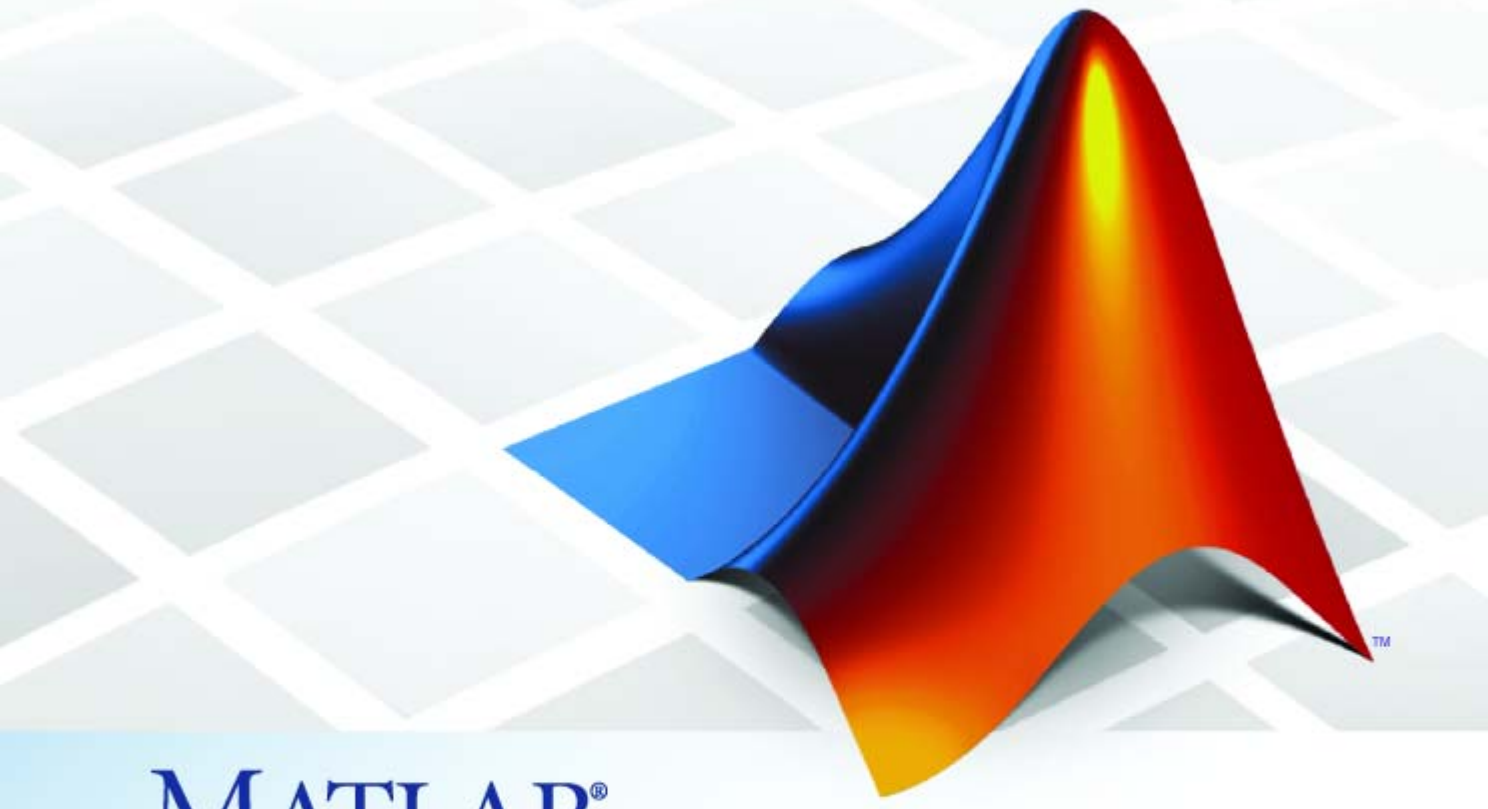
MATLAB® 7
Mathematics



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

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

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB® Mathematics

© COPYRIGHT 1984–2010 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

June 2004	First printing	New for MATLAB 7.0 (Release 14), formerly part of *Using MATLAB*
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
June 2005	Second printing	Minor revision for MATLAB 7.0.4
September 2005	Second printing	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Second printing	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Second printing	Revised for MATLAB 7.3 (Release 2006b)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b)
March 2008	Online only	Revised for MATLAB 7.6 (Release 2008a)
October 2008	Online only	Revised for MATLAB 7.7 (Release 2008b)
March 2009	Online only	Revised for MATLAB 7.8 (Release 2009a)
September 2009	Online only	Revised for MATLAB 7.9 (Release 2009b)
March 2010	Online only	Revised for MATLAB 7.10 (Release 2010a)

Contents

1 Matrices and Arrays

Creating and Concatenating Matrices 1-2
- Overview 1-2
- Constructing a Simple Matrix 1-3
- Specialized Matrix Functions 1-4
- Concatenating Matrices 1-6
- Matrix Concatenation Functions 1-7
- Generating a Numeric Sequence 1-9

Matrix Indexing 1-12
- Accessing Single Elements 1-12
- Linear Indexing 1-13
- Functions That Control Indexing Style 1-13
- Accessing Multiple Elements 1-14
- Using Logicals in Array Indexing 1-16
- Single-Colon Indexing with Different Array Types 1-20
- Indexing on Assignment 1-20

Getting Information About a Matrix 1-22
- Dimensions of the Matrix 1-22
- Classes Used in the Matrix 1-23
- Data Structures Used in the Matrix 1-24

Resizing and Reshaping Matrices 1-25
- Expanding the Size of a Matrix 1-25
- Diminishing the Size of a Matrix 1-29
- Reshaping a Matrix 1-30
- Preallocating Memory 1-32

Shifting and Sorting Matrices 1-35
- Shift and Sort Functions 1-35
- Shifting the Location of Matrix Elements 1-35
- Sorting the Data in Each Column 1-37
- Sorting the Data in Each Row 1-37
- Sorting Row Vectors 1-38

Operating on Diagonal Matrices **1-40**
Diagonal Matrix Functions **1-40**
Constructing a Matrix from a Diagonal Vector **1-40**
Returning a Triangular Portion of a Matrix **1-41**
Concatenating Matrices Diagonally **1-41**

Empty Matrices, Scalars, and Vectors **1-42**
Overview **1-42**
The Empty Matrix **1-43**
Scalars **1-45**
Vectors **1-46**

Full and Sparse Matrices **1-48**
Overview **1-48**
Sparse Matrix Functions **1-48**

Multidimensional Arrays **1-50**
Overview **1-50**
Creating Multidimensional Arrays **1-52**
Accessing Multidimensional Array Properties **1-56**
Indexing Multidimensional Arrays **1-56**
Reshaping Multidimensional Arrays **1-60**
Permuting Array Dimensions **1-62**
Computing with Multidimensional Arrays **1-64**
Organizing Data in Multidimensional Arrays **1-65**
Multidimensional Cell Arrays **1-67**
Multidimensional Structure Arrays **1-68**

Summary of Matrix and Array Functions **1-70**

2 Linear Algebra

MATLAB Linear Algebra Functions **2-2**

Matrices in the MATLAB Environment **2-5**
Creating Matrices **2-5**
Adding and Subtracting Matrices **2-7**

Vector Products and Transpose 2-7
Multiplying Matrices 2-9
Identity Matrix 2-11
Kronecker Tensor Product 2-12
Vector and Matrix Norms 2-13
Using Multithreaded Computation with Linear Algebra Functions 2-14

Systems of Linear Equations 2-15
Computational Considerations 2-15
The mldivide Algorithm 2-17
General Solution 2-18
Square Systems 2-18
Overdetermined Systems 2-21
Using Multithreaded Computation with Systems of Linear Equations 2-24
Iterative Methods for Solving Systems of Linear Equations 2-25

Inverses and Determinants 2-26
Introduction 2-26
Pseudoinverses 2-27

Factorizations 2-31
Introduction 2-31
Cholesky Factorization 2-31
LU Factorization 2-33
QR Factorization 2-34
Using Multithreaded Computation for Factorization 2-38

Powers and Exponentials 2-40
Positive Integer Powers 2-40
Inverse and Fractional Powers 2-40
Element-by-Element Powers 2-41
Exponentials 2-41

Eigenvalues 2-44
Eigenvalue Decomposition 2-44
Multiple Eigenvalues 2-45
Schur Decomposition 2-46

Singular Values **2-48**

3 Random Numbers

Generating Random Numbers **3-2**

Managing the Default Stream **3-3**
Random Number Data Types **3-8**

Creating and Controlling a Random Number Stream . . **3-10**
Substreams **3-11**
Choosing a Random Number Generator **3-12**
Compatibility Considerations **3-16**

Multiple streams **3-19**

Legacy Mode **3-22**

Selected Bibliography **3-25**

4 Sparse Matrices

Function Summary **4-2**
Functions That Support Sparse Matrices **4-2**
Functions That Do Not Support Sparse Matrices **4-6**
Functions with Sparse Alternatives **4-10**

Computational Advantages **4-10**
Memory Management **4-10**
Computational Efficiency **4-11**

Constructing Sparse Matrices **4-13**

Creating Sparse Matrices 4-13
Importing Sparse Matrices 4-18

Accessing Sparse Matrices 4-19
Nonzero Elements 4-19
Indices and Values 4-21
Indexing in Sparse Matrix Operations 4-21
Visualizing Sparse Matrices 4-24

Sparse Matrix Operations 4-25
Efficiency of Operations 4-25
Permutations and Reordering 4-26
Factoring Sparse Matrices 4-30
Systems of Linear Equations 4-37
Eigenvalues and Singular Values 4-40
Performance Limitations 4-42

Selected Bibliography 4-45

5 Functions of One Variable

Function Summary 5-2

Representing Polynomials 5-3

Evaluating Polynomials 5-4

Roots 5-5

Derivatives 5-6

Convolution 5-7

Partial Fraction Expansions 5-8

Polynomial Curve Fitting 5-9

Characteristic Polynomials 5-11

6 Computational Geometry

Querying Triangulation Representations 6-2

Delaunay Triangulation and Related Constructs 6-4
Constrained Delaunay Triangulation 6-9

Plotting Triangulations 6-15

7 Interpolation

Interpolating Uniform Data 7-2
One-Dimensional Data 7-2
Two-Dimensional Data 7-4
Multidimensional Data 7-7

Interpolating Scattered Data 7-11

8 Optimization

Function Summary 8-2

Minimizing Functions of One Variable 8-3

Minimizing Functions of Several Variables 8-5

fminsearch Algorithm 8-6

Example: Curve Fitting 8-9
Creating an Example File 8-9
Running the Example 8-10
Plotting the Results 8-10

Setting Options 8-12

Output Functions 8-14
What Is an Output Function? 8-14
Creating and Using an Output Function 8-14
Structure of the Output Function 8-16
Example of a Nested Output Function 8-17
Fields in optimValues 8-19
States of the Algorithm 8-19
Stop Flag 8-20

Plot Functions 8-22
Example: Plot Function 8-22

Finding Roots 8-25
Using a Starting Interval 8-26
Using a Starting Point 8-27

Troubleshooting 8-30

Tips 8-31

Reference 8-32

9 Function Handles

Introduction **9-2**

Defining Functions In Files **9-3**

Anonymous Functions **9-4**

Example: Function Plotting Function **9-5**

Parameterizing Functions **9-8**
Using Nested Functions **9-8**
Using Anonymous Functions **9-9**

10 Calculus

Ordinary Differential Equations **10-2**
Function Summary **10-2**
Initial Value Problems **10-4**
Types of Solvers **10-6**
Solver Syntax **10-8**
Integrator Options **10-9**
Examples **10-10**
Troubleshooting **10-39**

Delay Differential Equations **10-47**
Function Summary **10-47**
Initial Value Problems **10-48**
Types of Solvers **10-49**
Discontinuities **10-50**
Integrator Options **10-51**
Examples **10-51**

Boundary-Value Problems **10-59**
Function Summary **10-59**

Boundary Value Problems 10-60
BVP Solver 10-61
Integrator Options 10-64
Examples 10-65

Partial Differential Equations 10-87
Function Summary 10-87
Initial Value Problems 10-88
PDE Solver 10-89
Integrator Options 10-92
Examples 10-93

Selected Bibliography 10-105

Integration 10-106
Quadrature Functions 10-106
Example: Arc Length 10-107
Example: Double Integration 10-107

11 Fourier Transforms

Discrete Fourier Transform (DFT) 11-2
Introduction 11-2
Visualizing the DFT 11-3

Fast Fourier Transform (FFT) 11-8
Introduction 11-8
The FFT in One Dimension 11-9
The FFT in Multiple Dimensions 11-23

Function Summary 11-28

Index

1

Matrices and Arrays

- "Creating and Concatenating Matrices" on page 1-2
- "Matrix Indexing" on page 1-12
- "Getting Information About a Matrix" on page 1-22
- "Resizing and Reshaping Matrices" on page 1-25
- "Shifting and Sorting Matrices" on page 1-35
- "Operating on Diagonal Matrices" on page 1-40
- "Empty Matrices, Scalars, and Vectors" on page 1-42
- "Full and Sparse Matrices" on page 1-48
- "Multidimensional Arrays" on page 1-50
- "Summary of Matrix and Array Functions" on page 1-70

Creating and Concatenating Matrices

In this section...

"Overview" on page 1-2
"Constructing a Simple Matrix" on page 1-3
"Specialized Matrix Functions" on page 1-4
"Concatenating Matrices" on page 1-6
"Matrix Concatenation Functions" on page 1-7
"Generating a Numeric Sequence" on page 1-9

Overview

The most basic MATLAB® data structure is the *matrix*: a two-dimensional, rectangularly shaped data structure capable of storing multiple elements of data in an easily accessible format. These data elements can be numbers, characters, logical states of `true` or `false`, or even other MATLAB structure types. MATLAB uses these two-dimensional matrices to store single numbers and linear series of numbers as well. In these cases, the dimensions are 1-by-1 and 1-by-n respectively, where `n` is the length of the numeric series. MATLAB also supports data structures that have more than two dimensions. These data structures are referred to as *arrays* in the MATLAB documentation.

MATLAB is a matrix-based computing environment. All of the data that you enter into MATLAB is stored in the form of a matrix or a multidimensional array. Even a single numeric value like `100` is stored as a matrix (in this case, a matrix having dimensions 1-by-1):

```
A = 100;

whos A
  Name      Size                    Bytes  Class

  A         1x1                         8  double array
```

Regardless of the class being used, whether it is numeric, character, or logical `true` or `false` data, MATLAB stores this data in matrix (or array) form. For example, the string `'Hello World'` is a 1-by-11 matrix of individual character

elements in MATLAB. You can also build matrices composed of more complex classes, such as MATLAB structures and cell arrays.

To create a matrix of basic data elements such as numbers or characters, see

- "Constructing a Simple Matrix" on page 1-3
- "Specialized Matrix Functions" on page 1-4

To build a matrix composed of other matrices, see

- "Concatenating Matrices" on page 1-6
- "Matrix Concatenation Functions" on page 1-7

This section also describes

- "Generating a Numeric Sequence" on page 1-9
- "Combining Unlike Classes"

Constructing a Simple Matrix

The simplest way to create a matrix in MATLAB is to use the matrix constructor operator, `[]`. Create a row in the matrix by entering elements (shown as `E` below) within the brackets. Separate each element with a comma or space:

```
row = [E1, E2, ..., Em]          row = [E1 E2 ... Em]
```

For example, to create a one row matrix of five elements, type

```
A = [12 62 93 -8 22];
```

To start a new row, terminate the current row with a semicolon:

```
A = [row1; row2; ...; rown]
```

This example constructs a 3 row, 5 column (or 3-by-5) matrix of numbers. Note that all rows must have the same number of elements:

```
A = [12 62 93 -8 22; 16 2 87 43 91; -4 17 -72 95 6]
A =
```

```
    12    62    93    -8    22
    16     2    87    43    91
    -4    17   -72    95     6
```

The square brackets operator constructs two-dimensional matrices only, (including 0-by-0, 1-by-1, and 1-by-n matrices). To construct arrays of more than two dimensions, see "Creating Multidimensional Arrays" on page 1-52.

For instructions on how to read or overwrite any matrix element, see "Matrix Indexing" on page 1-12.

Entering Signed Numbers

When entering signed numbers into a matrix, make sure that the sign immediately precedes the numeric value. Note that while the following two expressions are equivalent,

```
7 -2 +5
ans =
    10
```

```
7 - 2 + 5
  ans =
      10
```

the next two are *not*:

```
[7 -2 +5]
ans =
     7    -2     5
```

```
[7 - 2 + 5]
   ans =
       10
```

Specialized Matrix Functions

MATLAB has a number of functions that create different kinds of matrices. Some create specialized matrices like the Hankel or Vandermonde matrix. The functions shown in the table below create matrices for more general use.

Function	Description
`ones`	Create a matrix or array of all ones.
`zeros`	Create a matrix or array of all zeros.
`eye`	Create a matrix with ones on the diagonal and zeros elsewhere.

Function	Description
`accumarray`	Distribute elements of an input matrix to specified locations in an output matrix, also allowing for accumulation.
`diag`	Create a diagonal matrix from a vector.
`magic`	Create a square matrix with rows, columns, and diagonals that add up to the same number.
`rand`	Create a matrix or array of uniformly distributed random numbers.
`randn`	Create a matrix or array of normally distributed random numbers and arrays.
`randperm`	Create a vector (1-by-n matrix) containing a random permutation of the specified integers.

Most of these functions return matrices of type `double` (double-precision floating point). However, you can easily build basic arrays of any numeric type using the `ones`, `zeros`, and `eye` functions.

To do this, specify the MATLAB class name as the last argument:

```
A = zeros(4, 6, 'uint32')
A =
        0        0        0        0        0        0
        0        0        0        0        0        0
        0        0        0        0        0        0
        0        0        0        0        0        0
```

Examples

Here are some examples of how you can use these functions.

Creating a Magic Square Matrix. A magic square is a matrix in which the sum of the elements in each column, or each row, or each main diagonal is the same. To create a 5-by-5 magic square matrix, use the `magic` function as shown.

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Note that the elements of each row, each column, and each main diagonal add up to the same value: 65.

Creating a Diagonal Matrix. Use `diag` to create a diagonal matrix from a vector. You can place the vector along the main diagonal of the matrix, or on a diagonal that is above or below the main one, as shown here. The `-1` input places the vector one row below the main diagonal:

```
A = [12 62 93 -8 22];

B = diag(A, -1)
B =
     0     0     0     0     0     0
    12     0     0     0     0     0
     0    62     0     0     0     0
     0     0    93     0     0     0
     0     0     0    -8     0     0
     0     0     0     0    22     0
```

Concatenating Matrices

Matrix concatenation is the process of joining one or more matrices to make a new matrix. The brackets `[]` operator discussed earlier in this section serves not only as a matrix constructor, but also as the MATLAB concatenation operator. The expression `C = [A B]` horizontally concatenates matrices `A` and `B`. The expression `C = [A; B]` vertically concatenates them.

This example constructs a new matrix `C` by concatenating matrices `A` and `B` in a vertical direction:

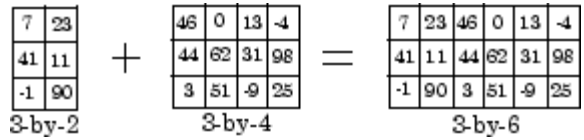
```
A = ones(2, 5) * 6;         % 2-by-5 matrix of 6's
B = rand(3, 5);             % 3-by-5 matrix of random values
```

```
C = [A; B]                  % Vertically concatenate A and B
C =
    6.0000    6.0000    6.0000    6.0000    6.0000
    6.0000    6.0000    6.0000    6.0000    6.0000
    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
```

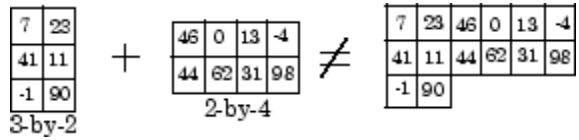
Keeping Matrices Rectangular

You can construct matrices, or even multidimensional arrays, using concatenation as long as the resulting matrix does not have an irregular shape (as in the second illustration shown below). If you are building a matrix horizontally, then each component matrix must have the same number of rows. When building vertically, each component must have the same number of columns.

This diagram shows two matrices of the same height (i.e., same number of rows) being combined horizontally to form a new matrix.



The next diagram illustrates an attempt to horizontally combine two matrices of unequal height. MATLAB does not allow this.



Matrix Concatenation Functions

The following functions combine existing matrices to form a new matrix.

Function	Description
cat	Concatenate matrices along the specified dimension
horzcat	Horizontally concatenate matrices
vertcat	Vertically concatenate matrices
repmat	Create a new matrix by replicating and tiling existing matrices
blkdiag	Create a block diagonal matrix from existing matrices

Examples

Here are some examples of how you can use these functions.

Concatenating Matrices and Arrays. An alternative to using the `[]` operator for concatenation are the three functions `cat`, `horzcat`, and `vertcat`. With these functions, you can construct matrices (or multidimensional arrays) along a specified dimension. Either of the following commands accomplish the same task as the command `C = [A; B]` used in the section on "Concatenating Matrices" on page 1-6:

```
C = cat(1, A, B);       % Concatenate along the first dimension
C = vertcat(A, B);      % Concatenate vertically
```

Replicating a Matrix. Use the `repmat` function to create a matrix composed of copies of an existing matrix. When you enter

```
repmat(M, v, h)
```

MATLAB replicates input matrix `M` `v` times vertically and `h` times horizontally. For example, to replicate existing matrix `A` into a new matrix `B`, use

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

B = repmat(A, 2, 4)
B =
```

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

Creating a Block Diagonal Matrix. The `blkdiag` function combines matrices in a diagonal direction, creating what is called a block diagonal matrix. All other elements of the newly created matrix are set to zero:

```
A = magic(3);
B = [-5 -6 -9; -4 -4 -2];
C = eye(2) * 8;

D = blkdiag(A, B, C)
D =
   8   1   6   0   0   0   0   0
   3   5   7   0   0   0   0   0
   4   9   2   0   0   0   0   0
   0   0   0  -5  -6  -9   0   0
   0   0   0  -4  -4  -2   0   0
   0   0   0   0   0   0   8   0
   0   0   0   0   0   0   0   8
```

Generating a Numeric Sequence

Because numeric sequences can often be useful in constructing and indexing into matrices and arrays, MATLAB provides a special operator to assist in creating them.

This section covers

- "The Colon Operator" on page 1-10
- "Using the Colon Operator with a Step Value" on page 1-10

The Colon Operator

The colon operator (`first:last`) generates a 1-by-n matrix (or *vector*) of sequential numbers from the `first` value to the `last`. The default sequence is made up of incremental values, each 1 greater than the previous one:

```
A = 10:15
A =
    10    11    12    13    14    15
```

The numeric sequence does not have to be made up of positive integers. It can include negative numbers and fractional numbers as well:

```
A = -2.5:2.5
A =
   -2.5000   -1.5000   -0.5000    0.5000    1.5000    2.5000
```

By default, MATLAB always increments by exactly 1 when creating the sequence, even if the ending value is not an integral distance from the start:

```
A = 1:6.3
A =
     1     2     3     4     5     6
```

Also, the default series generated by the colon operator always increments rather than decrementing. The operation shown in this example attempts to increment from 9 to 1 and thus MATLAB returns an empty matrix:

```
A = 9:1
A =
   Empty matrix: 1-by-0
```

The next section explains how to generate a nondefault numeric series.

Using the Colon Operator with a Step Value

To generate a series that does not use the default of incrementing by 1, specify an additional value with the colon operator (`first:step:last`). In between the starting and ending value is a `step` value that tells MATLAB how much to increment (or decrement, if `step` is negative) between each number it generates.

To generate a series of numbers from 10 to 50, incrementing by 5, use

```
A = 10:5:50
A =
    10    15    20    25    30    35    40    45    50
```

You can increment by noninteger values. This example increments by `0.2`:

```
A = 3:0.2:3.8
A =
    3.0000    3.2000    3.4000    3.6000    3.8000
```

To create a sequence with a decrementing interval, specify a negative step value:

```
A = 9:-1:1
A =
    9    8    7    6    5    4    3    2    1
```

Matrix Indexing

In this section...

"Accessing Single Elements" on page 1-12
"Linear Indexing" on page 1-13
"Functions That Control Indexing Style" on page 1-13
"Accessing Multiple Elements" on page 1-14
"Using Logicals in Array Indexing" on page 1-16
"Single-Colon Indexing with Different Array Types" on page 1-20
"Indexing on Assignment" on page 1-20

Accessing Single Elements

To reference a particular element in a matrix, specify its row and column number using the following syntax, where `A` is the matrix variable. Always specify the row first and column second:

```
A(row, column)
```

For example, for a 4-by-4 magic square `A`,

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

you would access the element at row 4, column 2 with

```
A(4, 2)
ans =
    14
```

For arrays with more than two dimensions, specify additional indices following the `row` and `column` indices. See the section on "Multidimensional Arrays" on page 1-50.

Linear Indexing

You can refer to the elements of a MATLAB matrix with a single subscript, `A(k)`. MATLAB stores matrices and arrays not in the shape that they appear when displayed in the MATLAB Command Window, but as a single column of elements. This single column is composed of all of the columns from the matrix, each appended to the last.

So, matrix `A`

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

is actually stored in memory as the sequence

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

The element at row 3, column 2 of matrix `A` (value = `5`) can also be identified as element 6 in the actual storage sequence. To access this element, you have a choice of using the standard `A(3,2)` syntax, or you can use `A(6)`, which is referred to as *linear indexing*.

If you supply more subscripts, MATLAB calculates an index into the storage column based on the dimensions you assigned to the array. For example, assume a two-dimensional array like `A` has size `[d1 d2]`, where `d1` is the number of rows in the array and `d2` is the number of columns. If you supply two subscripts `(i, j)` representing row-column indices, the offset is

```
(j-1) * d1 + i
```

Given the expression `A(3,2)`, MATLAB calculates the offset into `A`'s storage column as `(2-1) * 3 + 3`, or `6`. Counting down six elements in the column accesses the value `5`.

Functions That Control Indexing Style

If you have row-column subscripts but want to use linear indexing instead, you can convert to the latter using the `sub2ind` function. In the 3-by-3 matrix

A used in the previous section, sub2ind changes a standard row-column index of (3,2) to a linear index of 6:

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

linearindex = sub2ind(size(A), 3, 2)
linearindex =
     6
```

To get the row-column equivalent of a linear index, use the ind2sub function:

```
[row col] = ind2sub(size(A), 6)
row =
     3
col =
     2
```

Accessing Multiple Elements

For the 4-by-4 matrix A shown below, it is possible to compute the sum of the elements in the fourth column of A by typing

```
A = magic(4);
A(1,4) + A(2,4) + A(3,4) + A(4,4)
```

You can reduce the size of this expression using the colon operator. Subscript expressions involving colons refer to portions of a matrix. The expression

```
A(1:m, n)
```

refers to the elements in rows 1 through m of column n of matrix A. Using this notation, you can compute the sum of the fourth column of A more succinctly:

```
sum(A(1:4, 4))
```

Nonconsecutive Elements

To refer to nonconsecutive elements in a matrix, use the colon operator with a step value. The m:3:n in this expression means to make the assignment to every third element in the matrix. Note that this example uses linear indexing:

```
B = A;

B(1:3:16) = -10
B =
   -10     2     3   -10
     5    11   -10     8
     9   -10     6    12
   -10    14    15   -10
```

MATLAB supports a type of array indexing that uses one array as the index into another array. You can base this type of indexing on either the values or the positions of elements in the indexing array.

Here is an example of value-based indexing where array `B` indexes into elements `1`, `3`, `6`, `7`, and `10` of array `A`. In this case, the *numeric values* of array `B` designate the intended elements of `A`:

```
A = 5:5:50
A =
     5    10    15    20    25    30    35    40    45    50
B = [1 3 6 7 10];

A(B)
ans =
     5    15    30    35    50
```

The end Keyword

MATLAB provides the keyword `end` to designate the last element in a particular dimension of an array. This keyword can be useful in instances where your program does not know how many rows or columns there are in a matrix. You can replace the expression in the previous example with

```
B(1:3:end) = -10
```

Note The keyword `end` has several meanings in MATLAB. It can be used as explained above, or to terminate a conditional block of code such as `if` and `for` blocks, or to terminate a nested function.

Specifying All Elements of a Row or Column

The colon by itself refers to *all* the elements in a row or column of a matrix. Using the following syntax, you can compute the sum of all elements in the second column of a 4-by-4 magic square `A`:

```
sum(A(:, 2))
ans =
    34
```

By using the colon with linear indexing, you can refer to all elements in the entire matrix. This example displays all the elements of matrix `A`, returning them in a column-wise order:

```
A(:)
ans =
    16
     5
     9
     4
     .
     .
     .
    12
     1
```

Using Logicals in Array Indexing

A logical array index designates the elements of an array `A` based on their *position* in the indexing array, `B`, not their value. In this *masking* type of operation, every `true` element in the indexing array is treated as a positional index into the array being accessed.

In the following example, `B` is a matrix of logical ones and zeros. The position of these elements in `B` determines which elements of `A` are designated by the expression `A(B)`:

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

```
B = logical([0 1 0; 1 0 1; 0 0 1]);
B =
     0     1     0
     1     0     1
     0     0     1

A(B)
ans =
     4
     2
     6
     9
```

The `find` function can be useful with logical arrays as it returns the linear indices of nonzero elements in `B`, and thus helps to interpret `A(B)`:

```
find(B)
ans =
     2
     4
     8
     9
```

Logical Indexing – Example 1

This example creates logical array `B` that satisfies the condition `A > 0.5`, and uses the positions of ones in `B` to index into `A`:

```
rand('twister', 5489);     % Initialize the state of the
                           % random number generator.
A = rand(5);
B = A > 0.5;

A(B) = 0
A =
         0    0.0975    0.1576    0.1419         0
         0    0.2785         0    0.4218    0.0357
    0.1270         0         0         0         0
         0         0    0.4854         0         0
         0         0         0         0         0
```

A simpler way to express this is

```
A(A > 0.5) = 0
```

Logical Indexing – Example 2

The next example highlights the location of the prime numbers in a magic square using logical indexing to set the nonprimes to 0:

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

B = isprime(A)
B =
     0     1     1     1
     1     1     0     0
     0     1     0     0
     0     0     0     0

A(~B) = 0;                          % Logical indexing

A
A =
     0     2     3    13
     5    11     0     0
     0     7     0     0
     0     0     0     0

find(B)
ans =
     2
     5
     6
     7
     9
    13
```

Logical Indexing with a Smaller Array

In most cases, the logical indexing array should have the same number of elements as the array being indexed into, but this is not a requirement. The indexing array may have smaller (but not larger) dimensions:

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

B = logical([0 1 0; 1 0 1])
B =
     0     1     0
     1     0     1

isequal(numel(A), numel(B))
ans =
     0

A(B)
ans =
     4
     7
     8
```

MATLAB treats the missing elements of the indexing array as if they were present and set to zero, as in array `C` below:

```
% Add zeros to indexing array C to give it the same number of
% elements as A.
C = logical([B(:);0;0;0]);

isequal(numel(A), numel(C))
ans =
     1

A(C)
ans =
     4
```

```
     7
     8
```

Single-Colon Indexing with Different Array Types

When you index into a standard MATLAB array using a single colon, MATLAB returns a column vector (see variable `n`, below). When you index into a structure or cell array using a single colon, you get a comma-separated list (see "Returning Data from a Struct Array" and "Returning Data from a Cell Array" for more information.)

Create three types of arrays:

```
n = [1 2 3; 4 5 6];
c = {1 2; 3 4};
s = cell2struct(c, {'a', 'b'}, 1);  s(:,2)=s(:,1);
```

Use single-colon indexing on each:

```
n(:)              c{:}              s(:).a
ans =             ans =             ans =
     1                 1                 1
     4             ans =             ans =
     2                 3                 2
     5             ans =             ans =
     3                 2                 1
     6             ans =             ans =
                       4                 2
```

Indexing on Assignment

When assigning values from one matrix to another matrix, you can use any of the styles of indexing covered in this section. Matrix assignment statements also have the following requirement.

In the assignment `A(J,K,...) = B(M,N,...)`, subscripts `J`, `K`, `M`, `N`, etc. may be scalar, vector, or array, provided that all of the following are true:

- The number of subscripts specified for `B`, not including trailing subscripts equal to 1, does not exceed `ndims(B)`.

- The number of nonscalar subscripts specified for `A` equals the number of nonscalar subscripts specified for `B`. For example, `A(5, 1:4, 1, 2) = B(5:8)` is valid because both sides of the equation use one nonscalar subscript.
- The order and length of all nonscalar subscripts specified for `A` matches the order and length of nonscalar subscripts specified for `B`. For example, `A(1:4, 3, 3:9) = B(5:8, 1:7)` is valid because both sides of the equation (ignoring the one scalar subscript `3`) use a 4-element subscript followed by a 7-element subscript.

Getting Information About a Matrix

In this section...
"Dimensions of the Matrix" on page 1-22
"Classes Used in the Matrix" on page 1-23
"Data Structures Used in the Matrix" on page 1-24

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.

Function	Description
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.
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.

Resizing and Reshaping Matrices

In this section...
"Expanding the Size of a Matrix" on page 1-25
"Diminishing the Size of a Matrix" on page 1-29
"Reshaping a Matrix" on page 1-30
"Preallocating Memory" on page 1-32

Expanding the Size of a Matrix

You can expand the size of any existing matrix as long as doing so does not give the resulting matrix an irregular shape. (See "Keeping Matrices Rectangular" on page 1-7). For example, you can vertically combine a 4-by-3 matrix and 7-by-3 matrix because all rows of the resulting matrix have the same number of columns (3).

Two ways of expanding the size of an existing matrix are

- Concatenating new elements onto the matrix
- Storing to a location outside the bounds of the matrix

Note If you intend to expand the size of a matrix repeatedly over time as it requires more room (usually done in a programming loop), it is advisable to preallocate space for the matrix when you initially create it. See "Preallocating Memory" on page 1-32.

Concatenating Onto the Matrix

Concatenation is most useful when you want to expand a matrix by adding new elements or blocks that are compatible in size with the original matrix. This means that the size of all matrices being joined along a specific dimension must be equal along that dimension. See "Concatenating Matrices" on page 1-6.

This example runs a user-defined function `compareResults` on the data in matrices `stats04` and `stats03`. Each time through the loop, it concatenates the results of this function onto the end of the data stored in `comp04`:

```
col = 10;
comp04 = [];

for k = 1:50
   t = compareResults(stats04(k,1:col), stats03(k,1:col));
   comp04 = [comp04; t];
end
```

Concatenating to a Structure or Cell Array. You can add on to arrays of structures or cells in the same way as you do with ordinary matrices. This example creates a 3-by-8 matrix of structures `S`, each having 3 fields: `x`, `y`, and `z`, and then concatenates a second structure matrix `S2` onto the original:

Create a 3-by-8 structure array `S`:

```
for k = 1:24
   S(k) = struct('x', 10*k, 'y', 10*k+1, 'z', 10*k+2);
end
S = reshape(S, 3, 8);
```

Create a second array that is 3-by-2 and uses the same field names:

```
for k = 25:30
   S2(k-24) = struct('x', 10*k, 'y', 10*k+1, 'z', 10*k+2);
end
S2= reshape(S2, 3, 2);
```

Concatenate `S2` onto `S` along the horizontal dimension:

```
S = [S S2]
S =
3x10 struct array with fields:
    x
    y
    z
```

Adding Smaller Blocks to a Matrix

To add one or more elements to a matrix where the sizes are not compatible, you can often just store the new elements outside the boundaries of the original matrix. The MATLAB software automatically pads the matrix with zeros to keep it rectangular.

Construct a 3-by-5 matrix, and attempt to add a new element to it using concatenation. The operation fails because you are attempting to join a one-column matrix with one that has five columns:

```
A = [ 10  20  30  40  50; ...
      60  70  80  90 100; ...
     110 120 130 140 150];

A = [A; 160]
??? Error using ==> vertcat
All rows in the bracketed expression must have the same
number of columns.
```

Try this again, but this time do it in such a way that enables MATLAB to make adjustments to the size of the matrix. Store the new element in row 4, a row that does not yet exist in this matrix. MATLAB expands matrix `A` by an entire new row by padding columns 2 through 5 with zeros:

```
A(4,1) = 160
A =
    10    20    30    40    50
    60    70    80    90   100
   110   120   130   140   150
   160     0     0     0     0
```

Note Attempting to read from nonexistent matrix locations generates an error. You can only write to these locations.

You can also expand the matrix by adding a matrix instead of just a single element:

```
A(4:6,1:3) = magic(3)+100
```

```
A =
    10    20    30    40    50
    60    70    80    90   100
   110   120   130   140   150
   108   101   106     0     0
   103   105   107     0     0
   104   109   102     0     0
```

You do not have to add new elements sequentially. Wherever you store the new elements, MATLAB pads with zeros to make the resulting matrix rectangular in shape:

```
A(4,8) = 300
A =
    10    20    30    40    50     0     0     0
    60    70    80    90   100     0     0     0
   110   120   130   140   150     0     0     0
     0     0     0     0     0     0     0   300
```

Expanding a Structure or Cell Array. You can expand a structure or cell array in the same way that you can a matrix. This example adds an additional cell to a cell array by storing it beyond the bounds of the original array. MATLAB pads the data structure with empty cells (`[]`) to keep it rectangular.

The original array is 2-by-3:

```
C = {'Madison', 'G', [5 28 1967]; ...
     46, '325 Maple Dr', 3015.28}
```

Add a cell to `C{3,1}` and MATLAB appends an entire row:

```
C{3, 1} = ...
struct('Fund_A', .45, 'Fund_E', .35, 'Fund_G', 20);
C =
    'Madison'       'G'                [1x3 double]
    [        46]    '325 Maple Dr'    [3.0153e+003]
    [1x1 struct]               []               []
```

Expanding a Character Array. You can expand character arrays in the same manner as other MATLAB arrays, but it is generally not recommended. MATLAB expands any array by padding uninitialized elements with zeros. Because zero is interpreted by MATLAB and some other programming languages as a string terminator, you may find that some functions treat the expanded string as if it were less than its full length.

