to the same number.
ones	Create a matrix of all ones.
rand	Create a matrix of uniformly distributed random numbers.
repmat	Create a new matrix by replicating or tiling another.
vertcat	Concatenate two or more matrices vertically.
zeros	Create a matrix of all zeros.

Functions to Modify the Shape of a Matrix

Function	Description
ctranspose	Flip a matrix about the main diagonal and replace each element with its complex conjugate.
flip	Flip a matrix along the specified dimension.
fliplr	Flip a matrix about a vertical axis.
flipud	Flip a matrix about a horizontal axis.
reshape	Change the dimensions of a matrix.
rot90	Rotate a matrix by 90 degrees.
transpose	Flip a matrix about the main diagonal.

Functions to Find the Structure or Shape of a Matrix

Function	Description
isempty	Return true for 0-by-0 or 0-by-n matrices.
isscalar	Return true for 1-by-1 matrices.
issparse	Return true for sparse matrices.
isvector	Return true for 1-by-n matrices.
length	Return the length of a vector.
ndims	Return the number of dimensions in a matrix.
numel	Return the number of elements in a matrix.
size	Return the size of each dimension.

Functions to Determine Class

Function	Description
iscell	Return `true` if the matrix is a cell array.
ischar	Return `true` if matrix elements are character arrays.
isfloat	Determine if input is a floating point array.
isinteger	Determine if input is an integer array.
islogical	Return `true` if matrix elements are logicals.
isnumeric	Return `true` if matrix elements are numeric.
isreal	Return `true` if matrix elements are real numbers.
isstruct	Return `true` if matrix elements are MATLAB structures.

Functions to Sort and Shift Matrix Elements

Function	Description
circshift	Circularly shift matrix contents.
issorted	Return `true` if the matrix elements are sorted.
sort	Sort elements in ascending or descending order.
sortrows	Sort rows in ascending order.

Functions That Work on Diagonals of a Matrix

Function	Description
blkdiag	Construct a block diagonal matrix.
diag	Return the diagonals of a matrix.
trace	Compute the sum of the elements on the main diagonal.
tril	Return the lower triangular part of a matrix.
triu	Return the upper triangular part of a matrix.

Functions to Change the Indexing Style

Function	Description
ind2sub	Convert a linear index to a row-column index.
sub2ind	Convert a row-column index to a linear index.

Functions for Working with Multidimensional Arrays

Function	Description
cat	Concatenate arrays.
circshift	Shift array circularly.
ipermute	Inverse permute array dimensions.
ndgrid	Generate arrays for n-dimensional functions and interpolation.
ndims	Return the number of array dimensions.
permute	Permute array dimensions.
shiftdim	Shift array dimensions.
squeeze	Remove singleton dimensions.

2

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

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

$$7x \qquad\qquad\qquad = \qquad\qquad\qquad 21$$

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:

$$x \qquad = \qquad 21/7 \qquad = \qquad 3.$$

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

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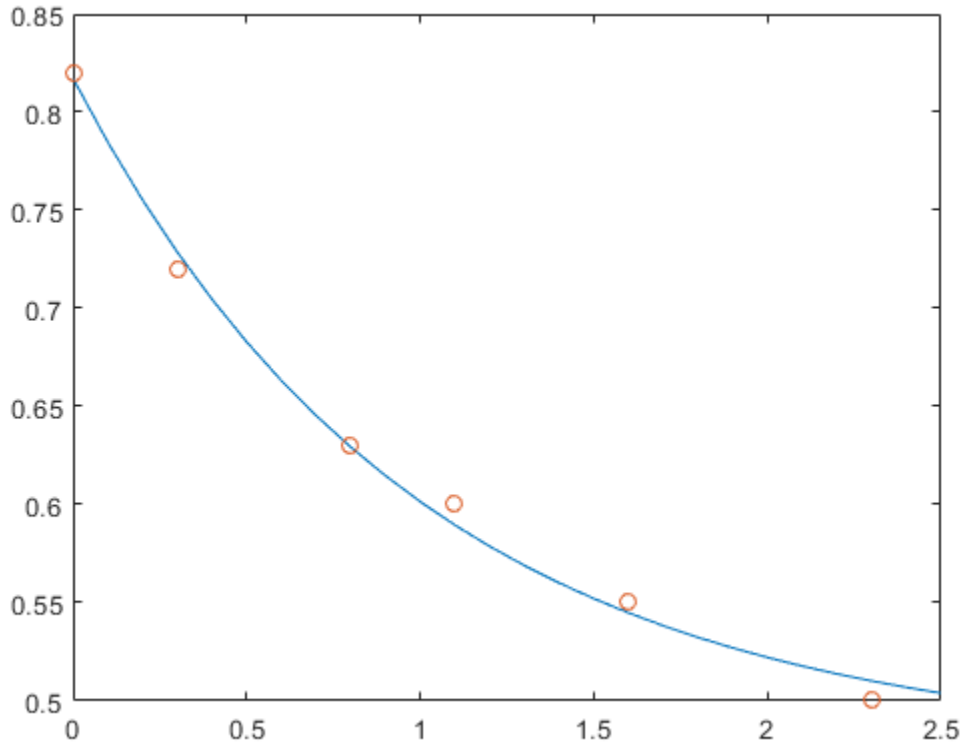
$E*c$ is not exactly equal to y, but the difference might well be less than measurement errors in the original data.

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:

$$Zq = \begin{pmatrix} \bar{x}_1 & \bar{x}_2 \end{pmatrix} \begin{pmatrix} u \\ w \end{pmatrix} = u\bar{x}_1 + w\bar{x}_2 \ .$$

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

Function	Description
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

$$Q'Q = 1.$$

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

In this section...
"Positive Integer Powers" on page 2-34
"Inverse and Fractional Powers" on page 2-34
"Element-by-Element Powers" on page 2-35
"Exponentials" on page 2-35

Positive Integer Powers

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

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

A =

     1     1     1
     1     2     3
     1     3     6

X = A^2

X =
     3     6    10
     6    14    25
    10    25    46
```

Inverse and Fractional Powers

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

```
Y = A^(-3)

Y =

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

Fractional powers, like `A^(2/3)`, are also permitted; the results depend upon the distribution of the eigenvalues of the matrix.

Element-by-Element Powers

The `.^` operator produces element-by-element powers. For example:

```
X = A.^2

A =
     1     1     1
     1     4     9
     1     9    36
```

Exponentials

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.

A system of linear, constant coefficient, ordinary differential equations can be written

$$dx/dt = Ax,$$

where $x = x(t)$ is a vector of functions of t and A is a matrix independent of t. The solution can be expressed in terms of the matrix exponential

$$x(t) = e^{tA}x(0).$$

The function `expm(A)` computes the matrix exponential. An example is provided by the 3-by-3 coefficient matrix,

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

A = 3×3

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

and the initial condition, $x(0)$.

```
x0 = [1 1 1]'
```

x0 = *3×1*

```
     1
     1
     1
```

The matrix exponential is used to compute the solution, $x(t)$, to the differential equation at 101 points on the interval $0 \leq t \leq 1$.

```
X = [];
for t = 0:.01:1
    X = [X expm(t*A)*x0];
end
```

A three-dimensional phase plane plot shows the solution spiraling in towards the origin. This behavior is related to the eigenvalues of the coefficient matrix.

```
plot3(X(1,:),X(2,:),X(3,:),'-o')
```
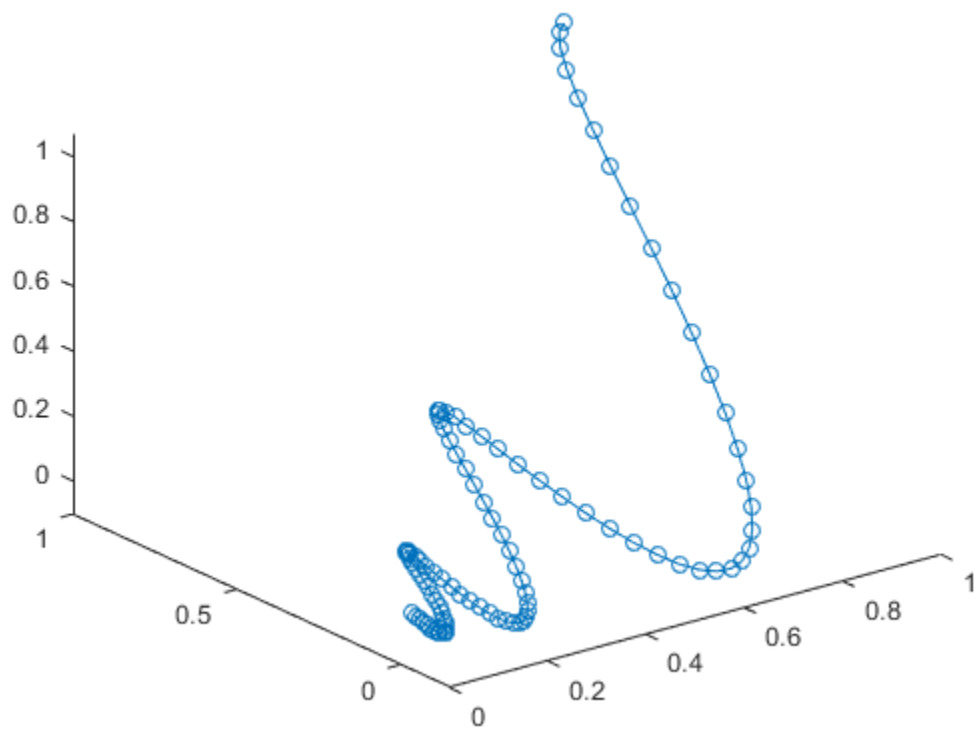
Eigenvalues

Eigenvalue Decomposition

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

$$Av = \lambda v.$$

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 this ordinary differential equation on page 2-35:

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

The statement

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

produces a column vector containing the eigenvalues. 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

$$A \qquad = \qquad USU' \qquad ,$$

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 on the diagonal of 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 an m-by-m U, an m-by-n Σ, and an n-by-n 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:

(m > n)
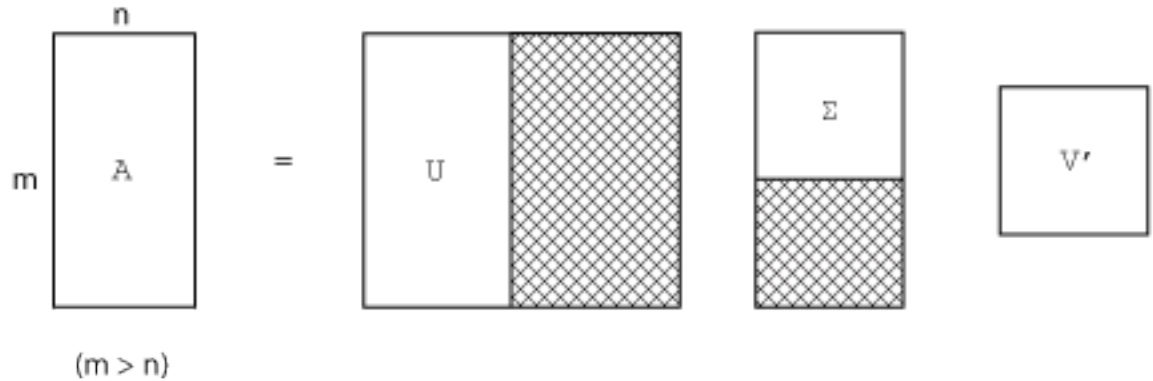
The eigenvalue decomposition is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space into itself, as it does for an ordinary differential equation. However, the singular value decomposition is the appropriate tool for analyzing a mapping from one vector space into 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.

If the matrix A is large and sparse, then using svd to calculate *all* of the singular values and 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 a lot of extra work. In cases where only a subset of the singular values and vectors are required, the `svds` function is preferred over `svd`.

For 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`. However, for truly large and sparse matrices, using `svds` 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.

Another recommended resource is the Pseudospectra Gateway website.

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.* Balitmore, 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*
10⁶ ×

```
   -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
```

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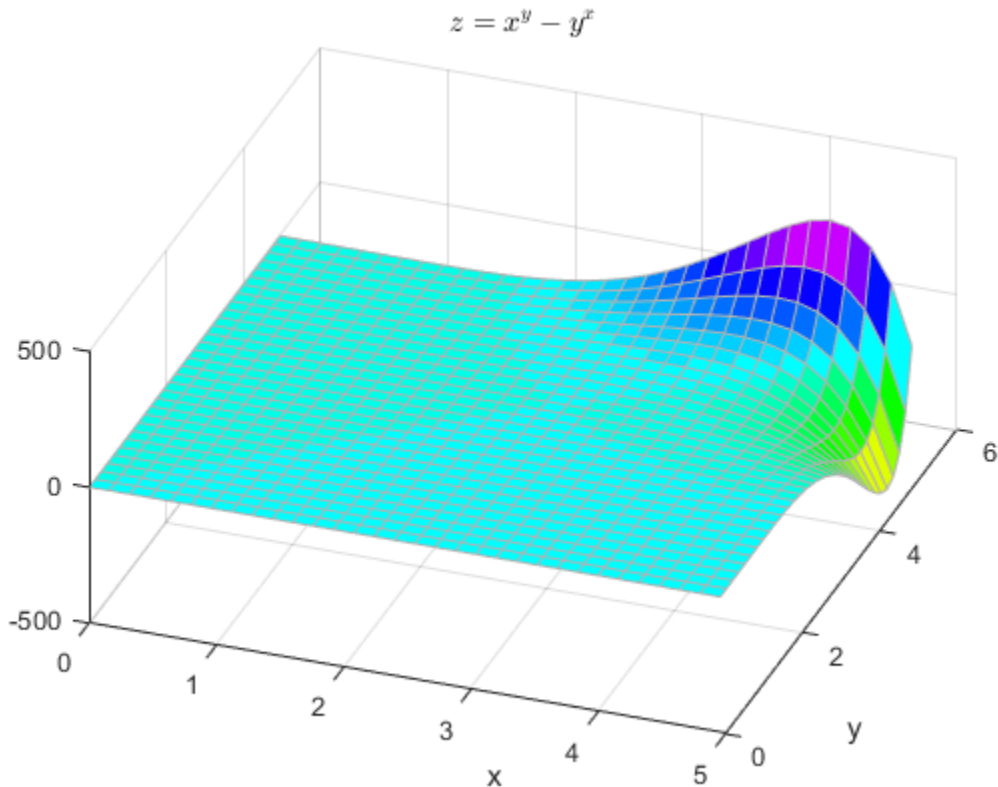
The solution of the equation $x^y - y^x = 0$ has a very interesting shape, and our original question is not easily solved by inspection. Here is a plot of the xy values that yield $z = 0$.

```
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');
```

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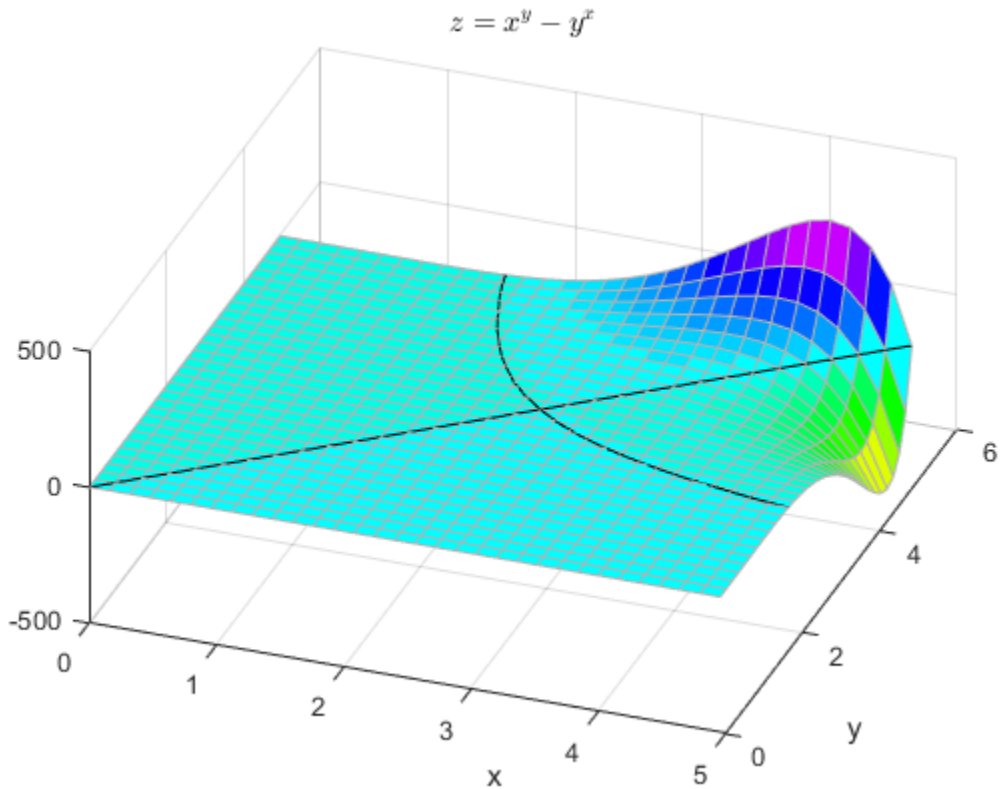
Some combinations of *x* and *y* along the black curve are integers. This next plot is of the integer solutions to the equation $x^y - y^x = 0$. Notice that $2^4 = 4^2$ is the *only* integer solution where $x \neq y$.

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

$$z = x^y - y^x$$
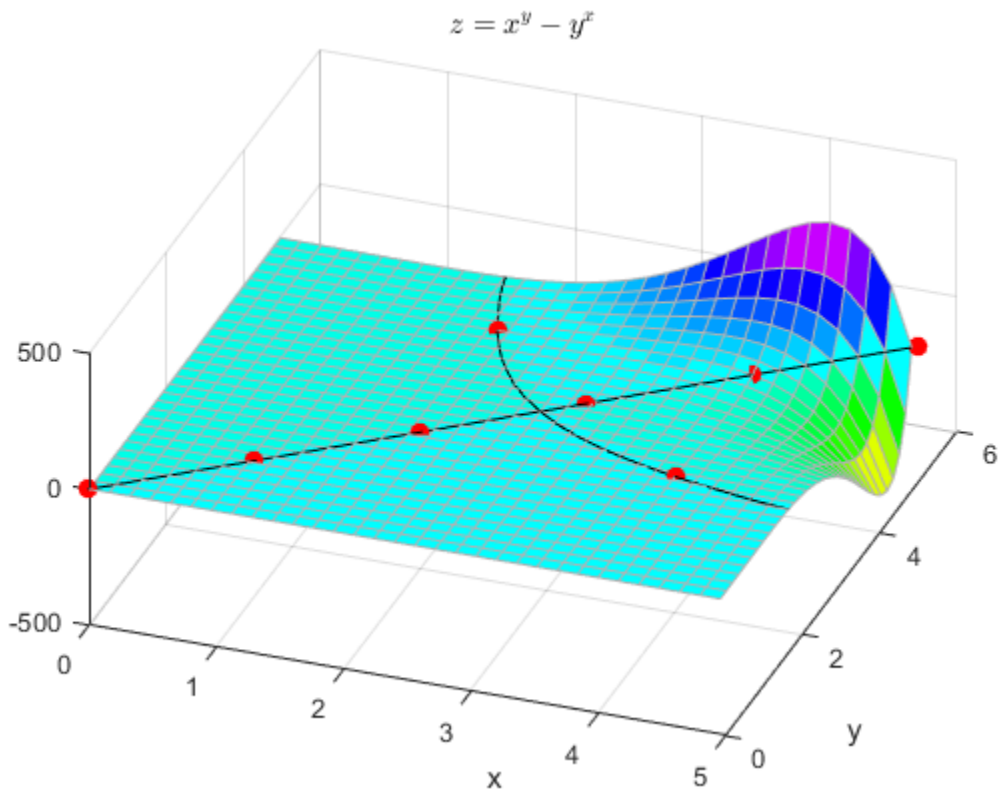
Finally, plot the points (π, e) and (e, π) on the surface. The result shows that e^π is indeed larger than π^e (though not by much).

```
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;
```

$$z = x^y - y^x$$
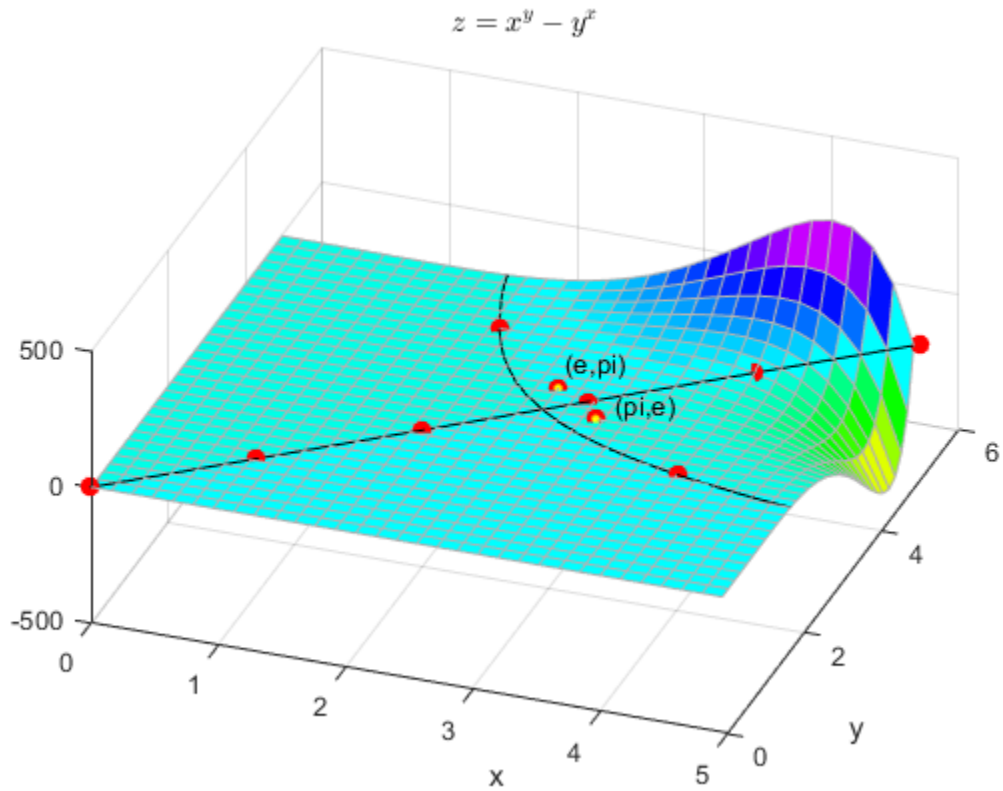
Verify the results.

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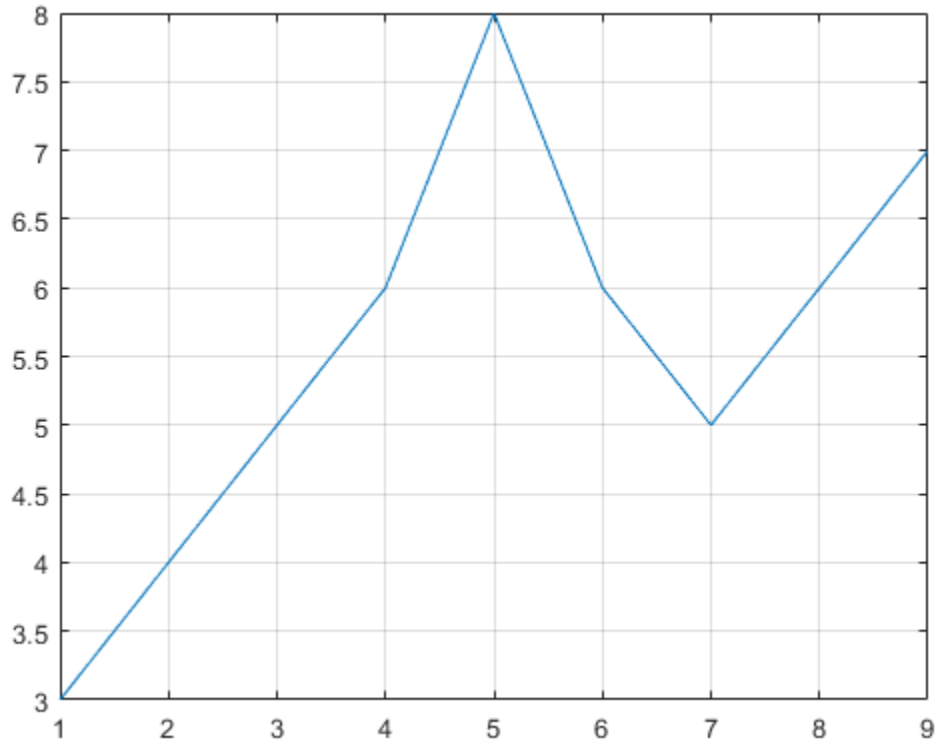
MATLAB can make other graph types as well, with axis labels.

```
bar(b)
xlabel('Sample #')
ylabel('Pounds')
```
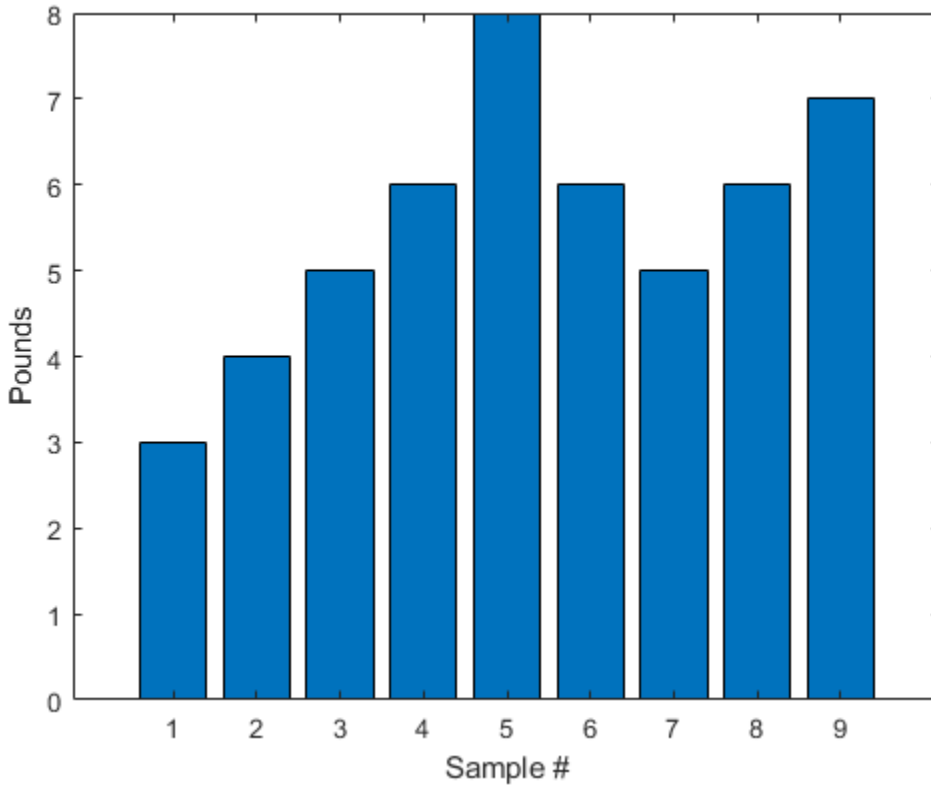
MATLAB can use symbols in plots as well. Here is an example using stars to mark the points. MATLAB offers a variety of other symbols and line types.

```
plot(b,'*')
axis([0 10 0 10])
```
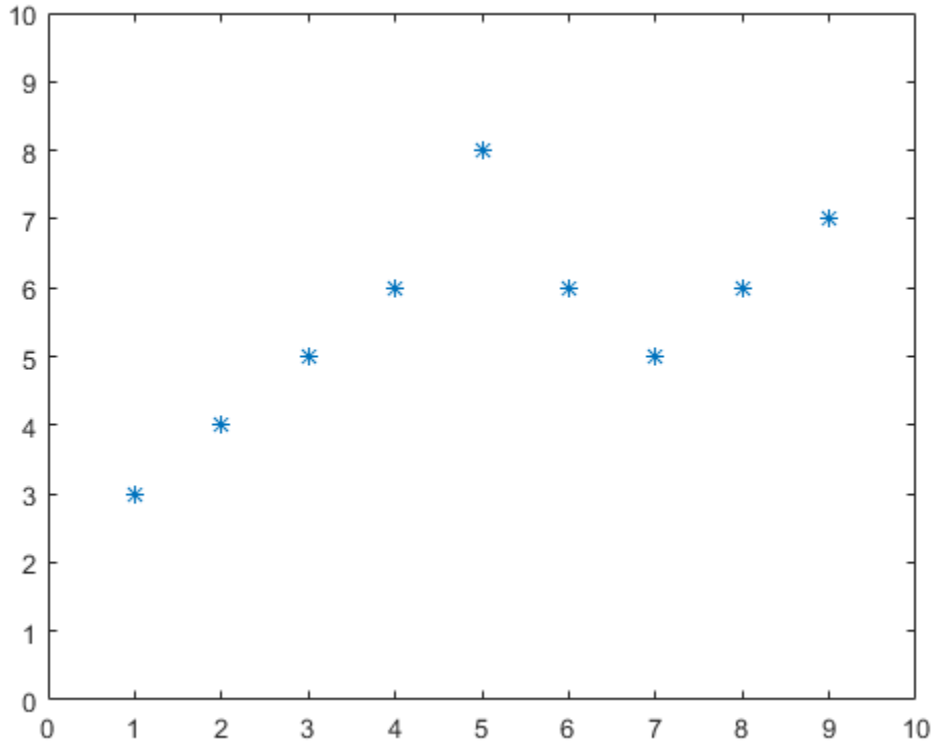
One area in which MATLAB excels is matrix computation.

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);
```
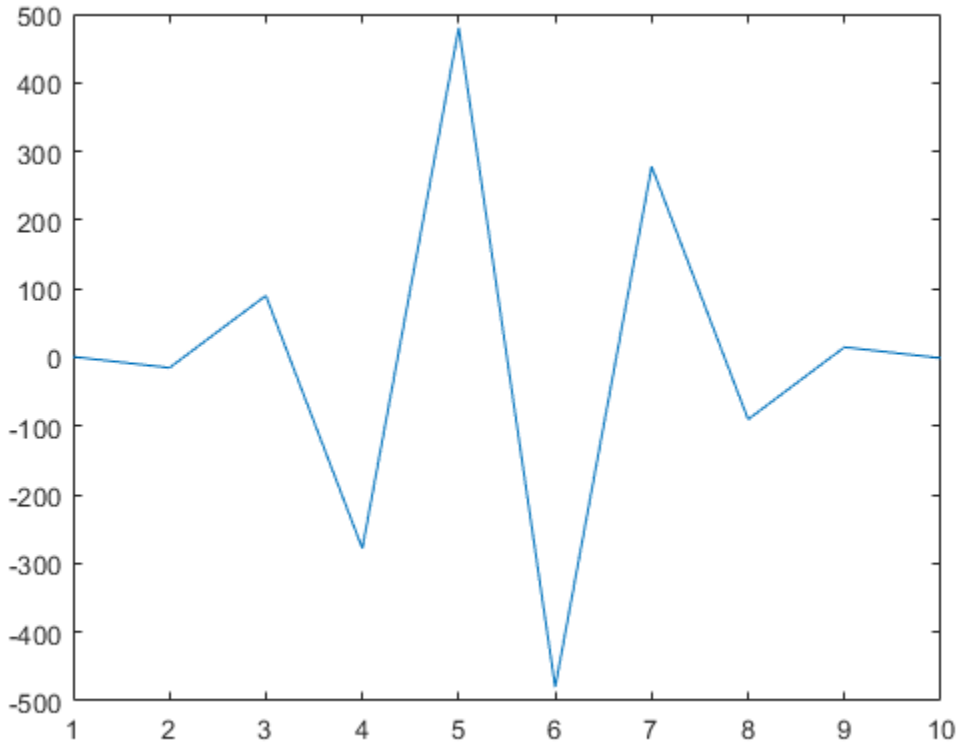

At any time, we can get a listing of the variables we have stored in memory using the who or whos command.

```
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"

3

Random Numbers

Random Numbers in MATLAB

When you create random numbers using software, the results are not random in a strict, mathematical sense. However, software applications, such as MATLAB, use algorithms that make your results appear to be random and independent. The results also pass various statistical tests of randomness and independence. These apparently random and independent numbers are often described as pseudorandom and pseudoindependent. You can use these numbers as if they are truly random and independent. One benefit of using pseudorandom, pseudoindependent numbers is that you can repeat a random number calculation at any time. This approach can be useful in testing or diagnostic situations.

Although repeatability can be useful, it is possible to repeat your results accidentally when you really want different results. There are several ways to avoid this problem. The documentation contains several examples that show how to ensure that your results are different when that is your intention.

Note You can assume any reference to random numbers in the MATLAB documentation is actually referring to pseudorandom numbers, unless otherwise stated.

See Also

Related Examples

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

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,

```
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 1,2,...,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 arrays of random integer values that have no repeated values. For example,

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

`r4` is a 1-by-5 array containing randomly selected integer values on the closed interval, [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.

See Also

rand | randi | randn | randperm

Related Examples

- "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

More About

- "Random Numbers in MATLAB" on page 3-2

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.

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.

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

r_range = *1×2*

```
   -10     10
```

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.

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, $(0, 2\pi)$.
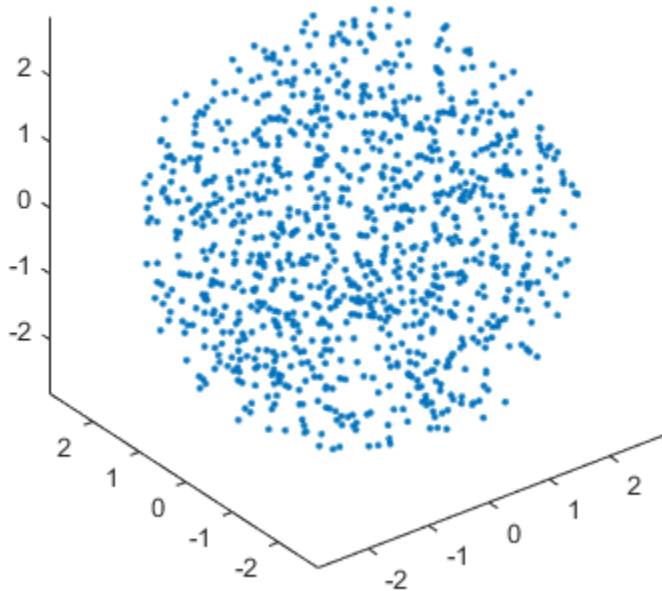
```
azimuth = 2*pi*rand(1000,1);
```

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.

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.

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

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

Save the generator settings in a structure s.

```
s = rng;
```

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
```

Repeat the same command.

```
A = randi(10,3,3)
```

A = 3×3

```
     6      3      7
     5      9      5
     7      1      6
```

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

Now, return the generator to the original state stored in `s` and reproduce the first array `A`.

```
rng(s);
A = randi(10,3,3)
```

A = 3×3

```
     5      4      2
     8      2      4
     1      1      4
```

Unlike reseeding, which reinitializes the generator, this approach allows you to save and restore the generator settings at any point.

See Also

`rng`

Related Examples

- "Generate Random Numbers That Are Different" on page 3-15

More About

- "Random Numbers in MATLAB" on page 3-2

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 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.

Alternatively, specify different seeds explicitly. For example,

```
rng(1);
A = rand(2,2);
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.

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.

See Also

rng

Related Examples

- "Generate Random Numbers That Are Repeatable" on page 3-11

More About

- "Random Numbers in MATLAB" on page 3-2
- "Startup Options in MATLAB Startup File"

Managing the Global Stream

`rand`, `randn`, and `randi` draw random numbers from an underlying random number stream, called the global stream. The `rng` function provides a simple way to control the global stream. For more comprehensive control, the `RandStream` class allows you to get a handle to the global stream and control random number generation.

Get a handle to the global stream as follows:

```
globalStream = RandStream.getGlobalStream
globalStream =

mt19937ar random stream (current global stream)
             Seed: 0
  NormalTransform: Ziggurat
```

Return the properties of the stream with the `get` method:

```
get(globalStream)
              Type: 'mt19937ar'
        NumStreams: 1
       StreamIndex: 1
         Substream: 1
              Seed: 0
             State: [625x1 uint32]
   NormalTransform: 'Ziggurat'
         Antithetic: 0
      FullPrecision: 1
```

Now, use the `rand` function to generate uniform random values from the global stream.

```
rand(1,5);
```

Use the `randn` and `randi` functions to generate normal random values and integer random values from the global stream.

```
A = randi(100,1,5);
A = randn(1,5);
```

The `State` property is the internal state of the generator. You can save the `State` of `globalStream`.

```
myState = globalStream.State;
```

Using `myState`, you can restore the state of `globalStream` and reproduce previous results.

```
myState = globalStream.State;
A = rand(1,100);
globalStream.State = myState;
B=rand(1,100);
isequal(A,B)

ans =

     1
```

`rand`, `randi`, and `randn` 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;
randi(100);
B = rand(1,100);
isequal(A,B)

ans =

     0
```

The global stream is a handle object of the `RandStream` class. `RandStream.getGlobalStream` returns a handle. The properties of the global stream can be viewed or modified from any handle to the stream.

```
stream1=RandStream.getGlobalStream;
stream2=RandStream.getGlobalStream;
stream1.NormalTransform='Polar';
stream2.NormalTransform
ans =

Polar
```

The following table shows the methods available for the `RandStream` class. Static methods are indicated with the syntax `RandStream.methodName`.

Method	Description
RandStream	Create a random number stream
RandStream.create	Create multiple independent random number streams
get	Get the properties of a random stream
RandStream.list	List available random number generator algorithms
RandStream.getGlobalStream	Get the global random number stream
RandStream.setGlobalStream	Set the global random number stream
set	Set a property of a random stream
reset	Reset a stream to its initial internal state
rand	Generate pseudorandom numbers from a uniform distribution
randn	Generate pseudorandom numbers from a standard normal distribution
randi	Generate pseudorandom integers from a uniform discrete distribution
randperm	Random permutation of a set of values

The properties of a random stream are given the following table.

Property	Description
Type	(Read-only) Generator algorithm used by the stream. RandStream.list specifies the possible generators.
Seed	(Read-only) Seed value used to create the stream.
NumStreams	(Read-only) Number of streams in the group in which the current stream was created.
StreamIndex	(Read-only) Index of the current stream from among the group of streams with which the current stream was created.

Property	Description
State	Internal state of the generator. Do not depend on the format of this property. The value you assign to S.State must be a value previously read from S.State.
Substream	Index of the substream to which the stream is currently set. The default is 1. Multiple substreams are not supported by all generator types; the multiplicative lagged Fibonacci generator (mlfg6331_64) and combined multiple recursive generator (mrg32k3a) support substreams.
NormalTransform	Transformation algorithm used by randn(s, ...) to generate normal pseudorandom values. Possible values are 'Ziggurat', 'Polar', or 'Inversion'.
Antithetic	Logical value indicating whether S generates antithetic pseudorandom values. For uniform values, these are the usual values subtracted from 1. The default is false.
FullPrecision	Logical value indicating whether s generates values using its full precision. Some generators can create pseudorandom values faster, but with fewer random bits, if FullPrecision is false. The default is true.

Suppose you want to repeat a simulation. The RandStream class gives you several ways to replicate output. As shown in the previous example, you can save the state of the global stream.

```
myState=GlobalStream.State;
A=rand(1,100);
GlobalStream.State=myState;
B=rand(1,100);
isequal(A,B)

ans =
```

```
     1
```

You can also reset a stream to its initial settings with the method `reset`.

```
reset(GlobalStream)
A=rand(1,100);
reset(GlobalStream)
B=rand(1,100);
isequal(A,B)

ans =

     1
```

Random Number Data Types

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

```
GlobalStream=RandStream.getGlobalStream;
myState=GlobalStream.State;
A=rand(1,5);
class(A)

ans =

double
```

To specify the class as double explicitly:

```
GlobalStream.State=myState;
B=rand(1,5,'double');
class(B)

ans =

double
isequal(A,B)

ans =

     1
```

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

```
GlobalStream.State=myState;
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 =

    0.8235    0.6948    0.3171    0.9502    0.0344


B =

    0.8235    0.6948    0.3171    0.9502    0.0344
```

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
```

Creating and Controlling a Random Number Stream

In this section...
"Substreams" on page 3-24
"Choosing a Random Number Generator" on page 3-25

The `RandStream` class allows you to create a random number stream. This is useful for several reasons. For example, you might want to generate random values without affecting the state of the global stream. You might want separate sources of randomness in a simulation. Or you may need to use a different generator algorithm than the one MATLAB software uses at startup. With the `RandStream` constructor, you can create your own stream, set the writable properties, and use it 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` constructor.

```
myStream=RandStream('mlfg6331_64');
rand(myStream,1,5)

ans =

    0.6530    0.8147    0.7167    0.8615    0.0764
```

The random stream `myStream` acts separately from the global stream. The functions `rand`, `randn`, and `randi` will continue to draw from the global stream, and will not affect the results of the `RandStream` methods `rand`, `randn` and `randi` applied to `myStream`.

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 may want to return to a previous part of a simulation. A random stream can be controlled by having it jump to fixed checkpoints, called substreams. The `Substream` property allows you to jump back and forth among multiple substreams. To use the `Substream` property, create a stream using a generator that supports substreams. (See "Choosing a Random Number Generator" on page 3-25 for a list of generator algorithms and their properties.)

```
stream=RandStream('mlfg6331_64');
RandStream.setGlobalStream(stream)
```

The initial value of `Substream` is 1.

```
stream.Substream
```

```
ans =

    1
```

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, they can be used in loops.

```
for i=1:5
    stream.Substream=i;
    rand(1,i)
end
```

```
ans =
    0.6530

ans =
    0.3364    0.8265

ans =
    0.9539    0.6446    0.4913
```

```
ans =
    0.0244    0.5134    0.6305    0.6534

ans =
    0.3323    0.9296    0.5767    0.1233    0.6934
```

Each of these substreams can reproduce its loop iteration. For example, you can return to the 5th substream. The result will return the same values as the 5th output above.

```
stream.Substream=5;
rand(1,5)

ans =

    0.3323    0.9296    0.5767    0.1233    0.6934
```

Choosing a Random Number Generator

MATLAB offers several generator algorithm options. The following 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})

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
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 may be faster or return values with more precision. All pseudorandom number generators are based on deterministic algorithms, and all will fail 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 MATLAB software's choice of generators provide you with the means to do that. Although it is unlikely that your results will differ by more than 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)`. The `randn` algorithm used by default for `mcg16807` streams 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. It has a period of 2^{191}

and supports up to 2^{63} parallel streams via 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, and 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 $\mathbf{I}$ is the identity operator, $\mathbf{L}$ is the left shift operator, and $\mathbf{R}$ 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 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.

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.

Multiple Streams

MATLAB software includes generator algorithms that allow you to create multiple independent random number streams. The RandStream.create factory method allows you to create three 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 =

    1.0000   -0.0017   -0.0010
```

```
   -0.0017    1.0000   -0.0050
   -0.0010   -0.0050    1.0000
```

By using different seeds, you can create streams that return different values and act separately from one another.

```
s1=RandStream('mt19937ar','seed',1);
s2=RandStream('mt19937ar','seed',2);
s3=RandStream('mt19937ar','seed',3);
```

Seed values must be integers between 0 and $2^{32}-1$. With different seeds, streams typically return values that are uncorrelated.

```
r1=rand(s1,100000,1);
r2=rand(s2,100000,1);
r3=rand(s3,100000,1);
corrcoef([r1,r2,r3])

ans =

    1.0000    0.0030    0.0045
    0.0030    1.0000   -0.0015
    0.0045   -0.0015    1.0000
```

For generator types that do not explicitly support independent streams, different seeds provide a method to create multiple streams. However, using a generator specifically designed for multiple independent streams is a better option, as the statistical properties across streams are better understood.

Depending on the application, it might be useful to create only some of the streams in a set of independent streams. The StreamIndex property returns the index of a specified stream from a set of factory-generated streams.

```
numLabs=256;
labIndex=4;
s1=RandStream.create('mlfg6331_64',
       'NumStreams',numLabs,'StreamIndices',labIndex)

s1=
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.

Note Not all generators algorithms support multiple streams. See the table of generator algorithms in "Choosing a Random Number Generator" on page 3-25 for a summary of generator properties.

Replace Discouraged Syntaxes of rand and randn

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.
```

4

Sparse Matrices

Computational Advantages of Sparse Matrices

In this section...
"Memory Management" on page 4-2
"Computational Efficiency" on page 4-3

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 4-18

Constructing Sparse Matrices

In this section...
"Creating Sparse Matrices" on page 4-4
"Importing Sparse Matrices" on page 4-8

Creating Sparse Matrices

- "Converting Full to Sparse" on page 4-4
- "Creating Sparse Matrices Directly" on page 4-5
- "Creating Sparse Matrices from Their Diagonal Elements" on page 4-7

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.

```
S = sparse(A)
```

For example:

```
A = [ 0   0   0   5
      0   2   0   0
      1   3   0   0
      0   0   4   0];
S = sparse(A)
```

produces

```
 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.

```
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'
```

For n = 5, MATLAB responds with

```
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 4-18

Accessing Sparse Matrices

In this section...
"Nonzero Elements" on page 4-10
"Indices and Values" on page 4-12
"Indexing in Sparse Matrix Operations" on page 4-12
"Visualizing Sparse Matrices" on page 4-15

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.

```
[i,j,s] = find(S);
[m,n] = size(S);
S = 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 4-18

Sparse Matrix Operations

In this section...

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

- "Reordering for Sparsity" on page 4-21
- "Reordering to Reduce Bandwidth" on page 4-21
- "Approximate Minimum Degree Ordering" on page 4-21
- "Nested Dissection Ordering" on page 4-22

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, the statements

```
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)
```

produce:

```
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` produce

```
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'` produce

```
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 representations, let `I = speye(n)` be an identity matrix of the appropriate size. Then,

```
P = I(p,:)
P' = I(:,p)
p = (1:n)*P'
p = (P*(1:n)')'
```

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" on page 4-23 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" on page 4-23
- "Cholesky Factorization" on page 4-27
- "QR Factorization" on page 4-28

- "Incomplete Factorizations" on page 4-29

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

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

$$\mathrm{minimize}\left\|Ax - b\right\|_2$$

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);
nnz(A)
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-017

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-005

opts.type = 'crout';
nnz(L)+nnz(U)-size(A,1)
ans =
       31083
norm(P*A-L*U,'fro')./norm(A,'fro')
ans =
  9.7344e-005
```

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-017

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-004
```

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.

Systems of Linear Equations

There are two different classes of methods for solving systems of simultaneous linear equations:

- *Direct methods* are usually variants of Gaussian elimination. These methods involve the individual matrix elements directly, through matrix operations such as LU or Cholesky factorization. MATLAB implements direct methods through the matrix division operators / and \, which you can use to solve linear systems.

- *Iterative methods* produce only an approximate solution after a finite number of steps. These methods involve the coefficient matrix only indirectly, through a matrix-vector product or an abstract linear operator. Iterative methods are usually applied only to sparse matrices.

Direct Methods

Direct methods are usually faster and more generally applicable than indirect methods, if there is enough storage available to carry them out. Iterative methods are usually applicable to restricted cases of equations and depend on properties like diagonal dominance or the existence of an underlying differential operator. Direct methods are implemented in the core of the MATLAB software and are made as efficient as possible for general classes of matrices. Iterative methods are usually implemented in MATLAB-language files and can use the direct solution of subproblems or preconditioners.

Using a Different Preordering

If A is not diagonal, banded, triangular, or a permutation of a triangular matrix, backslash (\) reorders the indices of A to reduce the amount of fill-in—that is, the number of nonzero entries that are added to the sparse factorization matrices. The new ordering, called a *preordering*, is performed before the factorization of A. In some cases, you might be able to provide a better preordering than the one used by the backslash algorithm.

To use a different preordering, first turn off both of the automatic preorderings that backslash might perform by default, using the function `spparms` as follows:

```
currentParms = spparms;
spparms('autoamd',0);
spparms('autommd',0);
```

Now, assuming you have created a permutation vector p that specifies a preordering of the indices of A, apply backslash to the matrix A(:,p), whose columns are the columns of A, permuted according to the vector p.

```
x = A (:,p) \ b;
x(p) = x;
spparms(currentParms);
```

The command `spparms(defaultParms)` restores the controls to their prior state, in case you use A\b later without specifying an appropriate preordering.

Iterative Methods

Eleven functions are available that implement iterative methods for sparse systems of simultaneous linear systems.

Functions for Iterative Methods for Sparse Systems

Function	Method
bicg	Biconjugate gradient
bicgstab	Biconjugate gradient stabilized
bicgstabl	Biconjugate gradient stabilized (l)
cgs	Conjugate gradient squared
gmres	Generalized minimum residual
lsqr	Least squares
minres	Minimum residual
pcg	Preconditioned conjugate gradient
qmr	Quasiminimal residual
symmlq	Symmetric LQ
tfqmr	Transpose-free quasiminimal residual

These methods are designed to solve $Ax = b$ or minimize the norm of $b - Ax$. For the Preconditioned Conjugate Gradient method, `pcg`, A must be a symmetric, positive definite matrix. `minres` and `symmlq` can be used on symmetric indefinite matrices. For `lsqr`, the matrix need not be square. The other seven can handle nonsymmetric, square matrices and each method has a distinct benefit.

All eleven methods can make use of preconditioners. The linear system

$$Ax = b$$

is replaced by the equivalent system

$$M^{-1}Ax = M^{-1}b$$

The preconditioner *M* is chosen to accelerate convergence of the iterative method. In many cases, the preconditioners occur naturally in the mathematical model. A partial differential equation with variable coefficients can be approximated by one with constant coefficients, for example. Incomplete matrix factorizations can be used in the absence of natural preconditioners.

The five-point finite difference approximation to Laplace's equation on a square, two-dimensional domain provides an example. The following statements use the preconditioned conjugate gradient method preconditioner *M = L*L'*, where *L* is the zero-fill incomplete Cholesky factor of *A*.

```
A = delsq(numgrid('S',50));
b = ones(size(A,1),1);
tol = 1e-3;
maxit = 100;
L = ichol(A);
[x,flag,err,iter,res] = pcg(A,b,tol,maxit,L,L');
```

Twenty-one iterations are required to achieve the prescribed accuracy. On the other hand, using a different preconditioner may yield better results. For example, using `ichol` to construct a modified incomplete Cholesky, the prescribed accuracy is met after only 15 iterations:

```
L = ichol(A,struct('type','nofill','michol','on'));
[x,flag,err,iter,res] = pcg(A,b,tol,maxit,L,L');
```

Background information on these iterative methods and incomplete factorizations is available in [2] and [6].

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: $Av = \lambda Bv$.

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.

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,1,0);
```

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
```


The numerical techniques used in `eigs` and `svds` are described in [6].

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 4-10

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')
```

A Finite Difference Grid

nz = 675

Show a smaller version of the matrix as a sample.

```
g = numgrid(R,10)
```

```
g = 10×10
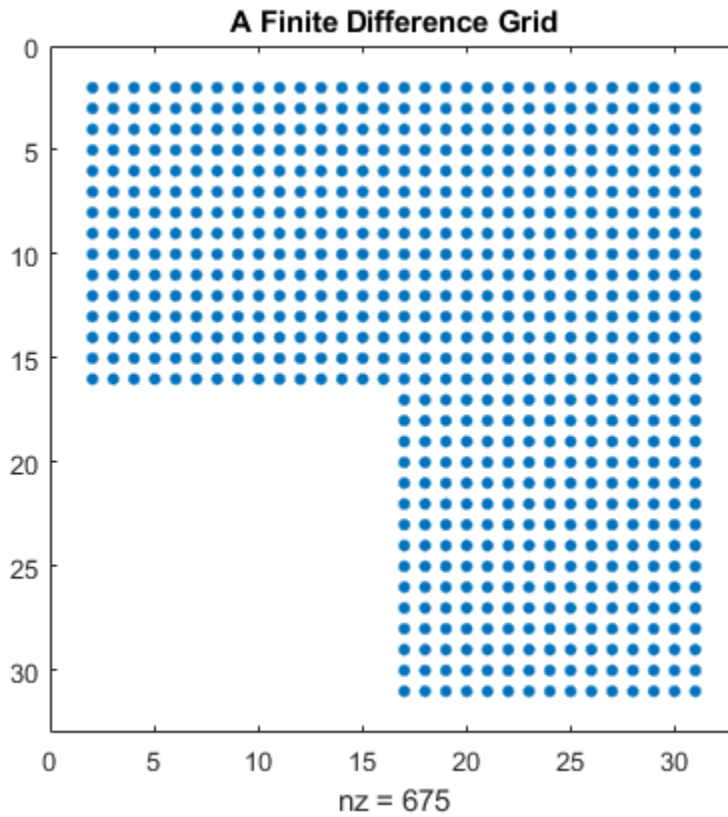
     0     0     0     0     0     0     0     0     0     0
     0     1     5     9    13    17    25    33    41     0
     0     2     6    10    14    18    26    34    42     0
     0     3     7    11    15    19    27    35    43     0
     0     4     8    12    16    20    28    36    44     0
     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);
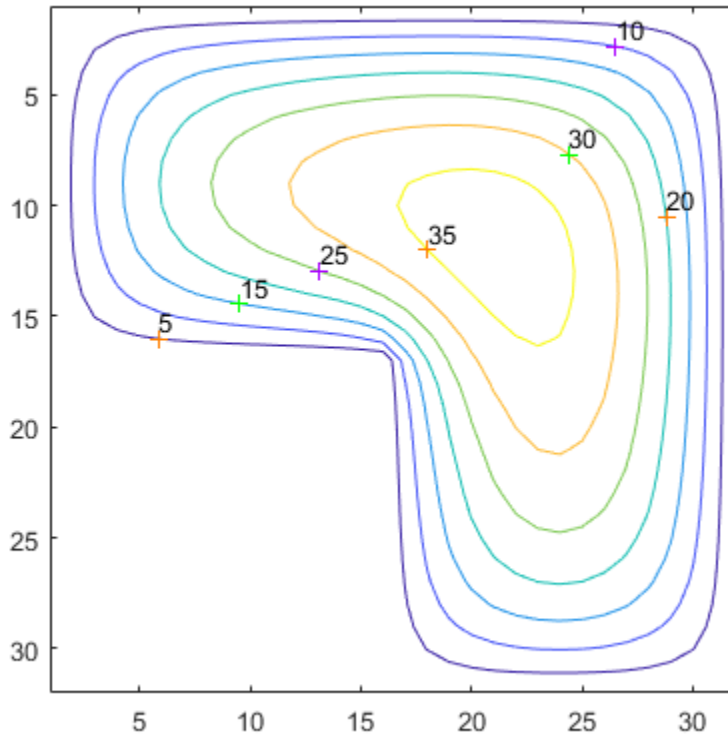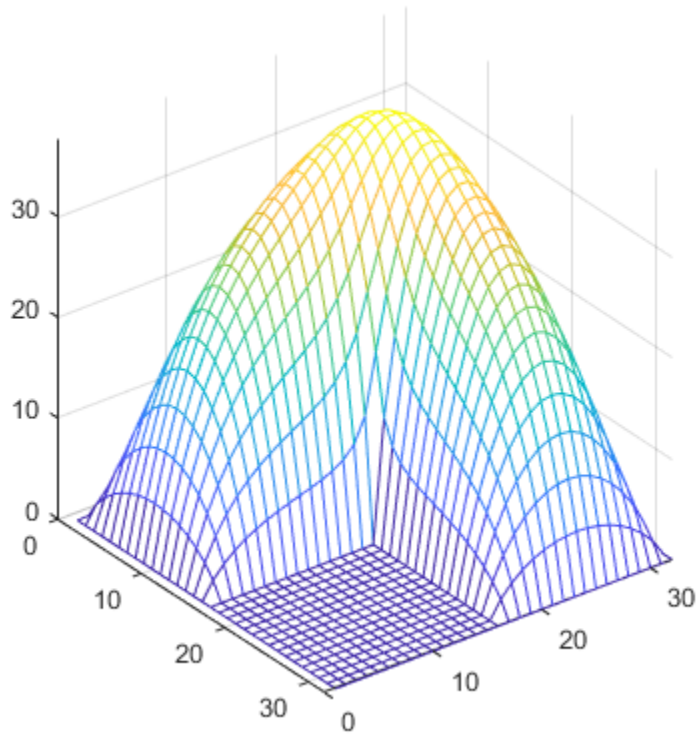
% Forming the sparse adjacency matrix and making it positive definite
n = max(max(i),max(j));
A = sparse(i,j,-1,n,n);
A = A + A';
d = abs(sum(A)) + 1;
A = A + diag(sparse(d));

% Plotting the finite element mesh
gplot(A,[x y])
title('Airfoil Cross-Section')
```

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')
```

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')
```

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')
```

Approximate Minimum Degree

nz = 28831

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];
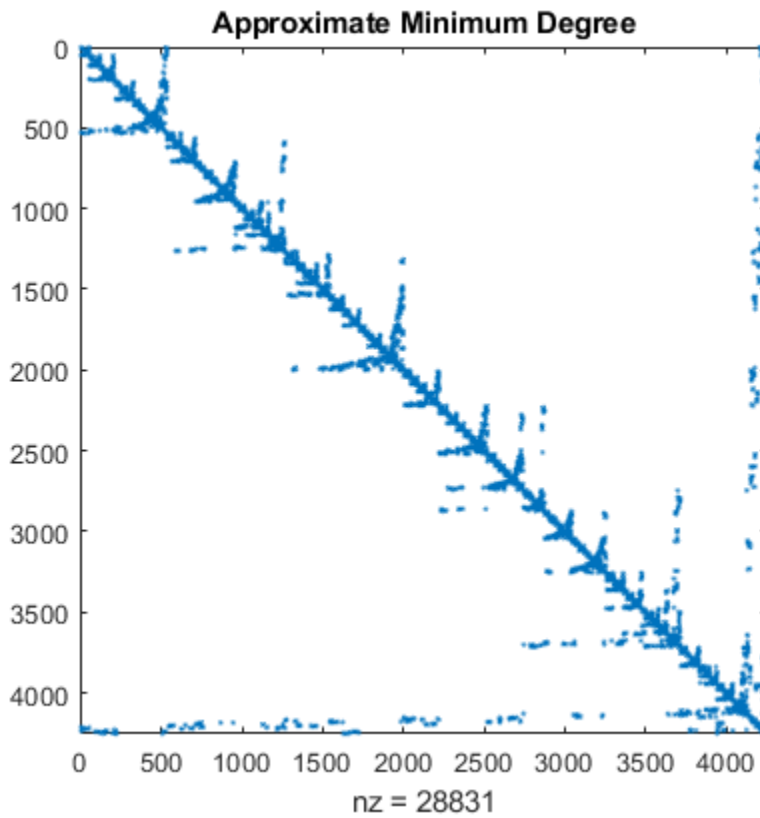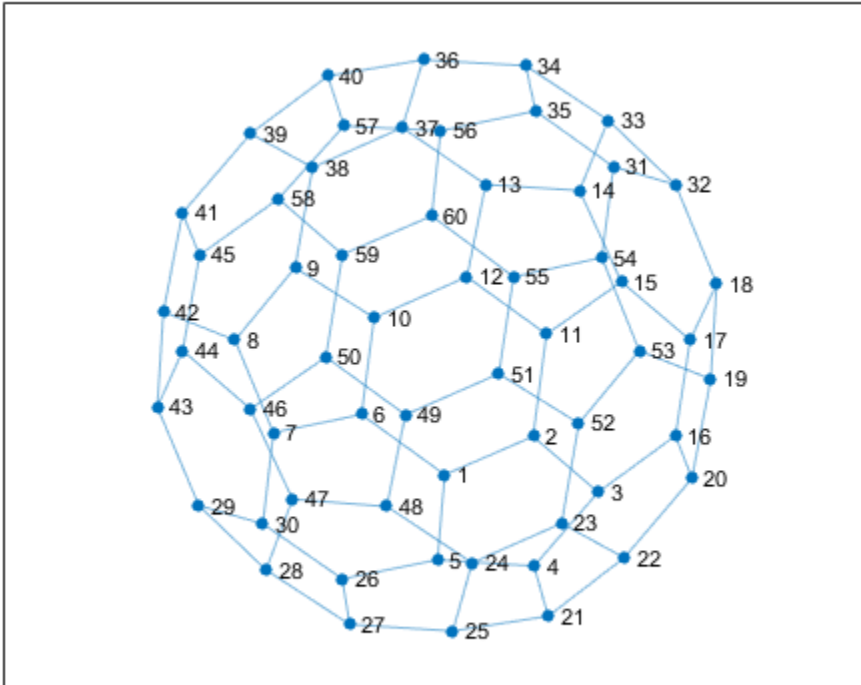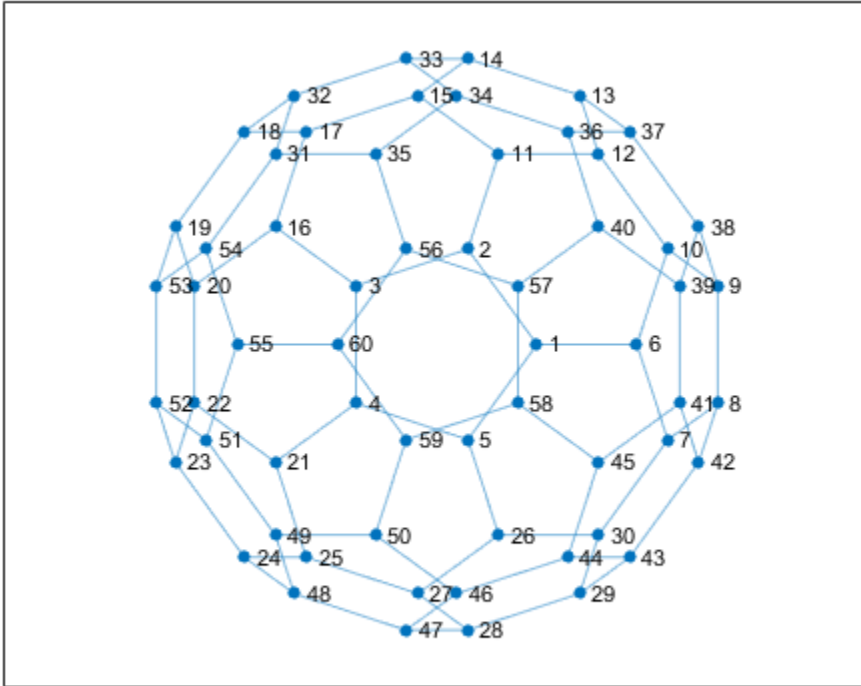
% Draw a picture showing the connected nodes.
```

```
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
```

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')
```

Top Left Corner of Bucky Ball Adjacency Matrix

nz = 80

Finally, here is a `spy` plot of the entire bucky ball adjacency matrix.

```
spy(A)
title('Bucky Ball Adjacency Matrix')
```

Bucky Ball Adjacency Matrix

nz = 180

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 Harwell-Boeing test matrix west0479. This matrix describes connections in a model of a diffraction column in a chemical plant.

```
load west0479.mat
A = west0479;
S = A * A' + speye(size(A));
pct = 100 / numel(A);
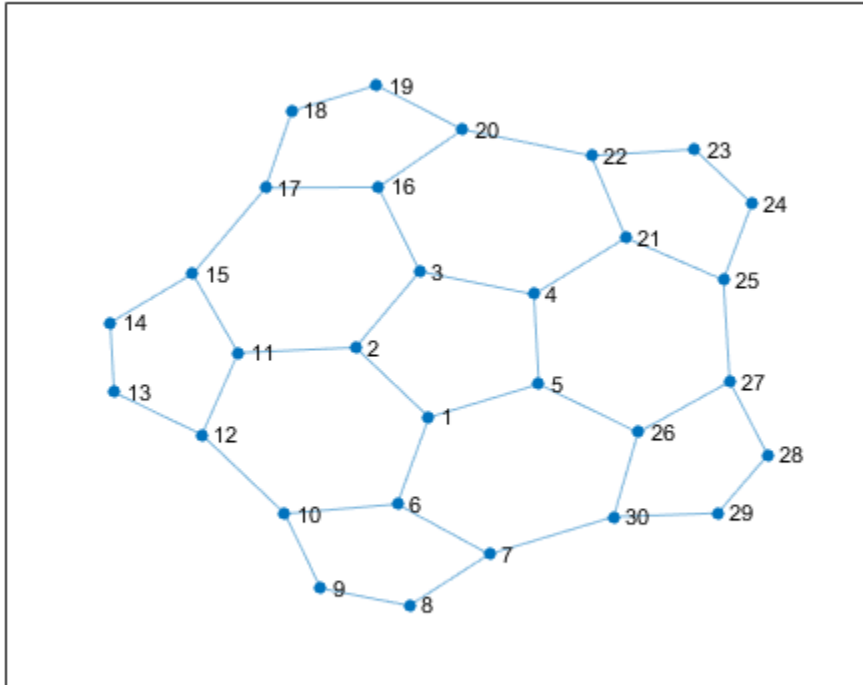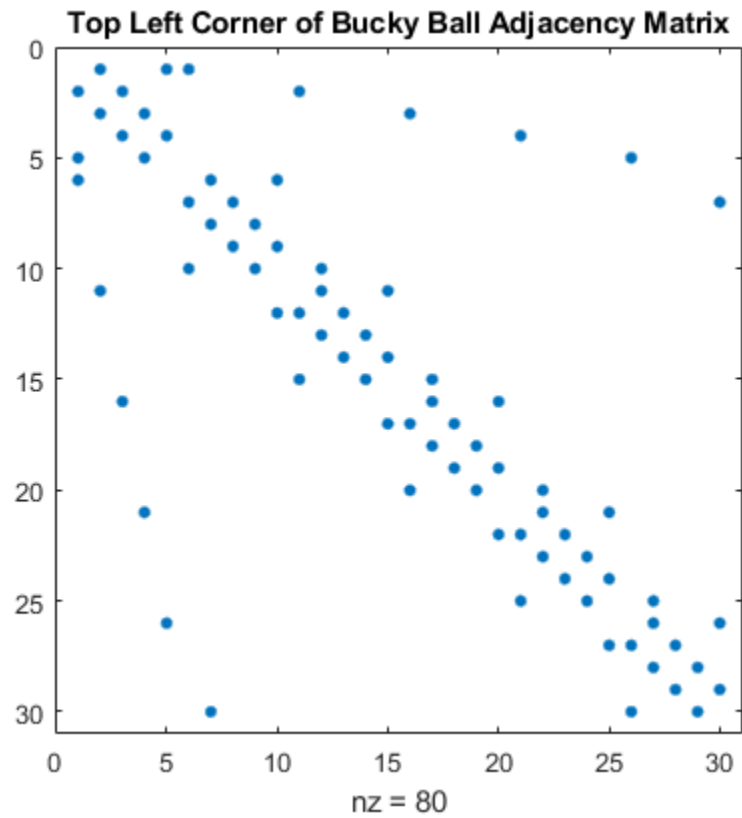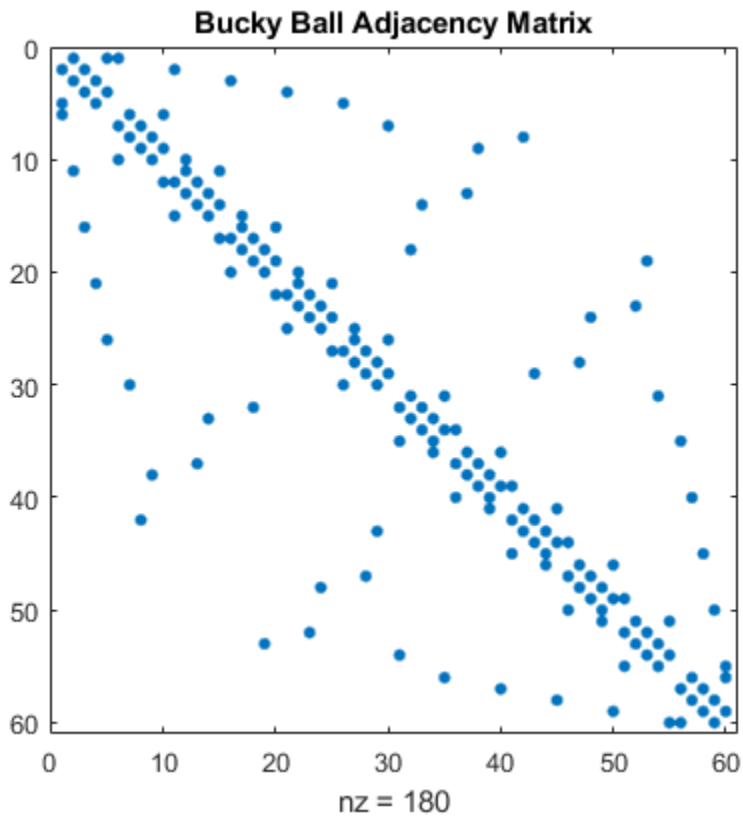
spy(S)
title('A Sparse Symmetric Matrix')
nz = nnz(S);
xlabel(sprintf('Nonzeros = %d (%.3f%%)',nz,nz*pct));
```

A Sparse Symmetric Matrix

Nonzeros = 7551 (3.291%)

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));
```

4-59

Cholesky Decomposition of S
Nonzeros = 30366 (13.23%)

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 time and storage cost.

Three different reorderings supported by MATLAB® are:

- Reverse Cuthill-McKee
- Column count
- Minimum degree

Test the effects of these sparse matrix reorderings on the west0479 matrix.

Reordering 1: Reverse Cuthill-McKee

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.

```
p = symrcm(S);
spy(S(p,p))
title('S(p,p) After Cuthill-McKee Ordering')
nz = nnz(S);
xlabel(sprintf('Nonzeros = %d (%.3f%%)',nz,nz*pct));
```


S(p,p) After Cuthill-McKee Ordering

Nonzeros = 7551 (3.291%)

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(p,p),'lower');
spy(L)
title('chol(S(p,p)) After Cuthill-McKee Ordering')
nc(2) = nnz(L);
xlabel(sprintf('Nonzeros = %d (%.2f%%)', nc(2),nc(2)*pct));
```


chol(S(p,p)) After Cuthill-McKee Ordering

Nonzeros = 23864 (10.40%)

Reordering 2: Column Count

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));
```


S(q,q) After Column Count Ordering

Nonzeros = 7551 (3.291%)

For this matrix, the column count ordering happens to reduce the time and storage for Cholesky factorization, but this behavior is not guaranteed in general.

```
L = chol(S(q,q),'lower');
spy(L)
title('chol(S(q,q)) After Column Count Ordering')
nc(3) = nnz(L);
xlabel(sprintf('Nonzeros = %d (%.2f%%)',nc(3),nc(3)*pct));
```

4-63

chol(S(q,q)) After Column Count Ordering

Nonzeros = 14874 (6.48%)

Reordering 3: Minimum Degree

The symamd command uses the approximate minimum degree algorithm (a powerful graph-theoretic technique) to produce large blocks of zeros in the matrix.

```
r = symamd(S);
spy(S(r,r))
title('S(r,r) After Minimum Degree Ordering')
nz = nnz(S);
xlabel(sprintf('Nonzeros = %d (%.3f%%)',nz,nz*pct));
```

S(r,r) After Minimum Degree Ordering

Nonzeros = 7551 (3.291%)

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));
```

chol(S(r,r)) After Minimum Degree Ordering

Nonzeros = 8412 (3.67%)

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 13% of its elements as nonzeros, using `symamd` reduces that density to only about 4%.

```
labels = {'Original','Cuthill-McKee','Column Count','Min Degree'};
bar(nc*pct)
title('Nonzeros After Cholesky Factorization')
ylabel('Percent');
ax = gca;
ax.XTickLabel = labels;
ax.XTickLabelRotation = -45;
```

Nonzeros After Cholesky Factorization

See Also
chol | colperm | nnz | spy | symamd | symrcm

Graph and Network Algorithms

Directed and Undirected Graphs

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 =
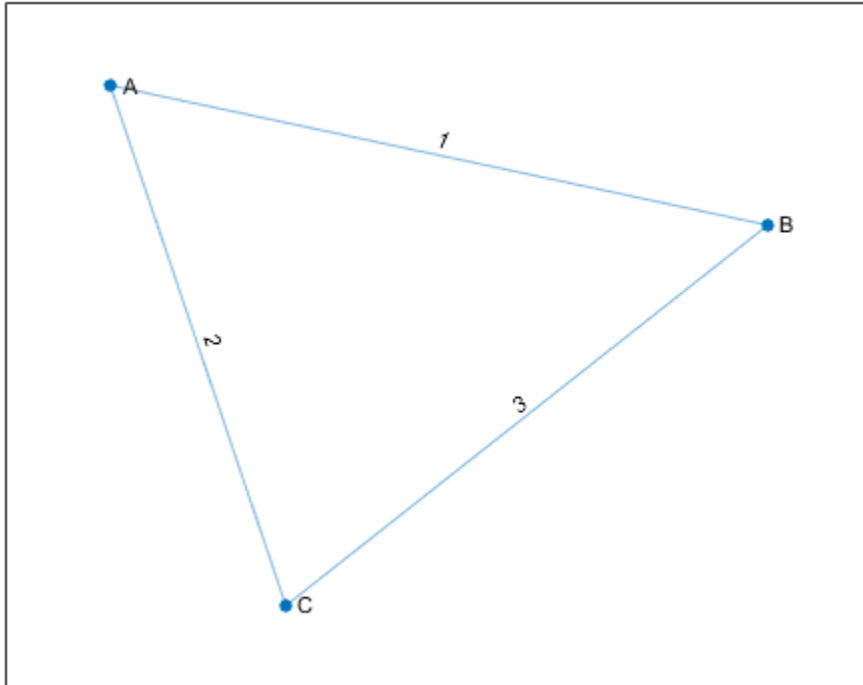
  graph with properties:

    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 5-13 for some common graph modification examples.