Expand a 1-by-5 character array to twelve characters. The result appears at first to be a typical string:

```
greeting = 'Hello';    greeting(1,8:12) = 'World'
greeting =
   Hello  World
```

Closer inspection however reveals string terminators at the point of expansion:

```
uint8(greeting)
ans =
   72  101  108  108  111    0    0   87  111  114  108  100
```

This causes some functions, like `strcmp`, to return what might be considered an unexpected result:

```
strcmp(greeting, 'Hello  World')
ans =
     0
```

Diminishing the Size of a Matrix

You can delete rows and columns from a matrix by assigning the empty array `[]` to those rows or columns. Start with

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1
```

Then, delete the second column of `A` using

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

This changes matrix A to

```
A =
    16     3    13
     5    10     8
     9     6    12
     4    15     1
```

If you delete a single element from a matrix, the result is not a matrix anymore. So expressions like

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

result in an error. However, you can use linear indexing to delete a single element, or a sequence of elements. This reshapes the remaining elements into a row vector:

```
A(2:2:10) = []
```

results in

```
A =
    16     9     3     6    13    12     1
```

Reshaping a Matrix

The following functions change the shape of a matrix.

Function	Description
`reshape`	Modify the shape of a matrix.
`rot90`	Rotate the matrix by 90 degrees.
`fliplr`	Flip the matrix about a vertical axis.
`flipud`	Flip the matrix about a horizontal axis.
`flipdim`	Flip the matrix along the specified direction.

Function	Description
`transpose`	Flip a matrix about its main diagonal, turning row vectors into column vectors and vice versa.
`ctranspose`	Transpose a matrix and replace each element with its complex conjugate.

Examples

Here are a few examples to illustrate some of the ways you can reshape matrices.

Reshaping a Matrix. Reshape 3-by-4 matrix `A` to have dimensions 2-by-6:

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

B = reshape(A, 2, 6)
B =
    1    3    5    7    9   11
    2    4    6    8   10   12
```

Transposing a Matrix. Transpose `A` so that the row elements become columns. You can use either the `transpose` function or the transpose operator (`.'`) to do this:

```
B = A.'
B =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

There is a separate function called `ctranspose` that performs a complex conjugate transpose of a matrix. The equivalent operator for `ctranpose` on a matrix `A` is `A'`:

```
A = [1+9i 2-8i 3+7i; 4-6i 5+5i 6-4i]
A =
   1.0000 + 9.0000i   2.0000 -8.0000i   3.0000 + 7.0000i
   4.0000 -6.0000i   5.0000 + 5.0000i   6.0000 -4.0000i

B = A'
B =
   1.0000 -9.0000i   4.0000 + 6.0000i
   2.0000 + 8.0000i   5.0000 -5.0000i
   3.0000 -7.0000i   6.0000 + 4.0000i
```

Rotating a Matrix. Rotate the matrix by 90 degrees:

```
B = rot90(A)
B =
    10    11    12
     7     8     9
     4     5     6
     1     2     3
```

Flipping a Matrix. Flip `A` in a left-to-right direction:

```
B = fliplr(A)
B =
    10     7     4     1
    11     8     5     2
    12     9     6     3
```

Preallocating Memory

Repeatedly expanding the size of an array over time, (for example, adding more elements to it each time through a programming loop), can adversely affect the performance of your program. This is because

- MATLAB has to spend time allocating more memory each time you increase the size of the array.
- This newly allocated memory is likely to be noncontiguous, thus slowing down any operations that MATLAB needs to perform on the array.

The preferred method for sizing an array that is expected to grow over time is to estimate the maximum possible size for the array, and preallocate this amount of memory for it at the time the array is created. In this way, your program performs one memory allocation that reserves one contiguous block.

The following command preallocates enough space for a 25,000 by 10,000 matrix, and initializes each element to zero:

```
A = zeros(25000, 10000);
```

Building a Preallocated Array

Once memory has been preallocated for the maximum estimated size of the array, you can store your data in the array as you need it, each time appending to the existing data. This example preallocates a large array, and then reads blocks of data from a file into the array until it gets to the end of the file:

```
blocksize = 5000;
maxrows = 2500000; cols = 20;
rp = 1;     % row pointer

% Preallocate A to its maximum possible size
A = zeros(maxrows, cols);

% Open the data file, saving the file pointer.
fid = fopen('statfile.dat', 'r');

while true
   % Read from file into a cell array.  Stop at EOF.
   block = textscan(fid, '%n', blocksize*cols);
   if isempty(block{1})   break,   end;

   % Convert cell array to matrix, reshape, place into A.
   A(rp:rp+blocksize-1, 1:cols) = ...
      reshape(cell2mat(block), blocksize, cols);

   % Process the data in A.
   evaluate_stats(A);              % User-defined function

   % Update row pointer
```

```
    rp = rp + blocksize;
end
```

Note If you eventually need more room in a matrix than you had preallocated, you can preallocate additional storage in the same manner, and concatenate this additional storage onto the original array.

Shifting and Sorting Matrices

In this section...
"Shift and Sort Functions" on page 1-35
"Shifting the Location of Matrix Elements" on page 1-35
"Sorting the Data in Each Column" on page 1-37
"Sorting the Data in Each Row" on page 1-37
"Sorting Row Vectors" on page 1-38

Shift and Sort Functions

Use these functions to shift or sort the elements of a matrix.

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

You can sort matrices, multidimensional arrays, and cell arrays of strings along any dimension and in ascending or descending order of the elements. The sort functions also return an optional array of indices showing the order in which elements were rearranged during the sorting operation.

Shifting the Location of Matrix Elements

The `circshift` function shifts the elements of a matrix in a circular manner along one or more dimensions. Rows or columns that are shifted out of the matrix circulate back into the opposite end. For example, shifting a 4-by-7 matrix one place to the left moves the elements in columns 2 through 7 to columns 1 through 6, and moves column 1 to column 7.

Create a 5-by-8 matrix named `A` and shift it to the right along the second (horizontal) dimension by three places. (You would use `[0, -3]` to shift to the left by three places):

```
A = [1:8; 11:18; 21:28; 31:38; 41:48]
A =
     1     2     3     4     5     6     7     8
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48

B = circshift(A, [0, 3])
B =
     6     7     8     1     2     3     4     5
    16    17    18    11    12    13    14    15
    26    27    28    21    22    23    24    25
    36    37    38    31    32    33    34    35
    46    47    48    41    42    43    44    45
```

Now take `A` and shift it along both dimensions: three columns to the right and two rows up:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48];

B = circshift(A, [-2, 3])
B =
    26    27    28    21    22    23    24    25
    36    37    38    31    32    33    34    35
    46    47    48    41    42    43    44    45
     6     7     8     1     2     3     4     5
    16    17    18    11    12    13    14    15
```

Since `circshift` circulates shifted rows and columns around to the other end of a matrix, shifting by the exact size of `A` returns all rows and columns to their original location:

```
B = circshift(A, size(A));

all(B(:) == A(:))          % Do all elements of B equal A?
ans =
     1                     % Yes
```

Sorting the Data in Each Column

The `sort` function sorts matrix elements along a specified dimension. The syntax for the function is

```
sort(matrix, dimension)
```

To sort the columns of a matrix, specify 1 as the `dimension` argument. To sort along rows, specify `dimension` as 2.

This example makes a 6-by-7 arbitrary test matrix:

```
A=floor(gallery('uniformdata',[6 7],0)*100)
A =
    95    45    92    41    13     1    84
    23     1    73    89    20    74    52
    60    82    17     5    19    44    20
    48    44    40    35    60    93    67
    89    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

Sort each column of `A` in ascending order:

```
c = sort(A, 1)
c =
    23     1    17     0    13     1     1
    48    44    40     5    19    41    20
    60    45    73    35    19    44    52
    76    61    91    41    20    46    67
    89    79    92    81    27    74    83
    95    82    93    89    60    93    84

issorted(c(:, 1))
ans =
     1
```

Sorting the Data in Each Row

Use `issorted` to sort data in each row. Using the example above, if you sort each row of `A` in descending order, `issorted` tests for an ascending sequence. You can flip the vector to test for a sorted descending sequence:

```
A=floor(gallery('uniformdata',[6 7],0)*100);

r = sort(A, 2, 'descend')
r =
    95    92    84    45    41    13     1
    89    74    73    52    23    20     1
    82    60    44    20    19    17     5
    93    67    60    48    44    40    35
    93    89    83    81    61    46    27
    91    79    76    41    19     1     0

issorted(fliplr(r(1, :)))
ans =
     1
```

When you specify a second output, `sort` returns the indices of the original matrix `A` positioned in the order they appear in the output matrix. In this next example, the second row of `index` contains the sequence `4 3 2 5 1`, which means that the sorted elements in output matrix `r` were taken from `A(2,4)`, `A(2,3)`, `A(2,2)`, `A(2,5)`, and `A(2,1)`:

```
[r index] = sort(A, 2, 'descend');
index
index =
     1     3     7     2     4     5     6
     4     6     3     7     1     5     2
     2     1     6     7     5     3     4
     6     7     5     1     2     3     4
     3     1     7     4     2     6     5
     3     2     1     6     5     7     4
```

Sorting Row Vectors

The `sortrows` function keeps the elements of each row in its original order, but sorts the entire row of vectors according to the order of the elements in the specified column.

The next example creates a random matrix `A`:

```
A=floor(gallery('uniformdata',[6 7],0)*100);
A =
    95    45    92    41    13     1    84
    23     1    73    89    20    74    52
    60    82    17     5    19    44    20
    48    44    40    35    60    93    67
    89    61    93    81    27    46    83
    76    79    91     0    19    41     1
```

To sort in ascending order based on the values in column 1, you can call `sortrows` with just the one input argument:

```
sortrows(A)
r =
    23     1    73    89    20    74    52
    48    44    40    35    60    93    67
    60    82    17     5    19    44    20
    76    79    91     0    19    41     1
    89    61    93    81    27    46    83
    95    45    92    41    13     1    84
```

To base the sort on a column other than the first, call `sortrows` with a second input argument that indicates the column number, column 4 in this case:

```
r = sortrows(A, 4)
r =
    76    79    91     0    19    41     1
    60    82    17     5    19    44    20
    48    44    40    35    60    93    67
    95    45    92    41    13     1    84
    89    61    93    81    27    46    83
    23     1    73    89    20    74    52
```

Operating on Diagonal Matrices

In this section...
"Diagonal Matrix Functions" on page 1-40
"Constructing a Matrix from a Diagonal Vector" on page 1-40
"Returning a Triangular Portion of a Matrix" on page 1-41
"Concatenating Matrices Diagonally" on page 1-41

Diagonal Matrix Functions

There are several MATLAB functions that work specifically on diagonal matrices.

Function	Description
`blkdiag`	Construct a block diagonal matrix from input arguments.
`diag`	Return a diagonal matrix or 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.

Constructing a Matrix from a Diagonal Vector

The `diag` function has two operations that it can perform. You can use it to generate a diagonal matrix:

```
A = diag([12:4:32])
A =
    12     0     0     0     0     0
     0    16     0     0     0     0
     0     0    20     0     0     0
     0     0     0    24     0     0
     0     0     0     0    28     0
     0     0     0     0     0    32
```

You can also use the `diag` function to scan an existing matrix and return the values found along one of the diagonals:

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

diag(A, 2)       % Return contents of second diagonal of A
ans =
     1
    14
    22
```

Returning a Triangular Portion of a Matrix

The `tril` and `triu` functions return a triangular portion of a matrix, the former returning the piece from the lower left and the latter from the upper right. By default, the main diagonal of the matrix divides these two segments. You can use an alternate diagonal by specifying an offset from the main diagonal as a second input argument:

```
A = magic(6);

B = tril(A, -1)
B =
     0     0     0     0     0     0
     3     0     0     0     0     0
    31     9     0     0     0     0
     8    28    33     0     0     0
    30     5    34    12     0     0
     4    36    29    13    18     0
```

Concatenating Matrices Diagonally

You can diagonally concatenate matrices to form a composite matrix using the `blkdiag` function. See “Creating a Block Diagonal Matrix” on page 1-9 for more information on how this works.

Empty Matrices, Scalars, and Vectors

In this section...

"Overview" on page 1-42

"The Empty Matrix" on page 1-43

"Scalars" on page 1-45

"Vectors" on page 1-46

Overview

Although matrices are two dimensional, they do not always appear to have a rectangular shape. A 1-by-8 matrix, for example, has two dimensions yet is linear. These matrices are described in the following sections:

- "The Empty Matrix" on page 1-43

  An *empty matrix* has one of more dimensions that are equal to zero. A two-dimensional matrix with both dimensions equal to zero appears in the MATLAB application as `[]`. The expression `A = []` assigns a 0-by-0 empty matrix to `A`.

- "Scalars" on page 1-45

  A *scalar* is 1-by-1 and appears in MATLAB as a single real or complex number (e.g., `7`, `583.62`, `-3.51`, `5.46097e-14`, `83+4i`).

- "Vectors" on page 1-46

  A *vector* is 1-by-n or n-by-1, and appears in MATLAB as a row or column of real or complex numbers:

```
Column Vector          Row Vector

   53.2                53.2 87.39 4-12i 43.9
   87.39
   4-12i
   43.9
```

The Empty Matrix

A matrix having at least one dimension equal to zero is called an empty matrix. The simplest empty matrix is 0-by-0 in size. Examples of more complex matrices are those of dimension 0-by-5 or 10-by-0.

To create a 0-by-0 matrix, use the square bracket operators with no value specified:

```
A = [];

whos A
  Name      Size         Bytes  Class

  A         0x0              0  double array
```

You can create empty matrices (and arrays) of other sizes using the `zeros`, `ones`, `rand`, or `eye` functions. To create a 0-by-5 matrix, for example, use

```
A = zeros(0,5)
```

Operating on an Empty Matrix

The basic model for empty matrices is that any operation that is defined for `m`-by-`n` matrices, and that produces a result whose dimension is some function of `m` and `n`, should still be allowed when `m` or `n` is zero. The size of the result of this operation is consistent with the size of the result generated when working with nonempty values, but instead is evaluated at zero.

For example, horizontal concatenation

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

requires that `A` and `B` have the same number of rows. So if `A` is `m`-by-`n` and `B` is `m`-by-`p`, then `C` is `m`-by-`(n+p)`. This is still true if `m` or `n` or `p` is zero.

As with all matrices in MATLAB, you must follow the rules concerning compatible dimensions. In the following example, an attempt to add a 1-by-3 matrix to a 0-by-3 empty matrix results in an error:

```
[1 2 3] + ones(0,3)
??? Error using ==> +
Matrix dimensions must agree.
```

Common Operations. The following operations return zero on an empty array:

```
A = [];
size(A), length(A), numel(A), any(A), sum(A)
```

These operations return a nonzero value on an empty array :

```
A = [];
ndims(A), isnumeric(A), isreal(A), isfloat(A), isempty(A), ...
   all(A), prod(A)
```

Using Empty Matrices in Relational Operations

You can use empty matrices in relational operations such as "equal to" (==) or "greater than" (>) as long as both operands have the same dimensions, or the nonempty operand is scalar. The result of any relational operation involving an empty matrix is the empty matrix. Even comparing an empty matrix for equality to itself does not return `true`, but instead yields an empty matrix:

```
x = ones(0,3);
y = x;

y == x
ans =
   Empty matrix: 0-by-3
```

Using Empty Matrices in Logical Operations

MATLAB has two distinct types of logical operators:

- Short-circuit (`&&`, `||`) — Used in testing multiple logical conditions (e.g., `x >= 50 && x < 100`) where each condition evaluates to a scalar `true` or `false`.
- Element-wise (`&`, `|`) — Performs a logical AND, OR, or NOT on each element of a matrix or array.

Short-circuit Operations. The rule for operands used in short-circuit operations is that each operand must be convertible to a logical scalar value. Because of this rule, empty matrices cannot be used in short-circuit logical operations. Such operations return an error.

The only exception is in the case where MATLAB can determine the result of a logical statement without having to evaluate the entire expression. This is true for the following two statements because the result of the entire statements are known by considering just the first term:

```
true || []
ans =
   1

false && []
ans =
   0
```

Elementwise Operations. Unlike the short-circuit operators, all elementwise operations on empty matrices are considered valid as long as the dimensions of the operands agree, or the nonempty operand is scalar. Element-wise operations on empty matrices always return an empty matrix:

```
true | []
ans =
   []
```

Note This behavior is consistent with the way MATLAB does scalar expansion with binary operators, wherein the nonscalar operand determines the size of the result.

Scalars

Any individual real or complex number is represented in MATLAB as a 1-by-1 matrix called a scalar value:

```
A = 5;

ndims(A)        % Check number of dimensions in A
```

```
ans =
     2

size(A)         % Check value of row and column dimensions
ans =
     1     1
```

Use the `isscalar` function to tell if a variable holds a scalar value:

```
isscalar(A)
ans =
    1
```

Vectors

Matrices with one dimension equal to one and the other greater than one are called vectors. Here is an example of a numeric vector:

```
A = [5.73 2-4i 9/7 25e3 .046 sqrt(32) 8j];

size(A)         % Check value of row and column dimensions
ans =
     1     7
```

You can construct a vector out of other vectors, as long as the critical dimensions agree. All components of a row vector must be scalars or other row vectors. Similarly, all components of a column vector must be scalars or other column vectors:

```
A = [29 43 77 9 21];
B = [0 46 11];

C = [A 5 ones(1,3) B]
C =
   29   43   77    9   21    5    1    1    1    0   46   11
```

Concatenating an empty matrix to a vector has no effect on the resulting vector. The empty matrix is ignored in this case:

```
A = [5.36; 7.01; []; 9.44]
A =
```

```
    5.3600
    7.0100
    9.4400
```

Use the `isvector` function to tell if a variable holds a vector:

```
isvector(A)
ans =
    1
```

Full and Sparse Matrices

In this section...
"Overview" on page 1-48
"Sparse Matrix Functions" on page 1-48

Overview

It is not uncommon to have matrices with a large number of zero-valued elements and, because the MATLAB software stores zeros in the same way it stores any other numeric value, these elements can use memory space unnecessarily and can sometimes require extra computing time.

Sparse matrices provide a way to store data that has a large percentage of zero elements more efficiently. While *full matrices* internally store every element in memory regardless of value, *sparse matrices* store only the nonzero elements and their row indices. Using sparse matrices can significantly reduce the amount of memory required for data storage.

You can create sparse matrices for the `double` and `logical` classes. All MATLAB built-in arithmetic, logical, and indexing operations can be applied to sparse matrices, or to mixtures of sparse and full matrices. Operations on sparse matrices return sparse matrices and operations on full matrices return full matrices.

See the section on Sparse Matrices in the MATLAB Mathematics documentation for more information on working with sparse matrices.

Sparse Matrix Functions

This table shows some of the functions most commonly used when working with sparse matrices.

Function	Description
`full`	Convert a sparse matrix to a full matrix.
`issparse`	Determine if a matrix is sparse.

Function	Description
`nnz`	Return the number of nonzero matrix elements.
`nonzeros`	Return the nonzero elements of a matrix.
`nzmax`	Return the amount of storage allocated for nonzero elements.
`spalloc`	Allocate space for a sparse matrix.
`sparse`	Create a sparse matrix or convert full to sparse.
`speye`	Create a sparse identity matrix.
`sprand`	Create a sparse uniformly distributed random matrix.

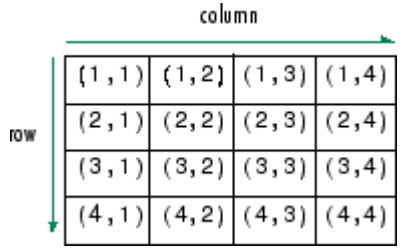
Multidimensional Arrays

In this section...

"Overview" on page 1-50
"Creating Multidimensional Arrays" on page 1-52
"Accessing Multidimensional Array Properties" on page 1-56
"Indexing Multidimensional Arrays" on page 1-56
"Reshaping Multidimensional Arrays" on page 1-60
"Permuting Array Dimensions" on page 1-62
"Computing with Multidimensional Arrays" on page 1-64
"Organizing Data in Multidimensional Arrays" on page 1-65
"Multidimensional Cell Arrays" on page 1-67
"Multidimensional Structure Arrays" on page 1-68

Overview

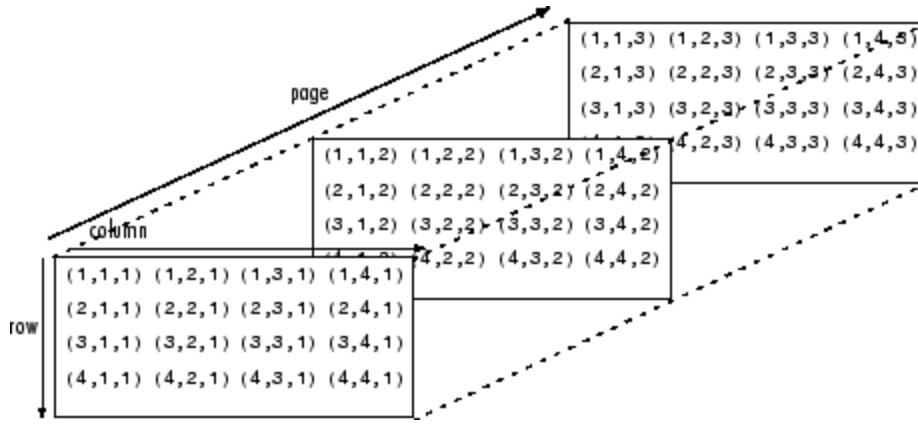
An array having more than two dimensions is called a multidimensional array in the MATLAB application. Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Matrices have two dimensions: the row dimension and the column dimension.



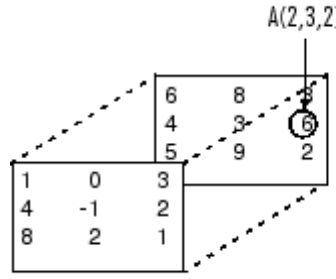
You can access a two-dimensional matrix element with two subscripts: the first representing the row index, and the second representing the column index.

Multidimensional arrays use additional subscripts for indexing. A three-dimensional array, for example, uses three subscripts:

- The first references array dimension 1, the row.
- The second references dimension 2, the column.
- The third references dimension 3. This illustration uses the concept of a *page* to represent dimensions 3 and higher.



To access the element in the second row, third column of page 2, for example, you use the subscripts `(2,3,2)`.




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

     1     0     3
     4    -1     2
     8     2     1

A(:,:,2) =

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

As you add dimensions to an array, you also add subscripts. A four-dimensional array, for example, has four subscripts. The first two

reference a row-column pair; the second two access the third and fourth dimensions of data.

Most of the operations that you can perform on matrices (i.e., two-dimensional arrays) can also be done on multidimensional arrays.

Note The general multidimensional array functions reside in the `datatypes` directory.

Creating Multidimensional Arrays

You can use the same techniques to create multidimensional arrays that you use for two-dimensional matrices. In addition, MATLAB provides a special concatenation function that is useful for building multidimensional arrays.

This section discusses

- "Generating Arrays Using Indexing" on page 1-52
- "Extending Multidimensional Arrays" on page 1-53
- "Generating Arrays Using MATLAB Functions" on page 1-54
- "Building Multidimensional Arrays with the cat Function" on page 1-54

Generating Arrays Using Indexing

One way to create a multidimensional array is to create a two-dimensional array and extend it. For example, begin with a simple two-dimensional array `A`.

```
A = [5 7 8; 0 1 9; 4 3 6];
```

`A` is a 3-by-3 array, that is, its row dimension is 3 and its column dimension is 3. To add a third dimension to `A`,

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

MATLAB responds with

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

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

A(:,:,2) =
     1     0     4
     3     5     6
     9     8     7
```

You can continue to add rows, columns, or pages to the array using similar assignment statements.

Extending Multidimensional Arrays

To extend `A` in any dimension:

- Increment or add the appropriate subscript and assign the desired values.
- Assign the same number of elements to corresponding array dimensions. For numeric arrays, all rows must have the same number of elements, all pages must have the same number of rows and columns, and so on.

You can take advantage of the MATLAB scalar expansion capabilities, together with the colon operator, to fill an entire dimension with a single value:

```
A(:,:,3) = 5;

A(:,:,3)
ans =
     5     5     5
     5     5     5
     5     5     5
```

To turn `A` into a 3-by-3-by-3-by-2, four-dimensional array, enter

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

Note that after the first two assignments MATLAB pads `A` with zeros, as needed, to maintain the corresponding sizes of dimensions.

Generating Arrays Using MATLAB Functions

You can use MATLAB functions such as `randn`, `ones`, and `zeros` to generate multidimensional arrays in the same way you use them for two-dimensional arrays. Each argument you supply represents the size of the corresponding dimension in the resulting array. For example, to create a 4-by-3-by-2 array of normally distributed random numbers:

```
B = randn(4,3,2)
```

To generate an array filled with a single constant value, use the `repmat` function. `repmat` replicates an array (in this case, a 1-by-1 array) through a vector of array dimensions.

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

B(:,:,1) =
     5     5     5     5
     5     5     5     5
     5     5     5     5

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

Note Any dimension of an array can have size zero, making it a form of empty array. For example, 10-by-0-by-20 is a valid size for a multidimensional array.

Building Multidimensional Arrays with the cat Function

The `cat` function is a simple way to build multidimensional arrays; it concatenates a list of arrays along a specified dimension:

```
B = cat(dim, A1, A2...)
```

where A1, A2, and so on are the arrays to concatenate, and dim is the dimension along which to concatenate the arrays.

For example, to create a new array with cat:

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

B(:,:,1) =
     2     8
     0     5

B(:,:,2) =
     1     3
     7     9
```

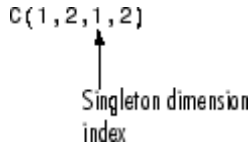
The cat function accepts any combination of existing and new data. In addition, you can nest calls to cat. The lines below, for example, create a four-dimensional array.

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

cat automatically adds subscripts of 1 between dimensions, if necessary. For example, to create a 2-by-2-by-1-by-2 array, enter

```
C = cat(4, [1 2; 4 5], [7 8; 3 2])
```

In the previous case, cat inserts as many singleton dimensions as needed to create a four-dimensional array whose last dimension is not a singleton dimension. If the dim argument had been 5, the previous statement would have produced a 2-by-2-by-1-by-1-by-2 array. This adds additional 1s to indexing expressions for the array. To access the value 8 in the four-dimensional case, use



Accessing Multidimensional Array Properties

You can use the following MATLAB functions to get information about multidimensional arrays you have created.

- `size` — Returns the size of each array dimension.

```
size(C)
ans =
     2      2      1      2
  rows  columns  dim3   dim4
```

- `ndims` — Returns the number of dimensions in the array.

```
ndims(C)
ans =
     4
```

- `whos` — Provides information on the format and storage of the array.

```
whos
Name      Size        Bytes   Class

A         2x2x2          64   double array
B         2x2x2          64   double array
C         4-D            64   double array
D         4-D           192   double array

Grand total is 48 elements using 384 bytes
```

Indexing Multidimensional Arrays

Many of the concepts that apply to two-dimensional matrices extend to multidimensional arrays as well.

To access a single element of a multidimensional array, use integer subscripts. Each subscript indexes a dimension—the first indexes the row dimension, the second indexes the column dimension, the third indexes the first page dimension, and so on.

Consider a 10-by-5-by-3 array `nddata` of random integers:

```
nddata = fix(8 * randn(10,5,3));
```

To access element `(3,2)` on page 2 of `nddata`, for example, use `nddata(3,2,2)`.

You can use vectors as array subscripts. In this case, each vector element must be a valid subscript, that is, within the bounds defined by the dimensions of the array. To access elements `(2,1)`, `(2,3)`, and `(2,4)` on page 3 of `nddata`, use

```
nddata(2,[1 3 4],3);
```

The Colon and Multidimensional Array Indexing

The MATLAB colon indexing extends to multidimensional arrays. For example, to access the entire third column on page 2 of `nddata`, use `nddata(:,3,2)`.

The colon operator is also useful for accessing other subsets of data. For example, `nddata(2:3,2:3,1)` results in a 2-by-2 array, a subset of the data on page 1 of `nddata`. This matrix consists of the data in rows 2 and 3, columns 2 and 3, on the first page of the array.

The colon operator can appear as an array subscript on both sides of an assignment statement. For example, to create a 4-by-4 array of zeros:

```
C = zeros(4, 4)
```

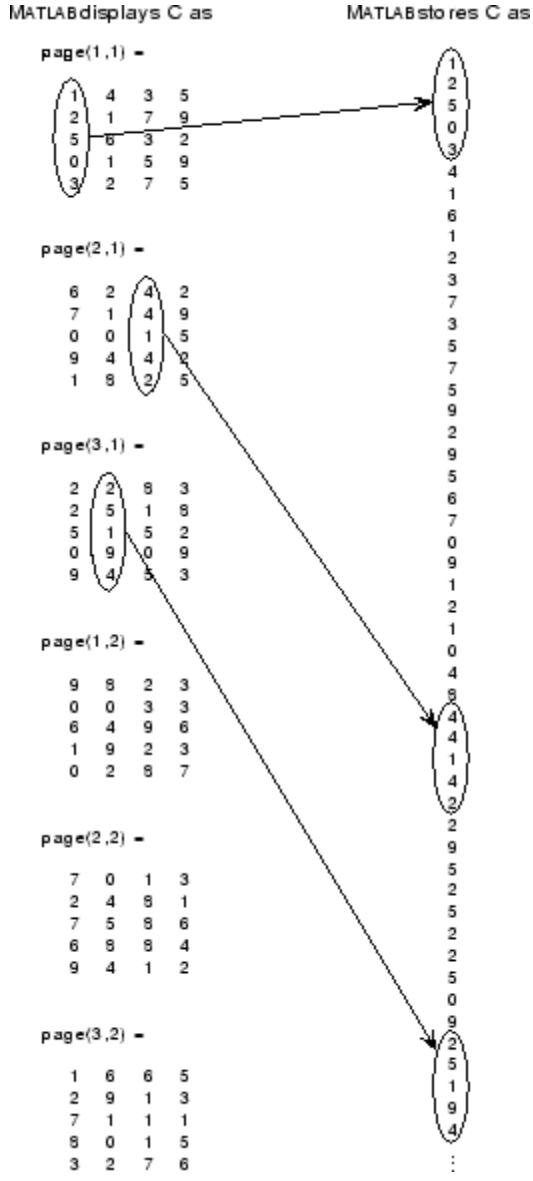
Now assign a 2-by-2 subset of array `nddata` to the four elements in the center of `C`.

```
C(2:3,2:3) = nddata(2:3,1:2,2)
```

Linear Indexing with Multidimensional Arrays

MATLAB linear indexing also extends to multidimensional arrays. In this case, MATLAB operates on a page-by-page basis to create the storage column, again appending elements columnwise. See “Linear Indexing” on page 1-13 for an introduction to this topic.

For example, consider a 5-by-4-by-3-by-2 array `C`.



Again, a single subscript indexes directly into this column. For example, `C(4)` produces the result

```
ans =
     0
```

If you specify two subscripts `(i,j)` indicating row-column indices, MATLAB calculates the offset as described above. Two subscripts always access the first page of a multidimensional array, provided they are within the range of the original array dimensions.

If more than one subscript is present, all subscripts must conform to the original array dimensions. For example, `C(6,2)` is invalid because all pages of `C` have only five rows.

If you specify more than two subscripts, MATLAB extends its indexing scheme accordingly. For example, consider four subscripts `(i,j,k,l)` into a four-dimensional array with size `[d1 d2 d3 d4]`. MATLAB calculates the offset into the storage column by

```
(l-1)(d3)(d2)(d1)+(k-1)(d2)(d1)+(j-1)(d1)+i
```

For example, if you index the array `C` using subscripts (3, 4, 2, 1), MATLAB returns the value 5 (index 38 in the storage column).

In general, the offset formula for an array with dimensions $[d_1\ d_2\ d_3\ \ldots\ d_n]$ using any subscripts $(s_1\ s_2\ s_3\ \ldots\ s_n)$ is

$$(s_n-1)(d_{n-1})(d_{n-2})\ldots(d_1)+(s_{n-1}-1)(d_{n-2})\ldots(d_1)+\ldots+(s_2-1)(d_1)+s_1$$

Because of this scheme, you can index an array using any number of subscripts. You can append any number of `1`s to the subscript list because these terms become zero. For example,

```
C(3,2,1,1,1,1,1,1)
```

is equivalent to

```
C(3,2)
```

Avoiding Ambiguity in Multidimensional Indexing

Some assignment statements, such as

```
A(:,:,2) = 1:10
```

are ambiguous because they do not provide enough information about the shape of the dimension to receive the data. In the case above, the statement tries to assign a one-dimensional vector to a two-dimensional destination. MATLAB produces an error for such cases. To resolve the ambiguity, be sure you provide enough information about the destination for the assigned data, and that both data and destination have the same shape. For example:

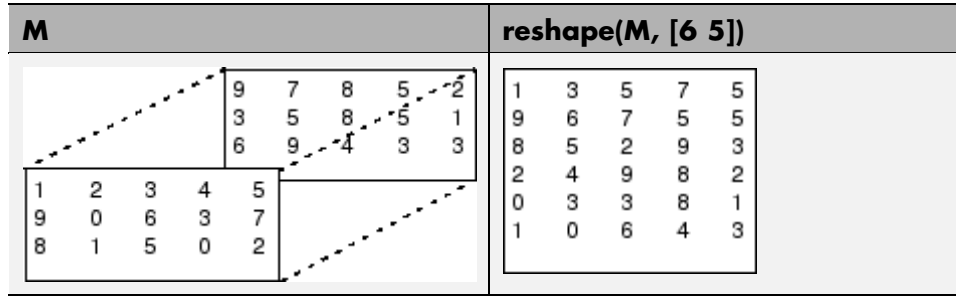
```
A(1,:,2) = 1:10;
```

Reshaping Multidimensional Arrays

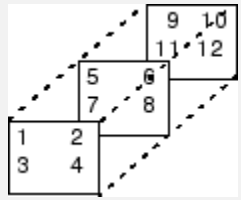
Unless you change its shape or size, a MATLAB array retains the dimensions specified at its creation. You change array size by adding or deleting elements. You change array shape by respecifying the array's row, column, or page dimensions while retaining the same elements. The `reshape` function performs the latter operation. For multidimensional arrays, its form is

```
B = reshape(A,[s1 s2 s3 ...])
```

`s1`, `s2`, and so on represent the desired size for each dimension of the reshaped matrix. Note that a reshaped array must have the same number of elements as the original array (that is, the product of the dimension sizes is constant).



The `reshape` function operates in a columnwise manner. It creates the reshaped matrix by taking consecutive elements down each column of the original data construct.

C	reshape(C, [6 2])
	1 6 3 8 2 9 4 11 5 10 7 12

Here are several new arrays from reshaping `nddata`:

```
B = reshape(nddata, [6 25])
C = reshape(nddata, [5 3 10])
D = reshape(nddata, [5 3 2 5])
```

Removing Singleton Dimensions

MATLAB creates singleton dimensions if you explicitly specify them when you create or reshape an array, or if you perform a calculation that results in an array dimension of one:

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

size(B)
ans =
     2     3     1     4
```

The `squeeze` function removes singleton dimensions from an array:

```
C = squeeze(B);

size(C)
ans =
     2     3     4
```

The `squeeze` function does not affect two-dimensional arrays; row vectors remain rows.

Permuting Array Dimensions

The `permute` function reorders the dimensions of an array:

```
B = permute(A, dims);
```

`dims` is a vector specifying the new order for the dimensions of `A`, where 1 corresponds to the first dimension (rows), 2 corresponds to the second dimension (columns), 3 corresponds to pages, and so on.

A

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

     1     2     3
     4     5     6
     7     8     9

A(:,:,2) =

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

B = permute(A, [2 1 3])

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

     1     4     7
     2     5     8
     3     6     9

B(:,:,2) =

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

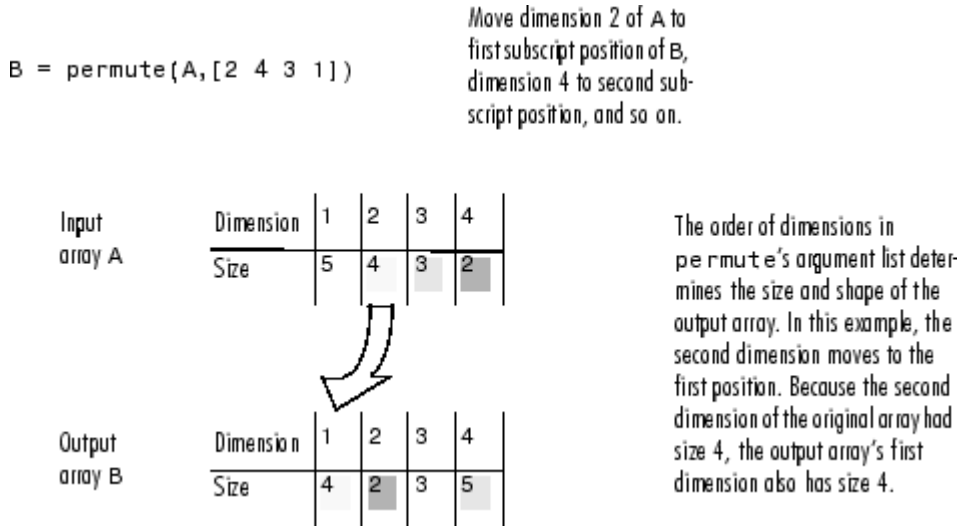
Row and column subscripts are reversed (page-by-page transposition).

C = permute(A, [3 2 1])

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

     1     2     3
     0     5     4

C(:,:,2) =

     4     5     6
     2     7     6

C(:,:,3) =

     7     8     9
     9     3     1
```

Row and page subscripts are reversed.

For a more detailed look at the `permute` function, consider a four-dimensional array `A` of size 5-by-4-by-3-by-2. Rearrange the dimensions, placing the column dimension first, followed by the second page dimension, the first page dimension, then the row dimension. The result is a 4-by-2-by-3-by-5 array.



You can think of `permute`'s operation as an extension of the `transpose` function, which switches the row and column dimensions of a matrix. For `permute`, the order of the input dimension list determines the reordering of the subscripts. In the example above, element `(4,2,1,2)` of `A` becomes element `(2,2,1,4)` of `B`, element `(5,4,3,2)` of `A` becomes element `(4,2,3,5)` of `B`, and so on.

Inverse Permutation

The `ipermute` function is the inverse of `permute`. Given an input array `A` and a vector of dimensions `v`, `ipermute` produces an array `B` such that `permute(B,v)` returns `A`.

For example, these statements create an array `E` that is equal to the input array `C`:

```
D = ipermute(C, [1 4 2 3]);
E = permute(D, [1 4 2 3])
```

You can obtain the original array after permuting it by calling `ipermute` with the same vector of dimensions.

Computing with Multidimensional Arrays

Many of the MATLAB computational and mathematical functions accept multidimensional arrays as arguments. These functions operate on specific dimensions of multidimensional arrays; that is, they operate on individual elements, on vectors, or on matrices.

Operating on Vectors

Functions that operate on vectors, like `sum`, `mean`, and so on, by default typically work on the first nonsingleton dimension of a multidimensional array. Most of these functions optionally let you specify a particular dimension on which to operate. There are exceptions, however. For example, the `cross` function, which finds the cross product of two vectors, works on the first nonsingleton dimension having length 3.

Note In many cases, these functions have other restrictions on the input arguments — for example, some functions that accept multiple arrays require that the arrays be the same size. Refer to the online help for details on function arguments.

Operating Element-by-Element

MATLAB functions that operate element-by-element on two-dimensional arrays, like the trigonometric and exponential functions in the `elfun` directory, work in exactly the same way for multidimensional cases. For example, the `sin` function returns an array the same size as the function's input argument. Each element of the output array is the sine of the corresponding element of the input array.

Similarly, the arithmetic, logical, and relational operators all work with corresponding elements of multidimensional arrays that are the same size in every dimension. If one operand is a scalar and one an array, the operator applies the scalar to each element of the array.

Operating on Planes and Matrices

Functions that operate on planes or matrices, such as the linear algebra and matrix functions in the `matfun` directory, do not accept multidimensional

arrays as arguments. That is, you cannot use the functions in the `matfun` directory, or the array operators *, ^, \, or /, with multidimensional arguments. Supplying multidimensional arguments or operands in these cases results in an error.

You can use indexing to apply a matrix function or operator to matrices within a multidimensional array. For example, create a three-dimensional array `A`:

```
A = cat(3, [1 2 3; 9 8 7; 4 6 5], [0 3 2; 8 8 4; 5 3 5], ...
    [6 4 7; 6 8 5; 5 4 3]);
```

Applying the `eig` function to the entire multidimensional array results in an error:

```
eig(A)
??? Error using ==> eig
Input arguments must be 2-D.
```

You can, however, apply `eig` to planes within the array. For example, use colon notation to index just one page (in this case, the second) of the array:

```
eig(A(:,:,2))
ans =
   12.9129
   -2.6260
    2.7131
```

Note In the first case, subscripts are not colons; you must use `squeeze` to avoid an error. For example, `eig(A(2,:,:))` results in an error because the size of the input is `[1 3 3]`. The expression `eig(squeeze(A(2,:,:)))`, however, passes a valid two-dimensional matrix to `eig`.

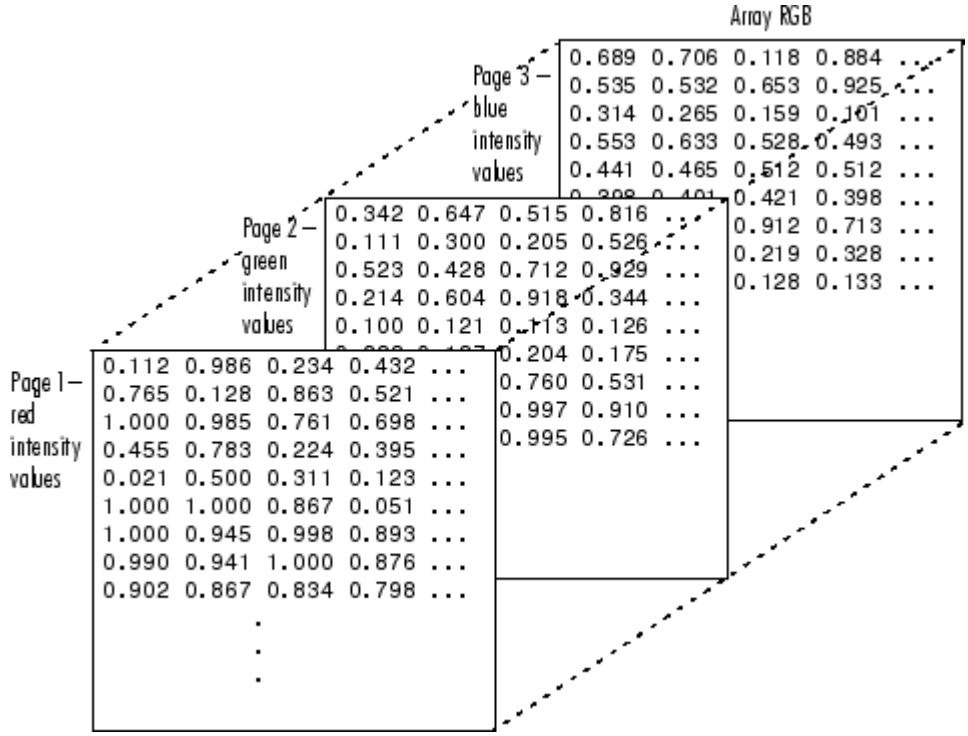
Organizing Data in Multidimensional Arrays

You can use multidimensional arrays to represent data in two ways:

- As planes or pages of two-dimensional data. You can then treat these pages as matrices.

- As multivariate or multidimensional data. For example, you might have a four-dimensional array where each element corresponds to either a temperature or air pressure measurement taken at one of a set of equally spaced points in a room.

For example, consider an RGB image. For a single image, a multidimensional array is probably the easiest way to store and access data.



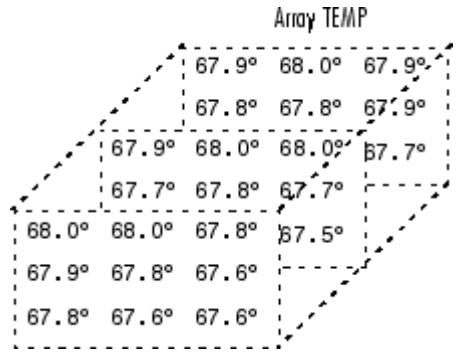
To access an entire plane of the image, use

```
redPlane = RGB(:,:,1);
```

To access a subimage, use

```
subimage = RGB(20:40,50:85,:);
```

The RGB image is a good example of data that needs to be accessed in planes for operations like display or filtering. In other instances, however, the data itself might be multidimensional. For example, consider a set of temperature measurements taken at equally spaced points in a room. Here the location of each value is an integral part of the data set—the physical placement in three-space of each element is an aspect of the information. Such data also lends itself to representation as a multidimensional array.



Now to find the average of all the measurements, use

```
mean(mean(mean(TEMP)));
```

To obtain a vector of the “middle” values (element (2,2)) in the room on each page, use

```
B = TEMP(2,2,:);
```

Multidimensional Cell Arrays

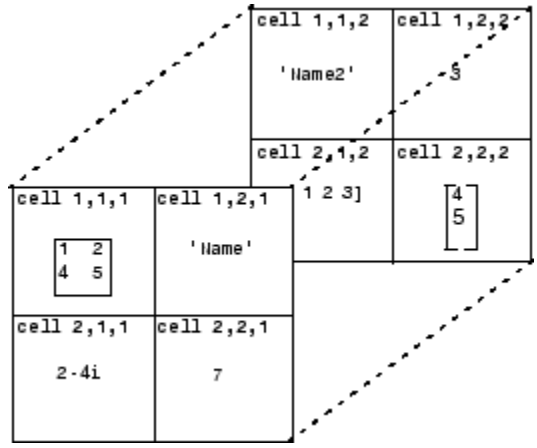
Like numeric arrays, the framework for multidimensional cell arrays in MATLAB is an extension of the two-dimensional cell array model. You can use the `cat` function to build multidimensional cell arrays, just as you use it for numeric arrays.

For example, create a simple three-dimensional cell array `C`:

```
A{1,1} = [1 2;4 5];
A{1,2} = 'Name';
A{2,1} = 2-4i;
```

```
A{2,2} = 7;
B{1,1} = 'Name2';
B{1,2} = 3;
B{2,1} = 0:1:3;
B{2,2} = [4 5]';
C = cat(3, A, B);
```

The subscripts for the cells of `C` look like

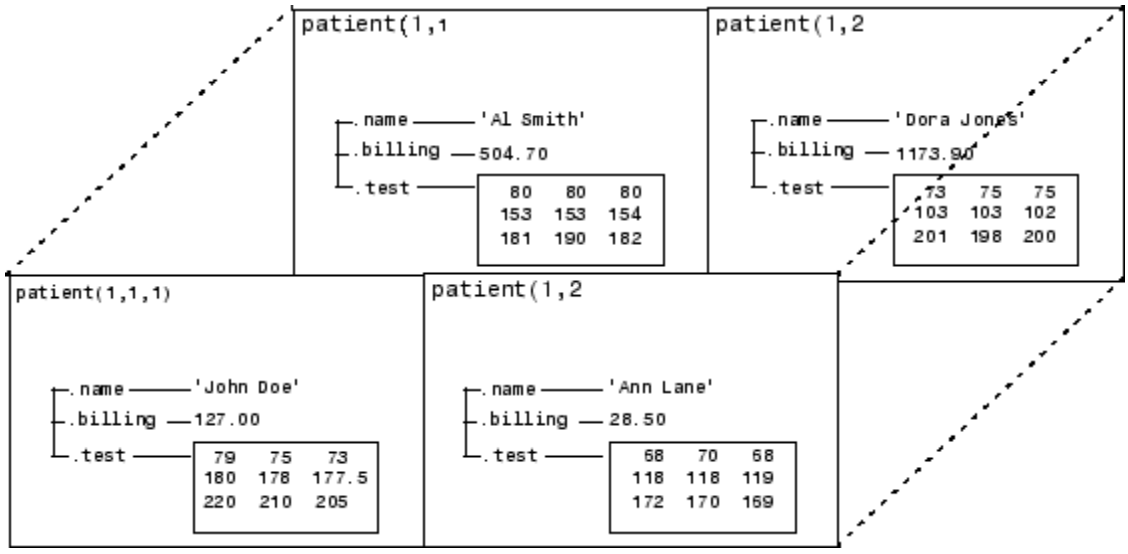


Multidimensional Structure Arrays

Multidimensional structure arrays are extensions of rectangular structure arrays. Like other types of multidimensional arrays, you can build them using direct assignment or the `cat` function:

```
patient(1,1,1).name = 'John Doe';
patient(1,1,1).billing = 127.00;
patient(1,1,1).test = [79 75 73; 180 178 177.5; 220 210 205];
patient(1,2,1).name = 'Ann Lane';
patient(1,2,1).billing = 28.50;
patient(1,2,1).test = [68 70 68; 118 118 119; 172 170 169];
patient(1,1,2).name = 'Al Smith';
patient(1,1,2).billing = 504.70;
patient(1,1,2).test = [80 80 80; 153 153 154; 181 190 182];
patient(1,2,2).name = 'Dora Jones';
```

```
patient(1,2,2).billing = 1173.90;
patient(1,2,2).test = [73 73 75; 103 103 102; 201 198 200];
```



Applying Functions to Multidimensional Structure Arrays

To apply functions to multidimensional structure arrays, operate on fields and field elements using indexing. For example, find the sum of the columns of the `test` array in `patient(1,1,2)`:

```
sum((patient(1,1,2).test));
```

Similarly, add all the `billing` fields in the `patient` array:

```
total = sum([patient.billing]);
```

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.
`flipdim`	Flip a matrix along the specified dimension.
`fliplr`	Flip a matrix about a vertical axis.

Functions to Modify the Shape of a Matrix (Continued)

Function	Description
`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 characters or strings.
`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.

Functions to Determine Class (Continued)

Function	Description
`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

- "MATLAB Linear Algebra Functions" on page 2-2
- "Matrices in the MATLAB Environment" on page 2-5
- "Systems of Linear Equations" on page 2-15
- "Inverses and Determinants" on page 2-26
- "Factorizations" on page 2-31
- "Powers and Exponentials" on page 2-40
- "Eigenvalues" on page 2-44
- "Singular Values" on page 2-48

MATLAB Linear Algebra Functions

The MATLAB `matfun` directory contains linear algebra functions. For a complete list, brief descriptions, and links to reference pages, type:

```
help matfun
```

The following table lists the MATLAB linear algebra functions by category.

Function Summary

Category	Function	Description
Matrix analysis	`norm`	Matrix or vector norm
	`normest`	Estimate the matrix 2-norm
	`rank`	Matrix rank
	`det`	Determinant
	`trace`	Sum of diagonal elements
	`null`	Null space
	`orth`	Orthogonalization
	`rref`	Reduced row echelon form
	`subspace`	Angle between two subspaces

Function Summary (Continued)

Category	Function	Description
Linear equations	\ and /	Linear equation solution
	`inv`	Matrix inverse
	`cond`	Condition number for inversion
	`condest`	1-norm condition number estimate
	`chol`	Cholesky factorization
	`cholinc`	Incomplete Cholesky factorization
	`linsolve`	Solve a system of linear equations
	`lu`	LU factorization
	`ilu`	Incomplete LU factorization
	`luinc`	Incomplete LU factorization
	`qr`	Orthogonal-triangular decomposition
	`lsqnonneg`	Nonnegative least-squares
	`pinv`	Pseudoinverse
	`lscov`	Least squares with known covariance

Function Summary (Continued)

Category	Function	Description
Eigenvalues and singular values	`eig`	Eigenvalues and eigenvectors
	`svd`	Singular value decomposition
	`eigs`	A few eigenvalues
	`svds`	A few singular values
	`poly`	Characteristic polynomial
	`polyeig`	Polynomial eigenvalue problem
	`condeig`	Condition number for eigenvalues
	`hess`	Hessenberg form
	`qz`	QZ factorization
	`schur`	Schur decomposition
Matrix functions	`expm`	Matrix exponential
	`logm`	Matrix logarithm
	`sqrtm`	Matrix square root
	`funm`	Evaluate general matrix function

Matrices in the MATLAB Environment

In this section...

"Creating Matrices" on page 2-5

"Adding and Subtracting Matrices" on page 2-7

"Vector Products and Transpose" on page 2-7

"Multiplying Matrices" on page 2-9

"Identity Matrix" on page 2-11

"Kronecker Tensor Product" on page 2-12

"Vector and Matrix Norms" on page 2-13

"Using Multithreaded Computation with Linear Algebra Functions" on page 2-14

Creating Matrices

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.

Symbolic Math Toolbox™ software extends the capabilities of MATLAB software to matrices of mathematical expressions.

MATLAB has dozens of functions that create different kinds of matrices. There are two functions you can use to create a pair of 3-by-3 example matrices for use throughout this chapter. The first example is symmetric:

```
A = pascal(3)

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

The second example is not symmetric:

```
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:

```
C = fix(10*rand(3,2))

C =
       9     4
       2     8
       6     7
```

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. The statements

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

v = [2 0 -1]

s = 7
```

produce a column vector, a row vector, and a scalar:

```
u =
       3
       1
       4

v =
       2     0    -1

s =
       7
```

For more information about creating and working with matrices, see in MATLAB Programming Fundamentals.

Adding and Subtracting Matrices

Addition and subtraction of matrices is defined just as it is for arrays, element by element. Adding `A` to `B` and then subtracting `A` from the result recovers `B`:

```
A = pascal(3);
B = magic(3);
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 the same dimension, or one of them be a scalar. If the dimensions are incompatible, an error results:

```
C = fix(10*rand(3,2))
X = A + C
Error using ==> +
Matrix dimensions must agree.

w = v + s

w =
       9     7     6
```

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, the *inner* product, or a matrix, 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}. MATLAB uses the apostrophe operator (') to perform a complex conjugate transpose, and uses 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` is *symmetric*, so `A'` is equal to `A`. But `B` is not symmetric:

```
B = magic(3);
X = B'

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

Transposition turns a row vector into a column vector:

```
x = v'

x =
       2
       0
      -1
```

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

```
x'*y
```

and

```
y'*x
```

are the same scalar. This quantity is used so frequently, it has three different names: *inner* product, *scalar* product, or *dot* product.

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. So if

```
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 + 9.0000i   4.0000 + 7.0000i
```

then

```
z'
ans =
   1.0000 - 2.0000i   6.0000 + 2.0000i
   7.0000 + 3.0000i        0 - 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 + 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 a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is not commutative; AB is usually not equal to BA:

```
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:

```
C = fix(10*rand(3,2));
X = A*C

X =
      17    19
      31    41
      51    70

Y = C*A

Error using ==> *
Inner matrix dimensions must agree.
```

Anything can be multiplied by a scalar:

```
s = 7;
w = s*v

w =
      14     0    -7
```

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.

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 in the following order:

```
[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 the two matrices

```
kron(X,I)
```

and

```
kron(I,X)
```

are

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

and

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

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 is defined by 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*:

```
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 $p = 2$:

```
C = fix(10*rand(3,2));
[norm(C,1) norm(C) norm(C,inf)]

ans =
     19.0000   14.8015   13.0000
```

Using Multithreaded Computation with Linear Algebra Functions

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 than 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, complex 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. To learn more about mulithreading, see in the MATLAB Programming Fundamentals documentation.

Systems of Linear Equations

In this section...

"Computational Considerations" on page 2-15

"The mldivide Algorithm" on page 2-17

"General Solution" on page 2-18

"Square Systems" on page 2-18

"Overdetermined Systems" on page 2-21

"Using Multithreaded Computation with Systems of Linear Equations" on page 2-24

"Iterative Methods for Solving Systems of Linear Equations" on page 2-25

Computational Considerations

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

In matrix notation, the general problem takes the following form: Given two matrices A and B, does there exist a unique matrix X so that $AX = B$ or $XA = B$?

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

$$7x = 21$$

have a unique solution?

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

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

The solution is *not* ordinarily obtained by computing the inverse of 7, that is $7^{-1} = 0.142857...$, and then multiplying 7^{-1} by 21. This would be more work and, if 7^{-1} is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown;

the MATLAB software 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 `mldivide` and `mrdivide`. `mldivide` and `mrdivide` 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 $XA = B$.
`X = A\B`	Denotes the solution to the matrix equation $AX = B$.

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` is m-by-n, there are three cases:

m = n	Square system. Seek an exact solution.
m > n	Overdetermined system. Find a least squares solution.
m < n	Underdetermined system. Find a basic solution with at most *m* nonzero components.

The mldivide Algorithm

The `mldivide` operator employs different algorithms to handle different kinds of coefficient matrices. The various cases are diagnosed automatically by examining the coefficient matrix.

Permutations of Triangular Matrices

`mldivide` checks for triangularity by testing for zero elements. If a matrix `A` is triangular, MATLAB software uses a substitution to compute the solution vector `x`. If `A` is a permutation of a triangular matrix, MATLAB software uses a permuted substitution algorithm.

Square Matrices

If A is symmetric and has real, positive diagonal elements, MATLAB attempts a Cholesky factorization. If the Cholesky factorization fails, MATLAB performs a symmetric, indefinite factorization. If `A` is upper Hessenberg, MATLAB uses Gaussian elimination to reduce the system to a triangular matrix. If `A` is square but is neither permuted triangular, symmetric and positive definite, or Hessenberg, MATLAB performs a general triangular factorization using LU factorization with partial pivoting (see `lu`).

Rectangular Matrices

If `A` is rectangular, `mldivide` returns a least-squares solution. MATLAB solves overdetermined systems with QR factorization (see `qr`). For an underdetermined system, MATLAB returns the solution with the maximum number of zero elements.

The `mldivide` function reference page contains a more detailed description of the algorithm.

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, the solution, `x = A\b`, is then 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 has a determinant equal to 1.

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 and raises an error condition 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
```

`P` is a pseudoinverse of A. If $AX = b$ does not have an exact solution, `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

```
det(A)

ans =
     0
```

Note For information about using `pinv` to solve systems with rectangular coefficient matrices, see "Pseudoinverses" on page 2-27.

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. On the other hand, 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

Overdetermined systems of simultaneous linear equations are often encountered in various kinds of curve fitting to experimental data. Here is a hypothetical example. A quantity y is measured at several different values of time, t, to produce the following observations:

t	y
0.0	0.82
0.3	0.72
0.8	0.63
1.1	0.60
1.6	0.55
2.3	0.50

Enter the data into MATLAB with the statements

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

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 the constant vector containing all ones and the other the vector with components e^{-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 the 6-by-2 matrix:

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

E =
      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 =
      0.4760
      0.3413
```

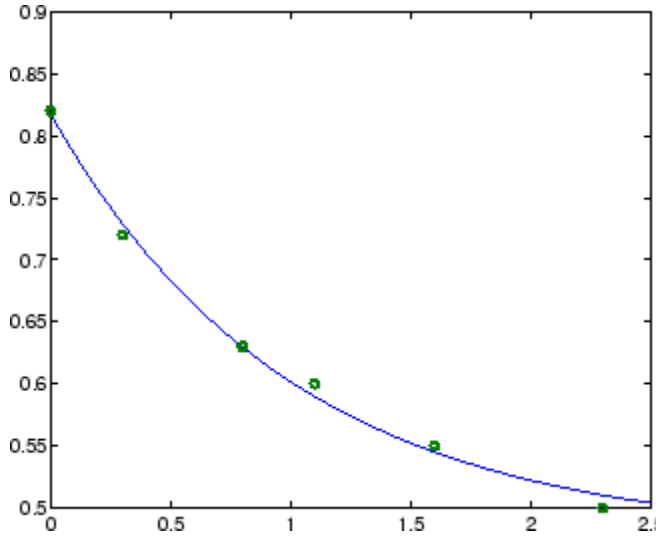
In other words, the least-squares fit to the data is

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

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, the least-squares solution to *AX* = *B* is not unique. The backslash operator, `A\B`, issues a warning if `A` is rank deficient and produces a least-use the `format` command to display the solution in *rational* format. The particular solution is obtained with

```
format rat
p = R\b
p =
        0
       -3/7
        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 solution to the underdetermined system can be characterized by adding an arbitrary vector from the null space, 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 any vector `x` where

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

for an arbitrary vector `q` satisfies `R*x = b`.

Using Multithreaded Computation with Systems of Linear Equations

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 than 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, complex 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. To learn more about mulithreading, see in the MATLAB Programming Fundamentals documentation.

Iterative Methods for Solving Systems of Linear Equations

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.

pcg
: Preconditioned conjugate gradients method. This method is appropriate for Hermitian positive definite coefficient matrix A.

bicg
: BiConjugate Gradients Method

bicgstab
: BiConjugate Gradients Stabilized Method

bicgstabl
: BiCGStab(l) Method

cgs
: Conjugate Gradients Squared Method

gmres
: Generalized Minimum Residual Method

lsqr
: LSQR Method

minres
: Minimum Residual Method. This method is appropriate for Hermitian coefficient matrix A.

qmr
: Quasi-Minimal Residual Method

symmlq
: Symmetric LQ Method

tfqmr
: Transpose-Free QMR Method

Inverses and Determinants

In this section...

"Introduction" on page 2-26

"Pseudoinverses" on page 2-27

Introduction

If A is square and nonsingular, the equations $AX = I$ and $XA = I$ have the same solution, X. This solution is called the inverse of A, is denoted by A^{-1}, and is computed by the function `inv`.

The *determinant* of a matrix is useful in theoretical considerations and some types of symbolic computation, but its scaling and roundoff error properties make it far less satisfactory for numeric computation. Nevertheless, the function `det` computes the determinant of a square matrix:

```
A = pascal(3)

A =
       1     1     1
       1     2     3
       1     3     6
d = det(A)
X = inv(A)

d =
       1

X =
       3    -3     1
      -3     5    -2
       1    -2     1
```

Again, because `A` is symmetric, has integer elements, and has determinant equal to one, so does its inverse. On the other hand,

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

```
B =
       8     1     6
       3     5     7
       4     9     2
d = det(B)
X = inv(B)

d =
     -360

X =
      0.1472   -0.1444    0.0639
     -0.0611    0.0222    0.1056
     -0.0194    0.1889   -0.1028
```

Closer examination of the elements of `X`, or use of `format rat`, would reveal that they are integers divided by 360.

If `A` is square and nonsingular, then, without roundoff error, `X = inv(A)*B` is theoretically the same as `X = A\B` and `Y = B*inv(A)` is theoretically the same as `Y = B/A`. But the computations involving the backslash and slash operators are preferable because they require less computer time, less memory, and have better error-detection properties.

Pseudoinverses

Rectangular matrices do not have inverses or determinants. At least one of the equations $AX = I$ and $XA = I$ does not have a solution. A partial replacement for the inverse is provided by the *Moore-Penrose pseudoinverse*, which is computed by the `pinv` function:

```
format short
C = fix(10*gallery('uniformdata',[3 2],0));
X = pinv(C)

X =
    0.1159   -0.0729    0.0171
   -0.0534    0.1152    0.0418
```

The matrix

```
Q = X*C

Q =
    1.0000    0.0000
    0.0000    1.0000
```

is the 2-by-2 identity, but the matrix

```
P = C*X

P =
    0.8293   -0.1958    0.3213
   -0.1958    0.7754    0.3685
    0.3213    0.3685    0.3952
```

is not the 3-by-3 identity. However, `P` acts like an identity on a portion of the space in the sense that `P` is symmetric, `P*C` is equal to `C`, and `X*P` is equal to `X`.

Solving a Rank-Deficient System

If `A` is m-by-n with $m > n$ and full rank n, each of the three statements

```
x = A\b
x = pinv(A)*b
x = inv(A'*A)*A'*b
```

theoretically computes the same least-squares solution `x`, although the backslash operator does it faster.

However, if `A` does not have full rank, the solution to the least-squares problem is not unique. There are many vectors `x` that minimize

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

The solution computed by `x = A\b` is a basic solution; it has at most r nonzero components, where r is the rank of `A`. The solution computed by `x = pinv(A)*b` is the minimal norm solution because it minimizes `norm(x)`. An attempt to compute a solution with `x = inv(A'*A)*A'*b` fails because `A'*A` is singular.

Here is an example that illustrates the various solutions:

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

does not have full rank. Its second column is the average of the first and third columns. If

```
b = A(:,2)
```

is the second column, then an obvious solution to `A*x = b` is `x = [0 1 0]'`. But none of the approaches computes that `x`. The backslash operator gives

```
x = A\b

Warning: Rank deficient, rank = 2.

x =
      0.5000
      0
      0.5000
```

This solution has two nonzero components. The pseudoinverse approach gives

```
y = pinv(A)*b

y =
      0.3333
      0.3333
      0.3333
```

There is no warning about rank deficiency. But `norm(y) = 0.5774` is less than `norm(x) = 0.7071`. Finally,

```
z = inv(A'*A)*A'*b
```

fails completely:

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 9.868649e-018.
z =
   -0.8594
```

```
    1.3438
   -0.6875
```

Factorizations

In this section...

"Introduction" on page 2-31

"Cholesky Factorization" on page 2-31

"LU Factorization" on page 2-33

"QR Factorization" on page 2-34

"Using Multithreaded Computation for Factorization" on page 2-38

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 offdiagonal 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 may 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 may 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-nQ 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, complex 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). To learn more about mulithreading, see in the MATLAB Programming Fundamentals documentation.

Powers and Exponentials

In this section...

"Positive Integer Powers" on page 2-40

"Inverse and Fractional Powers" on page 2-40

"Element-by-Element Powers" on page 2-41

"Exponentials" on page 2-41

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
```

and the initial condition, $x(0)$

```
x0 =
     1
     1
     1
```

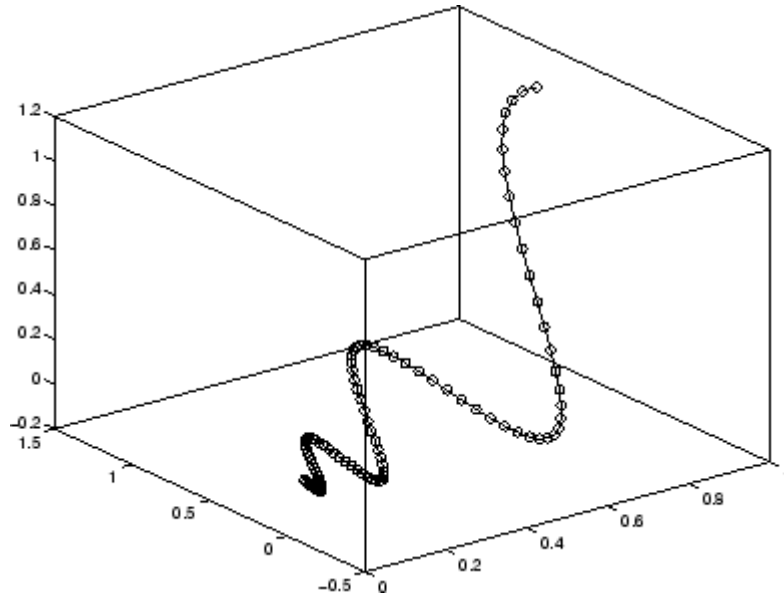
The matrix exponential is used to compute the solution, $x(t)$, to the differential equation at 101 points on the interval $0 \le t \le 1$ with

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

A three-dimensional phase plane plot obtained with

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

shows the solution spiraling in towards the origin. This behavior is related to the eigenvalues of the coefficient matrix, which are discussed in the next section.


1.2
1
0.8
0.6
0.4
0.2
0
−0.2
1.5
1
0.5
0
−0.5
0
0.2
0.4
0.6
0.8
1

Eigenvalues

In this section...

"Eigenvalue Decomposition" on page 2-44
"Multiple Eigenvalues" on page 2-45
"Schur Decomposition" on page 2-46

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 provided by the coefficient matrix of the ordinary differential equation in the previous section:

```
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 roundoff error of `A`. And, `inv(V)*A*V`, or `V\A*V`, is within roundoff error of `D`.

Multiple Eigenvalues

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

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

For this matrix

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

produces

```
V =

   -0.4741   -0.4082   -0.4082
    0.8127    0.8165    0.8165
   -0.3386   -0.4082   -0.4082

D =

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

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

Schur Decomposition

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

$$A = USU^T.$$

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 a basis with much better numerical properties than a set of eigenvectors. The Schur decomposition of this defective example is

```
[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     1.0000
```

The double eigenvalue is contained in the lower 2-by-2 block of `S`.

Note If `A` is complex, `schur` returns the complex Schur form, which is upper triangular with the eigenvalues of `A` on the diagonal.

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

$$Av = \sigma u$$
$$A^T u = \sigma v.$$

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 have

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

Since U and V are orthogonal, this becomes the singular value decomposition

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

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, the resulting U can be quite large, but most of its columns are multiplied by zeros in Σ. In this situation, the *economy* sized decomposition saves both time and storage by producing an m-by-n U, an n-by-n Σ and the same V.

The eigenvalue decomposition is the appropriate tool for analyzing a matrix when it represents a mapping from a vector space into itself, as it does for an ordinary differential equation. On the other hand, 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.

3

Random Numbers

- "Generating Random Numbers" on page 3-2
- "Managing the Default Stream" on page 3-3
- "Creating and Controlling a Random Number Stream" on page 3-10
- "Multiple streams" on page 3-19
- "Legacy Mode" on page 3-22
- "Selected Bibliography" on page 3-25

Pseudorandom numbers are generated by deterministic algorithms. They are "random" in the sense that, on average, they pass statistical tests regarding their distribution and correlation. They differ from true random numbers in that they are generated by an algorithm, rather than a truly random process. *Random number generators* (RNGs), like those in MATLAB are algorithms for generating pseudorandom numbers with a specified distribution. A given number may be repeated many times during the sequence, *but the entire sequence is not repeated*.

A pseudorandom sequence is described by several properties. A random *stream* is the sequence of values that are returned by a generator. The *period* of a sequence is its length, that is, the number of values it generates before the entire sequence is repeated. The *state* is the information the generator keeps internally in order to create successive values in the stream. The *seed* of a sequence is a single value to define its starting point.

Generating Random Numbers

The `rand` function returns uniformly distributed random numbers between 0 and 1.

```
rand(1,5)

ans =

    0.8147    0.9058    0.1270    0.9134    0.6324
```

MATLAB software initializes a default stream at startup. The `rand` function accesses the default stream and draws a set of numbers to create the output. This means that every time `rand` is called, the state of the default stream is changed and the output is different.

```
A=rand(1,5);B=rand(1,5);
A,B

A =

    0.0975    0.2785    0.5469    0.9575    0.9649

B =

    0.1576    0.9706    0.9572    0.4854    0.8003
```

- `rand` returns pseudorandom numbers from a uniform distribution.
- `randn` returns pseudorandom numbers from a normal distribution.
- `randi` returns pseudorandom numbers from a uniform discrete distribution.

Managing the Default Stream

`rand`, `randn`, and `randi` draw random numbers from an underlying random number stream, called the default stream. The `@RandStream` class allows you to get a handle to the default stream and control random number generation.

Get a handle to the default stream as follows:

```
defaultStream=RandStream.getDefaultStream

defaultStream =

mt19937ar random stream (current default)
             Seed: 0
         RandnAlg: Ziggurat
```

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

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

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

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

Use the `randn` and `randi` functions to generate normal random values and integer random values from the default 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 `defaultStream`.

```
myState=defaultStream.State;
```

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

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

ans =

     1
```

`rand`, `randi`, and `randn` access the default 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.

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

ans =

     0
```

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

```
stream1=RandStream.getDefaultStream;
stream2=RandStream.getDefaultStream;
stream1.RandnAlg='Polar';
stream2.RandnAlg
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.getDefaultStream`	Get the default random number stream
`RandStream.setDefaultStream`	Set the default 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.
`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.
`RandnAlg`	Algorithm used by `randn(s, ...)` to generate normal pseudorandom values. Possible values are `'Ziggurat'`, `'Polar'`, or `'Inversion'`.

Property	Description
`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 default stream.

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

ans =

     1
```

Random Number Data Types

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

```
defaultStream=RandStream.getDefaultStream;
myState=defaultStream.State;
A=rand(1,5);
class(A)

ans =

double
```

To specify the class as double explicitly:

```
defaultStream.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.

```
defaultStream.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

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 default 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 default stream. You can even replace the default 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 default stream. The functions `rand`, `randn`, and `randi` will continue to draw from the default stream, and will not affect the results of the `@RandStream` methods `rand`, `randn` and `randi` applied to `myStream`.

You can make `myStream` the default stream using the `RandStream.setDefaultStream` method.

```
RandStream.setDefaultStream(myStream)
RandStream.getDefaultStream

ans =

mlfg6331_64 random stream (current default)
             Seed: 0
         RandnAlg: Ziggurat

RandStream.getDefaultStream==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 3-12 for a list of generator algorithms and their properties.)

```
stream=RandStream('mlfg6331_64');
RandStream.setDefaultStream(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 software offers six generator algorithms. 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.

Generator algorithms

Keyword	Generator	Multiple Stream and Substream Support	Approximate Period In Full Precision
`mt19937ar`	Mersenne twister (default)	No	$2^{19937}-1$
`mcg16807`	Multiplicative congruential generator	No	$2^{31}-2$
`mlfg6331_64`	Multiplicative lagged Fibonacci generator	Yes	2^{124}
`mrg32k3a`	Combined multiple recursive generator	Yes	2^{127}
`shr3cong`	Shift-register generator summed with linear congruential generator	No	2^{64}
`swb2712`	Modified subtract with borrow generator	No	2^{1492}

Some of the generators (`mt16807`, `shr3`, `swb2712`) provide for backwards compatibility with earlier versions of MATLAB. Two generators (`mrg32k3a`, `mlfg6331_64`) provide explicit support for parallel random number generation. The remaining generator (`mt19937ar`) is designed primarily for sequential applications. Depending on the application, some generators may be faster or return values with more precision.

Another reason for the choice of generators has to do with applications. 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 [10] for an example).

Generator Algorithms

`mcg16807`

A 32-bit multiplicative congruential generator, as described in [11], 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 [8], 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^{-53} strictly within the interval (0,1). The `randn` algorithm used by default for `mlfg6331_64` streams is the ziggurat algorithm [5], but with the `mlfg6331_64` generator underneath.

`mrg32k3a`

A 32-bit combined multiple recursive generator, as described in [3]. This generator is similar to the CMRG implemented in the RngStreams

package. It has a period of 2^{127}, and supports up to 2^{63} parallel streams, via sequence splitting, and 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 [5], but with the `mrg32k3a` generator underneath.

`mt19937ar`

The Mersenne Twister, as described in [9], with Mersenne prime $2^{19937}-1$. This is the generator documented at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`. It has a period of $2^{19937}-1$. Each U(0,1) value is created using two 32-bit integers from the generator; the possible values are all multiples of 2^{-53} strictly within 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 [5], 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 using `rand('twister',s)`.

`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 $u = u(\mathbf{I}+\mathbf{L}^{17})(\mathbf{I}+\mathbf{R}^{13})(\mathbf{I}+\mathbf{L}^{5})$, 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 (described in [4]) 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 [7], but with the `shr3cong` generator underneath. Note: This generator is identical to the one used by the `randn` function beginning in MATLAB Version 5, activated using `randn('state',s)`.

`swb2712`
: A modified Subtract-with-Borrow generator, as described in [6]. 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 [5], 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 [5]. Approximately 2.02 uniform values are consumed per normal value, on average.

Compatibility Considerations

In MATLAB versions 7.6 and prior, the way to replicate results involving random numbers was to use keyword and seed value arguments with the `rand` and `randn` functions.

```
rand('twister',5489)
rand

ans =

    0.8147

rand('twister',5489)
rand

ans =

    0.8147
```

or to control the output by saving and restoring the state of the generator:

```
oldstate=rand('twister');
rand
ans =

    0.8147

rand('twister',oldstate)
rand

ans =

    0.8147
```

Using a `@RandStream` stream object simplifies this procedure. In the previous example, you need to know that you are using the generator for `'twister'` in order to restore the saved state `oldstate`. With the `@RandStream` syntax, you can check what type of generator is active using the `Type` property. You can even reproduce results without knowing what type of generator is active or what properties are set. All that is necessary is to get a handle to the default stream with `RandStream.getDefaultStream`.

```
savedStream=RandStream.getDefaultStream;
savedState=savedStream.State;
A=rand(100);
```

```
savedStream.State=savedState;
B=rand(100);
isequal(A,B)
ans =

     1
```

See the section “Legacy Mode” on page 3-22 for more information on compatibility issues.

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
        RandnAlg: Ziggurat

s2 =

mlfg6331_64 random stream
      StreamIndex: 2
       NumStreams: 3
             Seed: 0
        RandnAlg: Ziggurat

s3 =

mlfg6331_64 random stream
      StreamIndex: 3
       NumStreams: 3
             Seed: 0
        RandnAlg: 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
        RandnAlg: 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 3-12 for a summary of generator properties.

Legacy Mode

Prior to MATLAB Version 7.7, the `rand` and `randn` functions used a different syntax for controlling the random number generators. In addition, `rand` and `randn` had separate underlying random number streams. For example, in MATLAB Version 7.5, the syntax to reset the `rand` function to its default setting was:

```
rand('twister',5489)
```

To reset the `randn` function to its default setting, the syntax was:

```
randn('state',0)
```

All syntaxes from previous versions of MATLAB have been retained in the current version. Any use of the syntax `rand('keyword', seedvalue)`, where `seedvalue` is an integer or a state vector, will activate legacy mode.

```
rand('twister',5489)
RandStream.getDefaultStream

ans =

legacy random stream (current default)
   RAND algorithm: V7.4 (Mersenne Twister)
  RANDN algorithm: V5 (Ziggurat)
```

In legacy mode, `rand` and `randn` use separate generators. The generator used for each function is determined by the keyword argument of the function. For example, the `twister` keyword will activate the Mersenne Twister generator for `rand` and the ziggurat algorithm for `randn`. When in legacy mode, you can use the old control syntax to activate other legacy generators as well. The old control syntaxes for `rand` and `randn` and their equivalents with the `RandStream` constructor are given in the tables below. (The `randi` function is new in MATLAB Version 7.7 and does not accept legacy mode keyword arguments. It can be used while in legacy mode; in which case it uses the same generator as `rand`.)

When in legacy mode, you may still create random stream objects and generate values from those streams using `@RandStream` methods. However, the default stream used by the functions `rand`, `randn`, and `randi` has a

special setting, known as the legacy stream. This stream is similar to other random streams, except that it acts as an interface to the legacy generators from previous versions of MATLAB. All of the generator and transformation algorithms that were available in MATLAB Version 7.6 and earlier are available in the current version.

The legacy mode keywords syntaxes for the `rand` function and their `RandStream` equivalents are:

Legacy mode	Generator algorithm	RandStream syntax
`rand('twister',5489)`	Mersenne Twister (the default in MATLAB versions 7.4 and later).	`RandStream('mt19937ar','Seed', 5489)`
`rand('state'0)`	Modified subtract with borrow algorithm (the default in MATLAB versions 5 through 7.3).	`RandStream('swb2712','Seed',0)`
`rand('seed',0)`	Multiplicative congruential algorithm (the default in MATLAB version 4 and before).	`RandStream('mcg16807','Seed', 0)`

The legacy mode keywords syntaxes for the `randn` function and their `RandStream` equivalents are:

Legacy mode	Transformation algorithm	RandStream syntax
`randn('seed',0)`	Marsaglia's ziggurat algorithm (the default in MATLAB versions 5 and later).	`RandStream('mcg16807', 'Seed',0)`
`randn('state',0)`	Polar algorithm (the default in MATLAB version 4 and before).	`RandStream('shr3cong')`

To get out of legacy mode, create a new random stream with the constructor and make it the default stream.

```
rand('twister',5489)
% Go into legacy mode
RandStream.getDefaultStream

ans =

legacy random stream (current default)
   RAND algorithm: V7.4 (Mersenne Twister)
  RANDN algorithm: V5 (Ziggurat)


s=RandStream('mt19937ar')
RandStream.setDefaultStream(s)
s

s =

mt19937ar random stream (current default)
             Seed: 0
         RandnAlg: Ziggurat
```

Selected Bibliography

[1] Devroye, L. *Non-Uniform Random Variate Generation*, Springer-Verlag, 1986. Available online at `http://cg.scs.carleton.ca/~luc/rnbookindex.html`.

[2] 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.

[3] 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.

[4] Marsaglia, G. "Random numbers for C: The END?" Usenet posting to sci.stat.math. 1999. Available online at `http://groups.google.com/group/sci.crypt/browse_thread/thread/ca8682a4658a124d/`.

[5] Marsaglia G., and W. W. Tsang. "The ziggurat method for generating random variables." *Journal of Statistical Software*, 5:1–7. 2000. Available online at `http://www.jstatsoft.org/v05/i08`.

[6] Marsaglia, G., and A. Zaman. "A new class of random number generators." *Annals of Applied Probability* 1(3):462–480. 1991.

[7] 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.

[8] Mascagni, M., and A. Srinivasan. "Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators." *Parallel Computing*, 30: 899–916. 2004.

[9] 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.

[10] Moler, C.B. *Numerical Computing with MATLAB*. SIAM, 2004. Available online at `http://www.mathworks.com/moler`

[11] 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.