See Also
digraph | graph

More About

- "Modify Nodes and Edges of Existing Graph" on page 5-13
- "Add Graph Node Names, Edge Weights, and Other Attributes" on page 5-17
- "Graph Plotting and Customization" on page 5-23

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=1×1 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 array
    {'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:
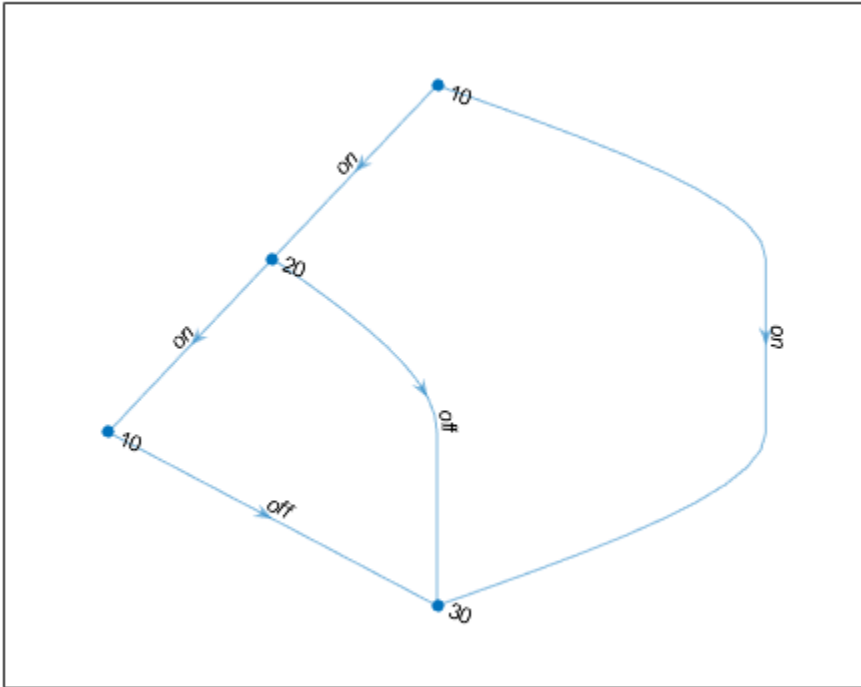
     NodeColor: [0 0.4470 0.7410]
    MarkerSize: 4
        Marker: 'o'
     EdgeColor: [0 0.4470 0.7410]
     LineWidth: 0.5000
     LineStyle: '-'
     NodeLabel: {'10'  '20'  '10'  '30'}
     EdgeLabel: {'on'  'on'  'on'  'off'  'off'}
         XData: [2 1.5000 1 2]
         YData: [4 3 2 1]
         ZData: [0 0 0 0]
```

5-21

```
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 5-13

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:
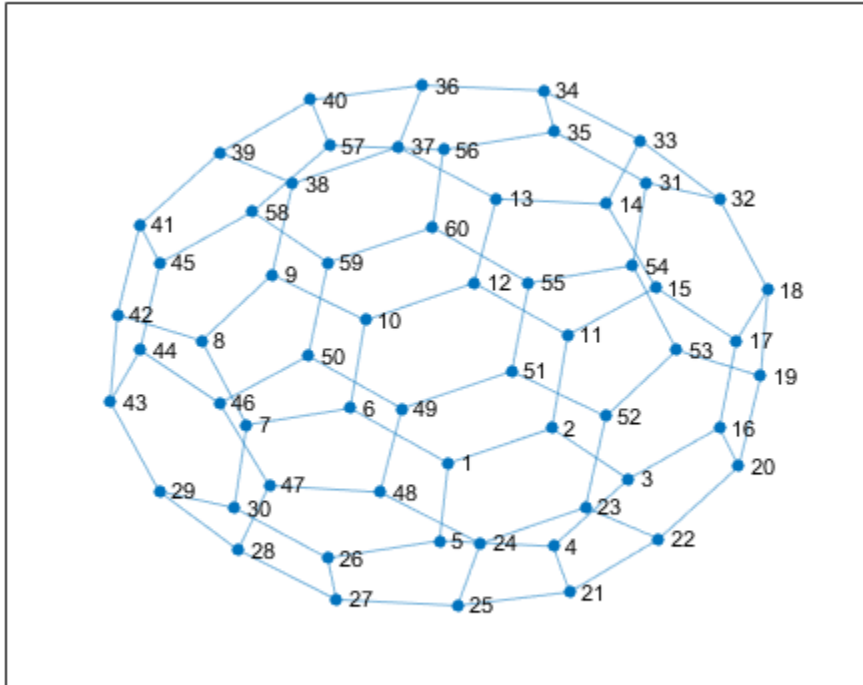
      NodeColor: [0 0.4470 0.7410]
     MarkerSize: 4
         Marker: 'o'
      EdgeColor: [0 0.4470 0.7410]
      LineWidth: 0.5000
      LineStyle: '-'
      NodeLabel: {1x60 cell}
      EdgeLabel: {}
          XData: [1x60 double]
          YData: [1x60 double]
          ZData: [1x60 double]
```

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')
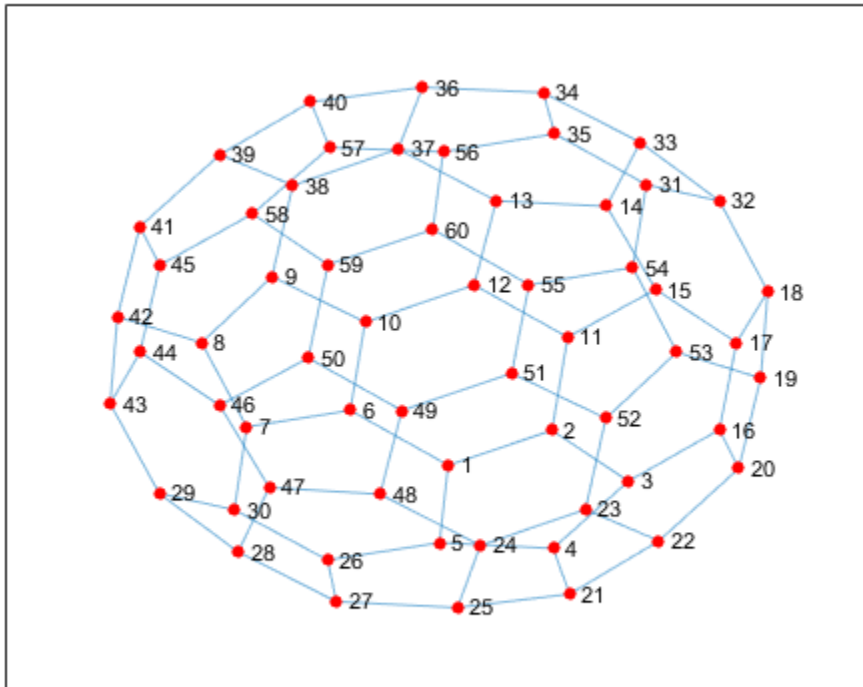
G =
  graph with properties:

    Edges: [130x2 table]
    Nodes: [75x0 table]


p = plot(G)
```

```
p =
  GraphPlot with properties:
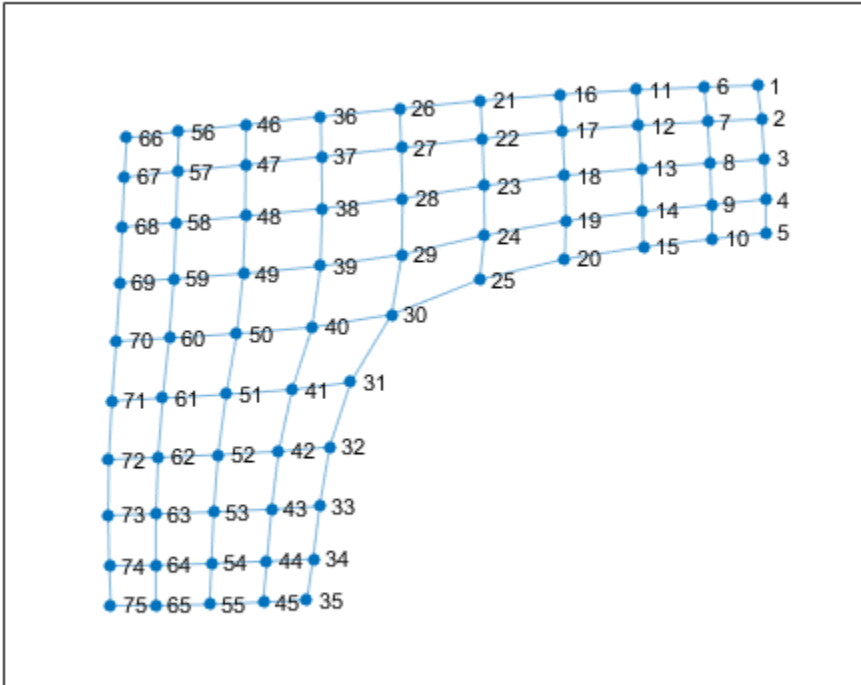
      NodeColor: [0 0.4470 0.7410]
     MarkerSize: 4
         Marker: 'o'
      EdgeColor: [0 0.4470 0.7410]
      LineWidth: 0.5000
      LineStyle: '-'
      NodeLabel: {1x75 cell}
      EdgeLabel: {}
          XData: [1x75 double]
          YData: [1x75 double]
          ZData: [1x75 double]
```

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
```

5-31

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)
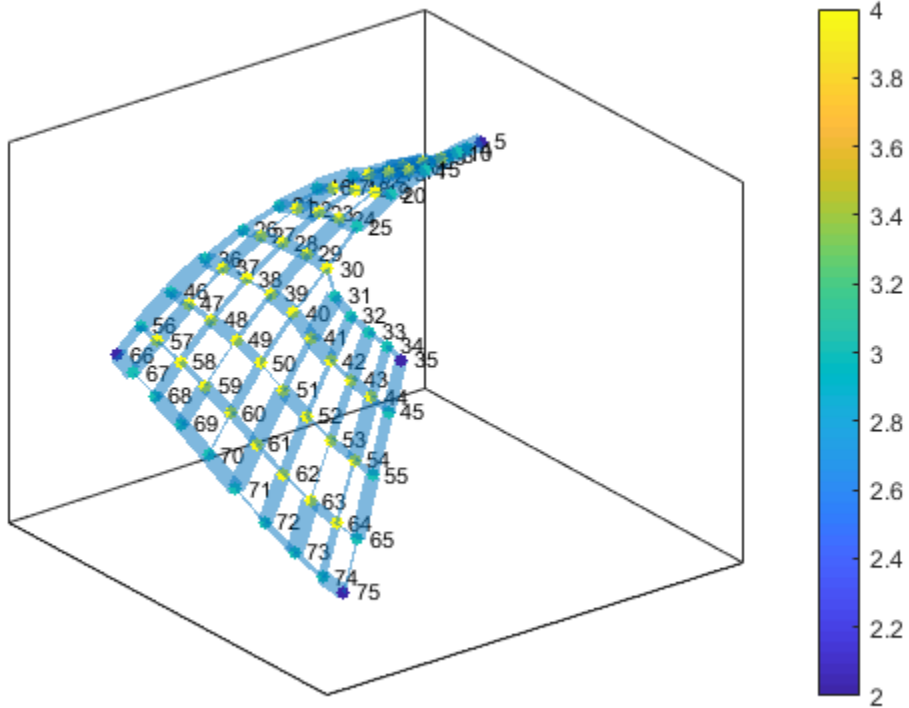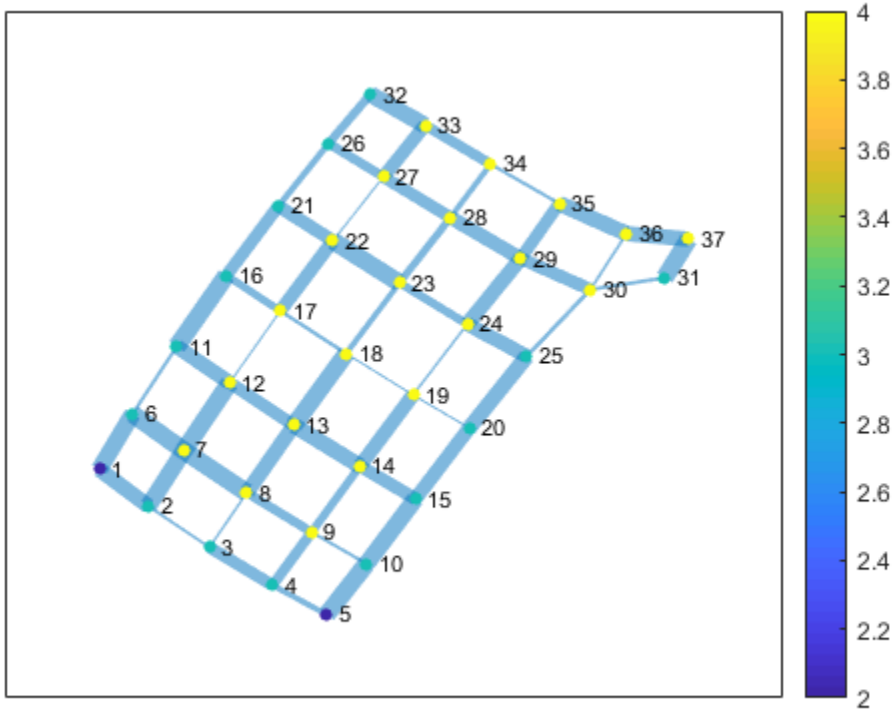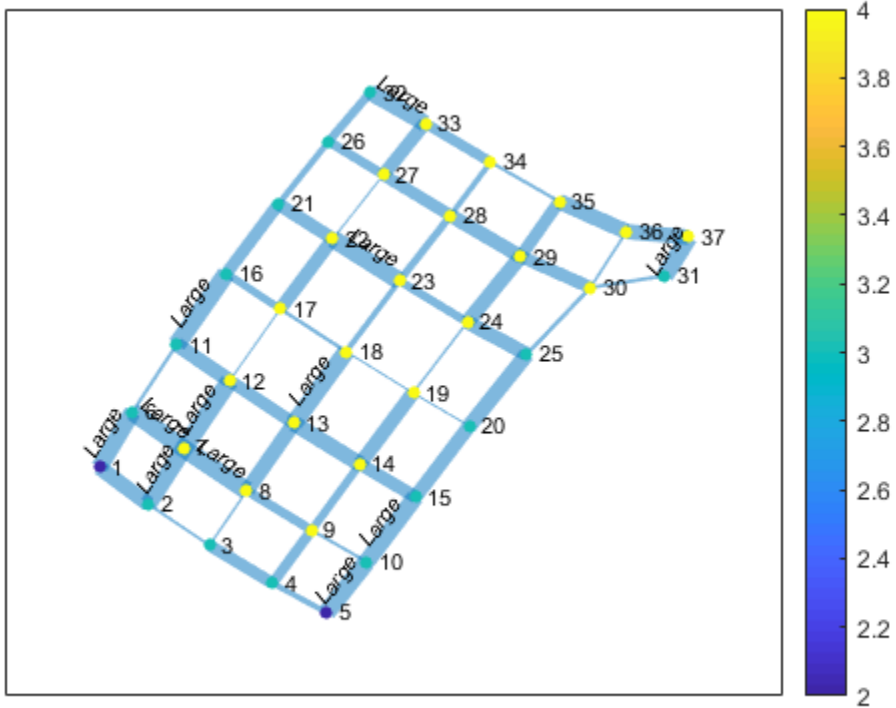
path = 1×10

   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 Cursor" on page 5-49

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 =
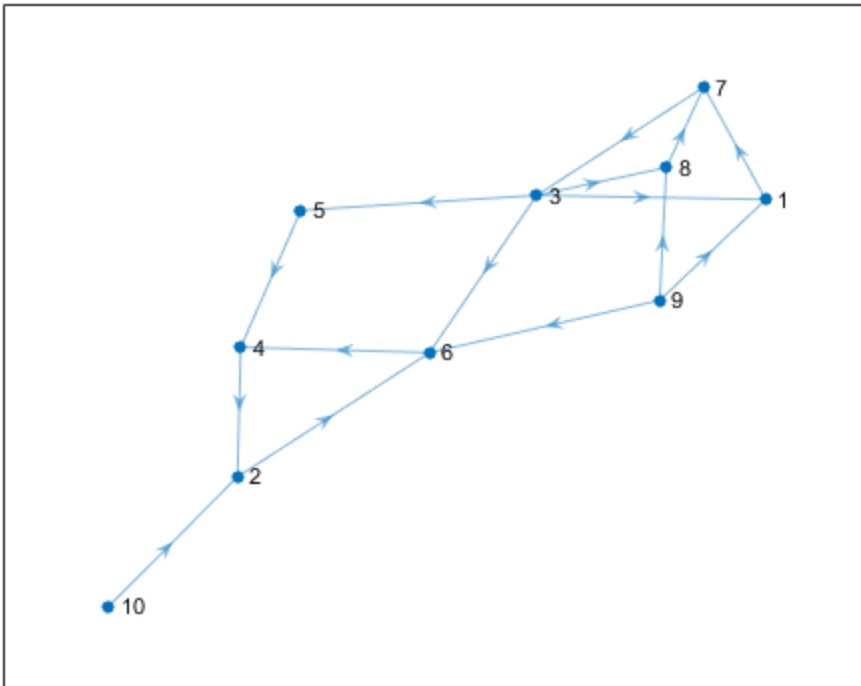
  38x4 table
```

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 1984-2017 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'
```

```matlab
                highlight(h,t.Node(ii),'NodeColor','r');
            case 'finishnode'
                highlight(h,t.Node(ii),'NodeColor','k');
            otherwise
                if isundirected
                    edgeind = intersect(find(G.Edges.Weight == t.EdgeIndex(ii)), ...
                        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`.

startnode on Node 1
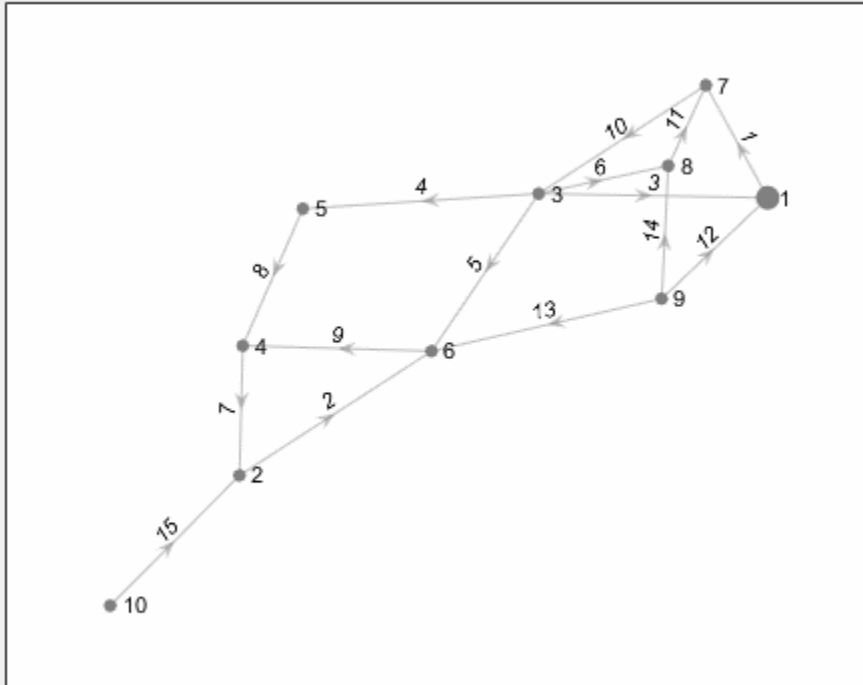
See Also

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
```
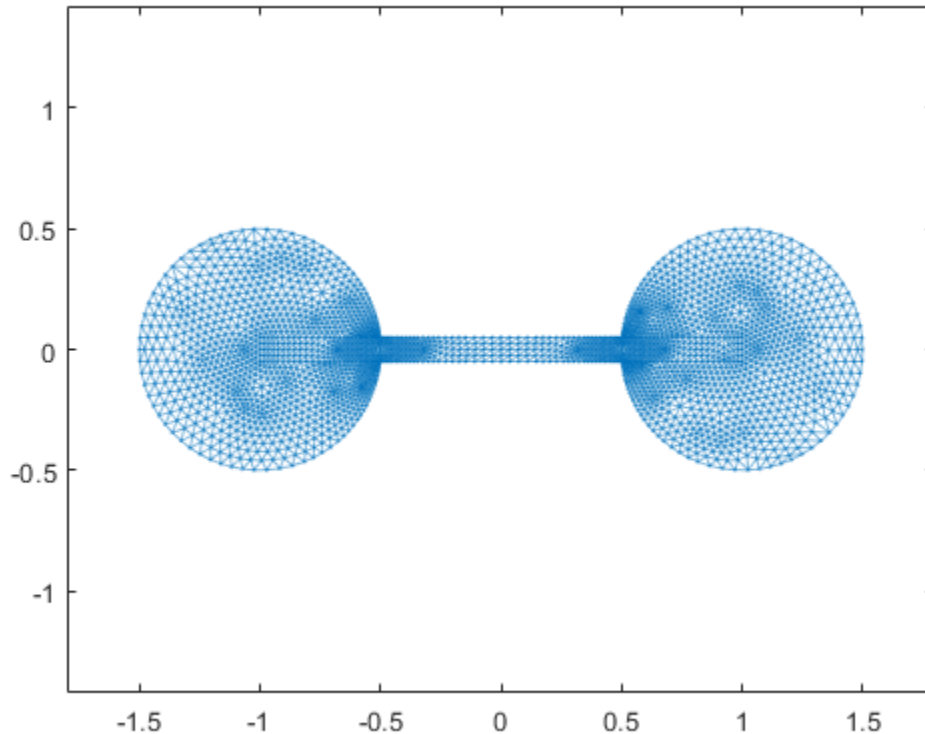
Calculate Laplacian Matrix and Fiedler Vector

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
```
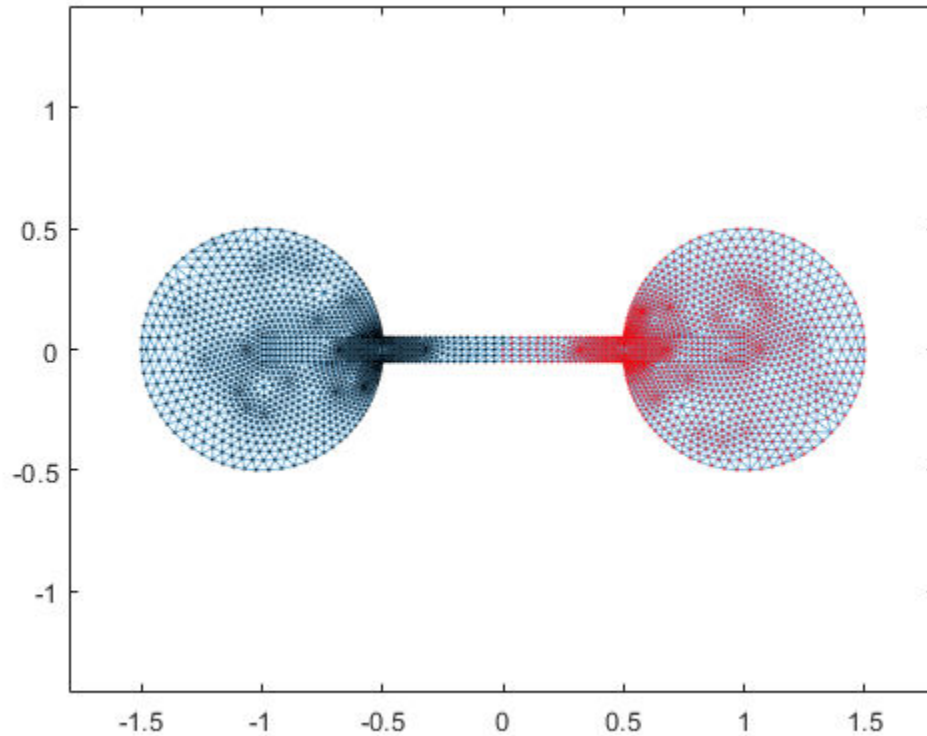
For the bar bell graph, this partition bisects the graph nicely into two equal sets of nodes. However, the sign cut does not always produce a balanced cut.

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 Cursor

This example shows how to customize the `GraphPlot` data cursor to display extra node properties of a graph.

Create `GraphPlot` object

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);
```
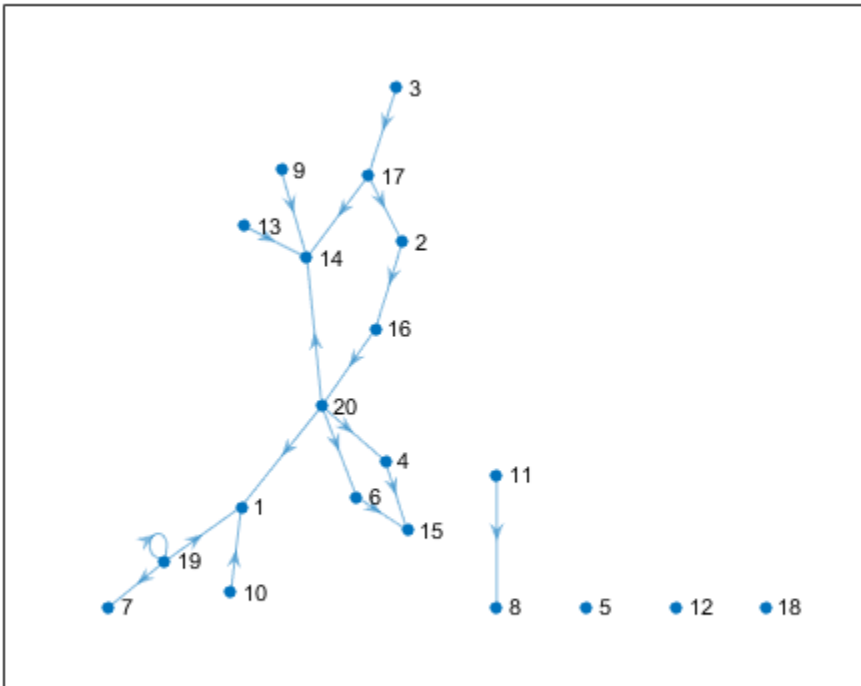
Enable Data Cursor Mode

Enable data cursor mode by selecting the **Data Cursor** item in the **Tools** menu.

With data cursor mode enabled, click a node in the graph to bring up the cursor display. The data cursor allows you to select nodes in a graph plot and view properties of the nodes. By default, the data cursor for an undirected graph displays the node ID number and degree. For directed graphs, the display includes the node ID number, in-degree, and out-degree.
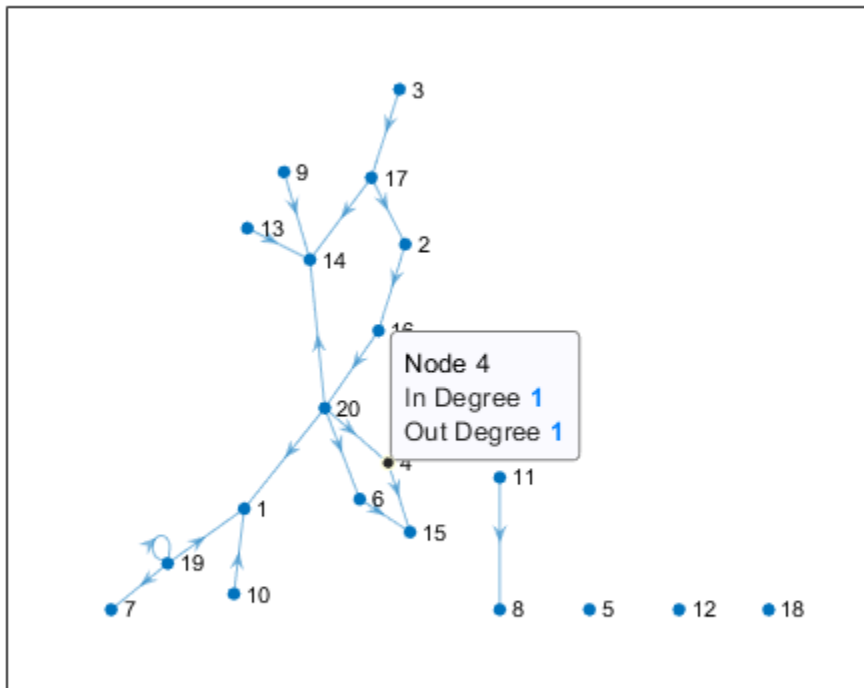
To display extra node properties, such as `wifi`, the data cursor callback function needs to be modified. For general information on using and customizing data cursors, see "Interactively Explore Plotted Data".

Customize Text Displayed by Data Cursor

Customize the text displayed by the data cursor by writing a new data cursor callback function.

1. Save the function `GraphCursorCallback.m` in your current directory:

```
function output_txt = GraphCursorCallback(obj,event_obj,NodeProperties)
% Display the position of the data cursor
% obj          Currently not used (empty)
% event_obj    Handle to event object
% output_txt   Data cursor text (character vector or cell array of character vectors).

h = get(event_obj,'Target');
pos = get(event_obj,'Position');
ind = find(h.XData == pos(1) & h.YData == pos(2), 1);

output_txt = {['Node ' num2str(ind)], ...
    ['Wifi: ' num2str(NodeProperties.wifi(ind))]};
```

A standard data cursor callback function accepts two input arguments. `GraphCursorCallback` accepts an additional input argument, `NodeProperties`, so that the data cursor gains access to the extra node properties in the graph, such as `wifi`.

2. Connect the `GraphCursorCallback` function to the data cursor by changing the `UpdateFcn` property of the data cursor manager object. This command uses an anonymous function to pass the `G.Nodes` table as the third input to `GraphCursorCallback`. This technique only takes a snapshot of the `G.Nodes` table, so if the graph properties subsequently change, then you need to change the `UpdateFcn` property again.

```
  hdt = datacursormode;
  hdt.UpdateFcn = @(obj,event_obj) GraphCursorCallback(obj,event_obj,G.Nodes);
```

3. Now that `GraphCursorCallback` is connected to the data cursor, delete any old data tips in the plot, then reenable data cursor mode and select a node. The new display of the data cursor includes the `wifi` node property as defined in `GraphCursorCallback`.
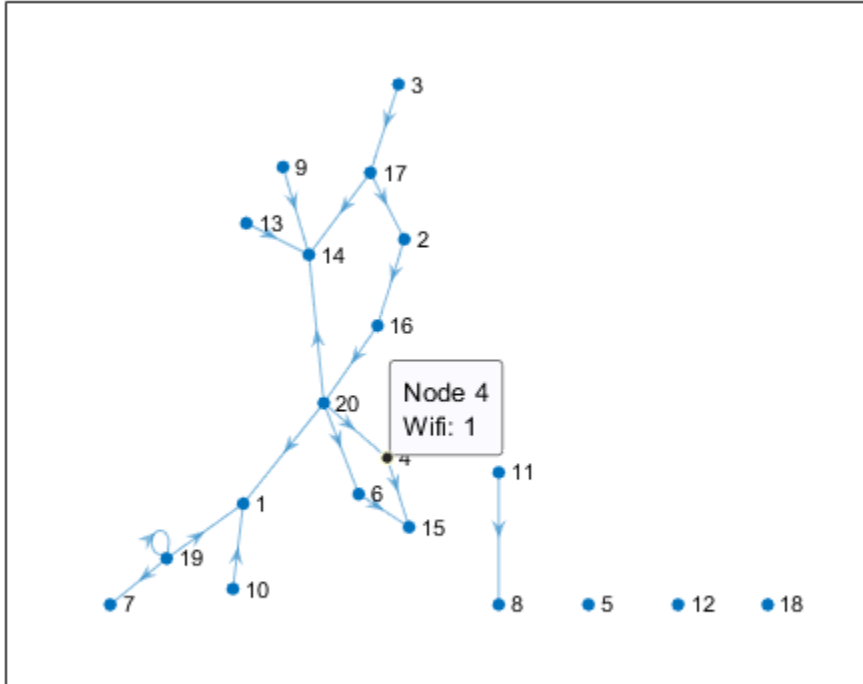
See Also

datacursormode | digraph | graph

More About

- "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`.

```
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

% Copyright 2015 The MathWorks, Inc.
```

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')
```

Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0$
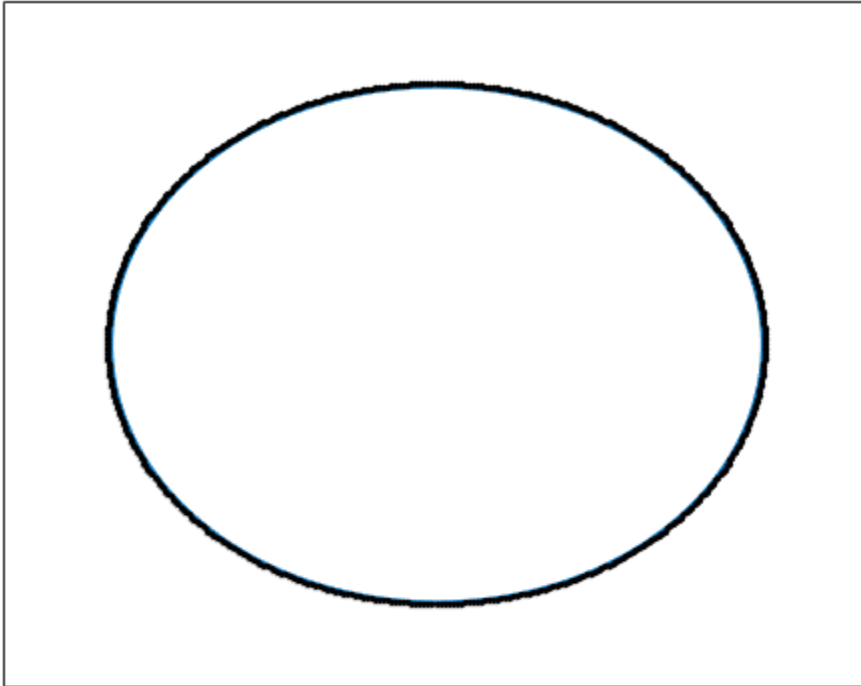
Some Random Edges

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')
```

Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0.15$
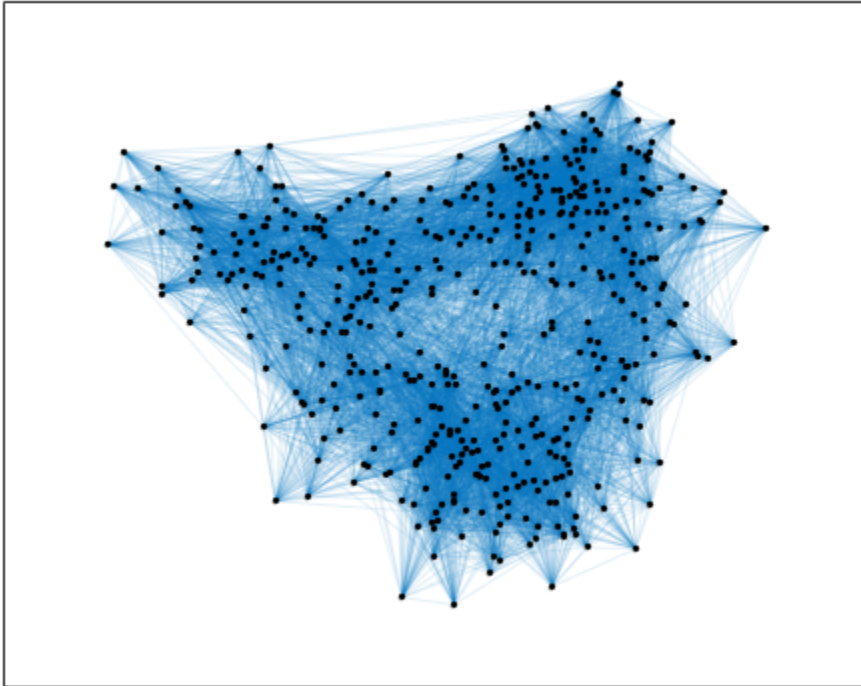
```
h3 = WattsStrogatz(500,25,0.50);
plot(h3,'NodeColor','k','EdgeAlpha',0.1);
title('Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0.50$', ...
    'Interpreter','latex')
```

Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0.50$
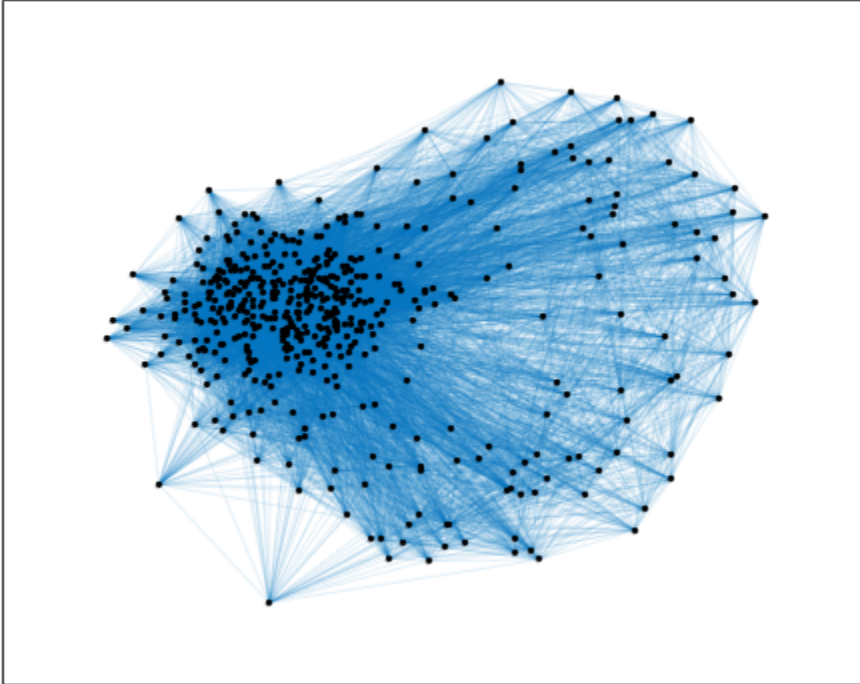
Random Graph

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 $\beta = 1$
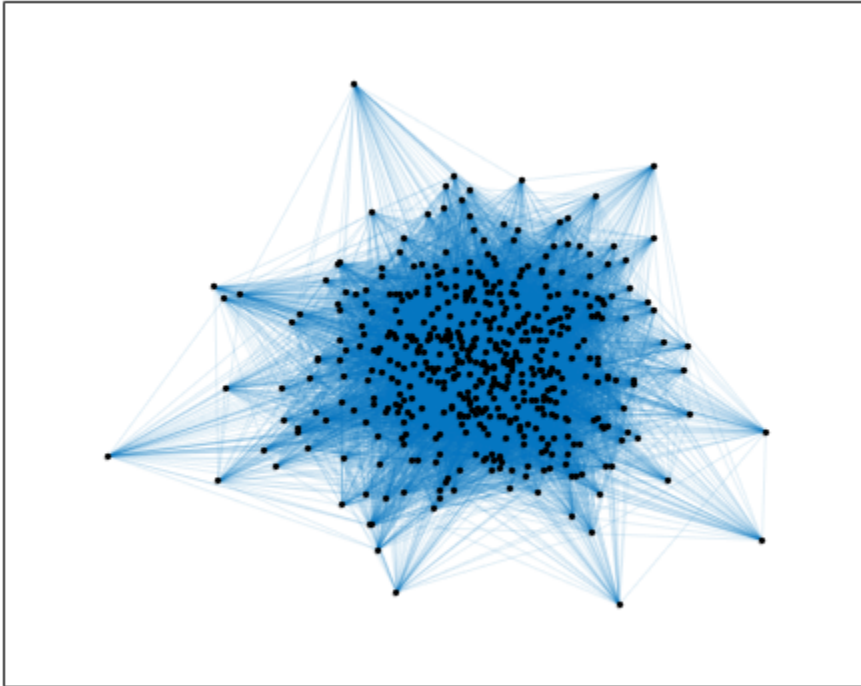
Degree Distribution

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')
```
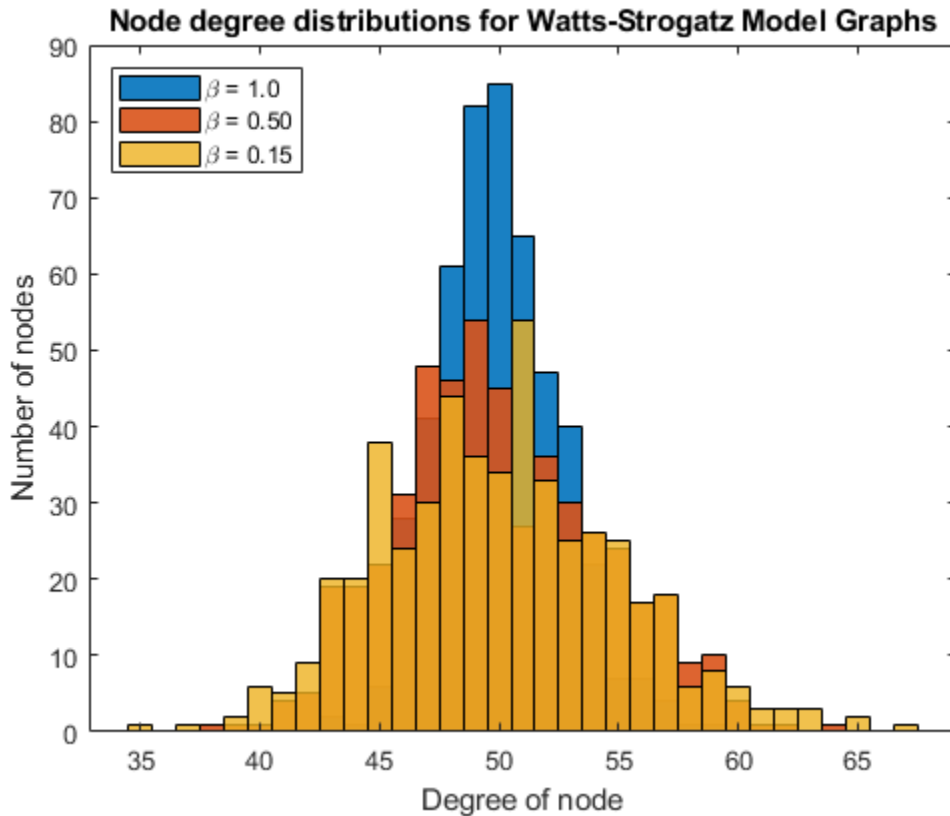

Hub Formation

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
```

Watts-Strogatz Graph with $N = 500$ nodes, $K = 25$, and $\beta = 0.15$
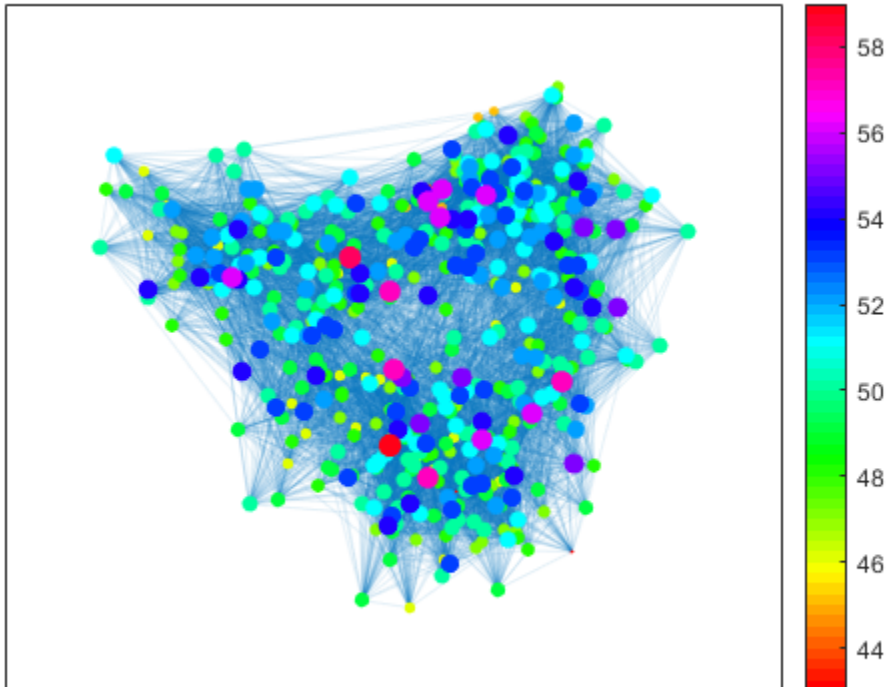
See Also

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')
```


PageRank Score Transfer Between Nodes
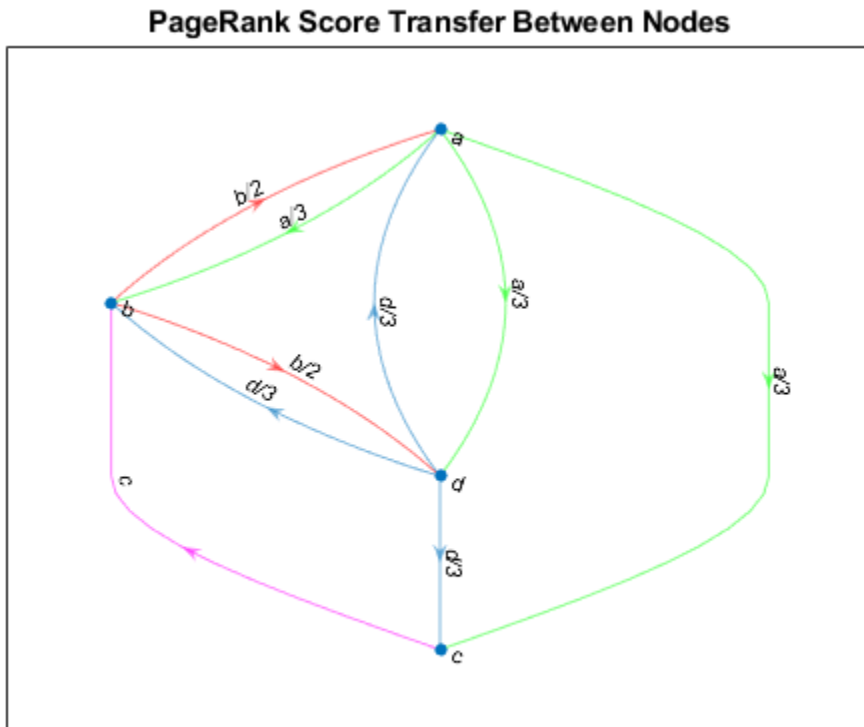
The `centrality` function contains an option for calculating PageRank scores.

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'})
```
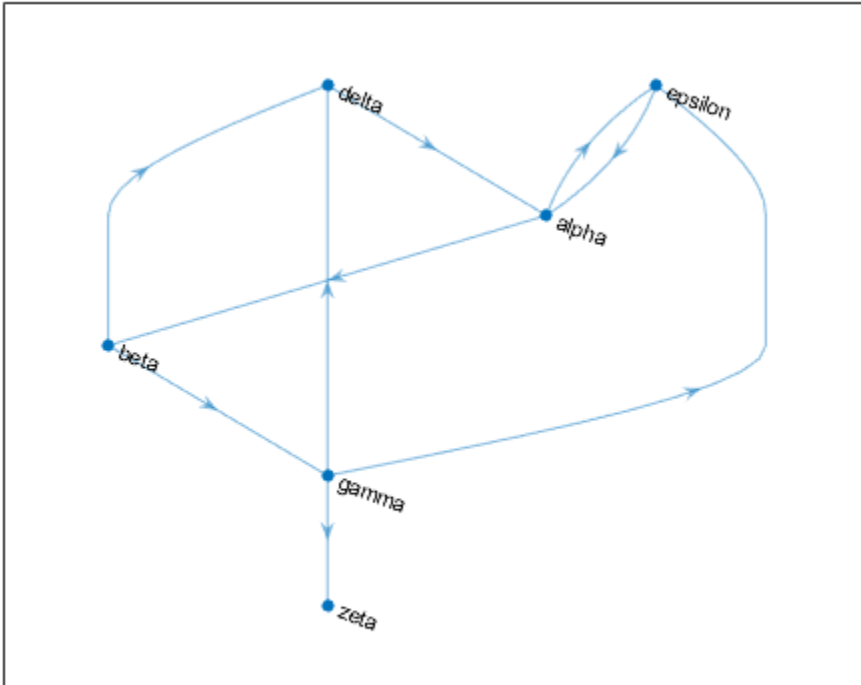
Calculate the PageRank centrality score for this graph. Use a follow probability (otherwise known as a damping factor) of `0.85`.

```
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)
```

nz = 632
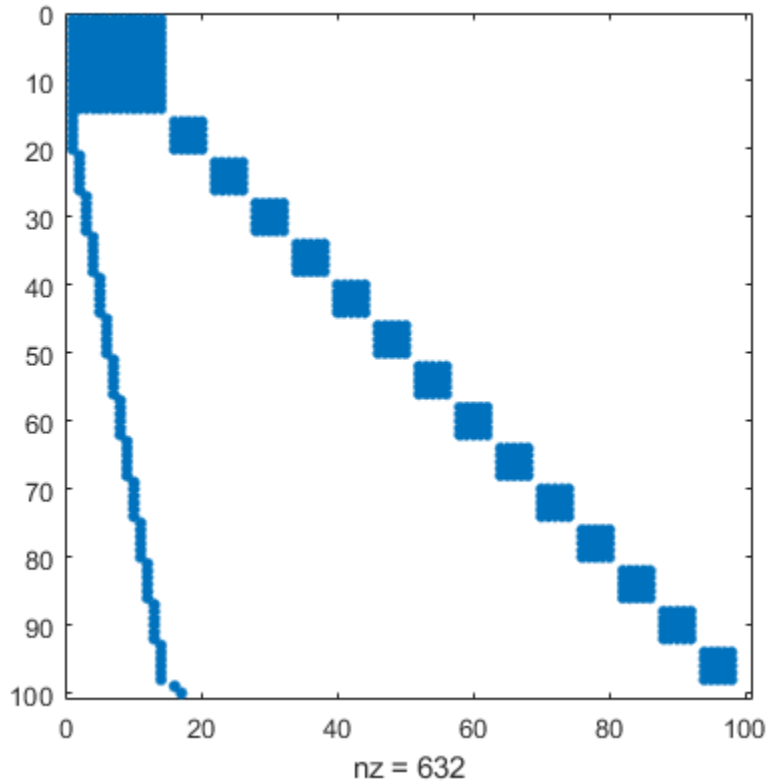
Create a directed graph with the sparse adjacency matrix, A, using the URLs contained in U as node names.

```
G = digraph(A,U)

G =
  digraph with properties:

    Edges: [632x1 table]
    Nodes: [100x1 table]
```

Plot the graph using the force layout.

```matlab
plot(G,'NodeLabel',{},'NodeColor',[0.93 0.78 0],'Layout','force');
title('Websites linked to https://www.mathworks.com')
```


Websites linked to https://www.mathworks.com
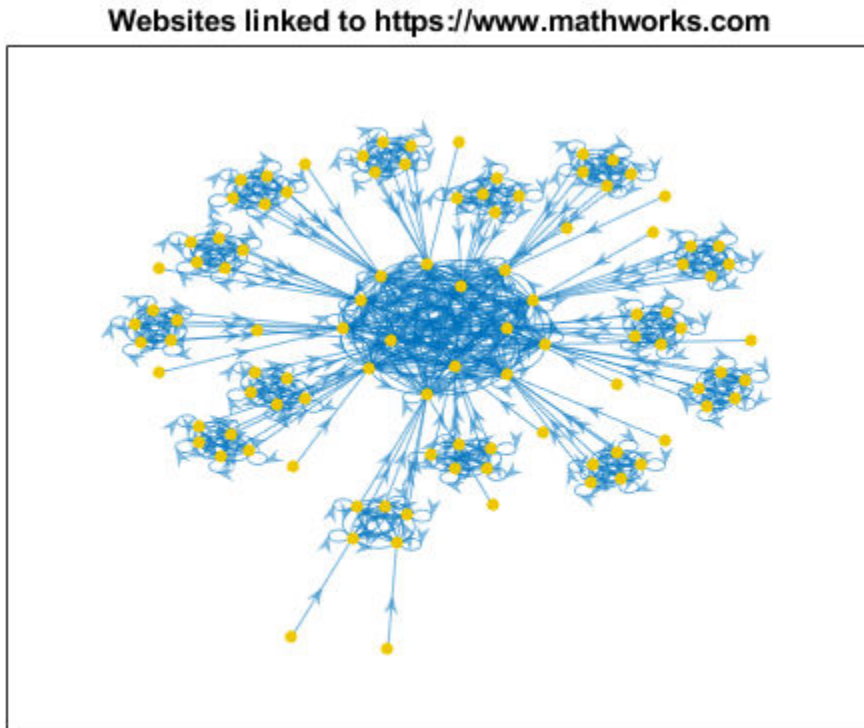
Compute the PageRank scores for the graph, G, using 200 iterations and a damping factor of 0.85. Add the scores and degree information to the nodes table of the graph.

```matlab
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.

```matlab
G.Nodes(1:25,:)
```

ans=*25×4 table*

Name	Pag
'https://www.mathworks.com'	0.0
'https://ch.mathworks.com'	0.0
'https://cn.mathworks.com'	0.0
'https://jp.mathworks.com'	0.0
'https://kr.mathworks.com'	0.0
'https://uk.mathworks.com'	0.0
'https://au.mathworks.com'	0.0
'https://de.mathworks.com'	0.0
'https://es.mathworks.com'	0.0
'https://fr.mathworks.com'	0.0
'https://in.mathworks.com'	0.0
'https://it.mathworks.com'	0.0
'https://nl.mathworks.com'	0.0
'https://se.mathworks.com'	0.0
'https://www.mathworks.com/index.html%3Fnocookie%3Dtrue'	0
'https://www.mathworks.com/company/aboutus/policies_statements/patents.html'	0.0
⋮	

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
```

5-69

Websites linked to https://www.mathworks.com
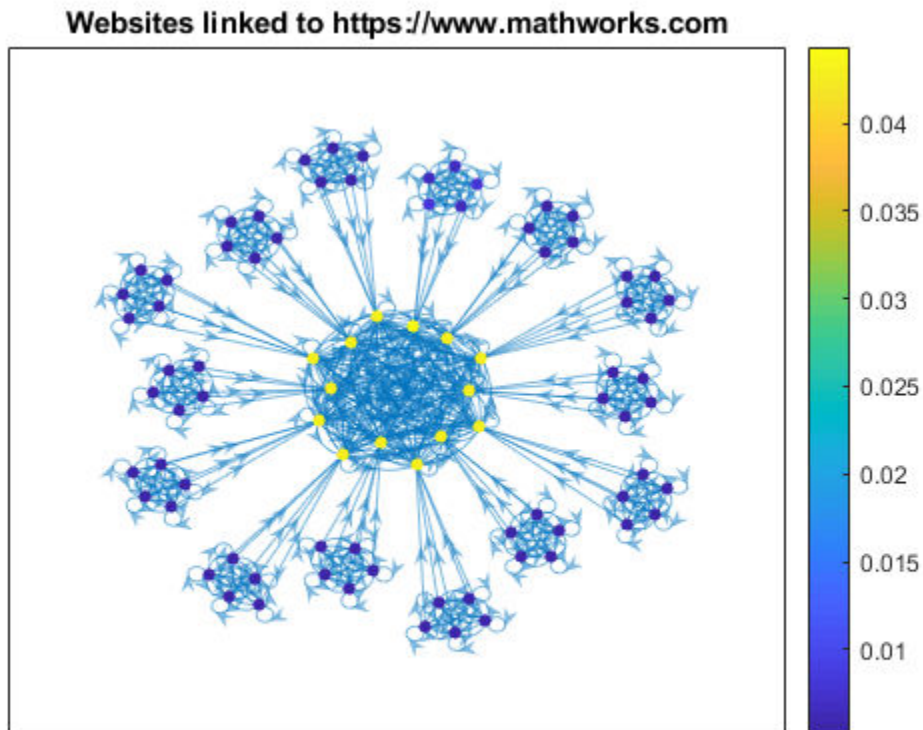
The PageRank scores for the top websites are all quite similar, such that a random web surfer has about a 4.5% chance to land on each page. This small group of highly connected pages forms a clique in the center of the plot. Connected to this central clique are several smaller cliques, which are highly 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);
```
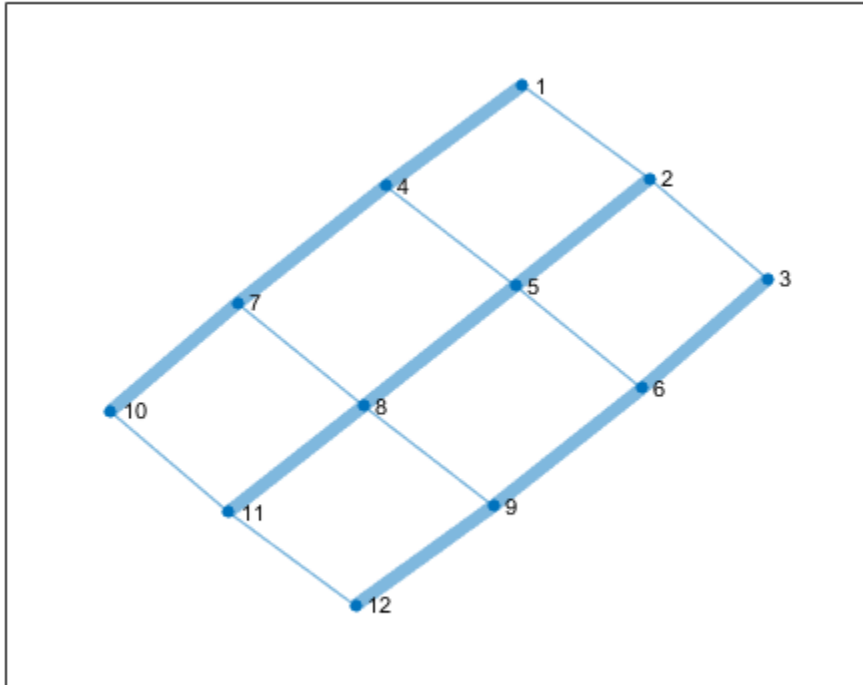
Add Node Labels

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')
```
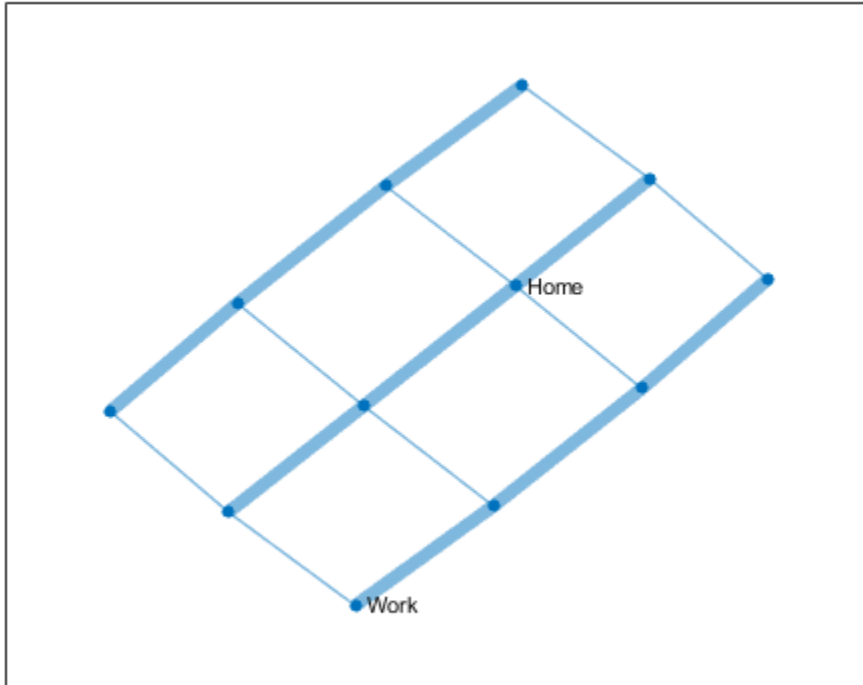
Add Edge Labels

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;
```
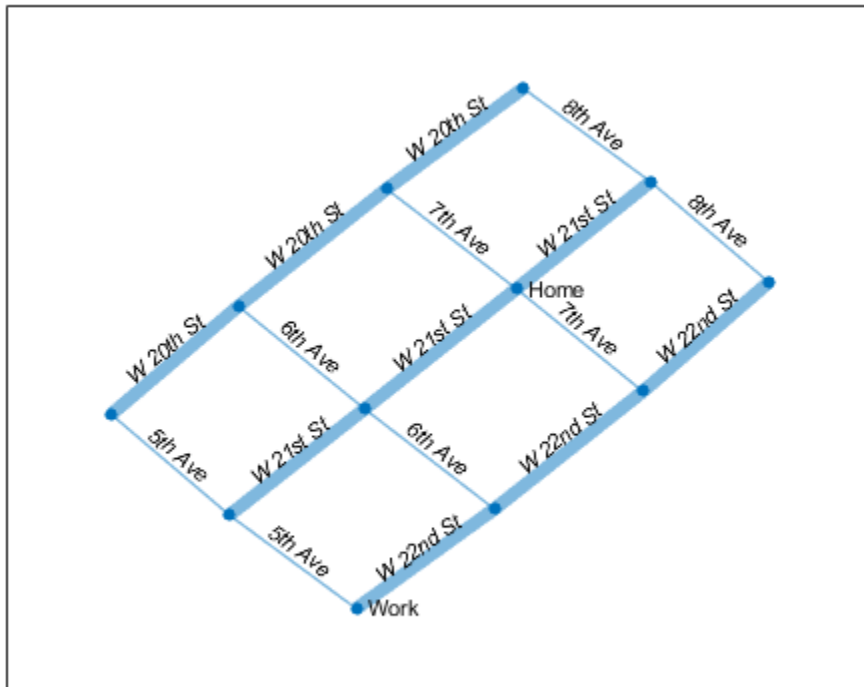
Adjust Font Properties

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';
```
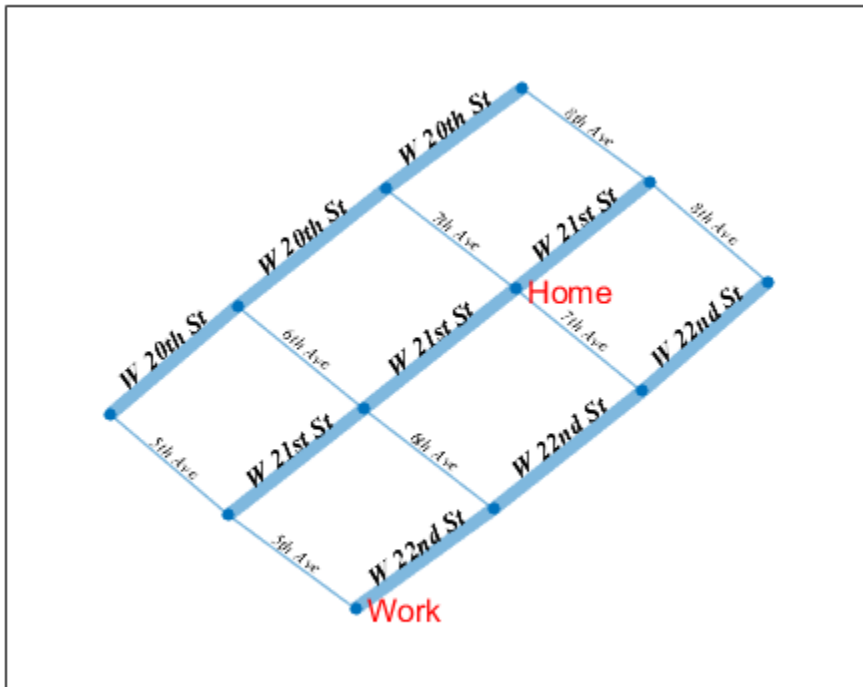

Highlight Edges

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 array
    {'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), '')
```
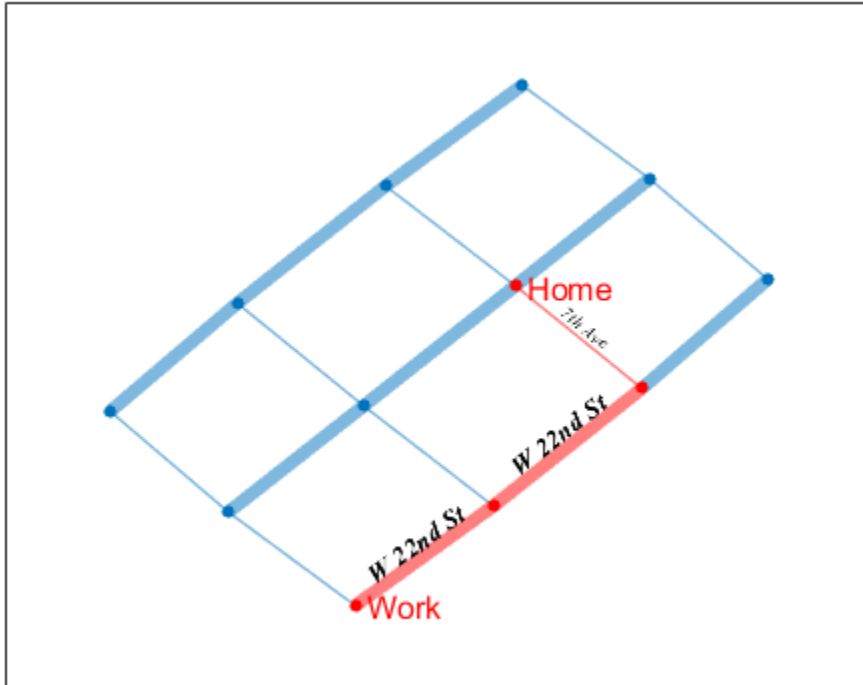
See Also

GraphPlot

More About

- "Graph Plotting and Customization" on page 5-23
- "Add Node Properties to Graph Plot Data Cursor" on page 5-49

6

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

$$p(x) = p_2 x^2 + p_1 x + p_0.$$

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 6-10
- "Polynomial Curve Fitting" on page 6-12

Roots of Polynomials

This example shows several different methods to calculate the roots of a polynomial.

In this section...
"Numeric Roots" on page 6-4
"Roots Using Substitution" on page 6-5
"Roots in a Specific Interval" on page 6-6
"Symbolic Roots" on page 6-8

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
```
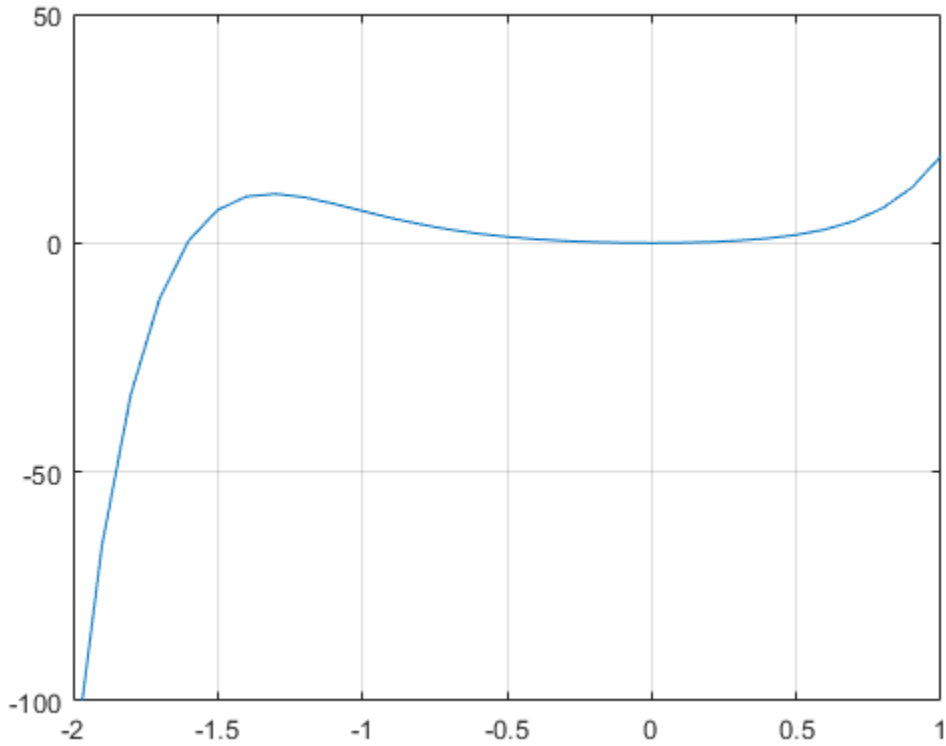
From the plot, the polynomial has a trivial root at `0` and another near `-1.5`. Use `fzero` to calculate and plot the root that is near `-1.5`.

```
Z = fzero(p, -1.5)
```

```
Z = -1.6056
```

```
plot(Z,p(Z),'r*')
```
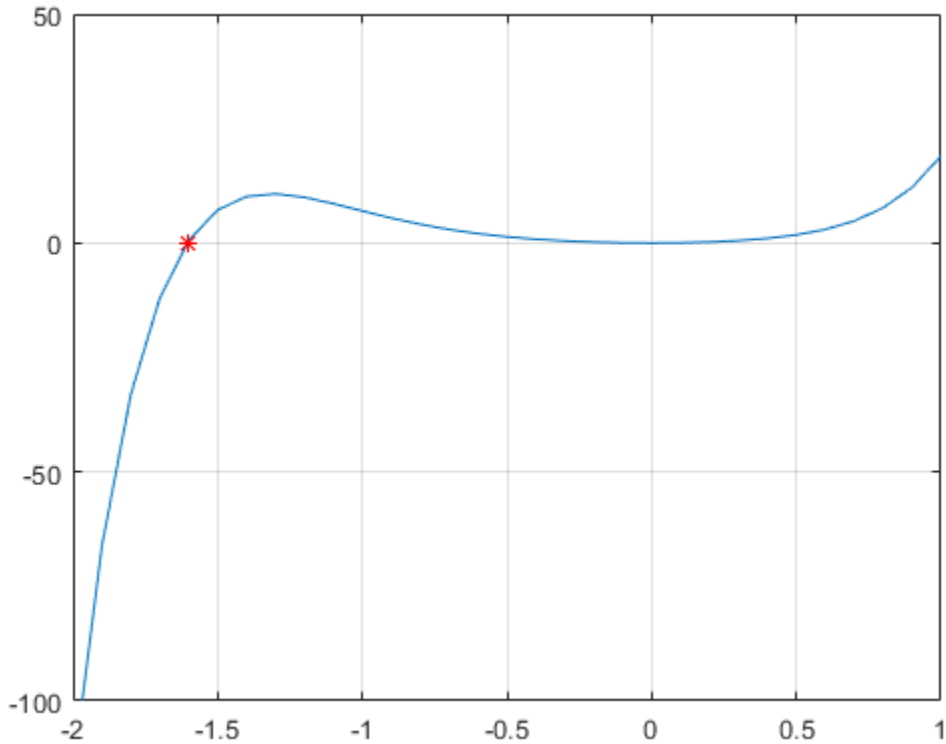
Symbolic Roots

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 6-21
- "Integrate and Differentiate Polynomials" on page 6-10

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 $\dfrac{d}{dx}\left[\dfrac{a(x)}{b(x)}\right]$ 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 15-9
*   "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)
```

$$y = (-0.2)\,x^3 + (31.6)\,x^2 + (-60.3)\,x + (35.3)$$
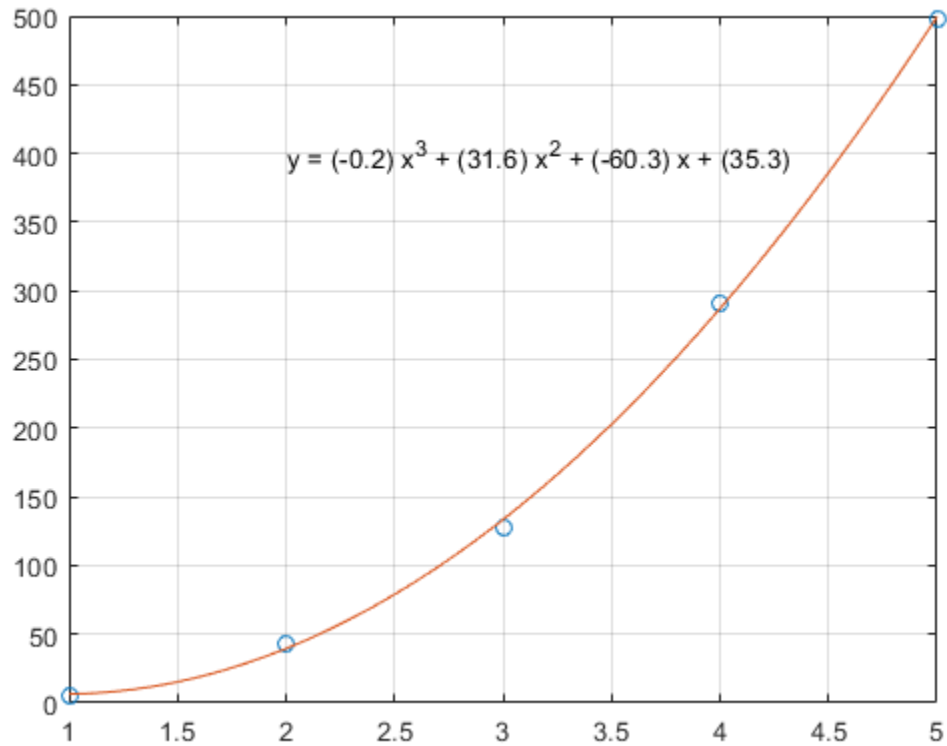
See Also
`polyfit` | `polyval`

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 1900 to 2000.

```matlab
% Time interval
t = (1900:10:2000)';

% Population
p = [75.995 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([1900 2020 0 400]);
title('Population of the U.S. 1900-2000');
ylabel('Millions');
```

Population of the U.S. 1900-2000
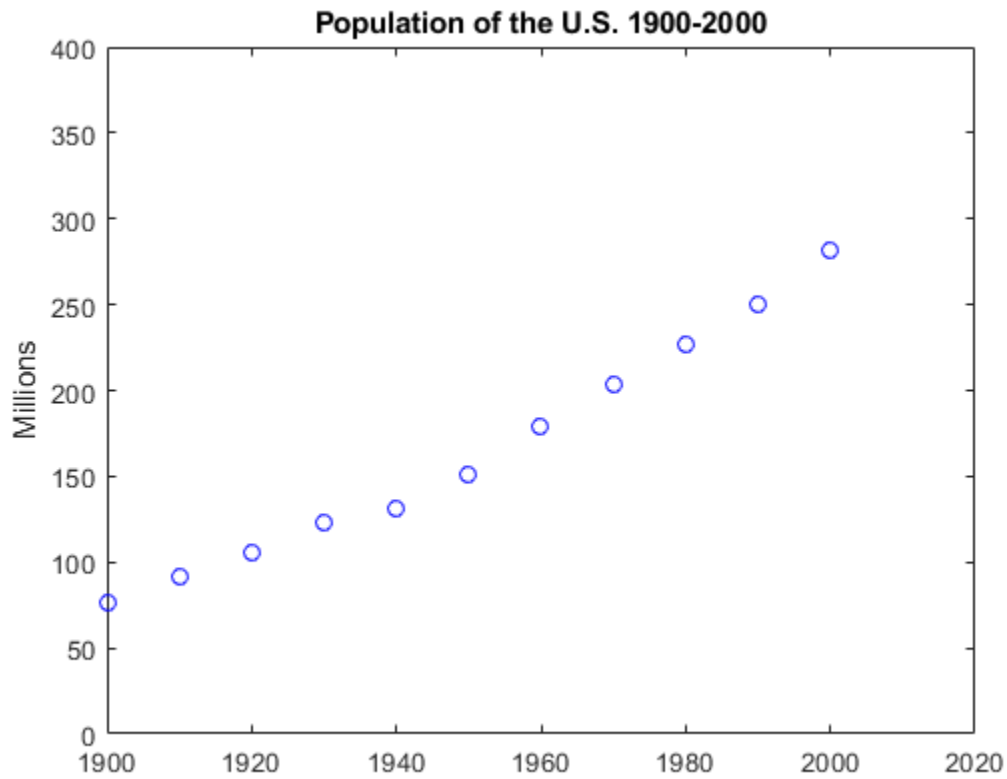
What is your guess for the population in the year 2010?

p

p = *11×1*

```
   75.9950
   91.9720
  105.7110
  123.2030
  131.6690
  150.6970
  179.3230
  203.2120
```

```
  226.5050
  249.6330
      ⋮
```

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*

```
    1.2629
   23.7261
  100.3659
  155.9043
```

Now evaluate the polynomial at every year from 1900 to 2010 and plot the results.

```
v = (1900:2020)';
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
```


Population of the U.S. 1900-2000
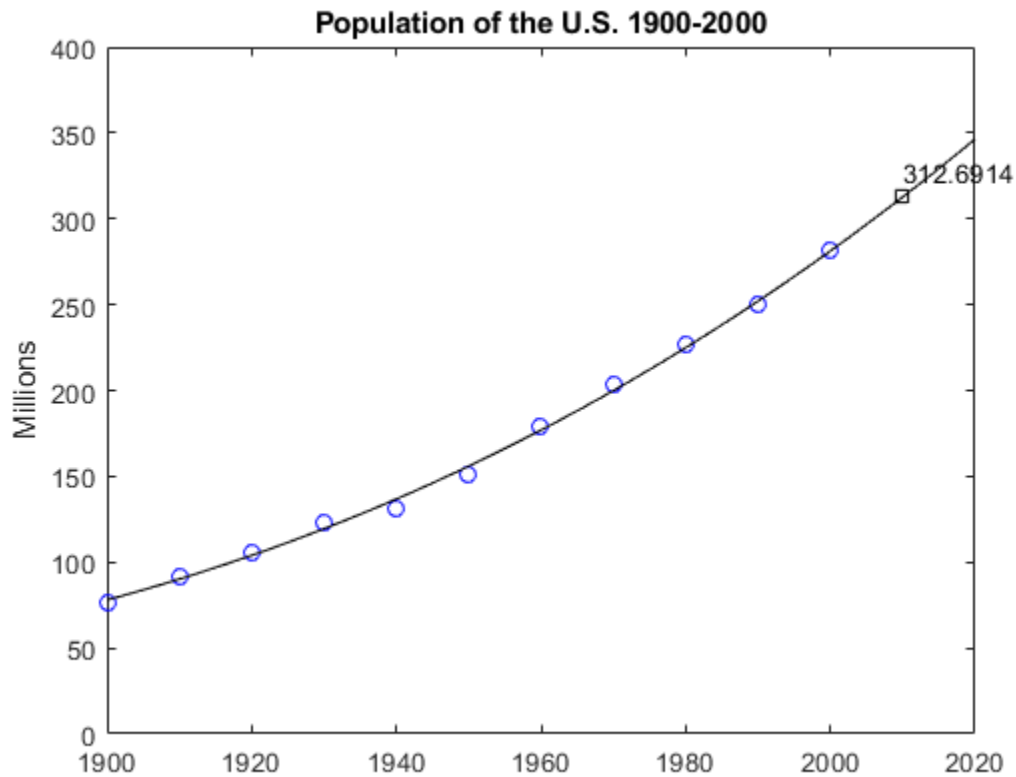
Compare the cubic fit with the quartic. Notice that the extrapolated point is very different.

```
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
```

Population of the U.S. 1900-2000
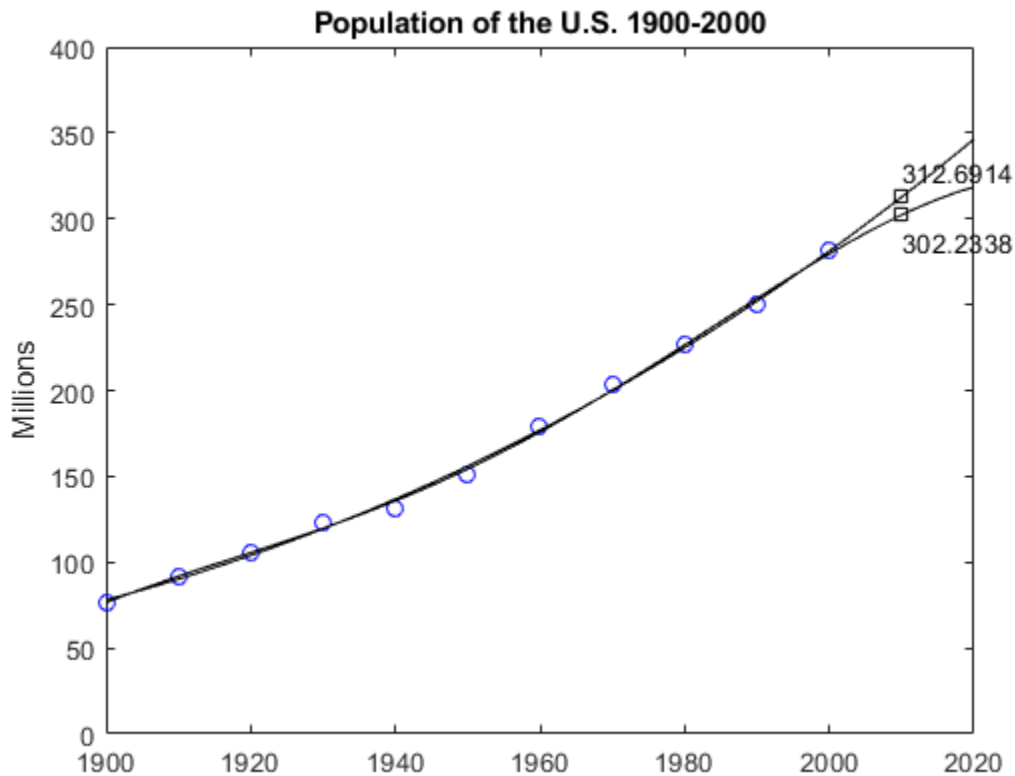
As the degree increases, the extrapolation becomes even more erratic.

```
cla
plot(t,p,'bo')
```

```matlab
hold on
axis([1900 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
```
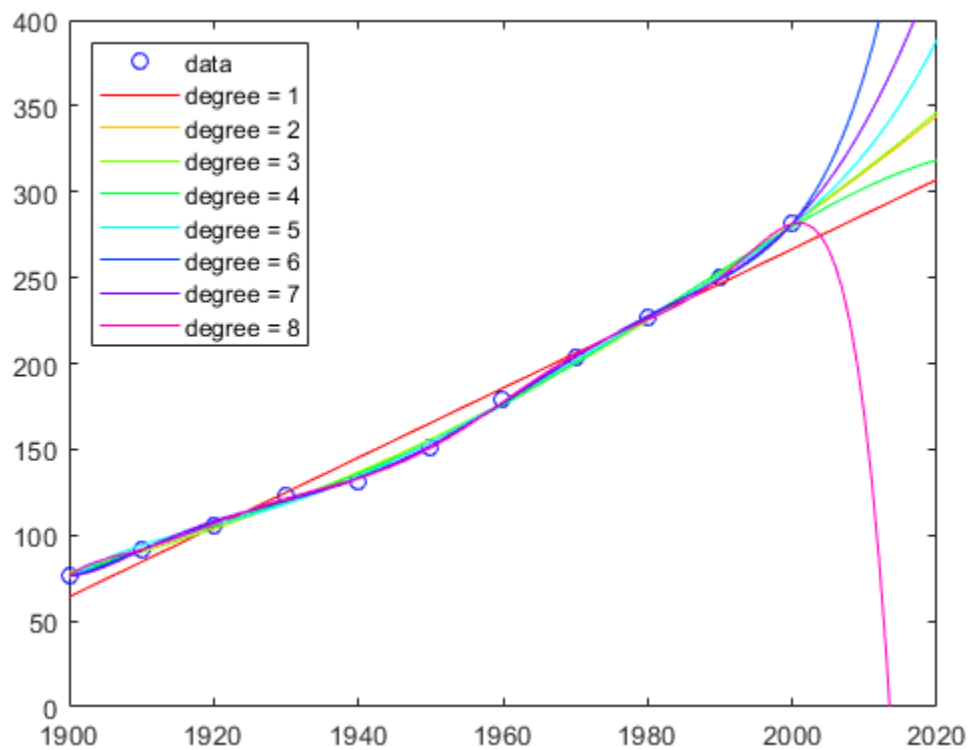
See Also

```
polyfit
```

Roots of Scalar Functions

In this section...

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
```
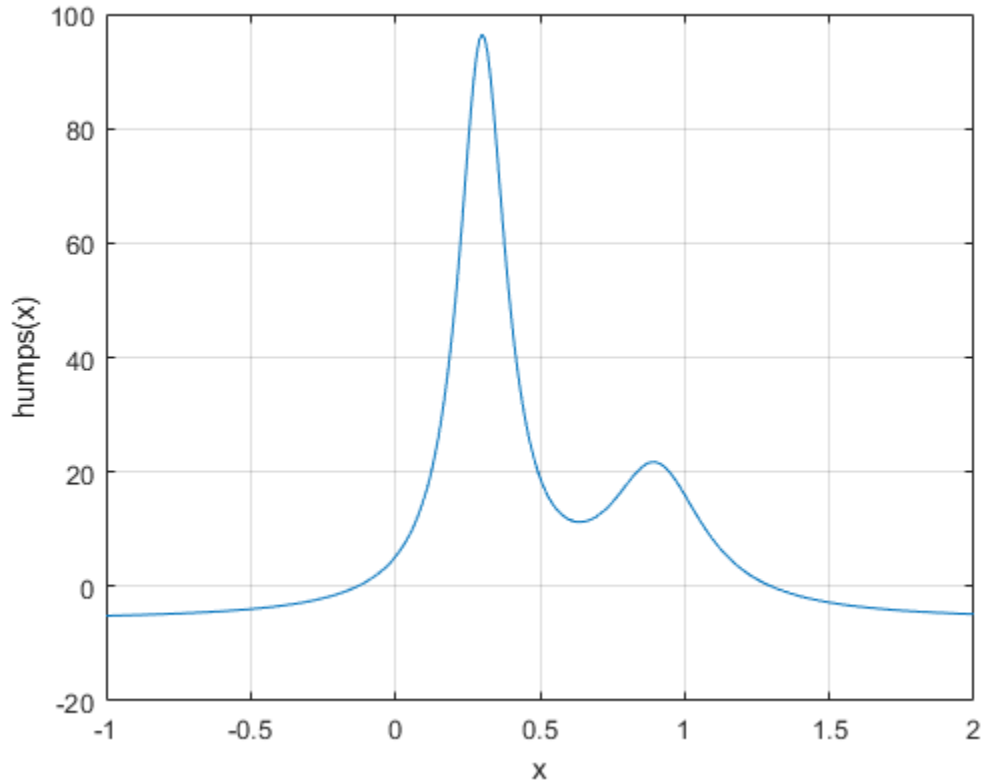
Setting Options For fzero

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 Options" on page 9-11, "Using a Starting Interval" on page 6-23, and "Using a Starting Point" on page 6-24.

- Choosing various tolerances that control how `fzero` determines it is at a root — see "Set Options" on page 9-11.

- Choosing a plot function for observing the progress of `fzero` towards a root — see "Plot Functions" on page 9-25.

- Using a custom-programmed output function for observing the progress of `fzero` towards a root — see "Output Functions" on page 9-17.

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 the endpoint, -0.10949, but negative at the endpoint, -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)

7

Computational Geometry

Overview

MATLAB provides a variety of tools to solve problems that are geometric in nature. These problems encompass simple geometric queries such as point-in-polygon tests, to more complex problems such as nearest-neighbor queries. Convex hulls, Delaunay triangulations, and Voronoi diagrams are examples of fundamental geometric algorithms that MATLAB provides. These algorithms have many practical uses in the scientific development.

MATLAB provides both function-based and class-based tools for computational geometry. The function-based tools are convenient when the output from the function is sufficiently complete to get you to the solution. The class-based tools are more versatile because they provide complementary methods that are often needed to work with the resulting data. A function typically takes input, performs some computation, and outputs the result in matrix format.

In contrast, a class may take input, perform some computation, cache the result and provide methods that allow you to query and work with the result more efficiently. For example, the `delaunay` function creates a triangulation from a set of points and returns the triangulation in an array. The `delaunayTriangulation` class also triangulates a set of points. In addition, it allows you to modify the triangulation and it provides methods for finding nearest neighbors and traversing the triangulation data structure, etc. These methods are very efficient because the class holds the triangulation in memory.

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
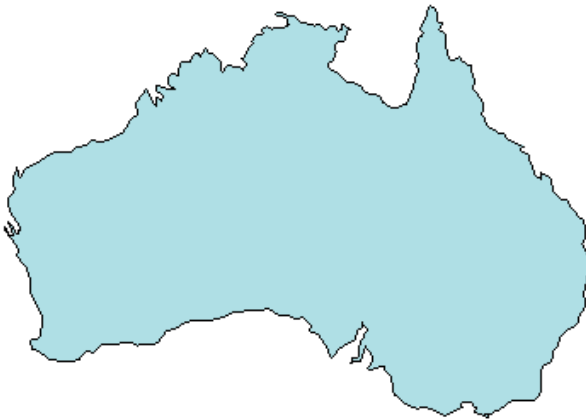
can be represented by the triangulation on the map shown below.
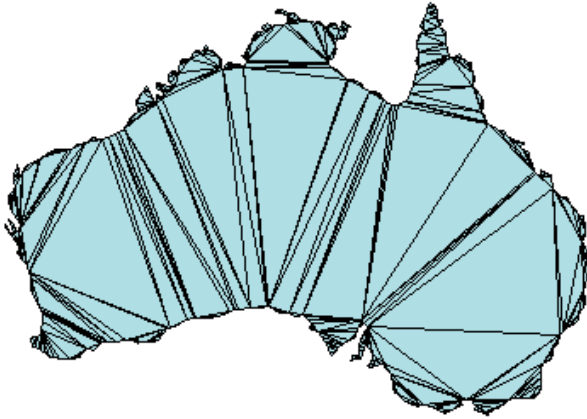
The triangulation decomposes a complex polygon into a collection of simpler triangular polygons. You can use these polygons for developing geometric-based algorithms or graphics applications.

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.
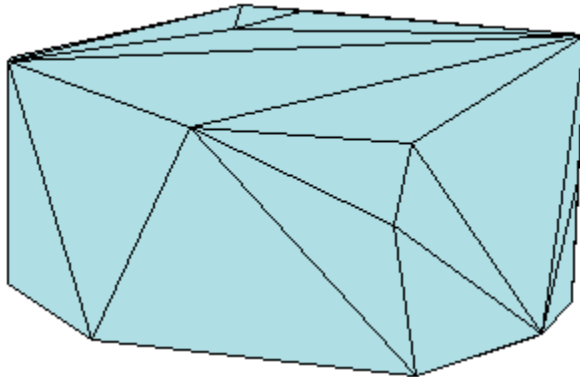
Triangulation Matrix Format

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 tetrahedon.

This figure shows a simple 2-D triangulation.
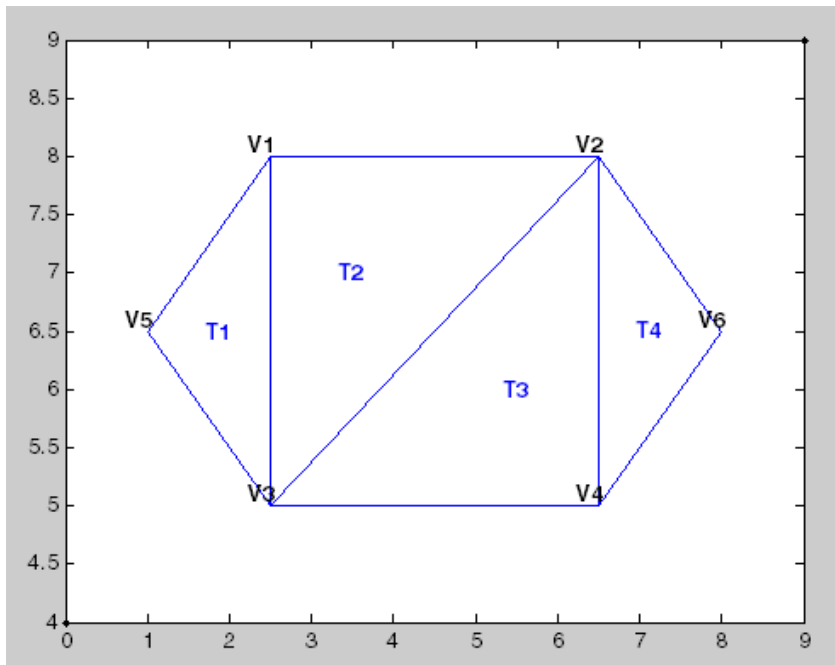
The following table shows the vertex information.

Vertices		
Vertex ID	x-coordinate	y-coordinate
V1	2.5	8.0
V2	6.5	8.0
V3	2.5	5.0
V4	6.5	5.0

Vertices		
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.
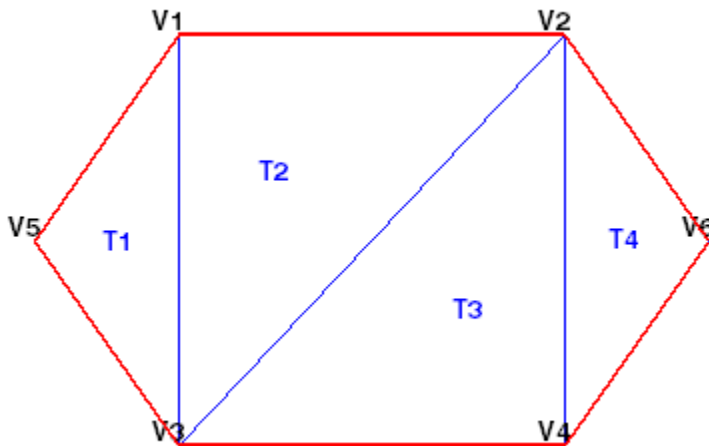
The triangulation Class

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
```
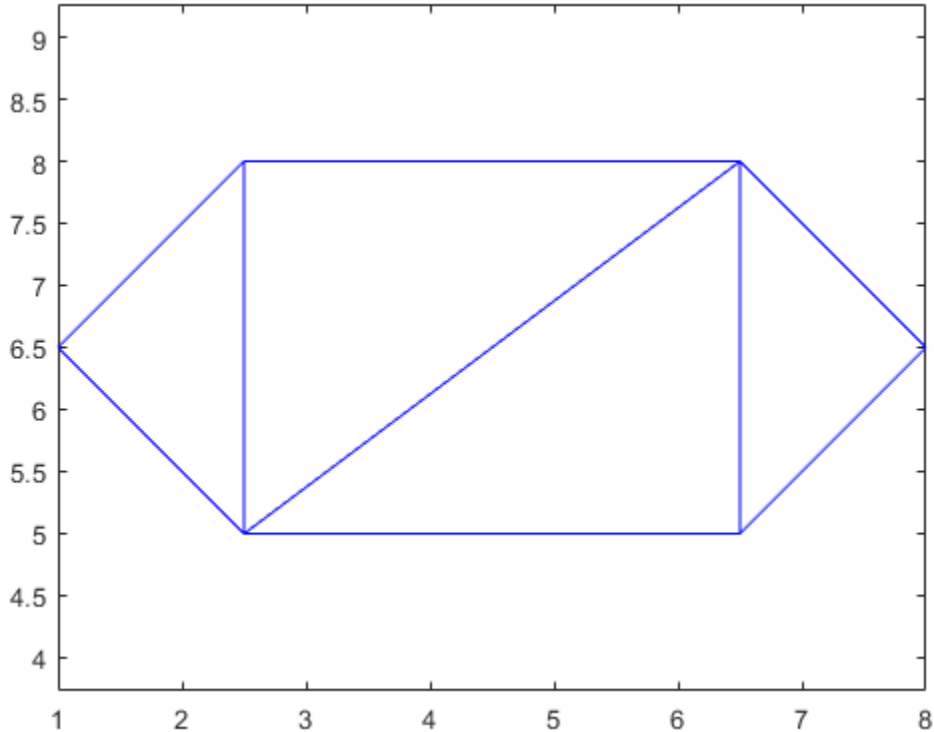
Use the `triangulation` method, `freeBoundary`, to query the free boundary and highlight it in a plot. This method returns the edges of the triangulation that are shared by only one triangle. The returned edges are expressed in terms of the vertex IDs.

```
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
```

7-11
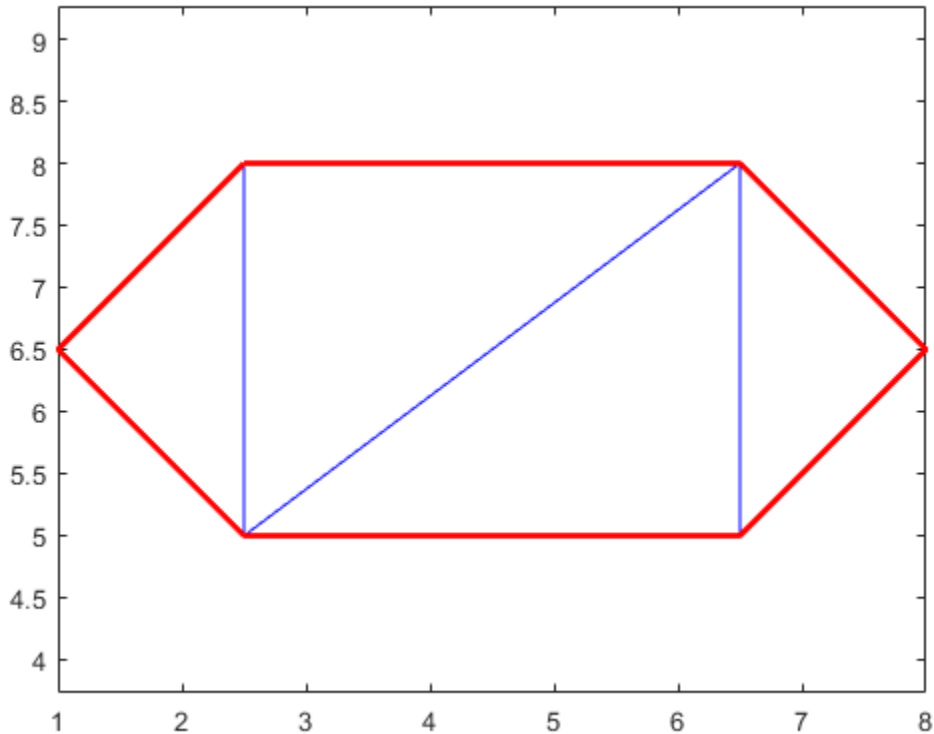
You can use the `freeBoundary` method to validate a triangulation. For example, if you observed red edges in the interior of the triangulation, then it would indicate a problem in how the triangles are connected.

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
```
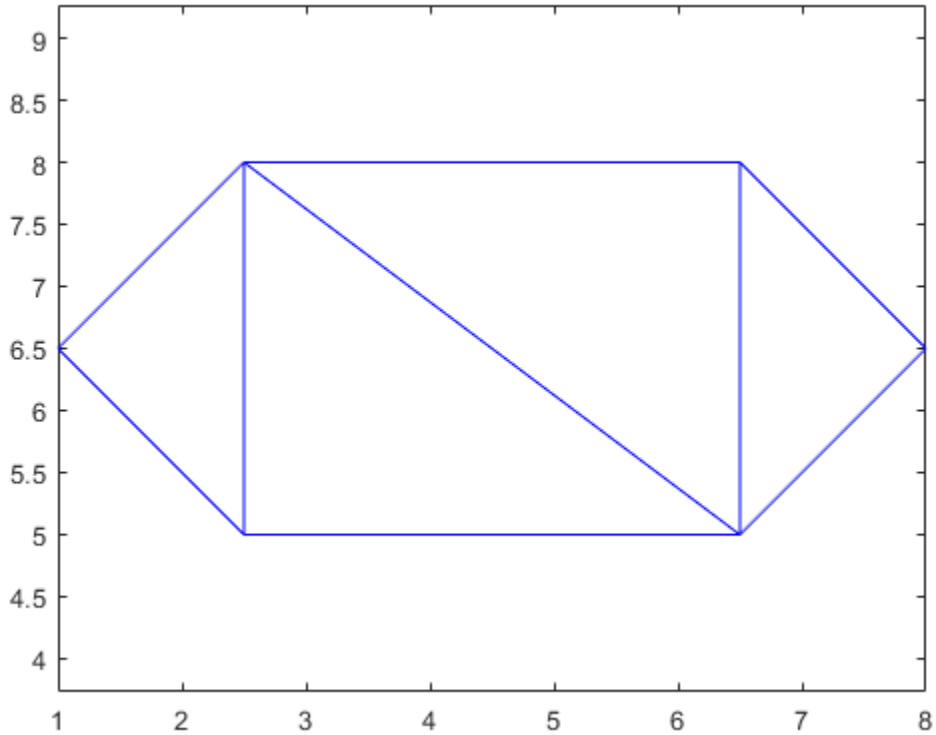

The parent class, `triangulation`, provides the `incenter` method to compute the incenters of each triangle.

```
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
```
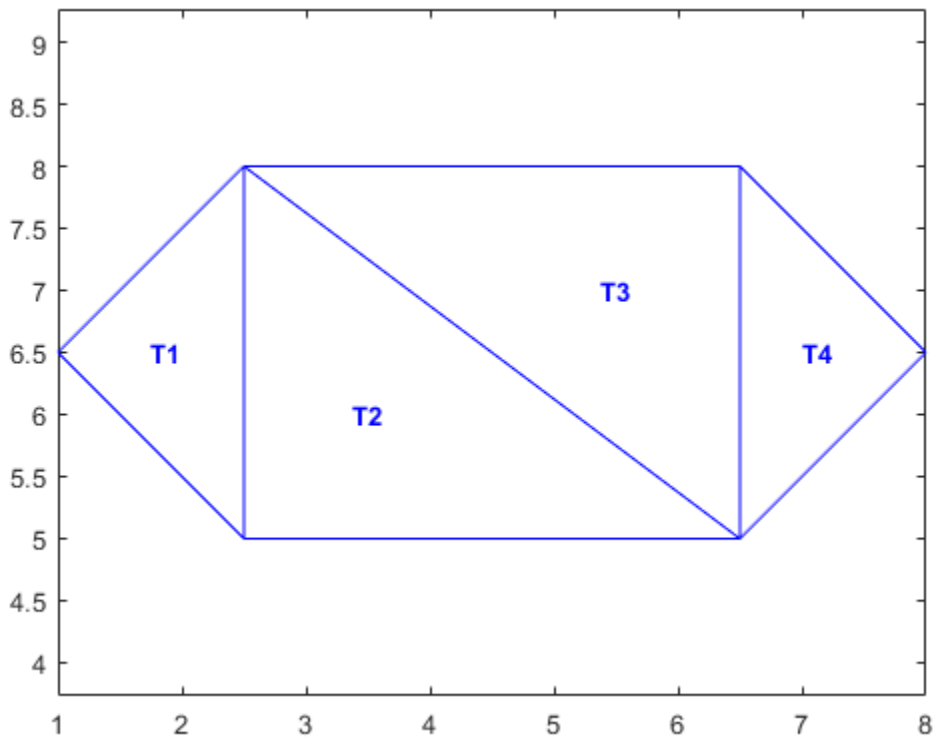
Instead of creating a Delaunay triangulation using `delaunayTriangulation`, you could use the `delaunay` function to create the triangulation connectivity data, and then pass the connectivity data to `triangulation`. For example,

```
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 7-3
- "Delaunay Triangulation" on page 7-17
- "Spatial Searching" on page 7-50

Delaunay Triangulation

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.
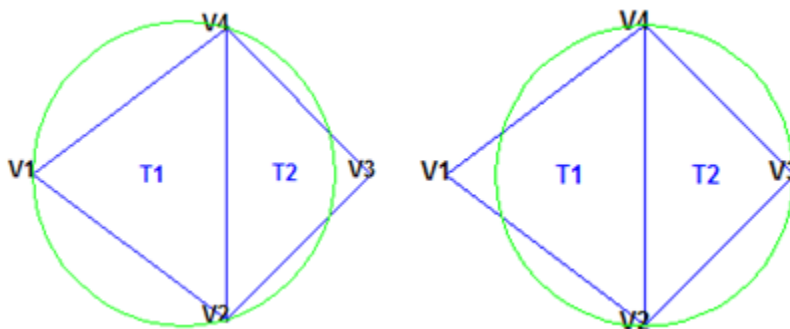
The triangles below are different. The circumcircle associated with T1 is not empty. It contains V3 in its interior. The circumcircle associated with T2 is not empty. It contains V1 in its interior. This triangulation is *not* a Delaunay triangulation.
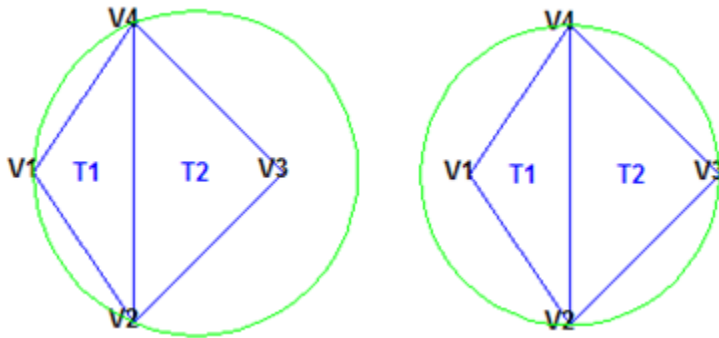
Delaunay triangles are said to be "well shaped" because in fulfilling the empty circumcircle property, triangles with large internal angles are selected over ones with small internal angles. The triangles in the non-Delaunay triangulation have sharp angles at vertices V2 and V4. If the edge {V2, V4} 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.
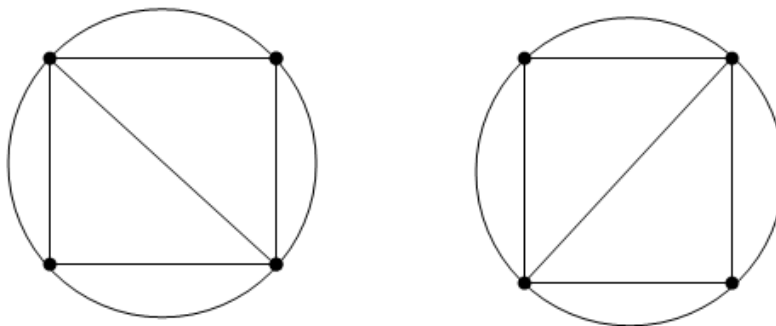
The properties of Delaunay triangulations extend to higher dimensions. The triangulation of a 3-D set of points is composed of tetrahedra. The next illustration shows a simple 3-D Delaunay triangulation made up of two tetrahedra. The circumsphere of one tetrahedron is shown to highlight the empty circumsphere criterion.
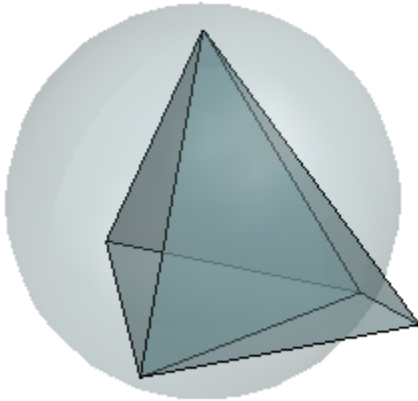
A 3-D Delaunay triangulation produces tetrahedra that satisfy the empty circumsphere criterion.

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 7-4 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
```
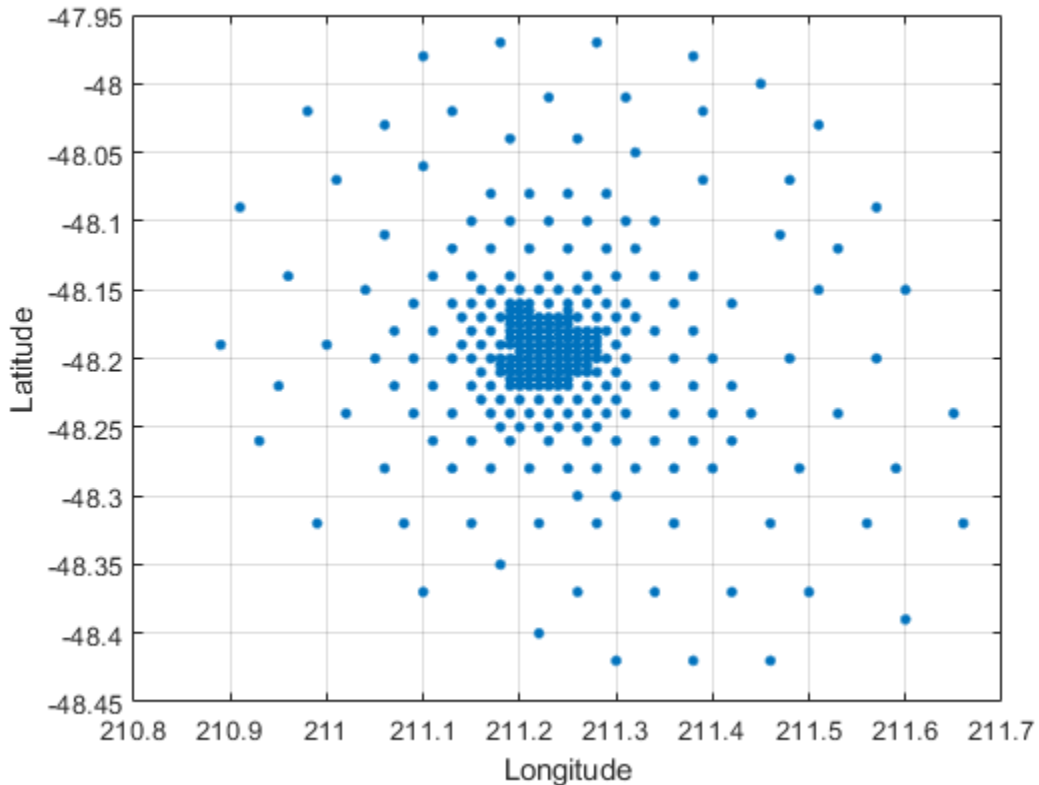

Construct a Delaunay triangulation from this point set and use `triplot` to plot the triangulation in the existing figure.

```
tri = delaunay(x,y);
hold on, triplot(tri,x,y), hold off
```
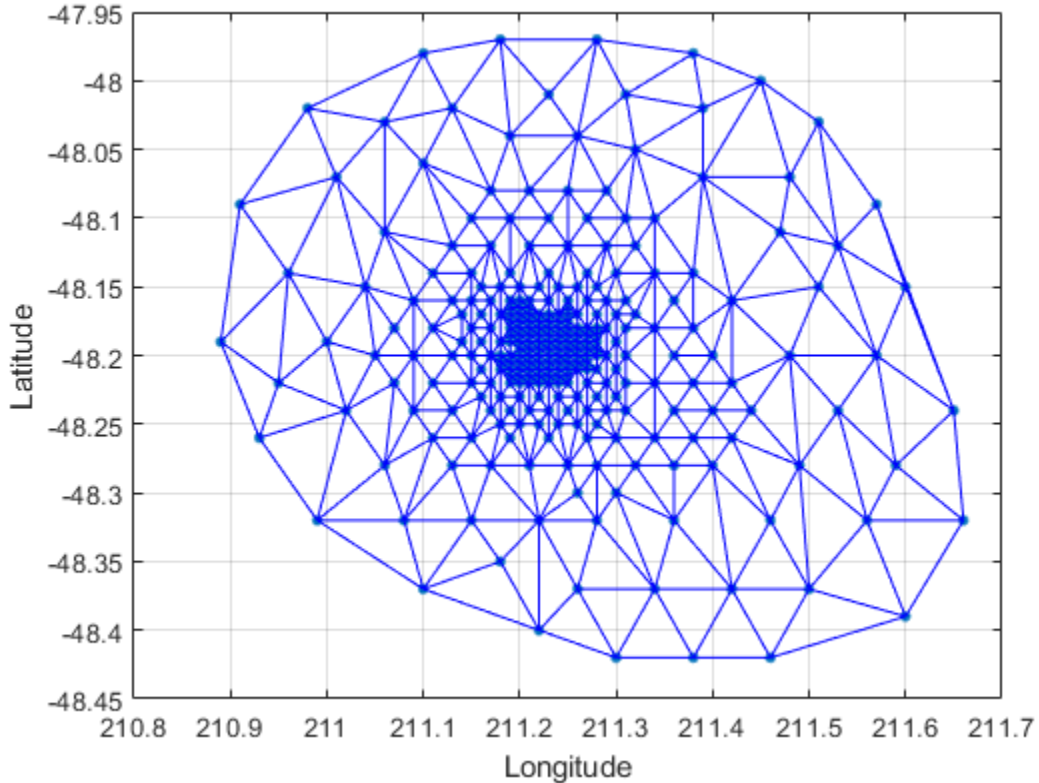

Add the depth data (z) from seamount to lift the vertices and create the surface. Create a new figure and use `trimesh` to plot the surface in wireframe mode.

```
figure
hidden on
trimesh(tri,x,y,z)
xlabel('Longitude'),ylabel('Latitude'),zlabel('Depth in Feet');
```
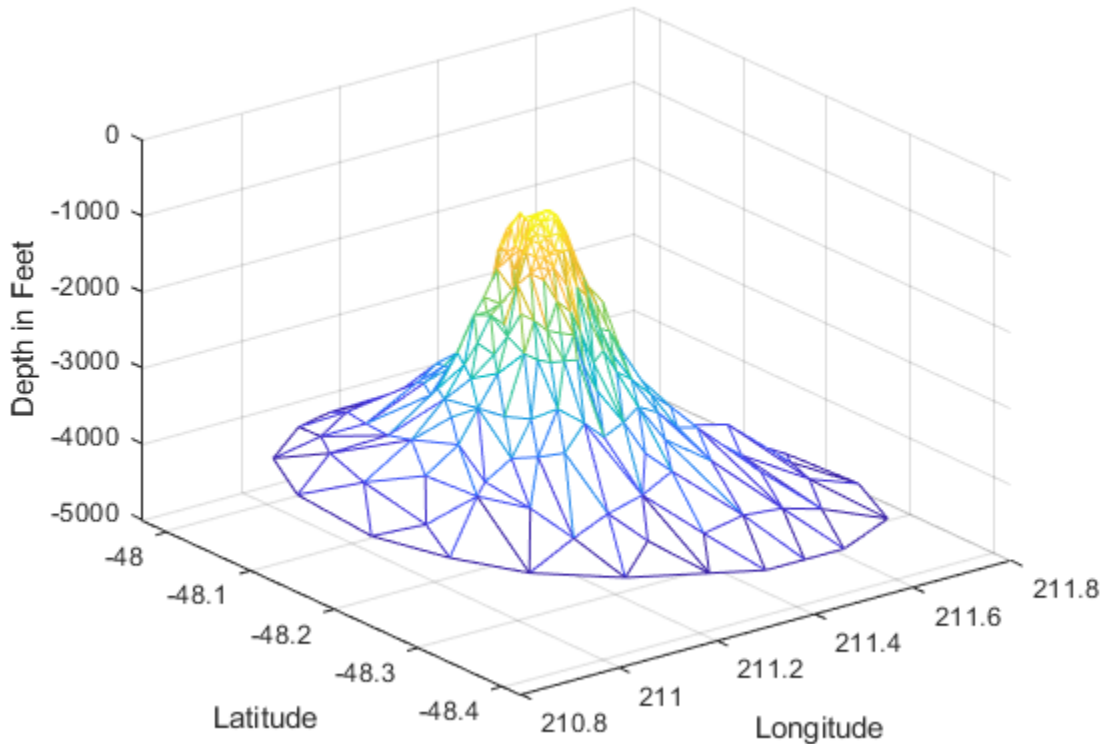
If you want to plot the surface in shaded mode, use `trisurf` instead of `trimesh`.

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.

```
X = gallery('uniformdata',[30 3],0);
tet = delaunay(X);
faceColor  = [0.6875 0.8750 0.8984];
tetramesh(tet,X,'FaceColor', faceColor,'FaceAlpha',0.3);
```

7-23
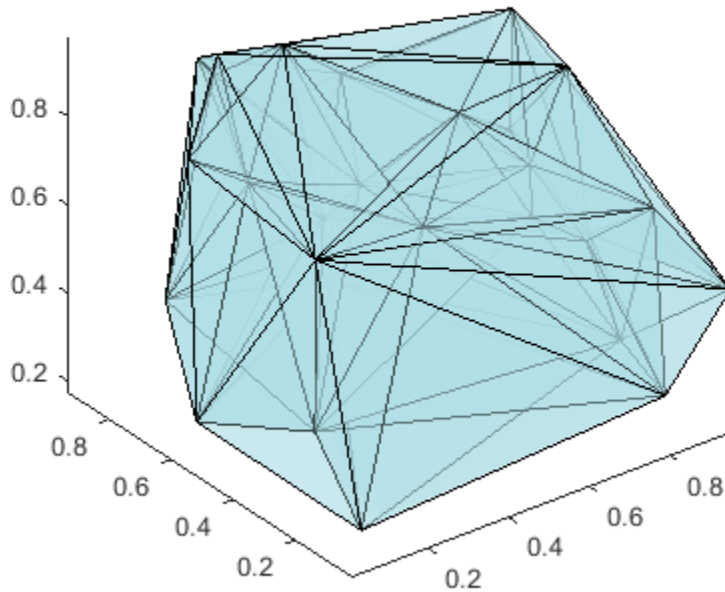
MATLAB provides the `delaunayn` function to support the creation of Delaunay triangulations in dimension 4-D and higher. Two complementary functions `tsearchn` and `dsearchn` are also provided to support spatial searching for N-D triangulations. See "Spatial Searching" on page 7-50 for more information 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 7-3 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
```
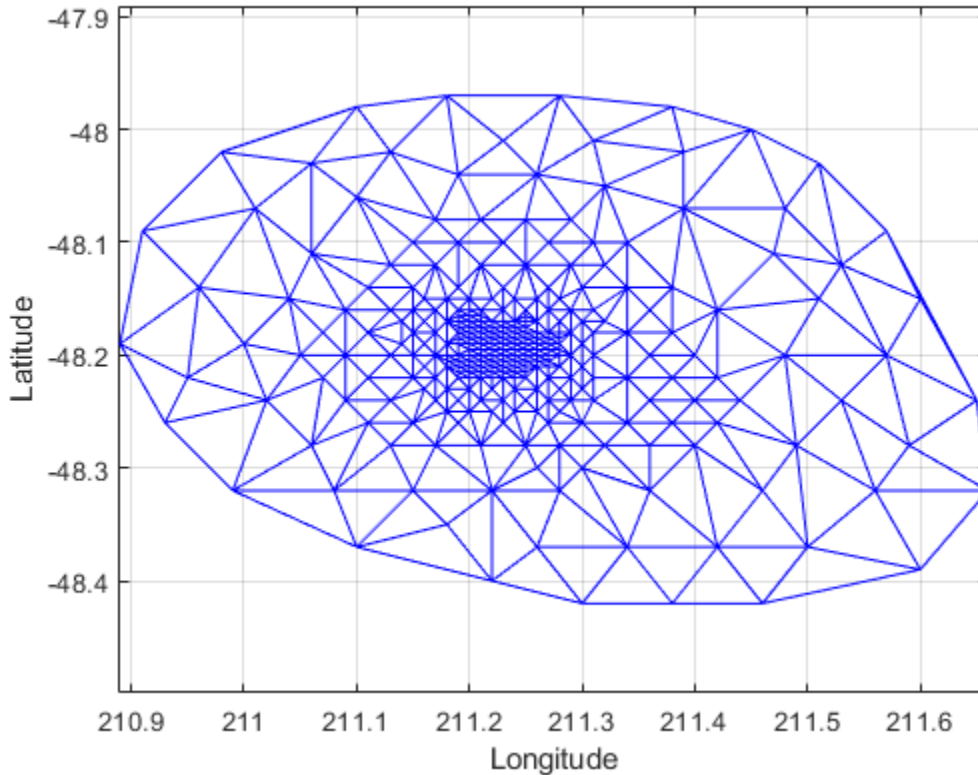
Alternatively, you could use `triplot(DT(:,:), DT.Points(:,1), DT.Points(:, 2));` to get the same plot.

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
```
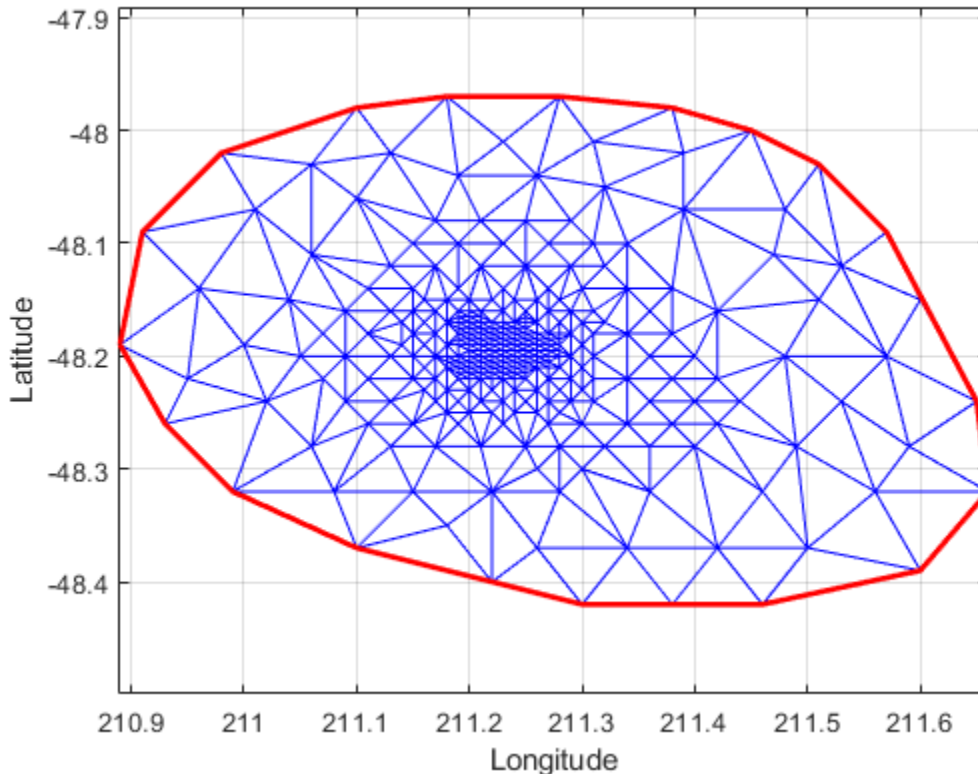
You can incrementally edit the `delaunayTriangulation` to add or remove points. If you need to add points to an existing triangulation, then an incremental addition is faster than a complete retriangulation of the augmented point set. Incremental removal of points is more efficient when the number of 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
```
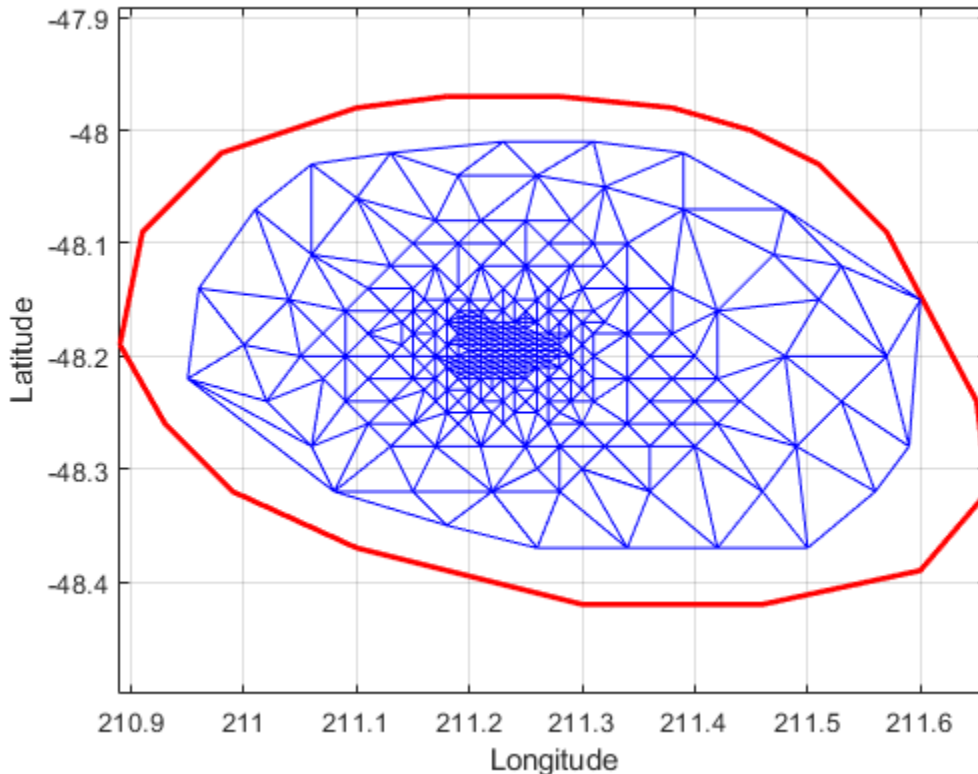
There is one vertex that is just inside the boundary of the convex hull that was not removed. The fact that it is interior to the hull can be seen using the Zoom-In tool in the figure. You could plot the vertex labels to determine the index of this vertex and remove it from the triangulation. Alternatively, 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
```
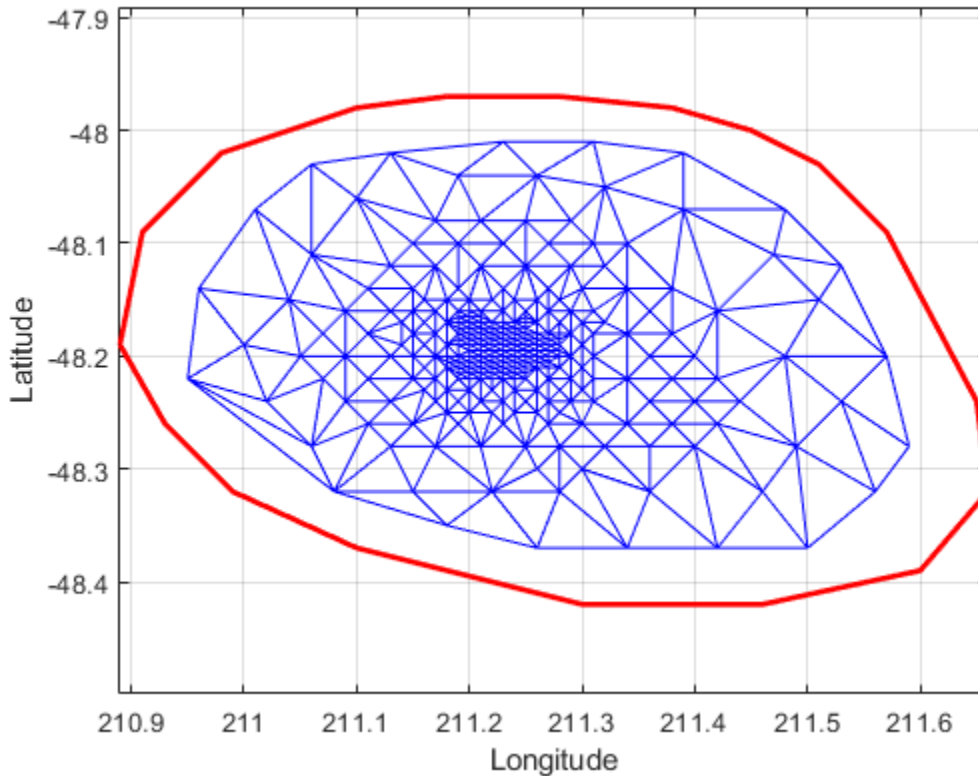
Add points to the existing triangulation. Add 4 points to form a rectangle around the triangulation.

```
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
```
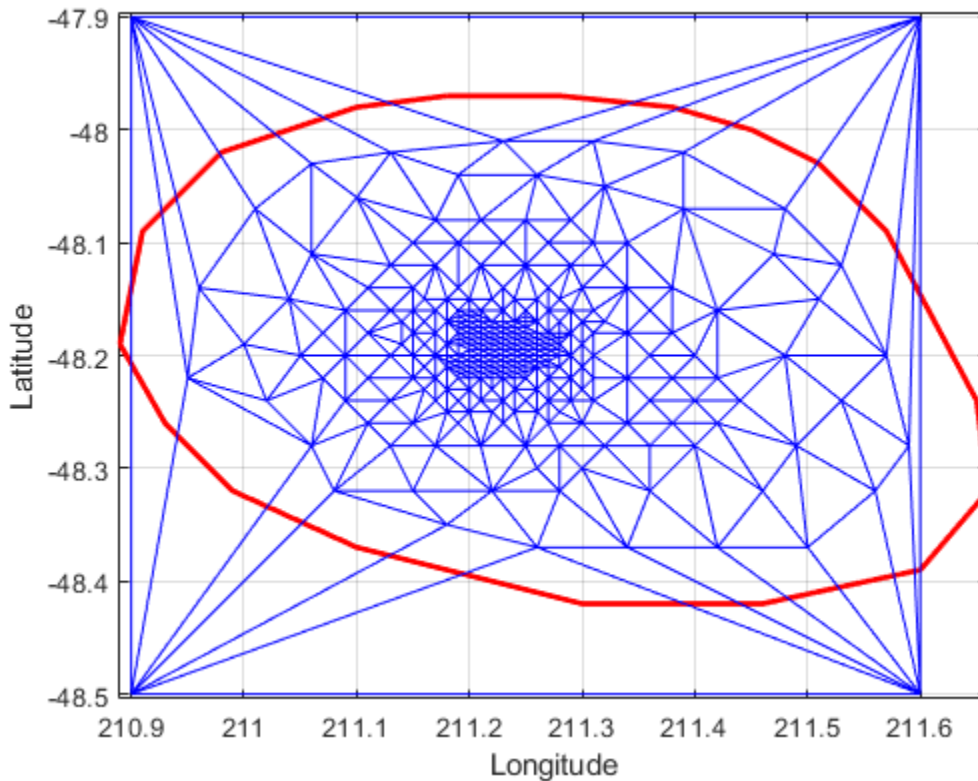
You can edit the points in the triangulation to move them to a new location. Edit the first of the additional point set (the vertex ID 274).

```
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
```
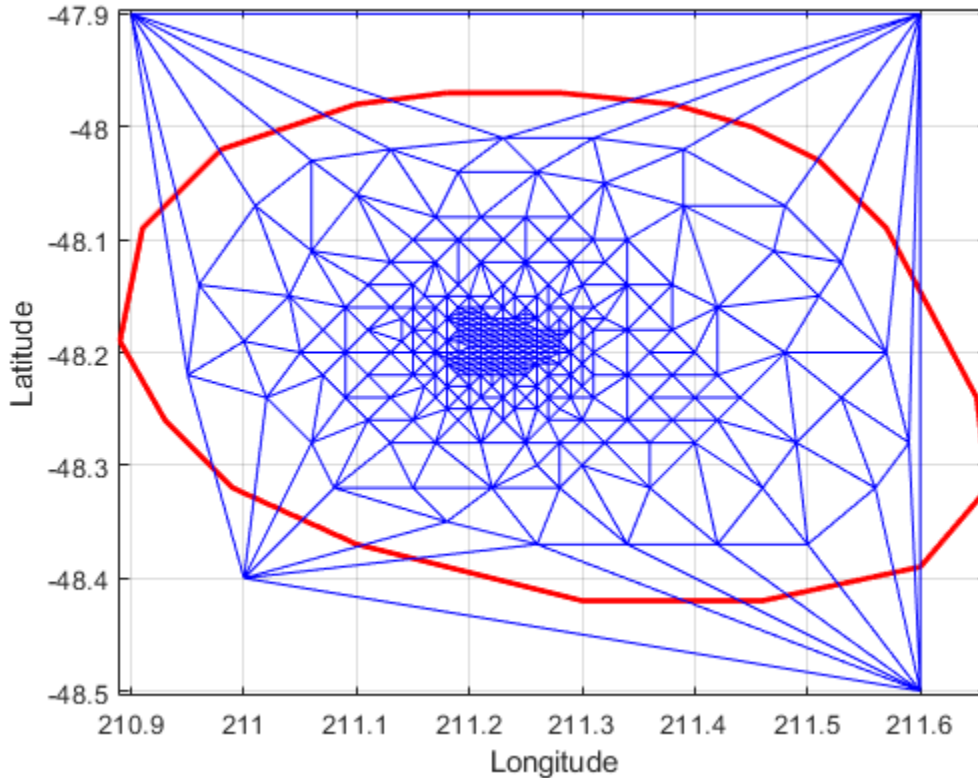
Use the a method of the `triangulation` class, `vertexAttachments`, to find the attached triangles. Since the number of triangles attached to a vertex is variable, the method returns the attached triangle IDs in a cell array. You need braces to extract the contents.

```
attTris = vertexAttachments(DT,274);
hold on
triplot(DT(attTris{:},:),DT.Points(:,1),DT.Points(:,2),'g')
hold off
```
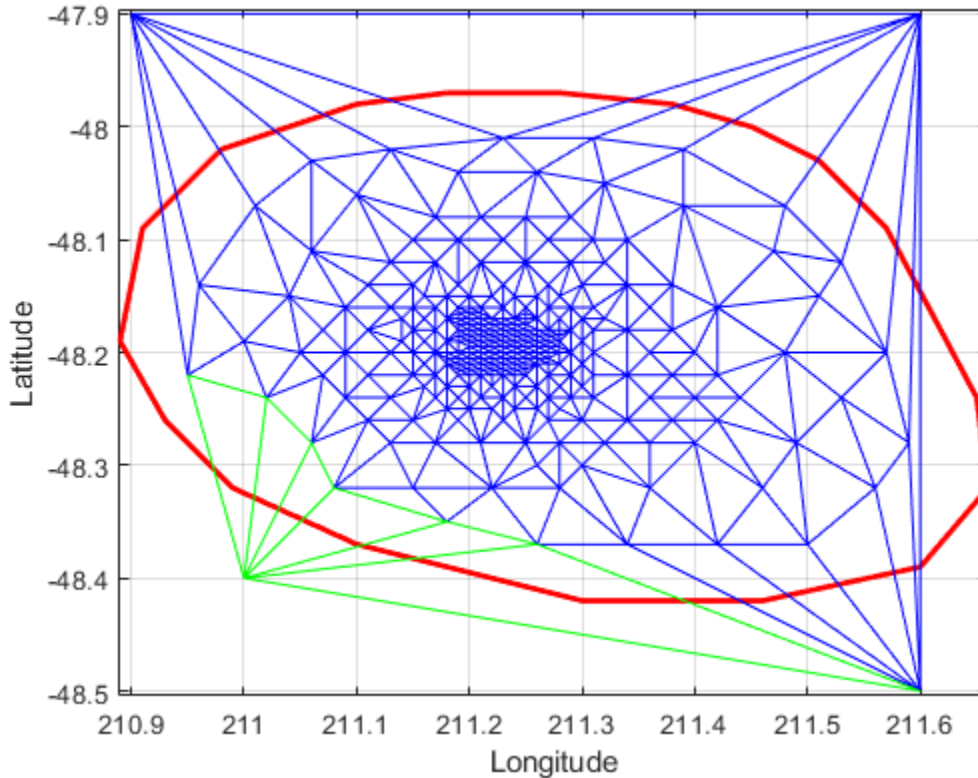
`delaunayTriangulation` also can be used to triangulate points in 3-D space. The resulting triangulation is composed of tetrahedra.

This example shows how to use a `delaunayTriangulation` to create and plot the triangulation of 3-D points.

```
P = gallery('uniformdata',30,3,0);
DT = delaunayTriangulation(P)

DT =
  delaunayTriangulation with properties:

             Points: [30x3 double]
    ConnectivityList: [117x4 double]
```

```
        Constraints: []


faceColor  = [0.6875 0.8750 0.8984];
tetramesh(DT,'FaceColor', faceColor,'FaceAlpha',0.3);
```
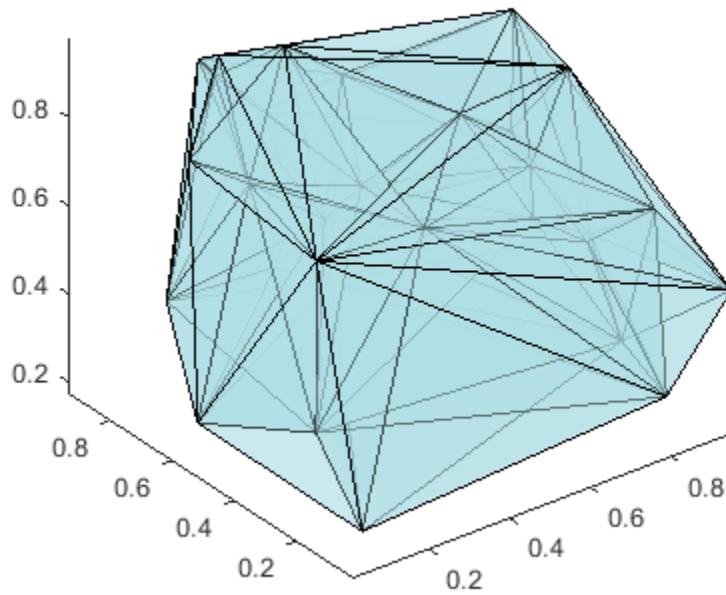

The `tetramesh` function plots both the internal and external faces of the triangulation. For large 3-D triangulations, plotting the internal faces might be an unnecessary use of resources. A plot of the boundary might be more appropriate. You can use the `freeBoundary` method to get the boundary triangulation in matrix format. Then pass the result to `trimesh` or `trisurf`.

7-37

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.
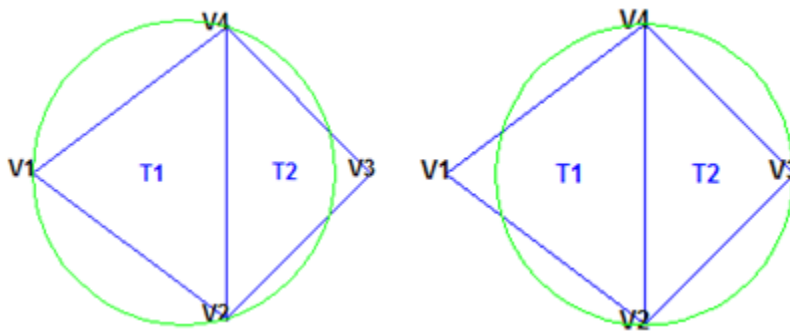
Triangulate a set of points with an edge constraint specified between vertex V1 and V3.

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
```
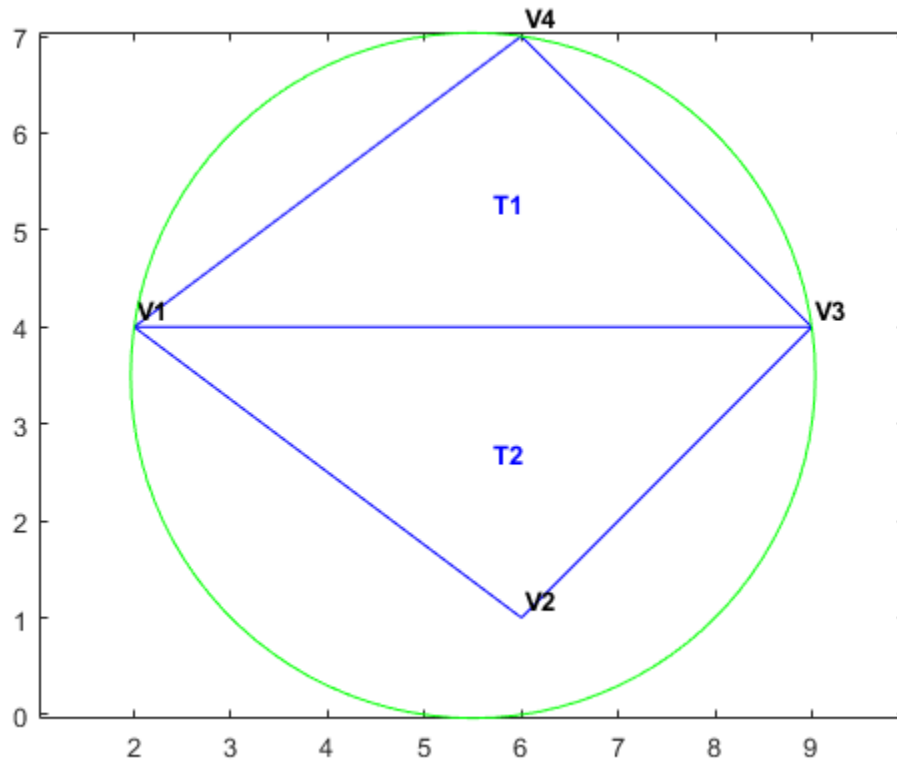
The constraint between vertices (`V1`, `V3`) was honored, however, the Delaunay criterion was invalidated. This also invalidates the nearest-neighbor relation that is inherent in a Delaunay triangulation. This means the `nearestNeighbor` search method provided by `delaunayTriangulation` cannot be supported 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
```
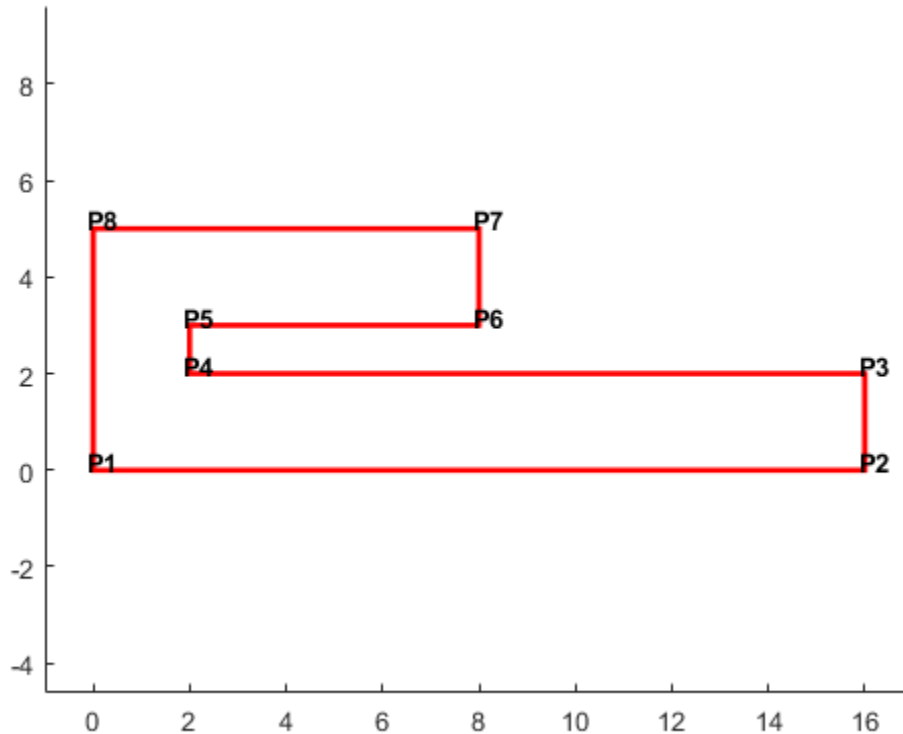
Create and plot the triangulation together with the polygon boundary.

```
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)
```
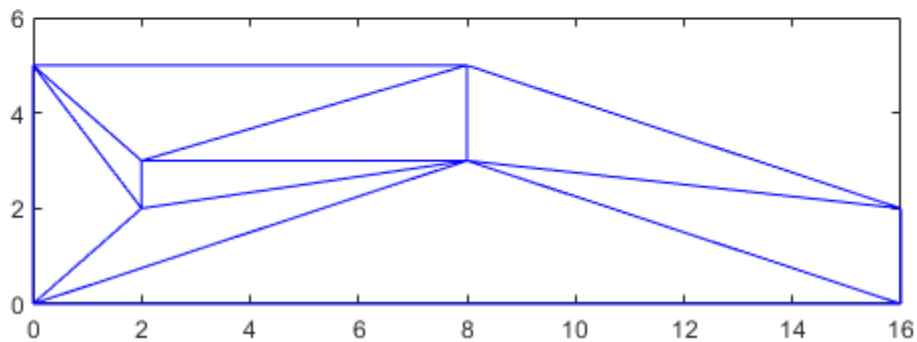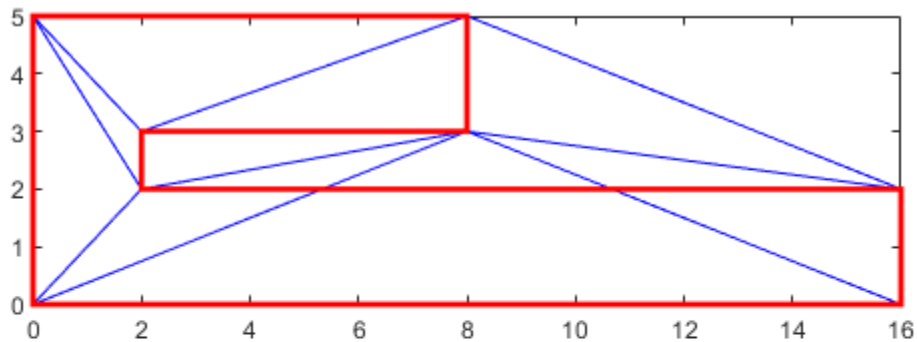

This triangulation cannot be used to represent the domain of the polygon because some triangles cut across the boundary. You need to impose a constraint on the edges that are cut by triangulation edges. Since all edges have to be respected, you need to constrain all edges. The steps below show how to 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)
```
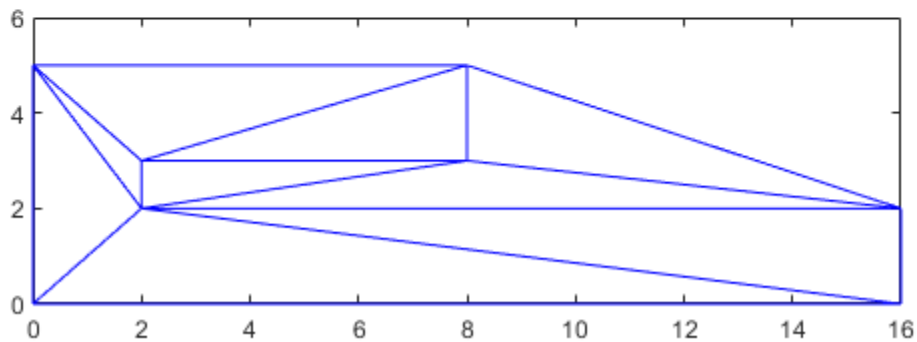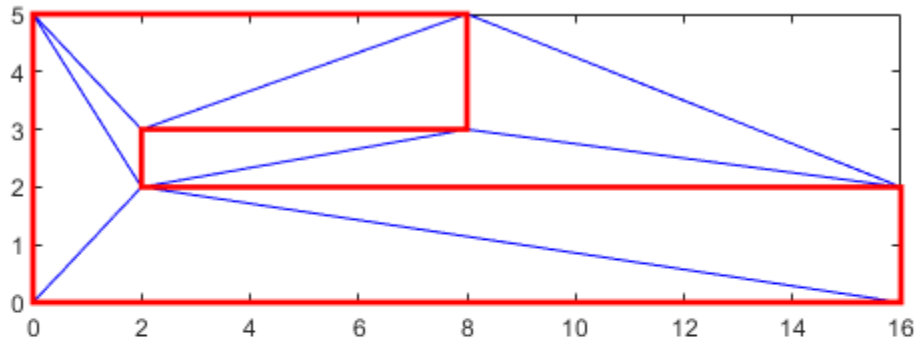
The plot shows that the edges of the triangulation respect the boundary of the polygon. However, the triangulation fills the concavities. What is needed is a triangulation that represents the polygonal domain. You can extract the triangles within the polygon using the `delaunayTriangulation` method, `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')
```

```matlab
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;
```
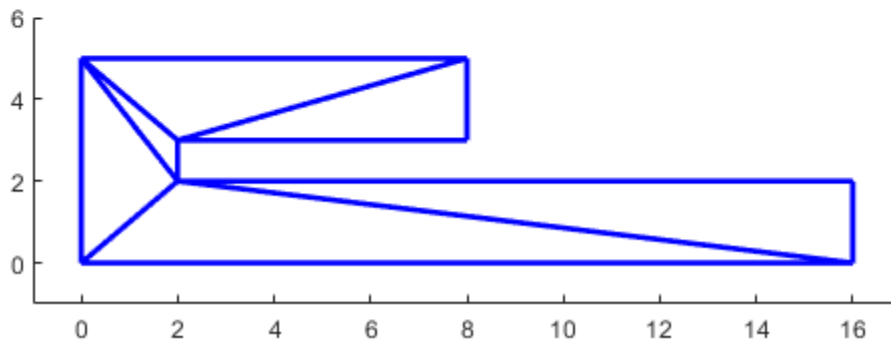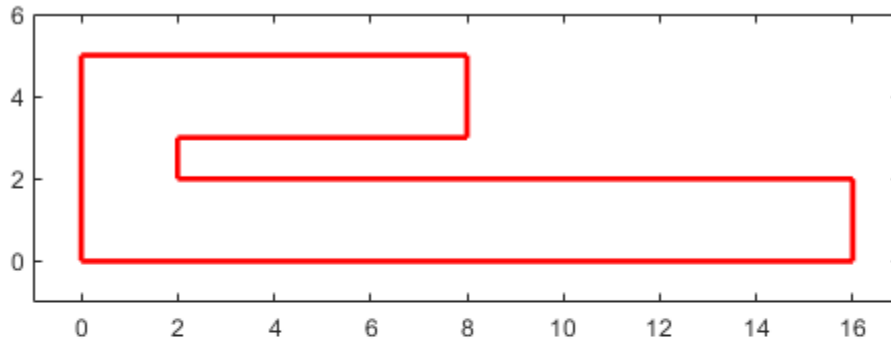
Triangulation of Point Sets Containing Duplicate Locations

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`:

```
P = gallery('uniformdata',[25 2],0);
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:

```
P = gallery('uniformdata',[25 2],0);
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 7-50

Spatial Searching

In this section...
"Introduction" on page 7-50
"Nearest-Neighbor Search" on page 7-50
"Point-Location Search" on page 7-54

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
```
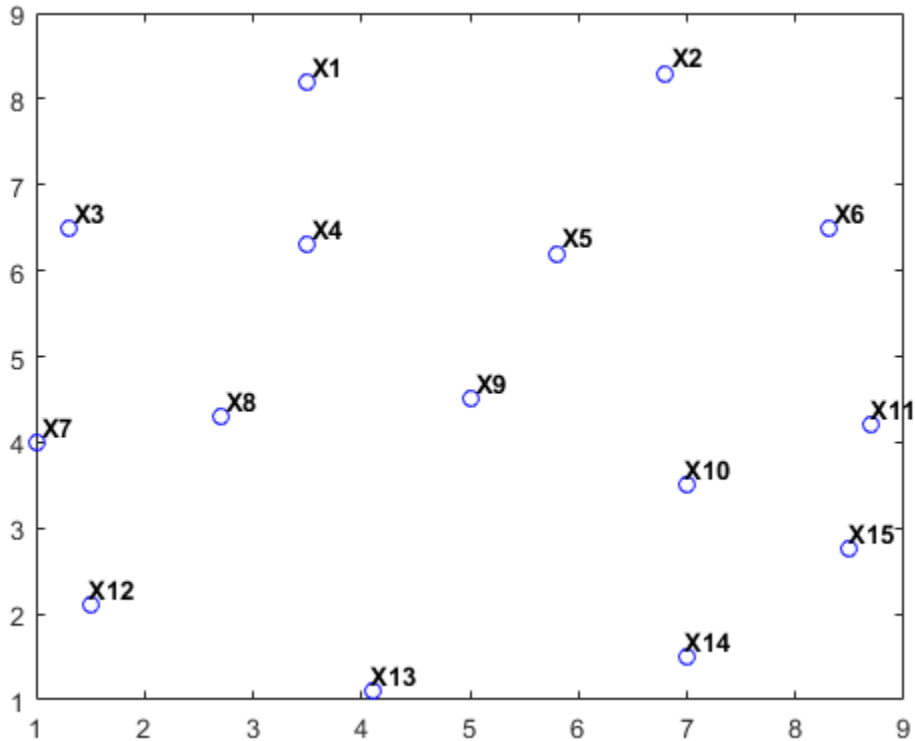
Create a Delaunay triangulation from the points.

```
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
```
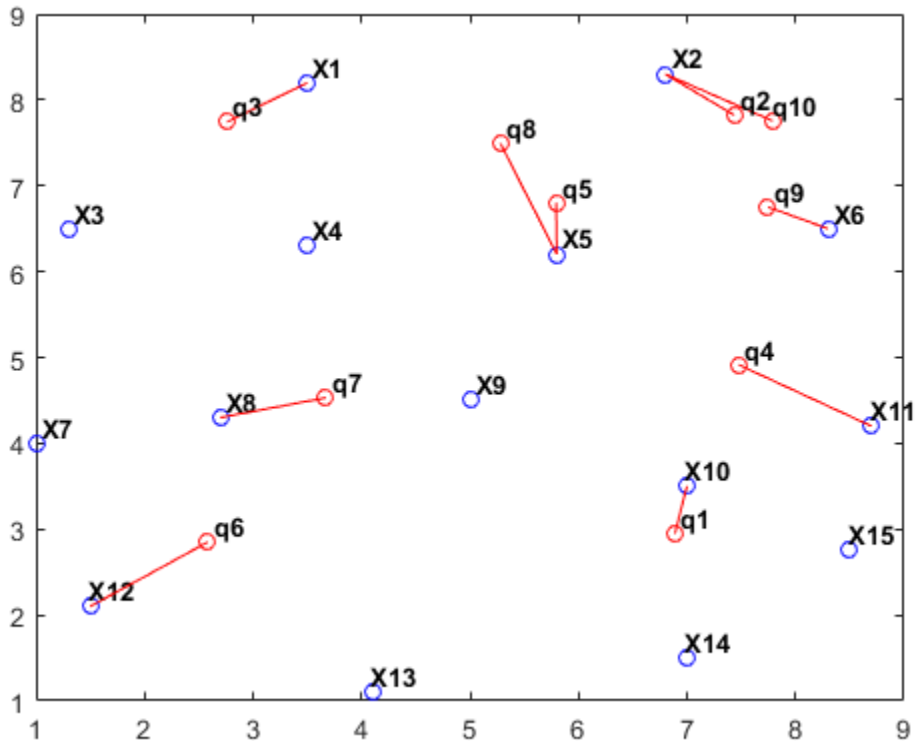
Performing a nearest-neighbor search in 3-D is a direct extension of the 2-D example based on `delaunayTriangulation`.

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
```
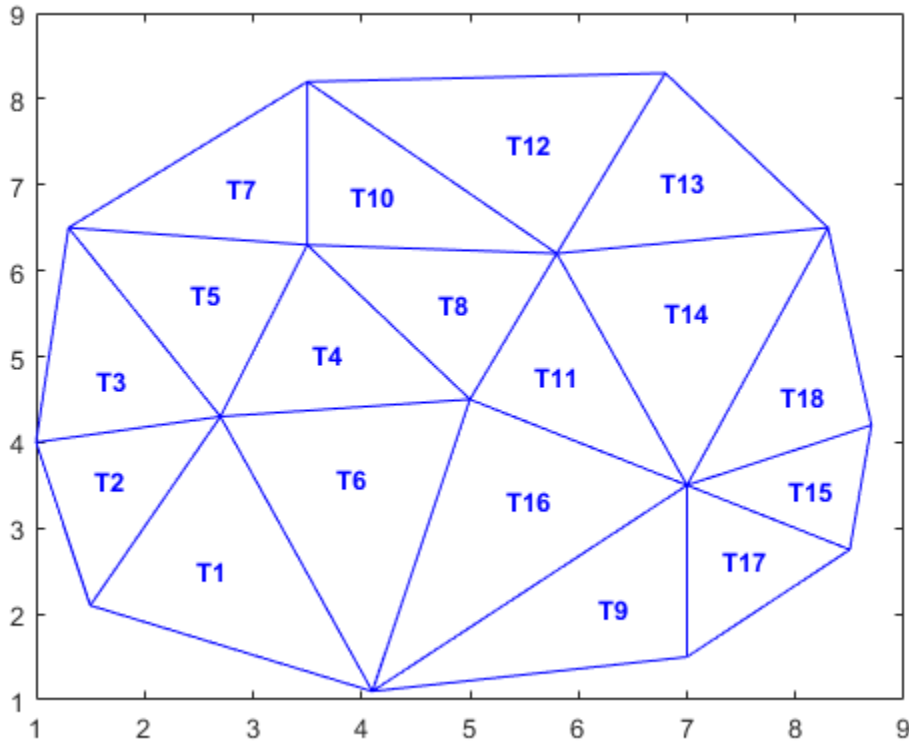
Now create some query points and add them to the plot. Then find the index of the corresponding enclosing triangles using the `pointLocation` method.

```
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
```
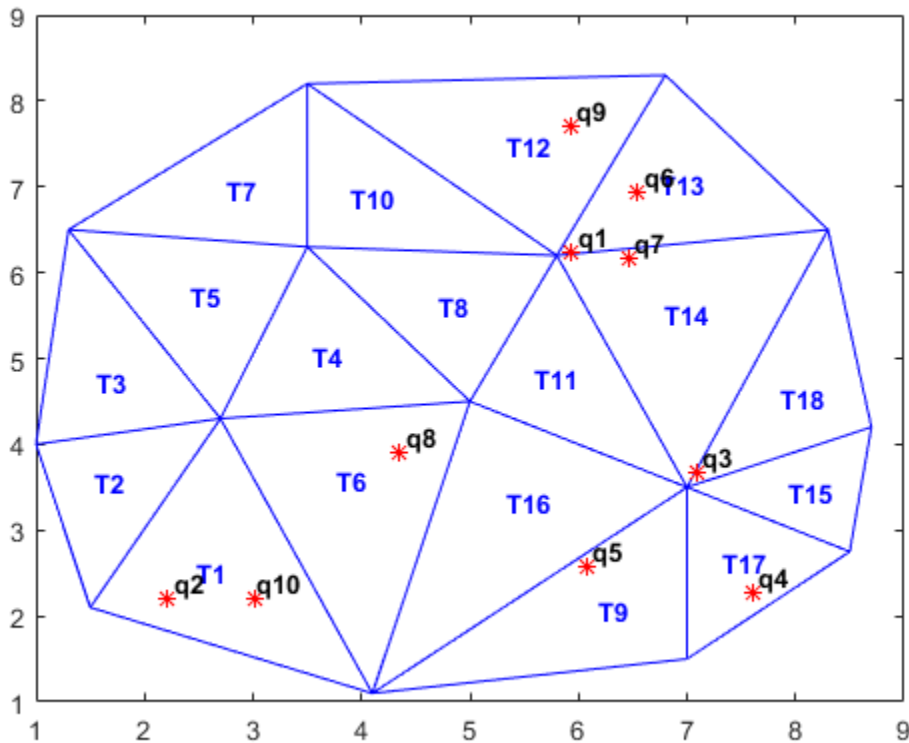


```
ti = pointLocation(dt,q);
```

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 8-55 for further details.

Voronoi Diagrams

In this section...
"Plot 2-D Voronoi Diagram and Delaunay Triangulation" on page 7-59
"Computing the Voronoi Diagram" on page 7-63

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];
```

```matlab
plot(P(1),P(2), '*r');
text(P(1), P(2), 'P', 'FontWeight', 'bold', ...
    'HorizontalAlignment','center', ...
    'BackgroundColor', 'none');
hold off
```
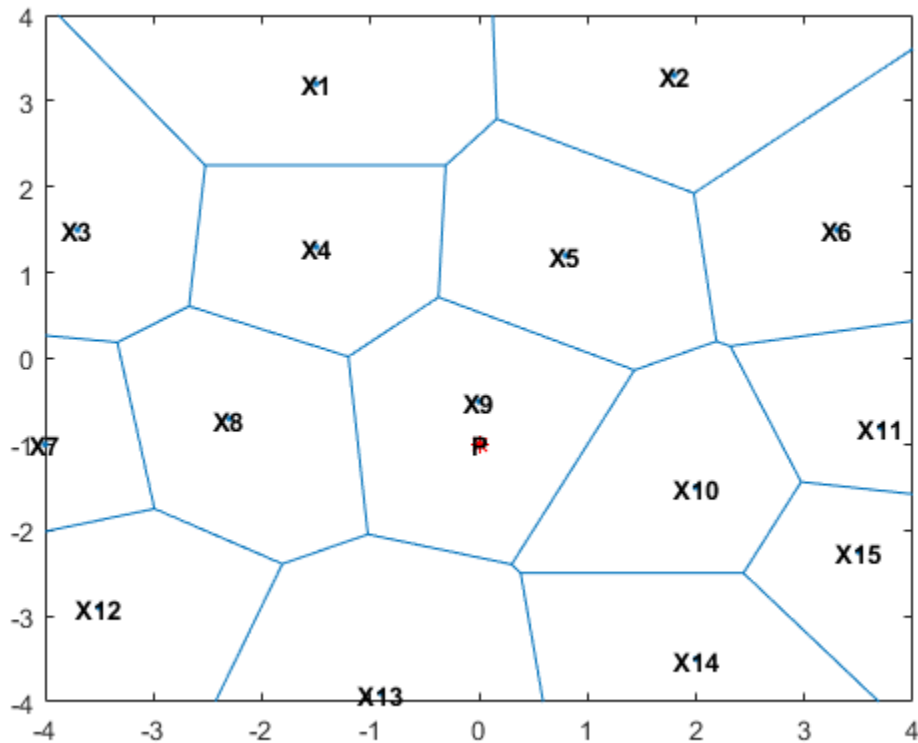

Observe that P is closer to X9 than to any other point in X, which is true for any point P within the region that bounds X9.

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
```
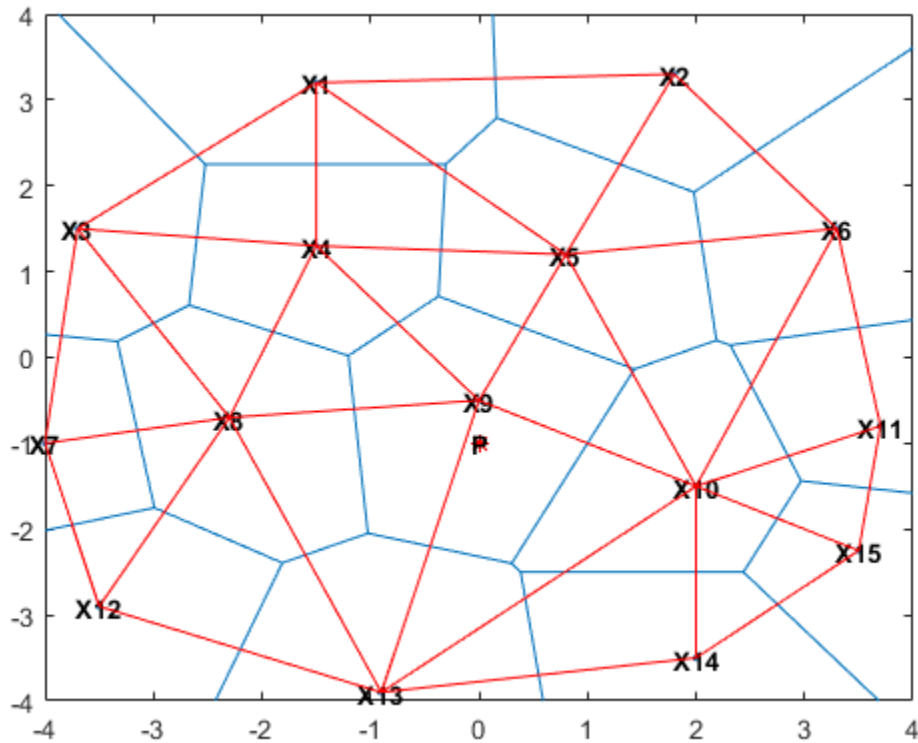

From the plot you can see that the Voronoi region associated with the point X9 is defined by the perpendicular bisectors of the Delaunay edges attached to X9. Also, the vertices of the Voronoi edges are located at the circumcenters of the Delaunay triangles. You can illustrate these associations by plotting 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
```
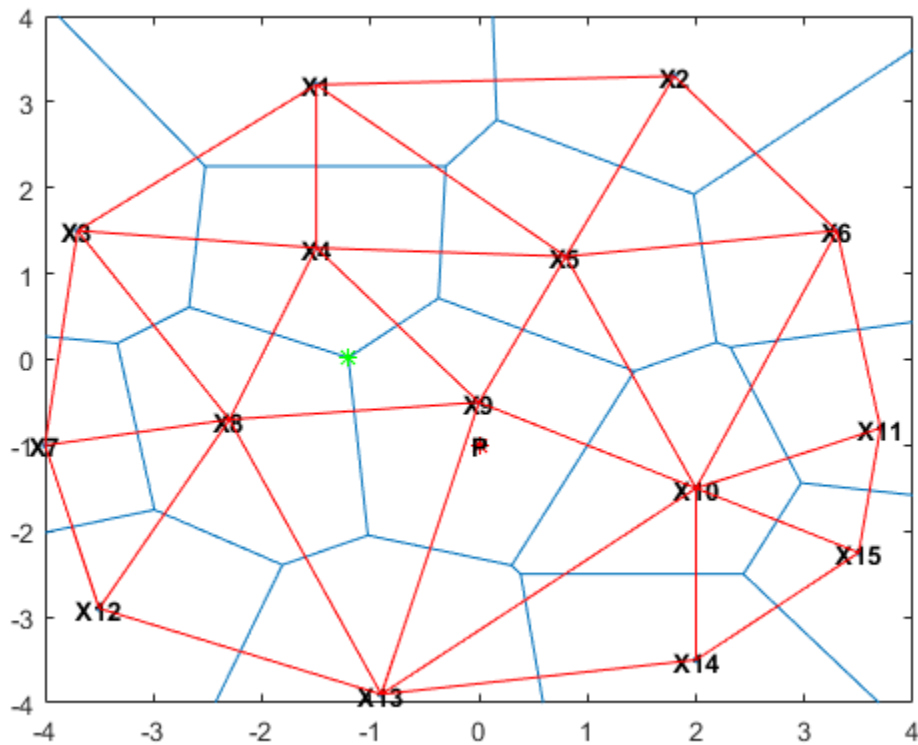

The Delaunay triangulation and Voronoi diagram are geometric duals of each other. You can compute the Voronoi diagram from the Delaunay triangulation and vice versa.