4

Sparse Matrices

- "Function Summary" on page 4-2
- "Computational Advantages" on page 4-10
- "Constructing Sparse Matrices" on page 4-13
- "Accessing Sparse Matrices" on page 4-19
- "Sparse Matrix Operations" on page 4-25
- "Selected Bibliography" on page 4-45

Function Summary

In this section...
"Functions That Support Sparse Matrices" on page 4-2
"Functions That Do Not Support Sparse Matrices" on page 4-6
"Functions with Sparse Alternatives" on page 4-10

Functions That Support Sparse Matrices

Sparse matrix functions are located in the MATLAB `sparfun` directory. For a complete list, brief descriptions, and links to reference pages, type:

```
help sparfun
```

Sparse matrix functions fall into the following categories:

- "Elementary Sparse Matrices" on page 4-2
- "Full to Sparse Conversion" on page 4-3
- "Working with Sparse Matrices" on page 4-3
- "Graph Theory" on page 4-4
- "Reordering Algorithms" on page 4-4
- "Linear Algebra" on page 4-4
- "Linear Equations (Iterative Methods)" on page 4-5
- "Other Miscellaneous Functions" on page 4-5

Elementary Sparse Matrices

Function	Description
`speye`	Sparse identity matrix
`sprand`	Sparse uniformly distributed random matrix
`sprandn`	Sparse normally distributed random matrix

Function	Description
sprandsym	Sparse random symmetric matrix
spdiags	Sparse matrix formed from diagonals

Full to Sparse Conversion

Function	Description
sparse	Create sparse matrix
full	Convert sparse matrix to full matrix
find	Find indices of nonzero elements
spconvert	Import from sparse matrix external format

Working with Sparse Matrices

Function	Description
nnz	Number of nonzero matrix elements
nonzeros	Nonzero matrix elements
nzmax	Amount of storage allocated for nonzero matrix elements
spones	Replace nonzero sparse matrix elements with ones
spalloc	Allocate space for sparse matrix
issparse	True for sparse matrix
spfun	Apply function to nonzero matrix elements
spy	Visualize sparsity pattern

Graph Theory

Function	Description
gplot	Plot graph, as in “graph theory”
etree	Elimination tree
etreeplot	Plot elimination tree
treelayout	Lay out tree or forest
treeplot	Plot picture of tree

Reordering Algorithms

Function	Description
colamd	Column approximate minimum degree permutation
symamd	Symmetric approximate minimum degree permutation
symrcm	Symmetric reverse Cuthill-McKee permutation
colperm	Column permutation
randperm	Random permutation
dmperm	Dulmage-Mendelsohn permutation

Linear Algebra

Function	Description
eigs	A few eigenvalues
svds	A few singular values
luinc	Incomplete LU factorization
ilu	Incomplete LU factorization
cholinc	Incomplete Cholesky factorization
normest	Estimate the matrix 2-norm

Function	Description
condest	1-norm condition number estimate
sprank	Structural rank

Linear Equations (Iterative Methods)

Function	Description
bicg	Biconjugate gradients method
bicgstab	Biconjugate gradients stabilized method
bicgstabl	Biconjugate gradients stabilized(l) method
cgs	Conjugate gradients squared method
gmres	Generalized minimum residual method
lsqnonneg	Solve nonnegative least-squares constraints problem
lsqr	LSQR method
minres	Minimum residual method
pcg	Preconditioned conjugate gradients method
qmr	Quasi-minimal residual method
tfqmr	Transpose-free quasi-minimal residual method
symmlq	Symmetric LQ method

Other Miscellaneous Functions

Function	Description
spaugment	Form least squares augmented system
spparms	Set parameters for sparse matrix routines
symbfact	Symbolic factorization analysis

Functions That Do Not Support Sparse Matrices

- "Elementary Matrices and Arrays" on page 4-6
- "Elementary Math Functions" on page 4-6
- "Bit-Wise Functions" on page 4-7
- "Eigenvalue and Singular Value Functions" on page 4-7
- "Matrix Analysis Functions" on page 4-8
- "Factorization Functions" on page 4-8
- "Linear Equation Functions" on page 4-8
- "Specialized Math Functions" on page 4-9
- "Filtering and Convolution Functions" on page 4-9
- "Fourier Transform Functions" on page 4-9
- "Histogram Plotting Functions" on page 4-10

These built-in functions do not accept sparse matrices as input.

Elementary Matrices and Arrays

Function	Description
`rand`	Uniformly distributed pseudorandom numbers

Elementary Math Functions

Complex Functions.

Function	Description
`complex`	Construct complex data from real and imaginary components

Real Array Exponential Functions.

Function	Description
reallog	Natural logarithm for nonnegative real arrays
realpow	Array power for real-only output
realsqrt	Square root for nonnegative real arrays

Bit-Wise Functions

Function	Description
bitand	Bitwise AND
bitcmp	Bitwise complement
bitget	Bit at specified position
bitmax	Maximum double-precision floating-point integer
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR

Eigenvalue and Singular Value Functions

Function	Description
hess	Hessenberg form of matrix
ordeig	Eigenvalues of quasitriangular matrices
ordqz	Reorder eigenvalues in QZ factorization
ordschur	Reorder eigenvalues in Schur factorization

Function	Description
schur	Schur decomposition
svd	Singular value decomposition

Matrix Analysis Functions

Function	Description
cond	Condition number with respect to inversion
null	Null space
orth	Range space of matrix
rcond	Matrix reciprocal condition number estimate

Factorization Functions

Function	Description
cholupdate	Rank 1 update to Cholesky factorization
gsvd	Generalized singular value decomposition
qz	QZ factorization for generalized eigenvalues

Linear Equation Functions

Function	Description
linsolve	Solve linear system of equations
pinv	Moore-Penrose pseudoinverse of matrix

Specialized Math Functions

Function	Description
`airy`	Airy functions
`besselh`	Bessel function of third kind (Hankel function)
`besseli`	Modified Bessel function of first kind
`besselj`	Bessel function of first kind
`besselk`	Modified Bessel function of second kind
`bessely`	Bessel function of second kind
`erfc`	Error function
`erf`	Error function
`erfcx`	Error function
`gamma`	Gamma function
`gammaln`	Gamma function
`psi`	Psi (polygamma) function

Filtering and Convolution Functions

Function	Description
`conv2`	2-D convolution
`convn`	N-D convolution
`filter`	1-D digital filter
`filter2`	2-D digital filter

Fourier Transform Functions

Function	Description
`fft`	Discrete Fourier transform
`fftn`	N-D discrete Fourier transform

Function	Description
`ifft`	Inverse discrete Fourier transform
`ifftn`	N-D inverse discrete Fourier transform

Histogram Plotting Functions

Function	Description
`histc`	Histogram count

Functions with Sparse Alternatives

These functions do not accept sparse inputs, but you can use other functions in their place.

Function	Replacement Function Supporting Sparse Inputs
`cond`	Use `condest` instead.
`eig`	Syntax `d = eig(S)` accepts a sparse symmetric matrix `S`. Otherwise, use `eigs` in place of `eig`.
`norm(S,2)`	Use `normest` for the 2-norm of a sparse matrix `S`.
`svd`	Use `svds` instead.

Computational Advantages

In this section...
"Memory Management" on page 4-10
"Computational Efficiency" on page 4-11

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              5004  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.

For more information, see “Sparse Matrix Operations” on page 4-25.

Constructing Sparse Matrices

In this section...

"Creating Sparse Matrices" on page 4-13

"Importing Sparse Matrices" on page 4-18

Creating Sparse Matrices

- "Converting Full to Sparse" on page 4-13
- "Creating Sparse Matrices Directly" on page 4-14
- "Creating Sparse Matrices from Their Diagonal Elements" on page 4-16

MATLAB software 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));
```

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 for 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.

Accessing Sparse Matrices

In this section...

"Nonzero Elements" on page 4-19

"Indices and Values" on page 4-21

"Indexing in Sparse Matrix Operations" on page 4-21

"Visualizing Sparse Matrices" on page 4-24

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

  M_full       1100x1100          9680000  double
  M_sparse     1100x1100             5004  double    sparse
  west0479      479x479             24564  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 software 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 indexing into a full matrix. For example, consider the 4-by-4 identity matrix:

```
A=eye(4);
```

To replace the 2,1 entry with the number 3, you would do this:

```
A(2,1)=3

A =

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

Now suppose you were working on a sparse matrix:

```
B=speye(4);
```

The `find` command returns the indices and values of the nonzero components of a matrix:

```
[i,j,s]=find(B);
[i,j,s]

ans =

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

If you wanted to change a value in this matrix, you might be tempted to use the same indexing:

```
B(3,1) = 42;
```

This code does work, however, it is slow. Since MATLAB stores sparse matrices in compressed sparse column format, it needs to overwrite multiple entries in `B`.

```
[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 overwrites all matrix values after 3,1 and adds an additional row to the nonzero values vector and the row indices vectors.

Instead of using subscripted assignment, try using the sparse command to construct the matrix from the triplets. For example, suppose you wanted the sparse form of the coordinate matrix `C`:

$$C = \begin{matrix} 4 & 0 & 0 & -1 \\ 0 & 4 & 0 & -1 \\ 0 & 0 & 4 & -1 \\ -1 & -1 & -1 & 4 \end{matrix}$$

You can use indexing:

```
C=4*speye(4);
C(1:3,4)=-1;
C(4,1:3)=-1;
```

but this has the same issue that many of the entries in `C` need to be overwritten. If you use a loop, you exacerbate the inefficiency:

```
C=4*speye(4);
for k=1:3
   C(k,4)=-1;
   C(4,k)=-1;
end
```

Instead of rewriting every matrix entry when you want to rewrite one, construct the three-column matrix directly with the `sparse` function:

```
i = [1 4 2 4 3 4 1 2 3 4]';
j = [1 1 2 2 3 3 4 4 4 4]';
s = [4 -1 4 -1 4 -1 -1 -1 -1 4]';
CSP = sparse(i,j,s);
```

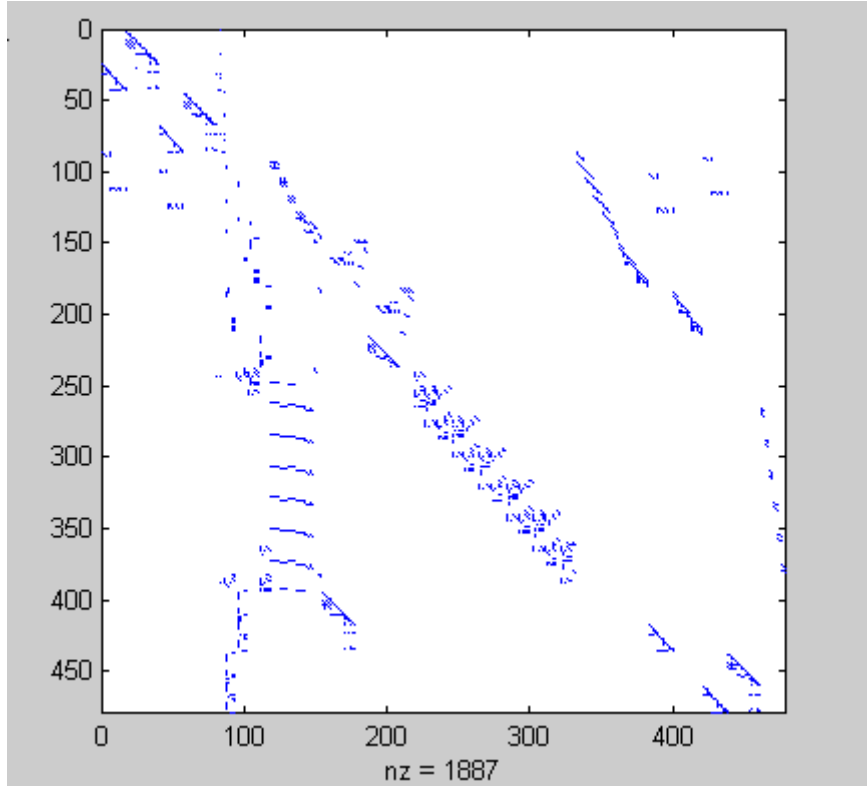
For more information on how MATLAB stores sparse matrices, see John R. Gilbert, Cleve Moler , and Robert Schreiber's Sparse Matrices In Matlab: Design and Implementation, (*SIAM Journal on Matrix Analysis and Applications* , 13:1, 333–356 (1992)).

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:

```
spy(west0479)
```



Sparse Matrix Operations

In this section...

"Efficiency of Operations" on page 4-25
"Permutations and Reordering" on page 4-26
"Factoring Sparse Matrices" on page 4-30
"Systems of Linear Equations" on page 4-37
"Eigenvalues and Singular Values" on page 4-40
"Performance Limitations" on page 4-42

Efficiency of Operations

- "Computational Complexity" on page 4-25
- "Algorithmic Details" on page 4-25

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.
- Submatrix indexing on the right side of an assignment preserves the storage format of the operand unless the result is a scalar. `T = S(i,j)` produces a sparse result if `S` is sparse and either `i` or `j` is a vector. It produces a full scalar if both `i` and `j` are scalars. Submatrix indexing on the left, as in `T(i,j) = S`, does not change the storage format of the matrix on the left.

Permutations and Reordering

- "Reordering for Sparsity" on page 4-29
- "Reordering to Reduce Bandwidth" on page 4-29
- "Approximate Minimum Degree Ordering" on page 4-29

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 function `p = colperm(S)` computes this column-count permutation. The `colperm` code has only a single line.

```
[ignore,p] = sort(sum(spones(S)));
```

This line performs these steps:

1. The inner call to `spones` creates a sparse matrix with ones at the location of every nonzero element in `S`.
2. The `sum` function sums down the columns of the matrix, producing a vector that contains the count of nonzeros in each column.
3. `full` converts this vector to full storage format.
4. `sort` sorts the values in ascending order. The second output argument from `sort` is the permutation that sorts this vector.

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 function `symrcm(A)` actually operates on the nonzero structure of the symmetric matrix `A + A'`, but the result is also useful for asymmetric 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 the 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.

The following 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 4-31 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].

Factoring Sparse Matrices

- "LU Factorization" on page 4-30
- "Cholesky Factorization" on page 4-33
- "QR Factorization" on page 4-34
- "Incomplete Factorizations" on page 4-35

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.

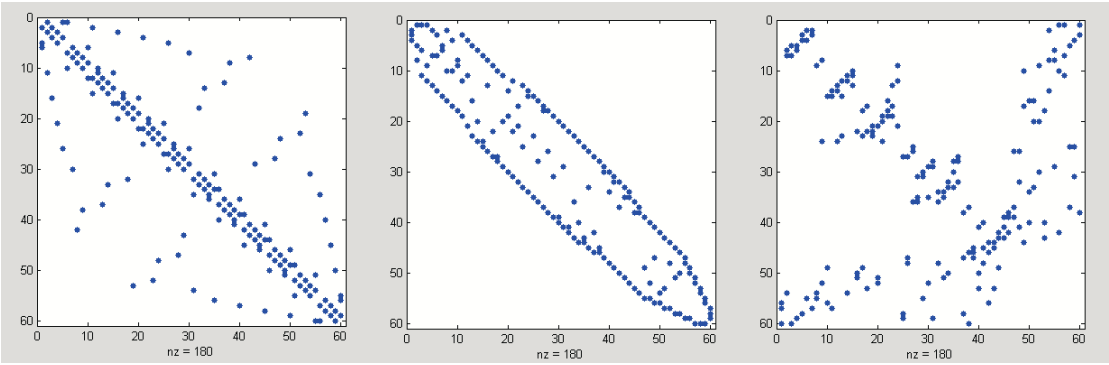
Reordering and Factorization. If you obtain a good column permutation `p` that reduces fill-in, perhaps from `symrcm` or `colamd`, then computing `lu(S(:,p))` takes less time and storage than computing `lu(S)`. Two permutations are the symmetric reverse Cuthill-McKee ordering and the symmetric approximate minimum degree ordering.

```
r = symrcm(B);
```

```
m = symamd(B);
```

The three spy plots produced by the lines below 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 other three.

```
spy(B)
spy(B(r,r))
spy(B(m,m))
```



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.

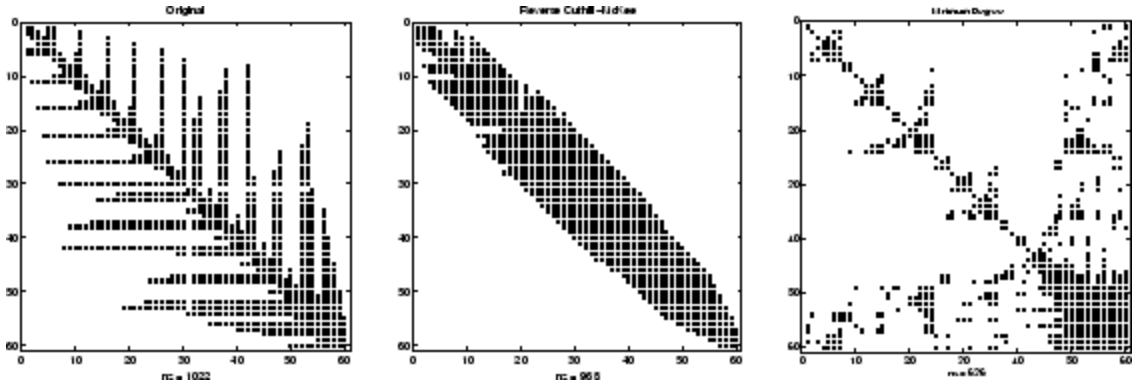
To see the fill-in generated in the LU factorization of the Bucky ball, use `speye(n,n)`, the sparse identity matrix, to insert -3s on the diagonal of `B`.

```
B = B - 3*speye(n,n);
```

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.

Original	`lu(B)`	1022
Reverse Cuthill-McKee	`lu(B(r,r))`	968
Approximate minimum degree	`lu(B(m,m))`	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.



Cholesky Factorization

If `S` is a symmetric (or Hermitian), positive definite, sparse matrix, the statement below returns a sparse, upper triangular matrix `R` so that `R'*R = S`.

```
R = chol(S)
```

`chol` does not automatically pivot for sparsity, but you can compute approximate minimum degree and profile limiting permutations for use with `chol(S(p,p))`.

Since the Cholesky algorithm does not use pivoting for sparsity and does not require pivoting for numerical stability, `chol` does a quick calculation of the amount of memory required and allocates all the memory at the start of the factorization. You can use `symbfact`, which uses the same algorithm as `chol`, to calculate how much memory is allocated.

QR Factorization

MATLAB computes the complete QR factorization of a sparse matrix `S` with

```
[Q,R] = qr(S)
```

or

```
[Q,R,E] = qr(S)
```

but this is often impractical. The unitary matrix `Q` often fails to have a high proportion of zero elements. A more practical alternative, sometimes known as "the Q-less QR factorization," is available.

With one sparse input argument and one output argument

```
R = qr(S)
```

returns just the upper triangular portion of the QR factorization. The matrix `R` provides a Cholesky factorization for the matrix associated with the normal equations:

```
R'*R = S'*S
```

However, the loss of numerical information inherent in the computation of `S'*S` is avoided.

With two input arguments having the same number of rows, and two output arguments, the statement

```
[C,R] = qr(S,B)
```

applies the orthogonal transformations to `B`, producing `C = Q'*B` without computing `Q`.

The Q-less QR factorization allows the solution of sparse least squares problems

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

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 `luinc`, `ilu`, and `cholinc` functions provide approximate, *incomplete* factorizations, which are useful as preconditioners for sparse iterative methods.

The `luinc` function produces two different kinds of incomplete LU (ILU) factorizations, one involving a drop tolerance and one involving fill-in level. The `ilu` function produces three incomplete LU factorizations, the ILU with level 0 fill-in (ILU(0)), the Crout version of ILU (ILUC), and the ILU with threshold and pivoting (ILUTP).

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:

```
nnz(luinc(A,'0'))
ans =
        7840

setup.type = 'nofill';
nnz(ilu(A,setup))
ans =
        7840

nnz(luinc(A,1e-6))
ans =
       51541

setup.type = 'ilutp';
setup.droptol = 1e-6;
nnz(luinc(A,setup))
ans =
       51541
```

These calculations, with both the `luinc` and `ilu` functions, show that with level 0 fill-in it has 7840 nonzeros, and with a drop tolerance of `1e-6` it has 51541 nonzeros. See the `luinc` and `ilu` reference pages for more options and details.

The `cholinc` function provides drop tolerance and level 0 fill-in Cholesky factorizations of symmetric, positive definite sparse matrices. See the `cholinc` 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:

```
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('autoamd',1);
spparms('autommd',1);
```

The commands spparms('autoamd',1) and spparms('autommd',1) turns the automatic preordering back on, in case you use A\b later without specifying an appropriate preordering.

Iterative Methods

Nine 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
cgs	Conjugate gradient squared

Functions for Iterative Methods for Sparse Systems (Continued)

Function	Method
`gmres`	Generalized minimum residual
`lsqr`	Least squares
`minres`	Minimum residual
`pcg`	Preconditioned conjugate gradient
`qmr`	Quasiminimal residual
`symmlq`	Symmetric LQ

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 five can handle nonsymmetric, square matrices.

All nine 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 = R'*R, where R is the incomplete Cholesky factor of A.

```
A = delsq(numgrid('S',50));
```

```
b = ones(size(A,1),1);
tol = 1.e-3;
maxit = 10;
R = cholinc(A,tol);
[x,flag,err,iter,res] = pcg(A,b,tol,maxit,R',R);
```

Only four iterations are required to achieve the prescribed accuracy.

Background information on these iterative methods and incomplete factorizations is available in [2] and [7].

Eigenvalues and Singular Values

Two functions are available that compute a few specified eigenvalues or singular values. `svds` is based on `eigs` that uses ARPACK [6].

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: $Au = \lambda Bu$.

The statement

```
[U,S,V] = svds(A,k)
```

finds the `k` largest singular values of `A` and

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

finds the `k` smallest singular values.

For example, the statements

```
L = numgrid('L',65);
A = delsq(L);
```

set up the five-point Laplacian difference operator on a 65-by-65 grid in an *L*-shaped, two-dimensional domain. The statements

```
size(A)
nnz(A)
```

show that `A` is a matrix of order 2945 with 14,473 nonzero elements.

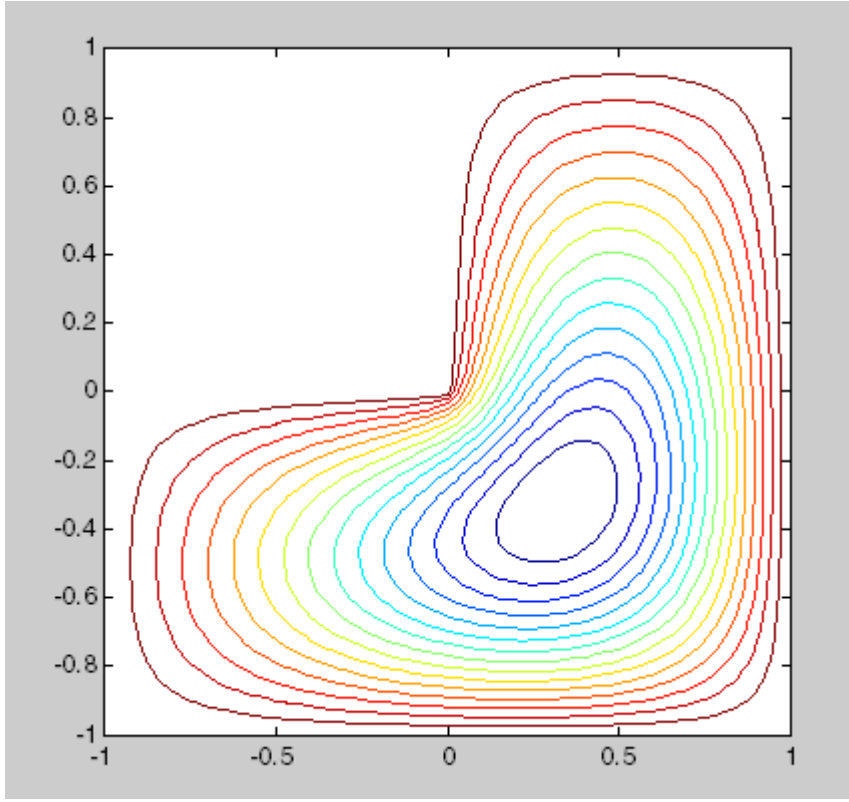
The statement

```
[v,d] = eigs(A,1,0);
```

computes the smallest eigenvalue and eigenvector. Finally,

```
L(L>0) = full(v(L(L>0)));
x = -1:1/32:1;
contour(x,x,L,15)
axis square
```

distributes the components of the eigenvector over the appropriate grid points and produces a contour plot of the result.



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

Performance Limitations

- "Creating Sparse Matrices" on page 4-43
- "Manipulating Sparse Matrices" on page 4-44

This section describes some limitations of the sparse matrix storage format and their impact on matrix creation, manipulation, and operations.

Creating Sparse Matrices

The best way to create a sparse matrix is to use the `sparse` function. If you do not have prior knowledge of the nonzero indices or their values, it is much more efficient to create the vectors containing these values, and then create the sparse matrix.

Preallocating the memory for a sparse matrix and filling it in an elementwise manner 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 26.281000 seconds.
```

Whereas 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 0.078000 seconds.
```

Manipulating Sparse Matrices

Sparse matrices are stored in a column-major format. In some cases, accessing the matrix by columns may be more efficient than accessing by rows. To do this, you can transpose the matrix, perform operations on the columns, and then retranspose the result:

```
S = sparse(10000,10000,1);
for n = 1:1000
   A = S(100,:)' + S(200,:)';
   A = A';
end;
```

The time required to transpose the matrix is negligible. Note that the sparse matrix memory requirements could prevent you from transposing a sparse matrix having a large number of rows. This might occur even when the number of nonzero values is small.

Using linear indexing to access or assign an element in a large sparse matrix will fail if the linear index exceeds `intmax`. To access an element whose linear index is greater than `intmax`, use array indexing:

```
S = spalloc(216^2, 216^2, 2)
S(1) = 1
S(end) = 1
S(216^2,216^2) = 1
```

Selected Bibliography

[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, available at `http://www.cise.ufl.edu/submit/files/file_281.ps` .

[6] Lehoucq, R. B., D. C. Sorensen, C. Yang, *ARPACK Users' Guide*, SIAM, Philadelphia, 1998.

[7] Saad, Yousef, *Iterative Methods for Sparse Linear Equations.* PWS Publishing Company, 1996.

5

Functions of One Variable

- "Function Summary" on page 5-2
- "Representing Polynomials" on page 5-3
- "Evaluating Polynomials" on page 5-4
- "Roots" on page 5-5
- "Derivatives" on page 5-6
- "Convolution" on page 5-7
- "Partial Fraction Expansions" on page 5-8
- "Polynomial Curve Fitting" on page 5-9
- "Characteristic Polynomials" on page 5-11

Function Summary

Polynomial functions are located in the MATLAB `polyfun` directory. For a complete list, brief descriptions, and links to reference pages, type:

```
help polyfun
```

The following table lists the MATLAB polynomial functions.

Function	Description
`conv`	Multiply polynomials
`deconv`	Divide polynomials
`poly`	Polynomial with specified roots
`polyder`	Polynomial derivative
`polyfit`	Polynomial curve fitting
`polyval`	Polynomial evaluation
`polyvalm`	Matrix polynomial evaluation
`residue`	Partial-fraction expansion (residues)
`roots`	Find polynomial roots

Symbolic Math Toolbox software contains additional specialized support for polynomial operations.

Representing Polynomials

MATLAB software represents polynomials as row vectors containing coefficients ordered by descending powers. For example, consider the equation

$$p(x) = x^3 - 2x - 5$$

This is the celebrated example Wallis used when he first represented Newton's method to the French Academy. To enter this polynomial into MATLAB, use

```
p = [1 0 -2 -5];
```

Evaluating Polynomials

The `polyval` function evaluates a polynomial at a specified value. To evaluate `p` at $s = 5$, use

```
polyval(p,5)

ans =
   110
```

It is also possible to evaluate a polynomial in a matrix sense. In this case $p(s) = x^3 - 2x - 5$ becomes $p(X) = X^3 - 2X - 5I$, where X is a square matrix and I is the identity matrix. For example, create a square matrix `X` and evaluate the polynomial `p` at `X`:

```
X = [2 4 5; -1 0 3; 7 1 5];
Y = polyvalm(p,X)

Y =
   377   179   439
   111    81   136
   490   253   639
```

Roots

The `roots` function calculates the roots of a polynomial:

```
r = roots(p)

r =
      2.0946
     -1.0473 +      1.1359i
     -1.0473 -      1.1359i
```

By convention, the MATLAB software stores roots in column vectors. The function `poly` returns to the polynomial coefficients:

```
p2 = poly(r)

p2 =
     1   8.8818e-16   -2   -5
```

`poly` and `roots` are inverse functions, up to ordering, scaling, and roundoff error.

Derivatives

The `polyder` function computes the derivative of any polynomial. To obtain the derivative of the polynomial `p = [1 0 -2 -5]`,

```
q = polyder(p)

q =
     3     0    -2
```

`polyder` also computes the derivative of the product or quotient of two polynomials. For example, create two polynomials `a` and `b`:

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

Calculate the derivative of the product `a*b` by calling `polyder` with a single output argument:

```
c = polyder(a,b)

c =
     8    30    56    38
```

Calculate the derivative of the quotient `a/b` by calling `polyder` with two output arguments:

```
[q,d] = polyder(a,b)

q =
    -2    -8    -2

d =
     4    16    40    48    36
```

`q/d` is the result of the operation.

Convolution

Polynomial multiplication and division correspond to the operations convolution and deconvolution. The functions `conv` and `deconv` implement these operations.

Consider the polynomials $a(s) = s^2 + 2s + 3$ and $b(s) = 4s^2 + 5s + 6$. To compute their product,

```
a = [1 2 3]; b = [4 5 6];
c = conv(a,b)

c =
     4    13    28    27    18
```

Use deconvolution to divide $a(s)$ back out of the product:

```
[q,r] = deconv(c,a)

q =
     4     5     6

r =
     0     0     0     0     0
```

Partial Fraction Expansions

`residue` finds the partial fraction expansion of the ratio of two polynomials. This is particularly useful for applications that represent systems in transfer function form. For polynomials b and a, if there are no multiple roots,

$$\frac{b(s)}{a(s)} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + \ldots + \frac{r_n}{s-p_n} + k_s$$

where r is a column vector of residues, p is a column vector of pole locations, and k is a row vector of direct terms. Consider the transfer function

$$\frac{-4s+8}{s^2+6s+8}$$

```
b = [-4 8];
a = [1 6 8];
[r,p,k] = residue(b,a)

r =
   -12
     8

p =
    -4
    -2

k =
     []
```

Given three input arguments (`r`, `p`, and `k`), `residue` converts back to polynomial form:

```
[b2,a2] = residue(r,p,k)

b2 =
    -4     8
a2 =
     1     6     8
```

Polynomial Curve Fitting

`polyfit` finds the coefficients of a polynomial that fits a set of data in a least-squares sense:

```
p = polyfit(x,y,n)
```

`x` and `y` are vectors containing the *x* and *y* data to be fitted, and `n` is the degree of the polynomial to return. For example, consider the *x*-*y* test data

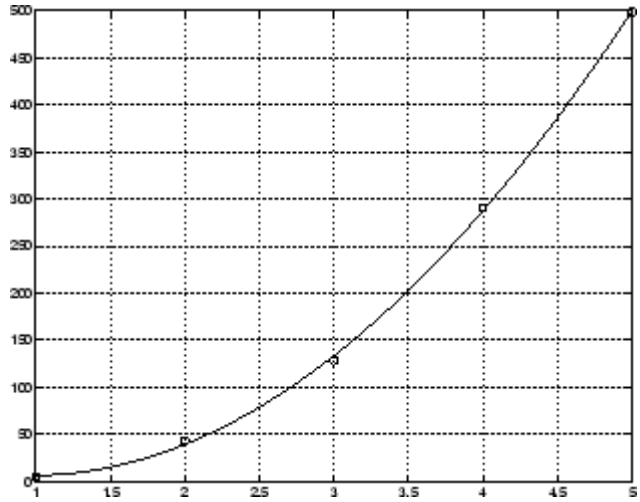
```
x = [1 2 3 4 5]; y = [5.5 43.1 128 290.7 498.4];
```

A third degree polynomial that approximately fits the data is

```
p = polyfit(x,y,3)

p =
   -0.1917   31.5821  -60.3262   35.3400
```

Compute the values of the `polyfit` estimate over a finer range, and plot the estimate over the real data values for comparison:

```
x2 = 1:.1:5;
y2 = polyval(p,x2);
plot(x,y,'o',x2,y2)
grid on
```



Note Least squares fitting of data is treated in the Data Analysis section of the MATLAB documentation.

Characteristic Polynomials

The `poly` function also computes the coefficients of the characteristic polynomial of a matrix:

```
A = [1.2 3 -0.9; 5 1.75 6; 9 0 1];
poly(A)

ans =
    1.0000   -3.9500   -1.8500  -163.2750
```

The roots of this polynomial, computed with `roots`, are the *characteristic roots*, or eigenvalues, of the matrix `A`. (Use `eig` to compute the eigenvalues of a matrix directly.)

6

Computational Geometry

- "Querying Triangulation Representations" on page 6-2
- "Delaunay Triangulation and Related Constructs" on page 6-4
- "Plotting Triangulations" on page 6-15

Querying Triangulation Representations

Triangulations 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 `TriRep` class allows you to represent triangulations that you have imported into MATLAB or created directly in MATLAB via the Delaunay triangulation tools. The `TriRep` class supports geometric queries and topological queries such as adjacency and incident information. The `TriRep` class provides a set of methods to simplify the development of triangulation-based algorithms.

The `TriRep` class provides topological and geometric queries for triangulations in 2-D and 3-D space. `DelaunayTri` is a subclass of `TriRep` and inherits methods of `TriRep` that let you represent and manipulate other types of triangulations. You also have more advanced control over the triangulation data structure. You can represent 2-D triangulations, 3-D triangulations made up of tetrahedra or 3-D surface triangulations.

You can import a triangulation into MATLAB and query it with `TriRep` methods. This example uses the sample data `tetmesh`.

```
load tetmesh
```

Use the `TriRep` constructor to create a triangulation representation of `tet` and `X`:

```
TR = TriRep(tet, X);
```

Now, you can use `TriRep` methods to query `TR`. For example, the `size` method provides size information for the triangulation matrix:

```
size(TR);
```

The `edges` method returns the edges in the triangulation in an `n`-by-2 matrix. `n` is the number of edges. The vertices of the edges index into `TR.X`, the array of points representing the vertex coordinates:

```
E = edges(TR);
```

The `edgeAttachments` method returns the simplices attached to a specified edge:

```
SI = edgeAttachments(TR,E(7,:));
```

The `incenters` and `circumcenters` methods return the incenters and circumcenters for a given simplex:

```
IC = incenters(TR,cell2mat(SI));
CC = circumcenters(TR, cell2mat(SI));
```

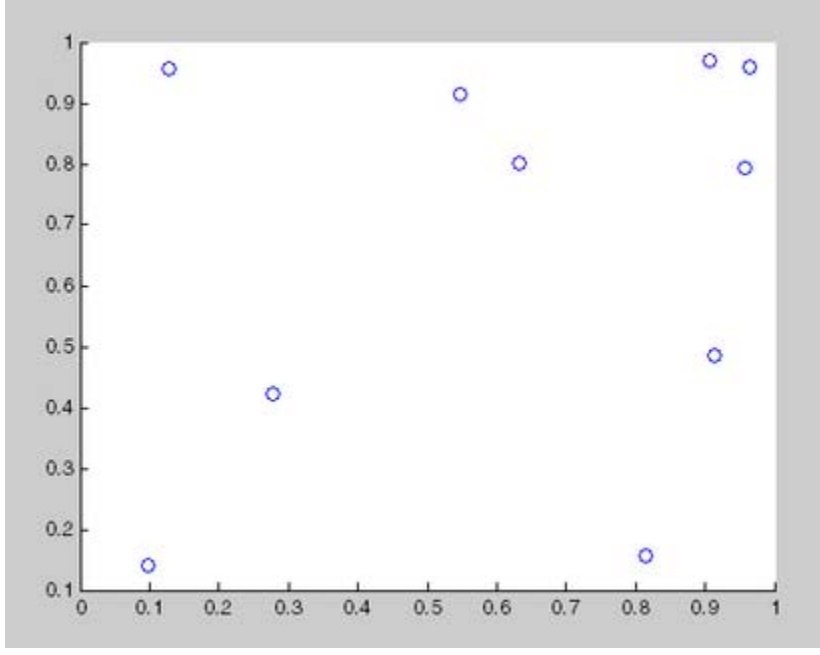
Note `SI` is a cell array because the number of simplices associated with each edge can vary.

Delaunay Triangulation and Related Constructs

The Delaunay triangulation is the most widely used triangulation in scientific computing. While there are numerous algorithms for computing Delaunay triangulations, it is the favorable properties of the triangulation that has promoted widespread adoption. The fundamental property is the Delaunay criterion, or in the case of 2-D triangulations the empty circumcircle criterion. Given a set of points in 2-D, the Delaunay criterion ensures that the circumcircle associated with each triangle contains no other point in its interior. This defines a nearest-neighbor relation and leads to the creation of "well shaped" triangles. These characteristics have important implications in practice, and motivate the use of Delaunay triangulations in scattered data interpolation.

To create a 2-D Delaunay triangulation, start with a random 2-D dataset:

```
x=rand(10,1);
y=rand(10,1);
scatter(x,y)
```



The `DelaunayTri` constructor creates the triangulation object representing the Delaunay triangulation of a set of points.

```
dt=DelaunayTri(x,y);
```

Find the properties of the Delaunay triangulation object `dt` with the `properties` function:

```
properties(dt)

Properties for class DelaunayTri:

Constraints
X
Triangulation
```

The `X` property is the original set of data points.

```
isequal(dt.X,[x,y])

ans =

 1
```

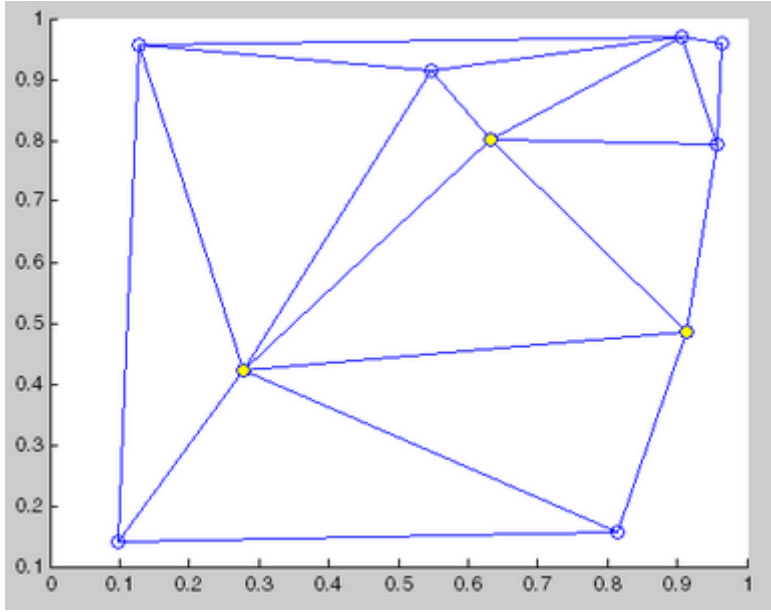
The Triangulation property is a matrix representing the set of simplices that make up the triangulation. in this case, the simplices are triangles.

```
dt.Triangulation

ans =

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

```
 2 9 10
```

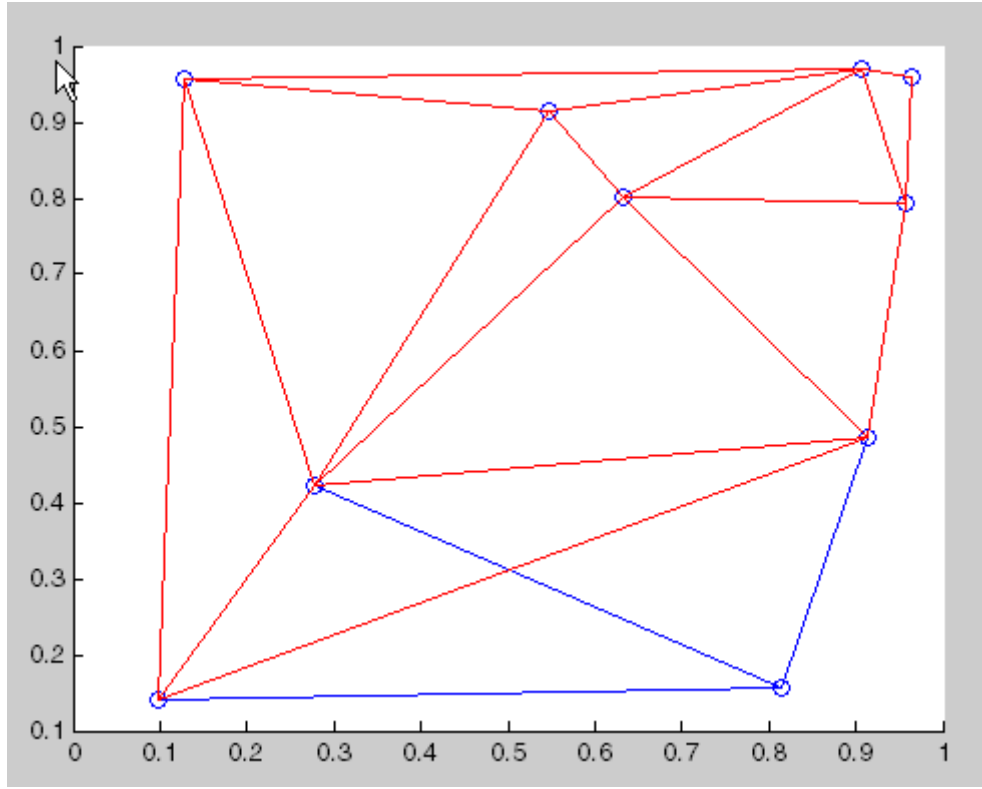
You can access the triangles by indexing directly into the `dt` object. For example, to return the vertices of the first triangle, use MATLAB's indexing capability.