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 MATLAB 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 7-4 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
```
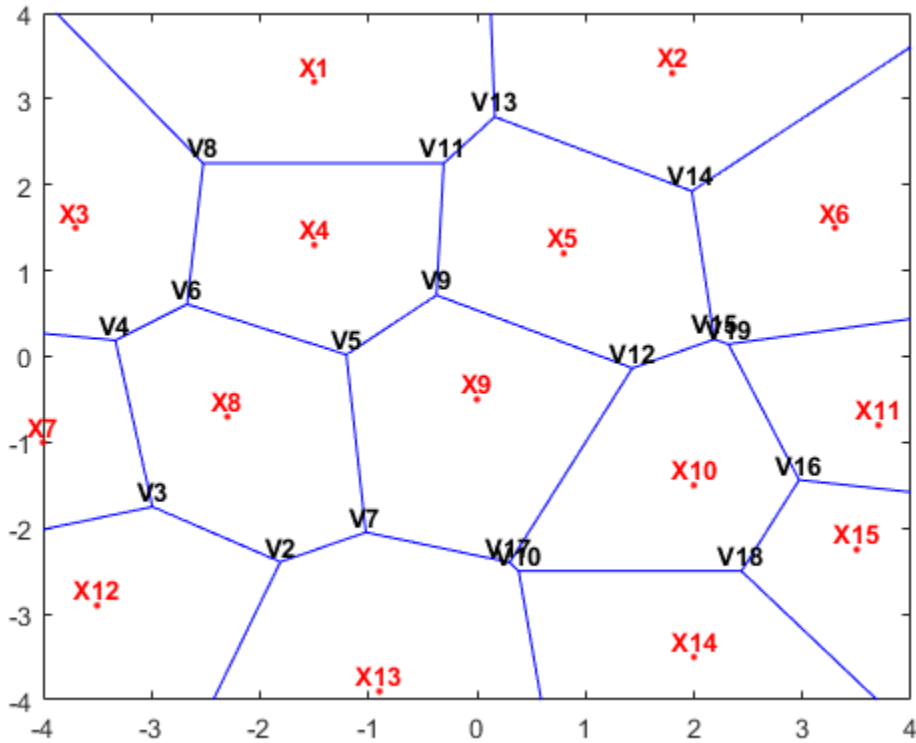
R{9} gives the indices of the Voronoi vertices associated with the point site X9.

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.

```
X = -3 + 6.*gallery('uniformdata',[25 3],0);
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')
```

3-D Voronoi Region
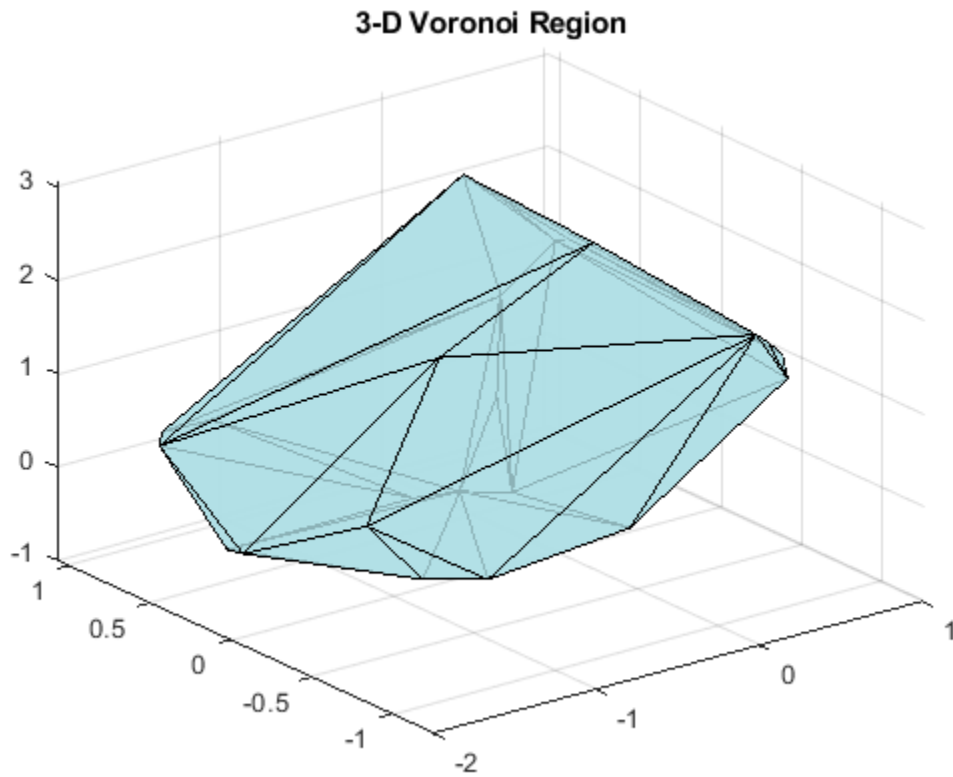
See Also

More About

- "Spatial Searching" on page 7-50

Types of Region Boundaries

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.

```
x = gallery('uniformdata',20,1,20);
y = gallery('uniformdata',20,1,30);
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
```

The Convex Hull of a Set of Points
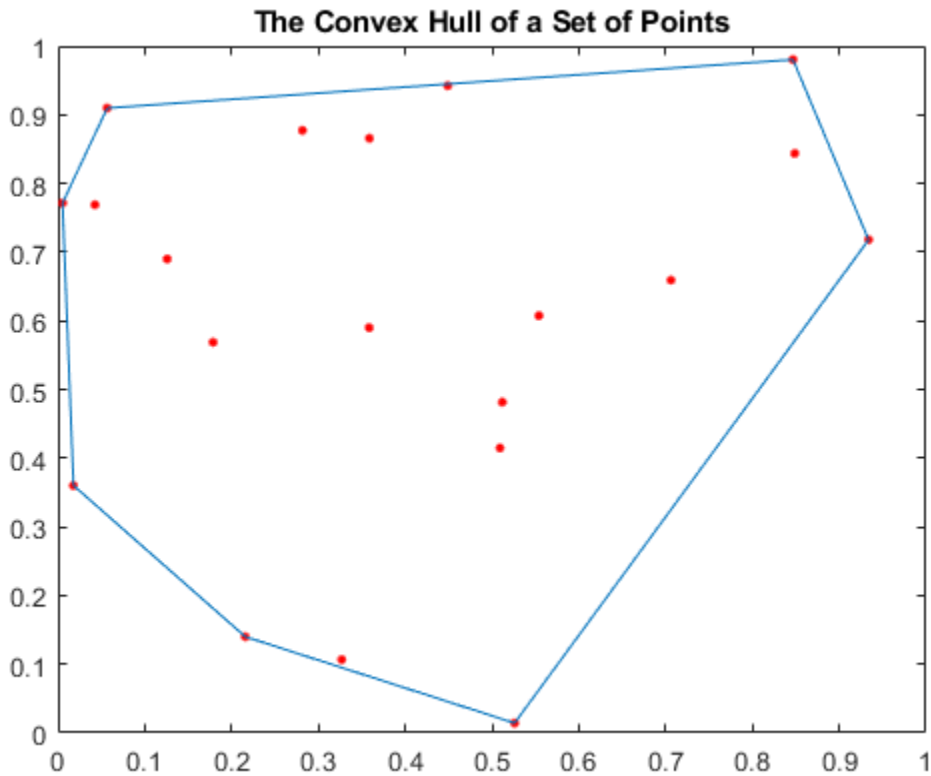
A convex polygon is a polygon that does not have concave vertices, for example:

```
x = gallery('uniformdata',20,1,30);
y = gallery('uniformdata',20,1,40);
k = convhull(x,y);
plot(x(k),y(k))
title('Convex Polygon')
```
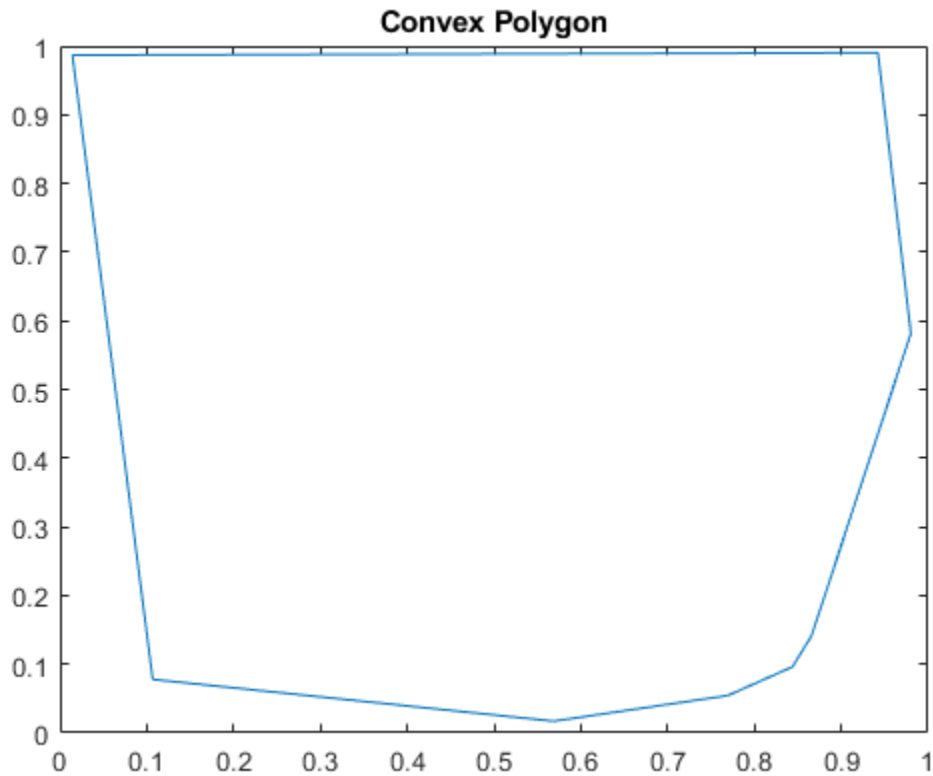
Convex Polygon

You can also create a boundary of a point set that is nonconvex. If you "vacuum pack" 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')
```
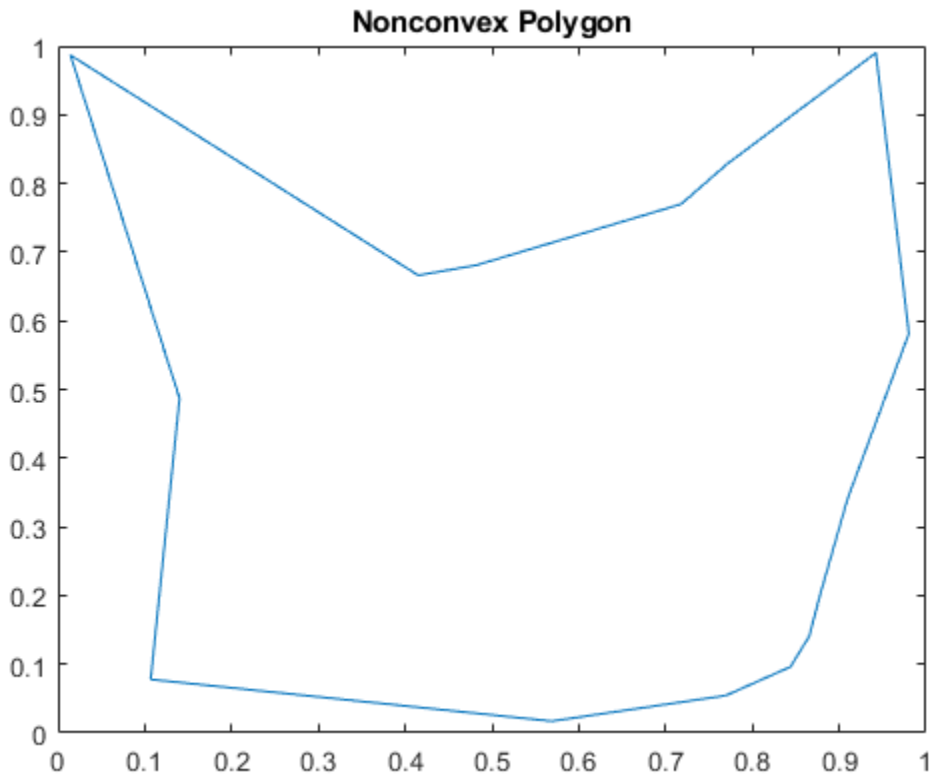
Nonconvex Polygon

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.

```
x = gallery('uniformdata',20,1,10);
y = gallery('uniformdata',20,1,20);
plot(x,y,'r.','MarkerSize',20)
hold on
shp = alphaShape(x,y,100);
plot(shp)
title('Convex Alpha Shape')
hold off
```
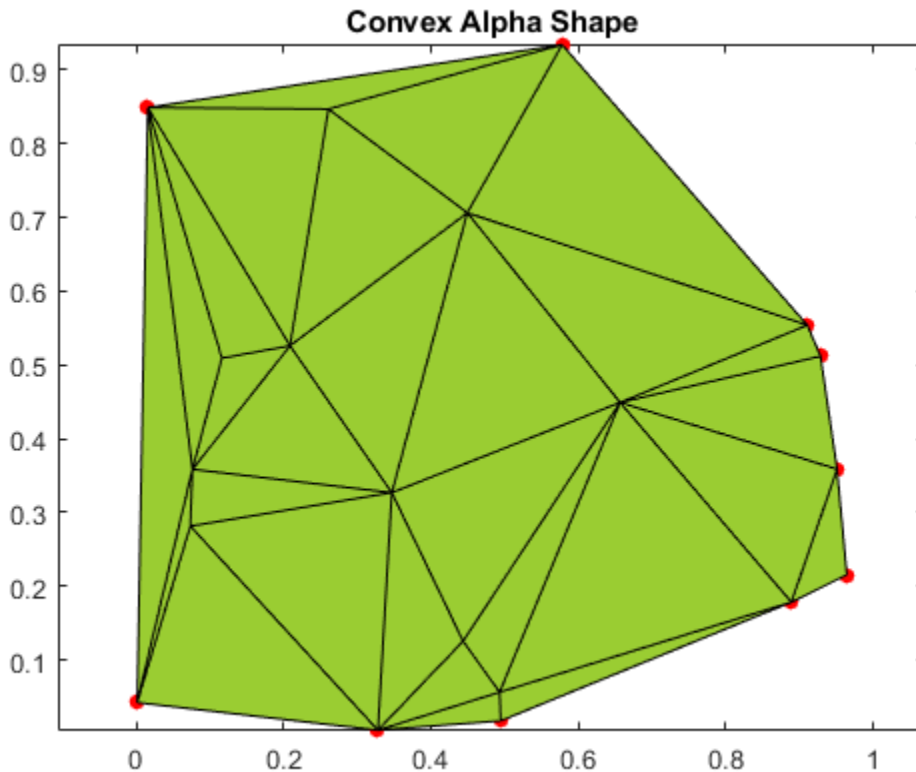
Unlike the convex hull, alpha shapes have a parameter that controls the level of detail, or how tightly the boundary fits around the point set. The parameter is called *alpha* or the *alpha radius*. Varying the alpha radius from 0 to `Inf` produces a set of different alpha shapes unique for that point set.

```
plot(x,y,'r.','MarkerSize',20)
hold on
shp = alphaShape(x,y,.5);
plot(shp)
title('Nonconvex Alpha Shape')
hold off
```

Nonconvex Alpha Shape
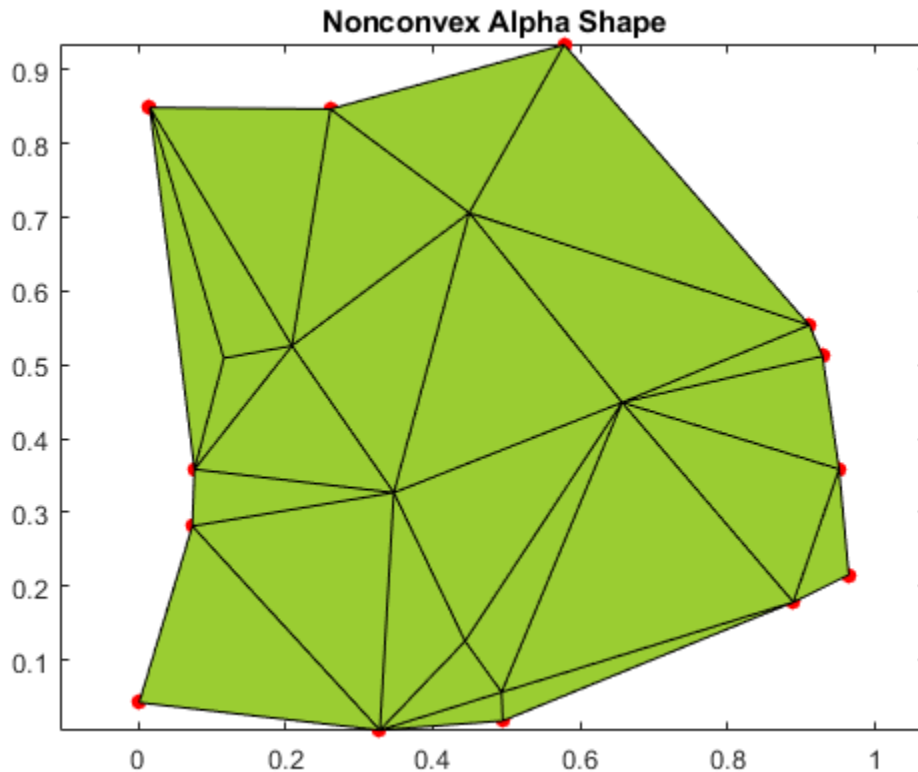
Varying the alpha radius can sometimes result in an alpha shape with multiple regions, which might or might not contain holes. However, the `alphaShape` function in MATLAB® always returns regularized alpha shapes, which prevents isolated or dangling points, edges, or faces.

```
plot(x,y,'r.','MarkerSize',20)
hold on
shp = alphaShape(x,y);
plot(shp)
title('Alpha Shape with Multiple Regions')
hold off
```

Alpha Shape with Multiple Regions
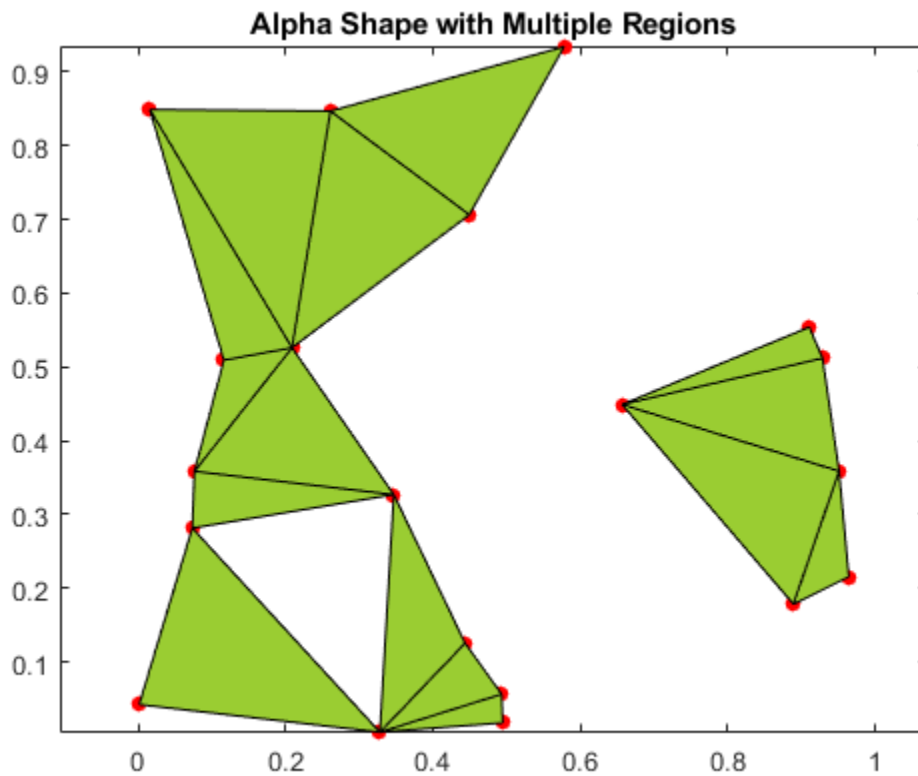
See Also
alphaShape | convhull

More About

- "Using the delaunayTriangulation Class" on page 7-24
- "Triangulation Matrix Format" on page 7-4

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')
```
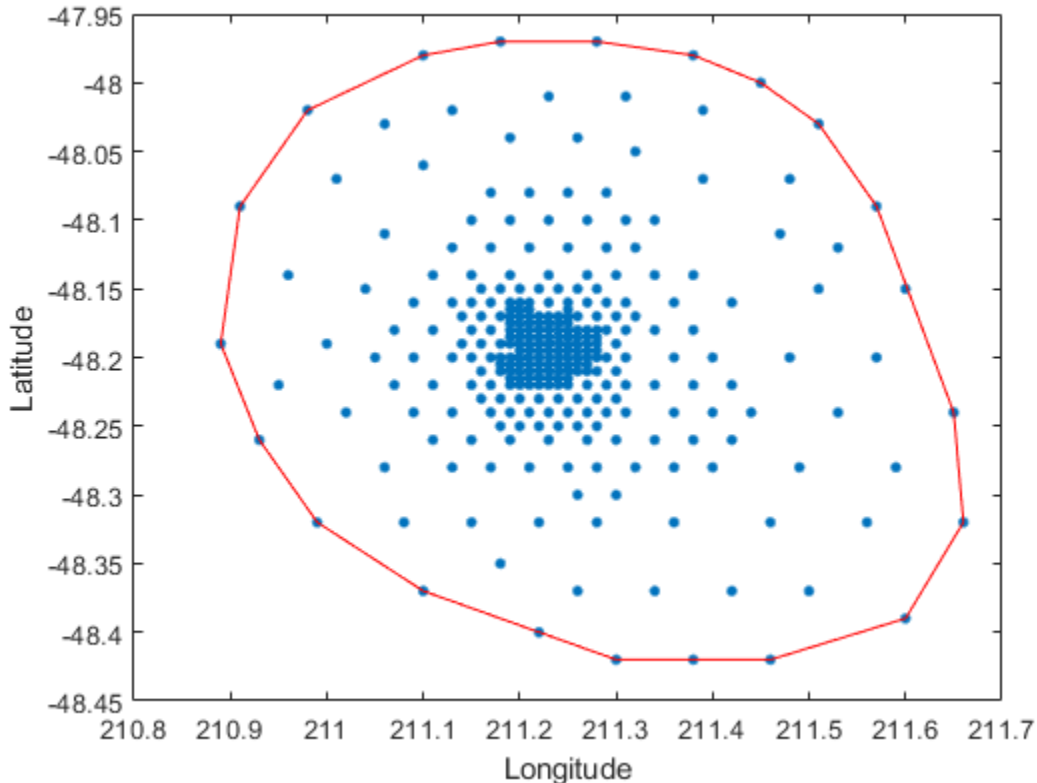
Add point labels to the points on the convex hull to observe the structure of K.

```
[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')
```
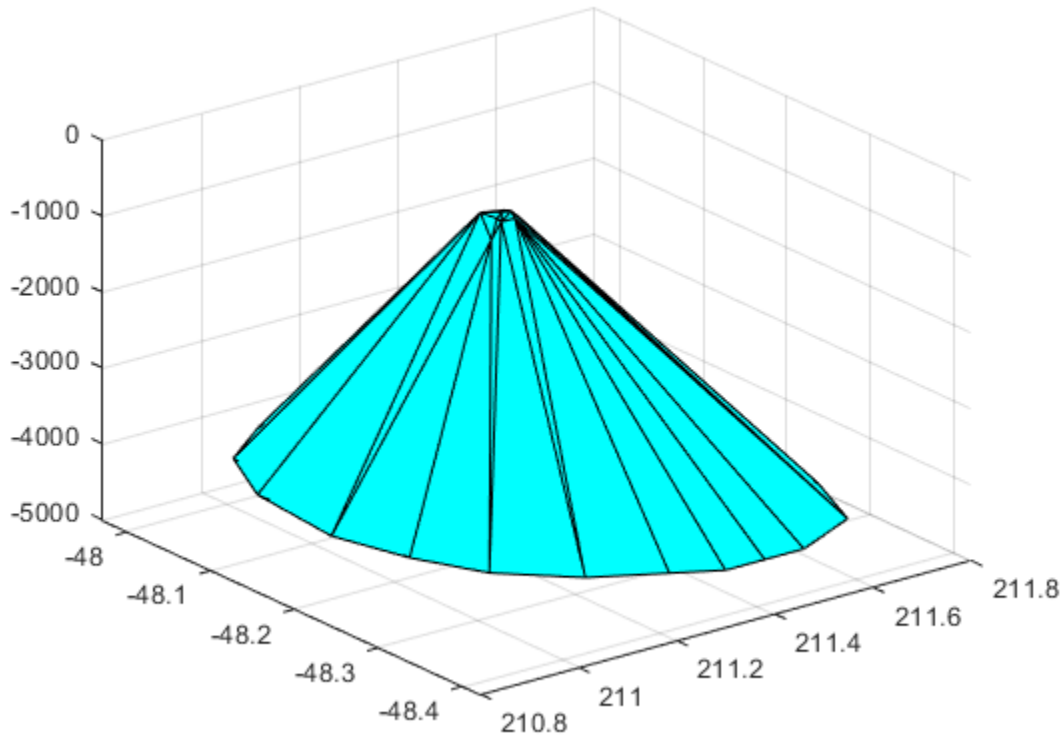

The volume bounded by the 3-D convex hull can optionally be returned by `convhull`, the syntax is as follows.

```
[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
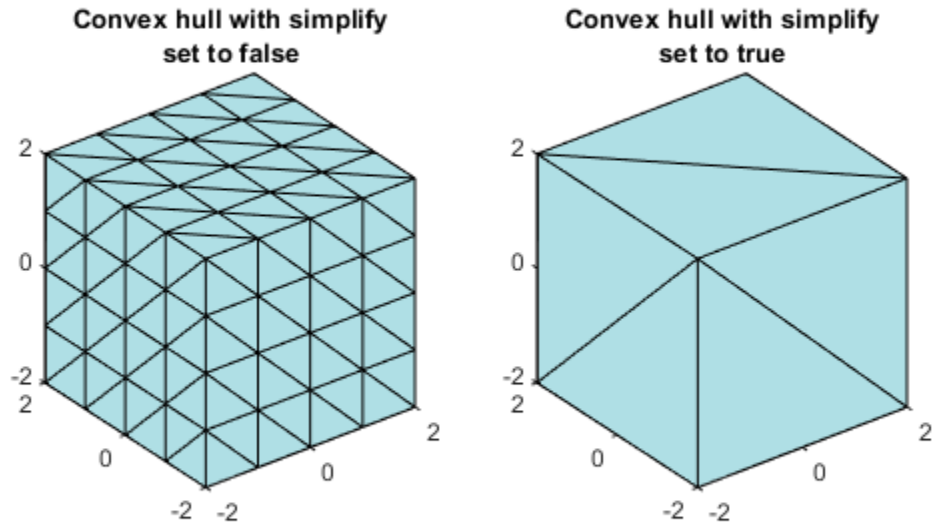
MATLAB provides the `convhulln` function to support the computation of convex hulls and hypervolumes in higher dimensions. Though `convhulln` supports N-D, problems in more than 10 dimensions present challenges due to the rapidly growing memory requirements.

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
```
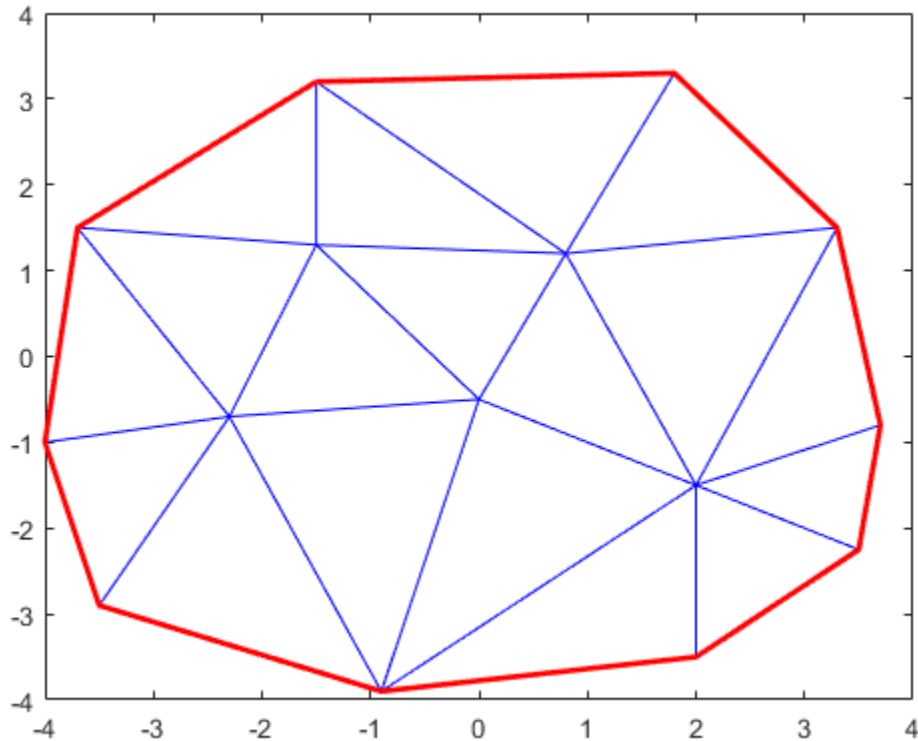
In 3-D, the facets of the triangulation that are shared only by one tetrahedron represent the boundary of the convex hull.

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)
```

8
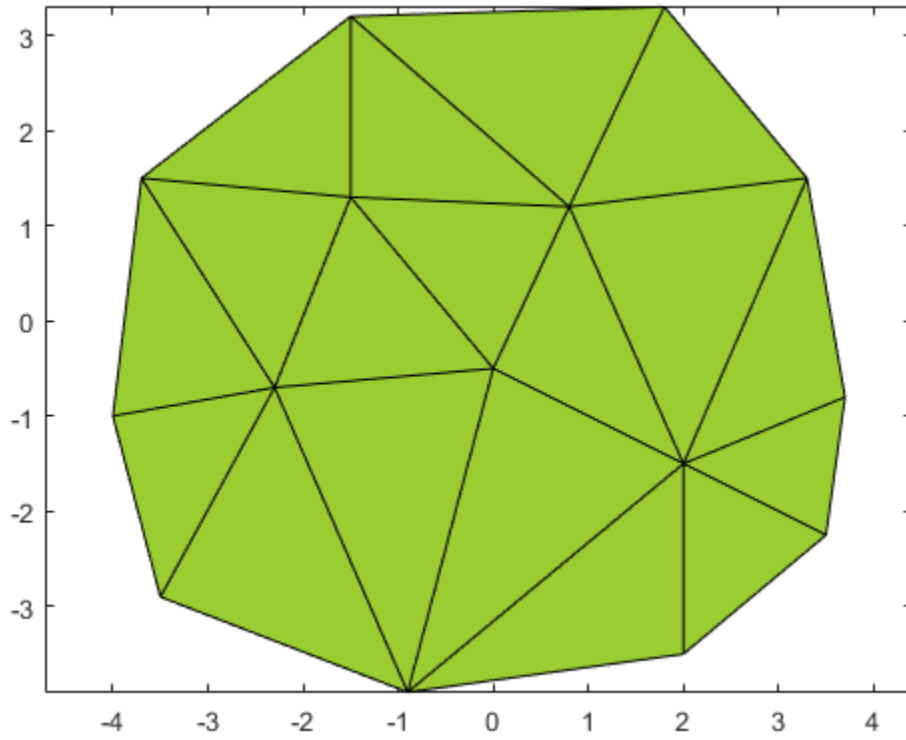
Interpolation

Gridded and Scattered Sample Data

Interpolation is a method for estimating the value at a query location that lies within the domain of a set of sample data points. A sample data set defined by locations X and corresponding values V can be interpolated to produce a function of the form V = F(X). This function can then be used to evaluate a query point Xq, to give Vq = F(Xq). This is a single-valued function; for any query Xq within the domain 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. One other interesting characteristic is that the interpolating function passes through the data points. This is an important distinction between interpolation and curve/surface fitting. In fitting, the function does not necessarily pass through the sample data points.

The computation of the value Vq is generally based on the data points in the neighborhood of the query point Xq. There are numerous approaches to performing interpolation. In MATLAB interpolation is classified into two categories depending on the structure of the sample data. The sample data may be ordered in a axis-aligned grid or they may be scattered. In the case of a gridded distribution of sample points, you can leverage the organized structure of the data to efficiently find the sample points in the neighborhood of the query. Interpolation of scattered data on the other hand requires a triangulation of the data points, and this introduces an additional level of computation.

The two approaches to interpolation are covered in the following sections:

- The "Interpolating Gridded Data" on page 8-4 section covers 1-D interpolation, and the N-D (N ≥ 2) interpolation of sample data in axis-aligned grid format:
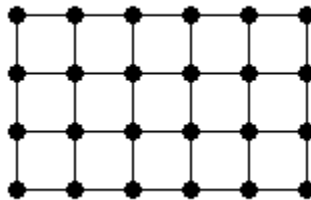
- The "Interpolating Scattered Data" on page 8-55 section covers the N-D (N ≥ 2) interpolation of scattered data:
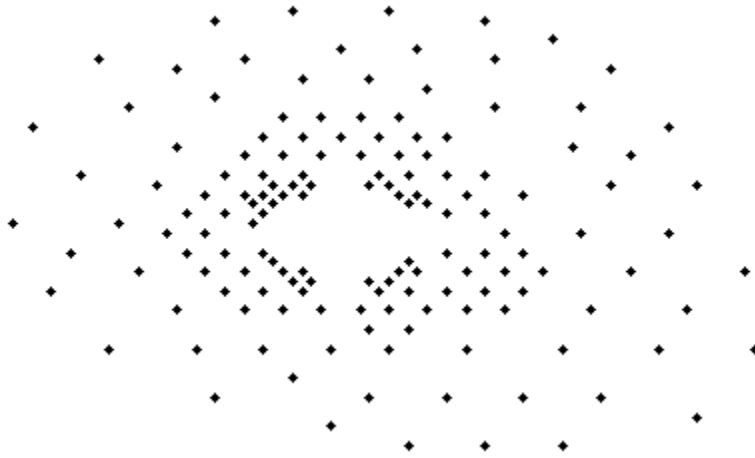
Interpolating Gridded Data

In this section...

Gridded Data Representation

Grid Representation

This example shows how to create a 2-D grid using `meshgrid` and `ndgrid`.

In MATLAB®, *gridded data* means data *ordered* in a grid. You can understand ordered data by thinking about how MATLAB stores data in matrices.

Define some data.

```
A = gallery('uniformdata',[3 5],0)

A = 3×5

    0.9501    0.4860    0.4565    0.4447    0.9218
    0.2311    0.8913    0.0185    0.6154    0.7382
    0.6068    0.7621    0.8214    0.7919    0.1763
```

MATLAB stores the data in a matrix. You can think of A as a set of places for the elements that are ordered by the indices of the matrix. The linear indices of A are:

$$\begin{bmatrix} 1 & 4 & 7 & 10 & 13 \\ 2 & 5 & 8 & 11 & 14 \\ 3 & 6 & 9 & 12 & 15 \end{bmatrix}$$

Any element in the matrix can be retrieved by indexing, that is, by asking for the element at that place in the matrix. The ith element in A is retrieved by `A(i)`.

Retrieve the 7th element in A.

`A(7)`

`ans = 0.4565`

For an m-by-n matrix, you can find the column elements adjacent to the ith element by offsetting i by 1. To find the row elements adjacent to the ith element, offset i by m.

$$i - 1$$
$$i - m \quad i \quad i + m$$
$$i + 1$$

Retrieve the column elements adjacent to `A(7)`.

`A(6),A(8)`

`ans = 0.7621`

`ans = 0.0185`

MATLAB uses a similar idea for creating data grids. A grid is not just a set of points that meet certain geometric properties. Rather, a *gridded data set* relies on an ordered relationship among the points in the grid. The adjacency information readily available in the grid structure is very useful for many applications and particularly grid-based interpolation.

MATLAB provides two functions for creating grids:

- `meshgrid` creates 2-D and 3-D grids that are Cartesian axis aligned. To create a 2-D grid, the syntax is `[X,Y] = meshgrid(xgv, ygv)` where `xgv` is a vector of length m, and `ygv` is a vector of length n. `meshgrid` replicates `xgv` to form the n-by-m matrix X, and it replicates `ygv` to form another n-by-m matrix Y. X and Y represent the coordinates of the grid points. The rows of X are aligned with the horizontal X-axis, and the columns of Y are aligned with the negative Y-axis.
- `ndgrid` creates N-D grids that are array space aligned. In array space the axes are row, column, page, etc. The calling syntax is `[X1, X2, X3,...,Xn] = ndgrid(x1gv, x2gv, x3gv,...,xngv)` where `x1gv,x2gv,x3gv,...,xngv` are vectors that span the grid in each dimension. `X1,X2,X3,...,Xn` are output arrays

that can be used for evaluating functions of multiple variables and for multidimensional interpolation.

Use `meshgrid` to create a 2-D axis aligned grid from two vectors, `xgv` and `ygv`.

```
xgv = [1 2 3];
ygv = [1 2 3 4 5];
[X,Y] = meshgrid(xgv, ygv)
```

X = 5×3

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

Y = 5×3

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

Now use `ndgrid` to create a 2-D space aligned grid from the same two vectors, `xgv` and `ygv`.

```
[X1,X2] = ndgrid(xgv,ygv)
```

X1 = 3×5

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

X2 = 3×5

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

Notice that `ndgrid`'s X1 is the transpose of `meshgrid`'s X. The same is true for X2 and Y.

For a given set of inputs, the `meshgrid` and `ndgrid` functions will produce grids with exactly the same coordinates. The only difference between their outputs is the format of the coordinate arrays. Plot both outputs and see that they are the same.

```
figure()
[X1_ndgrid,X2_ndgrid] = ndgrid(1:3,1:5);
Z = zeros(3,5);
mesh(X1_ndgrid,X2_ndgrid,Z,'EdgeColor','black')
axis equal;

% Set the axis labeling and title
h1 = gca;
h1.XTick = [1 2 3];
h1.YTick = [1 2 3 4 5];
xlabel('ndgrid Output')
```
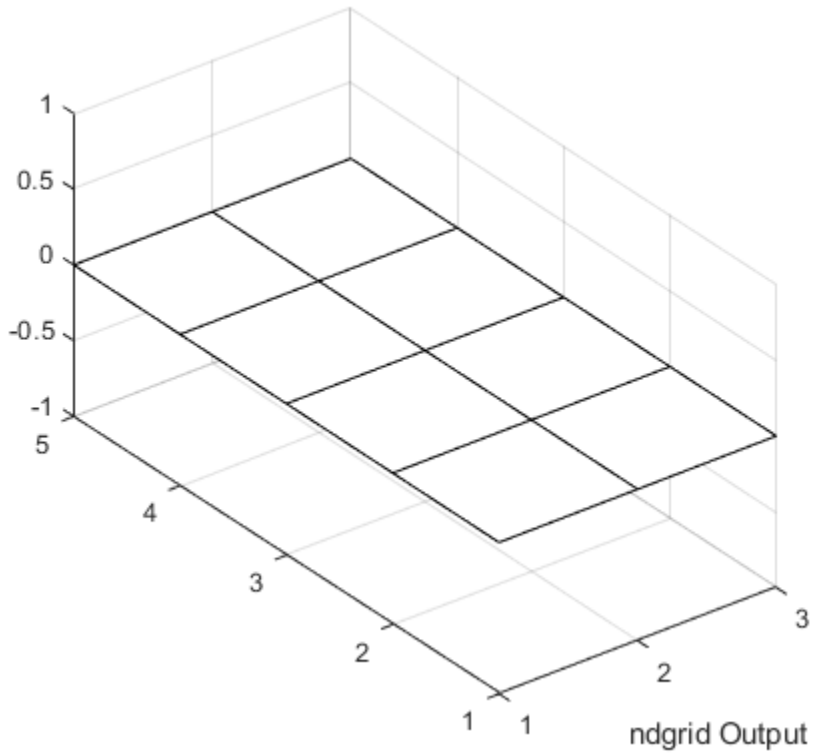
```
figure()
[X_meshgrid,Y_meshgrid] = meshgrid(1:3, 1:5);
mesh(X_meshgrid,Y_meshgrid,Z','EdgeColor','black')
axis equal;

% Set the axis labeling and title
h2 = gca;
h2.XTick = [1 2 3];
h2.YTick = [1 2 3 4 5];
xlabel('meshgrid Output')
```

meshgrid Output
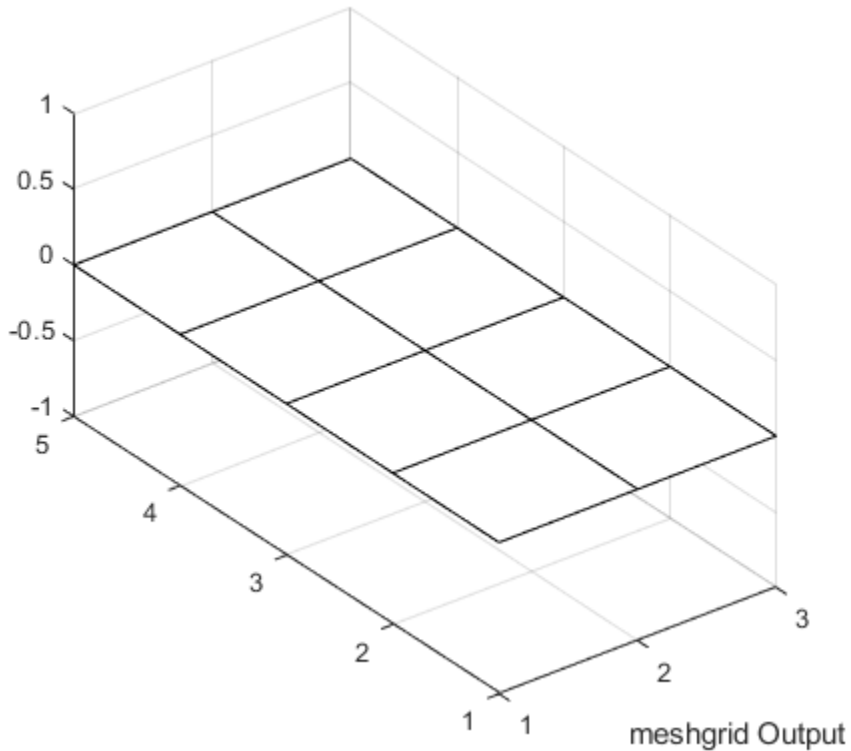
Depending on how you intend to use your grid, you may prefer one format or the other. Some functions in MATLAB might require your data to have a `meshgrid` format while others might require an `ndgrid` format.

Converting Between Grid Formats

To convert a 2-D grid output from `meshgrid` to the `ndgrid` format, transpose the coordinate matrices:

```
[X_meshgrid,Y_meshgrid] = meshgrid(1:3, 1:5);
[X1_ndgrid,X2_ndgrid] = ndgrid(1:3,1:5);

isequal(X_meshgrid',X1_ndgrid)
ans =
```

```
        1
isequal(Y_meshgrid',X2_ndgrid)
ans =
     1
```

You can also use the `permute` function.

```
isequal(permute(X_meshgrid,[2 1]),X1_ndgrid)
ans =
     1
```

To convert a 3-D `meshgrid` to `ndgrid`, transpose each page of the coordinate array. For a given array `my_array`, `permute(my_array, [2 1 3])` interchanges the rows and columns, and the net effect is the transposition of every page:

```
[X_meshgrid,Y_meshgrid,Z_meshgrid] = meshgrid(1:3, 1:5, [1 2]);
[X1_ndgrid,X2_ndgrid,X3_ndgrid] = ndgrid(1:3,1:5, [1 2]);

isequal(permute(X_meshgrid,[2 1 3]),X1_ndgrid)
ans =
     1

isequal(permute(Y_meshgrid,[2 1 3]),X2_ndgrid)
ans =
     1

isequal(permute(Z_meshgrid,[2 1 3]),X3_ndgrid)
ans =
     1
```

Grid Vectors

The inputs that you pass to the grid functions are called grid vectors. The grid vectors implicitly define the grid. Consider two vectors, `x1gv = (1:3)` and `x2gv = (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:
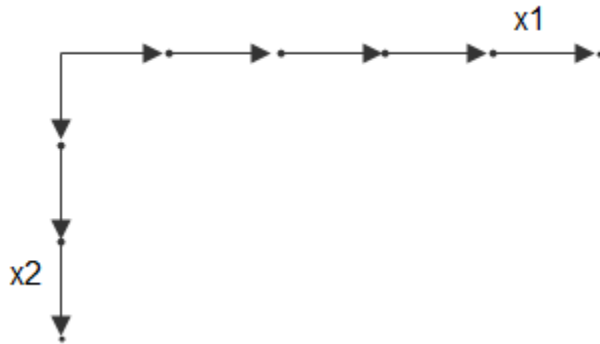
Each arrow points to a location. You can use these two vectors to define a set of grid points, where one set of coordinates is given by `x1gv` and the other set of coordinates is given by `x2gv`. When the grid vectors are replicated they form two coordinate arrays that make up the full grid:
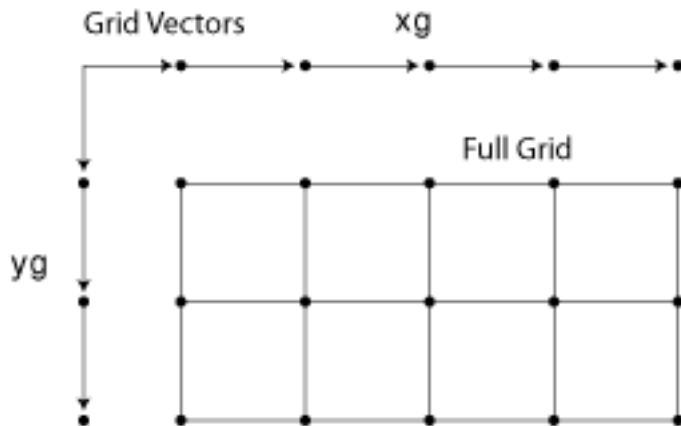
Monotonic and Nonmonotonic Grids

Your input grid vectors may 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 8 3 1]`, `ndgrid` outputs the following:

```
[X1,X2] = ndgrid([2 4 6 3 1])
X1 =
     2     2     2     2     2
     4     4     4     4     4
     6     6     6     6     6
     3     3     3     3     3
     1     1     1     1     1

X2 =
     2     4     6     3     1
     2     4     6     3     1
     2     4     6     3     1
     2     4     6     3     1
     2     4     6     3     1
```

Your grid vectors should be monotonic if you intend to pass the grid to other MATLAB functions.

Uniform and Nonuniform Grids

A uniform grid is one in which all neighboring points in a given dimension have equal spacing. For example, `[X1, X2] = ndgrid([1 3 5 9],[11 13 15])` is a uniform with a spacing of 2 units in each dimension.

It is not necessary for the spacing in a uniform grid to be equal in all dimensions. For example, `[X1, X2] = ndgrid([1 2 3 4],[11 13 15])` is considered uniform even though the spacing in X1 and X2 are different.

A grid whose spacing varies within any dimension is a nonuniform grid. For example, `[X1, X2] = ndgrid([1 5 6 9],[11 13 15])` creates a nonuniform grid because the spacing varies along the first dimension.

| Uniform | Uniform | Nonuniform |
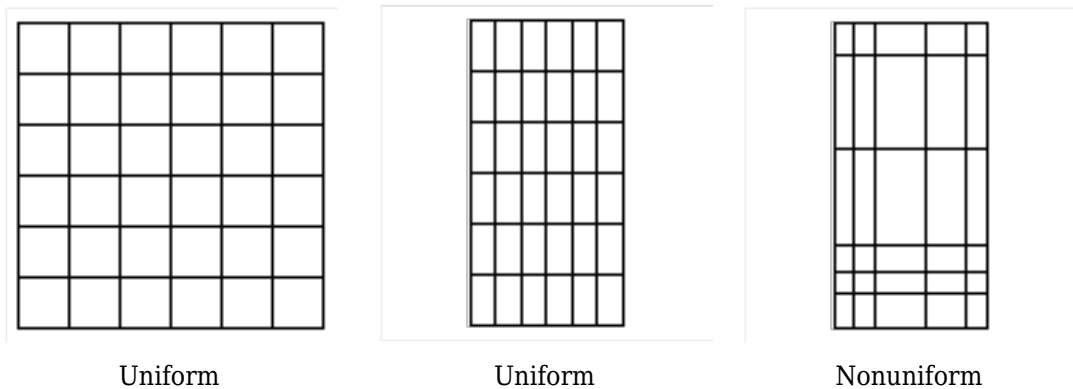
Types of Grid Representations

MATLAB allows you to represent a grid in one of three representations: full grid, compact grid, or default grid. The compact grid and default grid are used primarily for convenience and improved efficiency.

Full Grid

A full grid is one in which the points are explicitly defined. The outputs of `ndgrid` and `meshgrid` define a full grid.

Compact Grid

The explicit definition of every point in a grid is expensive in terms of memory. The compact grid representation is a way to dispense with the memory overhead of a full grid. The compact grid representation stores just the grid vectors instead of the full grid. (For information on how the `griddedInterpolant` class uses the compact grid representation, see "Interpolation with the griddedInterpolant Class" on page 8-39.)

Default Grid

For many cases, when you are analyzing data, you care about both the distances between points in the grid and their value. For example, a data set might represent rainfall at specific points in a geographic area. In this case, you would care about the value at each grid point and the distance between a point and its neighbor. In other cases, you might care only about the value for a given point and not about the relative distances. For example, you could be working with input data from an MRI scan where the distance between the points is completely uniform. In this case, you would care about the values of the points, but you can be certain of a completely uniform grid. In this case, the default

grid representation is useful. The default grid representation stores the value at a grid point explicitly and creates grid point coordinates implicitly.

This example shows how you can use the default grid instead of the full grid to produce a plot.

Create a grid and a function that you want to plot.

```
[X,Y] = meshgrid(11:15,11:16);
Z = X.^2 + Y.^2;
```

Use the full grid that you created with `meshgrid` by using X and Y.

```
figure
surf(X,Y,Z)
title('Full Grid')
```
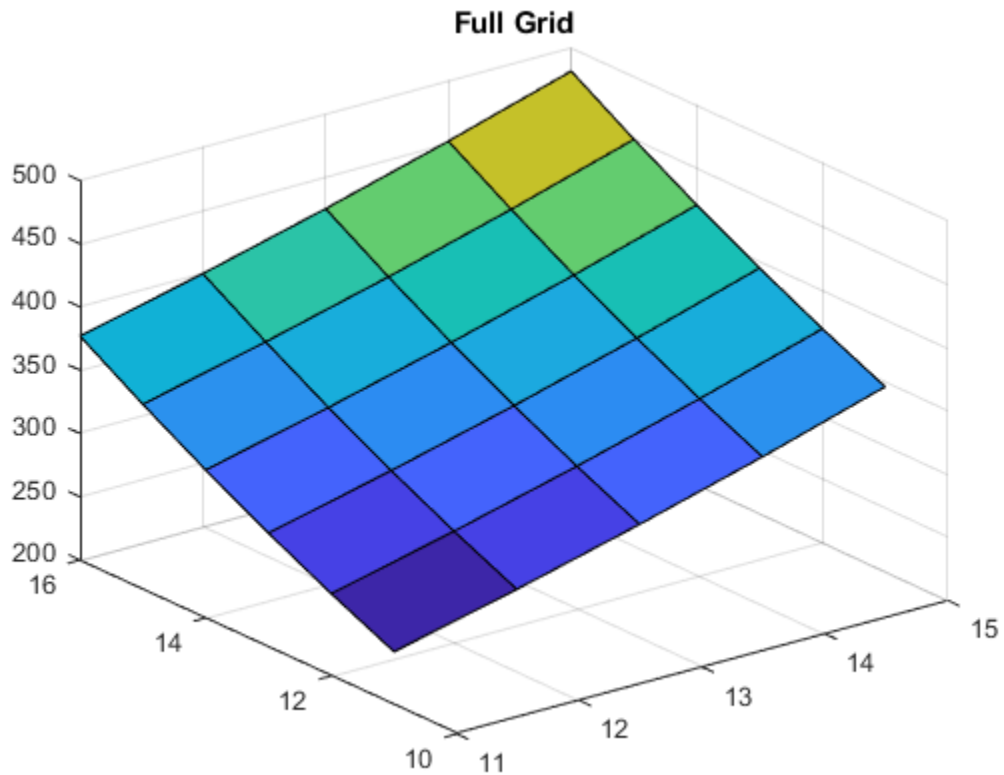
Full Grid

In contrast, have MATLAB create a default grid instead of using the full grid.

```
figure
surf(Z)
title('Default Grid')
```
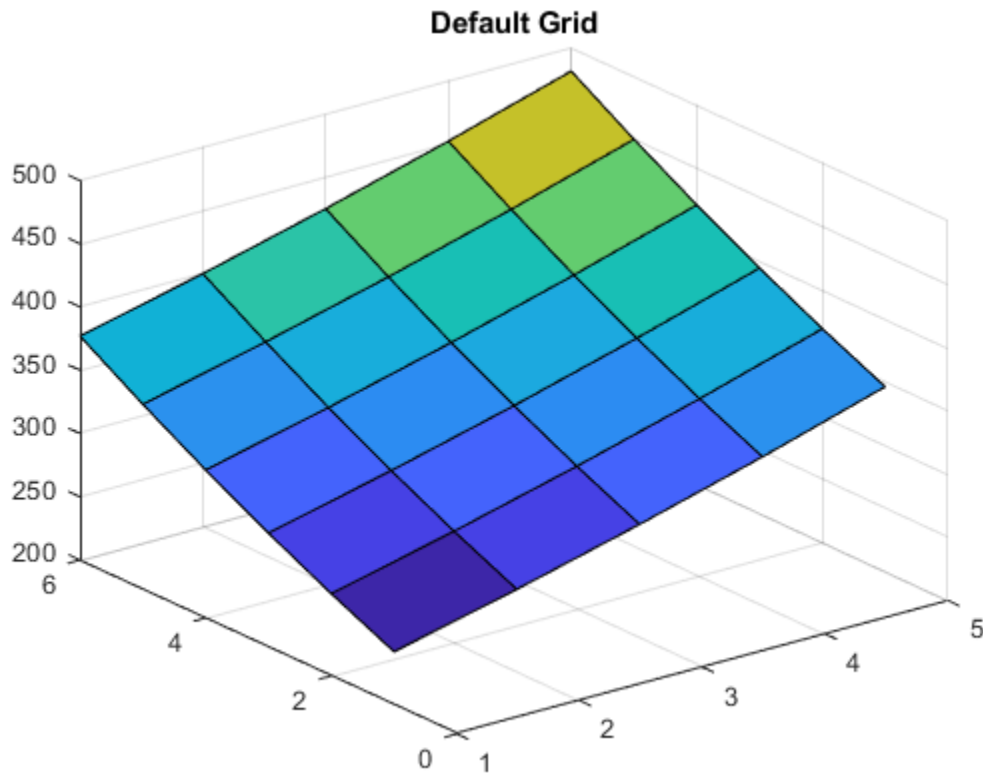
Notice the difference in the scaling of the axes. When you plot the full grid the X- and Y-axis have ranges of 11 to 15 and 10 to 16 respectively. When you plot the default grid the X- and Y-axis have ranges of 1 to m and 1 to n, where Z is m-by-n.

Grid Approximation Techniques

In some cases, you may need to approximate a grid for your data. An approximate grid can be idealized by a standard MATLAB grid by choosing an appropriate set of grid vectors. 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:
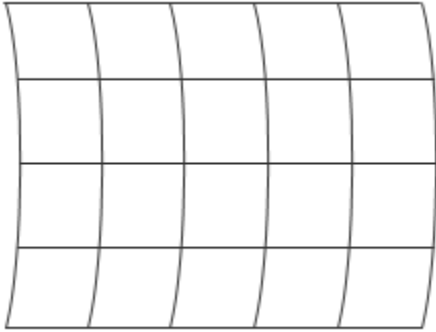
In this case, although the input data cannot be gridded directly, you can approximate straight grid lines at appropriate intervals:
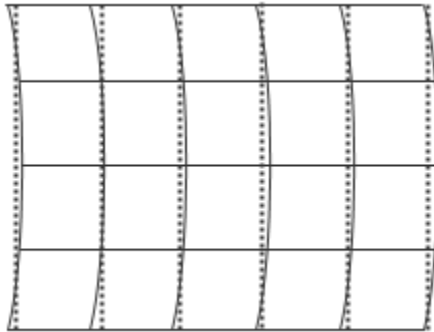
You can also use the default grid on page 8-13.

Degenerate Grid

A degenerate grid is a special case of grid where one or more dimensions of the grid are singletons. A singleton dimension can be internal, as in 7:7 in this example:

```
[X1,X2,X3] = ndgrid(1:2:10,7:7,1:3:15);
```

or a singleton dimension can be trailing:

```
[X1,X2,X3] = ndgrid(1:2:10,1:3:15,7:7);
```

8-17

You can create a degenerate grid if you are trying to take a slice of a larger data set. For example, you may want to analyze just a slice of a 3-D MRI scan. In this case, you will need a slice of data from a multidimensional grid, such as the dotted slice in the following figure:
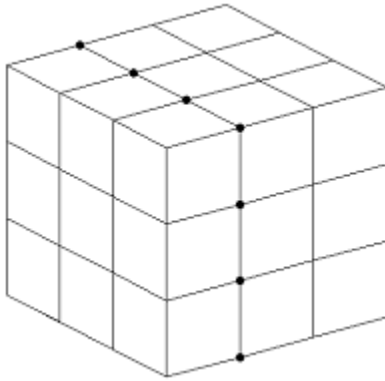
If you use indexing to extract the desired data, the resultant grid is degenerate in the X3 dimension:

```
[X1,X2,X3] = ndgrid(1:3);

X1_slice = X1(:,:,2)
X1_slice =
     1     1     1
     2     2     2
     3     3     3

X2_slice = X2(:,:,2)
X2_slice =
     1     2     3
     1     2     3
     1     2     3

X3_slice = X3(:,:,2)
X3_slice =
     2     2     2
     2     2     2
     2     2     2
```

The concept of data gridding is very important for understanding the ways in which MATLAB does grid-based interpolation.

Grid-Based Interpolation

- "Benefits of Using Grid-Based Interpolation" on page 8-19
- "Interpolation versus Fit" on page 8-22
- "Interpolation Methods" on page 8-22

In grid-based interpolation, the data to be interpolated is represented by an ordered grid. For example, an arrangement of temperature measurements over a rectangular flat surface at 1-cm intervals from top-to-bottom vertically and left-to-right horizontally is considered 2-D gridded data. Grid-based interpolation provides an efficient way to approximate the temperature at any location between the grid points.

Benefits of Using Grid-Based Interpolation

Grid-based interpolation provides significant savings in computational overhead because the gridded structure allows MATLAB to locate a query point and its adjacent neighbors very quickly. To understand how this works, consider the following points on a 1-D grid:
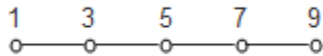
The lines connecting adjacent points represent the cells of the grid. The first cell occurs between x = 1 and x = 3, the second occurs between x = 3 and x = 5, and so on. Each number represents a coordinate in the grid. If you want to query the grid at x = 6, you would have to use interpolation because 6 is not explicitly defined in the grid. Since this grid has a uniform spacing of 2, you can narrow down the location of the query point with a single integer division (6/2 = 3). This tells you that the point is in the 3rd cell of the grid. Locating a cell in a 2-D grid involves performing this operation once in each dimension. This operation is called a fast lookup, and MATLAB uses this technique only when the data is arranged in a uniform grid.

This fast lookup efficiently locates the cell containing a query point Xq. A binary search proceeds as follows:

1 Locate the center grid point.

2   Compare Xq to the point at the center of the grid.

3   If Xq is less than the point found at the center, eliminate all of the grid points greater than central point from the search. Similarly, if Xq is greater than the one found at the center, we eliminate all of the grid points that are less than the central point. Note that by doing this, we have reduced the number of points we must search by half.

4   Find the center of the remaining grid points and repeat from Step 2 until you are left with one grid point on either side of your query. These two points mark the boundary of the cell that contains Xq.
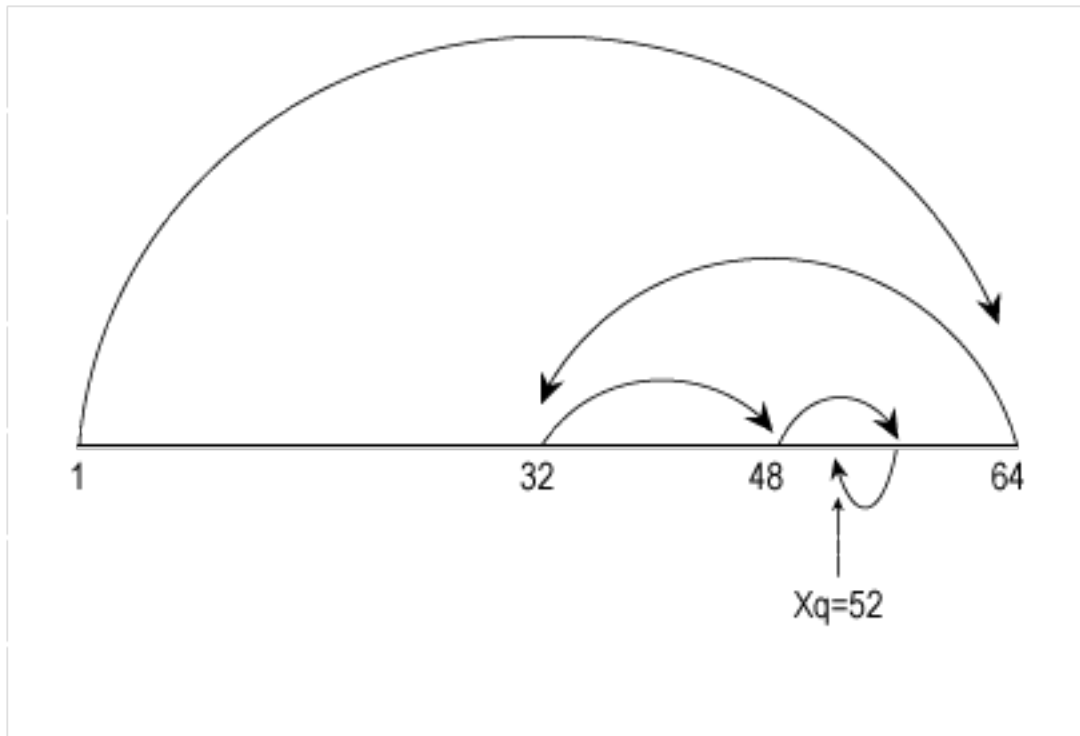
To illustrate the power of binary searches, consider the following example. Before the advent of electronic credit card authorizations, the only protection merchants had against fraudulent credit card purchases was to compare the account number on each customer's credit card against a list of "bad" account numbers. These lists were bound booklets with tens of thousands of card numbers arranged in ascending order. How many comparisons

would be required to search through a list of 10,000 account numbers for one sale? It turns out that for any list of n ordered items, the maximum number of comparisons will be no more than the number of times you can divide the list in half, or `log2(n)`. Therefore, the credit card search will take no more than `log2(10e3)` or about 13 comparisons. That's a pretty impressive if you consider how many comparisons it would take to perform a sequential search.

By contrast, consider a problem with a scattered data set.

```
x = rand(20,1);
y = rand(20,1);
scatter(x,y)
```
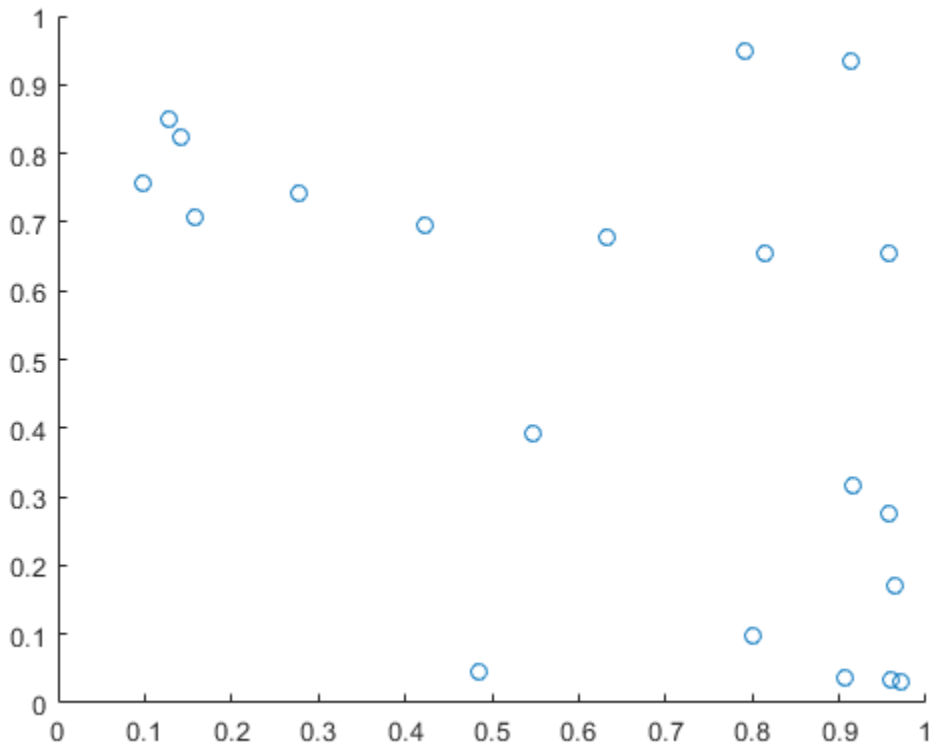
To find the points in close proximity to a query point would require many more operations. If your data can be approximated as a grid, grid-based interpolation will provide substantial savings in computation and memory usage.

If your data is scattered, you can use the tools detailed in "Interpolating Scattered Data" on page 8-55.

Interpolation versus Fit

The interpolation methods available in MATLAB create interpolating functions that pass though the sample data points. If you were to query the interpolation function at a sample location, you would get back the value at that sample data point. By contrast, curve and surface fitting algorithms do not necessarily pass through the sample data points.

Interpolation Methods

Grid-based interpolation offers several different methods for interpolation. When choosing an interpolation method, keep in mind that some require more memory or longer computation time than others. However, you may need to trade off these resources to achieve the desired smoothness in the result. The following table provides an overview of the benefits, trade-offs, and requirements for each method.

Method	Description	Continuity	Memory Usage and Performance	Requirements
Nearest Neighbor	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.
Next Neighbor	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 1D interpolation only. • Requires at least 2 grid points.

Method	Description	Continuity	Memory Usage and Performance	Requirements
Previous Neighbor	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 1D interpolation only. • Requires at least 2 grid points.
Linear	The interpolated value at a query point is based on linear interpolation of the values at neighboring grid points in each respective dimension.	C0	• Requires more memory than nearest neighbor. • Requires more computation time than nearest neighbor.	• Requires at least 2 grid points in each dimension.
Pchip	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	• Requires more memory than linear. • Requires more computation time than linear.	• Available for 1D interpolation only. • Requires at least 4 grid points.

Method	Description	Continuity	Memory Usage and Performance	Requirements
Cubic	The interpolated value at a query point is based on cubic interpolation of the values at neighboring grid points in each respective dimension.	C1	• Requires more memory than linear. • Requires more 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.
Modified Akima	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	• 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

Method	Description	Continuity	Memory Usage and Performance	Requirements
Spline	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	• Requires more memory than cubic. • Requires more computation time than cubic.	• Requires 4 points in each dimension.

This figure has some visual comparisons of the different interpolation methods for 1-D data.

Comparison of Interpolation Methods
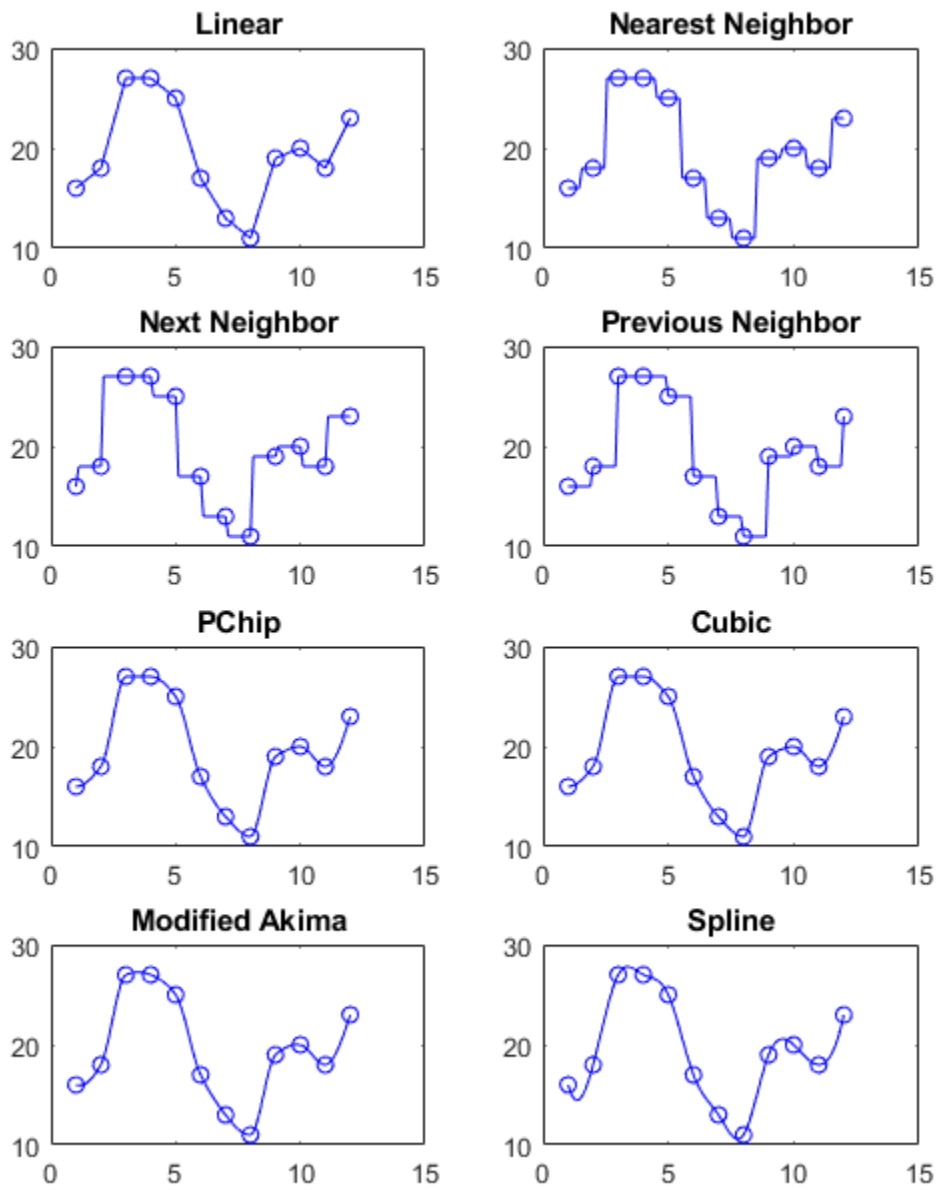
MATLAB provides support for grid-based interpolation in several ways:

- The `interp` family of functions: `interp1`, `interp2`, `interp3`, and `interpn`.
- The `griddedInterpolant` class.

Both the `interp` family of functions and `griddedInterpolant` support N-D grid-based interpolation. However, there are memory and performance benefits to using the `griddedInterpolant` class over the `interp` functions. Moreover, the `griddedInterpolant` class provides a single consistent interface for working with gridded data in any number of dimensions.

Interpolation with the interp Family of Functions

- "The interp1 Function" on page 8-27
- "1-D Extrapolation With interp1" on page 8-28
- "The interp2 Function" on page 8-31
- "The interp3 Function" on page 8-34
- "The interpn Function" on page 8-35

The interp1 Function

This example shows how the `interp1` function can be used to interpolate a set of sample values using the `'pchip'` method.

The function `interp1` performs one-dimensional interpolation. Its most general form is:

```
Vq = interp1(X,V,Xq,method)
```

where X is a vector of coordinates and V is a vector containing the values at those coordinates. Xq is a vector containing the query points at which to interpolate, and `method` optionally specifies any of four interpolation methods: `'nearest'`, `'linear'`, `'pchip'`, or `'spline'`.

Create a set of 1-D grid points X and corresponding sample values V.

```
X = [1 2 3 4 5];
V = [12 16 31 10 6];
```

Interpolate over finer intervals with 0.1 spacing.

```
Xq = (1:0.1:5);
Vq = interp1(X,V,Xq,'pchip');
```

Plot the samples and interpolated values.

```
plot(X,V,'o');
hold on
plot(Xq,Vq,'-');
legend('samples','pchip');
hold off
```
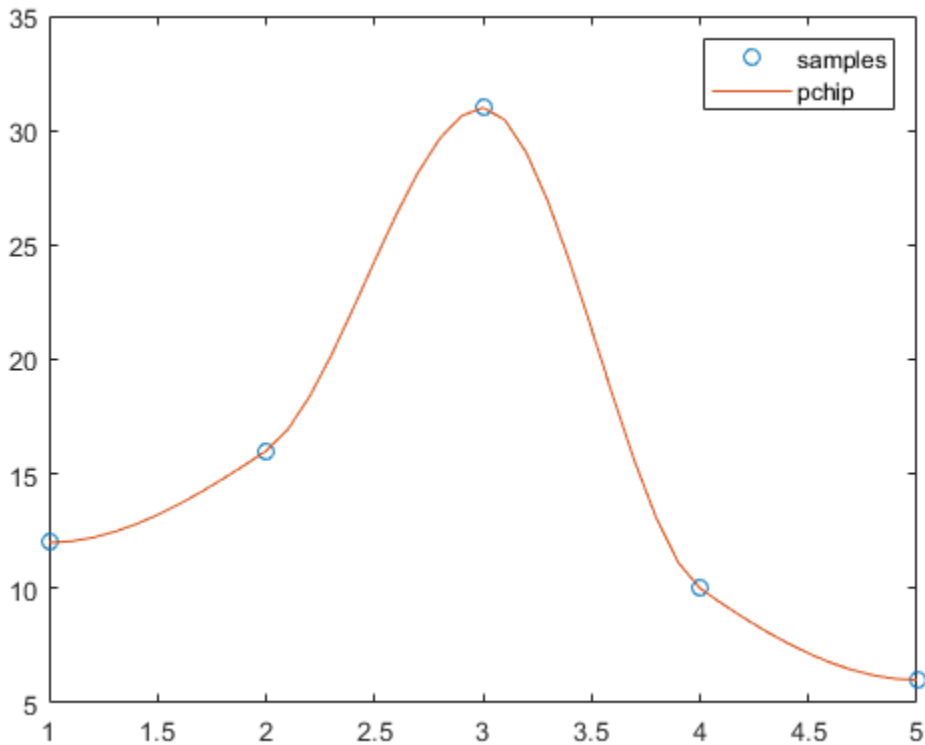

1-D Extrapolation With interp1

This example shows how to use the `'extrap'` option to interpolate beyond the domain of your sample points.

Define the sample points and values.

```
X = [1 2 3 4 5];
V = [12 16 31 10 6];
```

Specify the query points, Xq, that extend beyond the domain of X.

```
Xq = (0:0.1:6);
Vq = interp1(X,V,Xq,'pchip','extrap');
```

Plot the results.

```
figure
plot(X,V,'o');
hold on
plot(Xq,Vq,'-');
legend('samples','pchip');
hold off
```
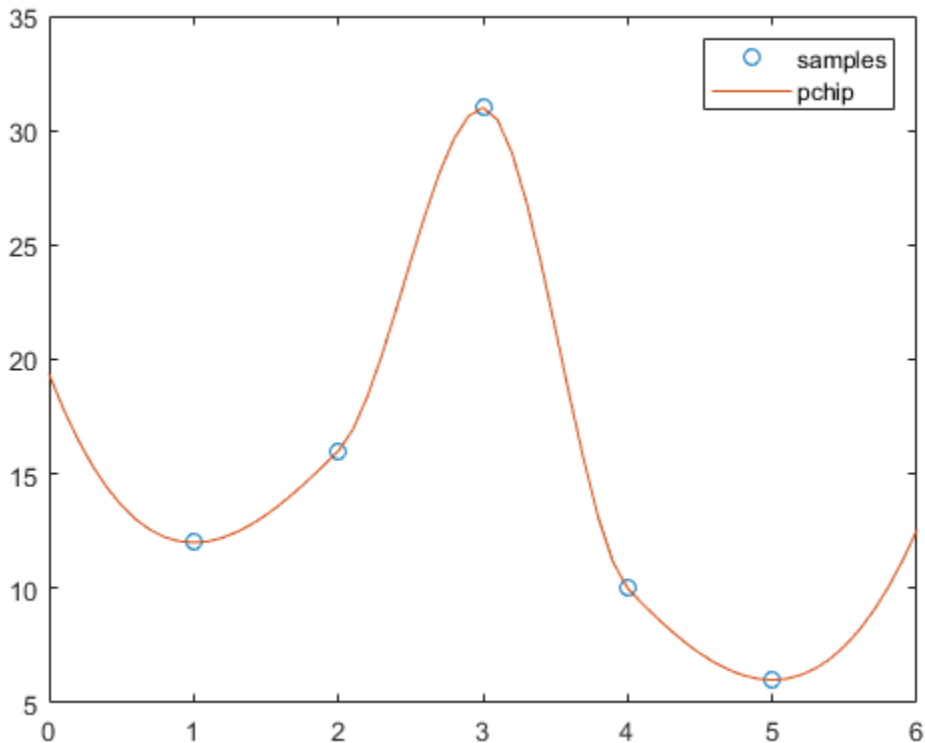
In an alternative approach, you can introduce additional points to gain more control over the behavior in the extrapolated regions. For example, you can constrain the curve to remain flat in the extrapolated region by extending the domain with repeated values.

```
X = [0 1 2 3 4 5 6];
V = [12 12 16 31 10 6 6];
```

Specify the query points, Xq, that extend further beyond the domain of X.

```
Xq = (-1:0.1:7);
```

Interpolate using 'pchip'. You can omit the 'extrap' option because it is the default with the 'pchip', 'makima', and 'spline' methods.

```
Vq = interp1(X,V,Xq,'pchip');
```

Plot the results.

```
figure
plot(X,V,'o');
hold on
plot(Xq,Vq,'-');
legend('samples','pchip');
hold off
```
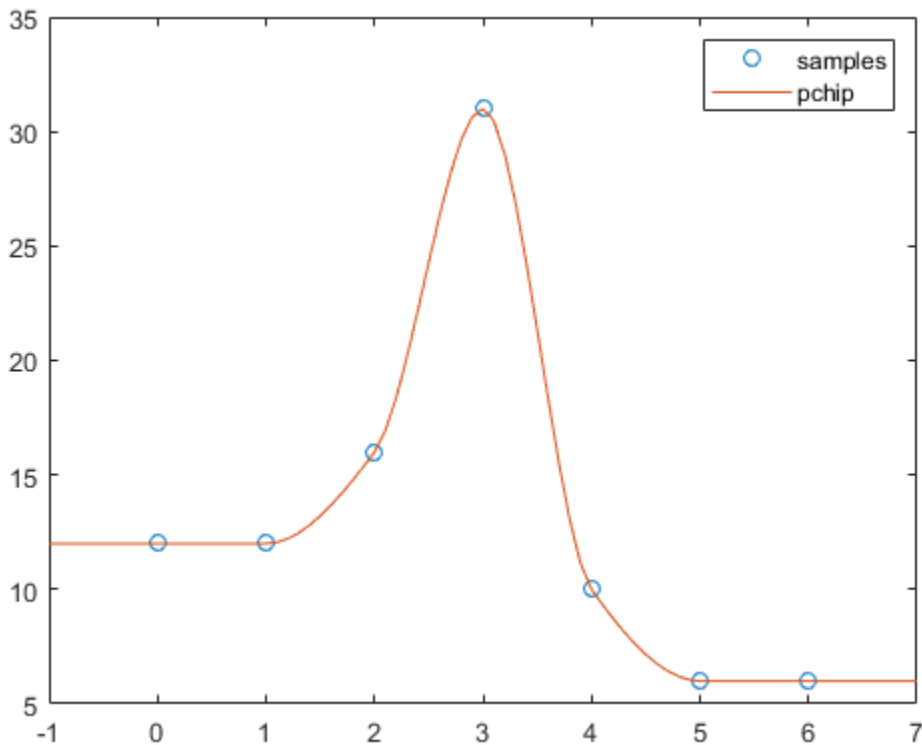

The interp2 Function

This example shows how the `interp2` function can be used to interpolate the coarsely sampled `peaks` function over a finer grid.