```
ind = dt(1,:)

ind =

 7 4 5
tri1 = dt.X(ind,:)

tri1 =

0.2785 0.4218
0.9134 0.4854
0.6324 0.8003
scatter(tri1(:,1),tri1(:,2),'MarkerFaceColor','y')
```



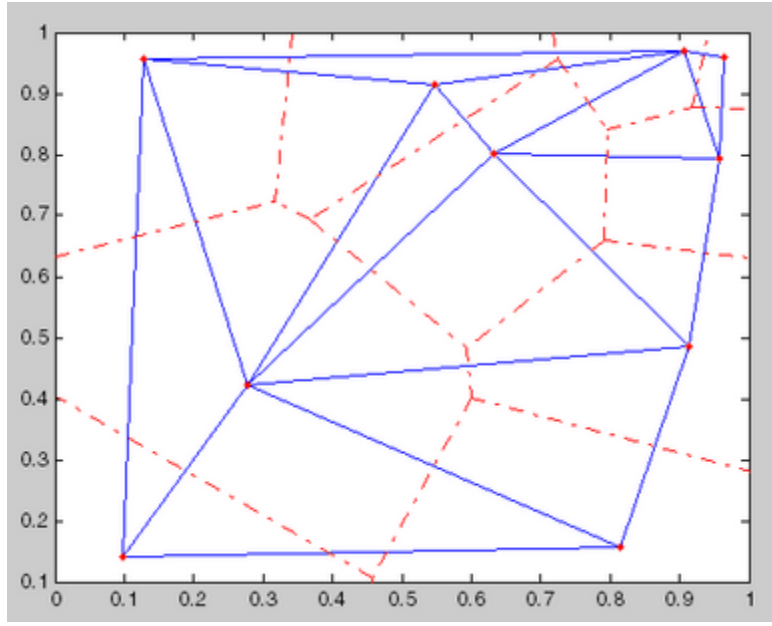
You can remove the point from the triangulation:

```
dt.X(1,:)=[];
triplot(dt,'r')
```



You can use the methods of the `DelaunayTri` class to find other geometric representations of the point set. 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}`. Locations within the region `R{i}` are closer to point i than any other point in `X`. The region of influence is the Voronoi region. The collection of all the Voronoi regions is the Voronoi diagram. `[V, R] = voronoiDiagram(dt)` returns the vertices `V` and regions `R` of the Voronoi diagram of the points `dt.X`. The region R{i} is a cell array of indices into `V` that represents the Voronoi vertices bounding the region. The Voronoi diagram of a set of points is the dual of the Delaunay triangulation;

```
hold off
dt=DelaunayTri(x,y);
voronoi(dt, '-.r')
```



Note `voronoi` is a MATLAB function and not a method of the `DelaunayTri` class.

The convex hull of a set of points is the smallest convex set that includes all the points. All triangulations of a set have the convex hull as their outer boundary. The `convexHull` method returns the indices of the vertices on the convex hull.

```
ch=convexHull(dt)

ch =

 1
 4
```

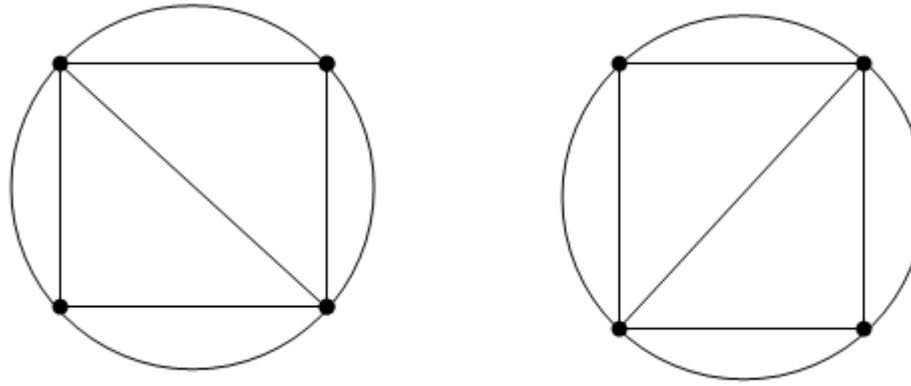
```
9
10
2
3
6
1
```

The `nearestNeighbor` method returns the index of nearest point in `dt.X` closest to a specified point. The`pointLocation` method returns the triangle that encloses a specified point.

To see all methods of the DelaunayTri class, type:

```
methods DelaunayTri -full
```

While the Delaunay property is well defined, the topology of the triangulation is not unique in the presence of degenerate pointsets. 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.



Constrained Delaunay Triangulation

Another property of the DelaunayTri object is `Constraints`. Given a set of points:

```
X = [0 0; 16 0; 16 2; 2 2; 2 3; 8 3; 8 5; 0 5];
```

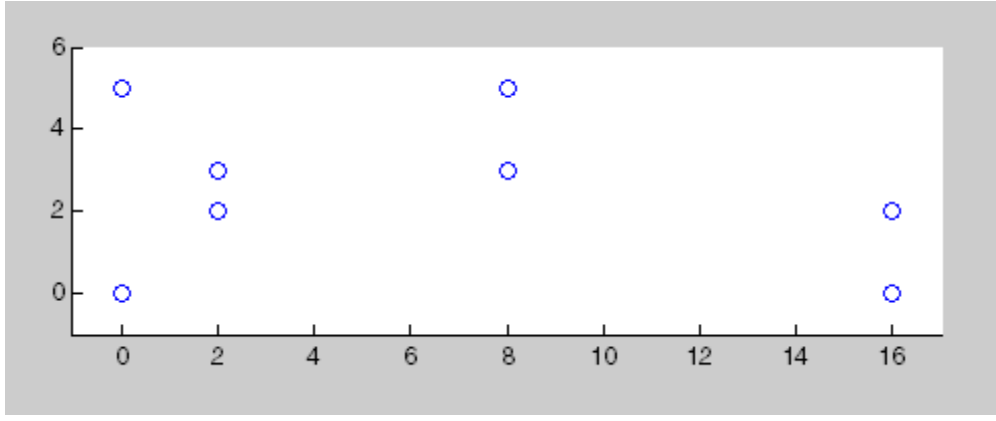
and its Delaunay triangulation:

```
DT = DelaunayTri(X);
```

You can constrain the triangulation to use a particular set of edges.

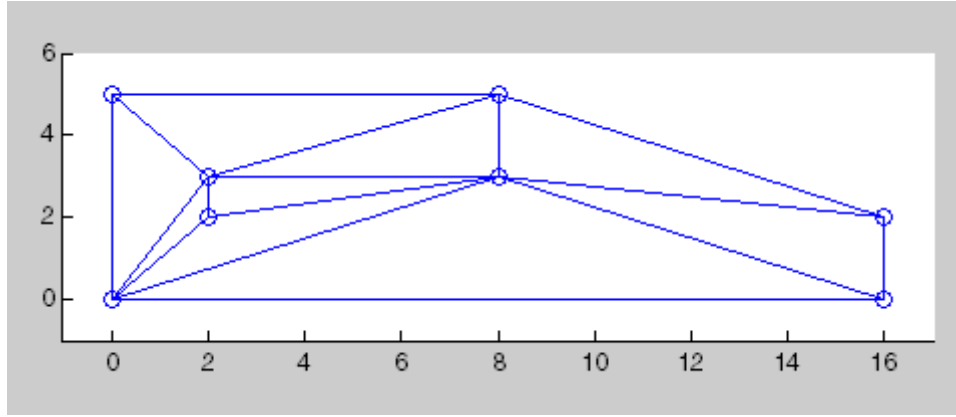
First, plot the point set `X`:

```
subplot(2,1,1);
hold on;
scatter(X(:,1),X(:,2))
axis([-1 17 -1 6]);
```



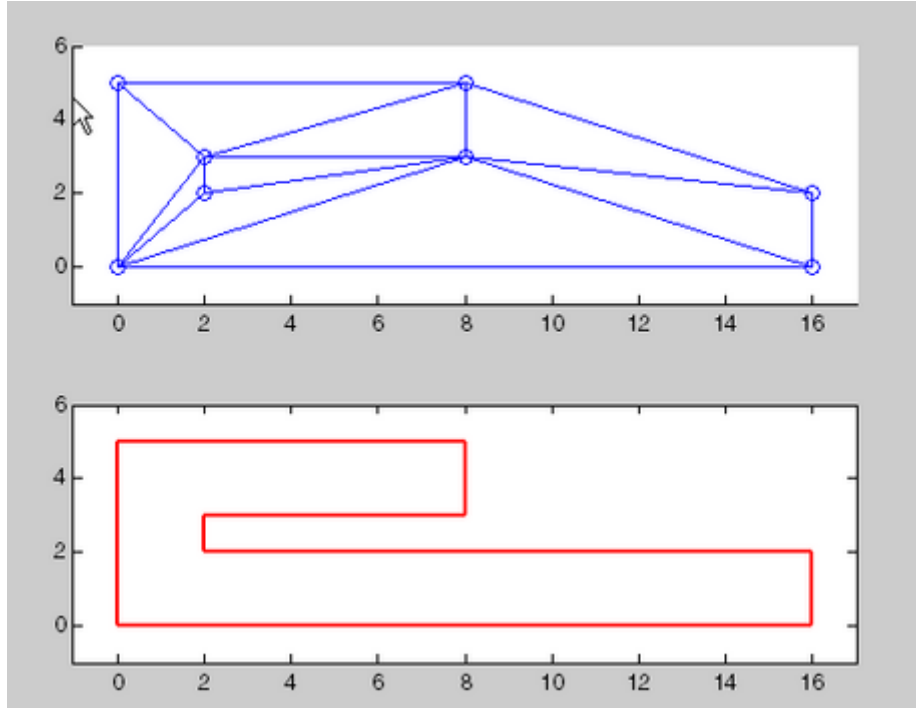
Then, plot the Delaunay triangulation of `X`:

```
triplot(DT)
```



Given a set of constraining points `C`, you can make the triangulation use the edges in `C`. For example, the edges represented by the set `C` are plotted below.

```
subplot(2,1,2);
C = [1 2; 2 3; 3 4; 4 5; 5 6; 6 7; 7 8; 8 1];
 axis([-1 17 -1 6]);
plot(X(C'),X(C'+size(X,1)),'-r', 'LineWidth', 2);
```

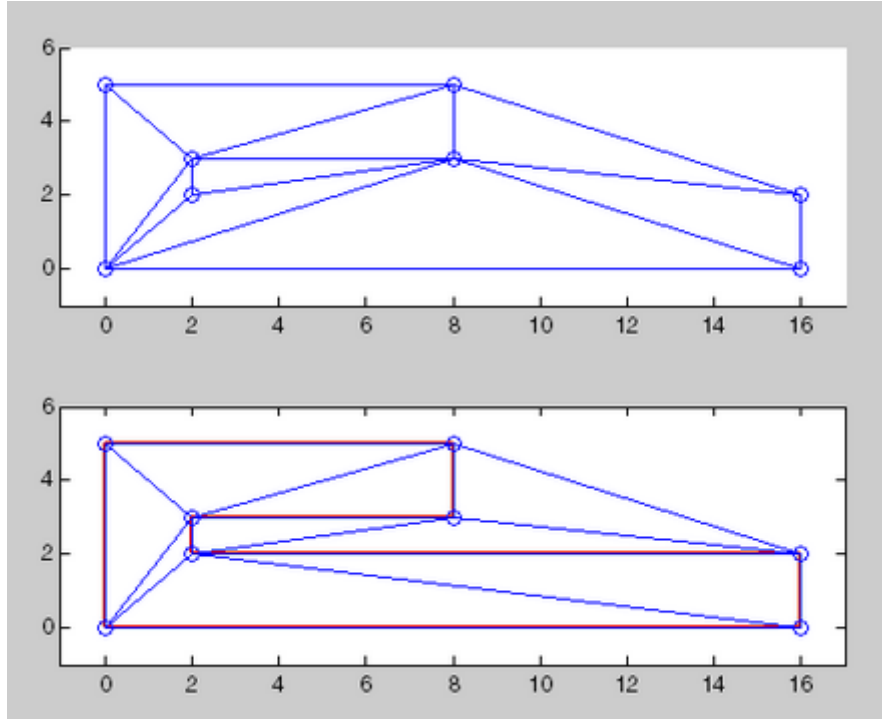


Set the `Constraints` property of the `DT` object to `C`. The constraints in `C` require that there be an edge between points 1 and 2; between points 2 and 3; between points 3 and 4; etc.

```
DT.Constraints = C;
```

A constrained triangulation is one in which specified edges are required to be part of the triangulation. Since there is no edge between points 3 and 4 in the original triangulation, the `Constraints` property creates one. Plot the triangulation which is constrained by the edges represented by `C`:

```
scatter(X(:,1),X(:,2))
triplot(dt)
```

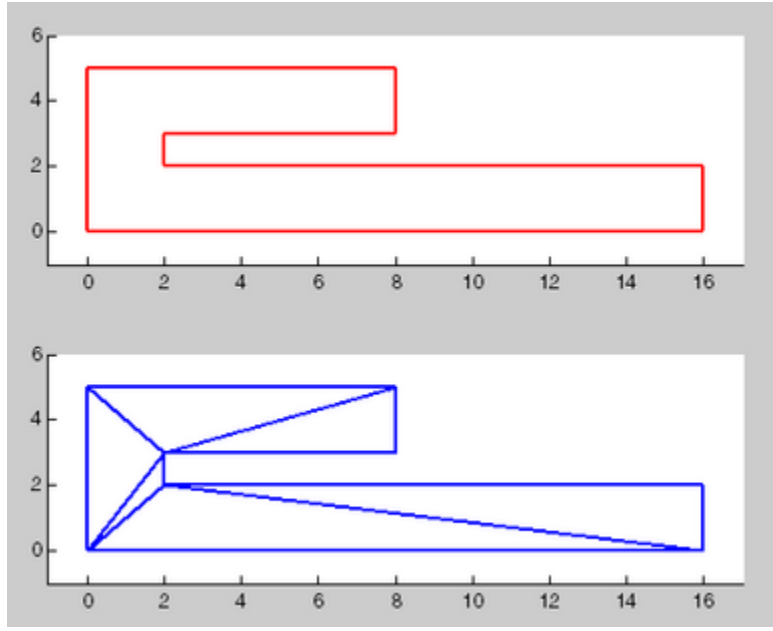


The top plot shows the original unconstrained Delaunay triangulation. The bottom plot shows the triangulation subject to the constraints `C`.

You can use the `inOutStatus` method to constrain a triangulation to the interior of a region. To return only the edges that are inside the region delineated by C:

```
hold off;
subplot(2,1,1);
hold on;
IO = inOutStatus(DT);
subplot(2,1,1);
hold on;
axis([-1 17 -1 6]);
plot(X(C'),X(C'+size(X,1)),'-r', 'LineWidth', 2);
hold off;
subplot(2,1,2);
```

```
hold on;
axis([-1 17 -1 6]);
triplot(DT(IO, :), DT.X(:,1), DT.X(:,2), 'LineWidth', 2)
hold off;
```

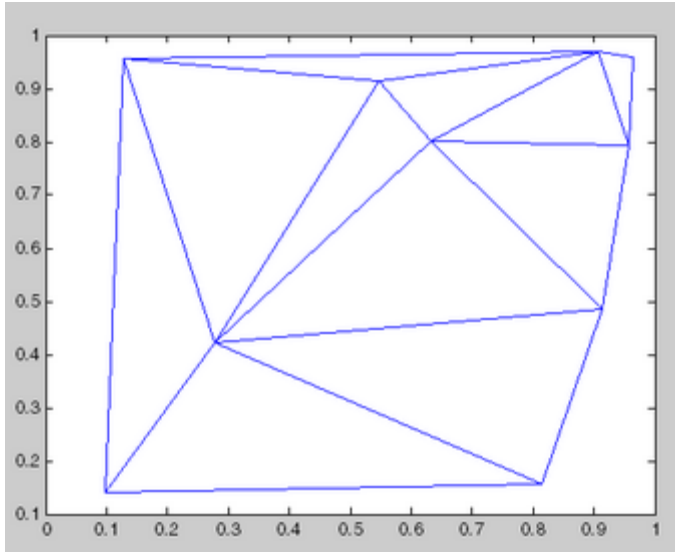


Plotting Triangulations

MATLAB provides standalone functions to plot triangulation representations. These functions can take 2- and 3-D `TriRep` and `DelaunayTri` objects as input.

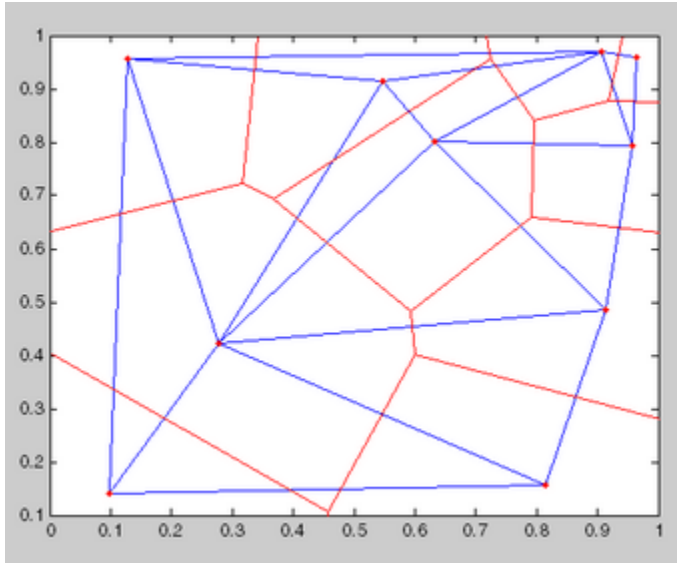
Create a 2-D Delaunay triangulation and plot it with `triplot`:

```
x=rand(10,1);
y=rand(10,1);
dt = DelaunayTri(x,y);
triplot(dt)
hold on;
```



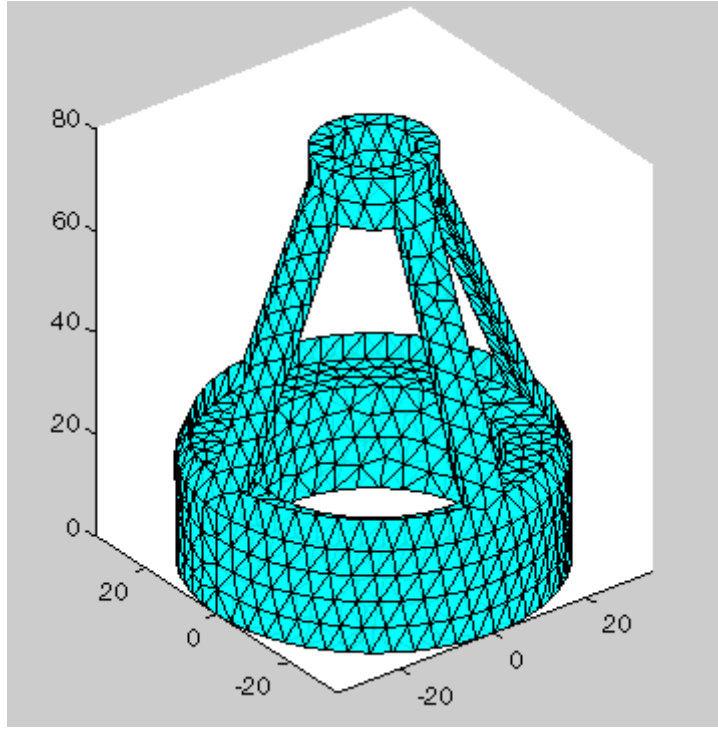
Plot the Voronoi diagram with `voronoi`:

```
voronoi(dt)
hold off;
```



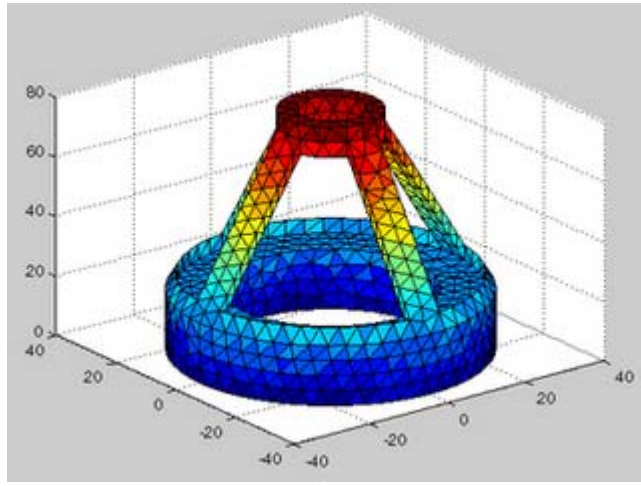
You can also plot 3-D triangulation representations.

```
load tetmesh
TR = TriRep(tet, X);
tetramesh(tet,X,'FaceColor','cyan')
```



You can plot a surface mesh with `trisurf`.

```
load trimesh3d
TR = TriRep(tri,x,y,z);
trisurf(TR)
```


80
60
40
20
0
40
20
0
-20
-40
-40
-20
0
20
40

7

Interpolation

- "Interpolating Uniform Data" on page 7-2
- "Interpolating Scattered Data" on page 7-11

Data interpolation is an application area based on underlying geometric algorithms. Data may be uniform, that is, sampling occurs over uniform intervals. Data may also be scattered, that is, sampling occurs over irregular intervals.

Interpolating Uniform Data

In this section...

"One-Dimensional Data" on page 7-2

"Two-Dimensional Data" on page 7-4

"Multidimensional Data" on page 7-7

One-Dimensional Data

- "Polynomial Interpolation" on page 7-2
- "FFT-Based Interpolation" on page 7-3

Polynomial Interpolation

The function `interp1` performs one-dimensional interpolation. This function uses polynomial techniques, fitting the supplied data with polynomial functions between data points and evaluating the appropriate function at the desired interpolation points. Its most general form is

```
yi = interp1(x,y,xi,method)
```

`y` is a vector containing the values of a function, and `x` is a vector of the same length containing the points for which the values in `y` are given. `xi` is a vector containing the points at which to interpolate. *`method`* is an optional string specifying an interpolation method:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method sets the value of an interpolated point to the value of the nearest existing data point.
- *Linear interpolation* (`method = 'linear'`). This method fits a different linear function between each pair of existing data points, and returns the value of the relevant function at the points specified by `xi`. This is the default method for the `interp1` function.
- *Cubic spline interpolation* (`method = 'spline'`). This method fits a different cubic function between each pair of existing data points, and uses the `spline` function to perform cubic spline interpolation at the data points.

- *Cubic interpolation* (`method = 'pchip'` or `'cubic'`). These methods are identical. They use the `pchip` function to perform piecewise cubic Hermite interpolation within the vectors `x` and `y`. These methods preserve monotonicity and the shape of the data.

If any element of `xi` is outside the interval spanned by `x`, the specified interpolation method is used for extrapolation. Alternatively, `yi = interp1(x,Y,xi,method,extrapval)` replaces extrapolated values with `extrapval`. `NaN` is often used for `extrapval`.

All methods work with nonuniformly spaced data.

Speed, Memory, and Smoothness Considerations. 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:

- Nearest neighbor interpolation is the fastest method. However, it provides the worst results in terms of smoothness.
- Linear interpolation uses more memory than the nearest neighbor method, and requires slightly more execution time. Unlike nearest neighbor interpolation its results are continuous, but the slope changes at the vertex points.
- Cubic spline interpolation has the longest relative execution time, although it requires less memory than cubic interpolation. It produces the smoothest results of all the interpolation methods. You may obtain unexpected results, however, if your input data is nonuniform and some points are much closer together than others.
- Cubic interpolation requires more memory and execution time than either the nearest neighbor or linear methods. However, both the interpolated data and its derivative are continuous.

FFT-Based Interpolation

The function `interpft` performs one-dimensional interpolation using an FFT-based method. This method calculates the Fourier transform of a vector that contains the values of a periodic function. It then calculates the inverse Fourier transform using more points. Its form is

```
y = interpft(x,n)
```

`x` is a vector containing the values of a periodic function, sampled at equally spaced points. `n` is the number of equally spaced points to return.

Two-Dimensional Data

- "Two-Dimensional Methods" on page 7-4
- "A Comparison of Methods" on page 7-5

Two-Dimensional Methods

The function `interp2` performs two-dimensional interpolation, an important operation for image processing and data visualization. Its most general form is

```
ZI = interp2(X,Y,Z,XI,YI,method)
```

`Z` is a rectangular array containing the values of a two-dimensional function, and `X` and `Y` are arrays of the same size containing the points for which the values in `Z` are given. `XI` and `YI` are matrices containing the points at which to interpolate the data. `method` is an optional string specifying an interpolation method.

There are three different interpolation methods for two-dimensional data:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method fits a piecewise constant surface through the data values. The value of an interpolated point is the value of the nearest point.
- *Bilinear interpolation* (`method = 'linear'`). This method fits a bilinear surface through existing data points. The value of an interpolated point is a combination of the values of the four closest points. This method is piecewise bilinear, and is faster and less memory-intensive than bicubic interpolation.
- *Bicubic interpolation* (`method = 'cubic'`). This method fits a bicubic surface through existing data points. The value of an interpolated point is a combination of the values of the sixteen closest points. This method is piecewise bicubic, and produces a much smoother surface than bilinear interpolation. This can be a key advantage for applications like image

processing. Use bicubic interpolation when the interpolated data and its derivative must be continuous.

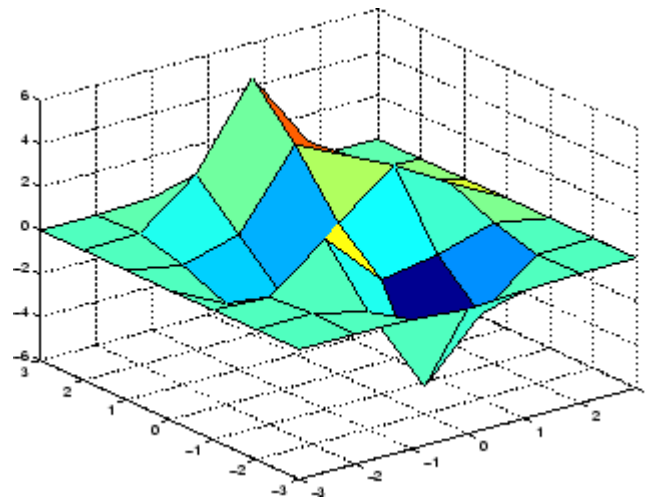
All of these methods require that `X` and `Y` be monotonic, that is, either always increasing or always decreasing from point to point. You should prepare these matrices using the `meshgrid` function, or else be sure that the "pattern" of the points emulates the output of `meshgrid`. In addition, each method automatically maps the input to an equally spaced domain before interpolating. If `X` and `Y` are already equally spaced, you can speed execution time by prepending an asterisk to the `method` string, for example, `'*cubic'`.

A Comparison of Methods

This example compares two-dimensional interpolation methods on a 7-by-7 matrix of data:

1 Generate the peaks function at low resolution:

```
[x,y] = meshgrid(-3:1:3);
z = peaks(x,y);
surf(x,y,z)
```



2 Generate a finer mesh for interpolation:

```
[xi,yi] = meshgrid(-3:0.25:3);
```

3 Interpolate using nearest neighbor interpolation:

```
zi1 = interp2(x,y,z,xi,yi,'nearest');
```

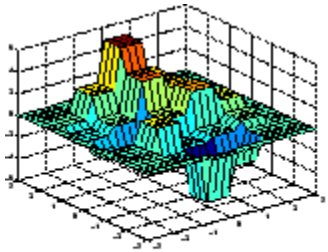
4 Interpolate using bilinear interpolation:

```
zi2 = interp2(x,y,z,xi,yi,'bilinear');
```

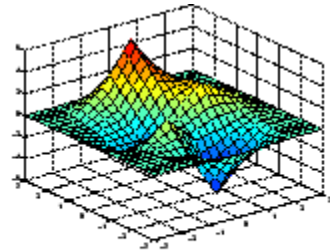
5 Interpolate using bicubic interpolation:

```
zi3 = interp2(x,y,z,xi,yi,'bicubic');
```

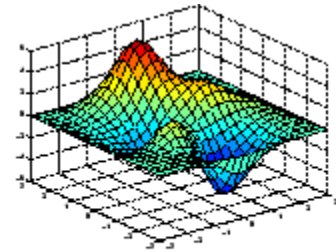
6 Compare the surface plots for the different interpolation methods.



```
surf(xi,yi,zi1)
% nearest
```

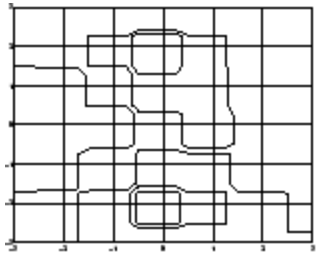


```
surf(xi,yi,zi2)
% bilinear
```

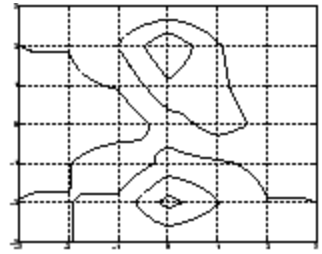


```
surf(xi,yi,zi3)
% bicubic
```

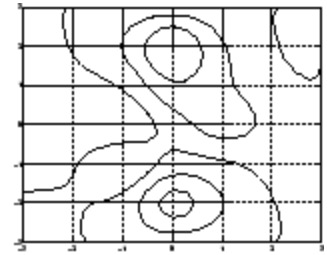
7 Compare the contour plots for the different interpolation methods.



```
contour(xi,yi,zi1)
% nearest
```



```
contour(xi,yi,zi2)
% bilinear
```



```
contour(xi,yi,zi3)
% bicubic
```

Notice that the bicubic method, in particular, produces smoother contours. This is not always the primary concern, however. For some applications, such as medical image processing, a method like nearest neighbor may be preferred because it doesn't generate any "new" data values.

Multidimensional Data

- "Function Summary" on page 7-7
- "Interpolation of Three-Dimensional Data" on page 7-7
- "Interpolation of Higher Dimensional Data" on page 7-8
- "Multidimensional Data Gridding" on page 7-9

Function Summary

The following table lists functions that operate specifically on multidimensional data.

Interpolation Functions for Multidimensional Data

Function	Description
`interp3`	Three-dimensional data interpolation
`interpn`	Multidimensional data interpolation
`ndgrid`	Multidimensional data gridding (`elmat` directory)

Interpolation of Three-Dimensional Data

The function `interp3` performs three-dimensional interpolation, finding interpolated values between points of a three-dimensional set of samples `V`. You must specify a set of known data points:

- `X`, `Y`, and `Z` matrices specify the points for which values of `V` are given.
- A matrix `V` contains values corresponding to the points in `X`, `Y`, and `Z`.

The most general form for `interp3` is

```
VI = interp3(X,Y,Z,V,XI,YI,ZI,method)
```

XI, YI, and ZI are the points at which interp3 interpolates values of V. For out-of-range values, interp3 returns NaN.

There are three different interpolation methods for three-dimensional data:

- *Nearest neighbor interpolation* (method = 'nearest'). This method chooses the value of the nearest point.
- *Trilinear interpolation* (method = 'linear'). This method uses piecewise linear interpolation based on the values of the nearest eight points.
- *Tricubic interpolation* (method = 'cubic'). This method uses piecewise cubic interpolation based on the values of the nearest sixty-four points.

All of these methods require that X, Y, and Z be *monotonic*, that is, either always increasing or always decreasing in a particular direction. In addition, you should prepare these matrices using the meshgrid function, or else be sure that the "pattern" of the points emulates the output of meshgrid.

Each method automatically maps the input to an equally spaced domain before interpolating. If x is already equally spaced, you can speed execution time by prepending an asterisk to the method string, for example, '*cubic'.

Interpolation of Higher Dimensional Data

The function interpn performs multidimensional interpolation, finding interpolated values between points of a multidimensional set of samples V. The most general form for interpn is

```
VI = interpn(X1,X2,X3...,V,Y1,Y2,Y3,...,method)
```

1, 2, 3, ... are matrices that specify the points for which values of V are given. V is a matrix that contains the values corresponding to these points. 1, 2, 3, ... are the points for which interpn returns interpolated values of V. For out-of-range values, interpn returns NaN.

Y1, Y2, Y3, ... must be either arrays of the same size, or vectors. If they are vectors of different sizes, interpn passes them to ndgrid and then uses the resulting arrays.

There are three different interpolation methods for multidimensional data:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method chooses the value of the nearest point.
- *Linear interpolation* (`method = 'linear'`). This method uses piecewise linear interpolation based on the values of the nearest two points in each dimension.
- *Cubic interpolation* (`method = 'cubic'`). This method uses piecewise cubic interpolation based on the values of the nearest four points in each dimension.

All of these methods require that `X1`, `X2,X3` be monotonic. In addition, you should prepare these matrices using the `ndgrid` function, or else be sure that the "pattern" of the points emulates the output of `ndgrid`.

Each method automatically maps the input to an equally spaced domain before interpolating. If `X` is already equally spaced, you can speed execution time by prepending an asterisk to the `method` string; for example, `'*cubic'`.

Multidimensional Data Gridding

The `ndgrid` function generates arrays of data for multidimensional function evaluation and interpolation. `ndgrid` transforms the domain specified by a series of input vectors into a series of output arrays. The `i`th dimension of these output arrays are copies of the elements of input vector x_i.

The syntax for `ndgrid` is

```
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)
```

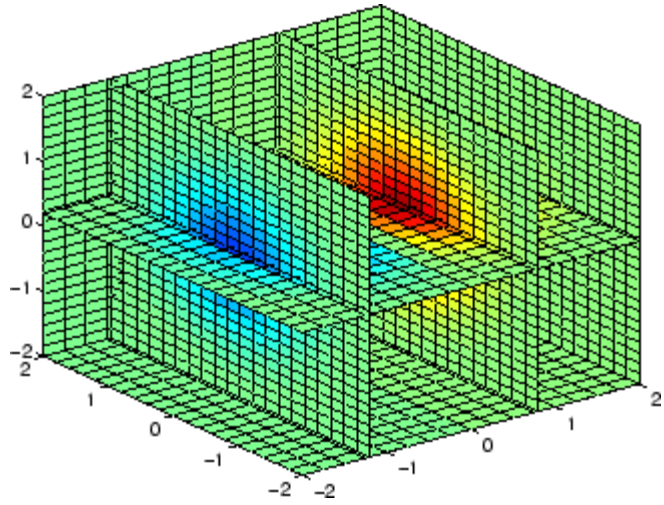
For example, assume that you want to evaluate a function of three variables over a given range. Consider the function

$$z = x_2 e^{(-x_1^2 - x_2^2 - x_3^2)}$$

for $-2\pi \le x_1 \le 0$, $2\pi \le x_2 \le 4\pi$, and $0 \le x_3 \le 2\pi$. To evaluate and plot this function,

```
x1 = -2:0.2:2;
```

```
x2 = -2:0.25:2;
x3 = -2:0.16:2;
[X1,X2,X3] = ndgrid(x1,x2,x3);
z = X2.*exp(-X1.^2 -X2.^2 -X3.^2);
slice(X2,X1,X3,z,[-1.2 0.8 2],2,[-2 0.2])
```



Interpolating Scattered Data

In the section "Interpolating Uniform Data" on page 7-2, the interpolation technique leveraged the uniformity or gridlike nature of the sample data to compute interpolated values.

When the sample data is scattered, the interpolation techniques use a triangulation-based approach as a basis for computing interpolated values. There are many ways to create a triangulation, but the triangulation that is best suited to interpolation is the Delaunay triangulation. `TriScatteredInterp` is a class for performing scattered data interpolation based on an underlying Delaunay triangulation. A scattered data set defined by locations `X` and corresponding values `V` can be interpolated using a Delaunay triangulation of `X`. This produces a surface of the form `V = F(X)`. The surface can be evaluated at any query location `QX`, using `QV = F(QX)`, where `QX` lies within the convex hull of `X`. The interpolant `F` always goes through the data points specified by the sample.

Construct the interpolant with `TriScatteredInterp`:

```
F = TriScatteredInterp(x,y,z)
TriScatteredInterp

  Properties:
         X: [100x2 double]
         V: [100x1 double]
    Method: 'linear'

  Methods
```

Evaluate the interpolant at the locations `(qx, qy)`:

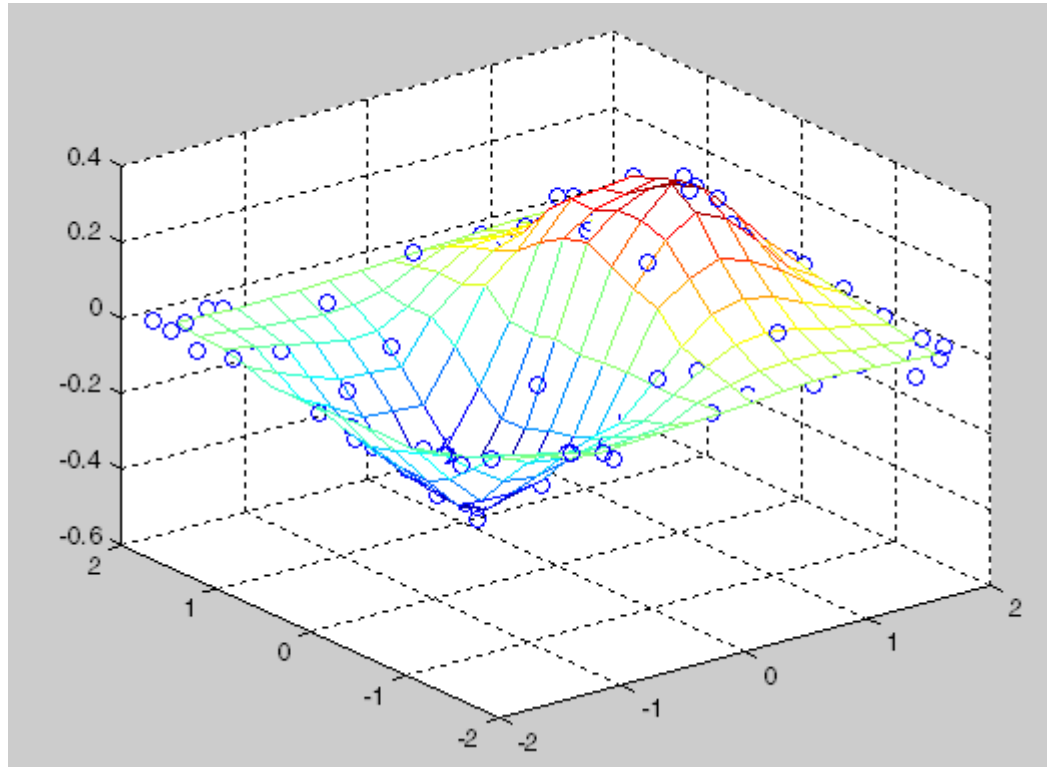
```
ti = -2:.25:2;
[qx,qy] = meshgrid(ti,ti);
```

Find the corresponding value of the interpolant at `[qx,qy]`:

```
qz = F(qx,qy);
```

Plot the result:

```
mesh(qx,qy,qz);
hold on;
plot3(x,y,z,'o');
hold off
```



You can insert additional points into the interpolation by editing the `X` and `V` properties of F:

```
x1 = rand(5,1)*4-2;
y1 = rand(5,1)*4-2;
v1 = x1.*exp(-x1.^2-y1.^2);
F.V(end+(1:5)) = v1;
F.X(end+(1:5), :) = [x1, y1];
F
```

```
F =

  TriScatteredInterp

  Properties:
         X: [105x2 double]
         V: [105x1 double]
    Method: 'linear'
```

You can also use the Delaunay triangulation as the basis for computing the interpolant:

```
x = rand(100,1)*4-2;
y = rand(100,1)*4-2;
v = x.*exp(-x.^2-y.^2);
dt=DelaunayTri(x,y);
F=TriScatteredInterp(dt,v);
```

8

Optimization

- "Function Summary" on page 8-2
- "Minimizing Functions of One Variable" on page 8-3
- "Minimizing Functions of Several Variables" on page 8-5
- "fminsearch Algorithm" on page 8-6
- "Example: Curve Fitting" on page 8-9
- "Setting Options" on page 8-12
- "Output Functions" on page 8-14
- "Plot Functions" on page 8-22
- "Finding Roots" on page 8-25
- "Troubleshooting" on page 8-30
- "Tips" on page 8-31
- "Reference" on page 8-32

Function Summary

The following table lists the MATLAB optimization functions.

Function	Description
`fminbnd`	Minimize a function of one variable on a fixed interval
`fminsearch`	Minimize a function of several variables
`fzero`	Find zero of a function of one variable
`lsqnonneg`	Linear least squares with nonnegativity constraints
`optimget`	Get optimization options structure parameter values
`optimset`	Create or edit optimization options parameter structure

For more optimization capabilities, see the *Optimization Toolbox™ User's Guide*.

Minimizing Functions of One Variable

Given a mathematical function of a single variable saved to a file, you can use the `fminbnd` function to find a local minimizer of the function in a given interval. For example, to find a minimum of the `humps.m` function in the range `(0.3,1)`, use

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

which returns

```
x =
    0.6370
```

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

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

which gives the output

```
Func-count     x          f(x)         Procedure
    3       0.567376      12.9098        initial
    4       0.732624      13.7746        golden
    5       0.465248      25.1714        golden
    6       0.644416      11.2693        parabolic
    7         0.6413      11.2583        parabolic
    8       0.637618      11.2529        parabolic
    9       0.636985      11.2528        parabolic
   10       0.637019      11.2528        parabolic
   11       0.637052      11.2528        parabolic

Optimization terminated:
 the current x satisfies the termination criteria using
OPTIONS.TolX of 1.000000e-004


x =

    0.6370
```

This 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.

Minimizing Functions of Several Variables

The `fminsearch` function is similar to `fminbnd` except that it handles functions of many variables, and you 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
```

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, and using these n vectors as elements of the simplex in addition to x_0. (It uses 0.00025 as component i if $x_0(i) = 0$.) Then, the algorithm modifies the simplex repeatedly according to the following procedure.