The `interp2` and `interp3` functions perform two and three-dimensional interpolation respectively, and they interpolate grids in the `meshgrid` format. The calling syntax for `interp2` has the general form:

```
Vq = interp2(X,Y,V,Xq,Yq,method)
```

where X and Y are arrays of coordinates that define a grid in `meshgrid` format, and V is an array containing the values at the grid points. Xq and Yq are arrays containing the coordinates of the query points at which to interpolate. `method` optionally specifies any of four interpolation methods: `'nearest'`, `'linear'`, `'cubic'`, or `'spline'`.

The grid points that comprise X and Y must be monotonically increasing and should conform to the `meshgrid` format.

Create a coarse grid and corresponding sample values.

```
[X,Y] = meshgrid(-3:1:3);
V = peaks(X,Y);
```

Plot the sample values.

```
surf(X,Y,V)
title('Sample Grid');
```

Sample Grid
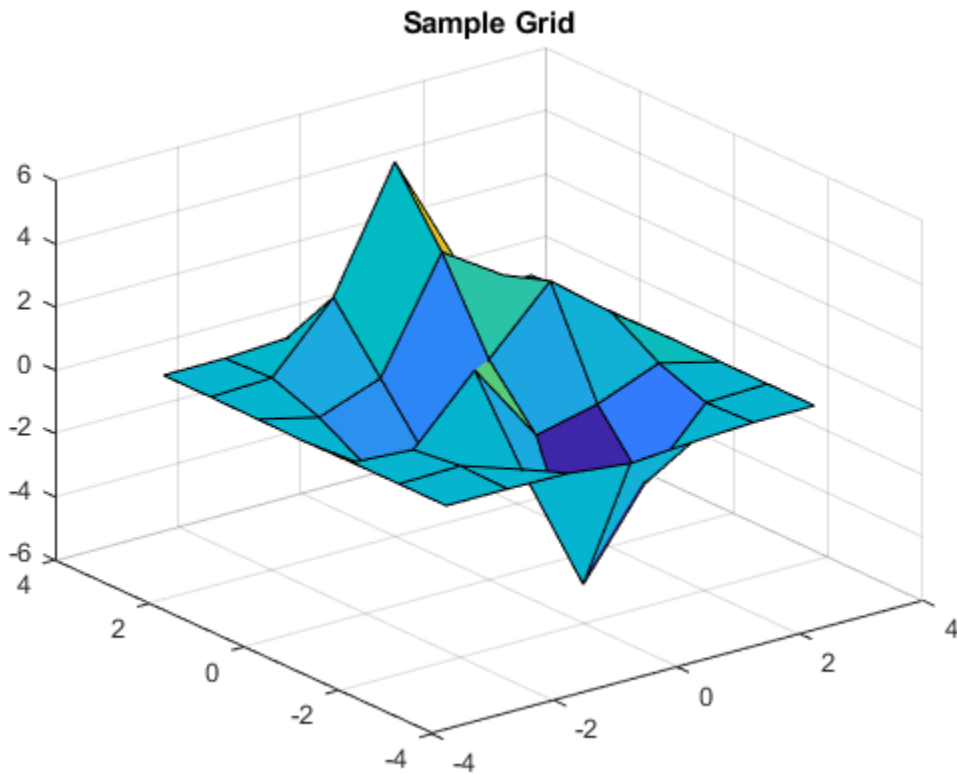
Generate a finer grid for interpolation.

```
[Xq,Yq] = meshgrid(-3:0.25:3);
```

Use interp2 to interpolate at the query points.

```
Vq = interp2(X,Y,V,Xq,Yq,'linear');
```

Plot the results.

```
surf(Xq,Yq,Vq);
title('Refined Grid');
```

Refined Grid
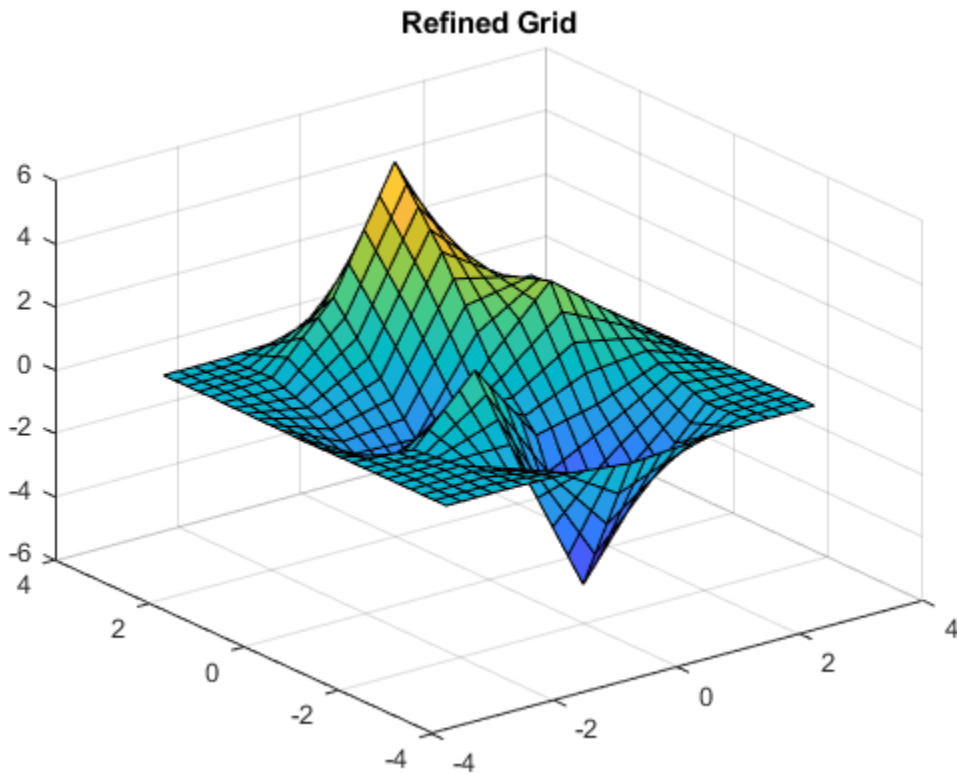
The interp3 Function

This example shows how to use `interp3` to interpolate a 3-D function at a single query point and compare it to the value generated by an analytic expression.

`interp3` works in the same way as `interp2` except that it takes two additional arguments: one for the third dimension in the sample grid and the other for the third dimension in the query points,

`Vq = interp3(X,Y,Z,V,Xq,Yq,Zq,method)`.

As is the case with `interp2`, the grid points you supply to `interp3` must be monotonically increasing and should conform to the `meshgrid` format.

Define a function that generates values for X, Y, and Z input.

```
generatedvalues = @(X,Y,Z)(X.^2 + Y.^3 + Z.^4);
```

Create the sample data.

```
[X,Y,Z] = meshgrid((-5:.25:5));
V = generatedvalues(X,Y,Z);
```

Interpolate at a specific query point.

```
Vq = interp3(X,Y,Z,V,2.35,1.76,0.23,'cubic')
```

```
Vq = 10.9765
```

Compare Vq to the value generated by the analytic expression.

```
V_actual = generatedvalues(2.35,1.76,0.23)
```

```
V_actual = 10.9771
```

The interpn Function

This example shows how the `interpn` function can be used to interpolate a coarsely sampled function over a finer grid using the `'cubic'` method.

The function `interpn` performs n-dimensional interpolation on grids that are in the `ndgrid` format. Its most general form is:

```
Vq = interpn(X1,X2,X3,...Xn,V,Y1,Y2,Y3,...,Yn,method)
```

where `X1,X2,X3,...,Xn` are arrays of coordinates that define a grid in `ndgrid` format, and `V` is an array containing the values at the grid points. `Y1,Y2,Y3,...,Yn` are arrays containing the coordinates of the query points at which to interpolate. `method` optionally specifies any of four interpolation methods: `'nearest'`, `'linear'`, `'cubic'`, or `'spline'`.

The grid points that comprise `X1,X2,X3,...Xn` must be monotonically increasing and should conform to the `ndgrid` format.

Create a set of grid points and corresponding sample values.

```
[X1,X2] = ndgrid((-5:1:5));
R = sqrt(X1.^2 + X2.^2)+ eps;
V = sin(R)./(R);
```

Plot the sample values.

```
mesh(X1,X2,V)
title('Sample Grid');
```
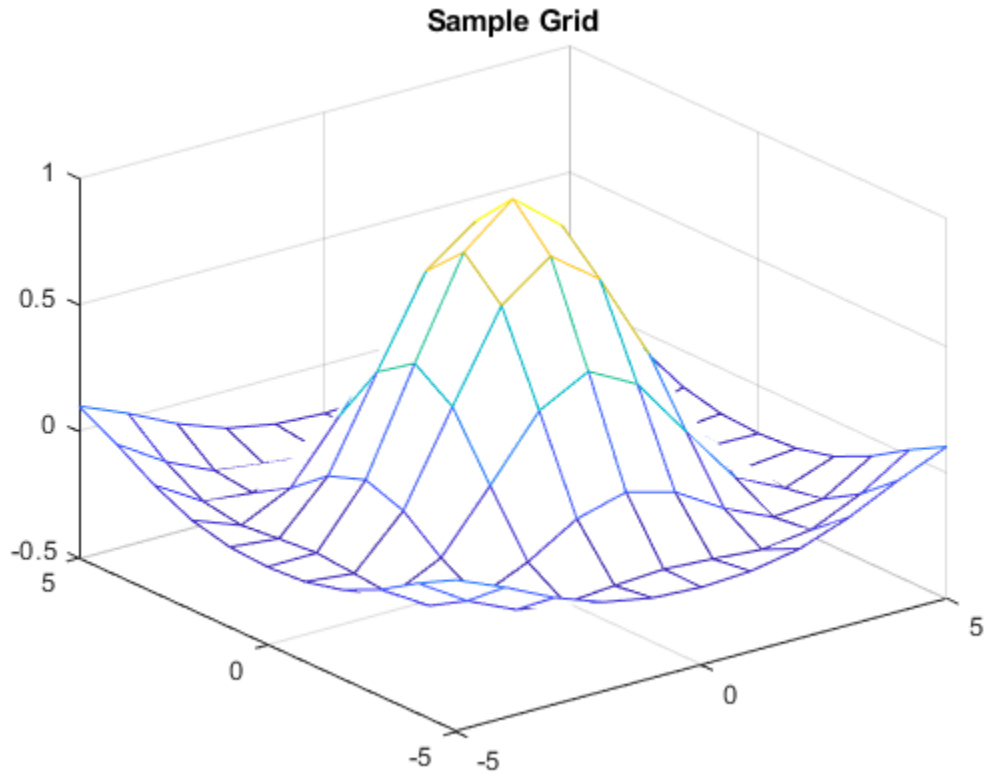

Create a finer grid for interpolation.

```
[Y1,Y2] = ndgrid((-5:.5:5));
```

Interpolate over the finer grid and plot the results.

```
Vq = interpn(X1,X2,V,Y1,Y2,'cubic');
mesh(Y1,Y2,Vq)
title('Refined Grid');
```
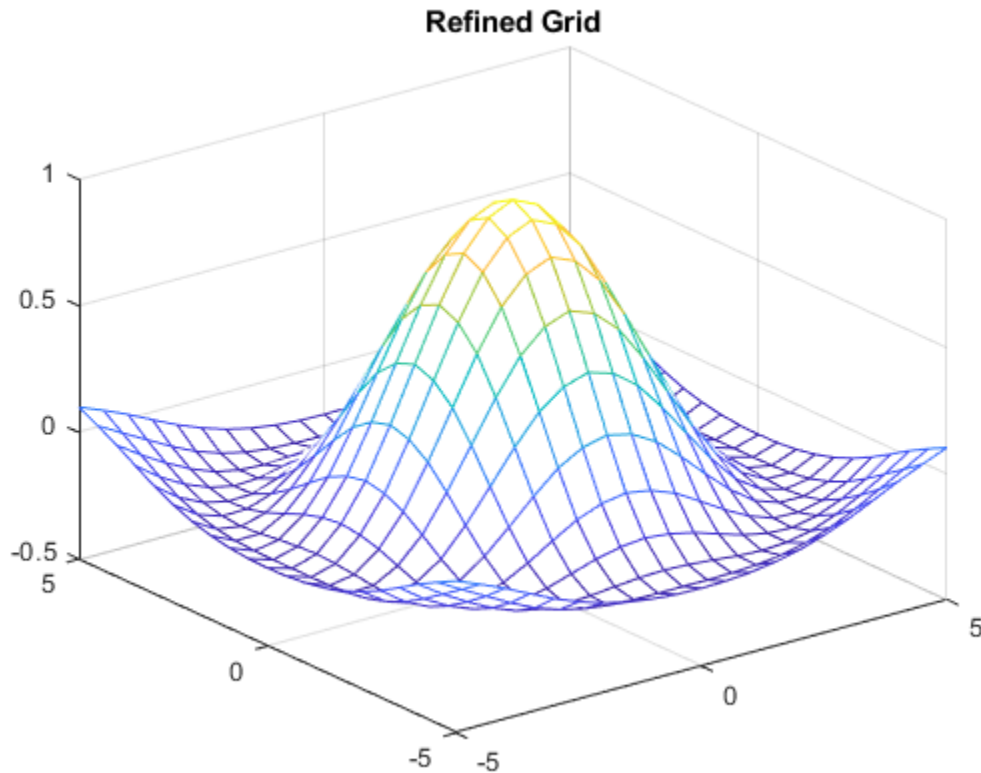
interpn has an alternate syntax: `Vq = interpn(V,ntimes,method)` that allows you to interpolate over a grid that is an integer number of times finer than the sample grid. In the previous code, Y1 and Y2 queried the interpolant over a grid that contained one extra point between each of the samples. The following code demonstrates how you can achieve the same result with `ntimes=1`.

Interpolate over a finer grid using `ntimes=1`.

```
Vq = interpn(V,1,'cubic');
```

Plot the result.

```
mesh(Vq)
title('Refined Grid with NTIMES');
```

Refined Grid with NTIMES
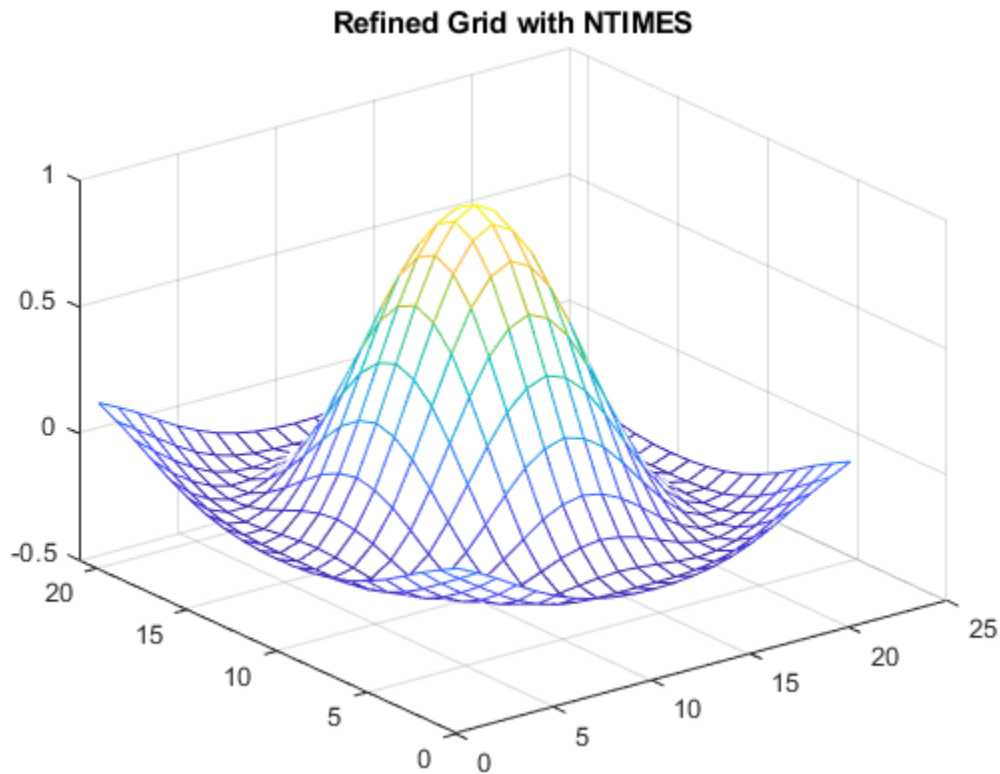
Notice that the plot is scaled differently than in the previous example. This is because we called `mesh` and passed the values only. The `mesh` function used a default grid based on the dimensions of `Vq`. The output values are the same in both cases.

You can also supply a nonuniform grid of query points. This can be useful if you are interested in querying the interpolant at a higher resolution in one region of the grid. The following code shows how this can be done.

Interpolate over a biased grid.

```
[Y1, Y2] = ndgrid([-5 -4 -3 -2.5 -2 -1.5 -1.25 -1 -0.75 -0.5 -0.20 0]);
Vq = interpn(X1,X2,V,Y1,Y2,'cubic');
```

Plot the result.

```
mesh(Y1,Y2,Vq);
title('Biased Grid');
```


Biased Grid
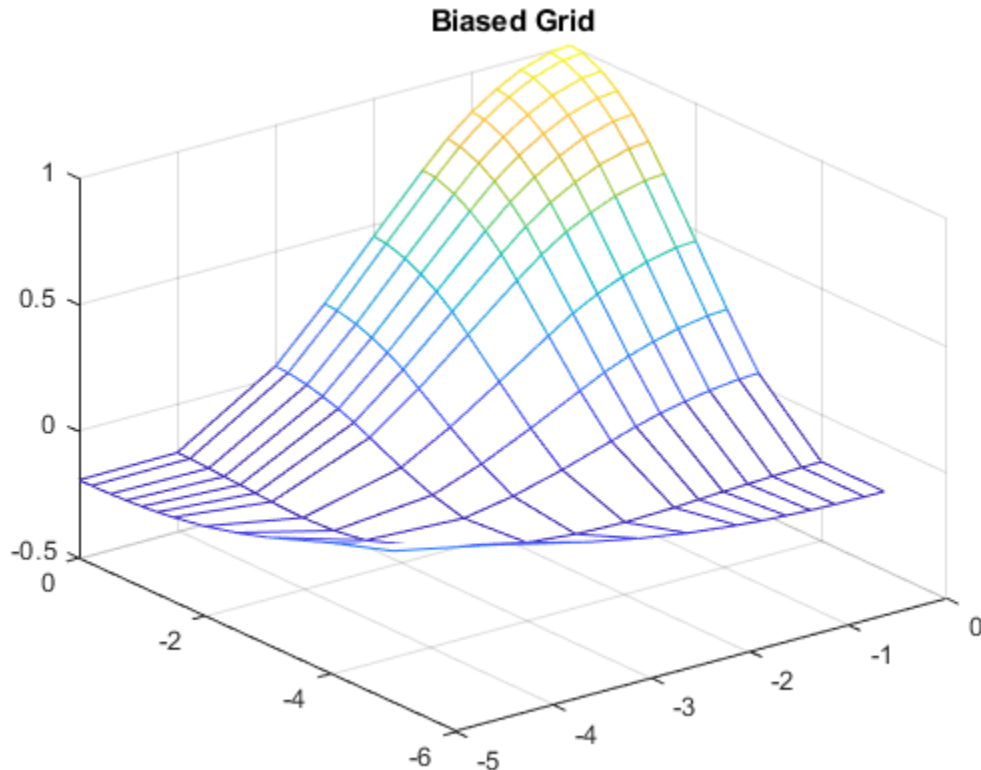
Interpolation with the griddedInterpolant Class

Like the `interpn` function, the `griddedInterpolant` class provides a single interface for grid-based interpolation in *n* dimensions. However `griddedInterpolant` offers the following additional benefits:

- It offers substantial performance improvements for repeated queries to the interpolant.
- It offers additional performance improvements and savings in memory consumption because it stores the sample points as a compact grid on page 8-13.

`griddedInterpolant` accepts sample data that conforms to the `ndgrid` format. If you wish to create a `griddedInterpolant` with `meshgrid` data, you will need to convert the data to the `ndgrid` format. See "Converting meshgrid Data to the ndgrid Format" on page 8-41 for a demonstration of how to convert 2-D and 3-D `meshgrids`.

The `griddedInterpolant` class supports the interpolation methods on page 8-22 that are also supported by `interpn`: `nearest`, `linear`, `pchip`, `cubic`, `makima`, and `spline`. However `griddedInterpolant` offers greater performance with less overhead.

Constructing the Interpolant

An *interpolant* is a function that performs interpolation. You create the interpolant by calling the `griddedInterpolant` constructor and passing the sample data: the grid and corresponding sample values. You can also specify the interpolation method if you wish to override the default "linear" method. The calling syntax has the following forms:

- For 1-D interpolation, you can pass `x`, a set of points, and `v`, a vector of the same length containing the corresponding values.

  ```
  F = griddedInterpolant(x,v)
  ```

- For higher dimensions, you can supply a full grid. `X1,X2,...,Xn` specify the grid as a set of *n* *n*-D arrays. These arrays conform to the `ndgrid` format and are the same size as the sample array `V`.

  ```
   F = griddedInterpolant(X1,X2,...,Xn,V)
  ```

- If you know that the distances between adjacent sample points are uniform, you can let `griddedInterpolant` create a default grid by passing only the sample points `V`.

  ```
   F = griddedInterpolant(V)
  ```

- You can also specify the coordinates of your sample data as a compact grid. The compact grid is represented by a set of vectors. These vectors are then packaged into a cell array by enclosing them in curly braces; for example,`{x1g,x2g,...,xng}` where vectors `x1g,x2g,...,xng` define the grid coordinates in each dimension.

  ```
   F = griddedInterpolant({x1g,x2g,...,xng},V)
  ```

- You can also specify the interpolation method as a final input argument with any of the calling syntaxes. This example specifies nearest neighbor interpolation.

  ```
   F = griddedInterpolant({x1g,x2g,x3g},V,'nearest')
  ```

Querying the Interpolant

The `griddedInterpolant`, F, is evaluated in the same way as you would call a function. Your query points may be scattered or gridded, and you can pass them to F in any of the following ways:

- You can specify an *m*-by-*n* matrix Xq, which contains *m* scattered points in *n* dimensions. The interpolated values Vq are returned as an *m*-by-*1* vector.

  ```
  Vq = F(Xq)
  ```
- You can also specify the query points as a series of *n* column vectors x1q,x2q,...,xnq of length *m*. These vectors represent *m* points in *n* dimensions. The interpolated values Vq are returned as an *m*-by-*1* vector.

  ```
  Vq = F(x1q,x2q,...,xnq)
  ```
- You can specify the query points as a series of *n* *n*-dimensional arrays representing a full grid. The arrays X1q,X2q,...,Xnq are all the same size and conform to the `ndgrid` format. The interpolated values Vq will also be the same size.

  ```
  Vq = F(X1q,X2q,...,Xnq)
  ```
- You can also specify the query points as a compact grid. x1gq,x2gq,...,xngq are vectors that define the grid points in each dimension.

  ```
  Vq = F({x1gq,x2gq,...,xngq})
  ```

  For example, in 2-D:

  ```
  Vq = F({(0:0.2:10),(-5:0.5:5)});
  ```

Converting meshgrid Data to the ndgrid Format

The `griddedInterpolant` class accepts `ndgrid` formatted sample data. If you want to create a `griddedInterpolant` with `meshgrid` data, you should convert it to the `ndgrid` format first.

The following example outlines the steps for converting 2-D `meshgrid` data to the `ndgrid` format. We begin by creating the `meshgrid` and corresponding sample values:

```
[X,Y] = meshgrid(-2:.1:2,-1:.1:1);
V=0.75*Y.^3-3*Y-2*X.^2;
```

To convert X, Y, and V to `ndgrid` format, follow these steps:

1 Transpose each array in the grid as well as the sample data.

```
X=X';
Y=Y';
V=V';
```

2 Now create the interpolant.

```
F = griddedInterpolant(X,Y,V);
```

To convert a 3-D `meshgrid`, use the `permute` function. Again, start by creating the `meshgrid` and corresponding sample values:

```
[X,Y,Z] = meshgrid(-5:5,-3:3,-10:10);
V = X.^3 + Y.^2 + Z;
```

To convert X, Y, Z, and V to `ndgrid` format, follow these steps:

1 Use the `permute` function to interchange the rows and columns of each array. The net effect will be the transpose of every page.

```
P = [2 1 3];
X = permute(X,P);
Y = permute(Y,P);
Z = permute(Z,P);
V = permute(V,P);
```

2 Now create the interpolant.

```
F = griddedInterpolant(X,Y,Z,V);
```

griddedInterpolant in One Dimension

This example shows how to create and plot a 1-D interpolant using `griddedInterpolant` with a cubic interpolation method.

Create a coarse grid and sample values.

```
X = [1 2 3 4 5];
V = [12 6 15 9 6];
```

Construct the `griddedInterpolant` using a cubic interpolation method.

```
F = griddedInterpolant(X,V,'cubic')

F =
  griddedInterpolant with properties:
```

```
        GridVectors: {[1 2 3 4 5]}
             Values: [12 6 15 9 6]
             Method: 'cubic'
  ExtrapolationMethod: 'cubic'
```

The `GridVectors` property contains the compact grid specifying the coordinates of the sample values V. The `Method` property specifies the interpolation method. Notice that we specified `'cubic'` when we created F. If you omit the `Method` argument, the default interpolation method, `linear`, will be assigned to F.

You can access any of the properties of F in the same way you would access the fields in a `struct`.

```
F.GridVectors;          % Displays the grid vectors as a cell array
F.Values;               % Displays the sample values
F.Method;               % Displays the interpolation method
```

Interpolate over finer intervals with 0.1 spacing.

```
Xq = (1:0.1:5);
Vq = F(Xq);
```

Plot the result.

```
plot(X,V,'o');
hold on
plot(Xq,Vq,'-');
legend('samples','Cubic Interpolation');
```
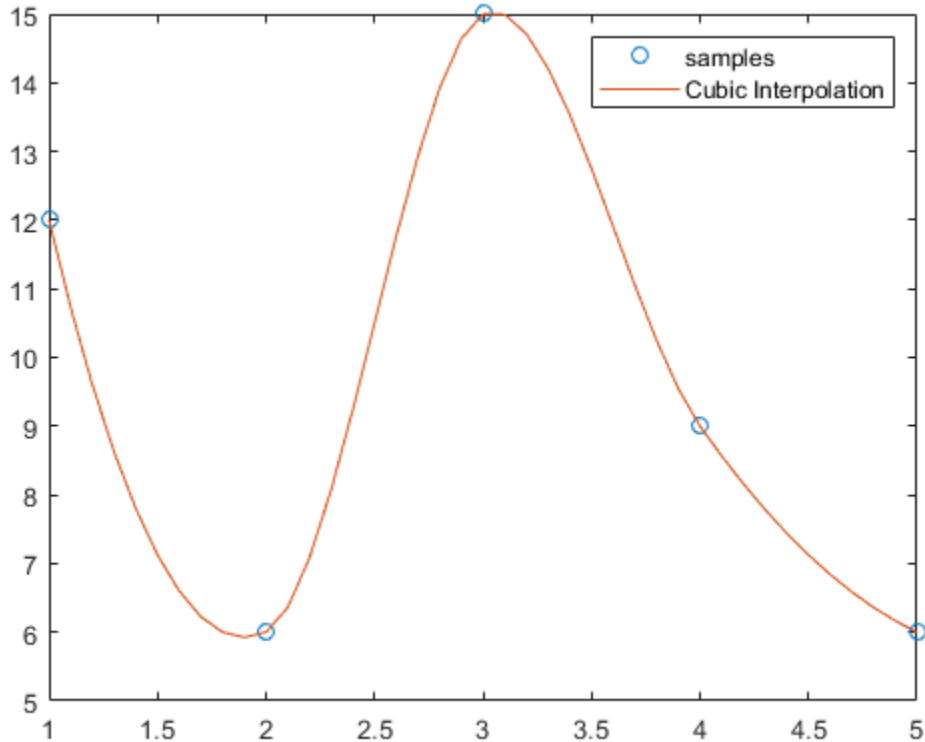
griddedInterpolant in Two Dimensions

This example shows how to create and plot a 2-D interpolant using `griddedInterpolant`.

In two dimensions and higher, you can specify the sample coordinates as an `ndgrid`, a compact grid, or a default grid. In this example, we'll supply an `ndgrid`.

Create a coarse grid and sample values.

```
[X,Y] = ndgrid(-1:.3:1,-2:.3:2);
V = 0.75*Y.^3 - 3*Y - 2*X.^2;
```

Construct the `griddedInterpolant`.

```
F = griddedInterpolant(X,Y,V,'spline');
```

Interpolate over finer intervals with 0.1 spacing.

```
[Xq,Yq] = ndgrid(-1:.1:1,-2:.1:2);
Vq = F(Xq,Yq);
```

Plot the result.

```
figure()
surf(X,Y,V);
view(65,60)
title('Sample Data');
```


Sample Data
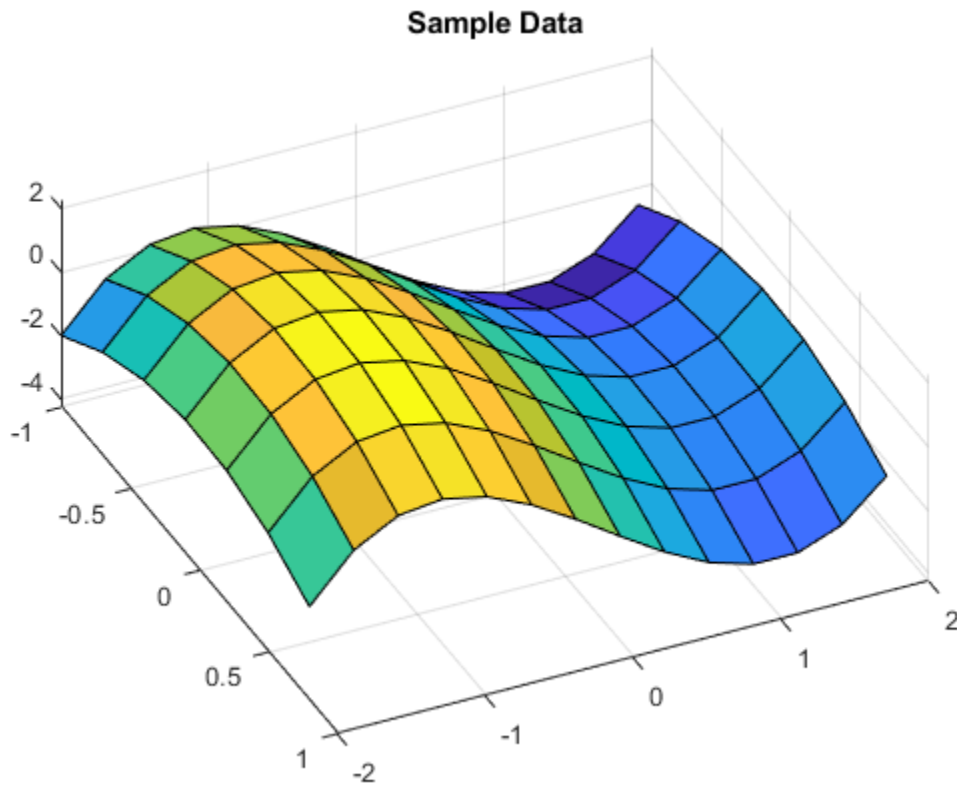
```
figure()
surf(Xq,Yq,Vq);
view(65,60)
title('Refined with Spline');
```


Refined with Spline
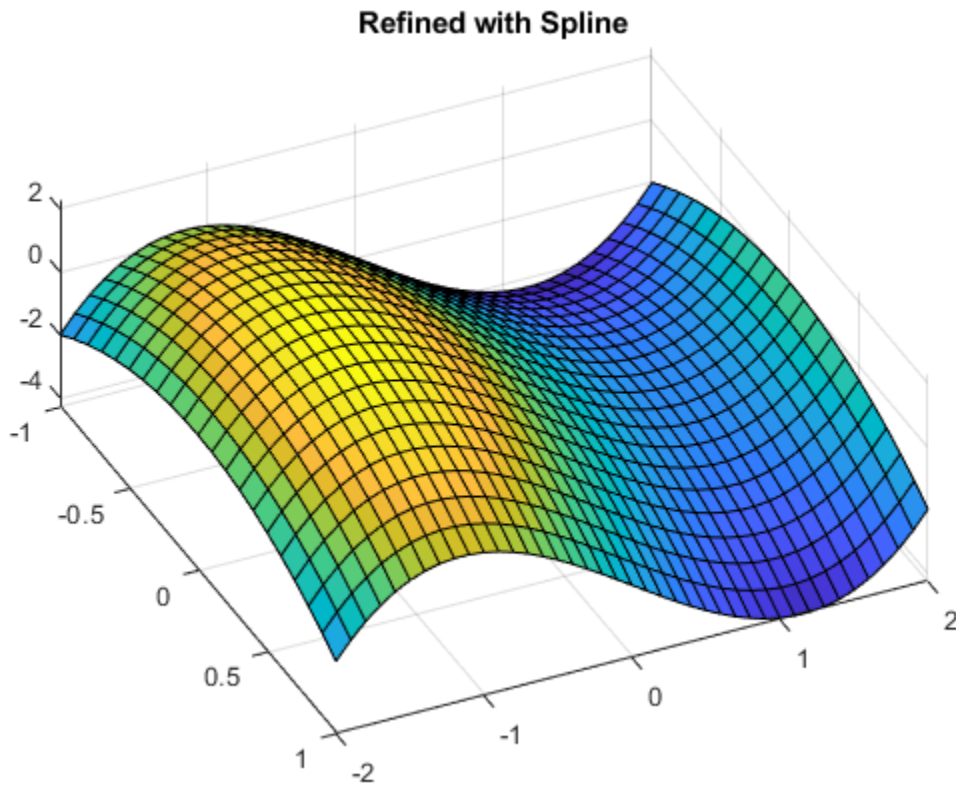
griddedInterpolant in Three Dimensions

This example shows how to create a 3-D interpolant and evaluate over a slice plane so you can plot the values on that plane.

Create a full grid and sample values.

```
[X,Y,Z] = ndgrid((-5:2:5));
V = X.^3 + Y.^2 + Z.^2;
```

Construct the `griddedInterpolant`.

```
F = griddedInterpolant(X,Y,Z,V,'cubic');
```

Since we already created the full grid to generate our sample values, we had nothing to lose in passing it to `griddedInterpolant`. In practice however, it's common to load the sample data into MATLAB from disk. The compact grid can very beneficial in such cases because it allows you to specify the entire grid in a form that is much more economical in terms of memory. If we had loaded V into MATLAB instead of calculating it from a full grid, we could have created a compact grid to conserve memory in our workspace. For example,

```
gv = {(-5:2:5),(-5:2:5),(-5:2:5)};
F = griddedInterpolant(gv,V,'cubic');
```

Now interpolate over a plane at Z = 2 with 0.25 spacing.

```
[Xq,Yq,Zq] = ndgrid((-5:.25:5),(-5:.25:5),2:2);
Vq = F(Xq,Yq,Zq);
```

Plot the result.

```
surf(Xq,Yq,Vq);
title('Refined with Linear Interpolation');
```

Refined with Linear Interpolation
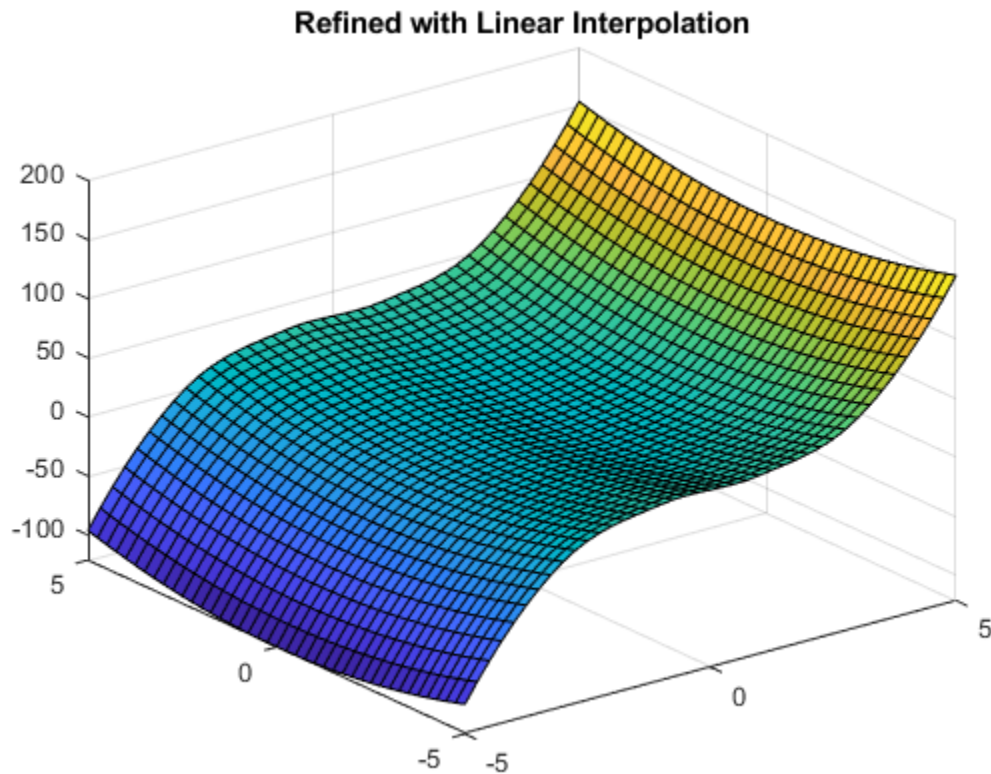
griddedInterpolant in Four Dimensions

This example creates a 4-D interpolant and evaluates it at a single point.

1 Create a coarse grid and sample values.

```
[X1,X2,X3,X4] = ndgrid((-5:2:5));
V = X1.^3 + X2.^2 + X3.^2 +X4;
```

2 Construct the `griddedInterpolant`.

```
F = griddedInterpolant(X1,X2,X3,X4,V,'linear');
```

3 Query the `griddedInterpolant` at a single point.

```
Vq = F([-3.2 2.1 4.7 -1.3])
```

MATLAB outputs the following:

```
ans =

   -10.1000
```

.

Other Ways of Working with griddedInterpolant

Depending on the arrangement of your query points, you may prefer one evaluation syntax over the others. For example, create the following interpolant:

```
[X,Y] = ndgrid(-1:.25:1,-2:.25:2);
V = 0.75*Y.^3-3*Y-2*X.^2;
F = griddedInterpolant(X,Y,V);
```

Query F using a full grid to give the values at the grid points:

```
[Xq,Yq] = ndgrid(-1:.1:0,-2:.1:0);
Vq = F(Xq,Yq);
```

You can also interpolate over the same grid using the compact grid format:

```
gvq = {-1:.1:0,-2:.1:0};
Vq = F(gvq);
```

Or you can query a single point:

```
Vq =  F(.315,.738)
```

which returns:

```
Vq =

   -2.1308
```

or a random set of scattered points:

```
rng('default')
Vq = F(rand(3,2))
```

which returns:

```
Vq =

   -3.4919
   -3.3557
   -0.3515
```

You can also examine the `Values` in F:

```
 F.Values(1,3)
```

which returns:

```
 ans =

   -0.0313
```

Or you can replace the `Values` array:

```
F.Values = 2*V;
```

You can edit the properties in F on-the-fly. For example, you can replace the interpolation method as follows:

```
 F.Method = 'cubic';
```

You can also replace the `GridVectors` in F. First, examine `GridVectors`:

```
gv = F.GridVectors;
gv{1}
```

`gv` is a cell array, and `gv{1}` displays the first grid vector:

```
ans =

  -1.0000   -0.7500   -0.5000   -0.2500    0    0.2500    0.5000    0.7500    1.0000
```

Now replace the `GridVectors` in F by creating a new cell array `new_gv`:

```
new_gv = {(0:0.3:1),(0:0.3:1)};
F.GridVectors = new_gv;
```

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 a 2-D grid of sample points.

```
samplePoints = {x, 1:size(V,2)};
```

This compact notation specifies a full 2-D grid. The first element, samplePoints{1}, contains the *x*-coordinates for V, and samplePoints{2} contains the *y*-coordinates. The orientation of each coordinate vector does not matter.

Create the interpolant, F, by passing the sample points and sample values to griddedInterpolant.

```
F = griddedInterpolant(samplePoints,V);
```

Create a 2-D query grid with 0.5 spacing along x over all columns of V.

```
queryPoints = {(1:0.5:5),1:size(V,2)};
```

Evaluate the interpolant at the *x*-coordinates for each value set.

```
Vq = F(queryPoints)
```

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

*   "Interpolation with the griddedInterpolant Class" on page 8-39

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');
```
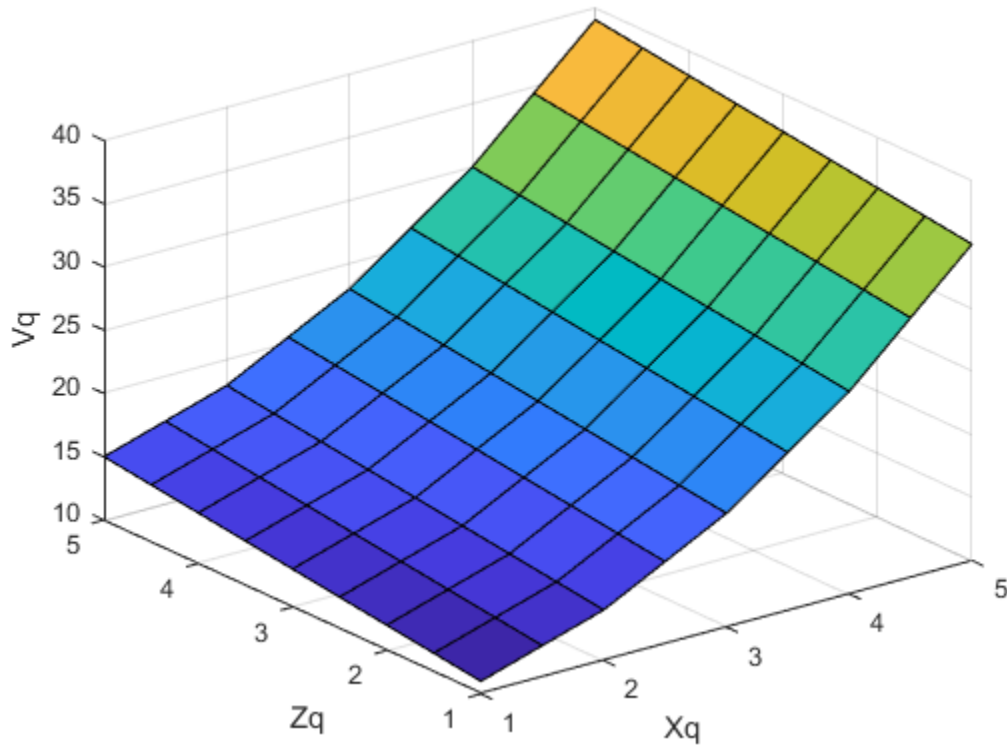
See Also

interpn | squeeze

More About

- "Gridded Data Representation" on page 8-4

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);
```
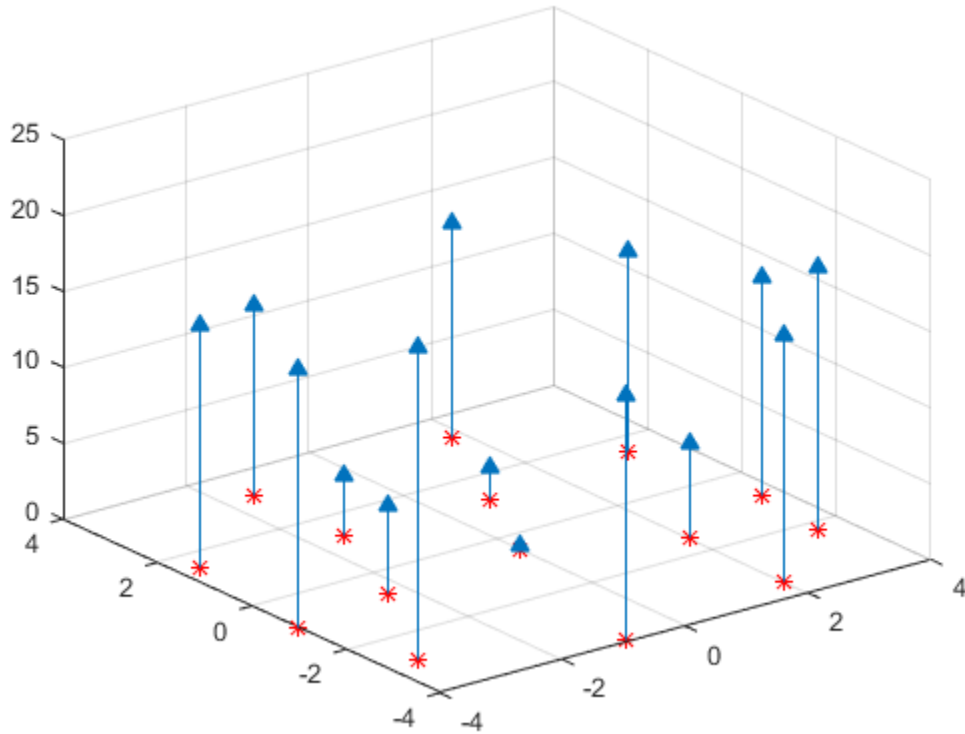
Create a Delaunay triangulation, lift the vertices, and evaluate the interpolant at the query point Xq.

```
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');
```
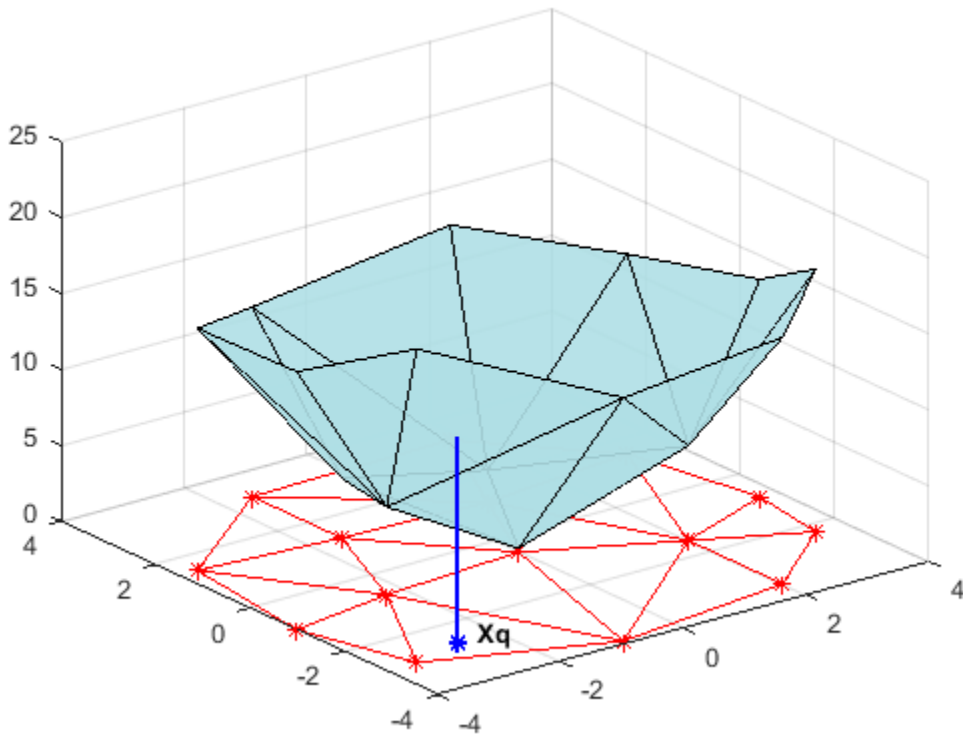

This step generally involves traversing of the triangulation data structure to find the triangle that encloses the query point. Once you find the point, the subsequent steps to compute the value depend on the interpolation method. You could compute the nearest

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
```
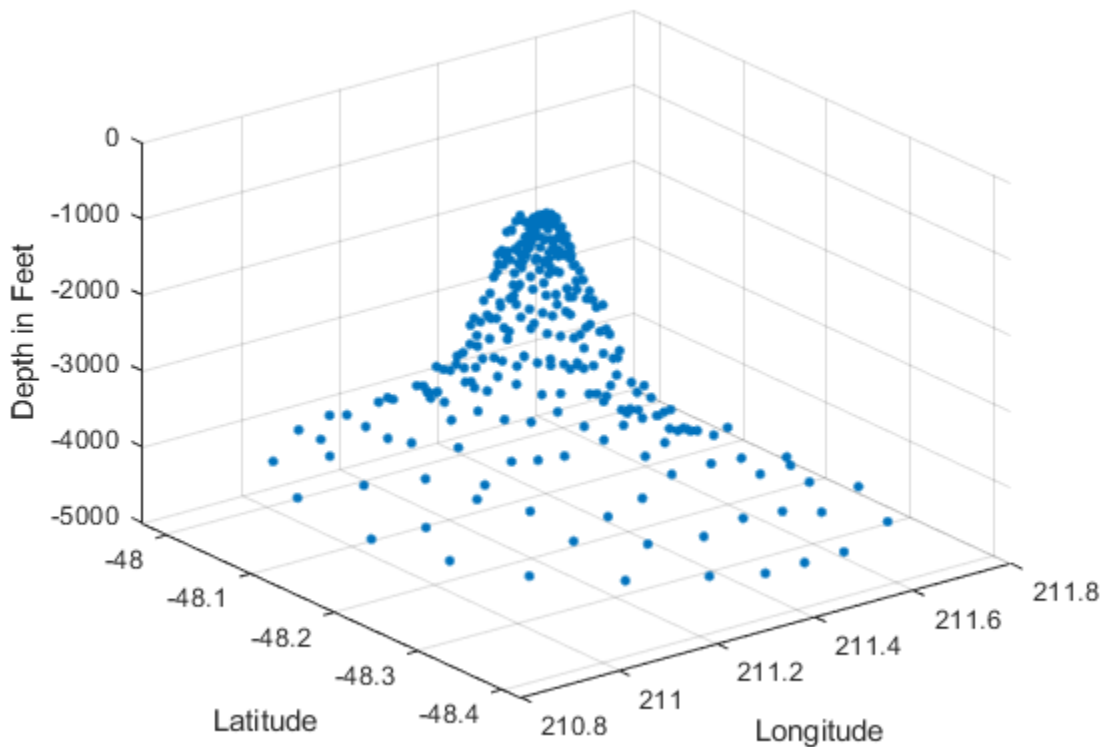

Use `meshgrid` to create a set of 2-D grid points in the longitude-latitude plane and then use `griddata` to interpolate the corresponding depth at those points.

```
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')
```
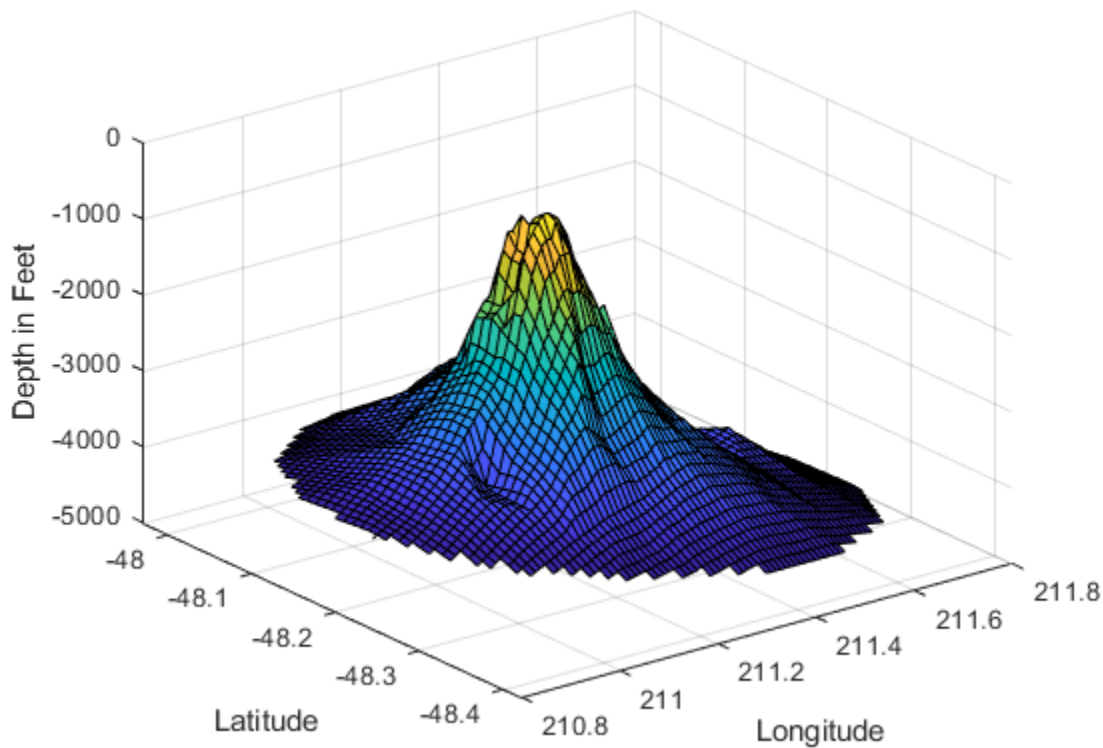

Now that the data is in a gridded format, compute and plot the contours.

```
figure
[c,h] = contour(xi,yi,zi);
clabel(c,h);
```

```
xlabel('Longitude')
ylabel('Latitude')
```
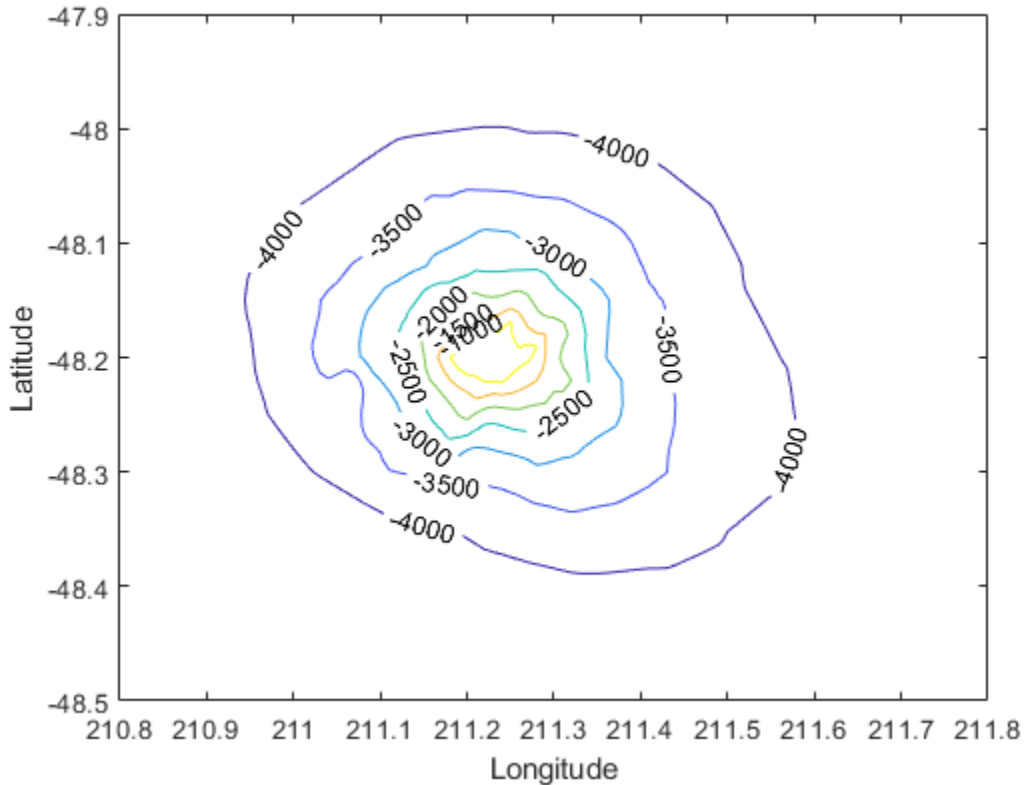

You can also use `griddata` to interpolate at arbitrary locations within the convex hull of the dataset. For example, the depth at coordinates (211.3, -48.2) is given by:

```
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 8-63 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 8-88 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.

```
X = -3 + 6.*gallery('uniformdata',[250 2],0);
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 = 1.3966
```

You can also pass individual coordinates:

```
Vq = F(1.5, 1.25)
```

```
Vq = 1.3966
```

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

    1.0880
    1.8127
    2.3472
```

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');
```

Linear Interpolation Method
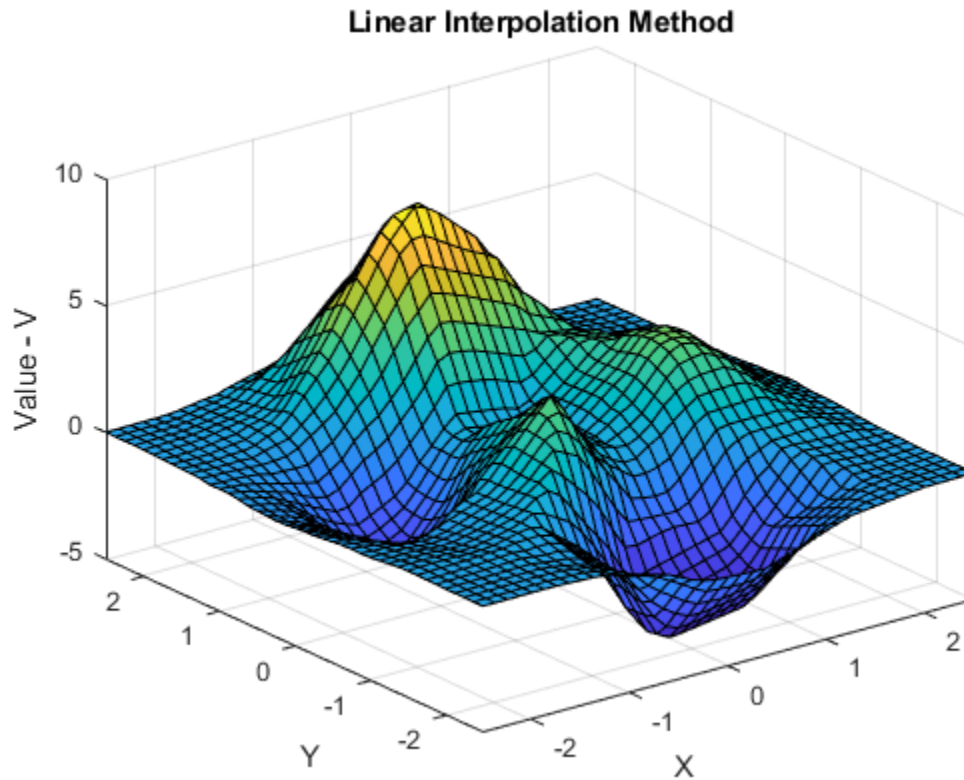
Change the interpolation method.

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');
```
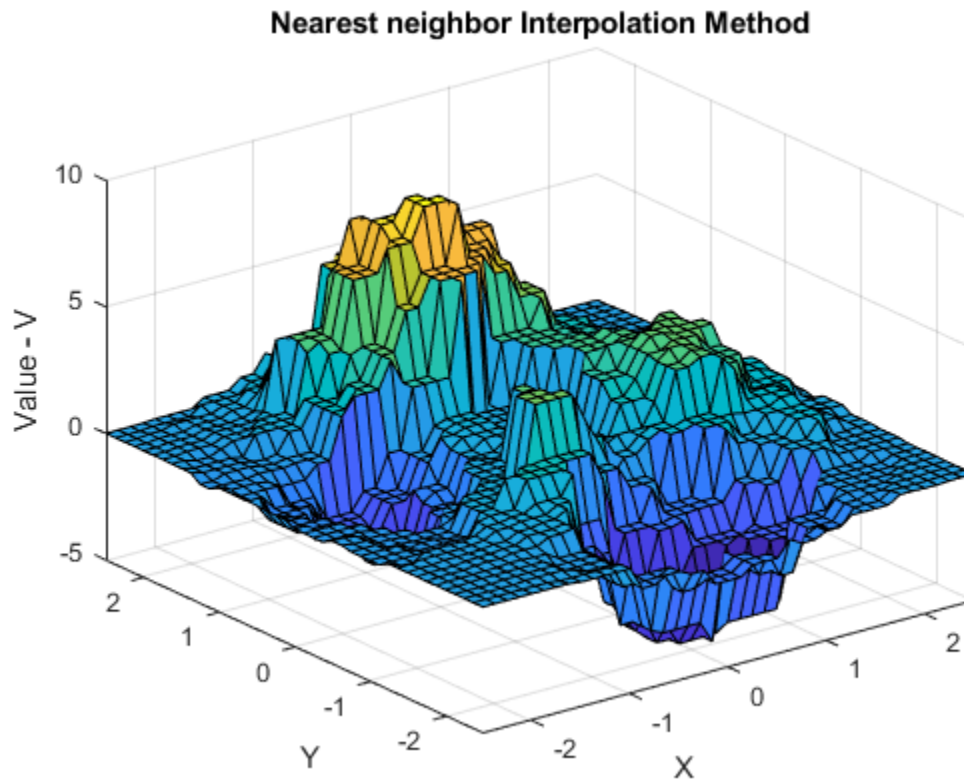
Change the interpolation method to natural neighbor, reevaluate, and plot the results.

```
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');
```

Natural neighbor Interpolation Method
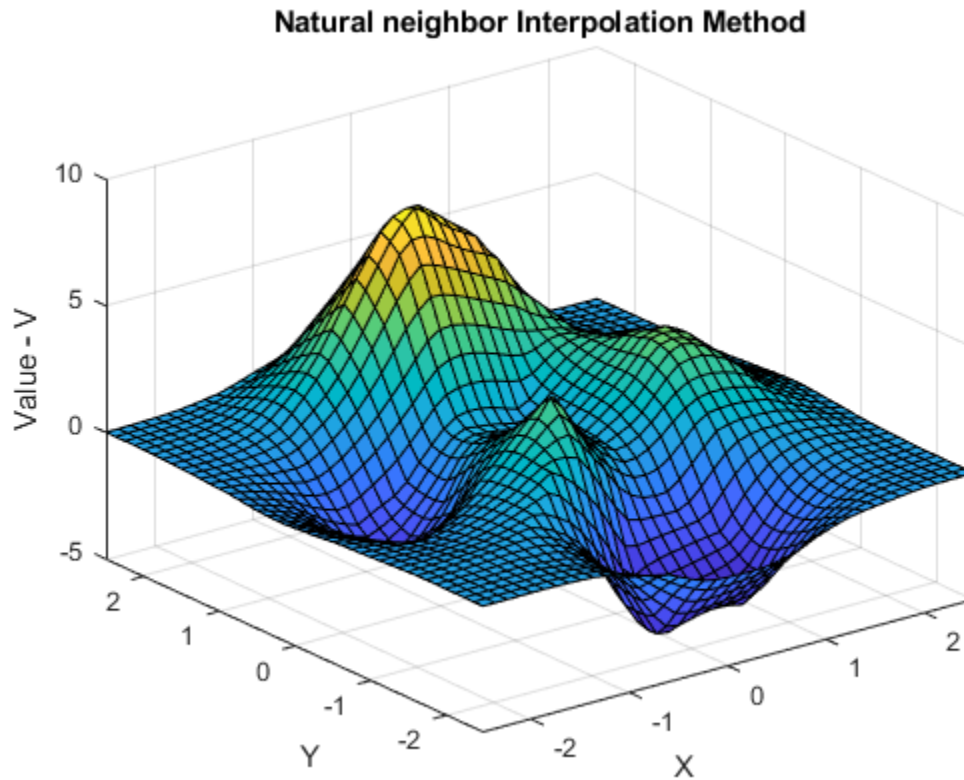
Replace the values at the sample data locations.

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)')
```
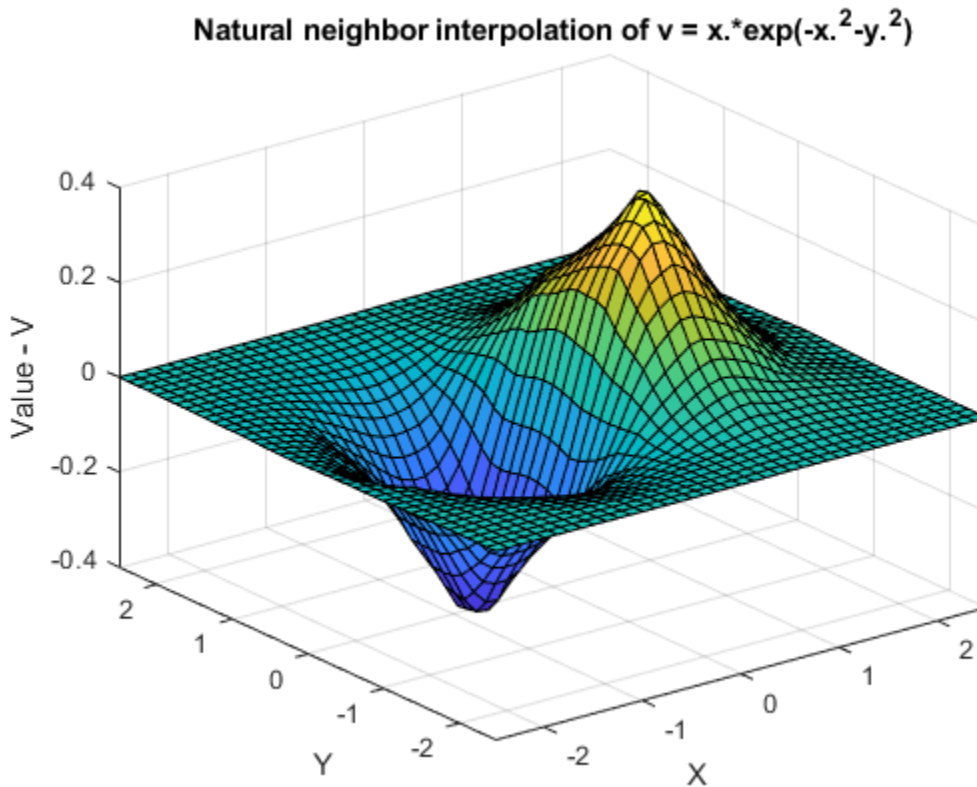
Add additional point locations and values to the existing interpolant.

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');
```
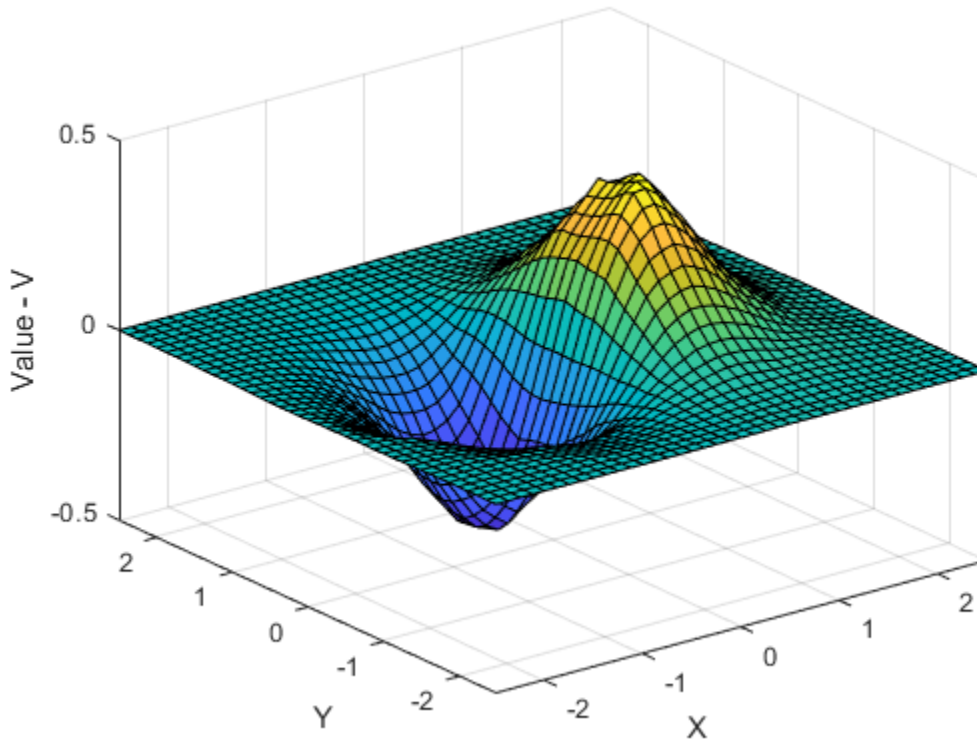
Remove data from the interpolant.

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 v = x.*exp(-x.^{2}-y.^{2}) with sample points removed
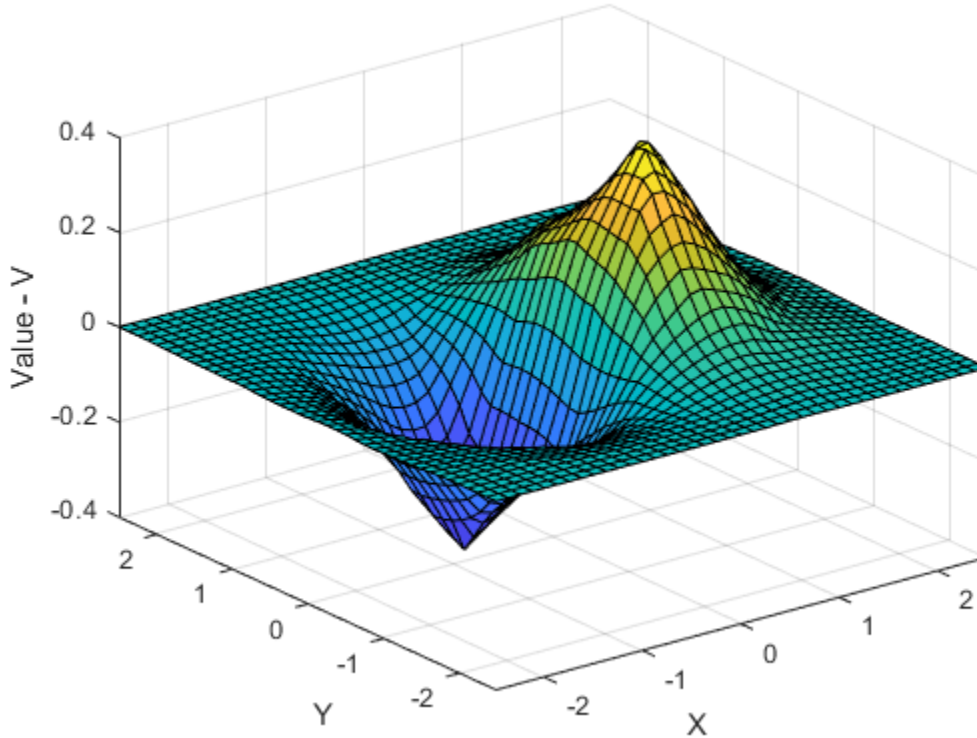
Interpolation of Complex Scattered Data

This example shows how to interpolate scattered data when the value at each sample location is complex.

Create the sample data.

```
X = -3 + 6 .* gallery('uniformdata',[250 2],0);
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');
```
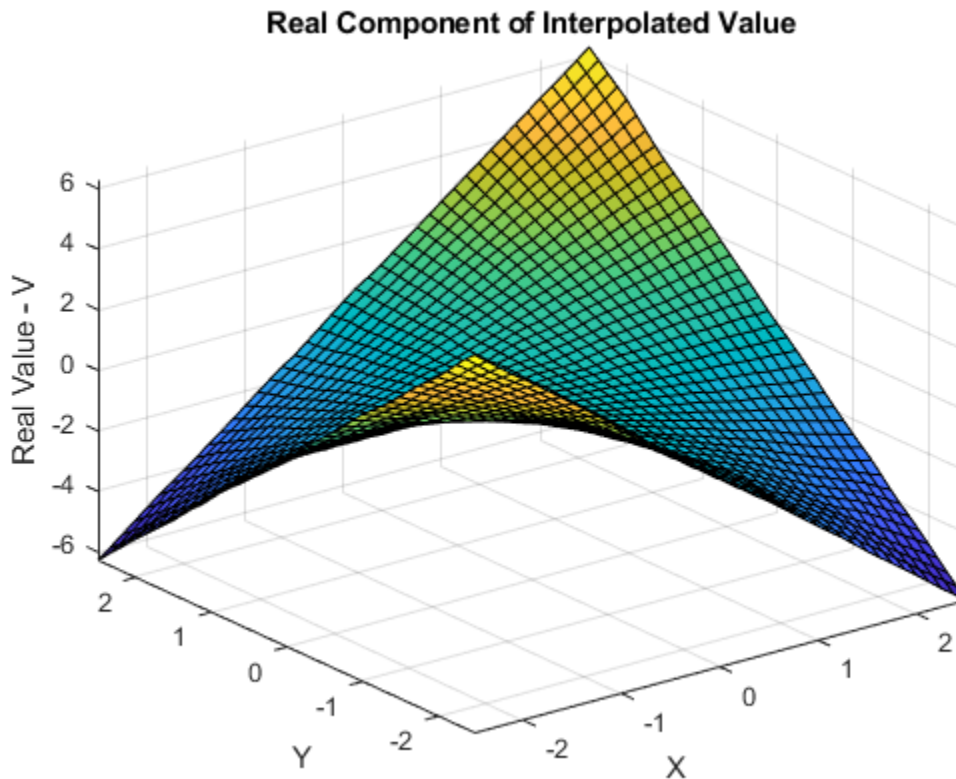
Plot the imaginary component of `Vq`.