Note The keywords for the `fminsearch` iterative display appear in bold after the description of the step.

---

1 Let $x(i)$ denote the list of points in the current simplex, $i = 1,...,n+1$.

2 Order the points in the simplex from lowest function value $f(x(1))$ to highest $f(x(n+1))$. At each step in the iteration, the algorithm discards the current worst point $x(n+1)$, and accepts another point into the simplex. [Or, in the case of step 7 below, it changes all n points with values above $f(x(1))$].

3 Generate the *reflected* point

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

where

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

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

a If $f(r) < f(x(n+1))$ (i.e., 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 $f(r) \geq f(x(n+1))$, calculate

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

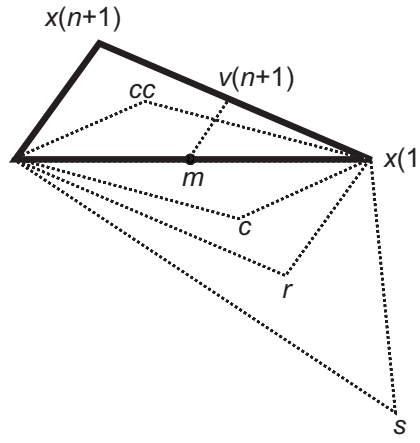
and calculate $f(cc)$. If $f(cc) < f(x(n+1))$, accept cc and terminate the iteration. **Contract inside** Otherwise, continue with Step 7 (Shrink).

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` might 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.


x(n+1)
v(n+1)
cc
x(1)
m
c
r
s

Example: Curve Fitting

This section gives an example that shows how to fit an exponential function of the form $Ae^{-\lambda t}$ to some data. The example uses the function `fminsearch` to minimize the sum of squares of errors between the data and an exponential function $Ae^{-\lambda t}$ for varying parameters A and λ. This section covers the following topics.

In this section...
"Creating an Example File" on page 8-9
"Running the Example" on page 8-10
"Plotting the Results" on page 8-10

Creating an Example File

To run the example, first create a file that:

- Accepts vectors corresponding to the x- and y-coordinates of the data
- Returns the parameters of the exponential function that best fits the data

To do so, copy and paste the following code into a file and save it as `fitcurvedemo` in a directory on the MATLAB path.

```
function [estimates, model] = fitcurvedemo(xdata, ydata)
% Call fminsearch with a random starting point.
start_point = rand(1, 2);
model = @expfun;
estimates = fminsearch(model, start_point);
% expfun accepts curve parameters as inputs, and outputs sse,
% the sum of squares error for A*exp(-lambda*xdata)-ydata,
% and the FittedCurve. FMINSEARCH only needs sse, but we want
% to plot the FittedCurve at the end.
    function [sse, FittedCurve] = expfun(params)
        A = params(1);
        lambda = params(2);
        FittedCurve = A .* exp(-lambda * xdata);
        ErrorVector = FittedCurve - ydata;
        sse = sum(ErrorVector .^ 2);
```

```
    end
end
```

The file calls the function `fminsearch`, which finds parameters `A` and `lambda` that minimize the sum of squares of the differences between the data and the exponential function `A*exp(-lambda*t)`. The nested function `expfun` computes the sum of squares.

Running the Example

To run the example, first create some random data to fit. The following commands create random data that is approximately exponential with parameters `A = 40` and `lambda = .5`.

```
xdata = (0:.1:10)';
ydata = 40 * exp(-.5 * xdata) + randn(size(xdata));
```

To fit an exponential function to the data, enter

```
[estimates, model] = fitcurvedemo(xdata,ydata)
```

This returns estimates for the parameters `A` and `lambda`,

```
estimates =

   40.1334    0.5025
```

and a function handle, `model`, to the function that computes the exponential function `A*exp(-lambda*t)`.

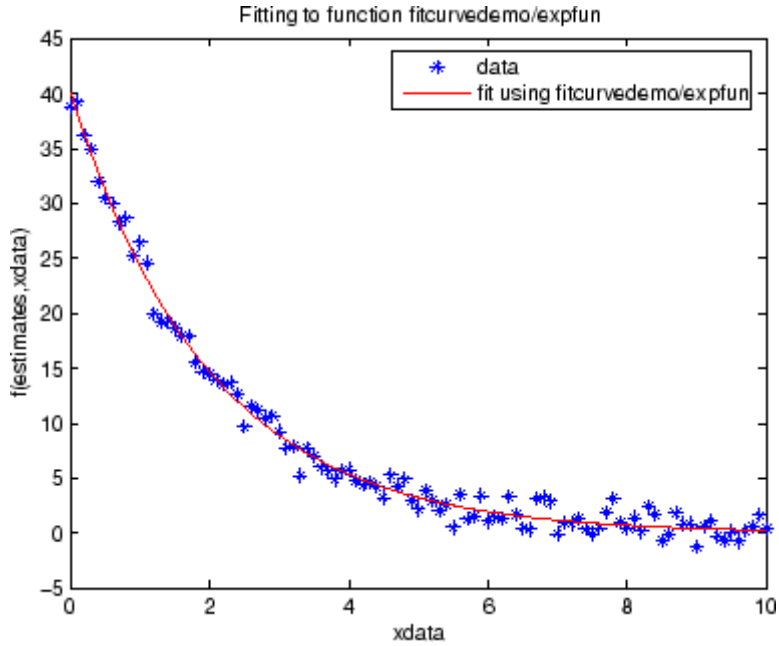
Plotting the Results

To plot the fit and the data, enter the following commands.

```
plot(xdata, ydata, '*')
hold on
[sse, FittedCurve] = model(estimates);
plot(xdata, FittedCurve, 'r')

xlabel('xdata')
ylabel('f(estimates,xdata)')
```

```
title(['Fitting to function ', func2str(model)]);
legend('data', ['fit using ', func2str(model)])
hold off
```

The resulting plot displays the data points and the exponential fit.



Setting Options

You can specify control options that set some minimization 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');
```

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`). • `'off'` displays no intermediate steps. • `'final'` displays just the final output.	`fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`
`FunValCheck`	Check whether objective function and constraints values are valid. `'on'` displays an error when the objective function or constraints return a value that is complex or NaN. The default `'off'` displays no error.	`fminbnd`, `fminsearch`, `fzero`
`MaxFunEvals`	The maximum number of function evaluations allowed. The default value is `500` for `fminbnd` and `200*length(x0)` for `fminsearch`.	`fminbnd`, `fminsearch`

Option	Description	Solvers
`MaxIter`	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). For details, see "Output Functions" on page 8-14.	`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 8-22.	`fminbnd`, `fminsearch`, `fzero`
`TolFun`	The termination tolerance for the function value. The default value is `1.e-4`.	`fminsearch`
`TolX`	The termination tolerance for *x*. The default value is `1.e-4`.	`fminbnd`, `fminsearch`, `fzero`, `lsqnonneg`

The `output` structure includes the number of function evaluations, the number of iterations, and the algorithm. The structure appears when you provide `fminbnd` or `fminsearch` with a fourth output argument, as in

```
[x,fval,exitflag,output] = fminbnd(@humps,0.3,1);
```

or

```
[x,fval,exitflag,output] = fminsearch(@three_var,v);
```

Output Functions

In this section...

"What Is an Output Function?" on page 8-14
"Creating and Using an Output Function" on page 8-14
"Structure of the Output Function" on page 8-16
"Example of a Nested Output Function" on page 8-17
"Fields in optimValues" on page 8-19
"States of the Algorithm" on page 8-19
"Stop Flag" on page 8-20

What Is an Output Function?

An *output function* is a function that an optimization function calls at each iteration of its algorithm. Typically, you might 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 subfunction, 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

$$\min_x f(x) = \min_x e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right).$$

To do so,

1. Create a file containing the preceding code and save it as `outfun.m` in a directory on the MATLAB path.
2. Enter the command

   ```
   options = optimset('OutputFcn', @outfun);
   ```

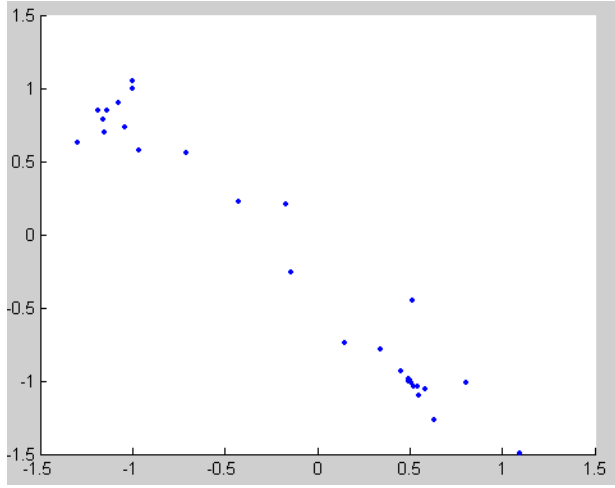
   to set the value of the `Outputfcn` field of the `options` structure to a function handle to `outfun`.
3. Enter the following commands:

   ```
   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
   ```

   This returns the solution

   ```
   x =
       0.1290   -0.5323

   fval =
      -0.5689
   ```

   and displays 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 should quit or continue. See "Stop Flag" on page 8-20.
- `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 8-19 describes the structure in detail.
- `state` is the current state of the algorithm. "States of the Algorithm" on page 8-19 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 8-14 does not require the output function to preserve data from one iteration to the next. When this is the case, you can write the output function as a function file and call the optimization function directly from the command line. However, if you want your output function to record data from one iteration to the next, you should 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 subfunction, although you could also write the objective function as a separate file or as an anonymous function.

Since the nested function has access to variables in the file that contains it, 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 points generated by `fminsearch` in solving the optimization problem

$$\min_x f(x) = \min_x e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right).$$

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 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 directory 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 =

    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,:) =

   -1.0000    1.0000
   -1.0000    1.0000
   -1.0750    0.9000
   -1.0125    0.8500
```

The final row of points 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 all three optimization functions, `fminbnd`, `fminsearch`, and `fzero`.

The "Command-Line Display Headings" column of the table lists the headings, corresponding to the `optimValues` fields that are displayed at the command line 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 interrupt the current iteration of the optimization. You might want the output function to do this 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 might use the value of state to decide which tasks to perform at the current iteration.

```
switch state
    case 'init'
          % Setup for plots or guis
    case 'iter'
          % Make updates to plot or guis as needed.
    case 'interrupt'
          % Check conditions to see whether optimization
          % should quit.
        case 'done'
          % Cleanup of plots, guis, or final plot
otherwise
end
```

Stop Flag

The output argument stop is a flag that is true or false. The flag tells the optimization function whether the optimization should quit or continue. 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 GUI Input

If you design a GUI to perform optimizations, you can make the output function stop an optimization when a user clicks a **Stop** button on the GUI. The following code shows how to do this, assuming 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');
```

Plot Functions

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 while the algorithm executes. 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 8-14.

You can use the `PlotFcns` option with the following MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`

The predefined plot functions for these optimization functions are:

- `@optimplotx` plots the current point
- `@optimplotfval` plots the function value
- `@optimplotfunccount` plots the function count (not available for `fzero`)

To view or modify a predefined plot function, open the function file in the MATLAB Editor. For example, to view the function file for plotting the current point, enter:

```
edit optimplotx.m
```

Example: Plot Function

View the progress of a minimization using `fminsearch` with the plot function `@optimplotfval`:

1 Write a file for the objective function. For this example, use:

```
function f = onehump(x)

r = x(1)^2 + x(2)^2;
s = exp(-r);
```

```
f = x(1)*s+r/20;
```

2 Set the options to use the plot function:

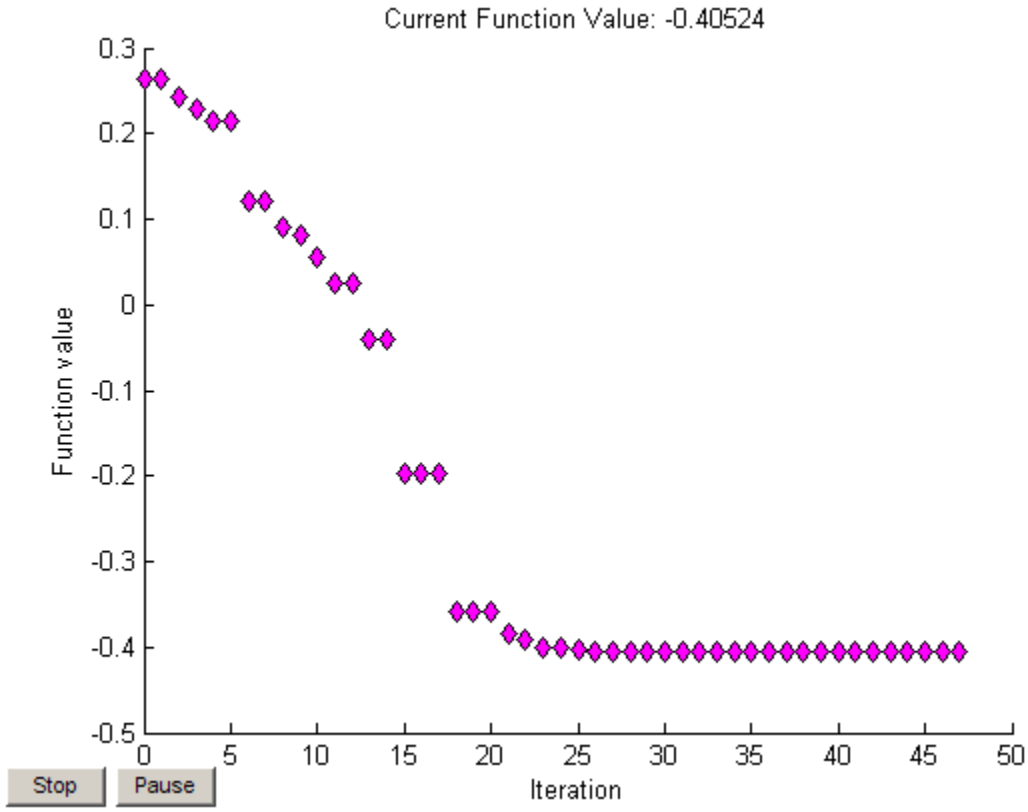
```
options = optimset('PlotFcns',@optimplotfval);
```

3 Call `fminsearch` starting from [2,1]:

```
[x ffinal] = fminsearch(@onehump,[2,1],options)
```

4 MATLAB returns the following:

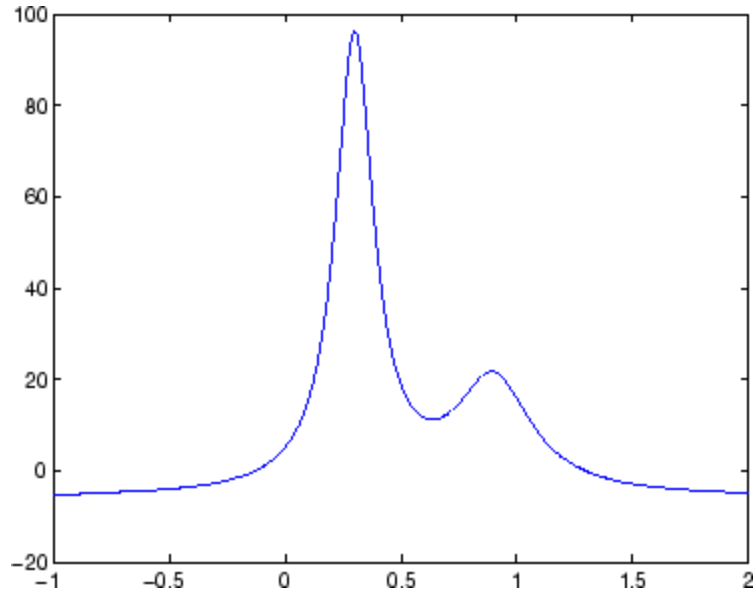
```
x =
   -0.6691    0.0000

ffinal =
   -0.4052
```


Current Function Value: -0.40524
Function value
Iteration
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
0
5
10
15
20
25
30
35
40
45
50
Stop
Pause

Finding Roots

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`.



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 function optimset.

```
options = optimset('Display','iter');
```

Then call fzero as follows:

```
a = fzero(@humps,[-1 1],options)
```

This returns the following iterative output:

```
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-007        interpolation
    11       -0.131618   -2.7935e-011        interpolation
    12       -0.131618   8.88178e-016        interpolation
    13       -0.131618   8.88178e-016        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-016
```

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)
```

`fzero` returns the following output:

```
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-005        interpolation
    22       -0.131618 -6.79505e-008        interpolation
    23       -0.131618 -2.98428e-013        interpolation
    24       -0.131618  8.88178e-016        interpolation
    25       -0.131618  8.88178e-016        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`.

Troubleshooting

Here is a list of typical problems and recommendations for dealing with them.

Problem	Recommendation
The solution found by `fminbnd` or `fminsearch` does not appear to be a global minimum.	There is no guarantee that you have a global minimum unless your problem is continuous and has only one minimum. Starting the optimization from a number of different starting points (or intervals in the case of `fminbnd`) may help to locate the global minimum or verify that there is only one minimum. Use different methods, where possible, to verify results.
Sometimes an optimization problem has values of `x` for which it is impossible to evaluate `f`.	Modify your function to include a penalty function to give a large positive value to `f` when infeasibility is encountered.
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`).	Your objective function (`fun`) may be returning `NaN` or complex values. The optimization routines expect only real numbers to be returned. Any other values may cause unexpected results. To determine whether this is the case, set `options = optimset('FunValCheck', 'on')` and call the optimization function with `options` as an input argument. This displays an error when the objective function returns `NaN` or complex values.

Tips

Optimization problems may take many iterations to converge. Most optimization problems benefit from good starting guesses. Providing good starting guesses improves the execution efficiency and may help locate the global minimum instead of a local minimum.

Sophisticated problems are best solved by an evolutionary approach, whereby a problem with a smaller number of independent variables is solved first. Solutions from lower order problems can generally be used as starting points for higher order problems by using an appropriate mapping.

The use of simpler cost functions and less stringent termination criteria in the early stages of an optimization problem can also reduce computation time. Such an approach often produces superior results by avoiding local minima.

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.

9

Function Handles

- "Introduction" on page 9-2
- "Defining Functions In Files" on page 9-3
- "Anonymous Functions" on page 9-4
- "Example: Function Plotting Function" on page 9-5
- "Parameterizing Functions" on page 9-8

Introduction

MATLAB functions are written to named files or are produced at the command line as anonymous functions. In either case, a function handle is used to pass the function as an input argument to a function function.

Defining Functions In Files

Consider the example function:

$$f(x) = \frac{1}{(x-0.3)^2 + 0.01} + \frac{1}{(x-0.9)^2 + 0.04} - 6.$$

A file for the function is `humps.m`. Its contents look like this:

```
function y = humps(x)
y = 1./((x - 0.3).^2 + 0.01) ...
  + 1./((x - 0.9).^2 + 0.04) - 6;
```

To evaluate `humps` at `2.0`, first use `@` to obtain a function handle. Use the function handle in the same way you would use a function name to call a function:

```
fh = @humps;
fh(2.0)
ans =
   -4.8552
```

Anonymous Functions

Consider again the example function:

$$f(x) = \frac{1}{(x-0.3)^2 + 0.01} + \frac{1}{(x-0.9)^2 + 0.04} - 6.$$

A second way to represent the function in the MATLAB is to create an anonymous function at the command line, as follows:

```
fh = @(x)(1./((x-0.3).^2 + 0.01) ...
          + 1./((x-0.9).^2 + 0.04)-6);
```

You evaluate `fh` at `2.0` the same way you do with a function handle for a file-defined function:

```
fh(2.0)
ans =
   -4.8552
```

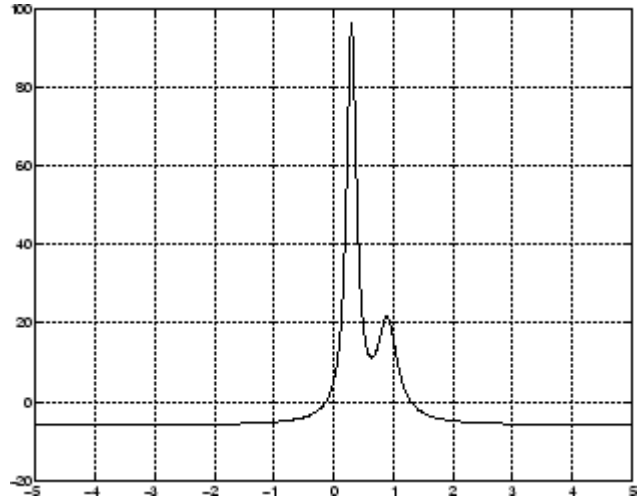
Anonymous functions can have any number of arguments. The following anonymous function has two input arguments `x` and `y`:

```
fh = @(x,y)(y*sin(x)+x*cos(y));
fh(pi,2*pi)
ans =
    3.1416
```

Example: Function Plotting Function

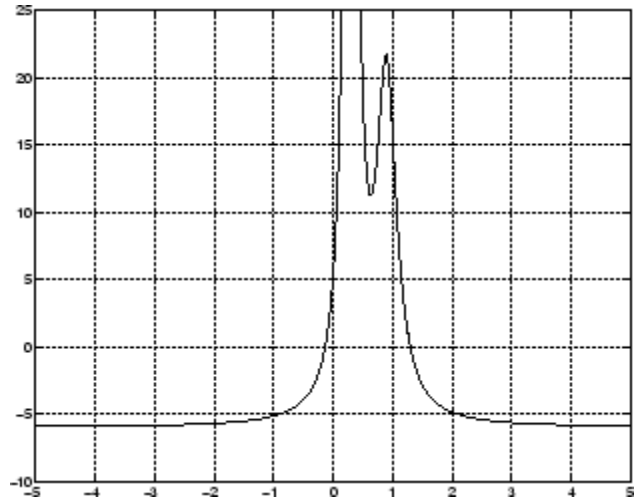
The `fplot` function plots a mathematical function between a given set of axes limits. You can control the *x*-axis limits only, or both the *x*- and *y*-axis limits. For example, to plot the `humps.m` function over the *x*-axis range `[-5 5]`, use

```
fplot(@humps,[-5 5])
grid on
```



You can zoom in on the function by selecting *y*-axis limits of `-10` and `25`, using

```
fplot(@humps,[-5 5 -10 25])
grid on
```



You can also pass the function handle for an anonymous function for `fplot` to graph, as in

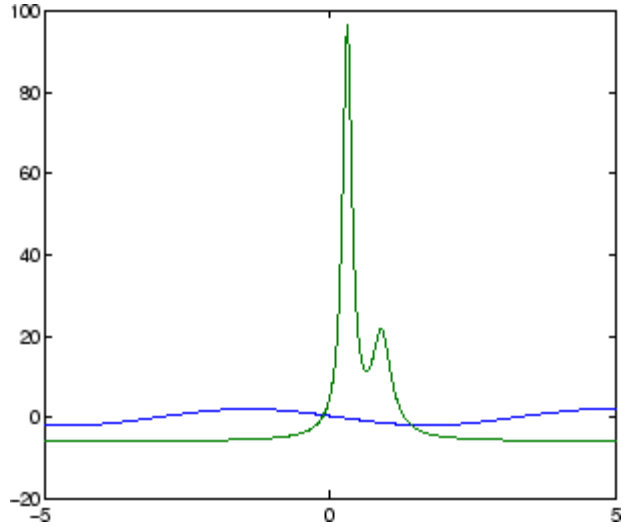
```
fplot(@(x)2*sin(x+3),[-1 1]);
```

You can plot more than one function on the same graph with one call to `fplot`. If you use this with a function, then the function must take a column vector `x` and return a matrix where each column corresponds to each function, evaluated at each value of `x`.

If you pass an anonymous function consisting of several functions to `fplot`, the anonymous function also must return a matrix where each column corresponds to each function evaluated at each value of `x`, as in

```
fplot(@(x)[2*sin(x+3),humps(x)],[-5 5])
```

which plots the first and second functions on the same graph.



Note that the anonymous function

```
fh = @(x)[2*sin(x+3),humps(x)];
```

evaluates to a matrix of two columns, one for each function, when `x` is a column vector.

```
fh([1;2;3])
```

returns

```
    -1.5136   16.0000
    -1.9178   -4.8552
    -0.5588   -5.6383
```

Parameterizing Functions

In this section...
"Using Nested Functions" on page 9-8
"Using Anonymous Functions" on page 9-9

Using Nested Functions

One way to provide additional parameters to a functional argument of a function function is to write a file that

- Accepts the additional parameters as inputs
- Invokes the function function
- Contains the function called by the function function as a nested function

The following example illustrates how to find a zero of the cubic polynomial $x^3 + bx + c$, for different values of the coefficients `b` and `c`, using this method. To do so, write a file with the following code:

```
function y = findzero(b, c, x0)

options = optimset('Display', 'off'); % Turn off Display
y = fzero(@poly, x0, options);

 function y = poly(x) % Compute the polynomial.
 y = x^3 + b*x + c;
 end
end
```

The main function, `findzero`, does two things:

- Invokes the function `fzero` to find a zero of the polynomial
- Computes the polynomial in a nested function, `poly`, which is called by `fzero`

You can call `findzero` with any values of the coefficients `b` and `c`, which are seen by `poly` because it is a nested function.

As an example, to find a zero of the polynomial with `b = 2` and `c = 3.5`, using the starting point `x0 = 0`, call `findzero` as follows.

```
x = findzero(2, 3.5, 0)
```

This returns the zero

```
x =

   -1.0945
```

Using Anonymous Functions

Suppose you have already written a standalone file for the function `poly` containing the following code, which computes the polynomial for any coefficients `b` and `c`,

```
function y = poly(x, b, c) % Compute the polynomial.
y = x^3 + b*x + c;
```

You then want to find a zero for the coefficient values `b = 2` and `c = 3.5`. You cannot simply apply `fzero` to `poly`, which has three input arguments, because `fzero` only accepts functions with a single input argument.

As an alternative to rewriting `poly` as a nested function (see) you can pass `poly` to `fzero` as a function handle to an anonymous function that has the form `@(x) poly(x, b, c)`. The function handle has just one input argument `x`, so `fzero` accepts it.

```
b = 2;
c = 3.5;
x = fzero(@(x) poly(x, b, c), 0)
```

This returns the zero

```
x =

   -1.0945
```

explains how to create anonymous functions.

If you later decide to find a zero for different values of `b` and `c`, you must redefine the anonymous function using the new values. For example,

```
b = 4;
c = -1;
fzero(@(x) poly(x, b, c), 0)

ans =

    0.2463
```

For more complicated objective functions, it is usually preferable to write the function as a nested function.

10

Calculus

- "Ordinary Differential Equations" on page 10-2
- "Delay Differential Equations" on page 10-47
- "Boundary-Value Problems" on page 10-59
- "Partial Differential Equations" on page 10-87
- "Selected Bibliography" on page 10-105
- "Integration" on page 10-106

Ordinary Differential Equations

In this section...
"Function Summary" on page 10-2
"Initial Value Problems" on page 10-4
"Types of Solvers" on page 10-6
"Solver Syntax" on page 10-8
"Integrator Options" on page 10-9
"Examples" on page 10-10
"Troubleshooting" on page 10-39

Function Summary

- "ODE Solvers" on page 10-2
- "Evaluation and Extension" on page 10-3
- "Solver Options" on page 10-3
- "Output Functions" on page 10-4

ODE Solvers

The following table lists the initial value problem solvers, the kind of problem you can solve with each solver, and the method each solver uses.

Solver	Solves These Kinds of Problems	Method
`ode45`	Nonstiff differential equations	Runge-Kutta
`ode23`	Nonstiff differential equations	Runge-Kutta
`ode113`	Nonstiff differential equations	Adams

Solver	Solves These Kinds of Problems	Method
`ode15s`	Stiff differential equations and DAEs	NDFs (BDFs)
`ode23s`	Stiff differential equations	Rosenbrock
`ode23t`	Moderately stiff differential equations and DAEs	Trapezoidal rule
`ode23tb`	Stiff differential equations	TR-BDF2
`ode15i`	Fully implicit differential equations	BDFs

Evaluation and Extension

You can use the following functions to evaluate and extend solutions to ODEs.

Function	Description
`deval`	Evaluate the numerical solution using the output of ODE solvers.
`odextend`	Extend the solution of an initial value problem for an ODE

Solver Options

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

Function	Description
`odeset`	Create or alter options structure for input to ODE solver.
`odeget`	Extract properties from options structure created with `odeset`.

Output Functions

If an output function is specified, the solver calls the specified function after every successful integration step. You can use `odeset` to specify one of these sample functions as the OutputFcn property, or you can modify them to create your own functions.

Function	Description
`odeplot`	Time-series plot
`odephas2`	Two-dimensional phase plane plot
`odephas3`	Three-dimensional phase plane plot
`odeprint`	Print to command window

Initial Value Problems

- "First Order ODEs" on page 10-4
- "Higher Order ODEs" on page 10-5
- "Initial Values" on page 10-5

First Order ODEs

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 derivative of y with respect to t is denoted as y', the second derivative as y'', and so on. Often $y(t)$ is a vector, having elements $y_1, y_2, ..., y_n$.

MATLAB solvers handle the following 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 matrix
- Fully implicit ODEs of the form $f(t, y, y') = 0$ (`ode15i` only)

Higher Order ODEs

MATLAB ODE solvers accept only first-order differential equations. To use the solvers with higher-order ODEs, you must rewrite each equation as an equivalent system of first-order differential equations of the form

$$y' = f(t, y)$$

You can write any ordinary differential equation

$$y^{(n)} = f(t, y, y', \ldots, y^{(n-1)})$$

as a system of first-order equations by making the substitutions

$$y_1 = y, \quad y_2 = y', \quad \ldots, \quad y_n = y^{(n-1)}$$

The result is an equivalent system of n first-order ODEs.

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ &\vdots \\ y_n' &= f(t, y_1, y_2, \ldots, y_n) \end{aligned}$$

Rewrite the second-order van der Pol equation

$$y_1'' - \mu(1 - y_1^2) y_1' + y_1 = 0$$

as a system of first-order ODEs.

Initial Values

Generally there are many functions $y(t)$ that satisfy a given ODE, and additional information is necessary to specify the solution of interest. In an *initial value problem,* the solution of interest satisfies a specific *initial condition*, that is, y is equal to y_0 at a given initial time t_0. An initial value problem for an ODE is then

$$y' = f(t, y)$$
$$y(t_0) = y_0$$

If the function $f(t, y)$ is sufficiently smooth, this problem has one and only one solution. Generally there is no analytic expression for the solution, so it is necessary to approximate $y(t)$ by numerical means.

Types of Solvers

- "Nonstiff Problems" on page 10-6
- "Stiff Problems" on page 10-7
- "Fully Implicit ODEs" on page 10-7

Nonstiff Problems

There are three solvers designed for nonstiff problems:

`ode45`	Based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver – in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, `ode45` is the best function to apply as a "first try" for most problems.
`ode23`	Based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than `ode45` at crude tolerances and in the presence of mild stiffness. Like `ode45`, `ode23` is a one-step solver.
`ode113`	Variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than `ode45` at stringent tolerances and when the ODE function is particularly expensive to evaluate. `ode113` is a *multistep* solver—it normally needs the solutions at several preceding time points to compute the current solution.

Stiff Problems

Not all difficult problems are stiff, but all stiff problems are difficult for solvers not specifically designed for them. Solvers for stiff problems can be used exactly like the other solvers. However, you can often significantly improve the efficiency of these solvers by providing them with additional information about the problem. (See "Integrator Options" on page 10-9.)

There are four solvers designed for stiff problems:

`ode15s`	Variable-order solver based on the numerical differentiation formulas (NDFs). Optionally it uses the backward differentiation formulas, BDFs (also known as Gear's method). Like `ode113`, `ode15s` is a multistep solver. If you suspect that a problem is stiff or if `ode45` failed or was very inefficient, try `ode15s`.
`ode23s`	Based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than `ode15s` at crude tolerances. It can solve some kinds of stiff problems for which `ode15s` is not effective.
`ode23t`	An implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.
`ode23tb`	An implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order 2. Like `ode23s`, this solver may be more efficient than `ode15s` at crude tolerances.

Fully Implicit ODEs

The solver `ode15i` solves fully implicit differential equations of the form

$$f(t, y, y') = 0$$

using the variable order BDF method. The basic syntax for `ode15i` is

```
[t,y] = ode15i(odefun,tspan,y0,yp0,options)
```

The input arguments are

`odefun`	A function that evaluates the left side of the differential equation of the form $f(t, y, y') = 0$.
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0`, `yp0`	Vectors of initial conditions for $y(t_0)$ and $y'(t_0)$, respectively. The specified values must be consistent; that is, they must satisfy `f(t0,y0,yp0) = 0`.
`options`	Optional integration argument created using the `odeset` function. See the `odeset` reference page for details.

The output arguments contain the solution approximated at discrete points:

`t`	Column vector of time points
`y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

See the `ode15i` reference page for more information about these arguments.

Solver Syntax

All of the ODE solver functions, except for `ode15i`, share a syntax that makes it easy to try any of the different numerical methods, if it is not apparent which is the most appropriate. To apply a different method to the same problem, simply change the ODE solver function name. The simplest syntax, common to all the solver functions, is

```
[t,y] = solver(odefun,tspan,y0,options)
```

where *`solver`* is one of the ODE solver functions listed previously.

The basic input arguments are

`odefun`	Handle to a function that evaluates the system of ODEs. The function has the form `dydt = odefun(t,y)` where `t` is a scalar, and `dydt` and `y` are column vectors. See "Function Handles" in MATLAB Programming Fundamentals for more information.
`tspan`	Vector specifying the interval of integration. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`.
`y0`	Vector of initial conditions for the problem See also "Initial Value Problems" on page 10-4.
`options`	Structure of optional parameters that change the default integration properties. "Integrator Options" on page 10-9 tells you how to create the structure and describes the properties you can specify.

The output arguments contain the solution approximated at discrete points:

`t`	Column vector of time points
`y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

See the reference page for the ODE solvers for more information about these arguments.

Integrator Options

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

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

1. Create an options structure using the function `odeset` by entering

   ```
   options = odeset('RelTol', 1e-4);
   ```

2. Pass the `options` structure to the solver as follows:

   - For all solvers except `ode15i`, use the syntax

     ```
     [t,y] = solver(odefun,tspan,y0,options)
     ```

   - For `ode15i`, use the syntax

     ```
     [t,y] = ode15i(odefun,tspan,y0,yp0,options)
     ```

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

Examples

- "van der Pol Equation (Nonstiff)" on page 10-11
- "van der Pol Equation (Stiff)" on page 10-13
- "van der Pol Equation (Parameterizing the ODE)" on page 10-15
- "van der Pol Equation (Evaluating the Solution)" on page 10-16
- "Euler Equations (Nonstiff)" on page 10-16
- "Fully Implicit ODE" on page 10-18
- "Finite Element Discretization" on page 10-19
- "Large Stiff Sparse Problem" on page 10-22
- "Event Location" on page 10-25
- "Advanced Event Location" on page 10-28
- "Differential-Algebraic Equations" on page 10-31
- "Nonnegative Solutions" on page 10-34

• "Additional Examples" on page 10-38

van der Pol Equation (Nonstiff)

This example illustrates the steps for solving an initial value ODE problem:

1 **Rewrite the problem as a system of first-order ODEs.** Rewrite the van der Pol equation (second-order)

$$y_1'' - \mu(1-y_1^2)y_1' + y_1 = 0$$

where $\mu > 0$ is a scalar parameter, by making the substitution $y'_1 = y_2$. The resulting system of first-order ODEs is

$$y'_1 = y_2$$
$$y'_2 = \mu(1-y_1^2)y_2 - y_1$$

2 **Code the system of first-order ODEs.** Once you represent the equation as a system of first-order ODEs, you can code it as a function that an ODE solver can use. The function must be of the form

```
dydt = odefun(t,y)
```

Although `t` and `y` must be the function's two arguments, the function does not need to use them. The output `dydt`, a column vector, is the derivative of `y`.

The code below represents the van der Pol system in the function, `vdp1`. The `vdp1` function assumes that $\mu = 1$. The variables y_1 and y_2 are the entries `y(1)` and `y(2)` of a two-element vector.

```
function dydt = vdp1(t,y)
dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
```

Note that, although `vdp1` must accept the arguments `t` and `y`, it does not use `t` in its computations.

3 **Apply a solver to the problem.**

Decide which solver you want to use to solve the problem. Then call the solver and pass it the function you created to describe the first-order system of ODEs, the time interval on which you want to solve the problem, and an initial condition vector.

For the van der Pol system, you can use `ode45` on time interval `[0 20]` with initial values `y(1) = 2` and `y(2) = 0`.

```
[t,y] = ode45(@vdp1,[0 20],[2; 0]);
```

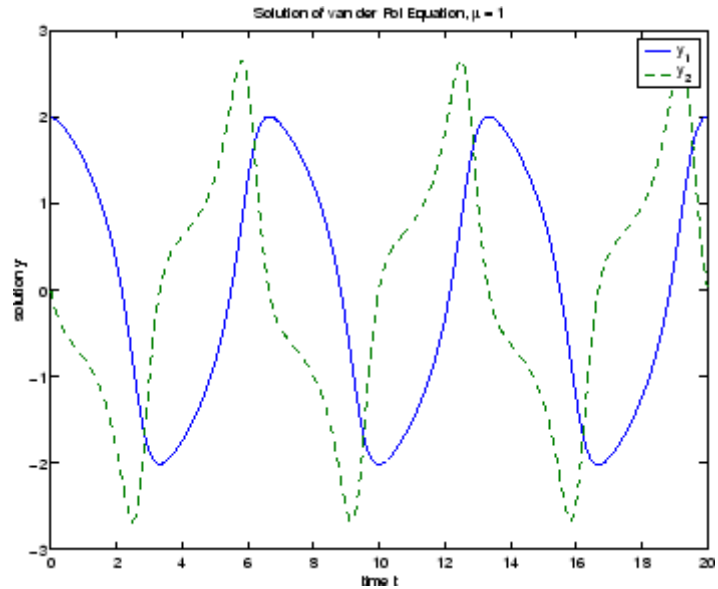
This example uses `@` to pass `vdp1` as a function handle to `ode45`. The resulting output is a column vector of time points `t` and a solution array `y`. Each row in `y` corresponds to a time returned in the corresponding row of `t`. The first column of `y` corresponds to y_1, and the second column to y_2.

Note For information on function handles, see the `function_handle` (@), `func2str`, and `str2func` reference pages, and the "Function Handles" section in MATLAB Programming Fundamentals.

---

4 **View the solver output.** You can simply use the `plot` command to view the solver output.

```
plot(t,y(:,1),'-',t,y(:,2),'--')
title('Solution of van der Pol Equation, \mu = 1');
xlabel('time t');
ylabel('solution y');
legend('y_1','y_2')
```



As an alternative, you can use a solver output function to process the output. The solver calls the function specified in the integration property `OutputFcn` after each successful time step. Use `odeset` to set `OutputFcn` to the desired function. See Solver Output Properties, in the reference page for `odeset`, for more information about `OutputFcn`.

van der Pol Equation (Stiff)

This example presents a stiff problem. For a stiff problem, solutions can change on a time scale that is very short compared to the interval of integration, but the solution of interest changes on a much longer time scale. Methods not designed for stiff problems are ineffective on intervals where the solution changes slowly because they use time steps small enough to resolve the fastest possible change.

When μ is increased to 1000, the solution to the van der Pol equation changes dramatically and exhibits oscillation on a much longer time scale. Approximating the solution of the initial value problem becomes a more difficult task. Because this particular problem is stiff, a solver intended for nonstiff problems, such as `ode45`, is too inefficient to be practical. A solver such as `ode15s` is intended for such stiff problems.

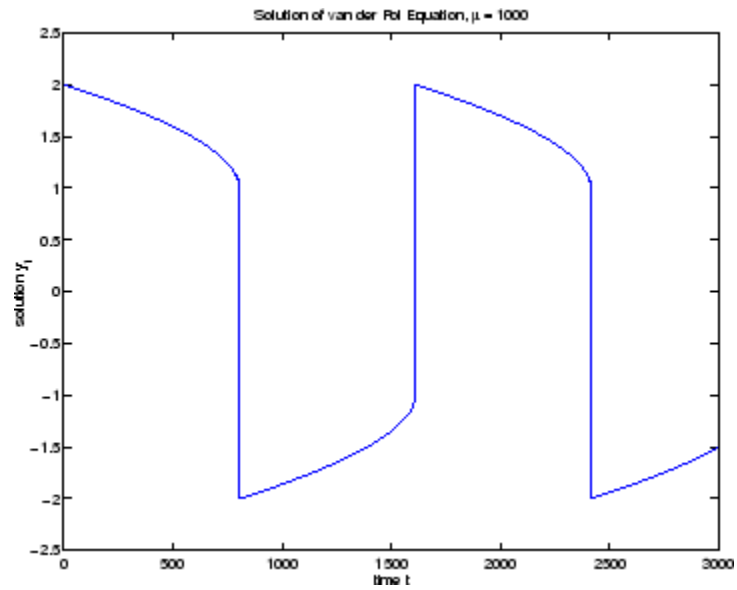
The `vdp1000` function evaluates the van der Pol system from the previous example, but with $\mu = 1000$.

```
function dydt = vdp1000(t,y)
dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

Note This example hardcodes μ in the ODE function. The `vdpode` example solves the same problem, but passes a user-specified μ as a parameter to the ODE function.

Now use the `ode15s` function to solve the problem with the initial condition vector of `[2; 0]`, but a time interval of `[0 3000]`. For scaling reasons, plot just the first component of `y(t)`.

```
[t,y] = ode15s(@vdp1000,[0 3000],[2; 0]);
plot(t,y(:,1),'-');
title('Solution of van der Pol Equation, \mu = 1000');
xlabel('time t');
ylabel('solution y_1');
```



van der Pol Equation (Parameterizing the ODE)

The preceding sections showed how to solve the van der Pol equation for two different values of the parameter μ. In those examples, the values μ = 1 and μ=1000 are hard-coded in the ODE functions. If you are solving an ODE for several different parameter values, it might be more convenient to include the parameter in the ODE function and assign a value to the parameter each time you run the ODE solver. This section explains how to do this for the van der Pol equation.

One way to provide parameter values to the ODE function is to write a MATLAB file that

- Accepts the parameters as inputs.
- Contains ODE function as a nested function, internally using the input parameters.
- Calls the ODE solver.

The following code illustrates this:

```
function [t,y] = solve_vdp(mu)
tspan = [0 max(20, 3*mu)];
y0 = [2; 0];

% Call the ODE solver ode15s.
[t,y] = ode15s(@vdp,tspan,y0);

    % Define the ODE function as nested function,
    % using the parameter mu.
    function dydt = vdp(t,y)
    dydt = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
    end
end
```

Because the ODE function `vdp` is a nested function, the value of the parameter `mu` is available to it.

To run the MATLAB file for `mu = 1`, enter

```
[t,y] = solve_vdp(1);
```

To run the code for μ = 1000, enter

```
[t,y] = solve_vdp(1000);
```

See the `vdpode` code for a complete example based on these functions.

van der Pol Equation (Evaluating the Solution)

The numerical methods implemented in the ODE solvers produce a continuous solution over the interval of integration $[a, b]$. You can evaluate the approximate solution, $S(x)$, at any point in $[a, b]$ using the function `deval` and the structure `sol` returned by the solver. For example, if you solve the problem described in "van der Pol Equation (Nonstiff)" on page 10-11 by calling `ode45` with a single output argument `sol`,

```
sol = ode45(@vdp1,[0 20],[2; 0]);
```

`ode45` returns the solution as a structure. You can then evaluate the approximate solution at points in the vector `xint = 1:5` as follows:

```
xint = 1:5;
Sxint = deval(sol,xint)

Sxint =

    1.5081    0.3235   -1.8686   -1.7407   -0.8344
   -0.7803   -1.8320   -1.0220    0.6260    1.3095
```

The `deval` function is vectorized. For a vector `xint`, the `i`th column of `Sxint` approximates the solution $y(\text{xint(i)})$.

Euler Equations (Nonstiff)

`rigidode` illustrates the solution of a standard test problem proposed by Krogh for solvers intended for nonstiff problems [8].

The ODEs are the Euler equations of a rigid body without external forces.

$$y_1' = y_2 y_3$$

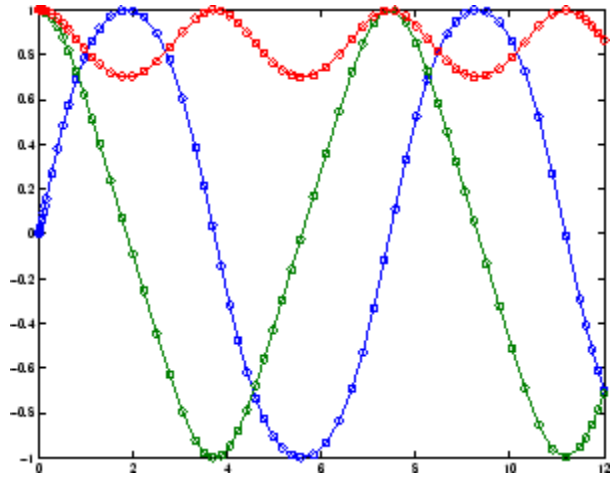
$$y_2' = -y_1 y_3$$

$$y_3' = -0.51 y_1 y_2$$

For your convenience, the entire problem is defined and solved in a single MATLAB file. The differential equations are coded as a subfunction `f`. Because the example calls the `ode45` solver without output arguments, the solver uses the default output function `odeplot` to plot the solution components.

To run this example, click the example name, or type `rigidode` at the command line.

```
function rigidode
%RIGIDODE  Euler equations: rigid body without external forces
tspan = [0 12];
y0 = [0; 1; 1];

% Solve the problem using ode45
ode45(@f,tspan,y0);
% --------------------------------------------------------------
function dydt = f(t,y)
dydt = [ y(2)*y(3)
        -y(1)*y(3)
        -0.51*y(1)*y(2) ];
```



Fully Implicit ODE

The following example shows how to use the function `ode15i` to solve the implicit ODE problem defined by Weissinger's equation

$$ty^2(y')^3 - y^3(y')^2 + t(t^2+1)y' - t^2y = 0$$

with the initial value $y(1) = \sqrt{3/2}$. The exact solution of the ODE is

$$y(t) = \sqrt{t^2 + 0.5}$$

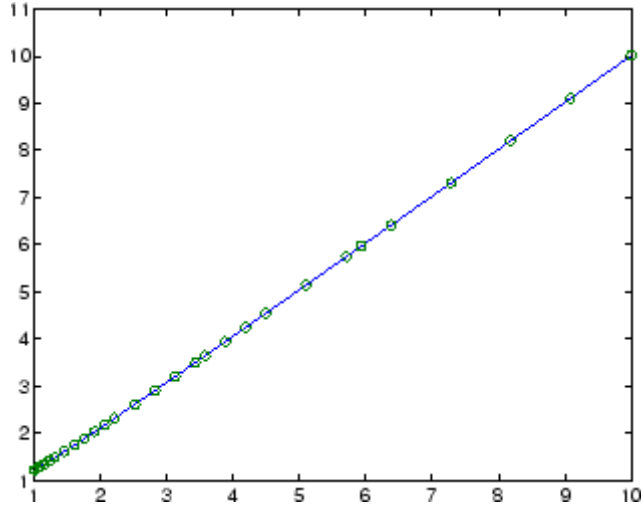
The example uses the function `weissinger`, which is provided with MATLAB, to compute the left-hand side of the equation.

Before calling `ode15i`, the example uses a helper function `decic` to compute a consistent initial value for $y'(t_0)$. In the following call, the given initial value $y(1) = \sqrt{3/2}$ is held fixed and a guess of 0 is specified for $y'(1)$. See the reference page for `decic` for more information.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0);
```

You can now call `ode15i` to solve the ODE and then plot the numerical solution against the analytical solution with the following commands.

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,t,ytrue,'o');
```



Finite Element Discretization

`fem1ode` illustrates the solution of ODEs that result from a finite element discretization of a partial differential equation. The value of `N` in the call `fem1ode(N)` controls the discretization, and the resulting system consists of `N` equations. By default, `N` is 19.

This example involves a mass matrix. The system of ODEs comes from a method of lines solution of the partial differential equation

$$e^{-t}\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

with initial condition $u(0,x) = \sin(x)$ and boundary conditions $u(t,0) = u(t,\pi) = 0$. An integer N is chosen, h is defined as $\pi/(N+1)$, and

the solution of the partial differential equation is approximated at $x_k = kh$ for $k = 0, 1, ..., N+1$ by

$$u(t,x_k) = \sum_{k=1}^{N} c_k(t)\phi_k(x)$$

Here $\phi_k(x)$ is a piecewise linear function that is 1 at x_k and 0 at all the other x_j. A Galerkin discretization leads to the system of ODEs

$$M(t)c' = Jc \quad \text{where} \quad c(t) = \begin{bmatrix} c_1(t) \\ \\ c_N(t) \end{bmatrix}$$

and the tridiagonal matrices $M(t)$ and J are given by

$$M_{ij} = \begin{cases} 2h/3 \exp(-t) & \text{if } i = j \\ h/6 \exp(-t) & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$J_{ij} = \begin{cases} -2/h & \text{if } i = j \\ 1/h & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

The initial values $c(0)$ are taken from the initial condition for the partial differential equation. The problem is solved on the time interval $[0, \pi]$.

In the `fem1ode` example, the properties

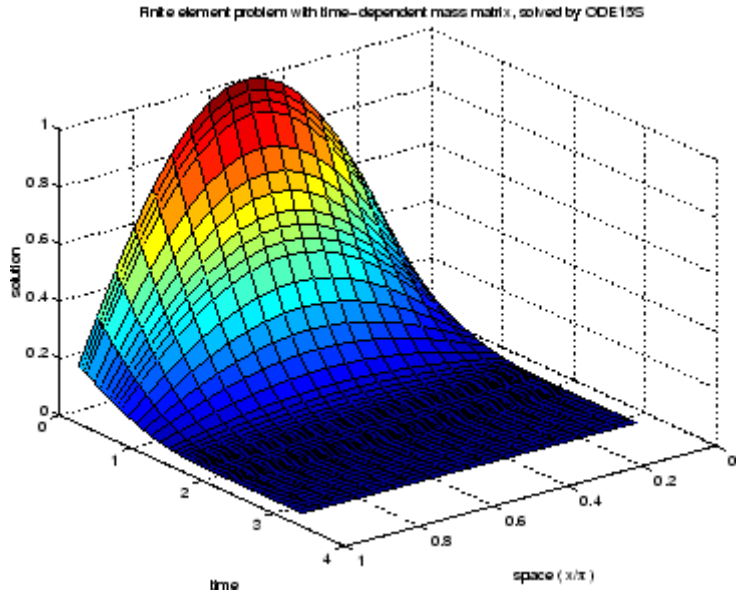
```
options = odeset('Mass',@mass,'MStateDep','none','Jacobian',J)
```

indicate that the problem is of the form $M(t)y' = Jy$. The nested function `mass(t)` evaluates the time-dependent mass matrix $M(t)$ and `J` is the constant Jacobian.

To run this example, click the example name, or type `fem1ode` at the command line. From the command line, you can specify a value of N as an argument to `fem1ode`. The default is $N = 19$.

```
function fem1ode(N)
%FEM1ODE  Stiff problem with a time-dependent mass matrix

if nargin < 1
  N = 19;
end
h = pi/(N+1);
y0 = sin(h*(1:N)');
tspan = [0; pi];

% The Jacobian is constant.
e = repmat(1/h,N,1);    %  e=[(1/h) ... (1/h)];
d = repmat(-2/h,N,1);   %  d=[(-2/h) ... (-2/h)];
% J is shared with the derivative function.
J = spdiags([e d e], -1:1, N, N);

d = repmat(h/6,N,1);
% M is shared with the mass matrix function.
M = spdiags([d 4*d d], -1:1, N, N);

options = odeset('Mass',@mass,'MStateDep','none', ...
                 'Jacobian',J);

[t,y] = ode15s(@f,tspan,y0,options);

figure;
surf((1:N)/(N+1),t,y);
set(gca,'ZLim',[0 1]);
view(142.5,30);
title(['Finite element problem with time-dependent mass ' ...
       'matrix, solved by ODE15S']);
xlabel('space ( x/\pi )');
ylabel('time');
zlabel('solution');
%-----------------------------------------------------------
function yp = f(t,y)
```

```
% Derivative function.
   yp = J*y;    % Constant Jacobian provided by outer function
end             % End nested function f
%-------------------------------------------------------------
function Mt = mass(t)
% Mass matrix function.
   Mt = exp(-t)*M;    % M is provided by outer function
end                   % End nested function mass
%-------------------------------------------------------------
end
```



Large Stiff Sparse Problem

`brussode` illustrates the solution of a potentially large stiff sparse problem. The problem is the classic “Brusselator“ system [3] that models diffusion in a chemical reaction

$$u_i' = 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1})$$
$$v_i' = 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1})$$

and is solved on the time interval `[0,10]` with $\alpha = 1/50$ and

$$\left.\begin{aligned} u_i(0) &= 1 + \sin(2\pi x_i) \\ v_i(0) &= 3 \end{aligned}\right\} \quad \text{with } x_i = i/(N+1), \text{ for } i = 1, \ldots, N$$

There are $2N$ equations in the system, but the Jacobian is banded with a constant width 5 if the equations are ordered as $u_1, v_1, u_2, v_2, \cdots$

In the call `brussode(N)`, where `N` corresponds to N, the parameter `N` ≥ 2 specifies the number of grid points. The resulting system consists of `2N` equations. By default, `N` is `20`. The problem becomes increasingly stiff and the Jacobian increasingly sparse as `N` increases.

The nested function `f(t,y)` returns the derivatives vector for the Brusselator problem. The subfunction `jpattern(N)` returns a sparse matrix of `1`s and `0`s showing the locations of nonzeros in the Jacobian $\partial f/\partial y$. The example assigns this matrix to the property `JPattern`, and the solver uses the sparsity pattern 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.

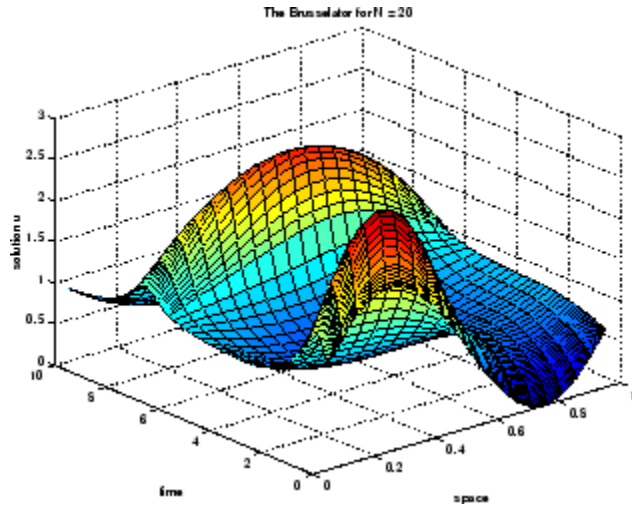
For the Brusselator problem, if the sparsity pattern is not supplied, `2N` evaluations of the function are needed to compute the `2N`-by-`2N` Jacobian matrix. If the sparsity pattern is supplied, only four evaluations are needed, regardless of the value of `N`.

To run this example, click on the example name, or type `brussode` at the command line. From the command line, you can specify a value of N as an argument to `brussode`. The default is $N = 20$.

```
function brussode(N)
%BRUSSODE  Stiff problem modeling a chemical reaction

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);
surf(x,t,u);
view(-40,30);
xlabel('space');
ylabel('time');
zlabel('solution u');
title(['The Brusselator for N = ' num2str(N)]);
% ------------------------------------------------------------
function dydt = f(t,y)
c = 0.02 * (N+1)^2;
dydt = zeros(2*N,size(y,2));      % preallocate dy/dt
% Evaluate the two 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 two 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 two 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 nested function f
end   % End function brussode
```

```
% ------------------------------------------------------------------
function S = jpattern(N)
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;
```



Event Location

`ballode` models the motion of a bouncing ball. This example illustrates the event location capabilities of the ODE solvers.

The equations for the bouncing ball are:

$$y_1' = y_2$$
$$y_2' = -9.8$$

In this example, the event function is coded in a subfunction `events`

```
[value,isterminal,direction] = events(t,y)
```

which returns

- A value of the event function
- The information whether or not the integration should stop when `value = 0` (`isterminal = 1` or `0`, respectively)
- The desired directionality of the zero crossings:

`-1`	Detect zero crossings in the negative direction only
`0`	Detect all zero crossings
`1`	Detect zero crossings in the positive direction only

The length of `value`, `isterminal`, and `direction` is the same as the number of event functions. The `i`th element of each vector, corresponds to the `i`th event function. For an example of more advanced event location, see `orbitode` ("Advanced Event Location" on page 10-28).

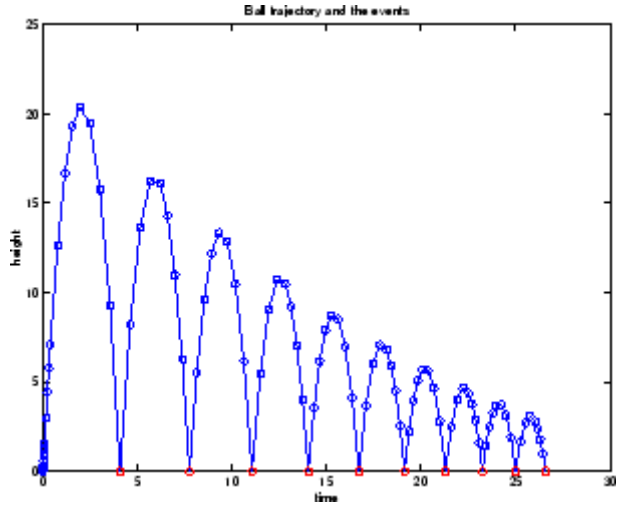
In `ballode`, setting the `Events` property to `@events` causes the solver to stop the integration (`isterminal = 1`) when the ball hits the ground (the height `y(1)` is `0`) during its fall (`direction = -1`). The example then restarts the integration with initial conditions corresponding to a ball that bounced.

To run this example, click on the example name, or type `ballode` at the command line.

```
function ballode
%BALLODE  Run a demo of a bouncing ball.

tstart = 0;
tfinal = 30;
y0 = [0; 20];
refine = 4;
options = odeset('Events',@events,'OutputFcn', @odeplot,...
                 'OutputSel',1,'Refine',refine);

set(gca,'xlim',[0 30],'ylim',[0 25]);
box on
hold on;

tout = tstart;
yout = y0.';
```

```
teout = [];
yeout = [];
ieout = [];
for i = 1:10
  % Solve until the first terminal event.
  [t,y,te,ye,ie] = ode23(@f,[tstart tfinal],y0,options);
  if ~ishold
    hold on
  end
  % Accumulate output.
  nt = length(t);
  tout = [tout; t(2:nt)];
  yout = [yout; y(2:nt,:)];
  teout = [teout; te];    % Events at tstart are never reported.
  yeout = [yeout; ye];
  ieout = [ieout; ie];

  ud = get(gcf,'UserData');
  if ud.stop
    break;
  end

  % Set the new initial conditions, with .9 attenuation.
  y0(1) = 0;
  y0(2) = -.9*y(nt,2);

  % A good guess of a valid first time step is the length of
  % the last valid time step, so use it for faster computation.
  options = odeset(options,'InitialStep',t(nt)-t(nt-refine),...
                           'MaxStep',t(nt)-t(1));
  tstart = t(nt);
end

plot(teout,yeout(:,1),'ro')
xlabel('time');
ylabel('height');
title('Ball trajectory and the events');
hold off
odeplot([],[],'done');
% --------------------------------------------------------------
```

```
function dydt = f(t,y)
dydt = [y(2); -9.8];
% ------------------------------------------------------------
function [value,isterminal,direction] = events(t,y)
% Locate the time when height passes through zero in a
% decreasing direction and stop integration.
value = y(1);     % Detect height = 0
isterminal = 1;   % Stop the integration
direction = -1;   % Negative direction only
```



Advanced Event Location

`orbitode` illustrates the solution of a standard test problem for those solvers that are intended for nonstiff problems. It traces the path of a spaceship traveling around the moon and returning to the earth (Shampine and Gordon [8], p. 246).

The `orbitode` problem is a system of the following four equations:

$$y_1' = y_3$$

$$y_2' = y_4$$

$$y_3' = 2y_4 + y_1 - \frac{\mu^*(y_1+\mu)}{r_1^3} - \frac{\mu(y_1-\mu^*)}{r_2^3}$$

$$y_4' = -2y_3 + y_2 - \frac{\mu^* y_2}{r_1^3} - \frac{\mu y_2}{r_2^3}$$

where

$$\mu = 1/82.45$$

$$\mu^* = 1-\mu$$

$$r_1 = \sqrt{(y_1+\mu)^2 + y_2^2}$$

$$r_2 = \sqrt{(y_1-\mu^*)^2 + y_2^2}$$

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 initial conditions have been chosen to make the orbit periodic. The value of μ corresponds to a spaceship traveling around the moon and the earth. Moderately stringent tolerances are necessary to reproduce the qualitative behavior of the orbit. Suitable values are `1e-5` for `RelTol` and `1e-4` for `AbsTol`.

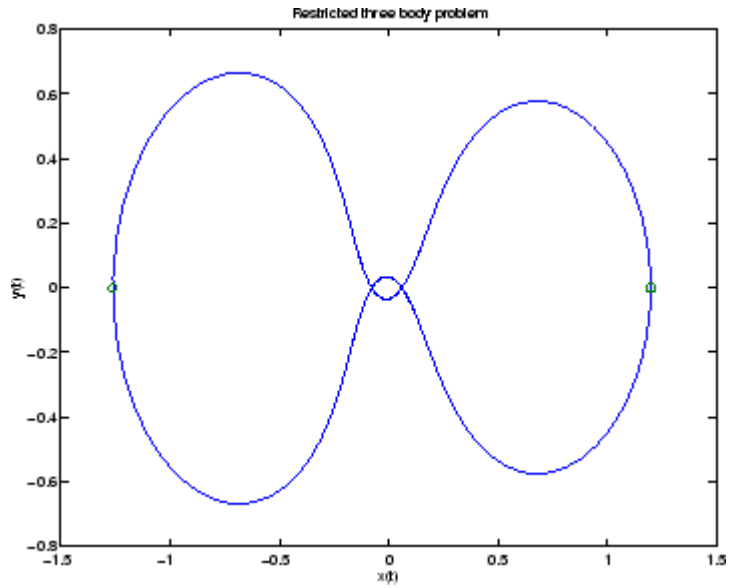
The nested `events` function includes event functions that locate the point of maximum distance from the starting point and the time the spaceship returns to the starting point. Note that the events are located accurately, even though the step sizes used by the integrator are *not* determined by the location 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 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.

To run this example, click on the example name, or type `orbitode` at the command line. The example uses the output function `odephas2` to produce

the two-dimensional phase plane plot and let you to see the progress of the integration.

```
function orbitode
%ORBITODE  Restricted three-body problem

mu = 1 / 82.45;
mustar = 1 - mu;
y0 = [1.2; 0; 0; -1.04935750983031990726];
tspan = [0 7];

options = odeset('RelTol',1e-5,'AbsTol',1e-4,...
                 'OutputFcn',@odephas2,'Events',@events);

[t,y,te,ye,ie] = ode45(@f,tspan,y0,options);

plot(y(:,1),y(:,2),ye(:,1),ye(:,2),'o');
title ('Restricted three body problem')
ylabel ('y(t)')
xlabel ('x(t)')
% ------------------------------------------------------------
function dydt = f(t,y)
r13 = ((y(1) + mu)^2 + y(2)^2) ^ 1.5;
r23 = ((y(1) - mustar)^2 + y(2)^2) ^ 1.5;
dydt = [ y(3)
         y(4)
         2*y(4) + y(1) - mustar*((y(1)+mu)/r13) - ...
                         mu*((y(1)-mustar)/r23)
        -2*y(3) + y(2) - mustar*(y(2)/r13) - mu*(y(2)/r23) ];
end   % End nested function f
% ------------------------------------------------------------
function [value,isterminal,direction] = events(t,y)
% Locate the time when the object returns closest to the
% initial point y0 and starts to move away; stop integration.
% Also locate the time when the object is farthest from the
% initial point y0 and starts to move closer.
%
% The current distance of the body is
%
%   DSQ = (y(1)-y0(1))^2 + (y(2)-y0(2))^2
```

```
%       = <y(1:2)-y0(1:2),y(1:2)-y0(1:2)>
%
% A local minimum of DSQ occurs when d/dt DSQ crosses zero
% heading in the positive direction. Compute d(DSQ)/dt as
%
%  d(DSQ)/dt = 2*(y(1:2)-y0(1:2))'*dy(1:2)/dt = ...
%                 2*(y(1:2)-y0(1:2))'*y(3:4)
%
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   % End nested function events
end
```



Differential-Algebraic Equations

`hb1dae` reformulates the `hb1ode` example as a *differential-algebraic equation* (DAE) problem. The Robertson problem coded in `hb1ode` is a classic test problem for codes that solve stiff ODEs.

$$y_1' = -0.04y_1 + 10^4 y_2 y_3$$

$$y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2$$

$$y_3' = 3 \cdot 10^7 y_2^2$$

Note The Robertson problem appears as an example in the prolog to LSODI [4].

In `hb1ode`, the problem is solved with initial conditions $y_1(0) = 1$, $y_2(0) = 0$, $y_3(0) = 0$ to steady state. These differential equations satisfy a linear conservation law that is used to reformulate the problem as the DAE

$$y_1' = -0.04y_1 + 10^4 y_2 y_3$$

$$y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2$$

$$0 = y_1 + y_2 + y_3 - 1$$

These equations do not have a solution for $y(0)$ with components that do not sum to 1. The problem has the form of $My' = f(t, y)$ with

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

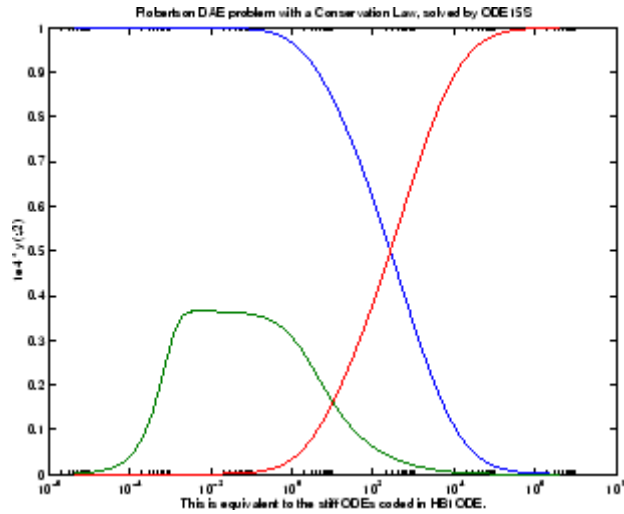
M is singular, but `hb1dae` does not inform the solver of this. The solver must recognize that the problem is a DAE, not an ODE. Similarly, although consistent initial conditions are obvious, the example uses an inconsistent value $y_3(0) = 10^{-3}$ to illustrate computation of consistent initial conditions.

To run this example, click on the example name, or type `hb1dae` at the command line. Note that `hb1dae`

- Imposes a much smaller absolute error tolerance on y_2 than on the other components. This is because y_2 is much smaller than the other components and its major change takes place in a relatively short time.
- Specifies additional points at which the solution is computed to more clearly show the behavior of y_2.
- Multiplies y_2 by 10^4 to make y_2 visible when plotting it with the rest of the solution.
- Uses a logarithmic scale to plot the solution on the long time interval.

```
function hb1dae
%HB1DAE  Stiff differential-algebraic equation (DAE)

% A constant, singular mass matrix
M = [1 0 0
     0 1 0
     0 0 0];

% Use inconsistent initial condition to test initialization.
y0 = [1; 0; 1e-3];
tspan = [0 4*logspace(-6,6)];

% Use the LSODI example tolerances.'MassSingular' is left
% at its default 'maybe' to test the automatic detection
% of a DAE.
options = odeset('Mass',M,'RelTol',1e-4,...
                 'AbsTol',[1e-6 1e-10 1e-6],...
                 'Vectorized','on');

[t,y] = ode15s(@f,tspan,y0,options);

y(:,2) = 1e4*y(:,2);

semilogx(t,y);
ylabel('1e4 * y(:,2)');
title(['Robertson DAE problem with a Conservation Law, '...
       'solved by ODE15S']);
xlabel('This is equivalent to the stiff ODEs in HB1ODE.');
% --------------------------------------------------------
function out = f(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 ];
```



Nonnegative Solutions

If certain components of the solution must be nonnegative, use `odeset` to set the `NonNegative` property for the indices of these components.

Note This option is not available for `ode23s`, `ode15i`, or for implicit solvers (`ode15s`, `ode23t`, `ode23tb`) applied to problems where there is a mass matrix.

Imposing nonnegativity is not always a trivial task. We suggest that you use this option only when necessary, for example in instances in which the application of a solution or integration will fail otherwise.

Consider the following initial value problem solved on the interval `[0, 40]`:

```
y' = - |y|, y(0) = 1
```

The solution of this problem decays to zero. If a solver produces a negative approximate solution, it begins to track the solution of the ODE through this

value, the solution goes off to minus infinity, and the computation fails. Using the `NonNegative` property prevents this from happening.

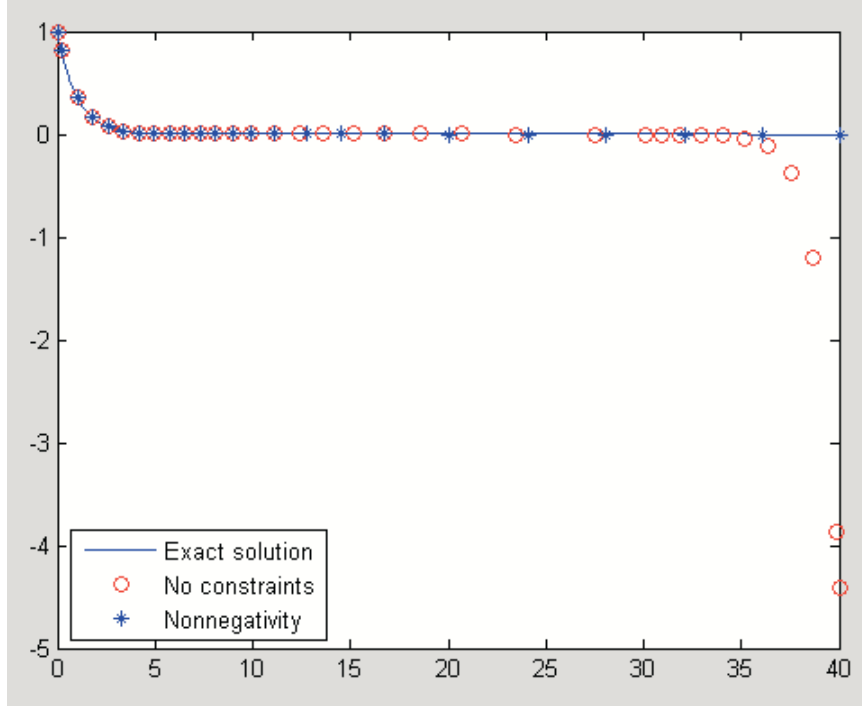
In this example, the first call to `ode45` uses the defaults for the solver parameters:

```
ode = @(t,y) -abs(y);
[t0,y0] = ode45(ode,[0, 40], 1);
```

The second uses options to impose nonnegativity conditions:

```
options = odeset('NonNegative',1);
[t1,y1] = ode45(ode,[0, 40], 1, options);
```

This plot compares the numerical solution to the exact solution.



Here is a more complete view of the code used to obtain this plot:

```
ode = @(t,y) -abs(y);
options = odeset('Refine',1);
[t0,y0] = ode45(ode,[0, 40], 1,options);
options = odeset(options,'NonNegative',1);
[t1,y1] = ode45(ode,[0, 40], 1, options);
t = linspace(0,40,1000);
y = exp(-t);
plot(t,y,'b-',t0,y0,'ro',t1,y1,'b*');
legend('Exact solution','No constraints','Nonnegativity', ...
       'Location','SouthWest')
```

The kneeode Demo. The MATLAB `kneeode` demo solves the "knee problem" by imposing a nonnegativity constraint on the numerical solution. The initial value problem is

```
 *y' = (1-x)*y - y^2,    y(0) = 1
```

For `0 < ε < 1`, the solution of this problem approaches null isoclines `y = 1 - x` and `y = 0` for `x < 1` and `x > 1`, respectively. The numerical solution, when computed with default tolerances, follows the `y = 1 - x` isocline for the whole interval of integration. Imposing nonnegativity constraints results in the correct solution.

Here is the code that makes up the `kneeode` demo:

```
function kneeode
%KNEEODE  The "knee problem" with Nonnegativity constraints.

% Problem parameter
epsilon = 1e-6;

y0 = 1;
xspan = [0, 2];

% Solve without imposing constraints
options = [];
[x1,y1] = ode15s(@odefcn,xspan,y0,options);

% Impose nonnegativity constraint
options = odeset('NonNegative',1);
```

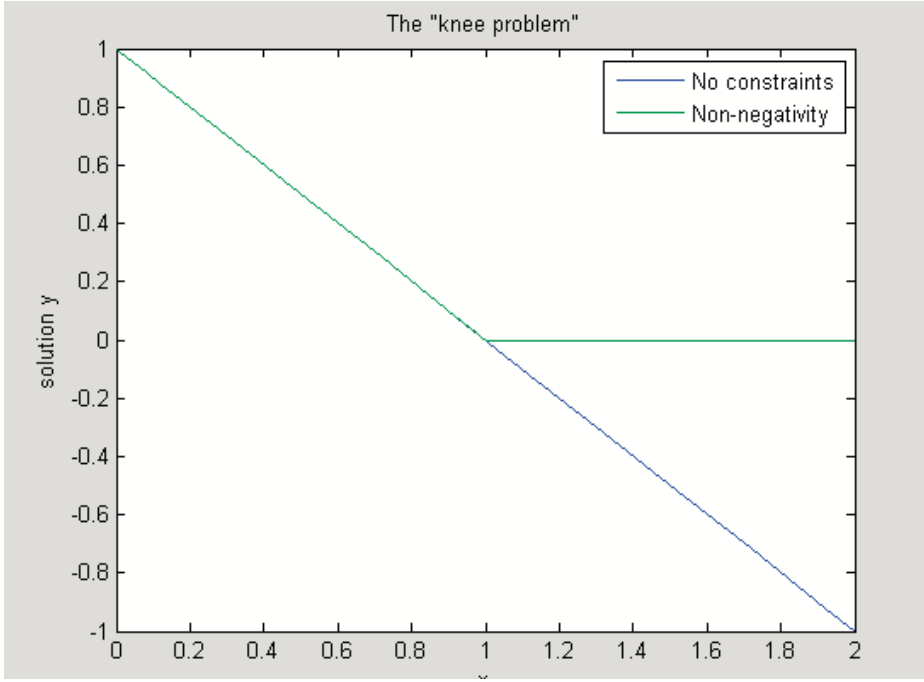
```
[x2,y2] = ode15s(@odefcn,xspan,y0,options);

figure
plot(x1,y1,'b.-',x2,y2,'g-')
axis([0,2,-1,1]);
title('The "knee problem"');
legend('No constraints','nonnegativity')
xlabel('x');
ylabel('solution y')

   function yp = odefcn(x,y)
      yp = ((1 - x)*y - y^2)/epsilon;
   end
end  % kneeode
```

The derivative function is defined within nested function `odefcn`. The value of epsilon used in `odefcn` is obtained from the outer function:

```
function yp = odefcn(x,y)
yp = ((1 - x)*y - y^2)/epsilon;
end
```

The demo solves the problem using the `ode15s` function, first with the default options, and then by imposing a nonnegativity constraint. To run the demo, type `kneeode` at the MATLAB command prompt.

Here is the output plot. The plot confirms correct solution behavior after imposing constraints.



Additional Examples

The following additional examples are available. Type

```
edit examplename
```

to view the code and

```
examplename
```

to run the example.

Example Name	Description
`amp1dae`	Stiff DAE — electrical circuit
`ballode`	Simple event location — bouncing ball
`batonode`	ODE with time- and state-dependent mass matrix — motion of a baton

Example Name	Description
`brussode`	Stiff large problem — diffusion in a chemical reaction (the Brusselator)
`burgersode`	ODE with strongly state-dependent mass matrix — Burgers' equation solved using a moving mesh technique
`fem1ode`	Stiff problem with a time-dependent mass matrix — finite element method
`fem2ode`	Stiff problem with a constant mass matrix — finite element method
`hb1ode`	Stiff ODE problem solved on a very long interval — Robertson chemical reaction
`hb1dae`	Robertson problem — stiff, linearly implicit DAE from a conservation law
`ihb1dae`	Robertson problem — stiff, fully implicit DAE
`iburgersode`	Burgers' equation solved as implicit ODE system
`kneeode`	The "knee problem" with nonnegativity constraints
`orbitode`	Advanced event location — restricted three body problem
`rigidode`	Nonstiff problem — Euler equations of a rigid body without external forces
`vdpode`	Parameterizable van der Pol equation (stiff for large μ)

Troubleshooting

- General Questions on page 10-40
- Memory and Computational Efficiency on page 10-40
- Time Steps for Integration on page 10-41
- Error Tolerance and Other Options on page 10-42

- Other Types of Equations on page 10-43
- Other Common Problems on page 10-44

General Questions

Question	Answer
How do the ODE solvers differ from `quad` or `quadl`?	`quad` and `quadl` solve problems of the form $y' = f(t)$. The ODE solvers handle more general problems $y' = f(t, y)$, linearly implicit problems that involve a mass matrix $M(t, y)\, y' = f(t, y)$, and fully implicit problems $f(t, y, y') = 0$.
Can I solve ODE systems in which there are more equations than unknowns, or vice versa?	No.

Memory and Computational Efficiency

Question	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 but also allocate an `n`-by-`n` Jacobian matrix. For these solvers it may be advantageous to use the sparse option. If the problem is nonstiff, or if you are using the sparse option, it may be possible to solve a problem with thousands of unknowns. In this case, however, storage of the result can be problematic. Try asking 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.

Memory and Computational Efficiency (Continued)

Question	Answer
I'm solving a very large system, but only care about a couple of the components of `y`. Is there any way to avoid storing all of the elements?	Yes. The user-installable output function capability 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` property. For example, if you repeatedly call the integrator in an event location loop, the last step that was taken before the event is probably on scale for the next integration. See `ballode` for an example.

Time Steps for Integration

Question	Answer
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` property. The integrator tries this value, then reduces it if necessary.
Can I integrate with fixed step sizes?	No.

Error Tolerance and Other Options

Question	Answer
How do I choose `RelTol` and `AbsTol`?	`RelTol`, the relative accuracy tolerance, controls the number of correct digits in the answer. `AbsTol`, the absolute error tolerance, controls the difference between the answer and the 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 except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` 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't I simply set `RelTol` to `eps`?	You can get close to machine precision, but not that close. The solvers do not allow `RelTol` near `eps` because they try to approximate a continuous function. At tolerances comparable to `eps`, the machine arithmetic causes all functions to look discontinuous.
How do I tell the solver that I don't care about getting an accurate answer for one of the solution components?	You can increase the absolute error tolerance corresponding to this solution component. If the tolerance is bigger than the component, this specifies no correct digits for the component. The solver may have to get some correct digits in this component to compute other components accurately, but it generally handles this automatically.