```
VqImag = imag(Vq);
figure
surf(Xq,Yq,VqImag);
xlabel('X');
ylabel('Y');
zlabel('Imaginary Value - V');
title('Imaginary Component of Interpolated Value');
```


Imaginary Component of Interpolated Value
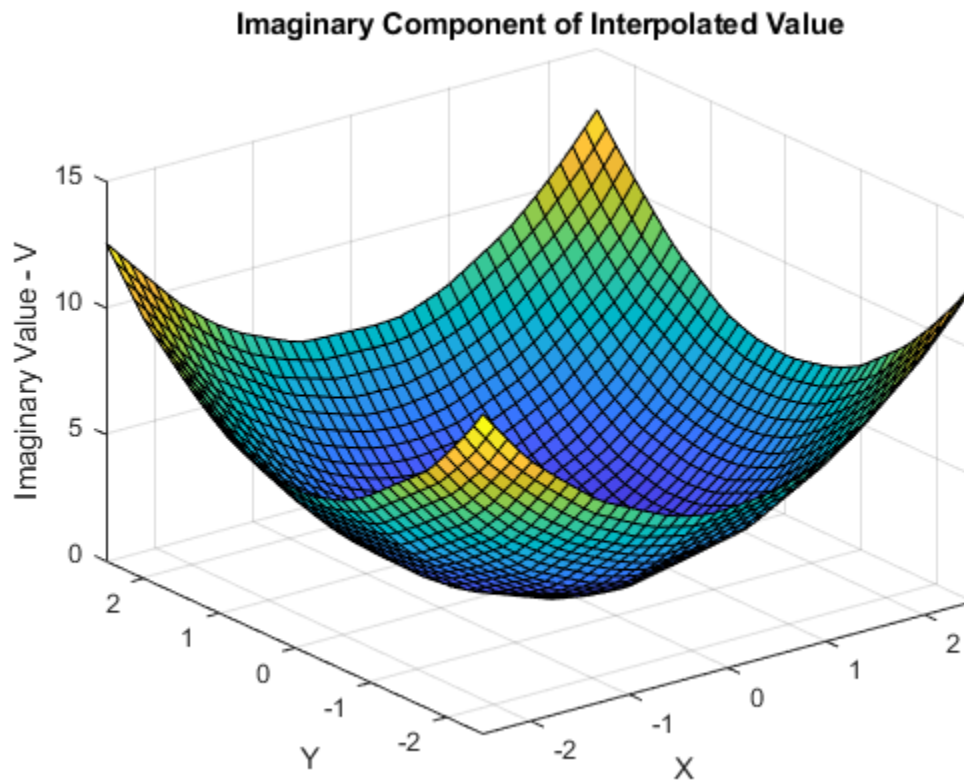
Addressing Problems in Scattered Data Interpolation

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 8-88 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');
```
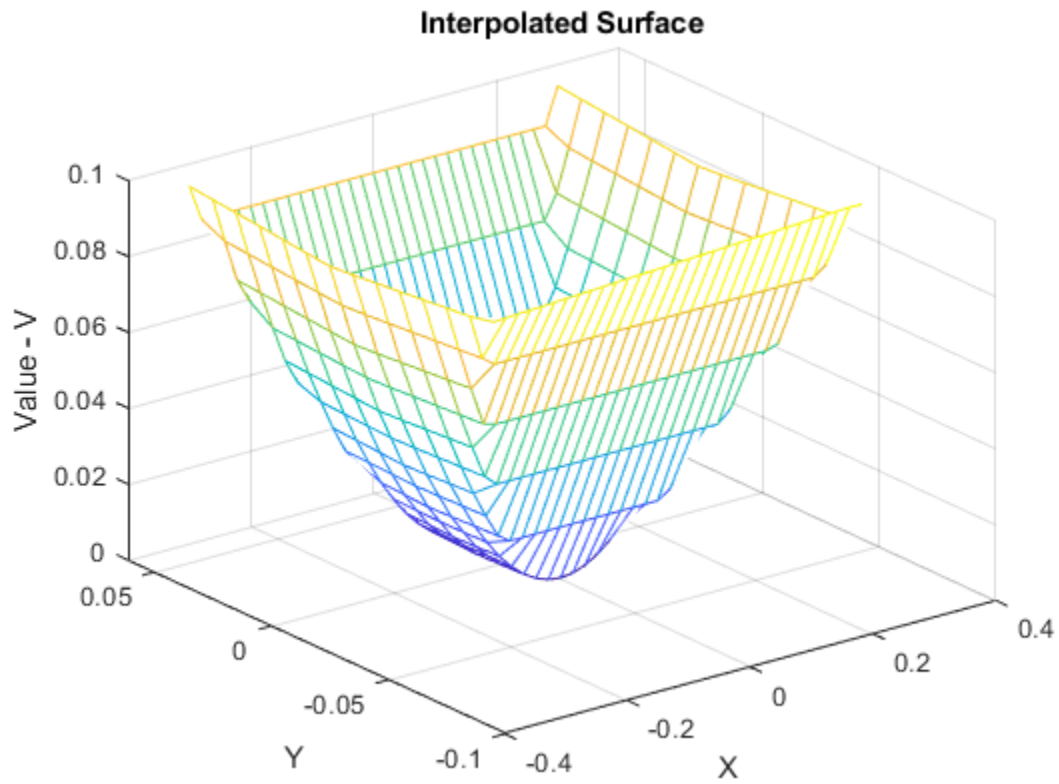
The actual surface is:

```
zi = xi.^2 + yi.^2;
figure
mesh(xi,yi,zi)
title('Actual Surface')
```

Actual Surface
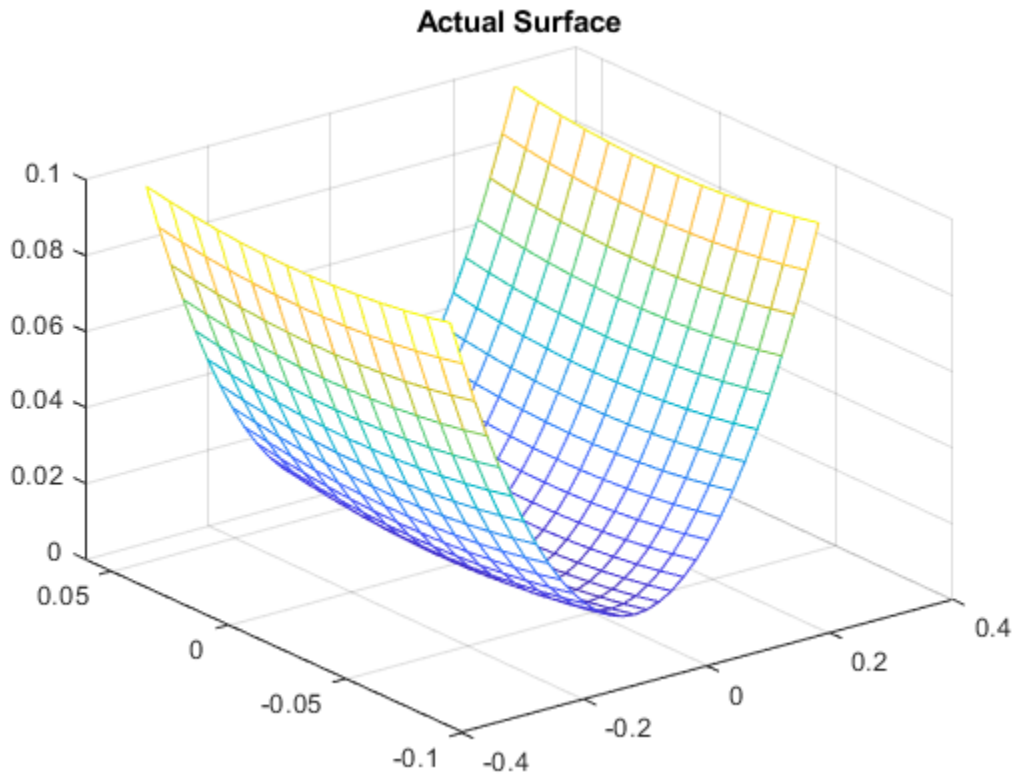
To understand why the interpolating surface deteriorates near the boundary, it is helpful to look at the underlying triangulation:

```
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
```

Triangulation Used to Create the Interpolant
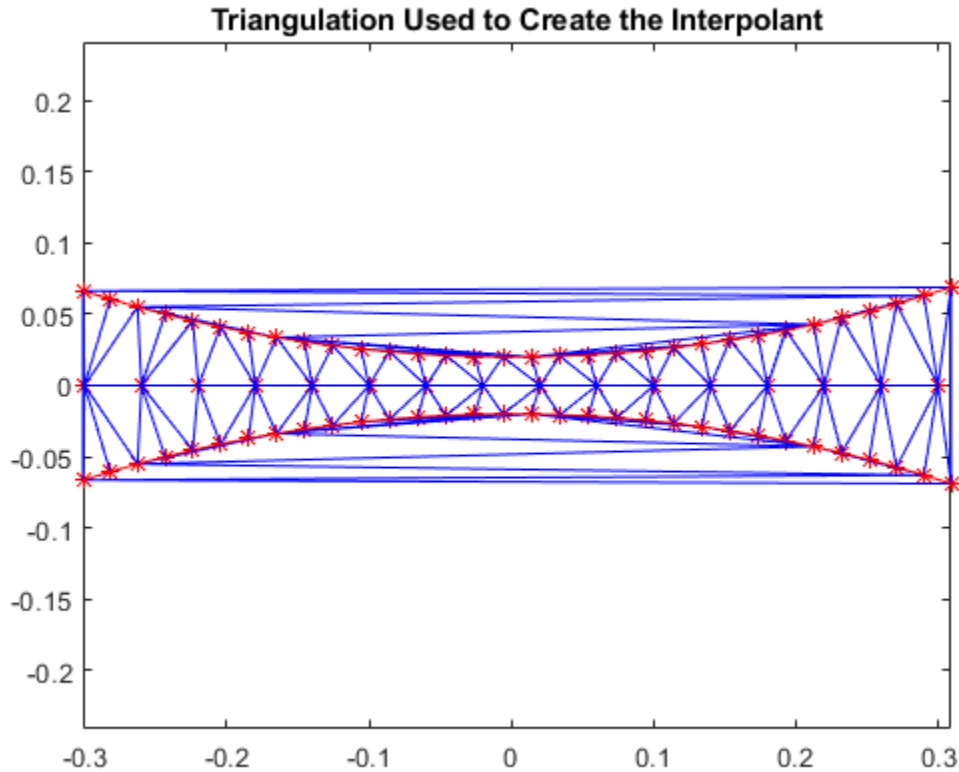
The triangles within the red boundaries are relatively well shaped; they are constructed from points that are in close proximity and the interpolation works well in this region. Outside the red boundary, the triangles are sliver-like and connect points that are remote from each other. There is not sufficient 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');
```


Interpolated surface from griddata with v4 method
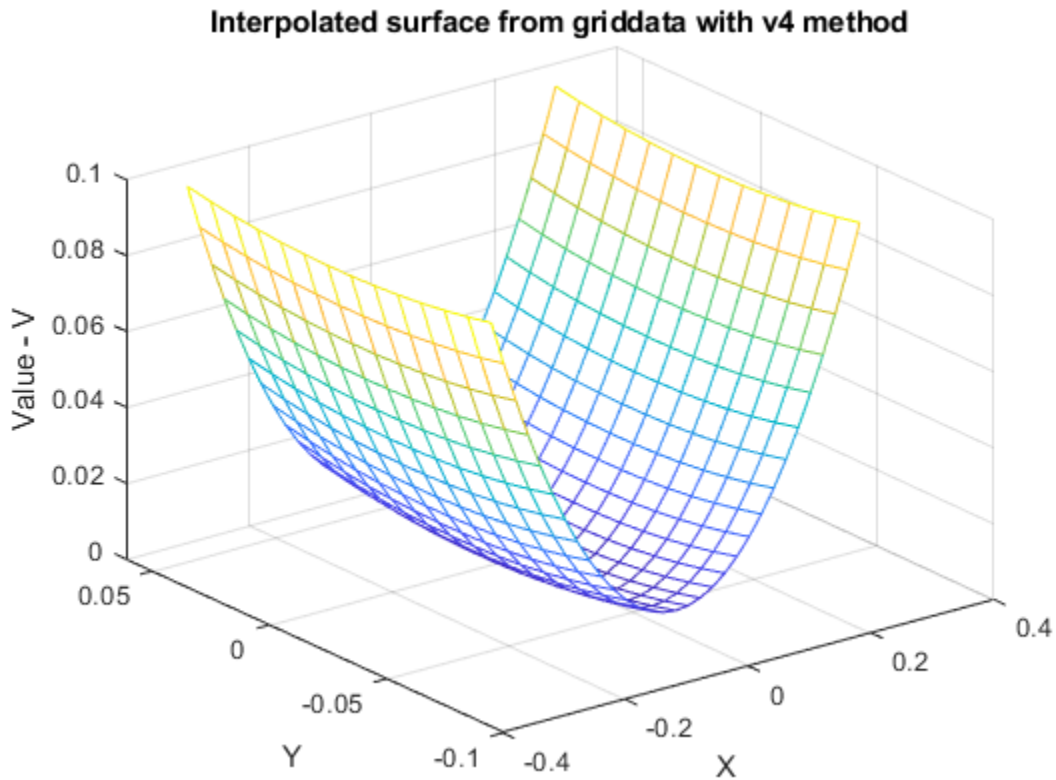
The interpolated surface from `griddata` using the `'v4'` method corresponds to the expected actual surface.

Interpolation Using a Specific Delaunay Triangulation

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.

```
P = -2.5 + 5*gallery('uniformdata',[50 2],0);
DT = delaunayTriangulation(P)

DT =
  delaunayTriangulation with properties:

              Points: [50x2 double]
    ConnectivityList: [87x3 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*gallery('uniformdata',[10 2],1);
```

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
```

```
          5.3163
          0.3453
          4.3026
          1.2579
          1.9435
          5.9194
          3.9030
          0.4172
         11.7282
          0.8641
```

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.

```
P = -2.5 + 5*gallery('uniformdata',[50 2],0);
DT = delaunayTriangulation(P)
```

```
DT =
  delaunayTriangulation with properties:

             Points: [50x2 double]
    ConnectivityList: [87x3 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*gallery('uniformdata',[10 2],1);
```

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*

```
    5.9456
    1.1222
    4.7963
    0.9373
    2.3533
    3.4219
    2.3104
    0.7728
    8.0479
    1.0886
```

See Also

delaunayTriangulation | nearestNeighbor | pointLocation

More About

- "Interpolating Scattered Data" on page 8-55

Extrapolating Scattered Data

In this section...

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 8-79 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
```
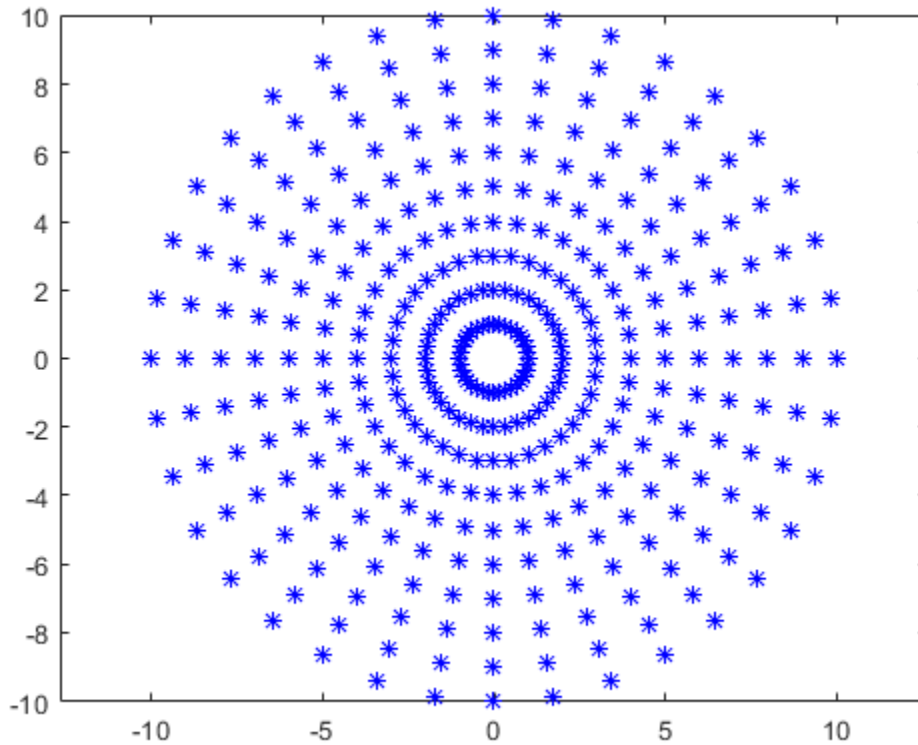

Create a second, more coarsely distributed set of points. Use the `gallery` function to create random samplings in the range, [-10, 10].

```
x2 = -10 + 20*gallery('uniformdata',[25 1],0);
y2 = -10 + 20*gallery('uniformdata',[25 1],1);
```

8-89

```
figure
plot(x2,y2,'*')
```
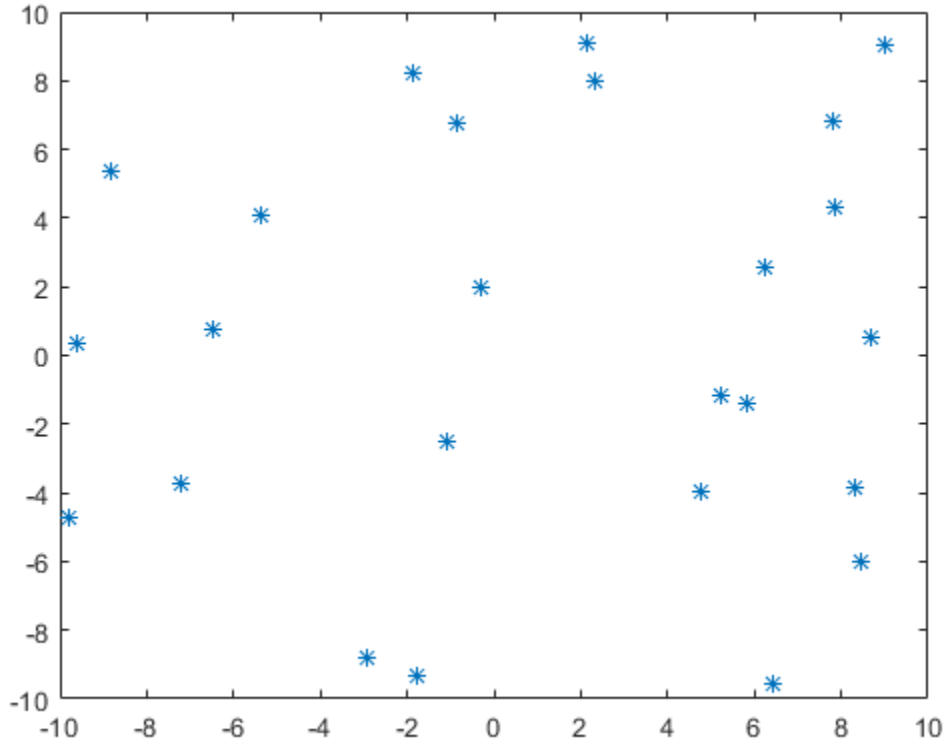

Sample a parabolic function, v(x,y), at both sets of points.

```
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)
```
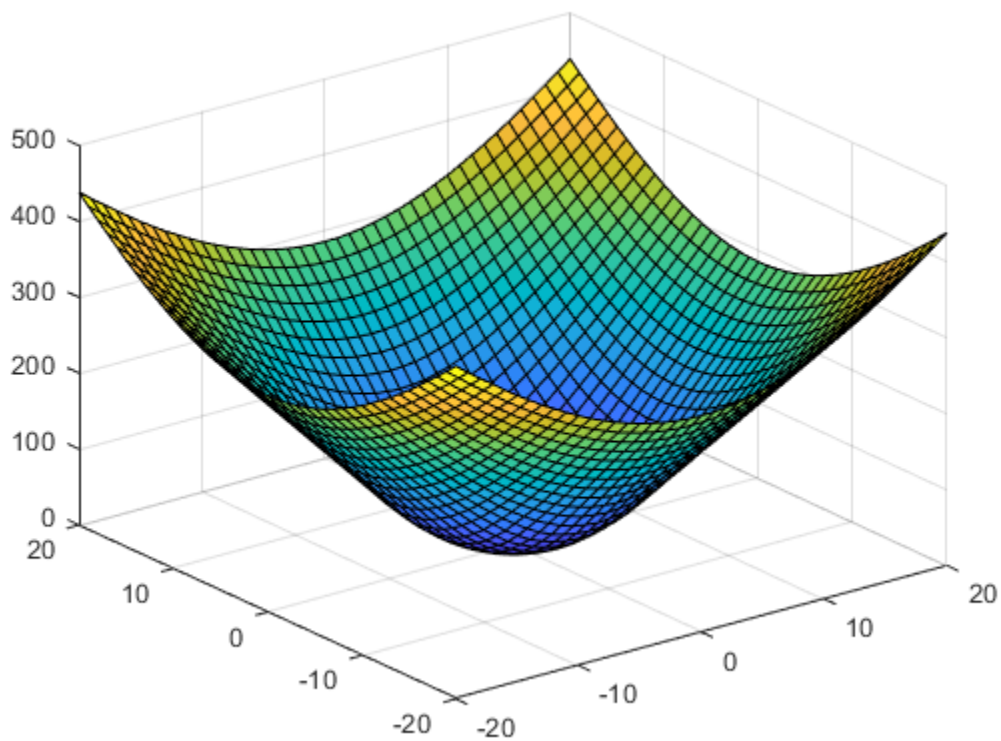

Evaluate F2 and plot the results.

```
figure
vq2 = F2(xq,yq);
surf(xq,yq,vq2)
```
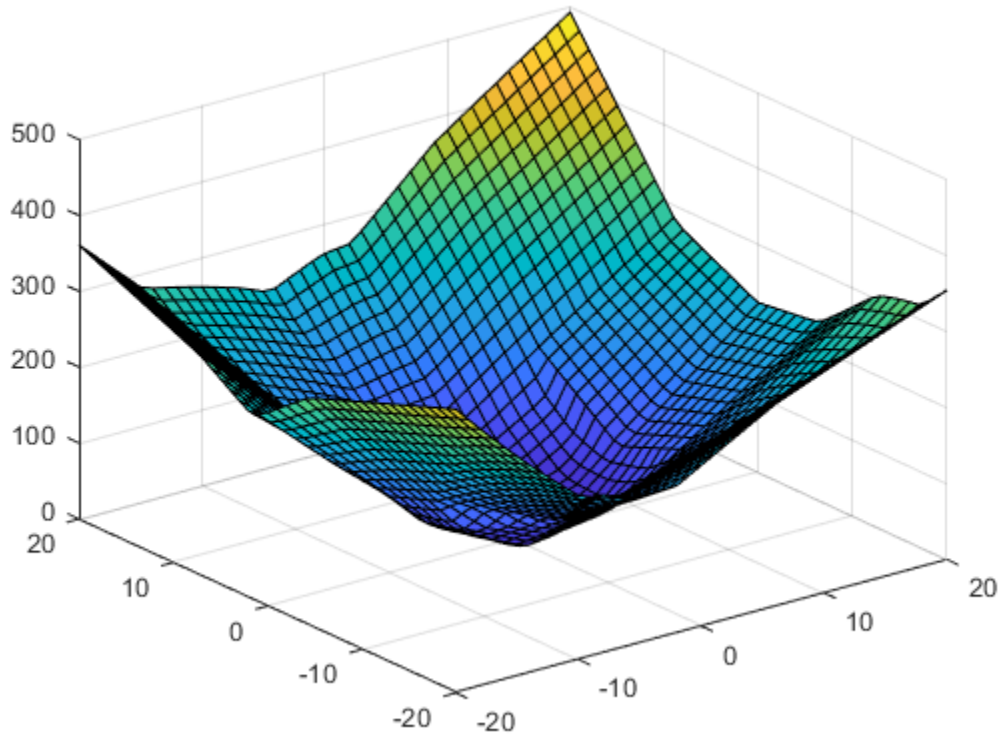
The quality of the extrapolation is not as good for F2 because of the coarse sampling of points in v2.

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)
```
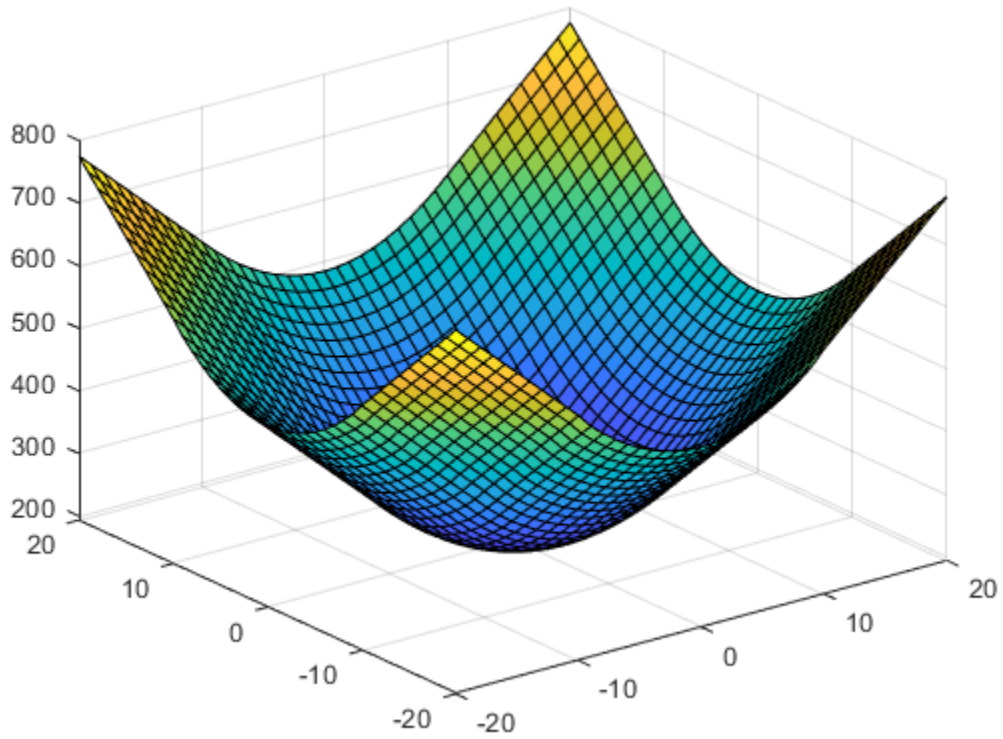
The extrapolation returned good results because the function is well sampled.

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
```
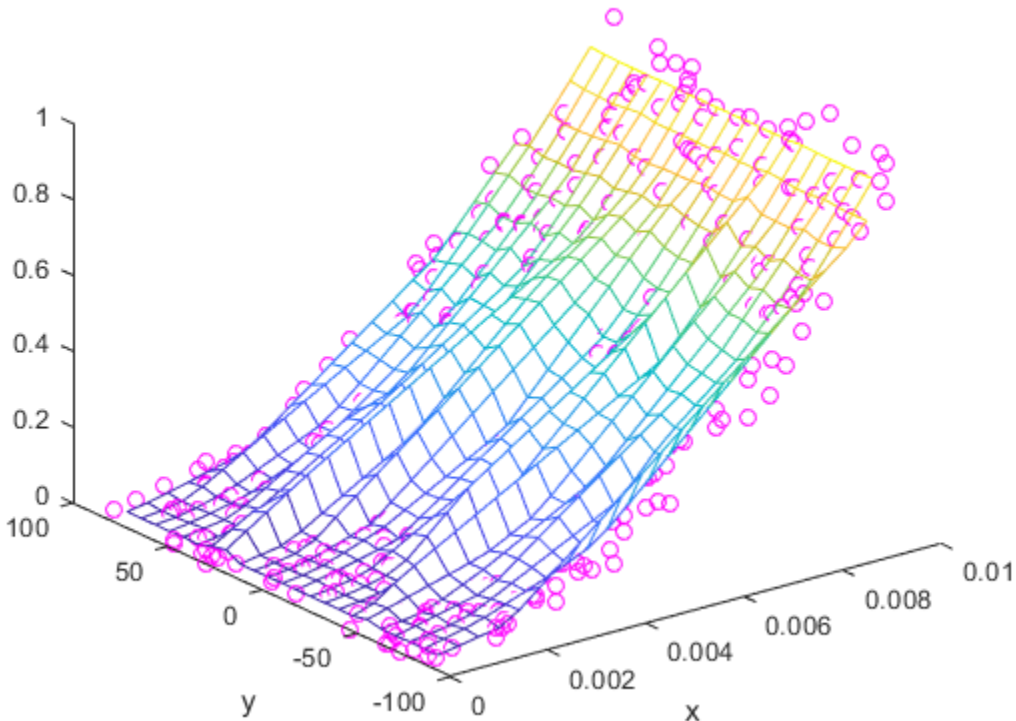

The result produced by `griddata` is not very smooth and seems to be noisy. The differing scales in the independent variables contribute to this, since a small change in the size of one variable can lead to a much larger change in the size of the other variable.

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 standard deviations and regenerate the interpolation using `griddata`.

```
% Normalize Sample Points
x = (x-mean(x))/std(x);
y = (y-mean(y))/std(y);
```

```matlab
% 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)
```
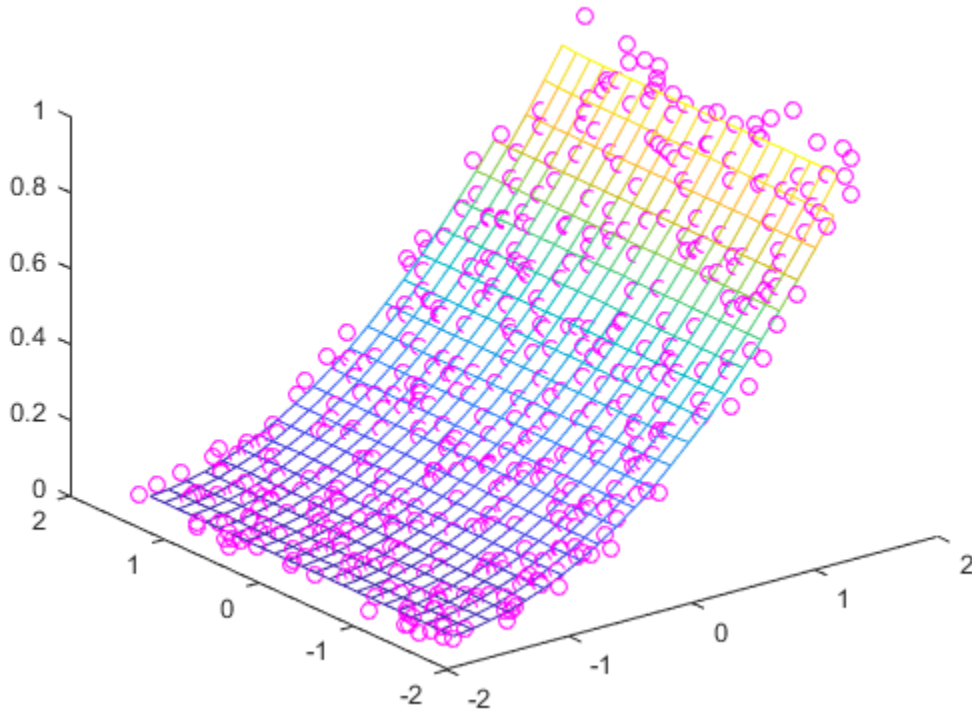

In this case, normalizing the sample points permits `griddata` to compute a smoother solution.

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

PR95-01a · ST Scl OPO · January 1995 · P. Harrington (U.MD), NASA

HST · WFPC2

12/13/94 zgl
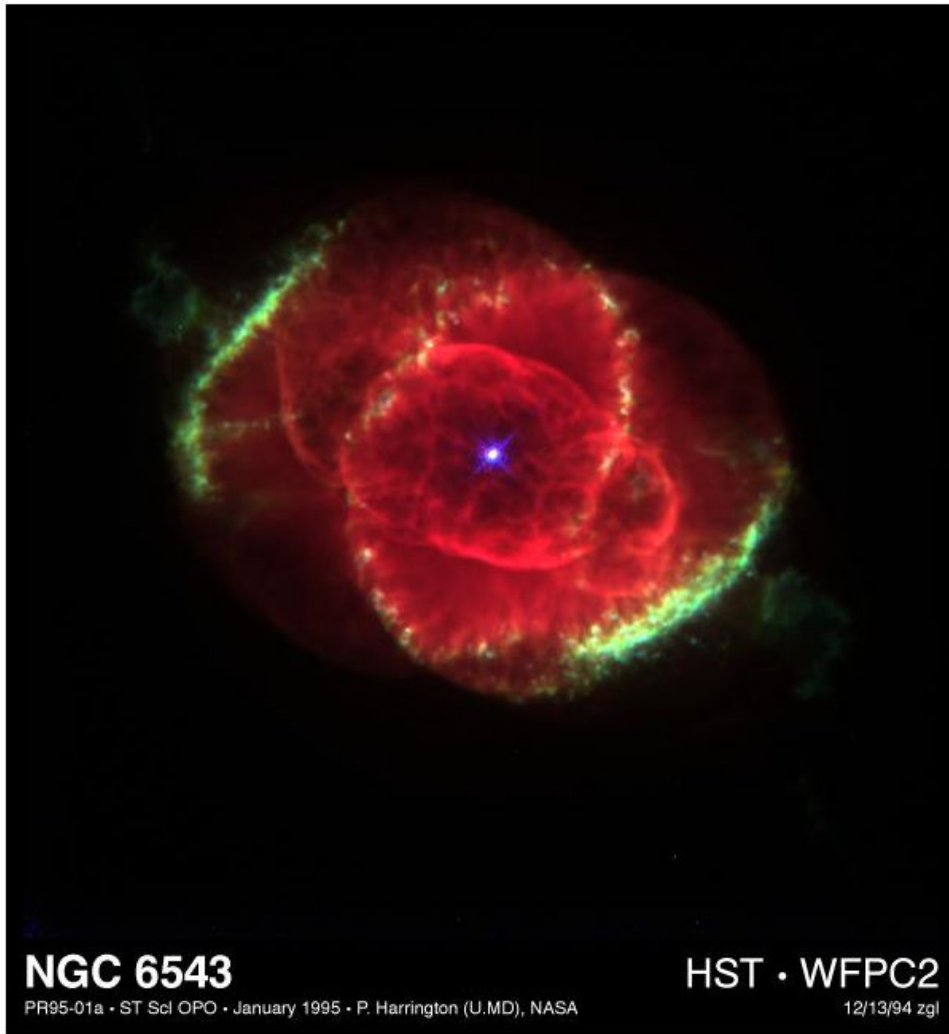
Create Interpolant

Create a gridded interpolant object for the image. For images it is suitable to use the default grid, since the pixels have positive integer locations. Since `griddedInterpolant` only works for double-precision and single-precision matrices, convert the uint8 matrix to double precision.

```
F = griddedInterpolant(double(A));
```

Resample Image Pixels

When you are resampling an image using a large number of grid points, the best way to query the interpolant is using *grid vectors*. The grid vectors are grouped together as column vectors in a cell array `{xg1,xg2,...,xgN}`. Grid vectors are a compact way to represent the grid of query points. With grid vectors, `griddedInterpolant` does not need to form the full grid to carry out the calculations.

Find the size of the original matrix dimensions, and use those dimension sizes 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.

```
[sx,sy,sz] = size(A);
xq = (0:5/6:sx)';
yq = (0:5/6:sy)';
zq = (1:sz)';
vq = uint8(F({xq,yq,zq}));
figure
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
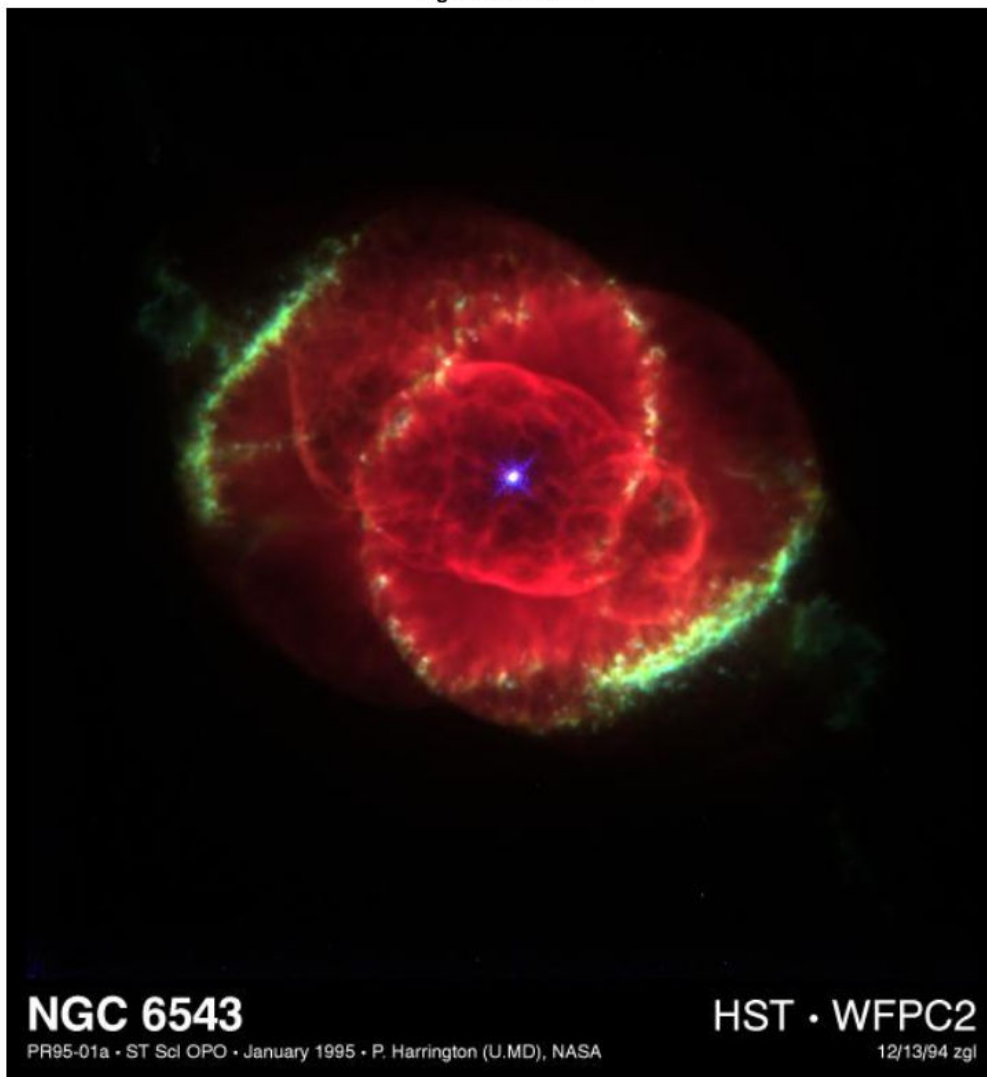
Similarly, reduce the size of the image by querying the interpolant with 55% fewer points than the original image. While you can simply index into the original image matrix to produce lower resolution images, interpolation enables you to resample the image at noninteger pixel locations.

```
xq = (0:1.55:sx)';
yq = (0:1.55:sy)';
zq = (1:sz)';
vq = uint8(F({xq,yq,zq}));
figure
imshow(vq)
title('Lower Resolution')
```

Lower Resolution
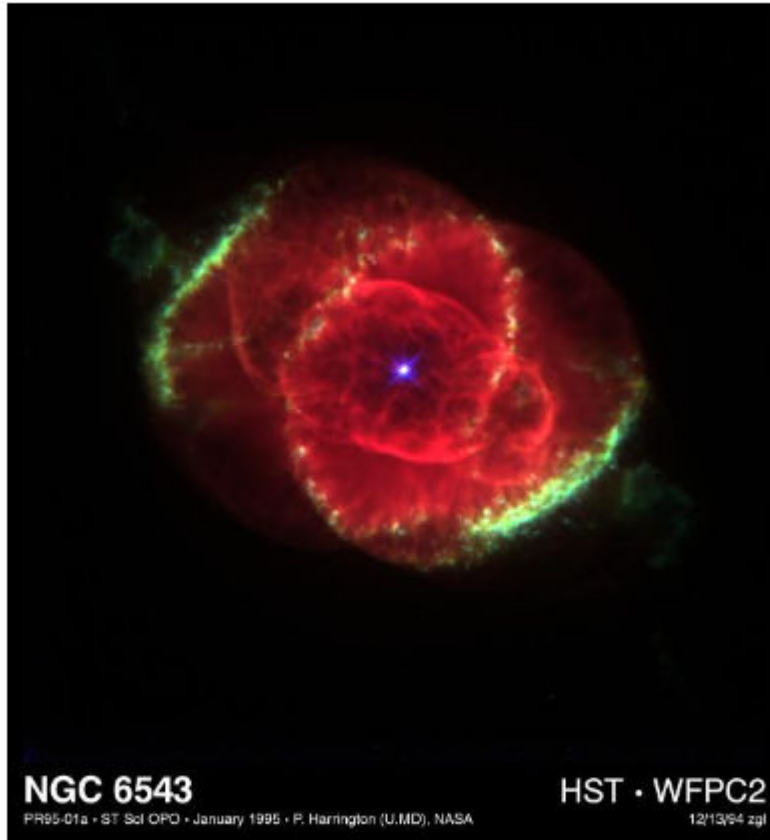
Smooth Out Zooming Artifacts

As you zoom in on an image, the pixels in the region of interest become larger and 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')
```


Original Image, 10x Zoom
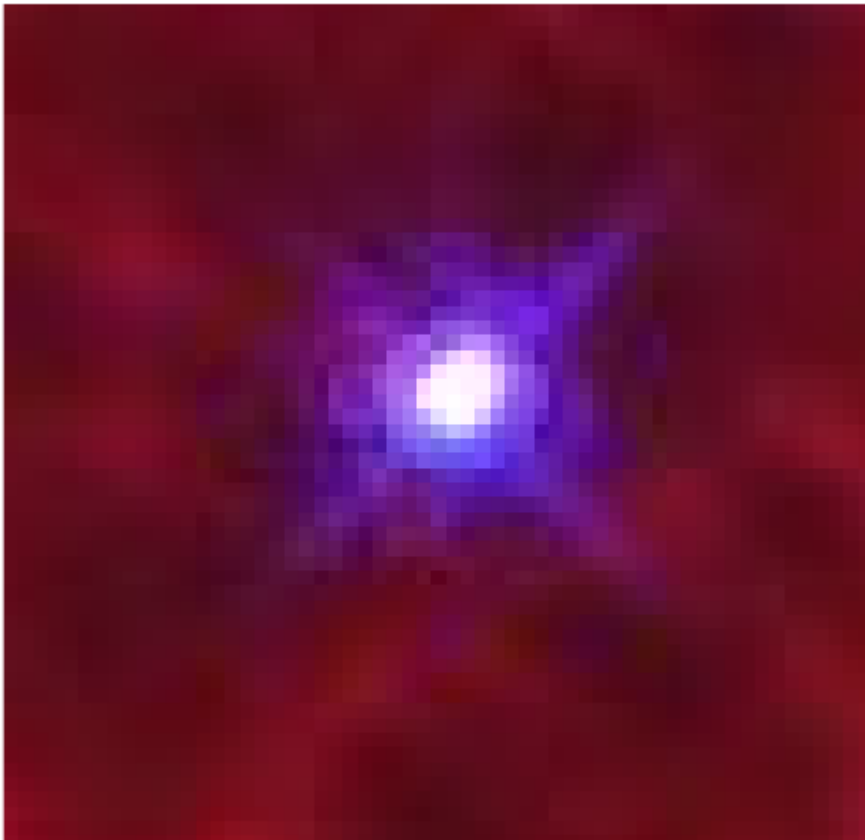
Query the interpolant F to reproduce this zoomed image (approximately) with 10x higher resolution. Compare the results from several different interpolation methods.

```
xq = (1:0.1:sx)';
yq = (1:0.1:sy)';
zq = (1:sz)';
figure
F.Method = 'linear';
vq = uint8(F({xq,yq,zq}));
imshow(vq(1:5700,150:5900,:),'InitialMagnification','fit')
zoom(10)
title('Linear method')
```


Linear method

```
figure
F.Method = 'cubic';
vq = uint8(F({xq,yq,zq}));
imshow(vq(1:5700,150:5900,:),'InitialMagnification','fit')
zoom(10)
title('Cubic method')
```


Cubic method

```
figure
F.Method = 'spline';
vq = uint8(F({xq,yq,zq}));
imshow(vq(1:5700,150:5900,:),'InitialMagnification','fit')
zoom(10)
title('Spline method')
```

Spline method
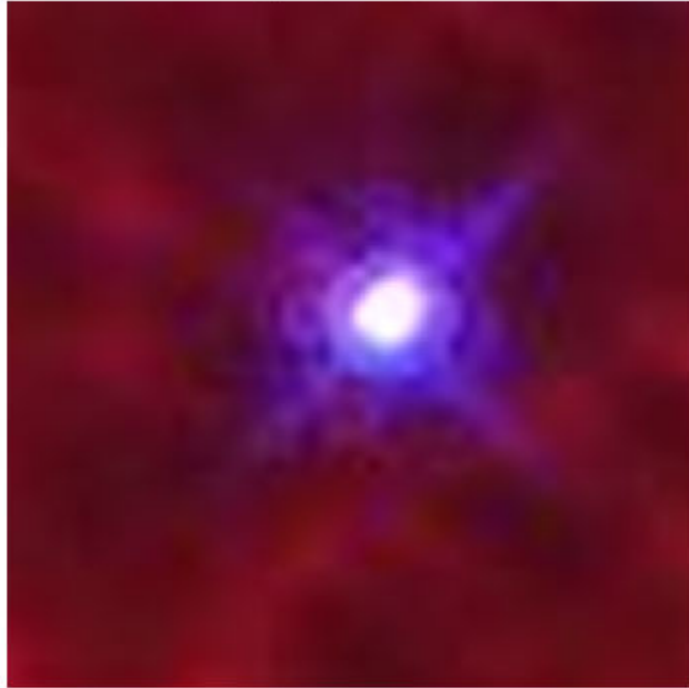
See Also

griddedInterpolant | imshow

More About

•   "Interpolating Gridded Data" on page 8-4

9

Optimization

Optimizing Nonlinear Functions

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
```
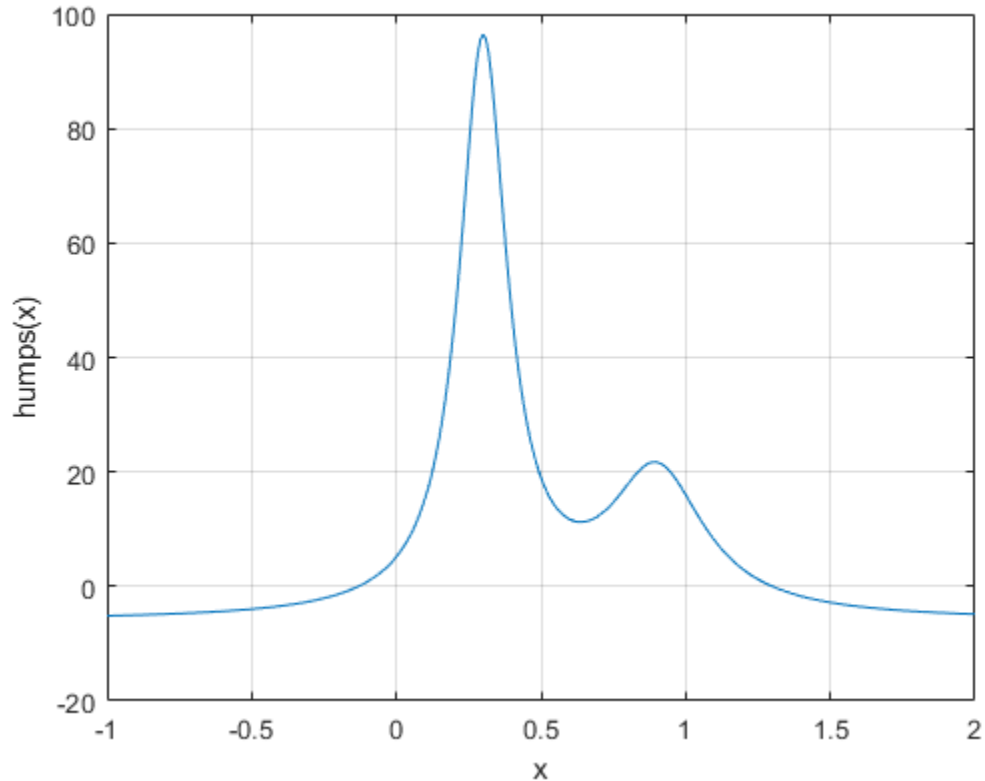
To find the minimum of the humps function in the range (0.3,1), use

```
x = fminbnd(@humps,0.3,1)
```

```
x = 0.6370
```

You can ask for a tabular display of output by passing a fourth argument created by the optimset command to fminbnd:

```
opts = optimset('Display','iter');
x = fminbnd(@humps,0.3,1,opts)
```

```
 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 what procedure is being used at each iteration, either a golden section search or a parabolic interpolation. For more information, see "Iterative Display" on page 9-15.

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 larger than $f(x(1))$].

3 Generate the reflected point

$$r = 2m - x(n+1),$$

where

$$m = \Sigma x(i)/n, \quad i = 1...n,$$

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

$$s = m + 2(m - x(n+1)),$$

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 $f(r) \ge f(x(n))$, perform a contraction between m and the better of $x(n+1)$ and r:

   a If $f(r) < f(x(n+1))$ (that is, r is better than $x(n+1)$), calculate

$$c = m + (r - m)/2$$

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 $f(r) \ge f(x(n+1))$, calculate

$$cc = m + (x(n+1) - m)/2$$

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

$$v(i) = x(1) + (x(i) - x(1))/2$$

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.
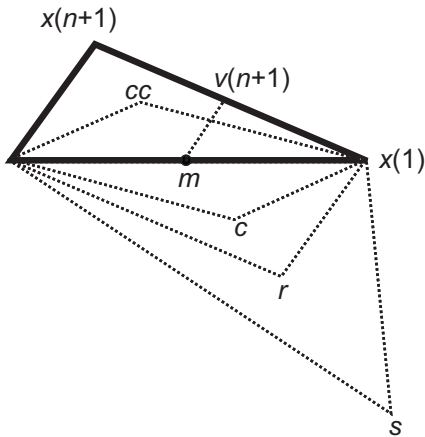
Reference

[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

- "Troubleshooting and Tips" on page 9-28
- "Nonlinear Optimization" (Optimization Toolbox)
- "Curve Fitting via Optimization" on page 9-8

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 9-4.

```
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
```

Data and Best Fitting Exponential Curve
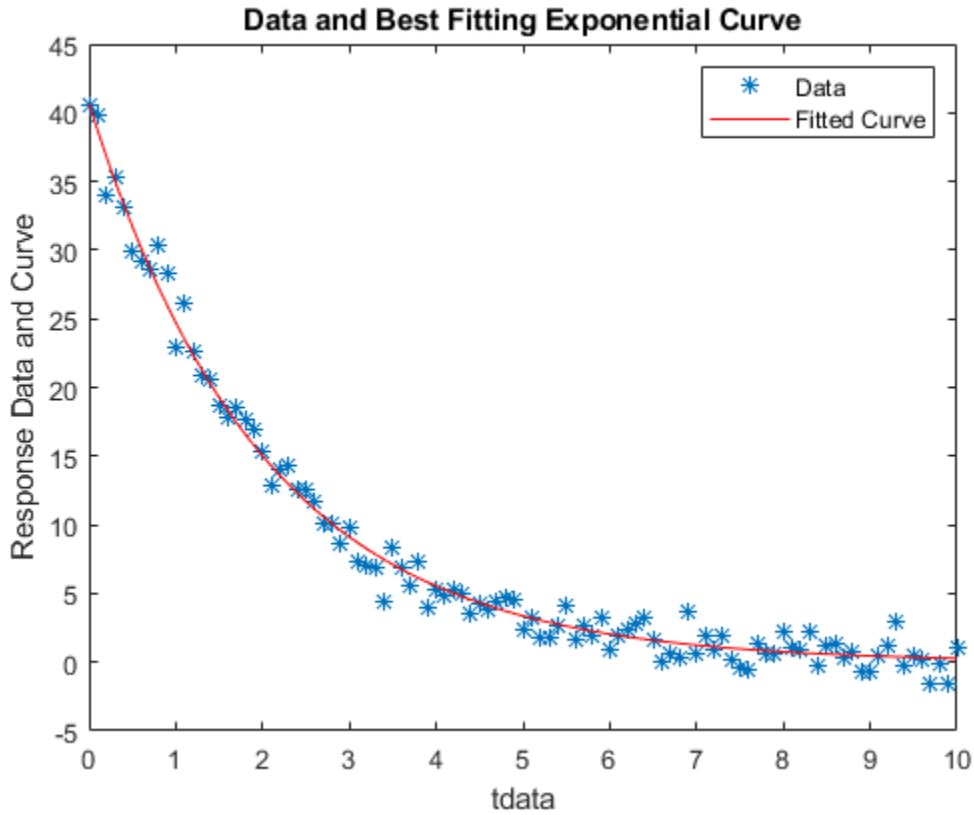
See Also

More About

- "Optimizing Nonlinear Functions" on page 9-2
- "Nonlinear Data-Fitting" (Optimization Toolbox)
- "Nonlinear Regression" (Statistics and Machine Learning Toolbox)

Set Options

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 "Iterative Display" on page 9-15. • `'off'` 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`
`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 "Output Functions" on page 9-17.	`fminbnd`, `fminsearch`, `fzero`
`PlotFcns`	Plot information on the iterations of the solver. The default is `[]` (none). For available predefined functions, see "Plot Functions" on page 9-25.	`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 9-13.	`fminsearch`

Option	Description	Solvers
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`). See "Tolerances and Stopping Criteria" on page 9-13.	`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 sometimes use `TolX` as a *relative* bound, meaning iterations end when $|(x_i – x_{i+1})| <$ `TolX`*(1 + $|x_i|$), or a similar relative measure.
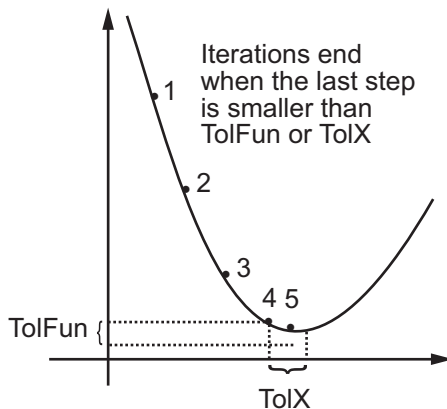
- `TolFun` is a lower bound on the change in the value of the objective function during a step. If $|f(x_i) - f(x_{i+1})|$ < `TolFun`, the iterations end. Solvers sometimes use `TolFun` as a *relative* bound, meaning iterations end when $|f(x_i) - f(x_{i+1})|$ < `TolFun`$(1 + |f(x_i)|)$, or a similar 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

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 9-5.	`fminsearch`
	• `initial` (initial point) • `search` (search for an interval containing a zero) • `bisection` • `interpolation` (linear interpolation or inverse quadratic interpolation)	`fzero`

Heading	Information Displayed	Solvers
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 Options" on page 9-11
- "Output Functions" on page 9-17
- "Plot Functions" on page 9-25

Output Functions

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

$$\min_x f(x) = \min_x e^{x_1} \left(4x_1^2 + 2x_2^2 + x_1 x_2 + 2x_2 \right).$$

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`:
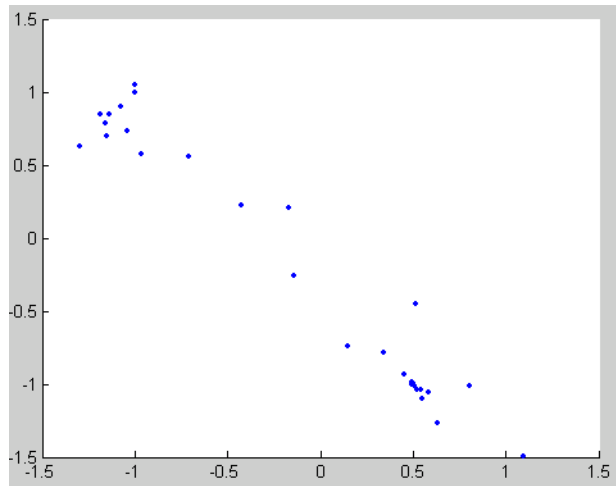
Structure of the Output Function

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 9-23.
- `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 9-22 describes the structure in detail.
- `state` is the current state of the algorithm. "States of the Algorithm" on page 9-22 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 9-17 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

$$\min_x f(x) = \min_x e^{x_1} \left(4x_1^2 + 2x_2^2 + x_1 x_2 + 2x_2 \right).$$

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.
`'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

- "Plot Functions" on page 9-25
- "Set Options" on page 9-11
- "Iterative Display" on page 9-15

Plot Functions

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 "Output Functions" on page 9-17.

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
```

Current Function Value: -0.40524
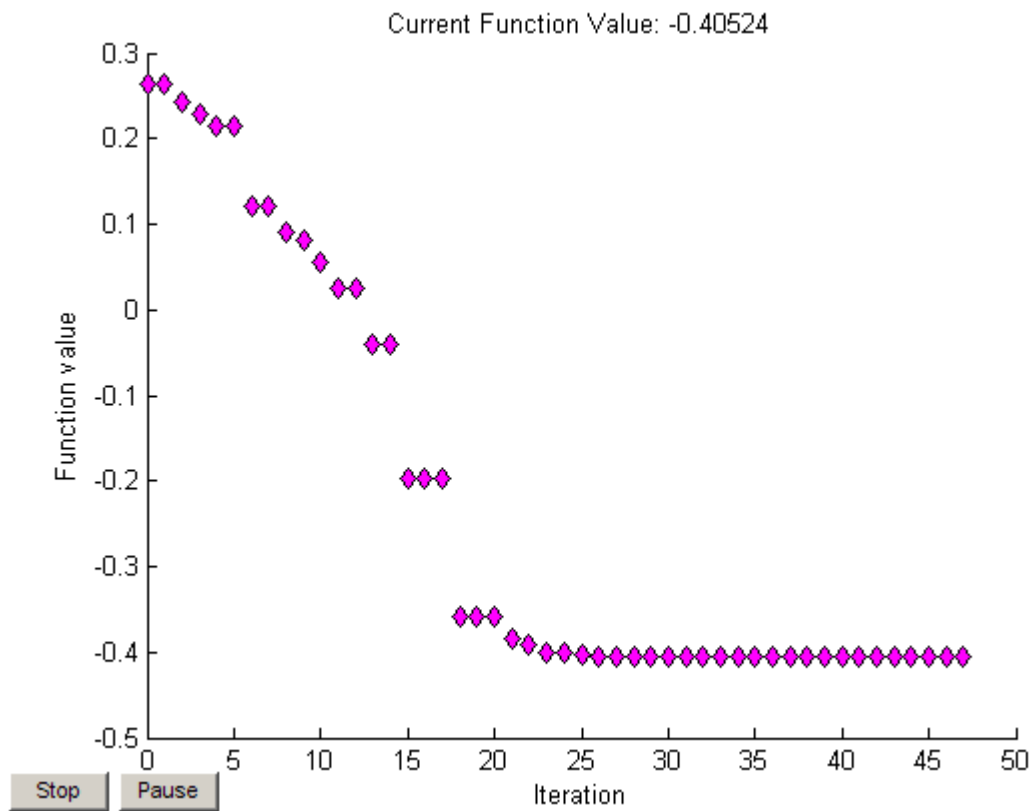
See Also

More About

- "Output Functions" on page 9-17
- "Set Options" on page 9-11
- "Iterative Display" on page 9-15

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.

Problem	Recommendation
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 "Plot Functions" on page 9-25. • Set the `Display` option to display information at the command line. See "Iterative Display" on page 9-15.

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

Function Handles

Parameterizing Functions

In this section...
"Overview" on page 10-2
"Parameterizing Using Nested Functions" on page 10-2
"Parameterizing Using Anonymous Functions" on page 10-3

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, (t_0, t_f), 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 $y'_1, y'_2, ..., y'_n$. 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

$$y_1 = y$$
$$y_2 = y'$$
$$y_3 = y''$$
$$\vdots$$
$$y_n = y^{(n-1)}.$$

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

$$y_1 = y$$
$$y_2 = y'$$
$$y_3 = y''$$

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

```matlab
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

$$yv = \begin{bmatrix} \text{Real}(y) & \text{Imag}(y) \end{bmatrix}$$
$$fv = \begin{bmatrix} \text{Real}(f(t,y)) & \text{Imag}(f(t,y)) \end{bmatrix}.$$

For example, if the ODE is $y' = yt + 2i$, then you can represent the equation using a function file.

```matlab
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

```matlab
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.

Solver	Problem Type	Accuracy	When to Use
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).

Solver	Problem Type	Accuracy	When to Use
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 Differential Algebraic Equation"
ballode	ode23	• `'Events'` • `'OutputFcn'` • `'OutputSel'` • `'Refine'` • `'InitialStep'` • `'MaxStep'`	Simple event location — bouncing ball	"ODE Event Location" on page 11-17
batonode	ode45	• `'Mass'`	ODE with time- and state-dependent mass matrix — motion of a baton	—
brussode	ode15s	• `'JPattern'` • `'Vectorized'`	Stiff large problem — diffusion in a chemical reaction (the Brusselator)	"Solve Stiff ODEs" on page 11-31
burgersode	ode15s	• `'Mass'` • `'MStateDependence'` • `'JPattern'` • `'MvPattern'` • `'RelTol'` • `'AbsTol'`	ODE with strongly state-dependent mass matrix — Burgers' equation solved using a moving mesh technique	—
fem1ode	ode15s	• `'Mass'` • `'MStateDependence'` • `'Jacobian'`	Stiff problem with a time-dependent mass matrix — finite element method	—

Example File	Solver Used	Options Specified	Description	Documentation Link
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	—
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 11-46
orbitode	ode45	• 'RelTol' • 'AbsTol' • 'Events' • 'OutputFcn'	Advanced event location — restricted three body problem	"ODE Event Location" on page 11-17
rigidode	ode45	—	Nonstiff problem — Euler equations of a rigid body without external forces	"Solve Nonstiff ODEs" on page 11-25

Example File	Solver Used	Options Specified	Description	Documentation Link
vdpode	ode15s	• `'Jacobian'`	Parameterizable van der Pol equation (stiff for large μ)	"Solve Stiff ODEs" on page 11-31

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

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	✓	✓	✓	✓	✓	✓	✓	✓

Option	ode45	ode23	ode113	ode15s	ode23s	ode23t	ode23t b	ode15i
Output Fcn Output Sel Refine Stats	✓	✓	✓	✓	✓	✓	✓	✓
NonNeg ative	✓	✓	✓	✓*	—	✓*	✓*	—
Events	✓	✓	✓	✓	✓	✓	✓	✓**
MaxSte p Initia lStep	✓	✓	✓	✓	✓	✓	✓	✓
Jacobi an JPatte rn Vector ized	—	—	—	✓	✓	✓	✓	✓
Mass MState Depend ence MvPatt ern MassSi ngular	✓ ✓ — —	✓ ✓ — —	✓ ✓ — —	✓ ✓ ✓ ✓	✓ — — —	✓ ✓ ✓ ✓	✓ ✓ ✓ —	— — — —
Initia lSlope	—	—	—	✓	—	✓	—	—
MaxOrd er BDF	—	—	—	✓ ✓	—	—	—	✓ —

* Use the NonNegative parameter with ode15s, ode23t, and ode23tb only for those problems in which there is no mass matrix.

11-15

** 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 11-9.

See Also

odeget | odeset

More About

- "Choose an ODE Solver" on page 11-2
- "ODE Event Location" on page 11-17
- "Nonnegative ODE Solution" on page 11-46

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 `ith` element corresponds to the `ith` event:

- `value(i)` is a mathematical expression describing the `ith` event. An event occurs when `value(i)` is equal to zero.
- `isterminal(i) = 1` if the integration is to terminate when the `ith` 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
```
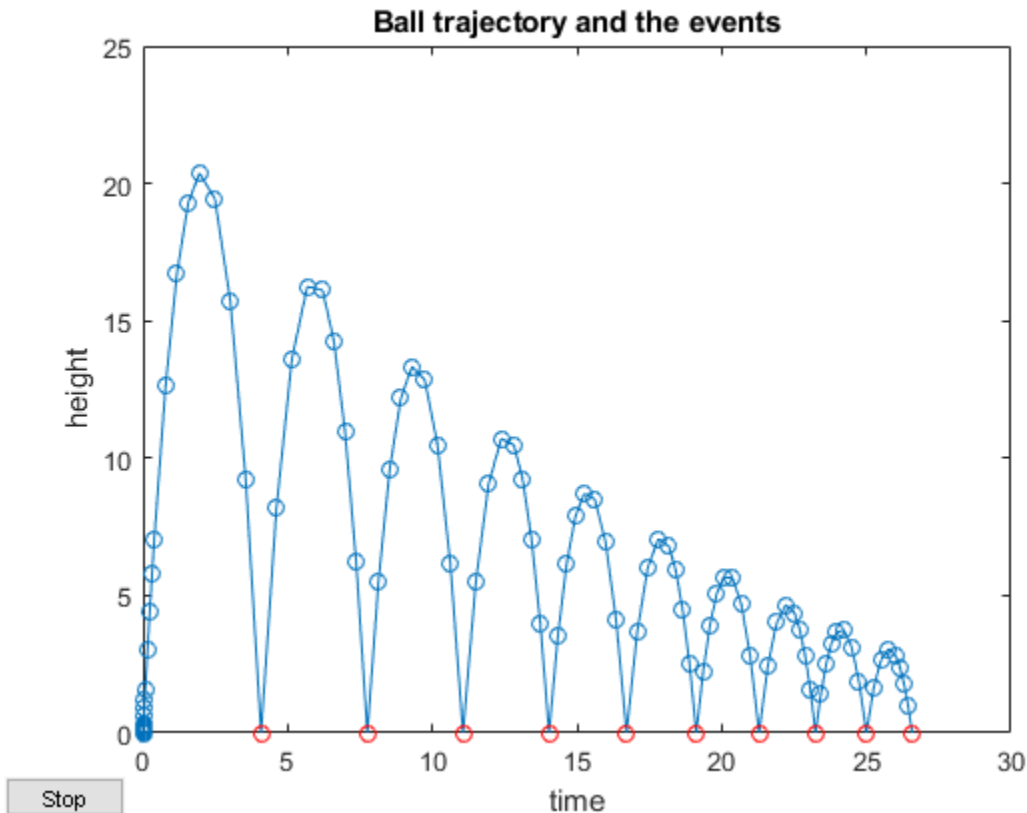

Advanced Event Location: Restricted Three Body Problem

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

```matlab
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.
```
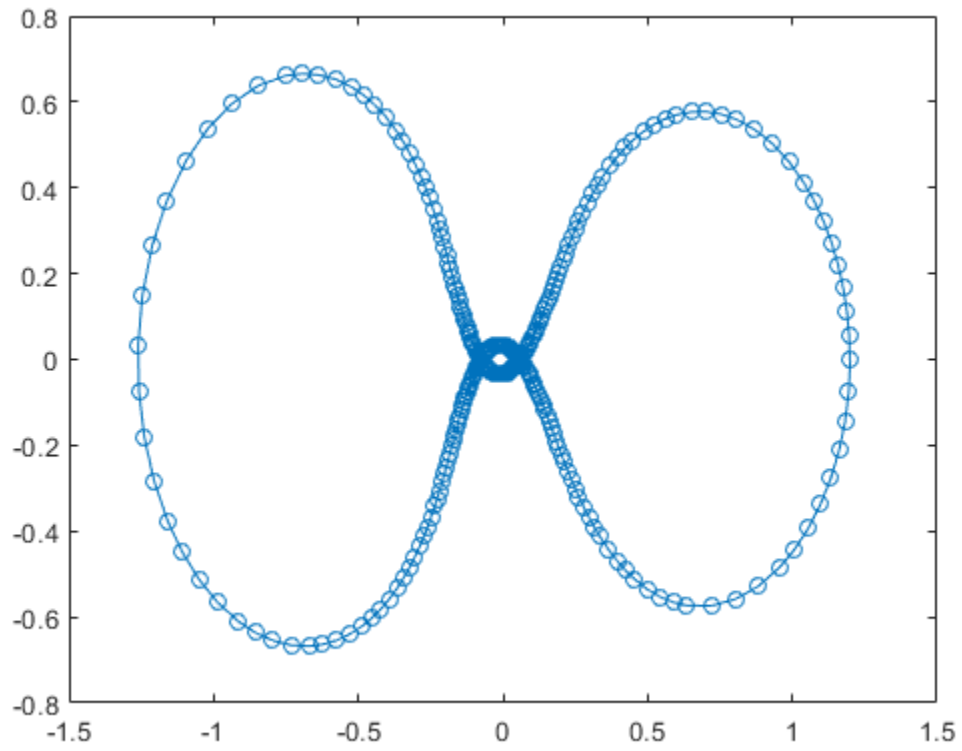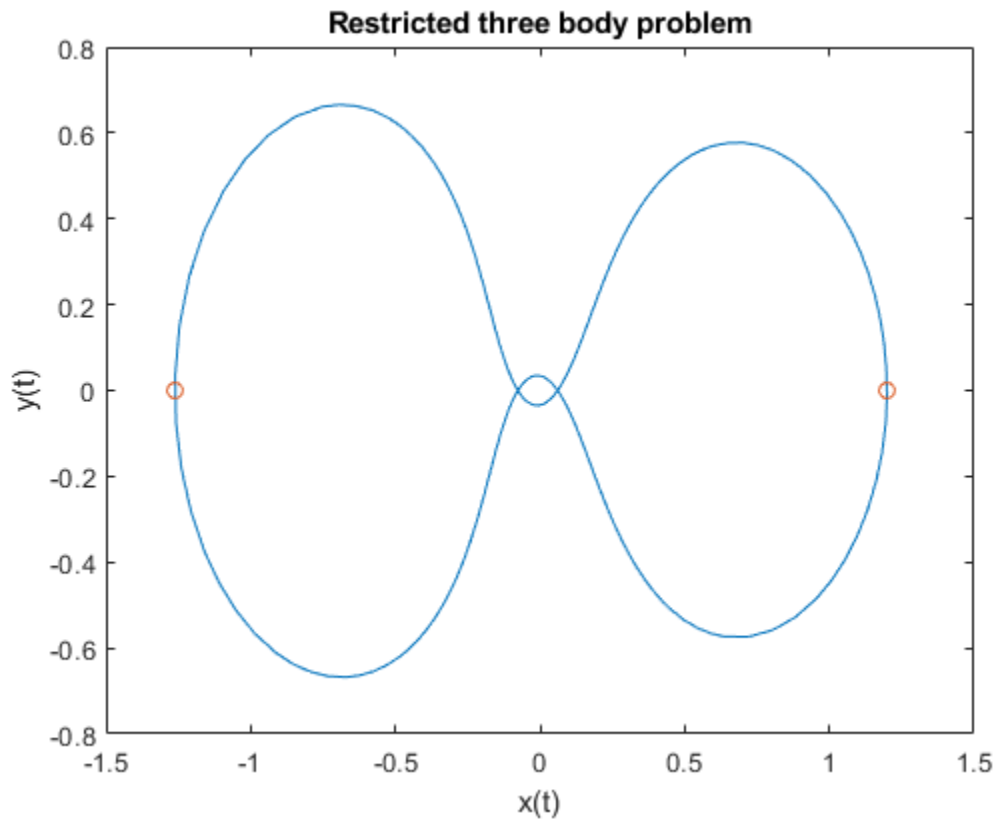
Restricted three body problem

See Also

odeget | odeset

More About

- "Choose an ODE Solver" on page 11-2
- "Summary of ODE Options" on page 11-14
- "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 11-31 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')
```
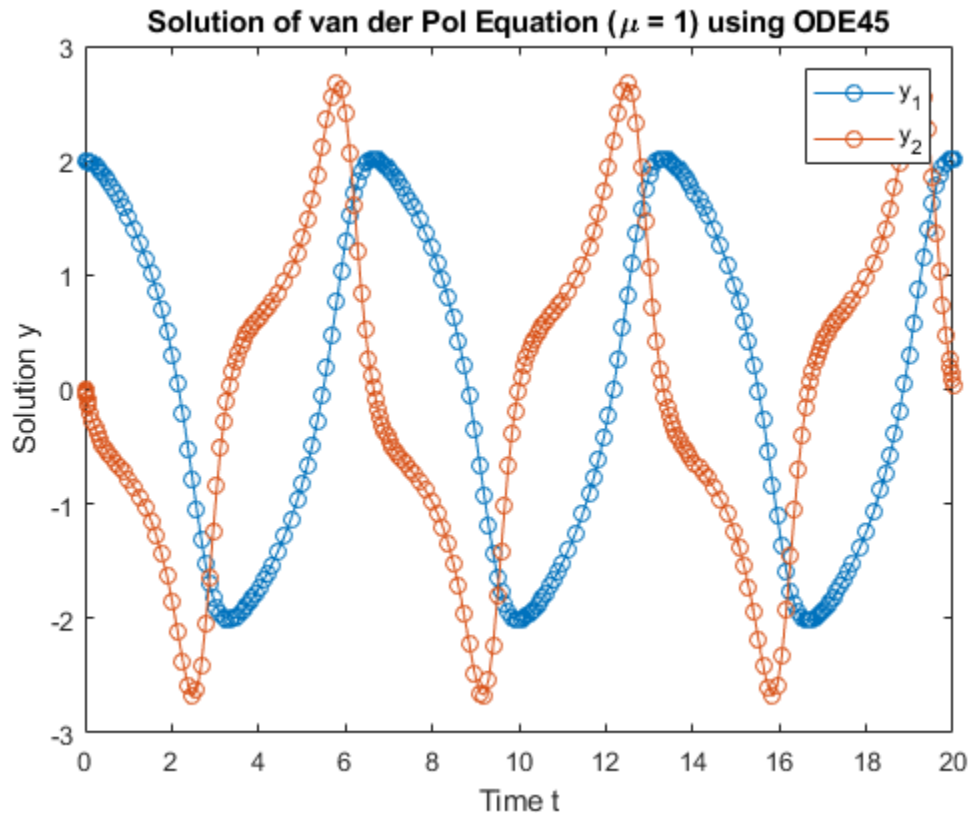
The `vdpode` function solves the same problem, but it accepts a user-specified value for μ. The van der Pol equations become stiff as μ increases. For example, with the value $\mu = 1000$ you need to use a stiff solver such as `ode15s` to solve the system.

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

$$y_1' = y_2 y_3$$
$$y_2' = -y_1 y_3$$
$$y_3' = -0.51 y_1 y_2.$$

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')
```
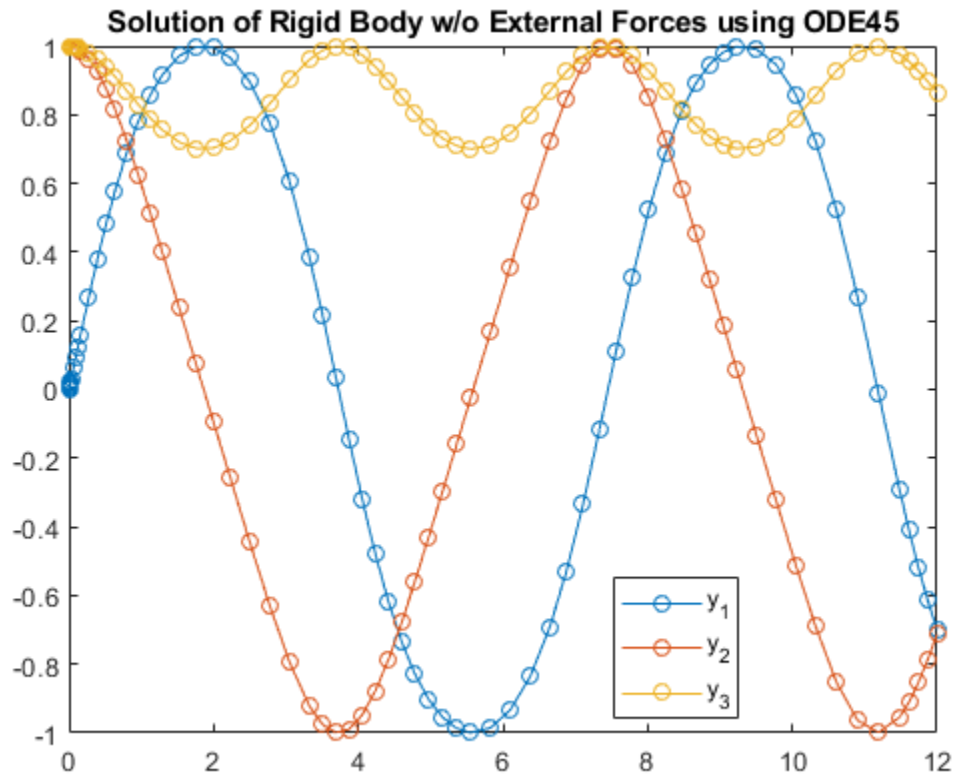

See Also

ode113 | ode23 | ode45

More About

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');
```
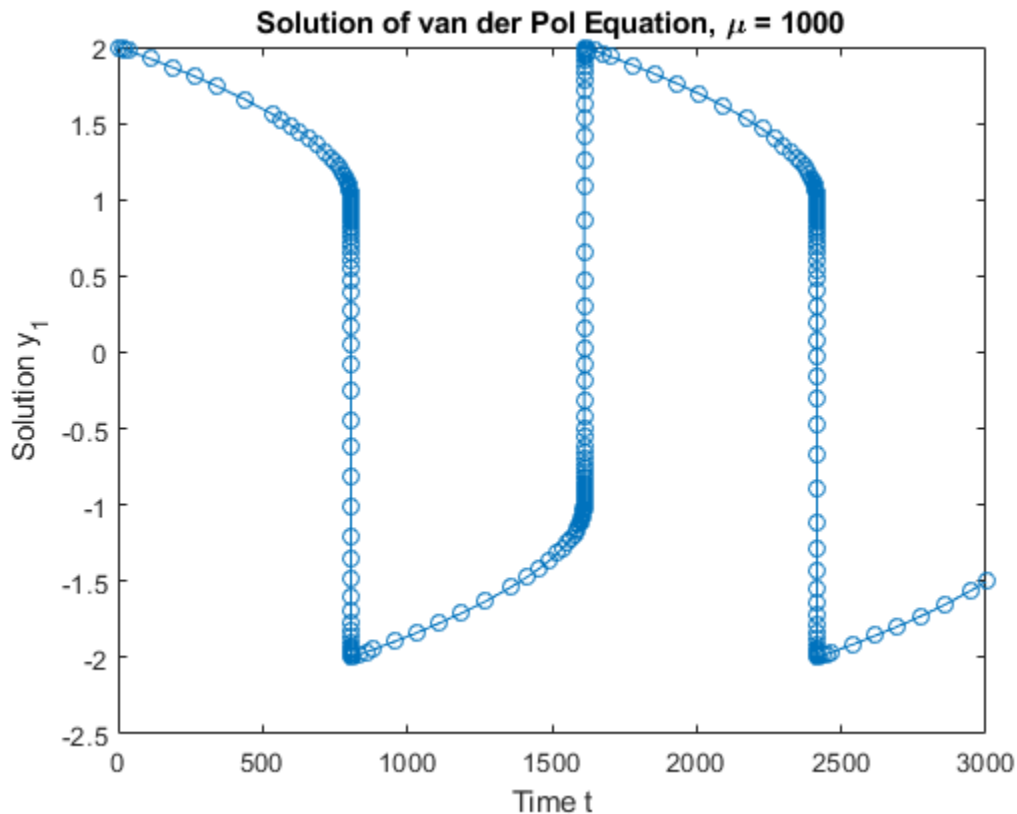
The vdpode function also solves the same problem, but it accepts a user-specified value for μ. The equations become increasingly stiff as μ increases.

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 $i = 1, ..., N$. 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 $u_1, v_1, u_2, v_2, \cdots$. 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
```

The Brusselator for N = 20
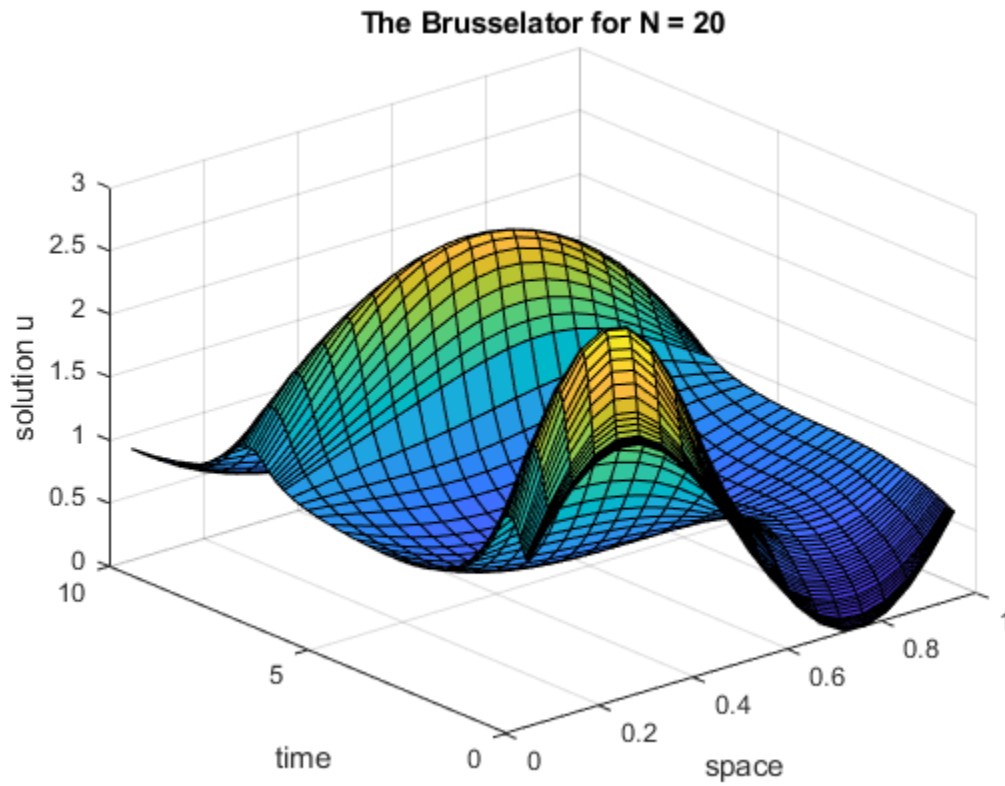
Solve the system for $N = 50$ by specifying an input to brussode.

```
brussode(50)
```

The Brusselator for N = 50
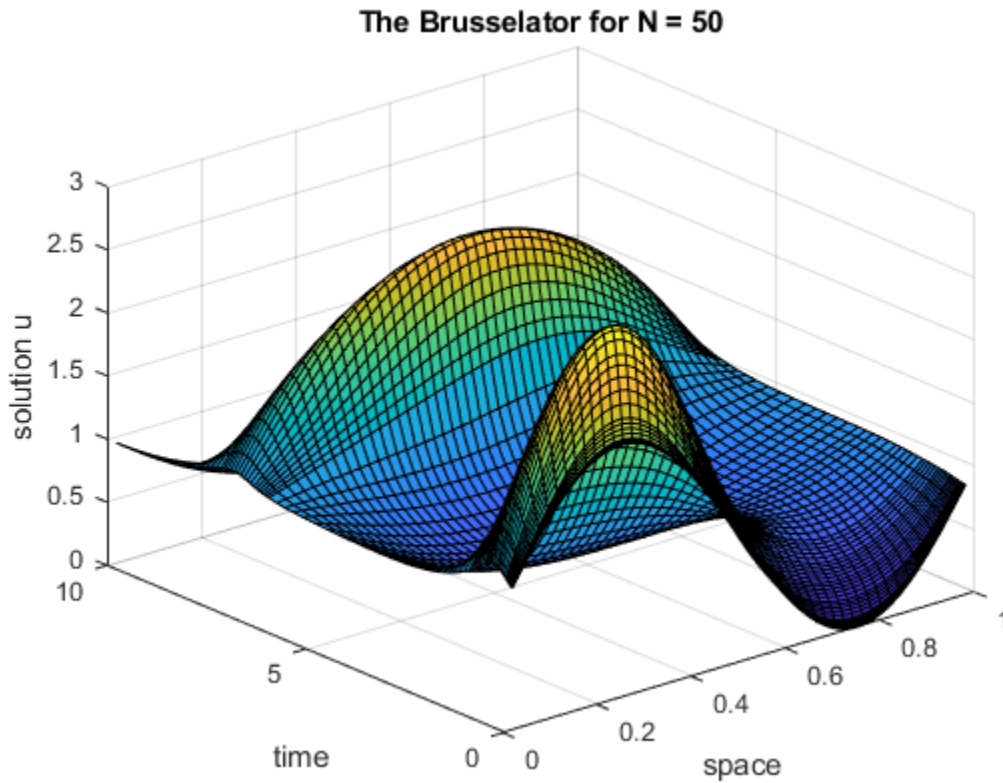
See Also
ode15s | ode23s | ode23t | ode23tb

More About
- "Choose an ODE Solver" on page 11-2
- "Summary of ODE Options" on page 11-14
- "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

  $$y' = f(t, y, z)$$
  $$0 = g(t, y, z).$$

  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.

$$J = \partial f / \partial y' = \begin{pmatrix} \dfrac{\partial f_1}{\partial y'_1} & \cdots & \dfrac{\partial f_1}{\partial y'_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial y'_1} & \cdots & \dfrac{\partial f_m}{\partial y'_n} \end{pmatrix}$$

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

$$y'_1 = y_2$$
$$0 = k(t) - y_1.$$

These equations require two derivatives to be rewritten in the explicit ODE form

$$y'_1 = k'(t)$$
$$y'_2 = k''(t).$$

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 "Step 3. Check and Reduce Differential Index" (Symbolic Math Toolbox) for more information.

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);
```

11-43

```
ylabel('1e4 * y(:,2)');
title('Robertson DAE problem with a Conservation Law, solved by ODE15S');
```
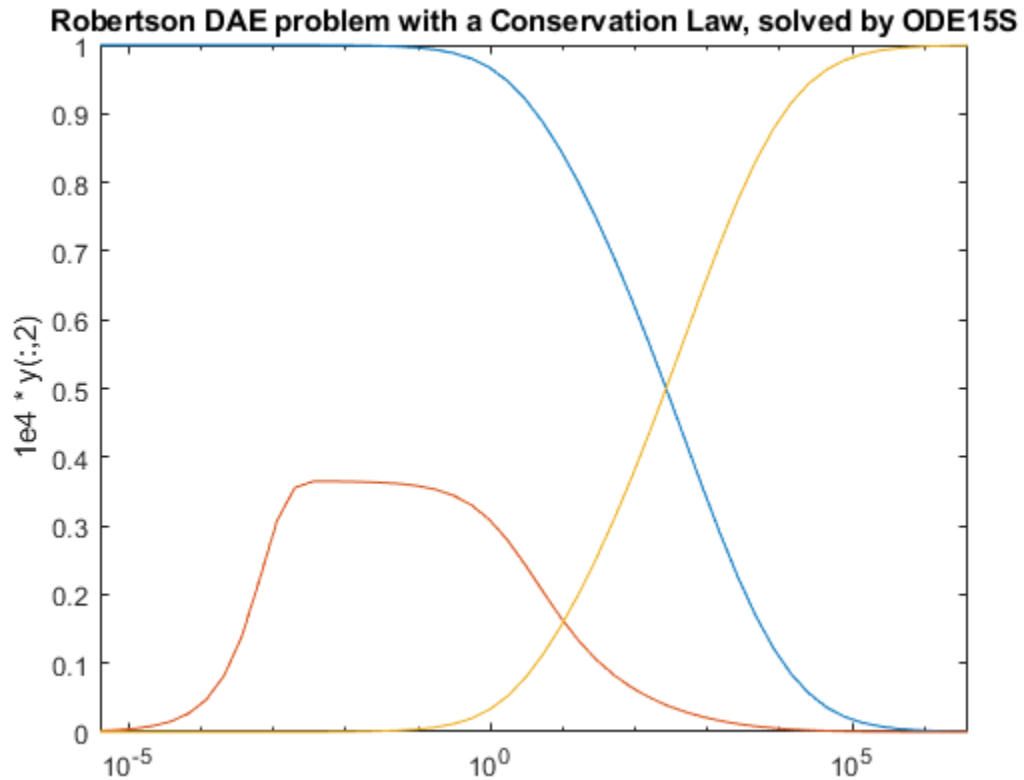

See Also
ode15i | ode15s | ode23t | odeset

More About
- "Choose an ODE Solver" on page 11-2
- "Summary of ODE Options" on page 11-14

- "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.

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.

```matlab
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')
```
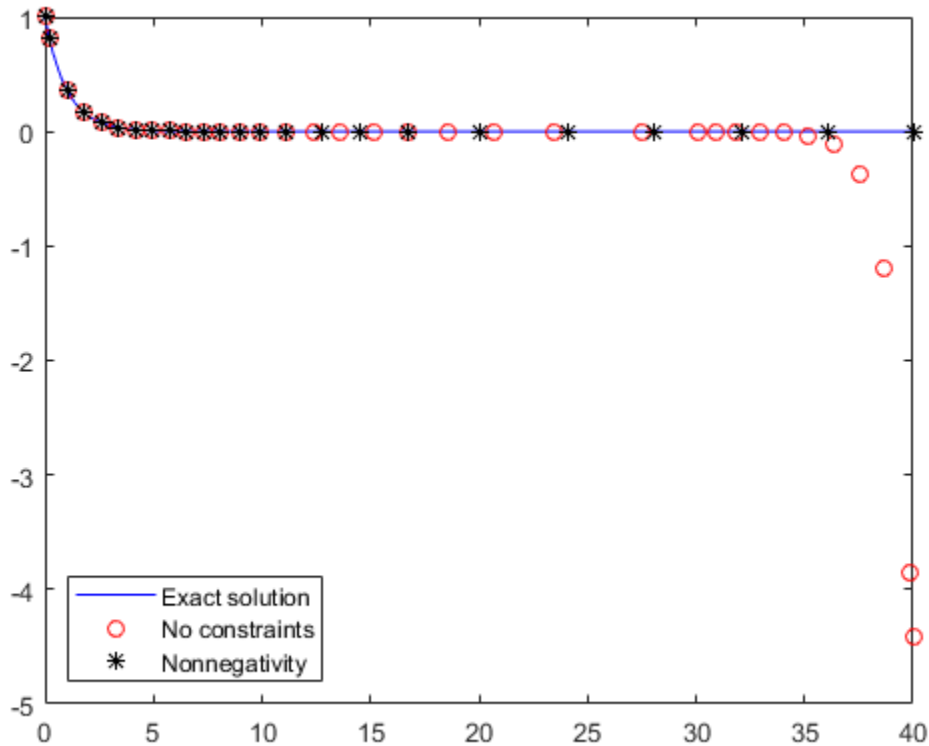

Example: The Knee Problem

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')
```

The "knee problem"
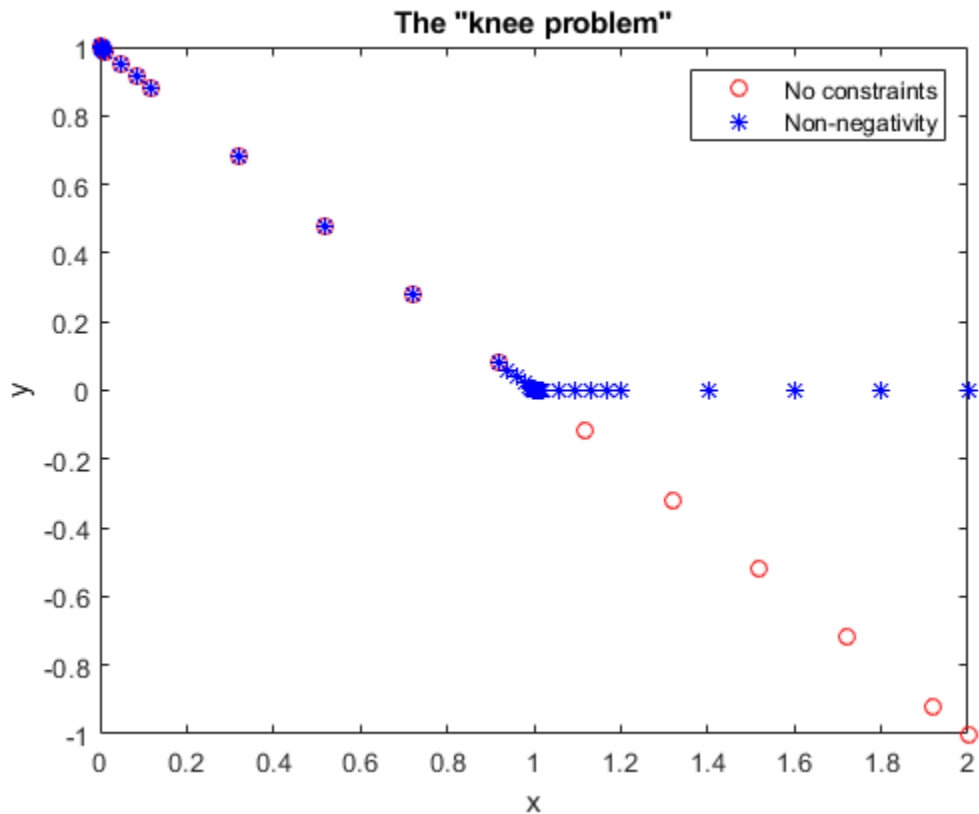
References

[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 11-14

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.

Question or Problem	Answer
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.

Question or Problem	Answer
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.

Question or Problem	Answer
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 globals, 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 11-14
- "ODE Event Location" on page 11-17
- "Nonnegative ODE Solution" on page 11-46

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

$$\frac{d^2y}{dt^2} - \mu(1-x^2)\frac{dy}{dt} + y = 0.$$

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 $\frac{dy}{dt}\big|_{t=0} = 0$.

```
tspan = [0 20];
y0 = [2; 0];
```

11-59

```matlab
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')
```
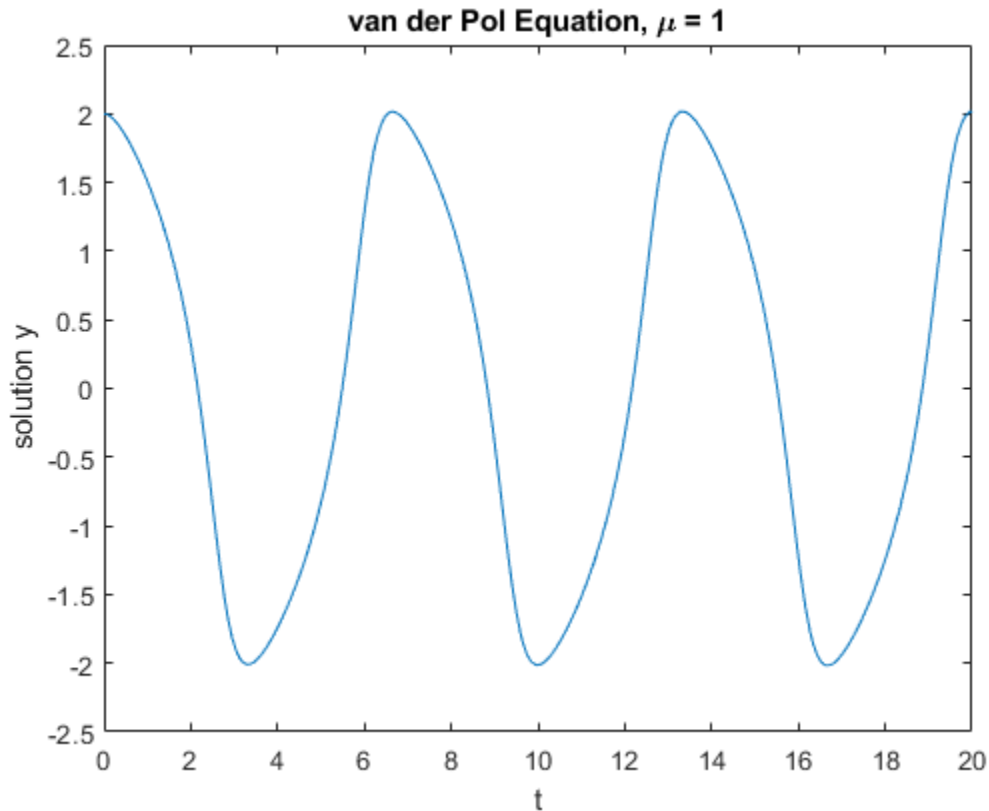

For larger magnitudes of μ, the problem becomes *stiff*. This label is for problems that resist attempts to be evaluated with ordinary techniques. Instead, special numerical

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')
```

van der Pol Equation, $\mu = 1000$
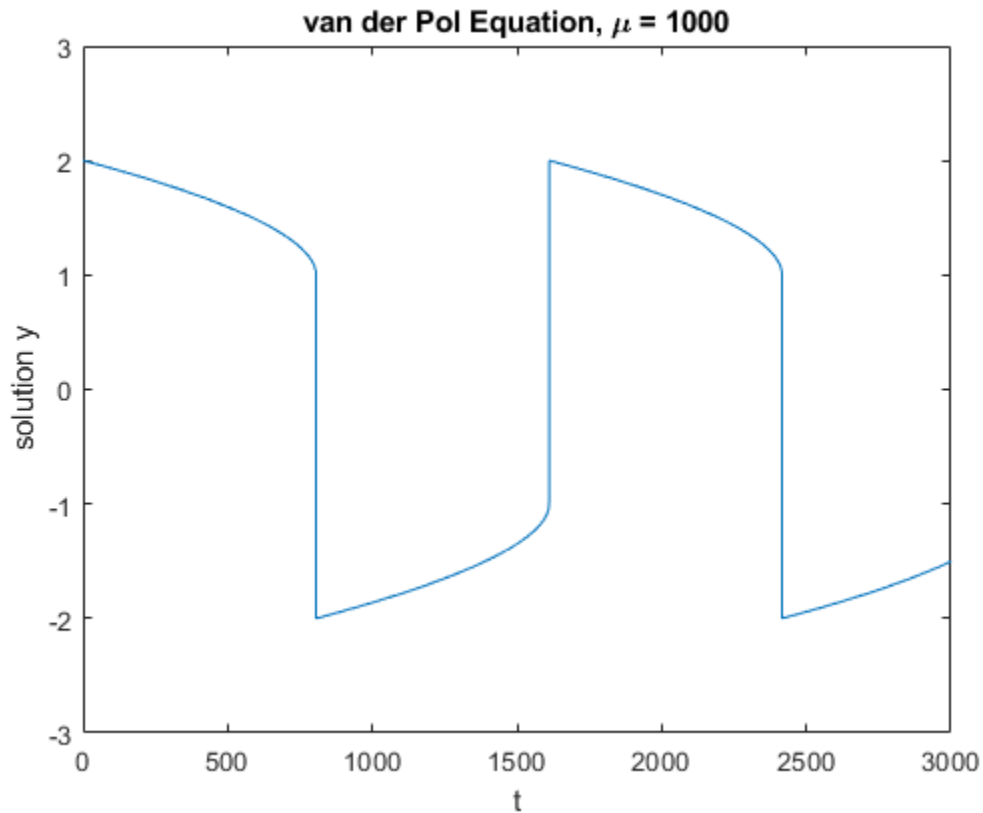
Boundary Value Problems

`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
```
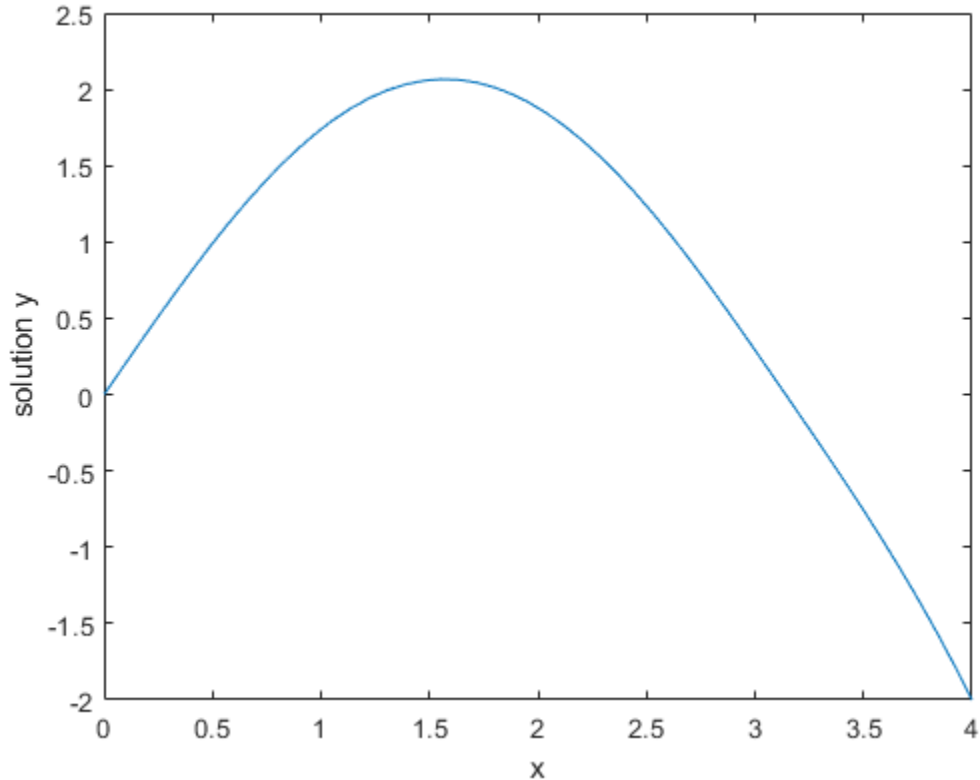
This particular boundary value problem has exactly two solutions. You can obtain the other solution by changing the initial guesses to $y(x) = -1$ and $y'(x) = 0$.

```
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
```
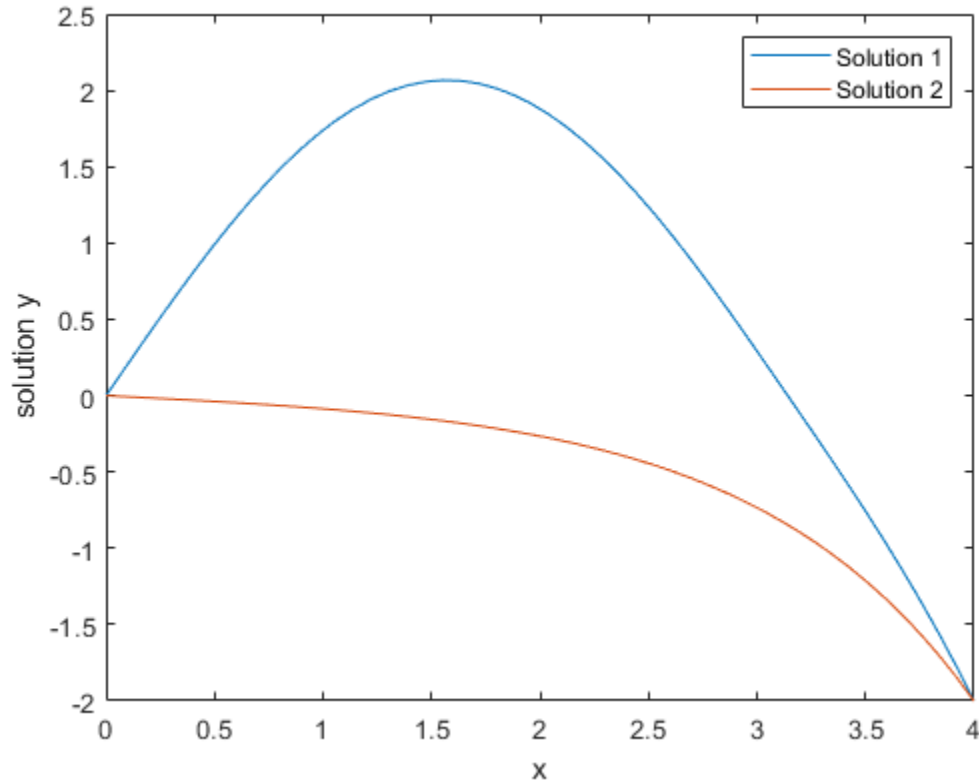
Delay Differential Equations

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 intergation $[0, 5]$ 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');
```

An example of Wille and Baker
DDE with Constant Delays
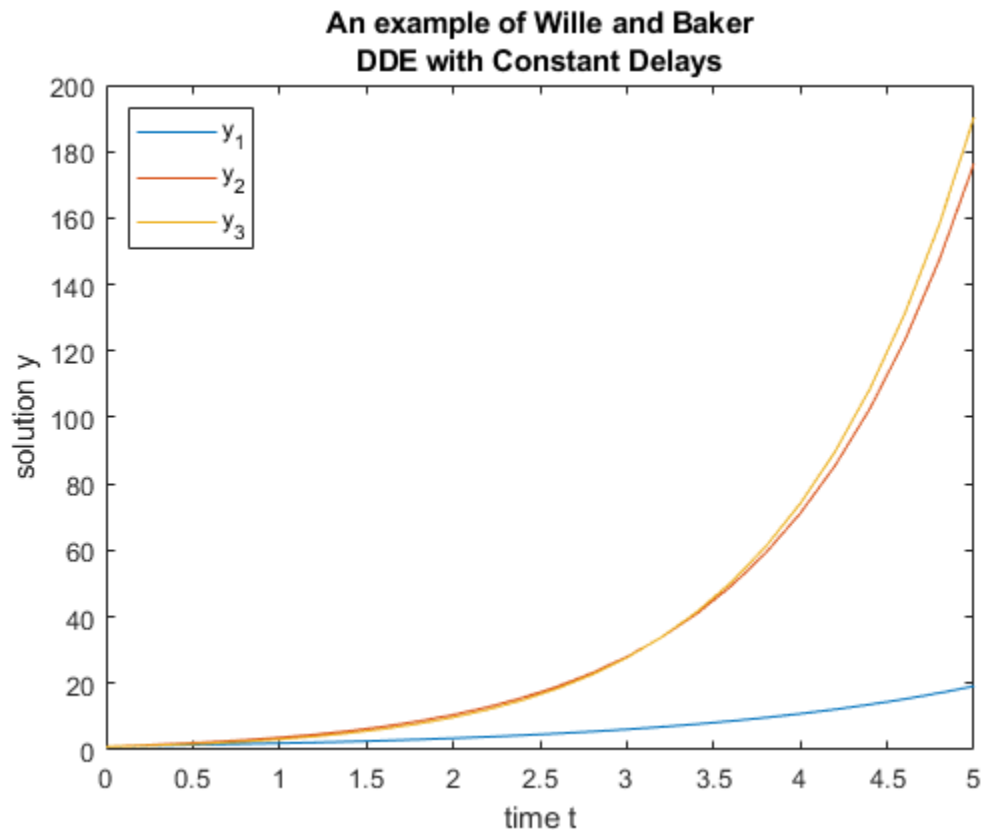
Partial Differential Equations

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');
```
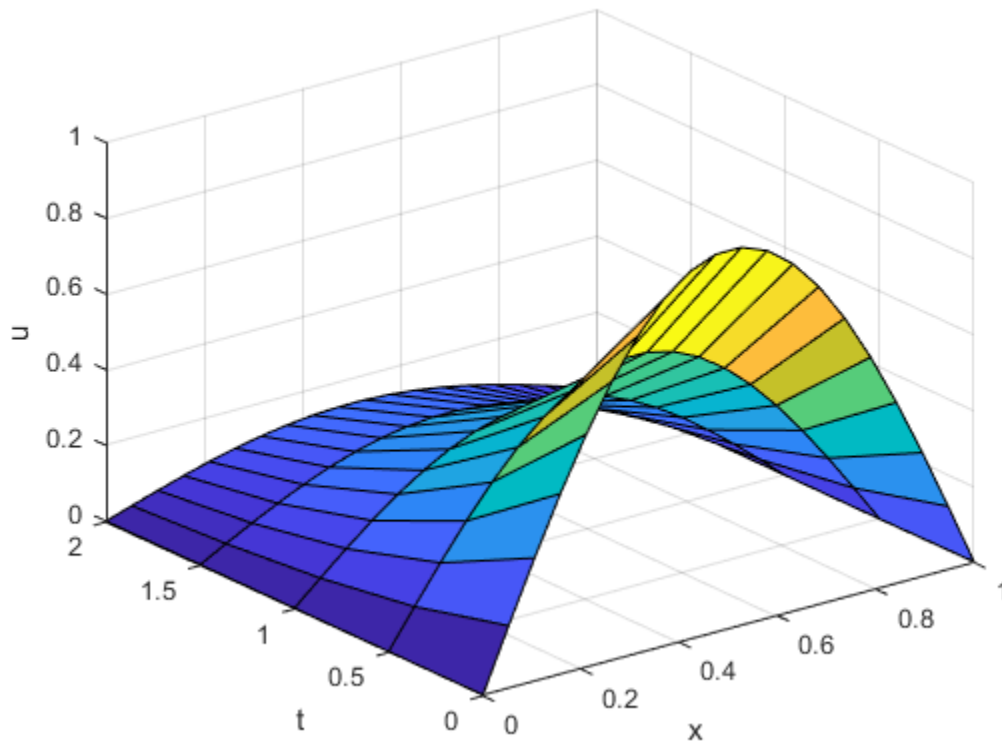
See Also
bvp4c | ode45 | pdepe

More About
- "Choose an ODE Solver" on page 11-2
- "Boundary-Value Problems" on page 12-2
- "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**:

$$\frac{dx}{dt} = x - \alpha xy$$

$$\frac{dy}{dt} = -y + \beta xy.$$

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')
```
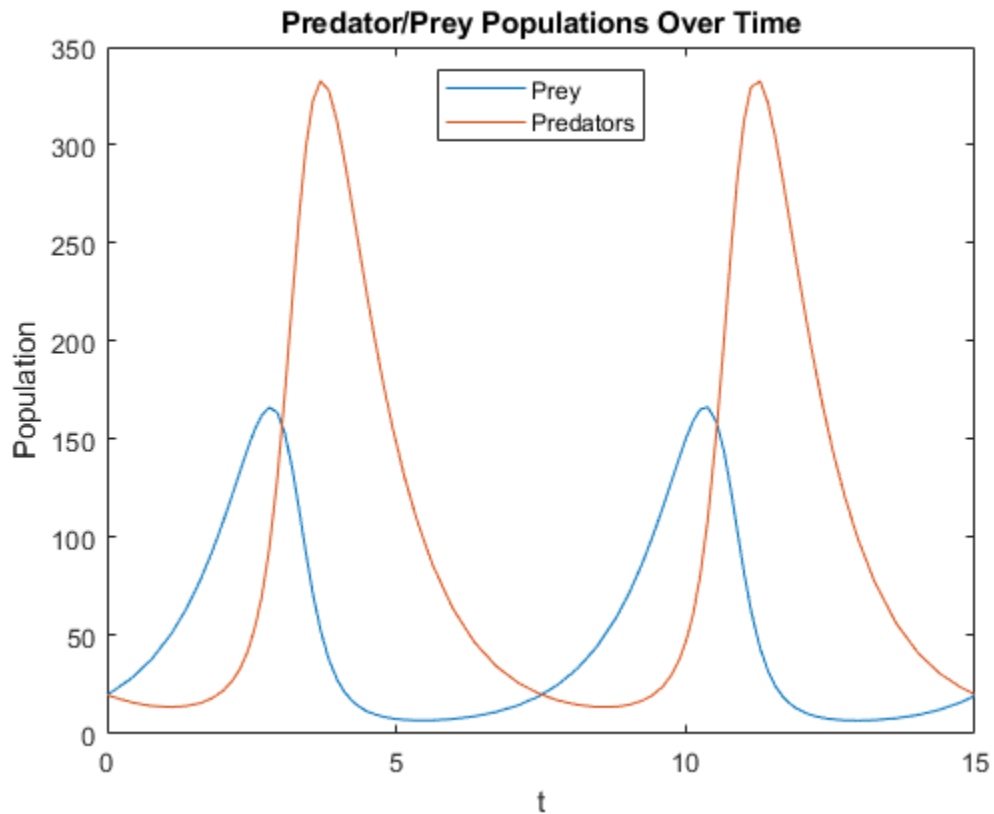
Now plot the populations against each other. The resulting phase plane plot makes the cyclic relationship between the populations very clear.

```
plot(y(:,1),y(:,2))
title('Phase Plane Plot')
xlabel('Prey Population')
ylabel('Predator Population')
```
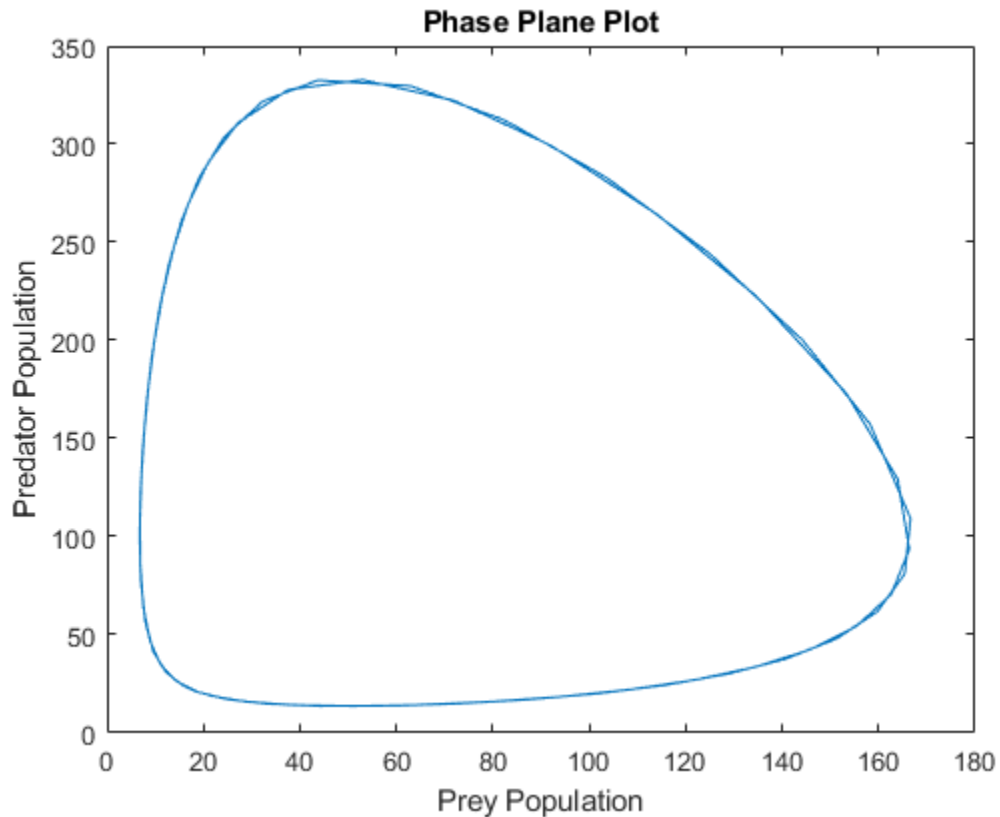
Compare Results of Different Solvers

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')
```

Phase Plane Plot
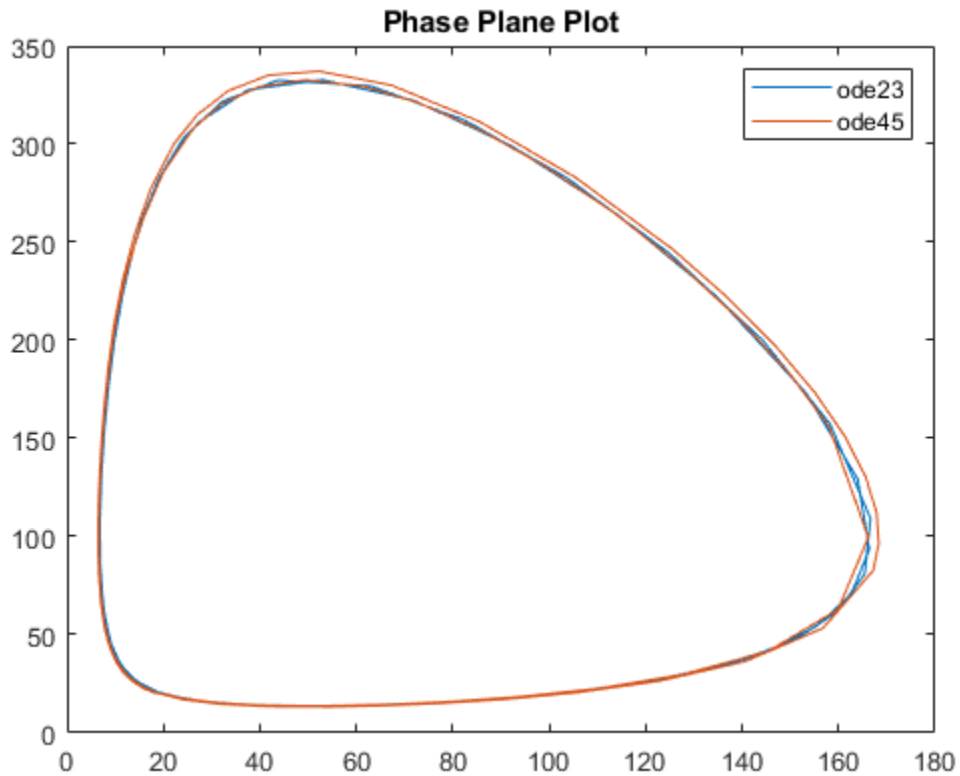
The results show that solving differential equations using different numerical methods can produce slightly different answers.

See Also

ode23 | ode45

More About

- "Choose an ODE Solver" on page 11-2
- "Solve Nonstiff ODEs" on page 11-25

Boundary Value Problems (BVPs)

Boundary-Value Problems

Function Summary

BVP Solver

Solver	Description
bvp4c	Solve boundary value problems for ordinary differential equations.
bvp5c	Solve boundary value problems for ordinary differential equations.

BVP Helper Functions

Function	Description
bvpinit	Form the initial guess for bvp4c.
deval	Evaluate the numerical solution using the output of bvp4c.

BVP Solver Options

An options structure contains named properties whose values are passed to bvp4c, and which affect problem solution. Use these functions to create, alter, or access an options structure.

Function	Description
`bvpset`	Create/alter the BVP options structure.
`bvpget`	Extract properties from options structure created with `bvpset`.

Boundary Value Problems

The BVP solver is designed to handle systems of ordinary differential equations

$$y' = f(x, y)$$

where x is the independent variable, y is the dependent variable, and y' represents the derivative of y with respect to x dy/dx.

See "Choose an ODE Solver" on page 11-2 for general information about ODEs.

Boundary Conditions

In a *boundary value problem,* the solution of interest satisfies certain boundary conditions. These conditions specify a relationship between the values of the solution at more than one x. In its basic syntax, `bvp4c` is designed to solve two-point BVPs, i.e., problems where the solution sought on an interval $[a, b]$ must satisfy the boundary conditions

$$g(y(a), y(b)) = 0$$

Unlike initial value problems, a boundary value problem may not have a solution, may have a finite number of solutions, or may have infinitely many solutions. As an integral part of the process of solving a BVP, you need to provide a guess for the required solution. The quality of this guess can be critical for the solver performance and even for a successful computation.

There may be other difficulties when solving BVPs, such as problems imposed on infinite intervals or problems that involve singular coefficients. Often BVPs involve unknown parameters p that have to be determined as part of solving the problem

$$y' = f(x, y, p)$$

$$g(y(a), y(b), p) = 0$$

In this case, the boundary conditions must suffice to determine the value of p.

BVP Solver

The BVP Solver

The function `bvp4c` solves two-point boundary value problems for ordinary differential equations (ODEs). It integrates a system of first-order ordinary differential equations

$$y' = f(x, y)$$

on the interval [*a*, *b*], subject to general two-point boundary conditions

$$bc(y(a), y(b)) = 0$$

It can also accommodate other types of BVP problems, such as those that have any of the following:

- Unknown parameters
- Singularities in the solutions
- Multipoint conditions

In this case, the number of boundary conditions must be sufficient to determine the solution and the unknown parameters.

`bvp4c` produces a solution that is continuous on [*a*,*b*] and has a continuous first derivative there. You can use the function `deval` and the output of `bvp4c` to evaluate the solution at specific points on the interval of integration.

`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, the solver adapts the mesh and repeats the process. The user *must* provide the points of the initial mesh as well as an initial approximation of the solution at the mesh points.

BVP Solver Syntax

The basic syntax of the BVP solver is

```
sol = bvp4c(odefun,bcfun,solinit)
```

The input arguments are

`odefun`	A function handle that evaluates the differential equations. It has the basic form `dydx = odefun(x,y)` where x is a scalar, and `dydx` and y are column vectors. `odefun` can also accept a vector of unknown parameters and a variable number of known parameters, (see "BVP Solver Options" on page 12-6).
`bcfun`	Handle to a function that evaluates the residual in the boundary conditions. It has the basic form `res = bcfun(ya,yb)` where `ya` and `yb` are column vectors representing y(a) and y(b), and `res` is a column vector of the residual in satisfying the boundary conditions. `bcfun` can also accept a vector of unknown parameters and a variable number of known parameters, (see "BVP Solver Options" on page 12-6).
`solinit`	Structure with fields x and y:

	x	Ordered nodes of the initial mesh. Boundary conditions are imposed at a = `solinit.x(1)` and b = `solinit.x(end)`.
	y	Initial guess for the solution with `solinit.y(:,i)` a guess for the solution at the node `solinit.x(i)`.

The structure can have any name, but the fields must be named x and y. It can also contain a vector that provides an initial guess for unknown parameters. You can form `solinit` with the helper function `bvpinit`. See the `bvpinit` reference page for details.

The output argument `sol` is a structure created by the solver. In the basic case the structure has fields x, y, yp, and `solver`.

`sol.x`	Nodes of the mesh selected by `bvp4c`
`sol.y`	Approximation to $y(x)$ at the mesh points of `sol.x`
`sol.yp`	Approximation to $y'(x)$ at the mesh points of `sol.x`
`sol.solver`	`'bvp4c'`

The structure `sol` returned by `bvp4c` contains an additional field if the problem involves unknown parameters:

`sol.parameters`	Value of unknown parameters, if present, found by the solver.

The function `deval` uses the output structure `sol` to evaluate the numerical solution at any point from `[a,b]`.

BVP Solver Options

For more advanced applications, you can specify solver options by passing an input argument `options`.

`options`	Structure of optional parameters that change the default integration properties. This is the fourth input argument.
	`sol = bvp4c(odefun,bcfun,solinit,options)`
	You can create the structure options using the function `bvpset`. The `bvpset` reference page describes the properties you can specify.

Integrator Options

The default integration properties in the BVP solver `bvp4c` are selected to handle common problems. In some cases, you can improve solver performance by overriding these defaults. You do this by supplying `bvp4c` with an `options` structure that specifies one or more property values.

For example, to change the value of the relative error tolerance of `bvp4c` from the default value of `1e-3` to `1e-4`,

1 Create an options structure using the function `bvpset` by entering

```
   options = bvpset('RelTol', 1e-4);
```
2   Pass the `options` structure to `bvp4c` as follows:

```
   sol = bvp4c(odefun,bcfun,solinit,options)
```

For a complete description of the available options, see the reference page for `bvpset`.

Examples

- "Mathieu's Equation" on page 12-7
- "Continuation" on page 12-11
- "Singular BVPs" on page 12-19
- "Multipoint BVPs" on page 12-23
- "Additional Examples" on page 12-27

Mathieu's Equation

- "Solving the Problem" on page 12-7
- "Finding Unknown Parameters" on page 12-10
- "Evaluating the Solution" on page 12-11

Solving the Problem

This example determines the fourth eigenvalue of Mathieu's Equation. It illustrates how to write second-order differential equations as a system of two first-order ODEs and how to use `bvp4c` to determine an unknown parameter λ.

The task is to compute the fourth ($q = 5$) eigenvalue lambda λ of Mathieu's equation

$$y'' + (\lambda - 2q \cos 2x)y = 0$$

Because the unknown parameter λ is present, this second-order differential equation is subject to *three* boundary conditions

$$y(0) = 1$$

$$y'(0) = 0$$

$$y'(\pi) = 0$$

12-7

Note The file, `mat4bvp.m`, contains the complete code for this example. All the functions required by `bvp4c` are coded in this file as nested or local functions. To see the code in an editor, type `edit mat4bvp` at the command line. To run it, type `mat4bvp` at the command line.

---

1 **Rewrite the problem as a first-order system.** To use `bvp4c`, you must rewrite the equations as an equivalent system of first-order differential equations. Using a substitution $y_1 = y$ and $y_2 = y'$, the differential equation is written as a system of two first-order equations

$$y_1' = y_2$$

$$y_2' = -(\lambda - 2q \cos 2x)y_1$$

Note that the differential equations depend on the unknown parameter λ. The boundary conditions become

$$y_1(0) - 1 = 0$$

$$y_2(0) = 0$$

$$y_2(\pi) = 0$$

2 **Code the system of first-order ODEs.** Once you represent the equation as a first-order system, you can code it as a function that `bvp4c` can use. Because there is an unknown parameter, the function must be of the form

```
dydx = odefun(x,y,parameters)
```

The following code represents the system in the function, `mat4ode`. Variable `q` is shared with the outer function:

```
function dydx = mat4ode(x,y,lambda)
dydx = [ y(2)
         -(lambda - 2*q*cos(2*x))*y(1) ];
end   % End nested function mat4ode
```

3 **Code the boundary conditions function.** You must also code the boundary conditions in a function. Because there is an unknown parameter, the function must be of the form

```
res = bcfun(ya,yb,parameters)
```

The code below represents the boundary conditions in the function, `mat4bc`.

```
function res = mat4bc(ya,yb,lambda)
res = [  ya(2)
         yb(2)
         ya(1)-1 ];
```

4 **Create an initial guess.** To form the guess structure `solinit` with `bvpinit`, you
need to provide initial guesses for both the solution and the unknown parameter.

The function `mat4init` provides an initial guess for the solution. `mat4init` uses $y =$
cos4x because this function satisfies the boundary conditions and has the correct
qualitative behavior (the correct number of sign changes).

```
function yinit = mat4init(x)
yinit = [  cos(4*x)
          -4*sin(4*x) ];
```

In the call to `bvpinit`, the third argument, `lambda`, provides an initial guess for the
unknown parameter λ.

```
lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
```

This example uses the `@` symbol to pass `mat4init` as a function handle to `bvpinit`.

5 **Apply the BVP solver.** The `mat4bvp` example calls `bvp4c` with the functions
`mat4ode` and `mat4bc` and the structure `solinit` created with `bvpinit`.

```
sol = bvp4c(@mat4ode,@mat4bc,solinit);
```

6 **View the results.** Complete the example by displaying the results:

a Print the value of the unknown parameter λ found by `bvp4c`.

```
fprintf('Fourth eigenvalue is approximately %7.3f.\n',...
        sol.parameters)
```

b Use `deval` to evaluate the numerical solution at 100 equally spaced points in the
interval [0, π], and plot its first component. This component approximates $y(x)$.

```
xint = linspace(0,pi);
Sxint = deval(sol,xint);
plot(xint,Sxint(1,:))
axis([0 pi -1 1.1])
title('Eigenfunction of Mathieu''s equation.')
xlabel('x')
ylabel('solution y')
```

The following plot shows the eigenfunction associated with the final eigenvalue λ = 17.097.

Eigenfunction of Mathieu's equation.
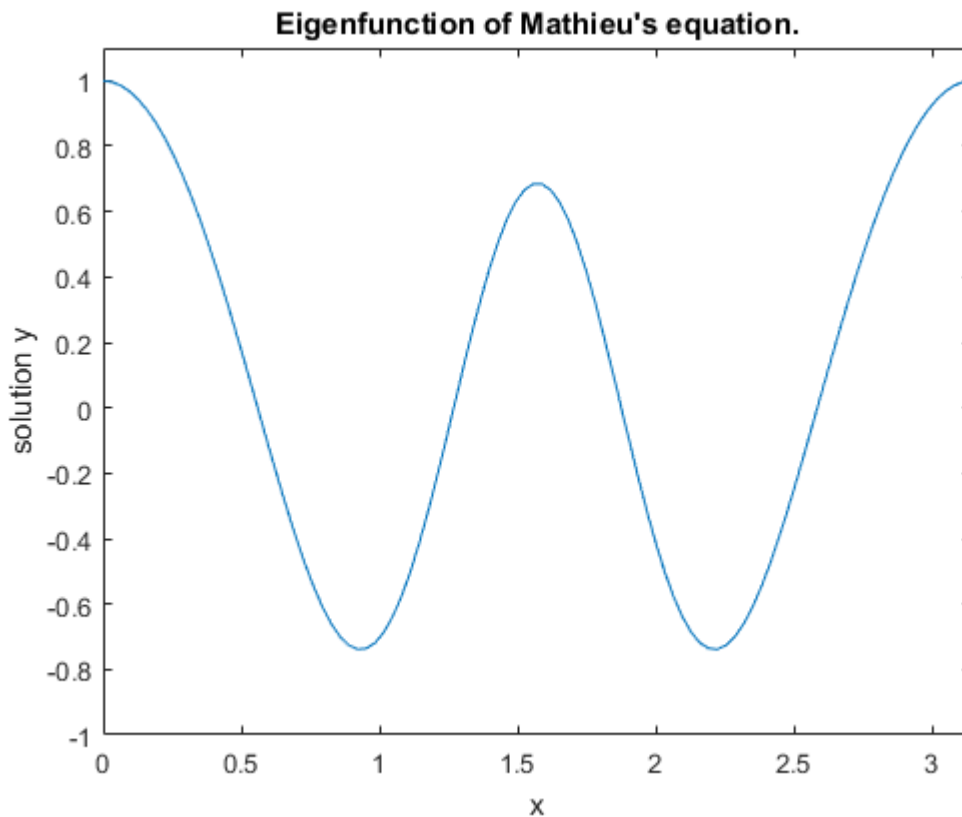
Finding Unknown Parameters

The `bvp4c` solver can find unknown parameters p for problems of the form

$$y' = f(x,y,p)$$

$$bc(y(a), y(b),p) = 0$$

You must provide `bvp4c` an initial guess for any unknown parameters in the vector `solinit.parameters`. When you call `bvpinit` to create the structure `solinit`, specify the initial guess as a vector in the additional argument `parameters`.

```
solinit = bvpinit(x,v,parameters)
```

The bvp4c function arguments odefun and bcfun must each have a third argument.

```
dydx = odefun(x,y,parameters)
res = bcfun(ya,yb,parameters)
```

While solving the differential equations, bvp4c adjusts the value of unknown parameters to satisfy the boundary conditions. The solver returns the final values of these unknown parameters in sol.parameters.

Evaluating the Solution

The collocation method implemented in bvp4c produces a C^1-continuous solution over the whole 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 bvp4c.

```
Sxint = deval(sol,xint)
```

The deval function is vectorized. For a vector xint, the ith column of Sxint approximates the solution $y(xint(i))$.

Continuation

Introduction

To solve a boundary value problem, you need to provide an initial guess for the solution. The quality of your initial guess can be critical to the solver performance, and to being able to solve the problem at all. However, coming up with a sufficiently good guess can be the most challenging part of solving a boundary value problem. Certainly, you should apply the knowledge of the problem's physical origin. Often a problem can be solved as a sequence of relatively simpler problems, i.e., a continuation.

This example shows how to use continuation to:

- Solve a difficult BVP
- Verify a solution's consistent behavior

Using Continuation to Solve a BVP

This example solves the differential equation

$$\varepsilon y'' \qquad + \qquad xy' \qquad = \qquad \varepsilon \pi^2 \qquad \cos(\pi x) \qquad - \qquad \pi x \qquad \sin(\pi x)$$

for $\varepsilon = 10^{-4}$, on the interval $[-1\ 1]$, with boundary conditions $y(-1) = -2$ and $y(1) = 0$. For $0 < \varepsilon < 1$, the solution has a transition layer at $x = 0$. Because of this rapid change in the solution for small values of ε, the problem becomes difficult to solve numerically.

The example solves the problem as a sequence of relatively simpler problems, i.e., a continuation. The solution of one problem is used as the initial guess for solving the next problem.

Note The file, shockbvp.m, contains the complete code for this example. All required functions are coded as nested functions in this file. To see the code in an editor, type edit shockbvp at the command line. To run it, type shockbvp at the command line.

Note This problem appears in [1] to illustrate the mesh selection capability of a well established BVP code COLSYS.

---

1 **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that bvp4c can use:

The code below represents the differential equation and the boundary conditions in the functions shockODE and shockBC. Note that shockODE is vectorized to improve solver performance. The additional parameter ε is represented by e and is shared with the outer function.

```
function dydx = shockODE(x,y)
pix = pi*x;
dydx = [   y(2,:)
           -x/e.*y(2,:) - pi^2*cos(pix) - pix/e.*sin(pix) ];
end %   End nested function shockODE

function res = shockBC(ya,yb)
res = [ ya(1)+2
        yb(1)    ];
end %   End nested function shockBC
```

2 **Provide analytical partial derivatives.** For this problem, the solver benefits from using analytical partial derivatives. The code below represents the derivatives in functions shockJac and shockBCJac.

```
function jac = shockJac(x,y)
jac = [ 0    1
        0 -x/e ];
end %   End nested function shockJac

function [dBCdya,dBCdyb] = shockBCJac(ya,yb)
dBCdya = [ 1 0
           0 0 ];
dBCdyb = [ 0 0
           1 0 ];
end %   End nested function shockBCJac
```

shockJac shares e with the outer function.

Tell bvp4c to use these functions to evaluate the partial derivatives by setting the options FJacobian and BCJacobian. Also set 'Vectorized' to 'on' to indicate that the differential equation function shockODE is vectorized.

```
options = bvpset('FJacobian',@shockJac,...
                 'BCJacobian',@shockBCJac,...
                 'Vectorized','on');
```

3 **Create an initial guess.** You must provide bvp4c with a guess structure that contains an initial mesh and a guess for values of the solution at the mesh points. A constant guess of $y(x) \equiv 1$ and $y'(x) \equiv 0$, and a mesh of five equally spaced points on [–1 1] suffice to solve the problem for $\varepsilon = 10^{-2}$. Use bvpinit to form the guess structure.

```
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
```

4 **Use continuation to solve the problem.** To obtain the solution for the parameter $\varepsilon = 10^{-4}$, the example uses continuation by solving a sequence of problems for $\varepsilon = 10^{-2}$, 10^{-3}, 10^{-4}. The solver bvp4c does not perform continuation automatically, but the code's user interface has been designed to make continuation easy. The code uses the output sol that bvp4c produces for one value of e as the guess in the next iteration.

```
e = 0.1;
for i=2:4
    e = e/10;
    sol = bvp4c(@shockODE,@shockBC,sol,options);
end
```

5 **View the results.** Complete the example by displaying the final solution

```
plot(sol.x,sol.y(1,:))
axis([-1 1 -2.2 2.2])
title(['There is a shock at x = 0 when \epsilon = '...
       sprintf('%.e',e) '.'])
xlabel('x')
ylabel('solution y')
```


There is a shock at x = 0 when ϵ =1e-04.
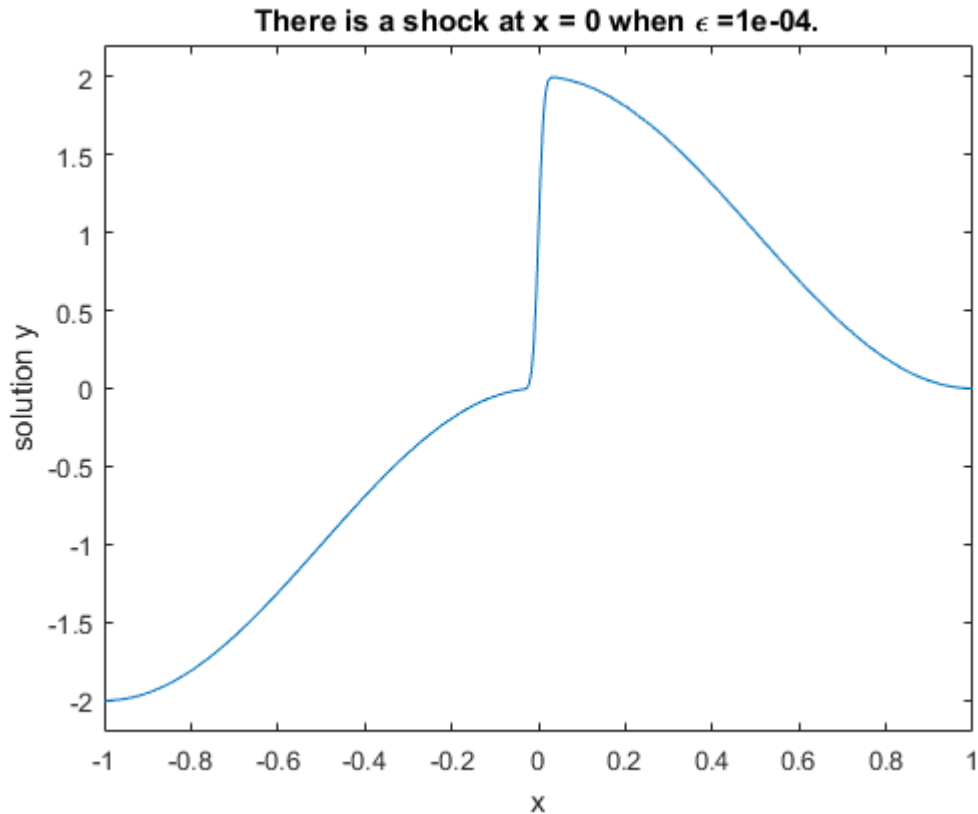
Using Continuation to Verify Consistency

Falkner-Skan BVPs arise from similarity solutions of viscous, incompressible, laminar flow over a flat plate. An example is

$$f'' + ff' + \beta(1 - (f')^2) = 0$$

for β = 0.5 on the interval [0, ∞] with boundary conditions $f(0) = 0$, $f'(0) = 0$, and $f'(∞) = 1$.

The BVP cannot be solved on an infinite interval, and it would be impractical to solve it for even a very large finite interval. So, the example tries to solve a sequence of problems posed on increasingly larger intervals to verify the solution's consistent behavior as the boundary approaches ∞.

The example imposes the infinite boundary condition at a finite point called `infinity`. The example then uses continuation in this end point to get convergence for increasingly larger values of `infinity`. It uses `bvpinit` to extrapolate the solution `sol` for one value of `infinity` as an initial guess for the new value of `infinity`. The plot of each successive solution is superimposed over those of previous solutions so they can easily be compared for consistency.

Note The file, `fsbvp.m`, contains the complete code for this example. All required functions are coded as nested functions in this file. To see the code in an editor, type `edit fsbvp` at the command line. To run it, type `fsbvp` at the command line.

---

1   **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that `bvp4c` can use. The problem parameter `beta` is shared with the outer function.

```
function dfdeta = fsode(eta,f)
dfdeta = [ f(2)
             f(3)
           -f(1)*f(3) - beta*(1 - f(2)^2) ];
end %   End nested function fsode

function res = fsbc(f0,finf)
res = [f0(1)
       f0(2)
       finf(2) - 1];
end %    End nested function fsbc
```
2   **Create an initial guess.** You must provide `bvp4c` with a guess structure that contains an initial mesh and a guess for values of the solution at the mesh points. A crude mesh of five points and a constant guess that satisfies the boundary conditions are good enough to get convergence when `infinity = 3`.

```
infinity = 3;
maxinfinity = 6;
```

```
solinit = bvpinit(linspace(0,infinity,5),[0 0 1]);
```

3 Solve on the initial interval. The example obtains the solution for `infinity = 3`. It then prints the computed value of $f''(0)$ for comparison with the value reported by Cebeci and Keller [2]:

```
sol = bvp4c(@fsode,@fsbc,solinit);
eta = sol.x;
f = sol.y;

fprintf('\n');
fprintf('Cebeci & Keller report that f''''(0) = 0.92768.\n')
fprintf('Value computed using infinity = %g is %7.5f.\n', ...
        infinity,f(3,1))
```

The example prints

```
Cebeci & Keller report that f''(0) = 0.92768.
Value computed using infinity = 3 is 0.92915.
```

4 Setup the figure and plot the initial solution.

```
figure
plot(eta,f(2,:),eta(end),f(2,end),'o');
axis([0 maxinfinity 0 1.4]);
title('Falkner-Skan equation, positive wall shear, \beta = 0.5.')
xlabel('\eta')
ylabel('df/d\eta')
hold on
drawnow
shg
```

Falkner-Skan equation, positive wall shear, $\beta = 0.5$.
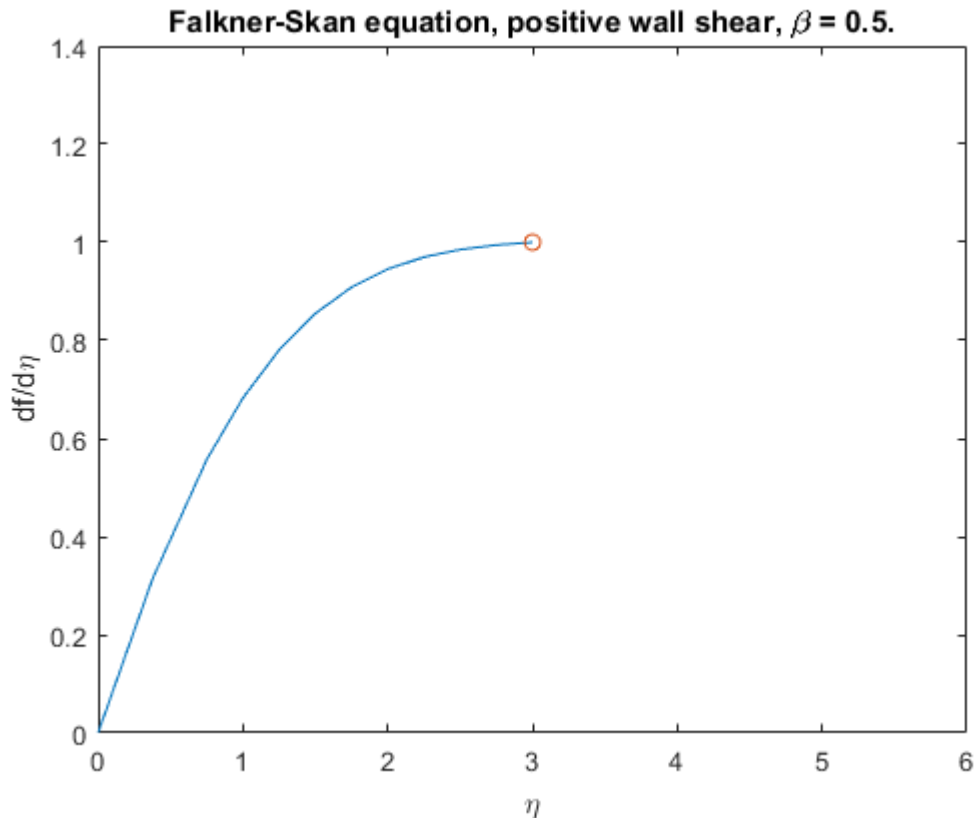
5   **Use continuation to solve the problem and plot subsequent solutions.** The
example then solves the problem for `infinity = 4, 5, 6`. It uses `bvpinit` to
extrapolate the solution `sol` for one value of `infinity` as an initial guess for the
next value of `infinity`. For each iteration, the example prints the computed value of
$f''(0)$ and superimposes a plot of the solution in the existing figure.

```
for Bnew = infinity+1:maxinfinity

    solinit = bvpinit(sol,[0 Bnew]); % Extend solution to Bnew.
    sol = bvp4c(@fsode,@fsbc,solinit);
    eta = sol.x;
    f = sol.y;

    fprintf('Value computed using infinity = %g is %7.5f.\n', ...
```

12-17

```
             Bnew,f(3,1))
  plot(eta,f(2,:),eta(end),f(2,end),'o');
  drawnow

end
hold off
```

The example prints

```
Value computed using infinity = 4 is 0.92774.
Value computed using infinity = 5 is 0.92770.
Value computed using infinity = 6 is 0.92770.
```

Note that the values approach 0.92768 as reported by Cebeci and Keller. The superimposed plots confirm the consistency of the solution's behavior.

Falkner-Skan equation, positive wall shear, $\beta = 0.5$.
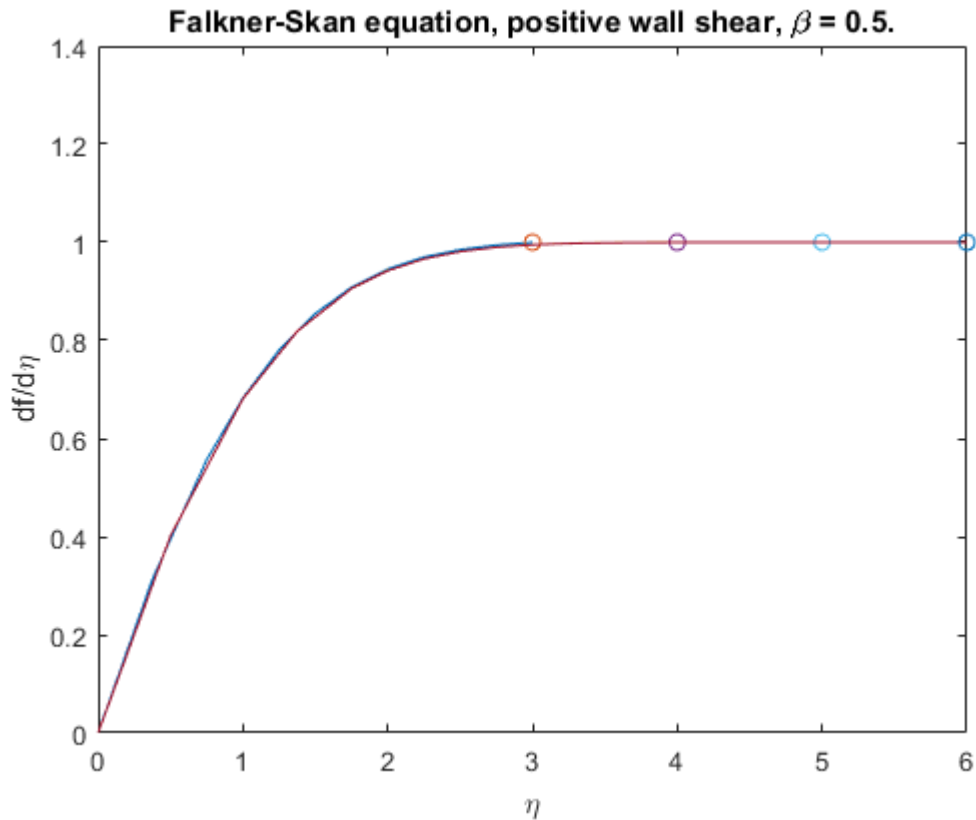
Singular BVPs

- "Introduction" on page 12-19
- "Emden's equation" on page 12-20

Introduction

The function bvp4c solves a class of singular BVPs of the form

$$y' = \frac{1}{x} S y + f(x, y)$$
$$0 = g\big(y(0), y(b)\big)$$

(12-1)

12-19

It can also accommodate unknown parameters for problems of the form

$$y' = \frac{1}{x} Sy + f(x, y, p)$$
$$0 = \big(g(y(0), y(b), p) \big)$$

Singular problems must be posed on an interval [0,*b*] with *b* > 0. Use `bvpset` to pass the constant matrix *S* to `bvp4c` as the value of the `'SingularTerm'` integration property. Boundary conditions at *x* = 0 must be consistent with the necessary condition for a smooth solution, *Sy*(0) = 0. An initial guess should also satisfy this necessary condition.

When you solve a singular BVP using

`sol = bvp4c(@odefun,@bcfun,solinit,options)`

`bvp4c` requires that your function `odefun(x,y)` return only the value of the *f*(*x*, *y*) term in Equation 5-2.

Emden's equation

Emden's equation arises in modeling a spherical body of gas. The PDE of the model is reduced by symmetry to the ODE

$$y'' + \frac{2}{x} y' + y^5 = 0$$

on an interval [0,1]. The coefficient 2/*x* is singular at *x* = 0, but symmetry implies the boundary condition *y*'(0) = 0. With this boundary condition, the term

$$\frac{2}{x} y'(0)$$

is well-defined as *x* approaches 0. For the boundary condition $y(1) = \sqrt{3/2}$, this BVP has the analytical solution

$$y(x) = \left(1 + \frac{x^2}{3} \right)^{-1/2}$$

Note The file, `emdenbvp.m`, contains the complete code for this example. It contains all the required functions coded as local functions. To see the code in an editor, type `edit emdenbvp` at the command line. To run it, type `emdenbvp` at the command line.

---

1 **Rewrite the problem as a first-order system and identify the singular term.**
Using a substitution $y_1 = y$ and $y_2 = y'$, write the differential equation as a system of two first-order equations

$$y_1' = y_2$$
$$y_2' = -\frac{2}{x}y_2 - y_1^5$$

The boundary conditions become

$$y_2(0) = 0$$
$$y_1(1) = \sqrt{3/2}$$

Writing the ODE system in a vector-matrix form

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \frac{1}{x}\begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} y_2 \\ -y_1^5 \end{bmatrix}$$

the terms of Equation 5-2 are identified as

$$S = \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix}$$

and

$$f(x,y) = \begin{bmatrix} y_2 \\ -y_1^5 \end{bmatrix}$$

2 **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that `bvp4c` can use.

```
function dydx = emdenode(x,y)
dydx = [  y(2)
```

```
                 -y(1)^5 ];
    function res = emdenbc(ya,yb)
    res = [ ya(2)
            yb(1) - sqrt(3)/2 ];
```

3 **Setup integration properties.** Use the matrix as the value of the `'SingularTerm'` integration property.

```
S = [0,0;0,-2];
options = bvpset('SingularTerm',S);
```

4 **Create an initial guess.** This example starts with a mesh of five points and a constant guess for the solution.

$$y_1(x) \equiv \sqrt{3/2}$$

$$y_2(x) \equiv 0$$

Use `bvpinit` to form the guess structure

```
guess = [sqrt(3)/2;0];
solinit = bvpinit(linspace(0,1,5),guess);
```

5 **Solve the problem.** Use the standard `bvp4c` syntax to solve the problem.

```
sol = bvp4c(@emdenode,@emdenbc,solinit,options);
```

6 **View the results.** This problem has an analytical solution

$$y(x) = \left(1 + \frac{x^2}{3} \right)^{-1/2}$$

The example evaluates the analytical solution at 100 equally spaced points and plots it along with the numerical solution computed using `bvp4c`.

```
x = linspace(0,1);
truy = 1 ./ sqrt(1 + (x.^2)/3);
plot(x,truy,sol.x,sol.y(1,:),'ro');
title('Emden problem -- BVP with singular term.')
legend('Analytical','Computed');
xlabel('x');
ylabel('solution y');
```
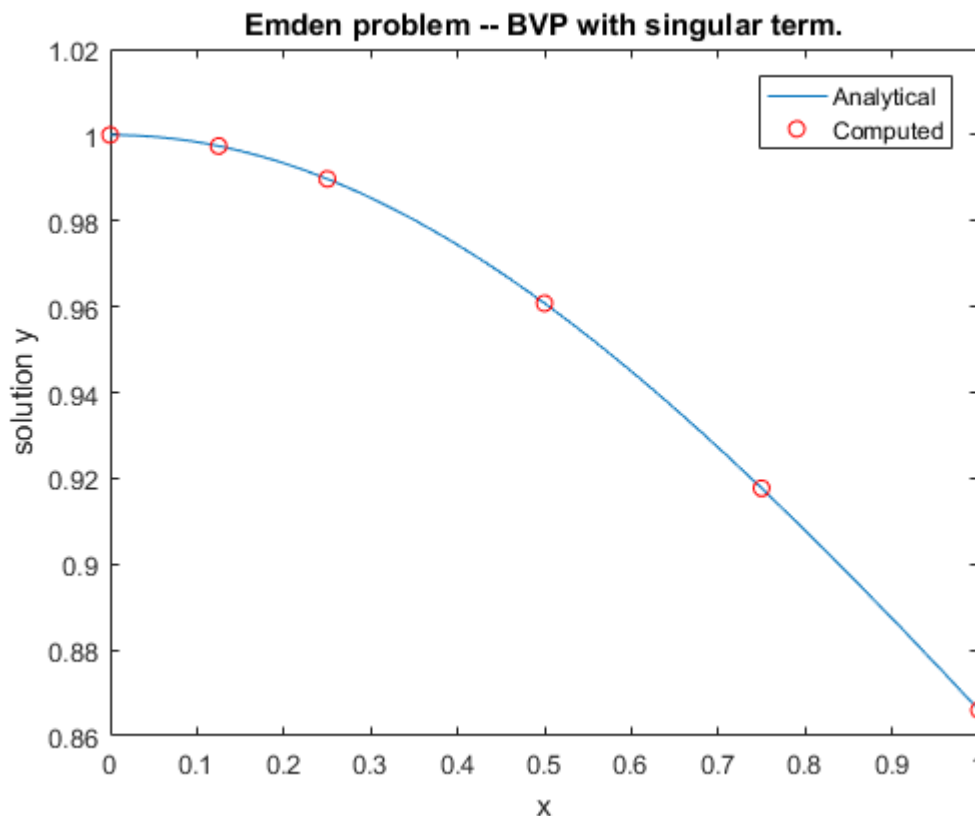
Multipoint BVPs

In multipoint boundary value problems, the solution of interest satisfies conditions at points inside the interval of integration. The `bvp4c` function is useful in solving such problems.

The following example shows how the multipoint capability in `bvp4c` can improve efficiency when you are solving a nonsmooth problem. The following equations are solved on $0 \leq x \leq \lambda$ for constant parameters n, κ, $\lambda > 1$, and $\eta = \lambda^2/(n \times \kappa^2)$. These are subject to boundary conditions `v(0) = 0` and $C(\lambda) = 1$:

```
v' = (C - 1)/n
C' = (v * C - min(x,1))/η
```

The term `min(x,1)` is not smooth at `xc = 1`, and this can affect the solver's efficiency. By introducing an interface point at `xc = 1`, smooth solutions can be obtained on [0,1] and [1,λ]. To get a continuous solution over the entire interval [0,λ], the example imposes matching conditions at the interface.

Note The file, `threebvp.m`, contains the complete code for this example, and it solves the problem for λ = 2, n = 0.05, and several values of κ. All required functions are coded as nested functions in `threebvp.m`. To see the code in an editor, type `edit threebvp` at the command line. To run it, type `threebvp` at the command line.

The example takes you through the following steps:

1 **Determine the interfaces and divide the interval of integration into regions.**
Introducing an interface point at `xc = 1` divides the problem into two regions in which the solutions remain smooth. The differential equations for the two regions are

Region 1: $0 \le x \le 1$

```
v' = (C - 1)/n
C' = (v * C - x)/η
```

Region 2: $1 \le x \le λ$

```
v' = (C - 1)/n
C' = (v * C - 1)/η
```

Note that the interface `xc = 1` is included in both regions. At `xc = 1`, `bvp4c` produces a *left* and *right* solution. These solutions are denoted as `v(1-)`, `C(1-)` and `v(1+)`, `C(1+)` respectively.

2 **Determine the boundary conditions.** Solving two first-order differential equations in two regions requires imposing four boundary conditions. Two of these conditions come from the original formulation; the others enforce the continuity of the solution across the interface `xc = 1`:

```
v(0) = 0
C(λ) - 1 = 0
v(1-) - v(1+) = 0
C(1-) - C(1+) = 0
```

Here, `v(1-)`, `C(1-)` and `v(1+)`, `C(1+)` denote the left and right solution at the interface.

3   **Code the derivative function.** In the derivative function, `y(1)` corresponds to $v(x)$, and `y(2)` corresponds to $C(x)$. The additional input argument `region` identifies the region in which the derivative is evaluated. `bvp4c` enumerates regions from left to right, starting with 1. Note that the problem parameters n and η are shared with the outer function:

```
function dydx = f(x,y,region)
   dydx = zeros(2,1);
   dydx(1) = (y(2) - 1)/n;

   % The definition of C'(x) depends on the region.
   switch region
      case 1                           % x in [0 1]
         dydx(2) = (y(1)*y(2) - x)/η;
      case 2                           % x in [1 λ ]
         dydx(2) = (y(1)*y(2) - 1)/η;
   end
end                 % End nested function f
```

4   **Code the boundary conditions function.** For multipoint BVPs, the arguments of the boundary conditions function, YL and YR, become matrices. In particular, the kth column `YL(:,k)` represents the solution at the left boundary of the kth region. Similarly, `YR(:,k)` represents the solution at the right boundary of the kth region.

In the example, y(0) is approximated by `YL(:,1)`, while y(λ) is approximated by `YR(:,end)`. Continuity of the solution at the internal interface requires that `YR(:,1) = YL(:,2)`. Nested function `bc` computes the residual in the boundary conditions:

```
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( λ ) = 1
end                               % End nested function bc
```

5   **Create an initial guess.** For multipoint BVPs, when creating an initial guess using `bvpinit`, use double entries in `xinit` for the interface point `xc`. This example uses a constant guess `yinit = [1;1]`:

```
xc = 1;
xinit = [0, 0.25, 0.5, 0.75, xc, xc, 1.25, 1.5, 1.75, 2];
solinit = bvpinit(xinit,yinit)
```

For multipoint BVPs, you can use different guesses in different regions. To do that, you specify the initial guess for y as a function using the following syntax:

```
solinit = bvpinit(xinit,@yinitfcn)
```

The initial guess function must have the following general form:

```
function y = yinitfcn(x,region)
   switch region
   case 1              % x in [0, 1]

      y = [1;1];    % initial guess for y(x)  0 ≤ x ≤ 1

   case 2 % x in [1, λ ]

      y = [1;1];    % initial guess for y(x), 1 ≤ x ≤ λ
end
```

6 **Apply the solver.** The bvp4c function uses the same syntax for multipoint BVPs as it does for two-point BVPs:

```
sol = bvp4c(@f,@bc,solinit);
```

The mesh points returned in sol.x are adapted to the solution behavior, but the mesh still includes a double entry for the interface point xc = 1. Corresponding columns of sol.y represent the left and right solution at xc.

7 **View the results.** Using deval, the solution can be evaluated at any point in the interval of integration.

Note that, with the left and right values computed at the interface, the solution is not uniquely defined at xc = 1. When evaluating the solution exactly at the interface, deval issues a warning and returns the average of the left and right solution values. Call deval at xc-eps(xc) and xc+eps(xc) to get the limit values at xc.

The example plots the solution approximated at the mesh points selected by the solver:

```
plot(sol.x,sol.y(1,:),sol.x,sol.y(2,:),'--')
legend('v(x)','C(x)')
title(['A three-point BVP solved with BVP4C'])
xlabel(['\lambda = 2, \kappa = 5.'])
ylabel('v and C')
```
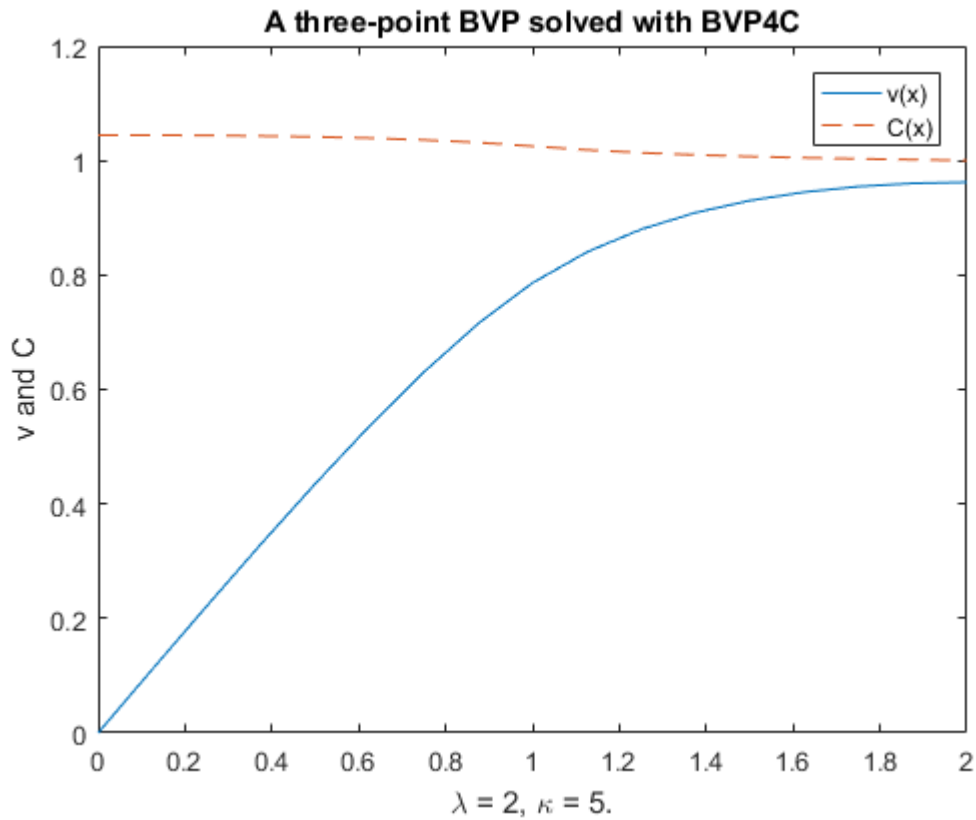
A three-point BVP solved with BVP4C

$\lambda = 2$, $\kappa = 5$.

Additional Examples

The following additional examples are available. Type

edit *examplename*

to view the code and

examplename

to run the example.

Example Name	Description
emdenbvp	Emden's equation, a singular BVP
fsbvp	Falkner-Skan BVP on an infinite interval
mat4bvp	Fourth eigenfunction of Mathieu's equation
shockbvp	Solution with a shock layer near x = 0
twobvp	BVP with exactly two solutions
threebvp	Three-point boundary value problem

For additional examples, see Tutorial on Solving BVPs with BVP4C.

References

[1] Ascher, U., R. Mattheij, and R. Russell, "*Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*," SIAM, Philadelphia, PA, 1995, p. 372.

[2] 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 | ode45 | pdepe

Partial Differential Equations (PDEs)

Partial Differential Equations

In this section...
"Function Summary" on page 13-2
"Initial Value Problems" on page 13-2
"PDE Solver" on page 13-3
"Integrator Options" on page 13-6
"Examples" on page 13-7

Function Summary

- "PDE Solver" on page 13-2
- "PDE Helper Function" on page 13-2

PDE Solver

This is the MATLAB PDE solver.

PDE Initial-Boundary Value Problem Solver	Description
pdepe	Solve initial-boundary value problems for systems of parabolic and elliptic PDEs in one space variable and time.

PDE Helper Function

PDE Helper Function	Description
pdeval	Evaluate the numerical solution of a PDE using the output of pdepe.

Initial Value Problems

pdepe solves systems of parabolic and elliptic PDEs in one spatial variable x and time t, of the form

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^{m}f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

$$(13\text{-}1)$$

The PDEs hold for $t_0 \leq t \leq t_f$ and $a \leq x \leq b$. The 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 $a \geq 0$ must also hold.

In "Equation 13-1", $f(x,t,u,\partial u/\partial x)$ is a flux term and $s(x,t,u,\partial u/\partial x)$ is a source term. The flux term must depend on $\partial u/\partial x$. The coupling of the partial derivatives with respect to time is restricted to multiplication by a diagonal matrix $c(x,t,u,\partial u/\partial x)$. The diagonal elements of this matrix are either identically zero or positive. An element that is identically zero corresponds to an elliptic equation and otherwise 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. Discontinuities in c and/or s due to material interfaces are permitted provided that a mesh point is placed at each interface.

At the initial time $t = t_0$, for all x the solution components satisfy initial conditions of the form

$$u(x,t_0) = u_0(x).$$

(13-2)

At the boundary $x = a$ or $x = b$, for all t the solution components satisfy a boundary condition of the form

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0.$$

(13-3)

$q(x,t)$ is a diagonal matrix with elements that are either identically zero or never zero. Note that the boundary conditions are expressed in terms of the f rather than partial derivative of u with respect to x $\partial u/\partial x$. Also, of the two coefficients, only p can depend on u.

PDE Solver

The PDE Solver

The MATLAB PDE solver, `pdepe`, solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t. There must be at least one parabolic equation in the system.

The `pdepe` solver converts the PDEs to ODEs using a second-order accurate spatial discretization based on a fixed set of user-specified nodes. The discretization method is

described in [1]. The time integration is done with `ode15s`. The `pdepe` solver exploits the capabilities of `ode15s` for solving the differential-algebraic equations that arise when "Equation 13-1" contains elliptic equations, and for handling Jacobians with a specified sparsity pattern. `ode15s` changes both the time step and the formula dynamically.

After discretization, elliptic equations give rise to algebraic equations. If the elements of the initial conditions vector that correspond to elliptic equations are not "consistent" with the discretization, `pdepe` tries to adjust them before beginning the time integration. For this reason, the solution returned for the initial time may have a discretization error comparable to that at any other time. If the mesh is sufficiently fine, `pdepe` can find consistent initial conditions close to the given ones. If `pdepe` displays a message that it has difficulty finding consistent initial conditions, try refining the mesh. No adjustment is necessary for elements of the initial conditions vector that correspond to parabolic equations.

PDE Solver Syntax

The basic syntax of the solver is:

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
```

Note Correspondences given are to terms used in "Initial Value Problems" on page 13-2.

The input arguments are

`m`	Specifies the symmetry of the problem. `m` can be `0` = slab, `1` = cylindrical, or `2` = spherical. It corresponds to m in "Equation 13-1".
`pdefun`	Function that defines the components of the PDE. It computes the terms c, f, and s in "Equation 13-1", and has the form `[c,f,s] = pdefun(x,t,u,dudx)` where `x` and `t` are scalars, and `u` and `dudx` are vectors that approximate the solution u and its partial derivative with respect to x. `c`, `f`, and `s` are column vectors. `c` stores the diagonal elements of the matrix c.

icfun   Function that evaluates the initial conditions. It has the form

```
u = icfun(x)
```

When called with an argument x, icfun evaluates and returns the initial values of the solution components at x in the column vector u.

bcfun   Function that evaluates the terms *p* and *q* of the boundary conditions. It has the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

where ul is the approximate solution at the left boundary xl = a and ur is the approximate solution at the right boundary xr = b. pl and ql are column vectors corresponding to *p* and the diagonal of *q* evaluated at xl. Similarly, pr and qr correspond to xr. When $m > 0$ and $a = 0$, boundedness of the solution near $x = 0$ requires that the *f* vanish at $a = 0$. pdepe imposes this boundary condition automatically and it ignores values returned in pl and ql.

xmesh   Vector [x0, x1, ..., xn] specifying the points at which a numerical solution is requested for every value in tspan. x0 and xn correspond to *a* and *b*, respectively.

Second-order approximation to the solution is made on the mesh specified in xmesh. Generally, it is best to use closely spaced mesh points where the solution changes rapidly. pdepe does *not* select the mesh in *x* automatically. You must provide an appropriate fixed mesh in xmesh. The cost depends strongly on the length of xmesh. When $m > 0$, it is not necessary to use a fine mesh near $x = 0$ to account for the coordinate singularity.

The elements of xmesh must satisfy x0 < x1 < ... < xn. The length of xmesh must be ≥ 3.

tspan	Vector [t0, t1, ..., tf] specifying the points at which a solution is requested for every value in xmesh. t0 and tf correspond to t_0 and t_f, respectively.
	pdepe performs the time integration with an ODE solver that selects both the time step and formula dynamically. The solutions at the points specified in tspan are obtained using the natural continuous extension of the integration formulas. The elements of tspan merely specify where you want answers and the cost depends weakly on the length of tspan.
	The elements of tspan must satisfy t0 < t1 < ... < tf. The length of tspan must be ≥ 3.

The output argument sol is a three-dimensional array, such that

- sol(:,:,k) approximates component k of the solution *u*.
- sol(i,:,k) approximates component k of the solution at time tspan(i) and mesh points xmesh(:).
- sol(i,j,k) approximates component k of the solution at time tspan(i) and the mesh point xmesh(j).

PDE Solver Options

For more advanced applications, you can also specify as input arguments solver options and additional parameters that are passed to the PDE functions.

options	Structure of optional parameters that change the default integration properties. This is the seventh input argument.
| | ```
sol = pdepe(m,pdefun,icfun,bcfun,...
            xmesh,tspan,options)
``` |
| | See "Integrator Options" on page 13-6 for more information. |

Integrator 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 defaults. You do this by supplying pdepe with one or more property values in an options structure.

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan,options)
```

Use `odeset` to create the options structure. Only those options of the underlying ODE solver shown in the following table are available for `pdepe`. The defaults obtained by leaving off the input argument `options` are generally satisfactory.

PDE Properties

Properties Category	Property Name
Error control	`RelTol`, `AbsTol`, `NormControl`
Step-size	`InitialStep`, `MaxStep`

Examples

- "Single PDE" on page 13-7
- "System of PDEs" on page 13-13
- "Additional Examples" on page 13-17

Single PDE

- "Solving the Equation" on page 13-7
- "Evaluating the Solution" on page 13-12

Solving the Equation

This example illustrates the straightforward formulation, solution, and plotting of the solution of a single PDE

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}.$$

This equation holds on an interval $0 \leq x \leq 1$ for times $t \geq 0$. At $t = 0$, the solution satisfies the initial condition

$$u(x,0) = \sin \pi x.$$

At $x = 0$ and $x = 1$, the solution satisfies the boundary conditions

$$u(0,t) = 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1,t) = 0.$$

Note The file, `pdex1.m`, contains the complete code for this example. It contains all the required functions coded as local functions. To see the code in an editor, type `edit pdex1` at the command line. To run it, type `pdex1` at the command line. See "PDE Solver Syntax" on page 13-4 for more information.

---

1   **Rewrite the PDE.** Write the PDE in the form

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right).$$

This is the form shown in "Equation 13-1" and expected by `pdepe`. See "Initial Value Problems" on page 13-2 for more information. For this example, the resulting equation is

$$\pi^2\frac{\partial u}{\partial t} = x^0\frac{\partial}{\partial x}\left(x^0\frac{\partial u}{\partial x}\right) + 0$$

with parameter $m = 0$ and the terms

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right) = \pi^2$$

$$f\left(x,t,u,\frac{\partial u}{\partial x}\right) = \frac{\partial u}{\partial x}$$

$$s\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0.$$

2   **Code the PDE.** Once you rewrite the PDE in the form shown above ("Equation 13-1") and identify the terms, you can code the PDE in a function that `pdepe` can use. The function must be of the form

```
[c,f,s] = pdefun(x,t,u,dudx)
```

where c, f, and s correspond to the *c*, *f*, and *s* terms. The code below computes c, f, and s for the example problem.

```
function [c,f,s] = pdex1pde(x,t,u,DuDx)
c = pi^2;
f = DuDx;
s = 0;
```

3 **Code the initial conditions function.** You must code the initial conditions in a function of the form

```
u = icfun(x)
```

The code below represents the initial conditions in the function pdex1ic.

```
function u0 = pdex1ic(x)
u0 = sin(pi*x);
```

4 **Code the boundary conditions function.** You must also code the boundary conditions in a function of the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

The boundary conditions, written in the same form as "Equation 13-3", are

$$u(0,t)+0\cdot\frac{\partial u}{\partial x}(0,t)=0 \ \ \text{at} \ \ x=0$$

and

$$\pi e^{-t}+1\cdot\frac{\partial u}{\partial x}(1,t)=0 \ \ \text{at} \ \ x=1.$$

The code below evaluates the components $p(x,t,u)$ and $q(x,t)$ of the boundary conditions in the function pdex1bc.

```
function [pl,ql,pr,qr] = pdex1bc(xl,ul,xr,ur,t)
pl = ul;
ql = 0;
pr = pi * exp(-t);
qr = 1;
```

In the function pdex1bc, pl and ql correspond to the left boundary conditions ($x = 0$), and pr and qr correspond to the right boundary condition ($x = 1$).

5 **Select mesh points for the solution.** Before you use the MATLAB PDE solver, 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 (see "PDE Solver" on page 13-3). In particular, the cost and the 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`.

This example requests the solution on the mesh produced by 20 equally spaced points from 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);
```

6 **Apply the PDE solver.** The example calls `pdepe` with `m = 0`, the functions `pdex1pde`, `pdex1ic`, and `pdex1bc`, and the mesh defined by `x` and `t` at which `pdepe` is to evaluate the solution. The `pdepe` function returns the numerical solution in a three-dimensional array `sol`, where `sol(i,j,k)` approximates the `k`th component of the solution, u_k, evaluated at `t(i)` and `x(j)`.

```
m = 0;
sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
```

This example uses `@` to pass `pdex1pde`, `pdex1ic`, and `pdex1bc` as function handles to `pdepe`.

Note See "Function Handles" for more information about function handles.

7 **View the results.** Complete the example by displaying the results:

a Extract and display the first solution component. In this example, the solution u has only one component, but for illustrative purposes, the example "extracts" it from the three-dimensional array. The surface plot shows the behavior of the solution.

```
u = sol(:,:,1);

surf(x,t,u)
title('Numerical solution computed with 20 mesh points')
xlabel('Distance x')
ylabel('Time t')
```

Numerical solution computed with 20 mesh points.

b   Display a solution profile at t_f, the final value of t. In this example, $t_f = t = 2$.

```
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')
```

13-11

Evaluating the Solution

After obtaining and plotting the solution above, you might be interested in a solution profile for a particular value of t, or the time changes of the solution at a particular point x. The kth column u(:,k) (of the solution extracted in step 7) contains the time history of the solution at x(k). The jth row u(j,:) contains the solution profile at t(j).

Using the vectors x and u(j,:), and the helper function pdeval, you can evaluate the solution u and its derivative $\partial u/\partial x$ at any set of points xout

```
[uout,DuoutDx] = pdeval(m,x,u(j,:),xout)
```

The example pdex3 uses pdeval to evaluate the derivative of the solution at xout = 0. See pdeval for details.

System of PDEs

This example illustrates the solution of a system of partial differential equations. The problem is taken from electrodynamics. It has boundary layers at both ends of the interval, and the solution changes rapidly for small t.

The PDEs are

$$\frac{\partial u_1}{\partial t} = 0.024 \frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)$$

$$\frac{\partial u_2}{\partial t} = 0.170 \frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$. The equations hold on an interval $0 \le x \le 1$ for times $t \ge 0$.

The solution u satisfies the initial conditions

$$u_1(x, 0) \equiv 1$$
$$u_2(x, 0) \equiv 0$$

and boundary conditions

$$\frac{\partial u_1}{\partial x}(0, t) \equiv 0$$
$$u_2(0, t) \equiv 0$$
$$u_1(1, t) \equiv 1$$
$$\frac{\partial u_2}{\partial x}(1, t) \equiv 0.$$

Note The file, `pdex4.m`, contains the complete code for this example. It contains all the required functions coded as local functions. To see the code in an editor, type `edit pdex4` at the command line. To run it, type `pdex4` at the command line.

1 Rewrite the PDE. In the form expected by `pdepe`, the equations are

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} .* \frac{\partial}{\partial t} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \frac{\partial}{\partial x} \begin{bmatrix} 0.024\,(\partial u_1 \,/\, \partial x) \\ 0.170\,(\partial u_2 \,/\, \partial x) \end{bmatrix} + \begin{bmatrix} -F\,(u_1 - u_2) \\ F\,(u_1 - u_2) \end{bmatrix}.$$

The boundary conditions on the partial derivatives of *u* have to be written in terms of the flux. In the form expected by `pdepe`, the left boundary condition is

$$\begin{bmatrix} 0 \\ u_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} .* \begin{bmatrix} 0.024\,(\partial u_1 \,/\, \partial x) \\ 0.170\,(\partial u_2 \,/\, \partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

and the right boundary condition is

$$\begin{bmatrix} u_1 - 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} .* \begin{bmatrix} 0.024\,(\partial u_1 \,/\, \partial x) \\ 0.170\,(\partial u_2 \,/\, \partial x) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

2   **Code the PDE.** After you rewrite the PDE in the form shown above, you can code it as a function that `pdepe` can use. The function must be of the form

```
[c,f,s] = pdefun(x,t,u,dudx)
```

where `c`, `f`, and `s` correspond to the *c*, *f*, and *s* terms in "Equation 13-1".

```
function [c,f,s] = pdex4pde(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];
```

3   **Code the initial conditions function.** The initial conditions function must be of the form

```
u = icfun(x)
```

The code below represents the initial conditions in the function `pdex4ic`.

```
function u0 = pdex4ic(x);
u0 = [1; 0];
```

4   **Code the boundary conditions function.** The boundary conditions functions must be of the form

```
[pl,ql,pr,qr] = bcfun(xl,ul,xr,ur,t)
```

The code below evaluates the components $p(x,t,u)$ and $q(x,t)$ ("Equation 13-3") of the boundary conditions in the function `pdex4bc`.

```
function [pl,ql,pr,qr] = pdex4bc(xl,ul,xr,ur,t)
pl = [0; ul(2)];
ql = [1; 0];
pr = [ur(1)-1; 0];
qr = [0; 1];
```

5 **Select mesh points for the solution.** The solution changes rapidly for small t. The program selects the step size in time to resolve this sharp change, but to see this behavior in the plots, output times must be selected accordingly. There are boundary layers in the solution at both ends of [0,1], so mesh points must be placed there to resolve these sharp changes. Often some experimentation is needed to select the mesh that reveals the behavior of the solution.

```
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];
```

6 **Apply the PDE solver.** The example calls `pdepe` with `m = 0`, the functions `pdex4pde`, `pdex4ic`, and `pdex4bc`, and the mesh defined by x and t at which `pdepe` is to evaluate the solution. The `pdepe` function returns the numerical solution in a three-dimensional array `sol`, where `sol(i,j,k)` approximates the kth component of the solution, μ_k, evaluated at `t(i)` and `x(j)`.

```
m = 0;
sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
```

7 **View the results.** The surface plots show the behavior of the solution components.

```
u1 = sol(:,:,1);
u2 = sol(:,:,2);

figure
surf(x,t,u1)
title('u1(x,t)')
xlabel('Distance x')
ylabel('Time t')
```

```
figure
surf(x,t,u2)
title('u2(x,t)')
xlabel('Distance x')
ylabel('Time t')
```

u2(x,t)

Additional Examples

The following additional examples are available. Type

edit *examplename*

to view the code and

examplename

to run the example.

Example Name	Description
pdex1	Simple PDE that illustrates the straightforward formulation, computation, and plotting of the solution
pdex2	Problem that involves discontinuities
pdex3	Problem that requires computing values of the partial derivative
pdex4	System of two PDEs whose solution has boundary layers at both ends of the interval and changes rapidly for small t.
pdex5	System of PDEs with step functions as initial conditions

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 | bvp5c | ode45 | pdepe

Delay Differential Equations (DDEs)

Types of DDEs

Constant Delay DDEs

A system of differential equations (DDEs) with constant delays has the following form:

$$y'(t) = f(t, y(t), y(t - \tau_1), \ldots, y(t - \tau_k)).$$

(14-1)

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 of the form given by "Equation 14-1" 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

"Equation 14-1" is a special case of

$$y'(t) = f(t, y(t), y(dy_1), \ldots, y(dy_p))$$

(14-2)

that involves delays, $dy_1,\ldots, dy_k$, which can depend on both time, t, and state, y. The delays, $dy_j(t, y)$, must satisfy $dy_j(t, y) \le t$ on the interval $[t_0, t_f]$ with $t_0 < t_f$.

The `ddesd` function finds the solution, $y(t)$, for DDEs of the form given by "Equation 14-2" 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:

$$y'(t) = f(t, y(t), y(dy_1), \ldots, y(dy_p), y'(dyp_1), \ldots, y'(dyp_q)).$$

(14-3)

The delays in the solution must satisfy $dy_i(t,y) \le t$. 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 by "Equation 14-2". 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, "Equation 14-1" shows that $y'(t_0)$ 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 $t < t_0$.

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, as in "Equation 14-1". If the delays are not constant, the propagation of discontinuities is more complicated. For neutral DDEs of the form given by "Equation 14-1" or "Equation 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 by "Equation 14-3" 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 $[t_0, t_f]$.

References

[1] Shampine, L.F. "Dissipative Approximations to Neutral DDEs." *Applied Mathematics & Computation*, Vol. 203, 2008, pp. 641–648.

See Also

dde23 | ddensd | ddesd

More About

*   "Discontinuities in DDEs" on page 14-5

Discontinuities in DDEs

If your problem has discontinuities, it's 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`.
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.

See Also

ddeget | ddeset

More About

- "Types of DDEs" on page 14-2

DDE with Constant Delays

This example shows how to use `dde23` to solve a system of DDEs with constant delays.

Click `ddex1.m` or type `edit ddex1.m` in a command window to view the code for this example in an editor.

The differential equations are:

$$y_1'(t) = y_1(t-1)$$
$$y_2'(t) = y_1(t-1) + y_2(t-0.2)$$
$$y_3'(t) = y_2(t).$$

The history of this problem is constant:

$$y_1(t) = 1$$
$$y_2(t) = 1$$
$$y_3(t) = 1$$

for $t \leq 0$.

1   Create a new program file in the editor. This file will contain a main function and two local functions.

2   Define the first-order DDE as a local function.

```
function dydt = ddex1de(t,y,Z)
  ylag1 = Z(:,1);
  ylag2 = Z(:,2);
  dydt = [ylag1(1); ylag1(1)+ylag2(2); y(2)];
end
```

3   Define the solution history as a local function.

```
function S = ddex1hist(t)
  S = ones(3,1);
end
```

4   Define the delays, $\tau_1, \ldots, \tau_k$ in the main function.

```
lags = [1,0.2];
```

5   Solve the DDE by calling `dde23` in the main function. Pass the DDE function, the delays, the solution history, and interval of integration, `[0,5]`, as inputs.

```
sol = dde23(@ddex1de,lags,@ddex1hist,[0,5]);
```

The `dde23` function produces a continuous solution over the whole interval of integration $[t_0, t_f]$.

6   Plot the solution returned by `dde23`. Add this code to your main function.

```
plot(sol.x,sol.y);
title('An example of Wille and Baker');
xlabel('time t');
ylabel('solution y');
legend('y_1','y_2','y_3','Location','NorthWest');
```

7   Evaluate the solution at 10 equally spaced points over the interval of integration. Then plot the results on the same axes as `sol.y`. Add this code to the main function.

```
tint = linspace(0,5,10);
Sint = deval(sol,tint)
hold on
plot(tint,Sint,'o');
```

8   Run your program to generate and plot the results.

An example of Wille and Baker

See Also

Related Examples

- "Create Functions in Files"
- "Run Functions in the Editor"

More About

- "Create Function Handle"

State-Dependent Delay Problem

This example shows how to use `ddesd` to solve a system of two DDEs with a state-dependent delay. This system of DDEs was used as a test problem by Enright and Hayashi [1].

Click `ddex3.m` or type `edit ddex3.m` in a command window to view the complete code for this example in an editor.

The equations for this system are:

$$y_1'(t) = y_2(t)$$
$$y_2'(t) = -y_2(e^{(1-y_2(t))}) \cdot y_2(t)^2 \cdot e^{(1-y_2(t))}.$$

The analytical solution

$$y_1(t) = \log(t)$$
$$y_2(t) = 1/t$$

is used as the history for $t \leq 0.1$ and the equations are solved on `[0.1, 5]` with `ddesd` rather than `dde23`. The `ddesd` function is appropriate in this case because the first factor in the second equation has the form $y_2(d(y))$ with a delay that depends on the second component of the solution.

1 Create a new program file in the editor. This file will contain a main function and three local functions.

2 Code the system of DDEs as a local function.

```
function dydt = ddex3de(t,y,Z)
  dydt = [y(2); -Z(2)*y(2)^2*exp(1 - y(2))];
end
```

3 Define the delay as a local function.

```
function d = ddex3delay(t,y)
  d = exp(1 - y(2));
end
```

4 Define the solution history as a local function.

```
function v = ddex3hist(t)
  v = [log(t); 1./t];
end
```

5 Define the interval of integration and solve the system. Add this code to the main function in your program file.

```
tspan = [0.1 5];
sol = ddesd(@ddex3de,@ddex3delay,@ddex3hist,tspan);
```

6 Use the history function to calculate the analytical solution within the integration interval. Add this code to the main function.

```
texact = linspace(0.1,5);
yexact = ddex3hist(texact);
```

7 Plot the numerical solution on the same axes as the analytical solution. Add this code to the main function.

```
figure
plot(texact,yexact,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')
```

8 Run your program to generate and plot the results.

D1 problem of Enright and Hayashi

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

Related Examples

- "Create Functions in Files"
- "Run Functions in the Editor"

More About

- "Create Function Handle"

Cardiovascular Model with Discontinuities

This example shows how to use `dde23` to solve a cardiovascular model that has a discontinuous derivative as presented by Ottesen [1].

Click `ddex2.m` or type `edit ddex2.m` in a command window to view the code for this example in an editor.

This is a problem with 1 delay, constant history, and 3 differential equations with 14 physical parameters. The system is heavily influenced by peripheral pressure, *R*, which decreases exponentially from 1.05 to 0.84, beginning at $t = 600$. As a result, the system has a discontinuity in a low-order derivative at $t = 600$.

1   Create a new program file in the editor. This file will contain a main function and a nested function. The main function accepts no inputs and returns no outputs.

2   Define the physical parameters. Add this code to the main function.

```
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;
```

3   Define the solution history. Add this code to the main function.

```
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];
```

4   Define the delay, `tau`. Add this code to the main function.

```
tau = 4;
```

5 Define the location of discontinuity, which occurs at $t = 600$. Add this code to the main function.

```
options = ddeset('Jumps',600);
```

When your DDE has discontinuities in low-order derivatives, and you know the locations in advance, it is better to use `ddeset` with the `Jumps` property.

6 Solve the DDE over the interval [0, 1000]. Add this code to the main function.

```
sol = dde23(@ddex2de,tau,history,[0,1000],options);
```

The function, `@ddex2de`, which defines the system of DDEs, is the first input argument. You define this function in step 8.

7 Plot the solution. Add this code to the main function.

```
figure
plot(sol.x,sol.y(3,:))
title('Heart Rate for Baroflex-Feedback Mechanism.')
xlabel('time t')
ylabel('H(t)')
```

8 Define the system of DDEs as a nested function inside the main function.

```
function dydt = ddex2de(t,y,Z)
    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
```

This function is nested so that the main function can access the 14 parameters defined in step 2.

9 Run your program to calculate the solution and display the plot.

Heart Rate for Baroflex-Feedback Mechanism.

References

[1] Ottesen, J. T. "Modelling of the Baroflex-Feedback Mechanism with Time-Delay." *J. Math. Biol.* Vol. 36, Number 1, 1997, pp. 41–63.

See Also

Related Examples

- "Create Functions in Files"

- "Run Functions in the Editor"

More About

- "Create Function Handle"

DDE of Neutral Type

This example shows how to use `ddensd` to solve the neutral DDE presented by Paul [1] for $0 \le t \le \pi$.

Click `ddex4.m` or type `edit ddex4.m` in a command window to view the code for this example in an editor.

The equation is

$$y'(t) = 1 + y(t) - 2y(t/2)^2 - y'(t - \pi)$$

with history:

$$y(t) = \cos(t) \quad \text{for} \quad t \le 0.$$

1   Create a new program file in the editor. This file will contain a main function and four local functions.

2   Define the first-order DDE as a local function.

```
function yp = ddefun(t,y,ydel,ypdel)
    yp = 1 + y - 2*ydel^2 - ypdel;
end
```

3   Define the solution delay as a local function.

```
function dy = dely(t,y)
    dy = t/2;
end
```

4   Define the derivative delay as a local function.

```
function dyp = delyp(t,y)
    dyp = t-pi;
end
```

5   Define the solution history as a local function.

```
function y = history(t)
    y = cos(t);
end
```

6   Define the interval of integration and solve the DDE using the `ddensd` function. Add this code to the main function.

```
tspan = [0 pi];
sol = ddensd(@ddefun,@dely,@delyp,@history,tspan);
```

7 Evaluate the solution at 100 equally spaced points between 0 and π. Add this code to the main function.

```
tn = linspace(0,pi);
yn = deval(sol,tn);
```

8 Plot the results. Add this code to the main function.

```
figure
plot(tn,yn);
xlim([0 pi]);
ylim([-1.2 1.2])
xlabel('time t');
ylabel('solution y');
title('Example of Paul with 1 equation and 2 delay functions')
```

9 Run your program to calculate the solution and display the plot.

Example of Paul with 1 equation and 2 delay functions

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

Related Examples

- "Create Functions in Files"
- "Run Functions in the Editor"

More About

- "Create Function Handle"

Initial Value DDE of Neutral Type

This example shows how to use `ddensd` to solve the initial value DDE presented by Jackiewicz [1] for $0 \leq t \leq 0.1$.

Click `ddex5.m` or type `edit ddex5.m` in a command window to view the code for this example in an editor.

The equation is

$$y'(t) = 2\cos(2t) \quad y(t/2)^{2\cos(t)} + \log(y'(t/2)) - \log(2\cos(t)) - \sin(t).$$

This is an initial value DDE because the delays are zero at t_0. The initial conditions are:

$$y(0) = 1$$

$$y'(0) = s,$$

where s is the solution of:

$$2 + \log(s) - \log(2) = 0.$$

This equation is satisfied by $s_1 = 2$ and $s_2 = 0.4063757399599599$.

1 Create a new program file in the editor. This file will contain a main function and one local function.

2 Define the DDE as a local function.

```
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
```

3 Define the solution delay and derivative delay. Add this line to the main function.

```
delay = @(t,y) t/2;
```

You can use one anonymous function to handle both delays since they are the same in the equation.

4 Define the initial conditions, `y0` and `s1`, and the interval of integration, `tspan`. Add this code to the main function.

```
y0 = 1;
s1 = 2;
tspan = [0 0.1];
```

5   Solve the DDE for $0 \le t \le 0.1$, with initial conditions $y(0) = 1$, and $y\,'(0) = 2$. Add this code to the main function.

```
sol1 = ddensd(@ddefun,delay,delay,{y0,s1},tspan);
```

6   Solve the equation again, this time using $y\,'(0) = 0.4063757399599599$. Add this code to the main function.

```
s2 = 0.4063757399599599;
sol2 = ddensd(@ddefun,delay,delay,{y0,s2},tspan);
```

7   Plot the results. Add this code to the main function.

```
figure
plot(sol1.x,sol1.y,sol2.x,sol2.y);
legend('y''(0) = 2','y''(0) = .40638','Location','NorthWest');
xlabel('time t');
ylabel('solution y');
title('Two solutions of Jackiewicz''s initial-value NDDE');
```

8   Run your program to calculate and plot the solutions for each value of *s*.

Two solutions of Jackiewicz's initial-value NDDE

References

[1] Jackiewicz, Z. "One step Methods of any Order for Neutral Functional Differential Equations." *SIAM J. Numer. Anal.* Vol. 21, Number 3. 1984. pp. 486–511.

See Also

Related Examples
- "Create Functions in Files"

- "Run Functions in the Editor"

More About

- "Create Function Handle"

15

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

$$x(t) = \sin(2t), \qquad y(t) = \cos(t), \qquad z(t) = t,$$

where $t \in [0,3\pi]$.

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 15-7
- "Integration of Numeric Data" on page 15-11

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 15-7
- "Integration of Numeric Data" on page 15-11

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 $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$.

```
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 =
   3.771236166328259 - 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 15-4
- "Integration of Numeric Data" on page 15-11

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,

$$\int (4x^5 - 2x^3 + x + 4)\, dx$$

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 15-7
- "Integration of Numeric Data" on page 15-11

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.

```
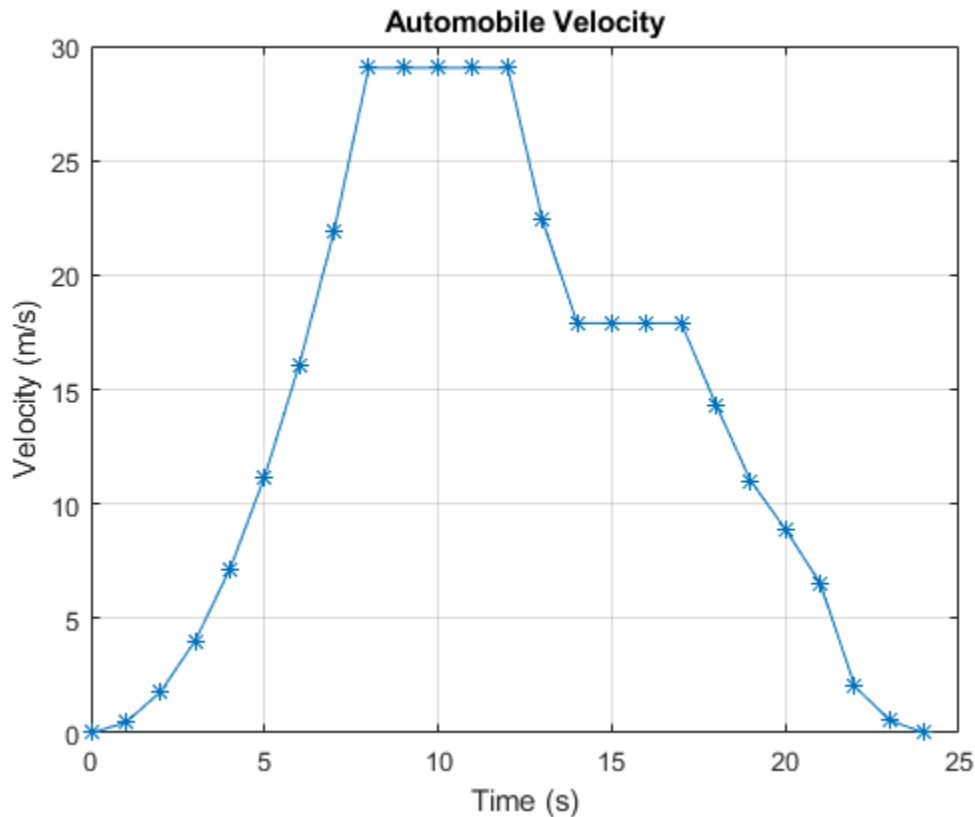vel = [0 .45 1.79 4.02 7.15 11.18 16.09 21.90 29.05 29.05 ...
29.05 29.05 29.05 22.42 17.9 17.9 17.9 17.9 14.34 11.01 ...
8.9 6.54 2.03 0.55 0];
time = 0:24;
```

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')
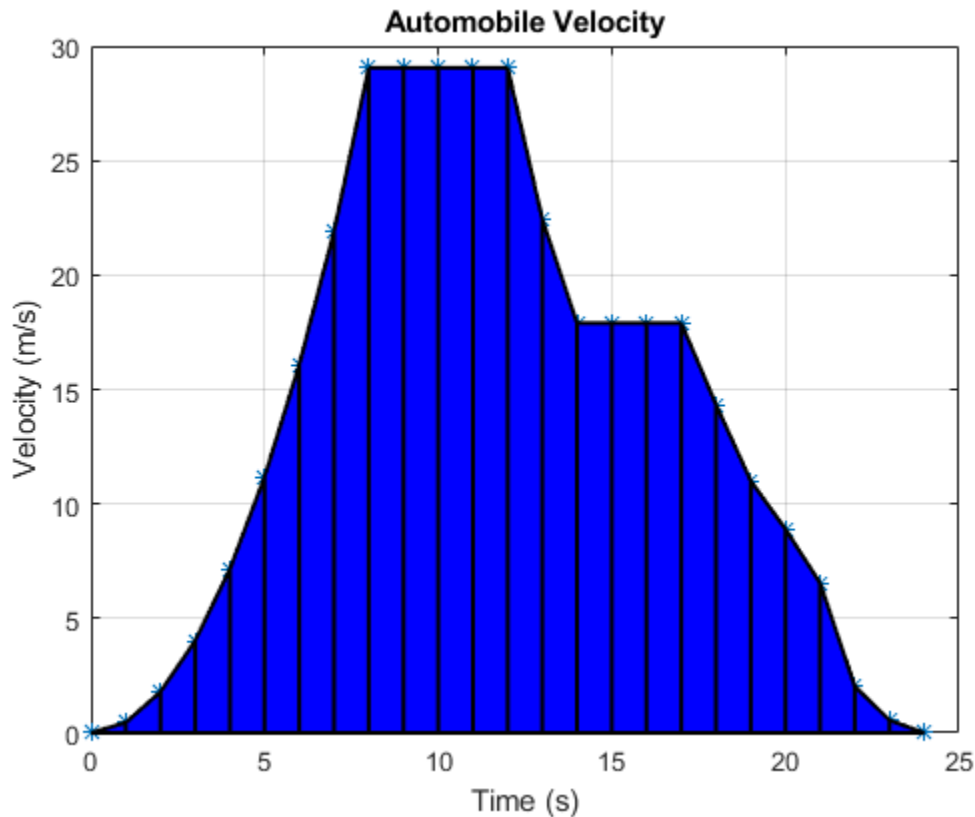xlabel('Time (s)')
ylabel('Velocity (m/s)')
```

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.

```
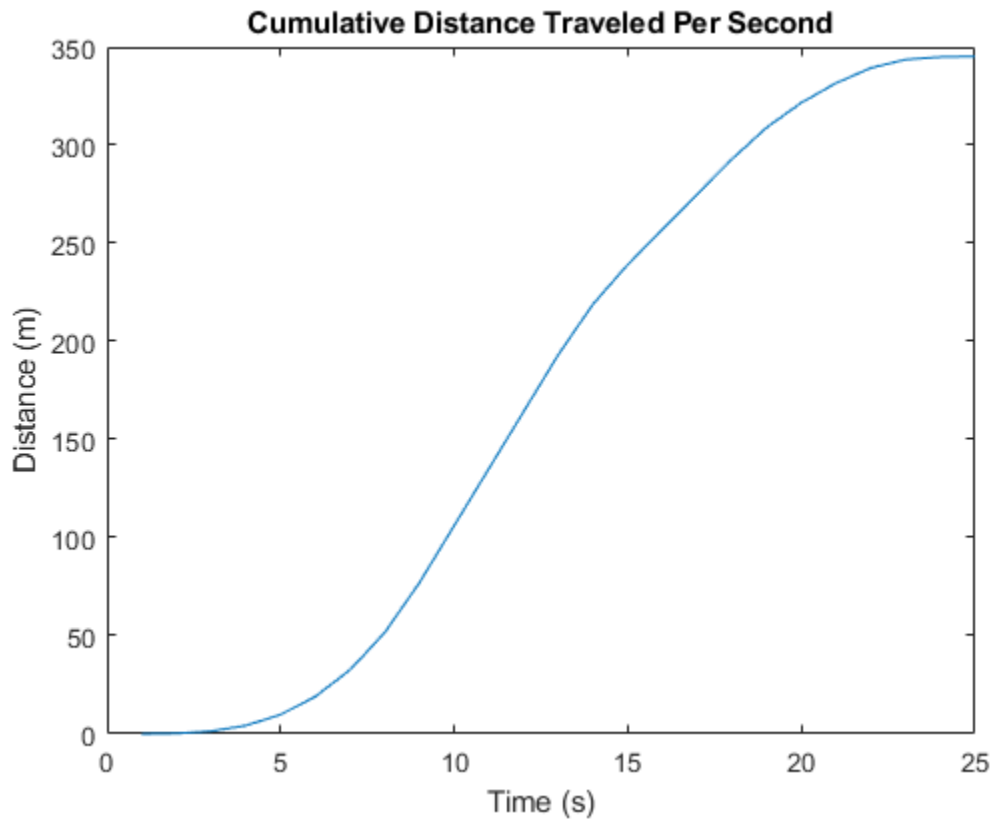cdistance = cumtrapz(vel);
T = table(time',cdistance','VariableNames',{'Time','CumulativeDistance'})
```

```
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 15-7
- "Analytic Solution to Integral of Polynomial" on page 15-9

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.

```
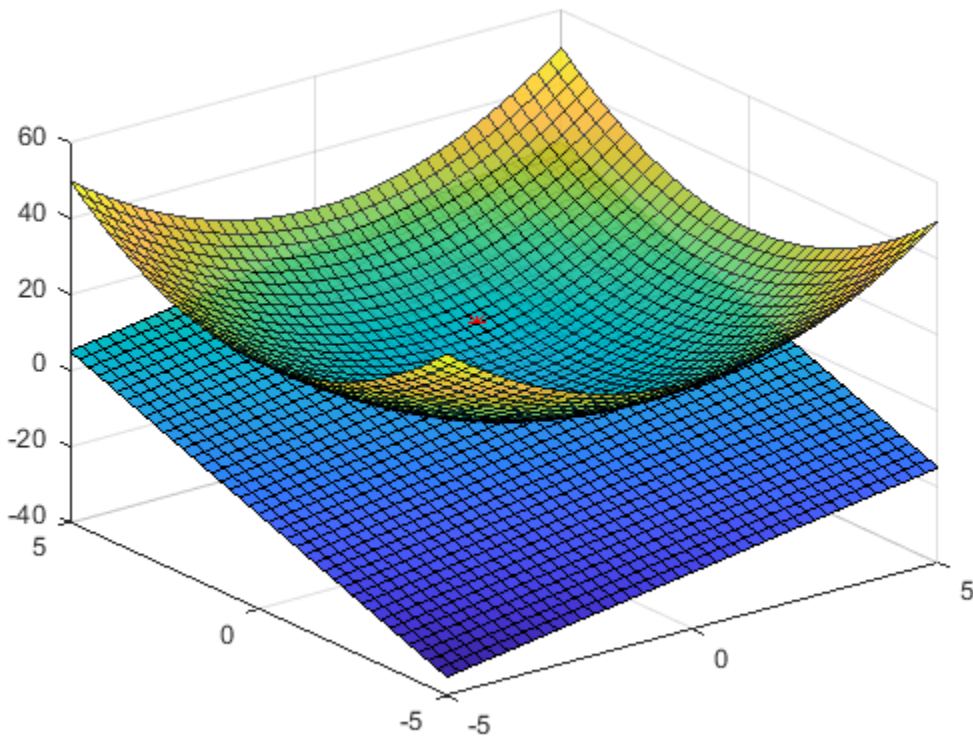x0 = 1;
y0 = 2;
t = (xx == x0) & (yy == y0);
indt = find(t);
fx0 = fx(indt);
fy0 = fy(indt);
```

Create a function handle with the equation of the tangent plane `z`.

```
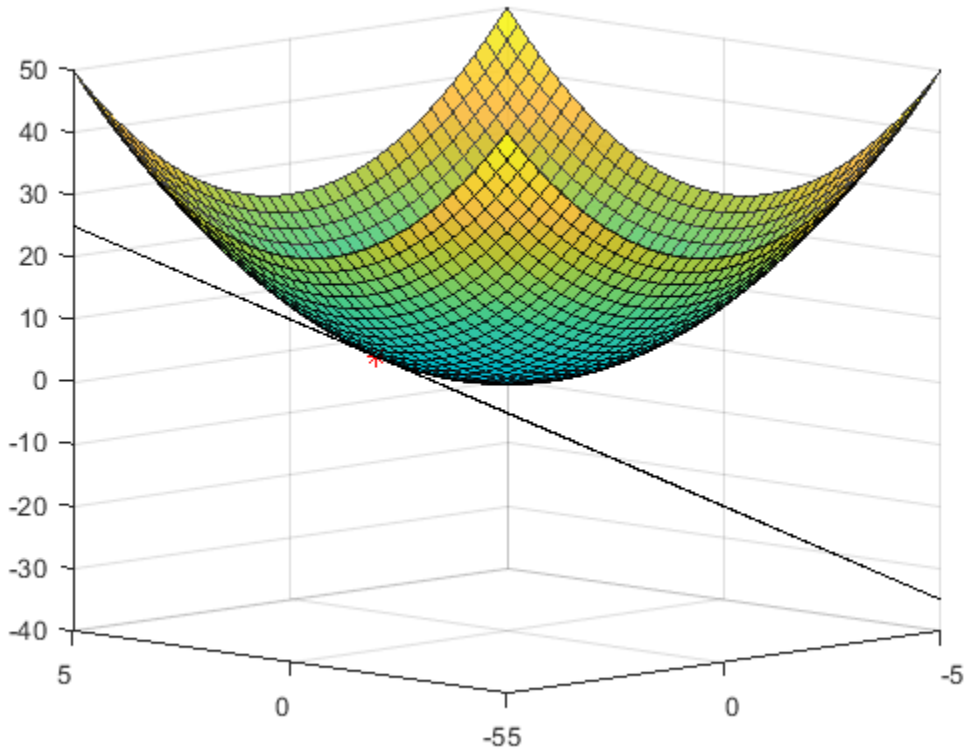z = @(x,y) f(x0,y0) + fx0*(x-x0) + fy0*(y-y0);
```

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"

16

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.

```
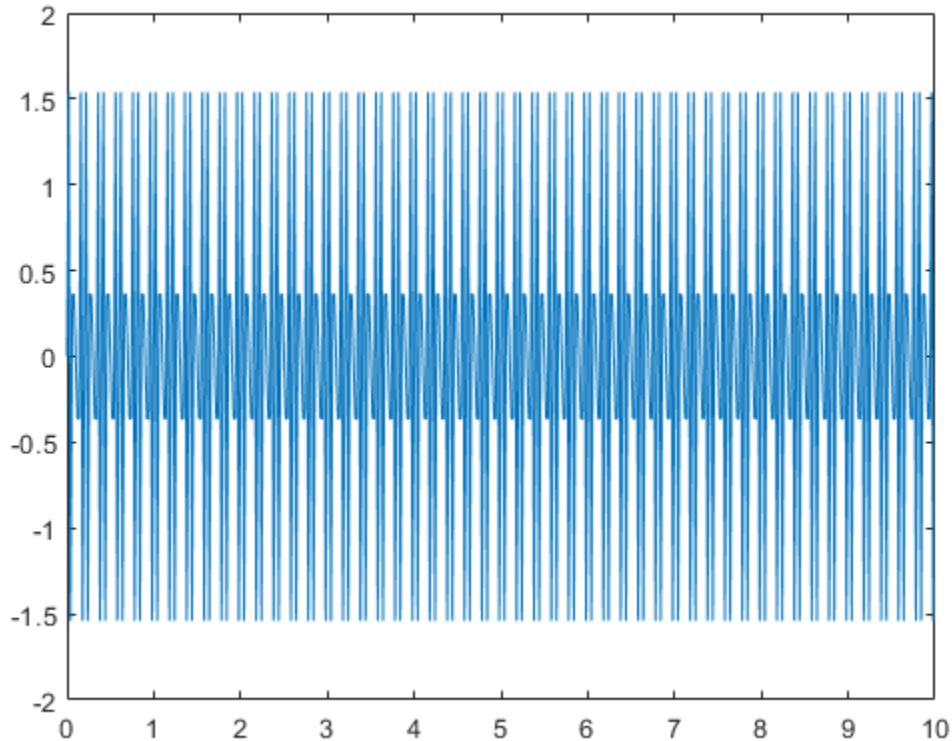t = 0:1/50:10-1/50;
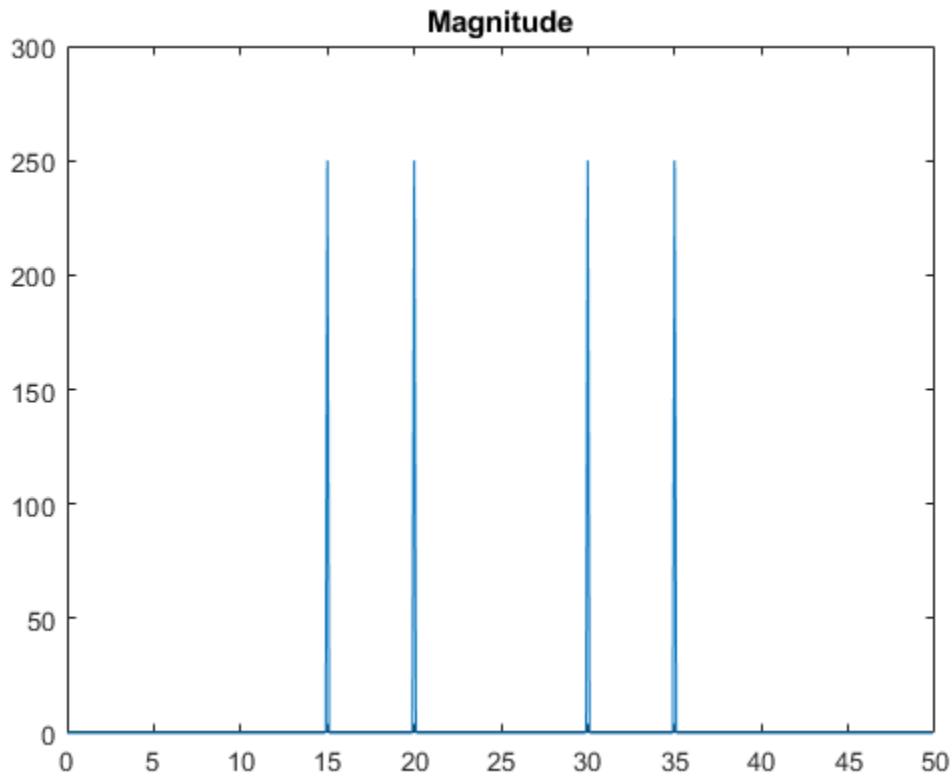x = sin(2*pi*15*t) + sin(2*pi*20*t);
plot(t,x)
```

Compute the Fourier transform of the signal, and create the vector f that corresponds to the signal's sampling in frequency space.

```
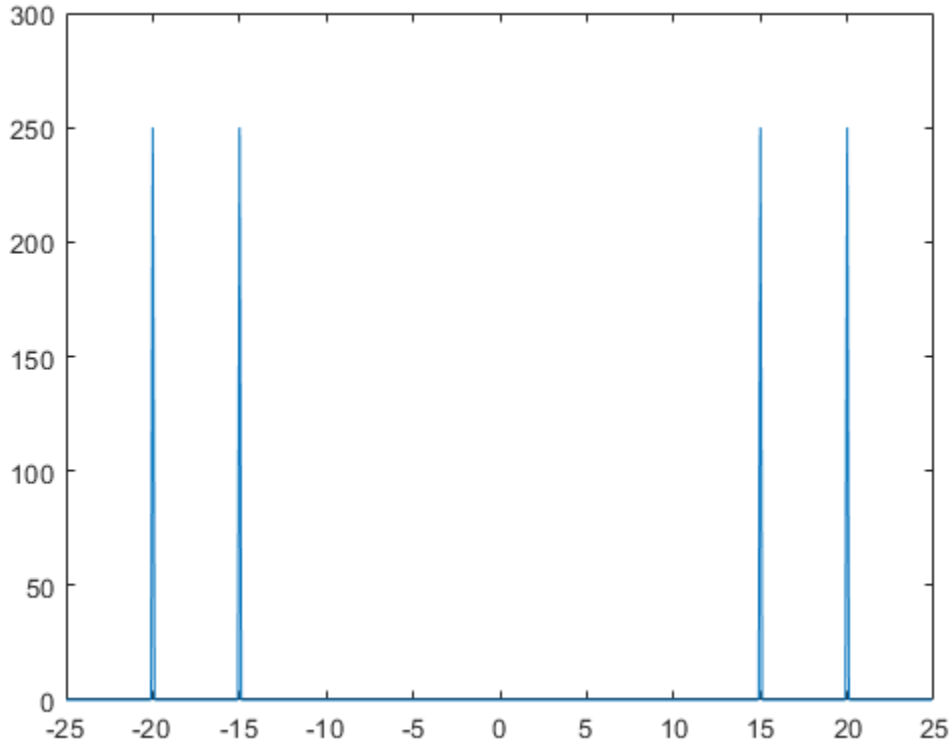y = fft(x);
f = (0:length(y)-1)*50/length(y);
```

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))
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)*(50/n);
yshift = fftshift(y);
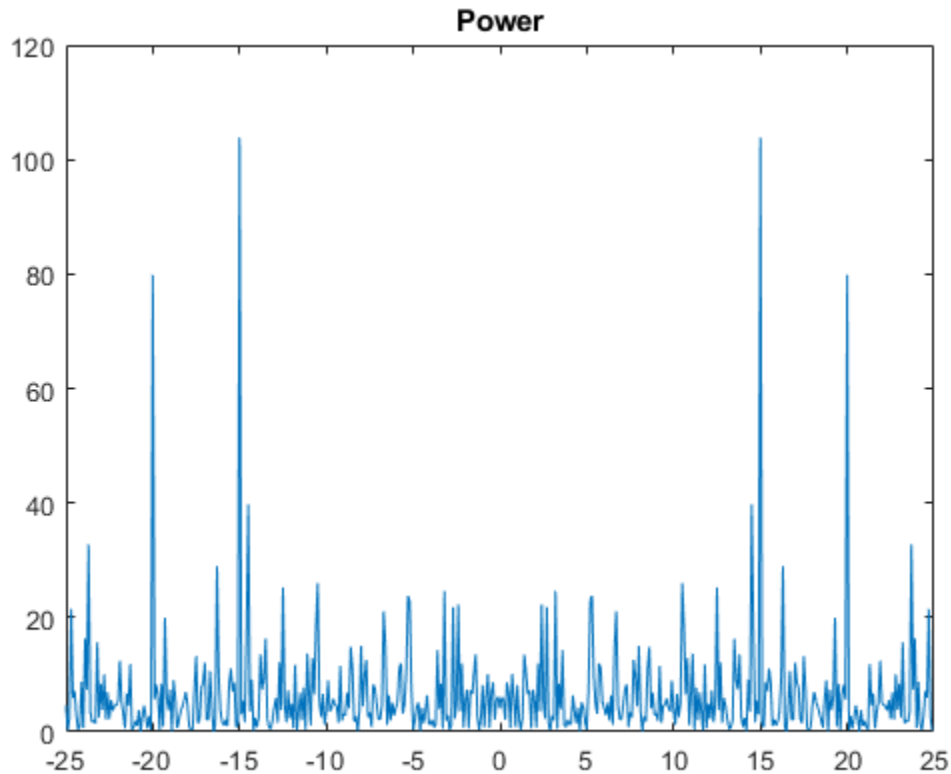plot(fshift,abs(yshift))
```

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`.

```
xnoise = x + 2.5*gallery('normaldata',size(t),4);
```

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;
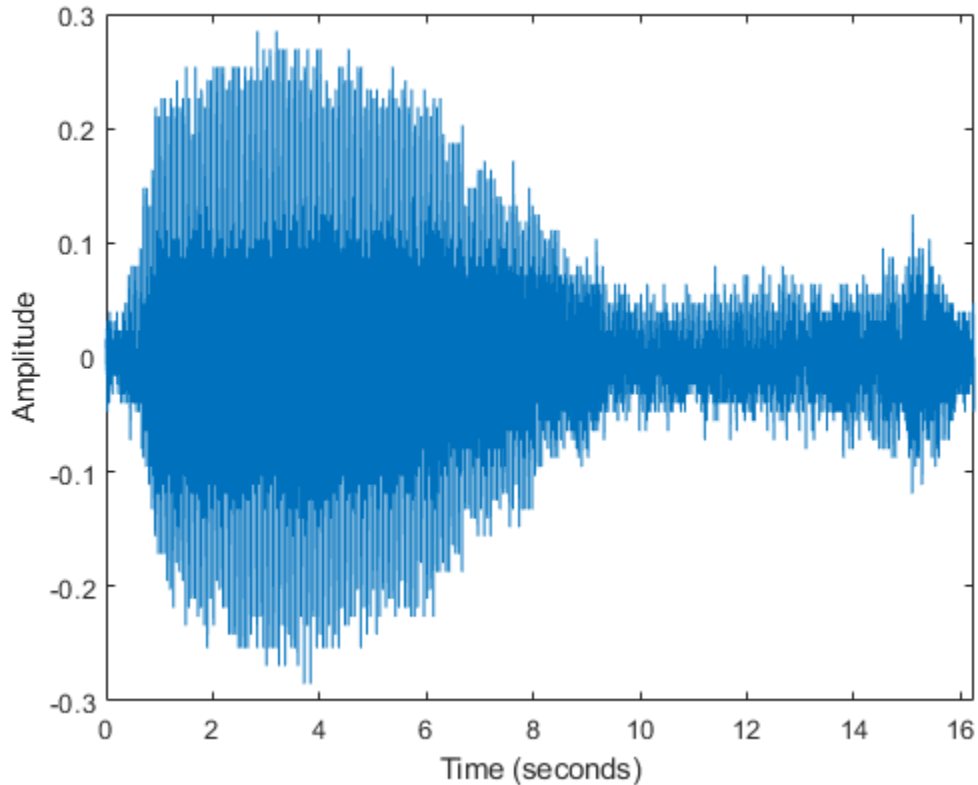plot(fshift,power)
title('Power')
```


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. 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)')
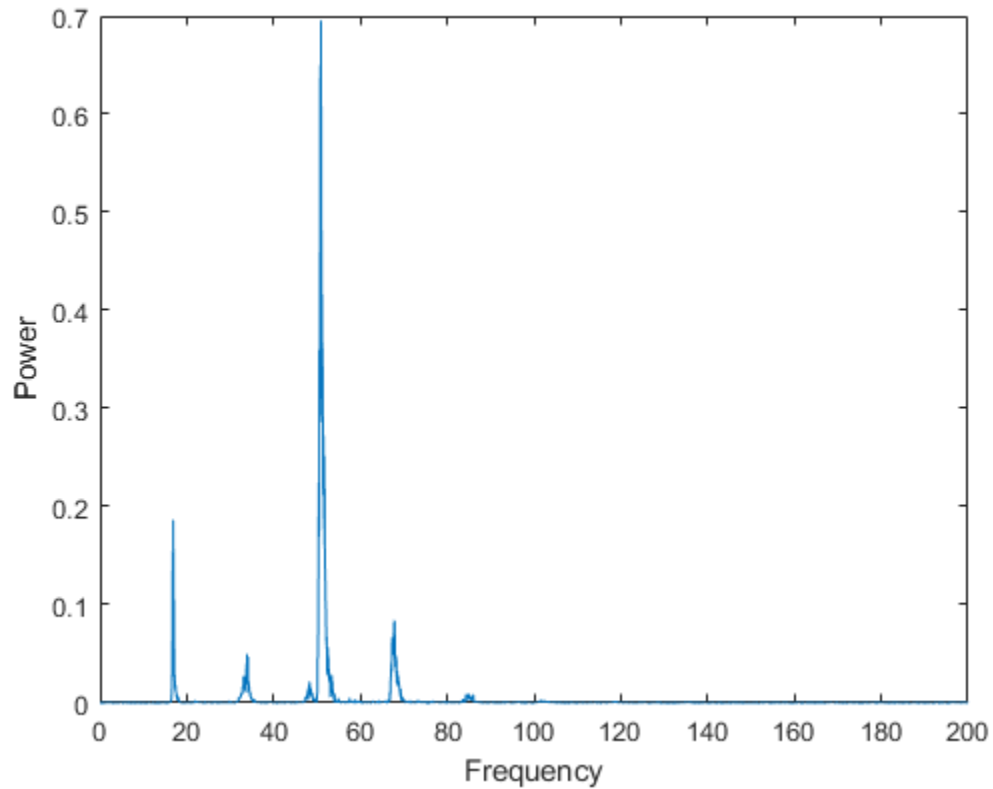ylabel('Amplitude')
xlim([0 t(end)])
```

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')
ylabel('Power')
```


See Also
ifft | fft2 | fftn | fftw | fft | fftshift | nextpow2

Related Examples

- "Analyzing Cyclical Data with FFT"
- "2-D Fourier Transforms" on page 16-24

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.

```
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
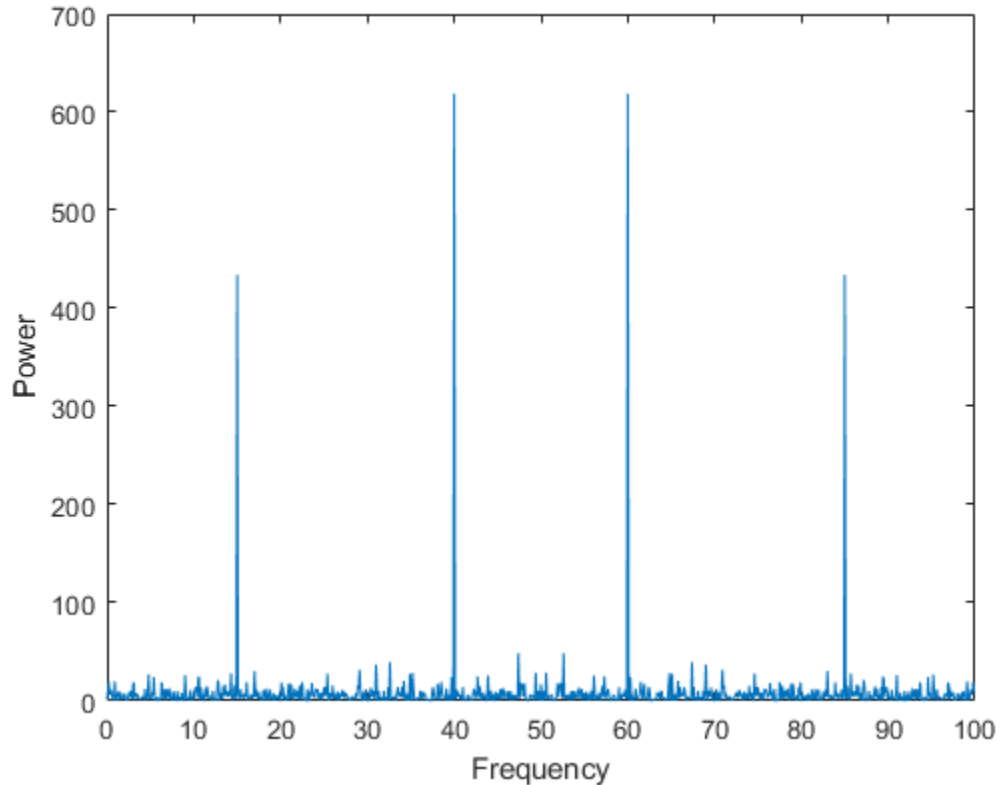  + 2.5*gallery('normaldata',size(t),4);  % Gaussian noise;
```

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
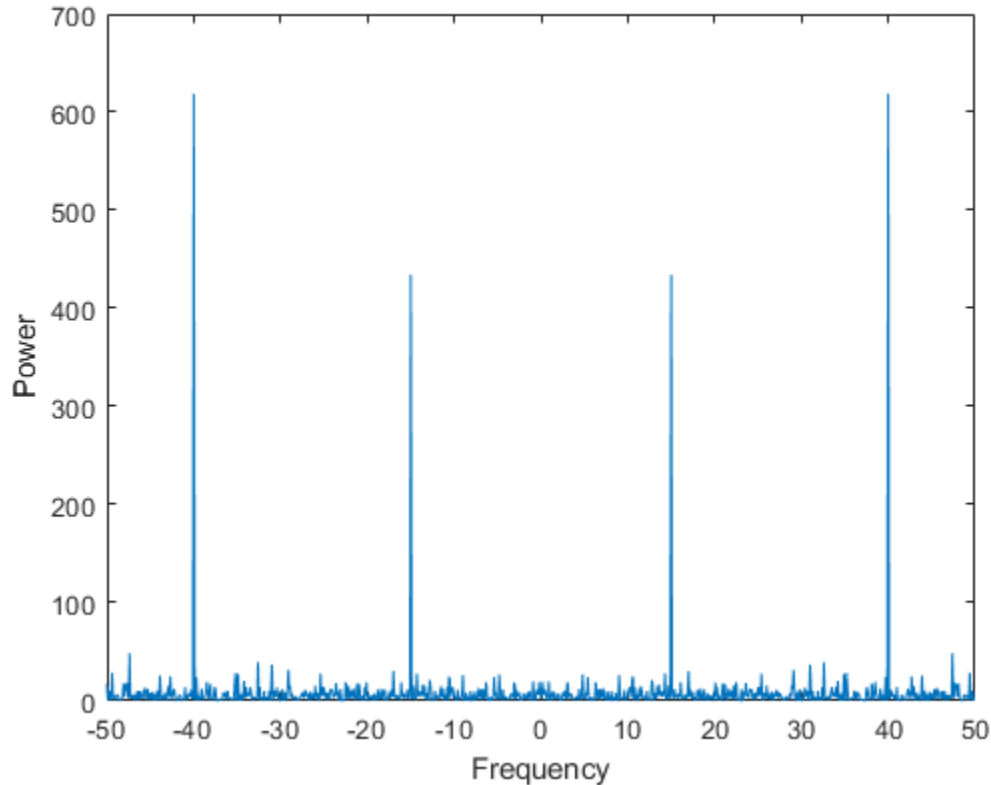power = abs(y).^2/n;    % power of the DFT

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)
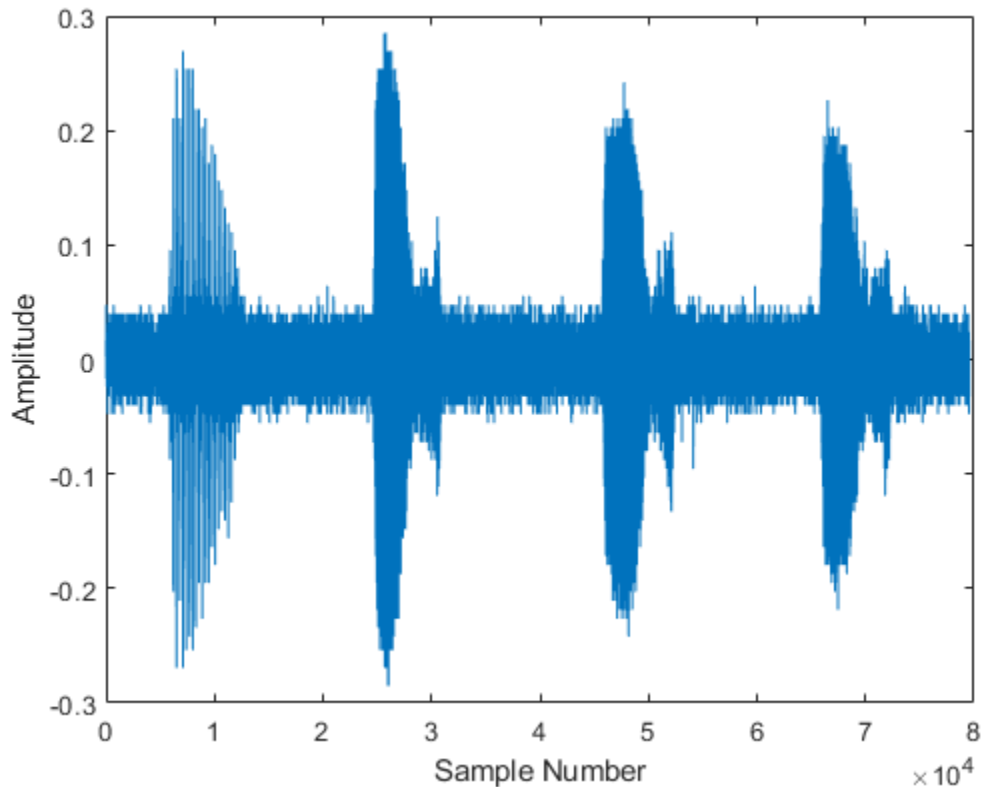xlabel('Frequency')
ylabel('Power')
```

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.

```
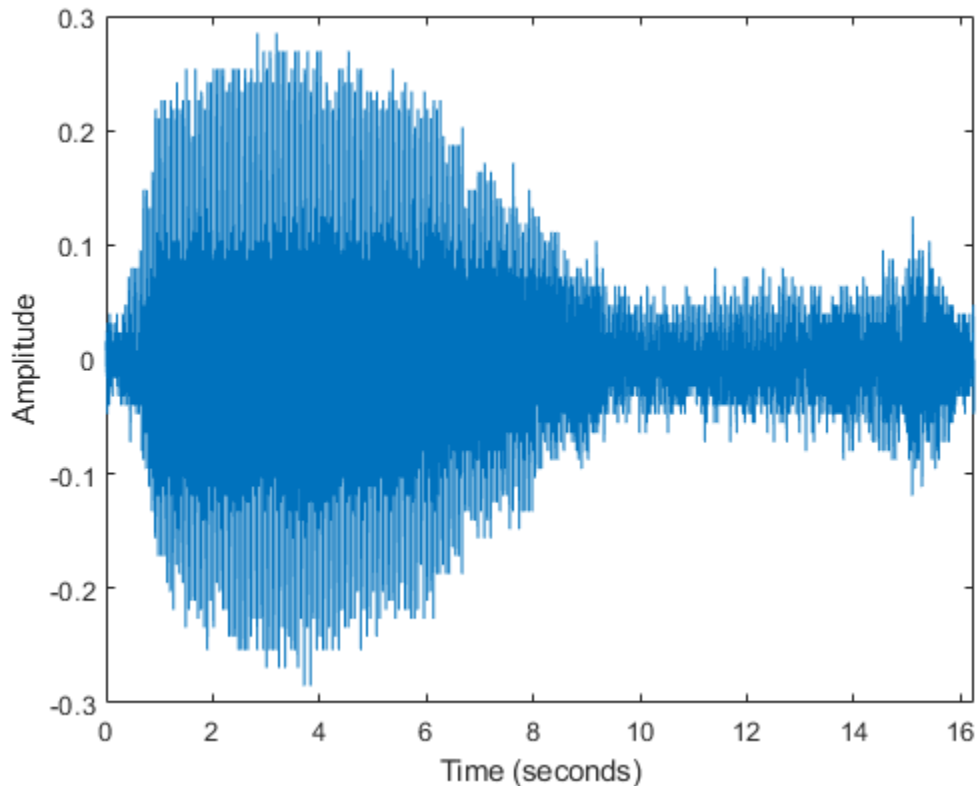whaleFile = 'bluewhale.au';
[x,fs] = audioread(whaleFile);

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;
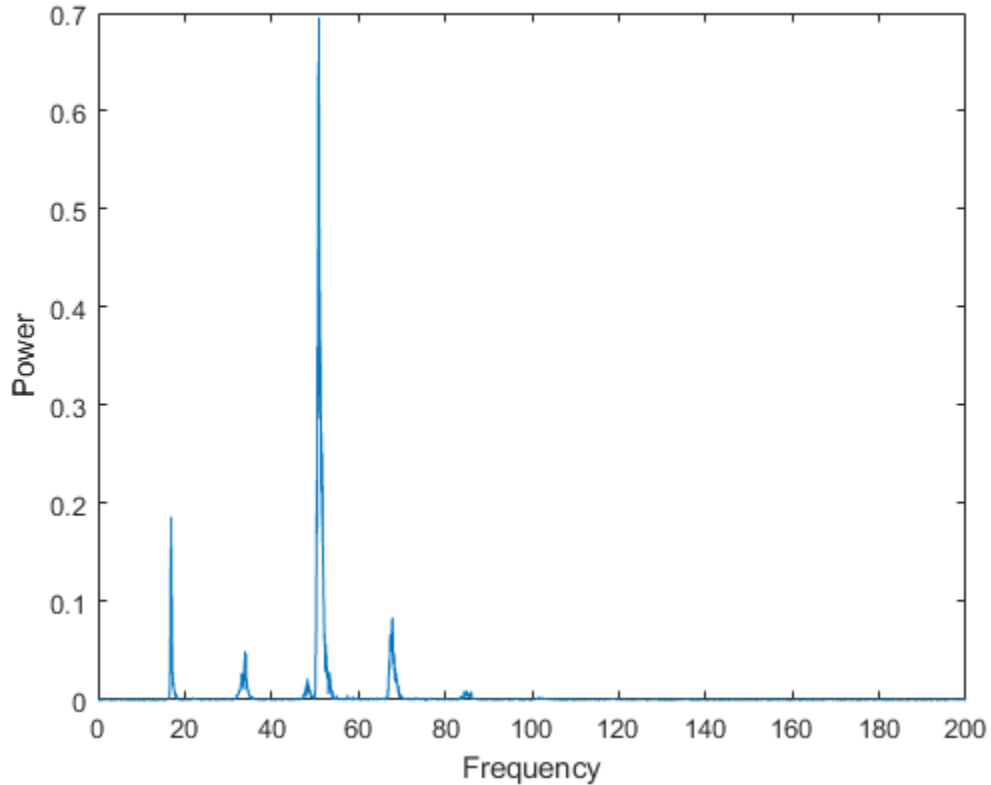
plot(f(1:floor(n/2)),power(1:floor(n/2)))
xlabel('Frequency')
ylabel('Power')
```

See Also
`ifft` | `fft2` | `fftn` | `fft` | `fftshift` | `nextpow2`

Related Examples
- "Fourier Transforms" on page 16-2
- "Analyzing Cyclical Data with FFT"
- "2-D Fourier Transforms" on page 16-24

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 $c_1 x^n + c_2 x^{n-1} + ... + c_n x + c_{n+1}$ 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.

```
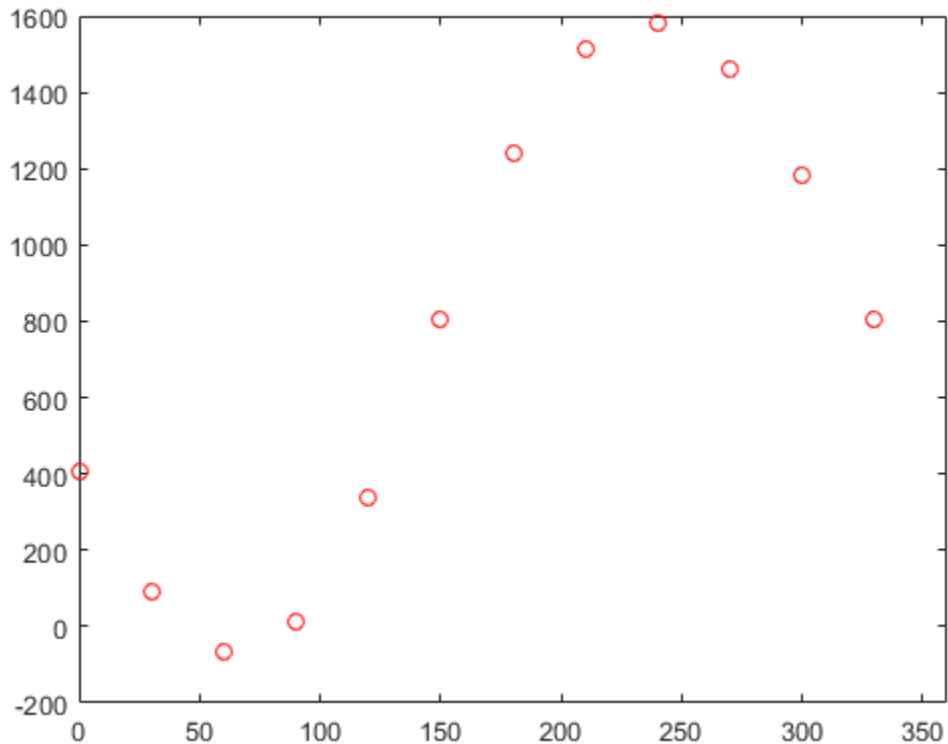x = 0:30:330;
y = [408 89 -66 10 338 807 1238 1511 1583 1462 1183 804];
plot(x,y,'ro')
xlim([0 360])
```

Gauss models the asteroid's orbit with a trigonometric polynomial of the following form.

$$y = a_0 \; + \; a_1 \cos(2\pi(x/360)) + b_1 \sin(2\pi(x/360))$$
$$a_2 \cos(2\pi(2x/360)) + b_2 \sin(2\pi(2x/360))$$
$$\dots$$
$$a_5 \cos(2\pi(5x/360)) + b_5 \sin(2\pi(5x/360))$$
$$a_6 \cos(2\pi(6x/360))$$

Use `fft` to compute the coefficients of the polynomial.

```
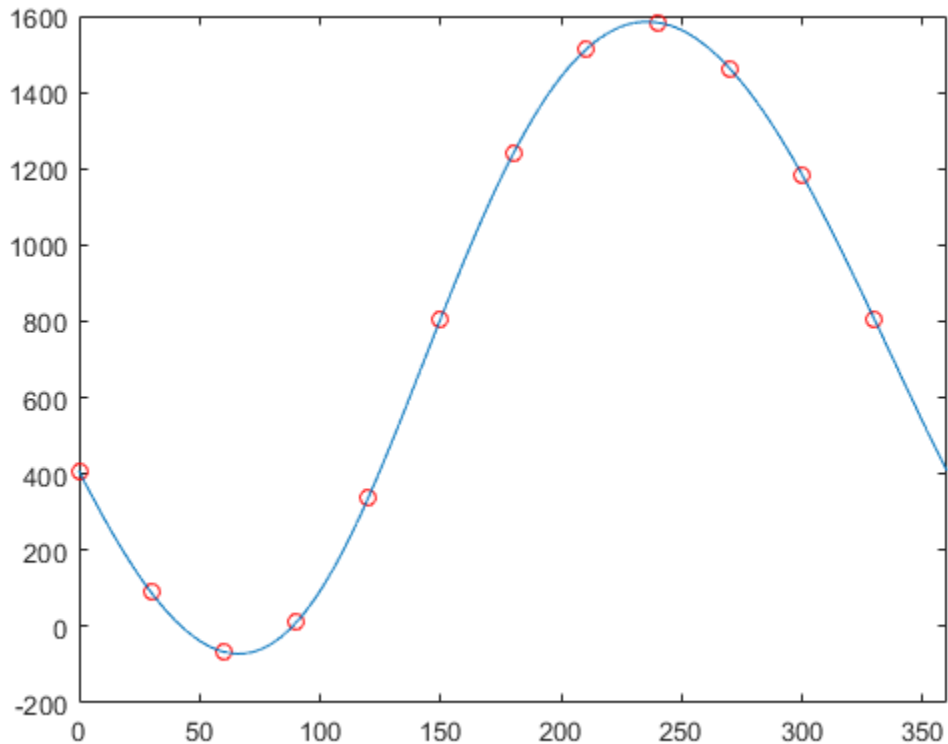m = length(y);
n = floor((m+1)/2);
z = fft(y)/m;

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.

```
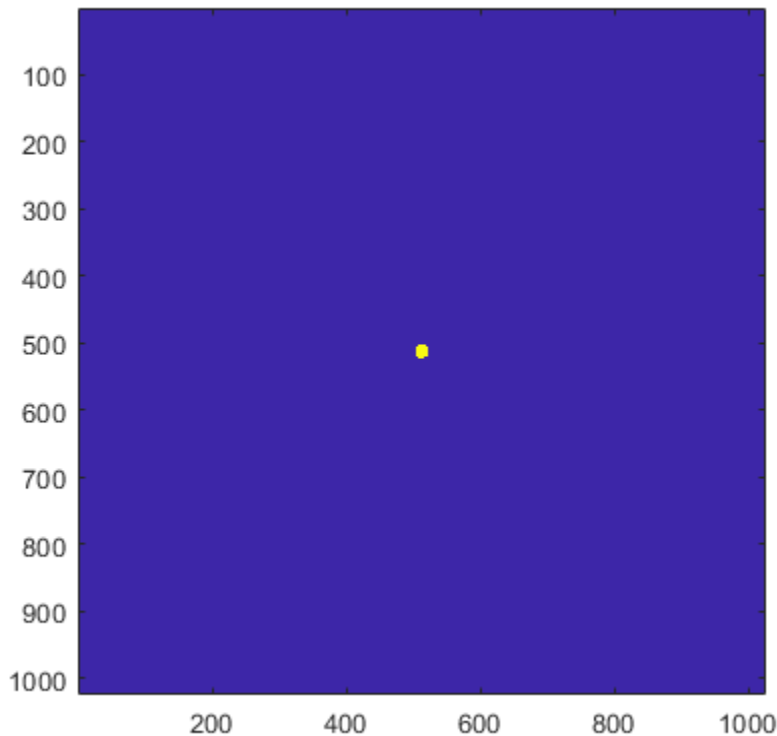n = 2^10;                    % size of mask
M = zeros(n);
I = 1:n;
x = I-n/2;                   % mask x-coordinates
y = n/2-I;                   % mask y-coordinates
```

```
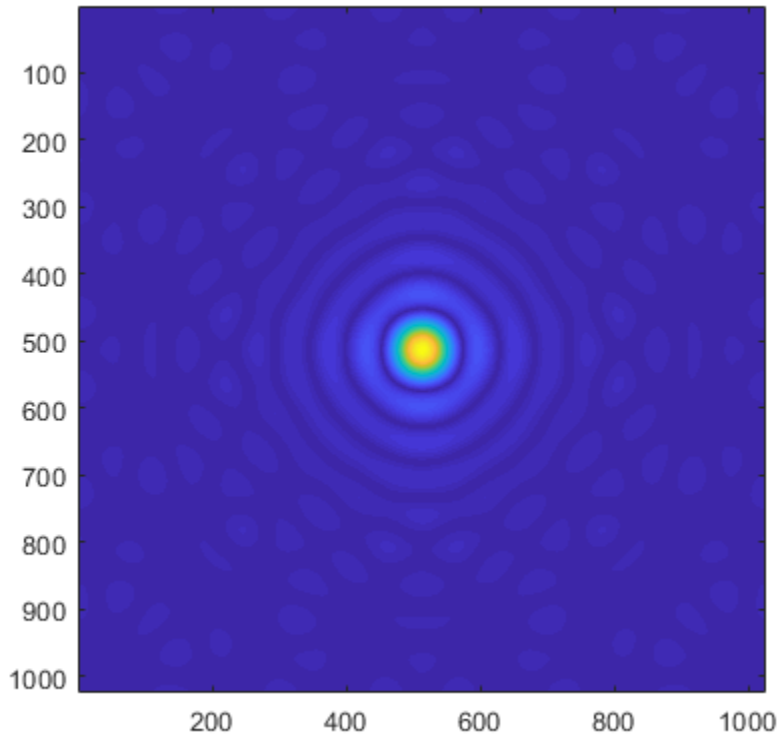[X,Y] = meshgrid(x,y);     % create 2-D mask grid
R = 10;                    % aperture radius
A = (X.^2 + Y.^2 <= R^2);  % circular aperture of radius R
M(A) = 1;                  % set mask elements inside aperture to 1
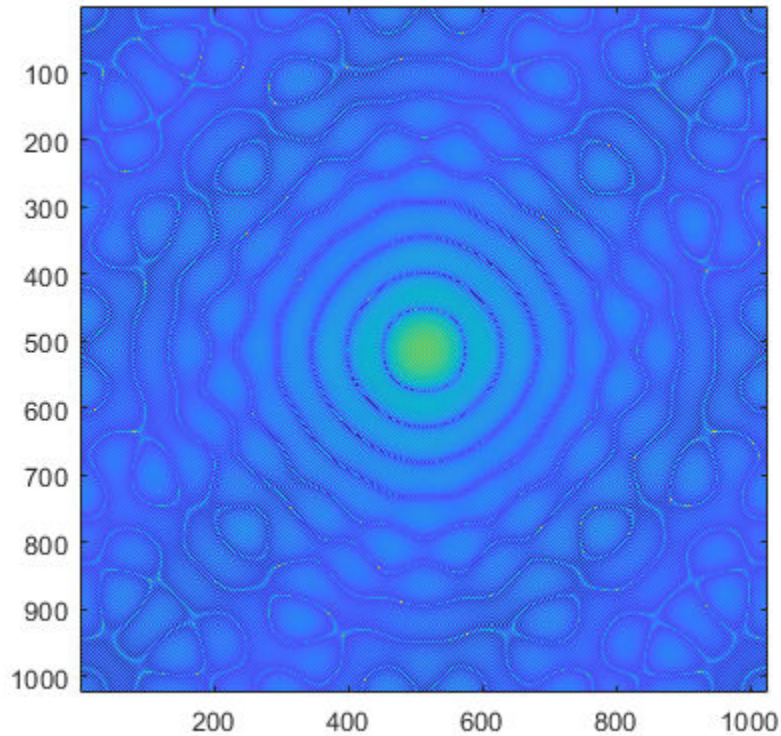imagesc(M)                 % plot mask
axis image
```


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

- "Analyzing Cyclical Data with FFT"