Other Types of Equations

Question	Answer
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 – the ODE solvers provide for this. Often the system is stiff. This is to be expected when the PDE is parabolic and 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 solvers: `ode15s`, `ode23s`, `ode23t`, `ode23tb`. If there are many equations, set the `JPattern` property. This might make the difference between success and failure due to the computation being too expensive. For an example that uses `JPattern`, see "Large Stiff Sparse Problem" on page 10-22. When the system is not stiff, or not very stiff, `ode23` or `ode45` is more efficient than `ode15s, ode23s`, `ode23t`, or `ode23tb`. Parabolic-elliptic partial differential equations in 1-D can be solved directly with the MATLAB PDE solver, `pdepe`. For more information, see "Partial Differential Equations" on page 10-87.
Can I solve differential-algebraic equation (DAE) systems?	Yes. The solvers `ode15s` and `ode23t` can solve some DAEs of the form $M(t,y)y' = f(t,y)$ where $M(t,y)$ is singular. The DAEs must be of index 1. `ode15i` can solve fully implicit DAEs of index 1, $f(t,y,y') = 0$. For examples, see `amp1dae`, `hb1dae`, or `ihb1dae`.

Other Types of Equations (Continued)

Question	Answer
Can I integrate a set of sampled data?	Not directly. You have to 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 like 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 this.
What do I do when I have the final and not the initial value?	All the solvers of the ODE suite allow you to solve backwards or forwards in time. The syntax for the solvers is `[t,y] = ode45(odefun,[t0 tf],y0);`and the syntax accepts `t0 > tf`.

Other Common Problems

Question	Answer
The solution doesn't look like what I expected.	If you're right about its appearance, you need to reduce the error tolerances from their default values. A smaller relative error tolerance is needed to compute accurately the solution of problems integrated over "long" intervals, as well as solutions of problems that are moderately unstable. You should check whether there are solution components that stay smaller than their absolute error tolerance for some time. If so, you are not asking for any correct digits in these components. This may be acceptable for these components, but failing to compute them

Other Common Problems (Continued)

Question	Answer
	accurately may degrade the accuracy of other components that depend on them.
My plots aren't smooth enough.	Increase the value of `Refine` from its default of `4` in `ode45` and `1` in the other solvers. The bigger the value of `Refine`, the more output points. Execution speed is not affected much by the value of `Refine`.
I'm 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 isn't, the solver must take small steps to deal with this. It may help to break `tspan` into pieces on 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 this purpose. Switch to `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.
My integration proceeds very slowly, using too many time steps.	First, check that your `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 the `tspan`, 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 `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

Other Common Problems (Continued)

Question	Answer
	the ODE function. Rather than recompute 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 there is a sharp change at time `t`, it might help to break the `tspan` interval into two pieces, `[t0 t]` and `[t tf]`, and call the integrator twice. If the differential equation has periodic coefficients or solution, you might restrict the maximum step size to the length of the period so the integrator won't step over periods.

Delay Differential Equations

In this section...
"Function Summary" on page 10-47
"Initial Value Problems" on page 10-48
"Types of Solvers" on page 10-49
"Discontinuities" on page 10-50
"Integrator Options" on page 10-51
"Examples" on page 10-51

Function Summary

- "DDE Solvers" on page 10-47
- "DDE Helper Functions" on page 10-47
- "DDE Solver Options" on page 10-48

DDE Solvers

Solver	Description
`dde23`	Solve initial value problems for delay differential equations with constant delays.
`ddesd`	Solve initial value problems for delay differential equations with general delays.

DDE Helper Functions

Function	Description
`deval`	Evaluate the numerical solution using the output of `dde23` or `ddesd`.

DDE Solver Options

Use these functions to create, alter, or access an options structure. An options structure contains named properties, the values of which are passed to `dde23` or `ddesd`, thus affecting the solution of the problem.

Function	Description
`ddeset`	Create/alter the DDE options structure.
`ddeget`	Extract properties from options structure created with `ddeset`.

Initial Value Problems

The DDE `dde23` solver can solve systems of ordinary differential equations, such as

$$y'(t) = f(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k))$$

where t is the independent variable, y is the dependent variable, and y' represents (derivative of y with respect to t)dy/dt. The delays (lags) $\tau_1, \ldots, \tau_k$ are positive constants. The solver `ddesd` allows delays that depend on t and y.

History and Initial Values

In an *initial value problem*, you seek the solution on an interval $[t_0, t_f]$ with $t_0 < t_f$. The DDE shows that $y'(t)$ depends on values of the solution at times prior to t. In particular, $y'(t_0)$ depends on $y(t_0 - \tau_1), \ldots, y(t_0 - \tau_k)$. Because of this, a solution on $[t_0, t_f]$ depends on its values for $t \le t_0$, i.e., its *history* $S(t)$.

Propagation of Discontinuities

Generally, the solution $y(t)$ of an IVP for a system of DDEs has a jump in its first derivative at the initial point t_0 because the first derivative of the history function does not satisfy the DDE there. A discontinuity in any derivative propagates into the future at spacings of $\tau_1, \tau_2, \ldots, \tau_k$ when the delays are constant, and in a more complicated way when they are not. For the DDEs solved by `dde23` and `ddesd`, the solution becomes smoother as the integration proceeds.

Types of Solvers

This section describes:

- "DDE Solver dde23" on page 10-49
- "DDE Solver ddesd" on page 10-49

The basic syntax for the two solvers is shown in the function reference pages for `dde23` and `ddesd`.

DDE Solver dde23

The function `dde23` solves initial value problems for DDEs with constant delays. It integrates a system of first-order differential equations

$$y'(t) = f(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k))$$

on the interval $[t_0, t_f]$, with $t_0 < t_f$ and given history $y(t) = S(t)$ for $t \le t_0$.

`dde23` produces a solution that is continuous on $[t_0, t_f]$. You can use the function `deval` and the output of `dde23` to evaluate the solution at specific points on the interval of integration.

`dde23` tracks low-order discontinuities and integrates the differential equations with the explicit Runge-Kutta (2,3) pair and interpolant used by `ode23`. The Runge-Kutta formulas are implicit for step sizes longer than the delays. When the solution is smooth enough that steps this big are justified, the implicit formulas are evaluated by a predictor-corrector iteration.

DDE Solver ddesd

The function `ddesd` solves initial value problems for DDEs with general delays. It integrates a system of first-order differential equations

$$y'(t) = f(t, y(t), y(d(1)), \ldots, y(d(k)))$$

on the interval $[t_0, t_f]$, with $t_0 < t_f$, where delays $d(j)$ can depend on both t and $y(t)$. Use the function `deval` and the output of `ddesd` to evaluate the solution at specific points on the interval of integration.

`ddesd` integrates with the classic four-stage, fourth-order explicit Runge-Kutta method, and controls the size of the residual of a natural interpolant. It uses iteration to take steps that are longer than the delays. For further details, see "Solving ODEs and DDEs with Residual Control," L.F. *Shampine, Applied Numerical Mathematics*, 52 (2005), pp 113-127.

Discontinuities

`dde23` performs better if it is informed of discontinuities in the history and at known locations. Discontinuities may be specified by event functions. There is a property with which you can specify a solution that is different from the value given by the history function.

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, which is to say that the solution is continuous at the initial point. However, 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` and `ddesd` 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

Discontinuity	Property	Comments
		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.

Integrator Options

The default integration properties in the DDE solver `dde23` are selected to handle common problems. In some cases, you can improve solver performance by overriding these defaults. You do this by supplying `dde23` with an `options` structure that specifies one or more property values.

For example, to change the relative error tolerance of `dde23` from the default value of `1e-3` to `1e-4`,

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

```
options = ddeset('RelTol', 1e-4);
```

2 Pass the `options` structure to `dde23` as follows:

```
sol = dde23(ddefun,lags,history,tspan,options)
```

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

Examples

- "Constant Delay" on page 10-52
- "State-Dependent Delay" on page 10-54
- "Cardiovascular Model" on page 10-55
- "Additional Examples" on page 10-57

Constant Delay

- "Solving the System" on page 10-52
- "Evaluating the Solution" on page 10-54

Solving the System. This example illustrates the straightforward formulation, computation, and display of the solution of a system of DDEs with constant delays. The history is constant, which is often the case. 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 example solves the equations on [0,5] with history

$$y_1(t) = 1$$
$$y_2(t) = 1$$
$$y_3(t) = 1$$

for $t \le 0$.

Note The demo `ddex1` contains the complete code for this example. To see the code in an editor, click the example name, or type `edit ddex1` at the command line. To run the example type `ddex1` at the command line.

---

1 **Rewrite the problem as a first-order system.** To use `dde23`, you must rewrite the equations as an equivalent system of first-order differential equations. Do this just as you would when solving IVPs and BVPs for ODEs. However, this example needs no such preparation because it already has the form of a first-order system of equations.

2 **Identify the lags.** The delays (lags) $\tau_1, \ldots, \tau_k$ are supplied to `dde23` as a vector. For the example we could use

```
lags = [1,0.2];
```

In coding the differential equations, τ_j = `lags(j)`.

3 **Code the system of first-order DDEs.** Once you represent the equations as a first-order system, and specify lags, you can code the equations as a function that `dde23` can use.

This code represents the system in the function, `ddex1de`.

```
function dydt = ddex1de(t,y,Z)
ylag1 = Z(:,1);
ylag2 = Z(:,2);
dydt = [ylag1(1)
        ylag1(1) + ylag2(2)
        y(2)               ];
```

4 **Code the history function.** The history function for this example is

```
function S = ddex1hist(t)
S = ones(3,1);
```

5 **Apply the DDE solver.** The example now calls `dde23` with the functions `ddex1de` and `ddex1hist`.

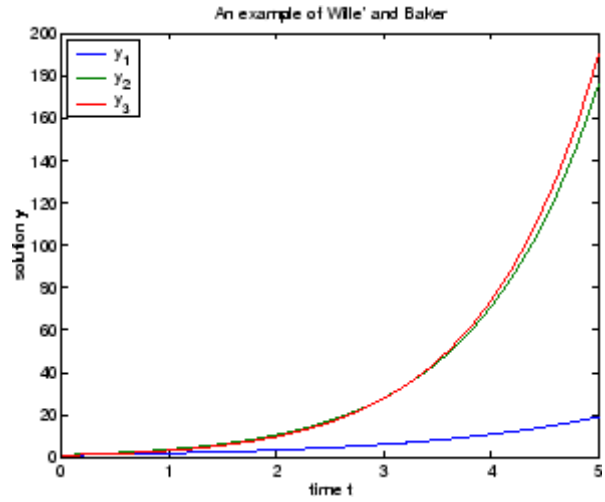
```
sol = dde23(@ddex1de,lags,@ddex1hist,[0,5]);
```

Here the example supplies the interval of integration [0,5] directly. Because the history is constant, we could also call `dde23` by

```
sol = dde23(@ddex1de,lags,ones(3,1),[0,5]);
```

6 **View the results.** Complete the example by displaying the results. `dde23` returns the mesh it selects and the solution there as fields in the solution structure `sol`. Often, these provide a smooth graph.

```
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',2)
```



Evaluating the Solution. The method implemented in `dde23` produces a continuous solution over the whole interval of integration $[t_0, t_f]$. You can evaluate the approximate solution, $S(t)$, at any point in $[t_0, t_f]$ using the helper function `deval` and the structure `sol` returned by `dde23`.

```
Sint = deval(sol,tint)
```

The `deval` function is vectorized. For a vector `tint`, the `i`th column of `Sint` approximates the solution $y(\text{tint(i)})$.

Using the output `sol` from the previous example, this code evaluates the numerical solution at 100 equally spaced points in the interval [0,5] and plots the result.

```
tint = linspace(0,5);
Sint = deval(sol,tint);
plot(tint,Sint);
```

State-Dependent Delay

This example solves a system of two DDEs with state-dependent delay that was used as a test problem by W.H. Enright and H. Hayashi [10] because it has an analytical solution. The differential equations are

$$y'_1(t) = y_2(t)$$
$$y'_2(t) = -y_2(e^{(1-y_2(t))}) \times y_2(t)^2 \times e^{(1-y_2(t))}$$

The analytical solution

$$y_1(t) = \log(t)$$
$$y_2(t) = 1/t$$

is used as the history for $t \le 0.1$ and the equations are solved on `[0.1, 5]`. The only thing different about solving this example is that it must be solved with `ddesd` rather than `dde23`. This is 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. The delay is provided to `ddesd` with a function like

```
function d = ddex3delay(t,y)
% State dependent delay function for DDEX3
d = exp(1 - y(2));
```

Note The demo `ddex3` contains the complete code for this example. To see the code in an editor, click the example name, or type `edit ddex3` at the command line. To run the example type `ddex3` at the command line.

Cardiovascular Model

This example solves a cardiovascular model due to J. T. Ottesen [6]. The equations are integrated over the interval [0,1000]. The situation of interest is when the peripheral pressure R is reduced exponentially from its value of 1.05 to 0.84 beginning at t = 600.

This is a problem with one delay, a constant history, and three differential equations with fourteen physical parameters. It has a discontinuity in a low order derivative at `t = 600`.

Note The demo `ddex2` contains the complete code for this example. To see the code in an editor, click the example name, or type `edit ddex2` at the command line. To run the example type `ddex2` at the command line.

In ddex2, the fourteen physical parameters are set as fields in a structure p that is shared with nested functions. The function ddex2de for evaluating the equations begins with

```
function dydt = ddex2de(t,y,Z)
if t <= 600
   p.R = 1.05;
else
   p.R = 0.21 * exp(600-t) + 0.84;
end
.
.
.
```

Jumps Property. The peripheral pressure R is a continuous function of t, but it does not have a continuous derivative at t = 600. The example uses the Jumps property to inform dde23 about the location of this discontinuity.

```
opts = ddeset('Jumps',600);
```

After defining the delay tau and the constant history, the call is

```
sol = dde23(@ddex2de,tau,history,[0, 1000],opts);
```

The demo ddex2 plots only the third component, the heart rate, which shows a sharp change at t = 600.

Restarting. The example could have solved this problem by breaking it into two pieces

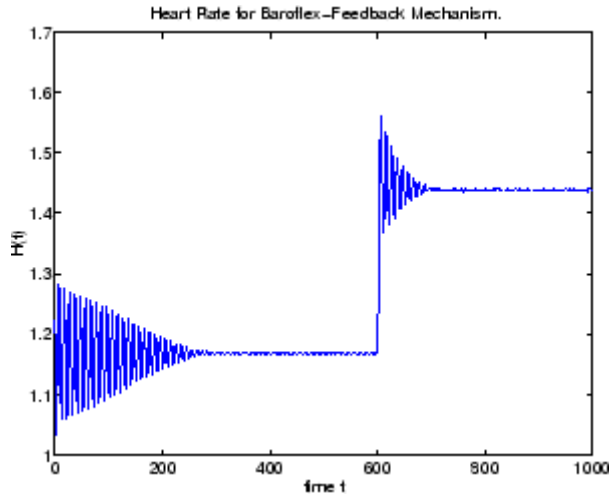
```
sol = dde23(@ddex2de,tau,history,[0, 600]);
sol = dde23(@ddex2de,tau,sol,[600, 1000]);
```

The solution structure sol on the interval [0,600] serves as history for restarting the integration at t = 600. In the second call, dde23 extends sol so that on return the solution is available on the whole interval [0,1000]. That is, after this second return,

```
Sint = deval(sol,[300,900]);
```

evaluates the solution obtained in the first integration at t = 300, and the solution obtained in the second integration at t = 900.

When discontinuities occur in low order derivatives at points known in advance, it is better to use the `Jumps` property. This is not an option with `ddesd`, which handles discontinuities in a different way. When you use event functions to locate such discontinuities, you must restart the integration at discontinuities.



Additional Examples

The following additional examples are available. Type

```
edit examplename
```

to view the code and

```
examplename
```

to run the example.

Example	Description
`ddex1`	Straightforward example
`ddex2`	Cardiovascular model with discontinuities
`ddex3`	Problem involving state-dependent delays

Additional examples are provided by "Tutorial on Solving DDEs with DDE23," available at `http://www.mathworks.com/dde_tutorial`.

Boundary-Value Problems

In this section...
"Function Summary" on page 10-59
"Boundary Value Problems" on page 10-60
"BVP Solver" on page 10-61
"Integrator Options" on page 10-64
"Examples" on page 10-65

Function Summary

- "BVP Solver" on page 10-59
- "BVP Helper Functions" on page 10-59
- "BVP Solver Options" on page 10-60

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 “First Order ODEs” on page 10-4 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" on page 10-61
- "BVP Solver Syntax" on page 10-62
- "BVP Solver Options" on page 10-64

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` Handle to a function 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. See "Function Handles" in MATLAB Programming Fundamentals for more information. odefun can also accept a vector of unknown parameters and a variable number of known parameters, (see "BVP Solver Options" on page 10-64).

`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 10-64).

`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 10-65
- "Continuation" on page 10-69
- "Singular BVPs" on page 10-77
- "Multipoint BVPs" on page 10-81
- "Additional Examples" on page 10-85

Mathieu's Equation

- "Solving the Problem" on page 10-65
- "Finding Unknown Parameters" on page 10-69
- "Evaluating the Solution" on page 10-69

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 lambdaλ 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

$$y(0) = 1$$
$$y'(0) = 0$$
$$y'(\pi) = 0$$

Note The demo `mat4bvp` contains the complete code for this example. The demo uses nested functions to place all functions required by `bvp4c` in a single MATLAB file. To run this example type `mat4bvp` at the command line. See "BVP Solver Syntax" on page 10-62 for more information.

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 $y = \cos 4x$ 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 `@` to pass `mat4init` as a function handle to `bvpinit`.

Note See the `function_handle` (`@`), `func2str`, and `str2func` reference pages, and see "Function Handles" in MATLAB Programming Fundamentals.

---

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:

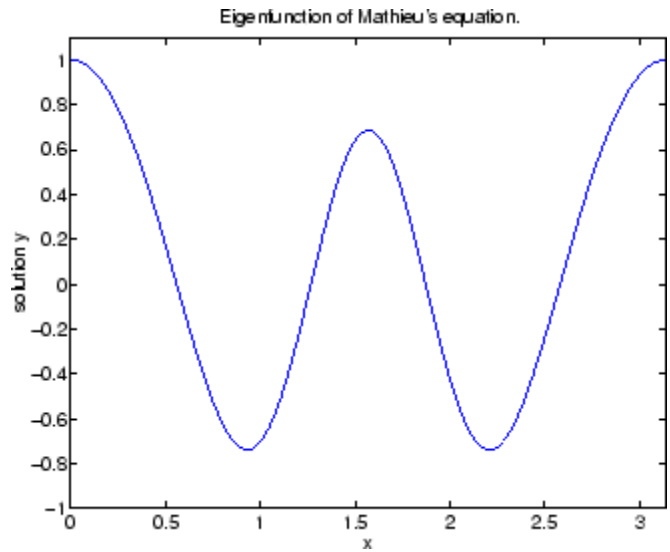
a Print the value of the unknown parameter λ found by `bvp4c`.

```
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 $[0, \pi]$, 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.



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 `i`th column of `Sxint` approximates the solution $y(\text{xint(i)})$.

Continuation

- "Introduction" on page 10-70
- "Using Continuation to Solve a BVP" on page 10-70
- "Using Continuation to Verify Consistency" on page 10-73

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

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

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 demo `shockbvp` contains the complete code for this example. The demo uses nested functions to place all required functions in a single MATLAB file. To run this example 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.

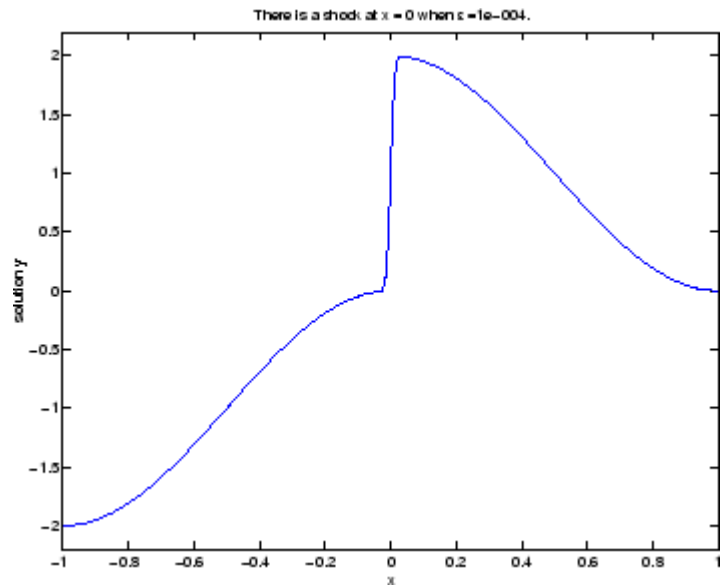
```
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
```

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')
```



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 $\beta = 0.5$ on the interval $[0, \infty)$ with boundary conditions $f(0) = 0$, $f'(0) = 0$, and $f'(\infty) = 1$.

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 demo `fsbvp` contains the complete code for this example. The demo uses nested functions to place all required functions in a single MATLAB file. To run this example 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);
```

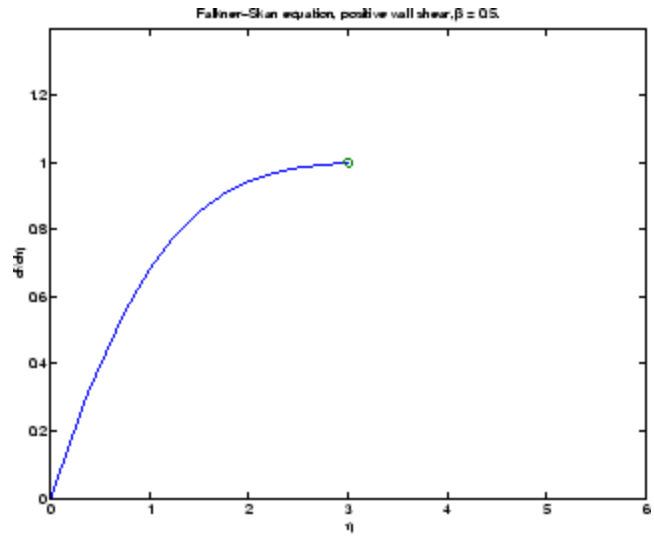
```
eta = sol.x;
f = sol.y;

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
```



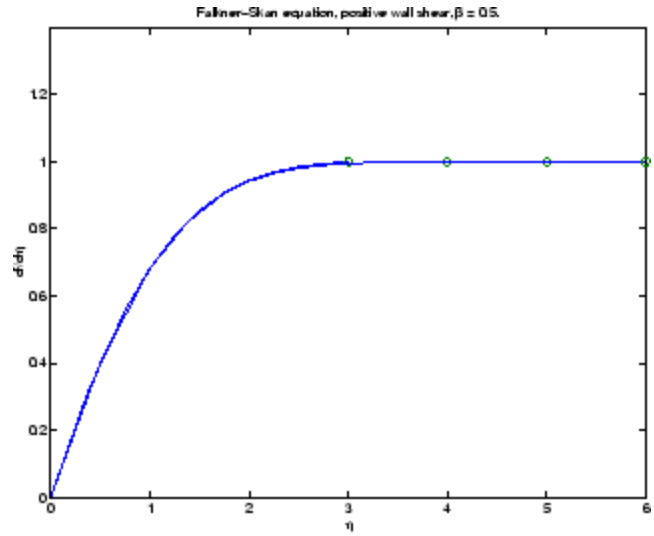
5 **Use continuation to solve the problem and plot subsequent solutions.** The example then solves the problem for `infinity = 4, 5, 6`. It uses `bvpinit` to extrapolate the solution `sol` for one value of `infinity` as an initial guess for the next value of `infinity`. For each iteration, the example 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', ...
           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.



Singular BVPs

- "Introduction" on page 10-77
- "Emden's equation" on page 10-78

Introduction. The function `bvp4c` solves a class of singular BVPs of the form

$$y' = \frac{1}{x}Sy + f(x, y)$$
$$0 = g(y(0), y(b)) \qquad \textbf{(10-1)}$$

It can also accommodate unknown parameters for problems of the form

$$y' = \frac{1}{x}Sy + f(x, y, p)$$
$$0 = g(y(0), y(b), p)$$

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 demo `emdenbvp` contains the complete code for this example. The demo uses subfunctions to place all required functions in a single MATLAB file. To run this example 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.

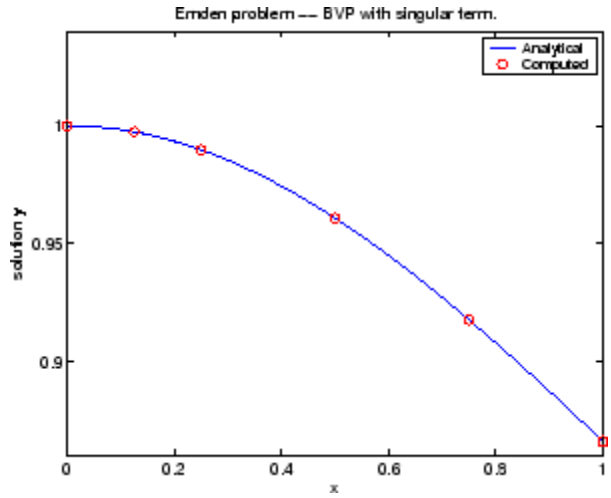
```
sol = bvp4c(@emdenode,@emdenbc,solinit,options);
```

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 $0 \le x \le \lambda$ 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 demo `threebvp` contains the complete code for this example and solves the problem for `λ = 2`, `n = 0.05`, and several values of `κ`. The demo uses nested functions to place all functions required by `bvp4c` in a single MATLAB file and to communicate problem parameters efficiently. To run this example, type `threebvp` at the MATLAB command prompt.

The demo 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: $1 \le x \le \lambda$

```
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 `k`th column `YL(:,k)` represents the solution at the left boundary of the `k`th region. Similarly, `YR(:,k)` represents the solution at the right boundary of the `k`th 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`.

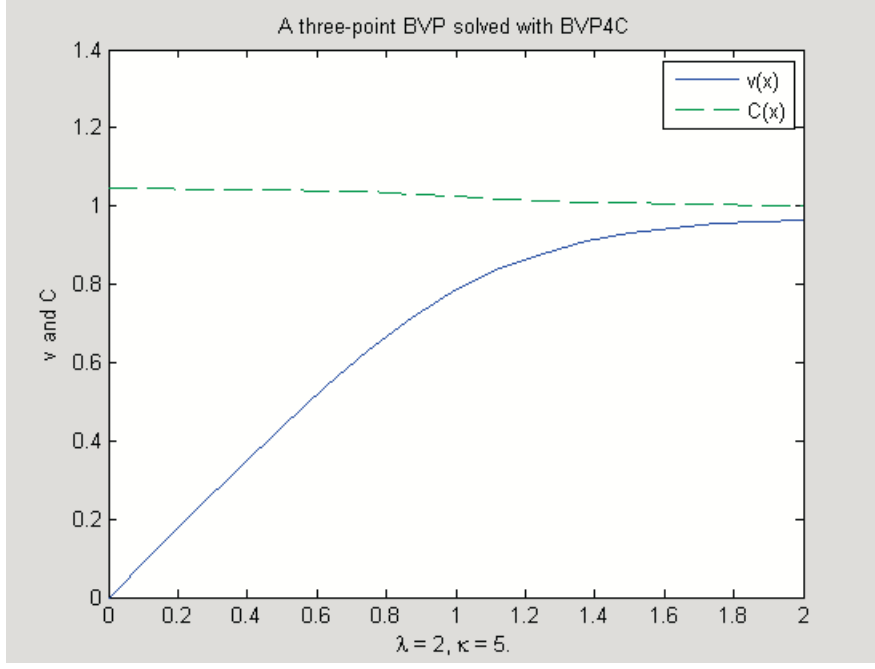
7 **View the results.** Using `deval`, the solution can be evaluated at any point in the interval of integration.

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 = ',num2str(lambda), ...
        ', \kappa = ',num2str(kappa),'.'])
ylabel('v and C')
```



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

Additional examples are provided by "Tutorial on Solving BVPs with BVP4C," available at `http://www.mathworks.com/bvp_tutorial`.

Partial Differential Equations

In this section...
"Function Summary" on page 10-87
"Initial Value Problems" on page 10-88
"PDE Solver" on page 10-89
"Integrator Options" on page 10-92
"Examples" on page 10-93

Function Summary

- "PDE Solver" on page 10-87
- "PDE Helper Function" on page 10-87

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

$$c\left(x,t,u,\frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x,t,u,\frac{\partial u}{\partial x}\right)\right) + s\left(x,t,u,\frac{\partial u}{\partial x}\right) \qquad \textbf{(10-2)}$$

The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$. 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 $a \ge 0$ must also hold.

In Equation 10-2, $f(x,t,u,\partial u/\partial x)$ 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

$$u(x,t_0) = u_0(x) \qquad \textbf{(10-3)}$$

At the boundary $x = a$ or $x = b$, for all t the solution components satisfy a boundary condition of the form

$$p(x,t,u) + q(x,t)f\left(x,t,u,\frac{\partial u}{\partial x}\right) = 0 \qquad \textbf{(10-4)}$$

$q(x, t)$ 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 [9]. 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 10-2 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 10-88.

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 10-2.

`pdefun` — Function that defines the components of the PDE. It computes the terms c, f, and s in Equation 10-2, 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 10-92 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. "Integrator Options" on page 10-9 tells you how to create the structure and describes the properties.

PDE Properties

Properties Category	Property Name
Error control	`RelTol`, `AbsTol`, `NormControl`
Step-size	`InitialStep`, `MaxStep`

Examples

- "Single PDE" on page 10-93
- "System of PDEs" on page 10-98
- "Additional Examples" on page 10-103

Single PDE

- "Solving the Equation" on page 10-93
- "Evaluating the Solution" on page 10-98

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 $0 \le x \le 1$ for times t ≥ 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 demo `pdex1` contains the complete code for this example. The demo uses subfunctions to place all functions it requires in a single MATLAB file. To run the demo type `pdex1` at the command line. See "PDE Solver Syntax" on page 10-89 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 10-2 and expected by `pdepe`. See "Initial Value Problems" on page 10-88 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 10-2) 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 10-4, are

$$u(0,t) + 0 \cdot \frac{\partial u}{\partial x}(0,t) = 0 \qquad \text{at } x = 0$$

and

$$\pi e^{-t} + 1 \cdot \frac{\partial u}{\partial x}(1,t) = 0 \qquad \text{at } x = 1$$

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 10-89). 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 kth 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`.

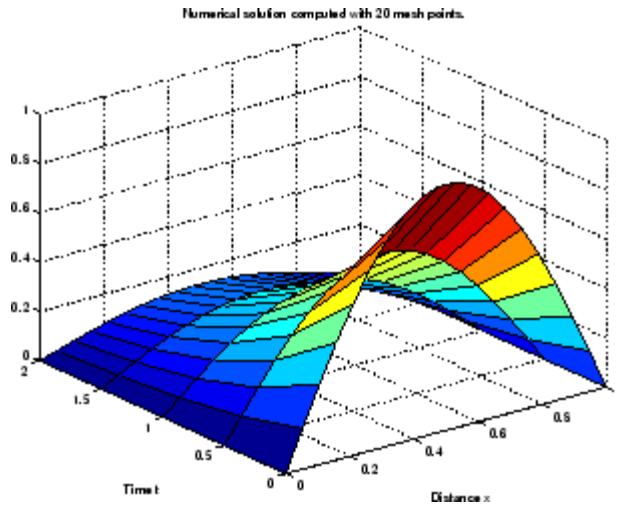
Note See the `function_handle` (@), `func2str`, and `str2func` reference pages, and the `@` section of MATLAB Programming Fundamentals for information about function handles.

---

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

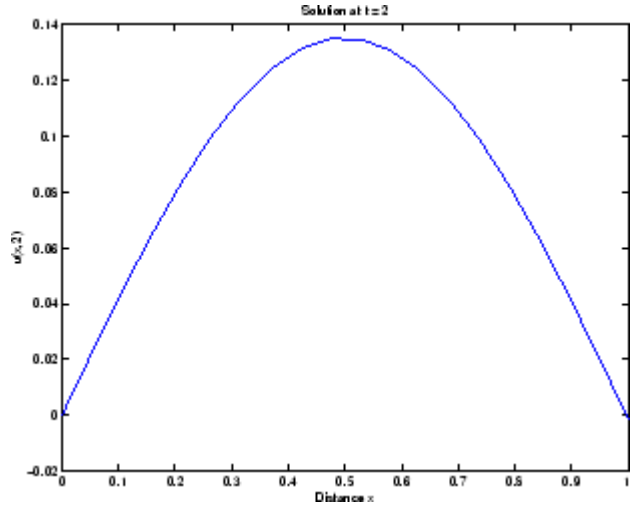
a Extract and display the first solution component. In this example, the solution u has only one component, but for illustrative purposes, the example "extracts" it from the three-dimensional array. The surface plot shows the behavior of the solution.

```
u = sol(:,:,1);

surf(x,t,u)
title('Numerical solution computed with 20 mesh points')
xlabel('Distance x')
ylabel('Time t')
```



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)')
```



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 `k`th column `u(:,k)` (of the solution extracted in step 7) contains the time history of the solution at `x(k)`. The `j`th 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 less than or equal to x less than or equal to 1 $0 \le x \le 1$ 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 demo `pdex4` contains the complete code for this example. The demo uses subfunctions to place all required functions in a single MATLAB file. To run this example 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 10-2.

```
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 10-4) 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];
```

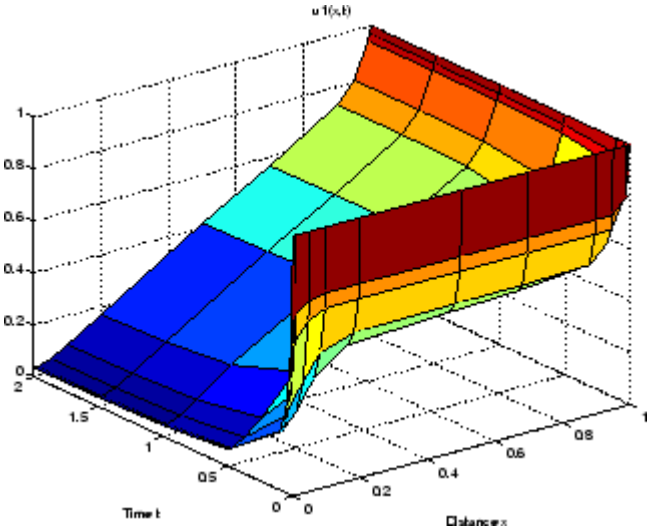
6 Apply the PDE solver. The example calls `pdepe` with `m = 0`, the functions `pdex4pde`, `pdex4ic`, and `pdex4bc`, and the mesh defined by x and t at which `pdepe` is to evaluate the solution. The `pdepe` function returns the numerical solution in a three-dimensional array `sol`, where `sol(i,j,k)` approximates the kth component of the solution, u_k, evaluated at `t(i)` and `x(j)`.

```
m = 0;
sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
```

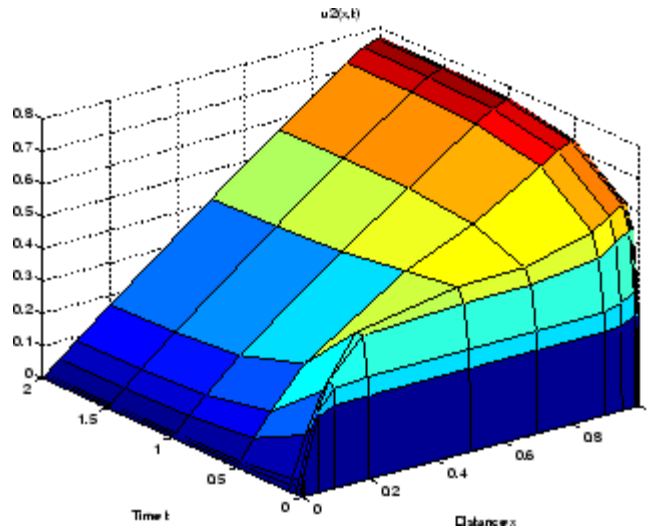
7 View the results. The surface plots show the behavior of the solution components.

```
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')
```



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

Example Name	Description
`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

Selected Bibliography

[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.

[3] Hairer, E., and G. Wanner, *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems*, Springer-Verlag, Berlin, 1991, pp. 5-8.

[4] Hindmarsh, A. C., "LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers," *SIGNUM Newsletter*, Vol. 15, 1980, pp. 10-11.

[5] Hindmarsh, A. C., and G. D. Byrne, "Applications of EPISODE: An Experimental Package for the Integration of Ordinary Differential Equations," *Numerical Methods for Differential Systems*, L. Lapidus and W. E. Schiesser eds., Academic Press, Orlando, FL, 1976, pp 147-166.

[6] Ottesen, J. T., "Modelling of the Baroflex-Feedback Mechanism with Time-Delay," *J. Math. Biol.*, Vol. 36, 1997.

[7] Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall Mathematics, 1994.

[8] Shampine, L. F., and M. K. Gordon, *Computer Solution of Ordinary Differential Equations*, W.H. Freeman & Co., 1975.

[9] 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.

[10] W.H. Enright and H. Hayashi, "The Evaluation of Numerical Software for Delay Differential Equations," R. Boisvert (Ed.), *The Quality of Numerical Software: Assessment and Enhancement*, Chapman & Hall, London, 1997, pp. 179-192.

Integration

In this section...
"Quadrature Functions" on page 10-106
"Example: Arc Length" on page 10-107
"Example: Double Integration" on page 10-107

Quadrature Functions

The area beneath a section of a function $F(x)$ can be determined by numerically integrating $F(x)$, a process referred to as *quadrature*. The MATLAB quadrature functions are:

`quad`	Use adaptive Simpson quadrature
`quadl`	Use adaptive Lobatto quadrature
`quadgk`	Use adaptive Gauss-Kronrod quadrature
`quadv`	Vectorized quadrature
`quad2d`	Numerically evaluate double integral over planar region
`dblquad`	Numerically evaluate double integral
`triplequad`	Numerically evaluate triple integral

To integrate the function defined by `humps.m` from `0` to `1`, use

```
q = quad(@humps,0,1)

q =
    29.8583
```

Both `quad` and `quadl` operate recursively. If either method detects a possible singularity, it prints a warning.

You can include a fourth argument for `quad` or `quadl` that specifies an absolute error tolerance for the integration. If a nonzero fifth argument is passed to `quad` or `quadl`, the function evaluations are traced.

Example: Arc Length

You can use `quad` or `quadl` to compute the length of a curve. Consider the curve parameterized by the equations

$$x(t) = \sin(2t),\ \ y(t) = \cos(t),\ \ z(t) = t,$$

where $t \in [0,3\pi]$.

A three-dimensional plot of this curve is

```
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

$$\int_0^{3\pi} \sqrt{4\cos(2t)^2 + \sin(t)^2 + 1}\,dt.$$

The function `hcurve` computes the integrand

```
function f = hcurve(t)
f = sqrt(4*cos(2*t).^2 + sin(t).^2 + 1);
```

Integrate this function with a call to `quad`

```
len = quad(@hcurve,0,3*pi)

len =
   1.7222e+01
```

The length of this curve is about 17.2.

Example: Double Integration

Consider the numerical solution of

$$\int_{ymin}^{ymax}\int_{xmin}^{xmax} f(x,y)dxdy.$$

For this example $f(x,y) = y\sin(x) + x\cos(y)$. The first step is to build the function to be evaluated. The function must be capable of returning a vector output when given a vector input. You must also consider which variable is in the inner integral, and which goes in the outer integral. In this example, the inner variable is x and the outer variable is y (the order in the integral is $dxdy$). In this case, the `integrand` function is

```
function out = integrnd(x,y)
out = y*sin(x) + x*cos(y);
```

To perform the integration, two functions are available in the `funfun` directory. The first, `dblquad`, is called directly from the command line. This M-file evaluates the outer loop using `quad`. At each iteration, `quad` calls the second helper function that evaluates the inner loop.

To evaluate the double integral, use

```
result = dblquad(@integrnd,xmin,xmax,ymin,ymax);
```

The first argument is a string with the name of the `integrand` function. The second to fifth arguments are

`xmin`	Lower limit of inner integral
`xmax`	Upper limit of the inner integral
`ymin`	Lower limit of outer integral
`ymax`	Upper limit of the outer integral

Here is a numeric example that illustrates the use of `dblquad`.

```
xmin = pi;
xmax = 2*pi;
ymin = 0;
ymax = pi;
result = dblquad(@integrnd,xmin,xmax,ymin,ymax)
```

The result is -9.8698.

By default, `dblquad` calls `quad`. To integrate the previous example using `quadl` (with the default values for the tolerance argument), use

```
result = dblquad(@integrnd,xmin,xmax,ymin,ymax,[],@quadl);
```

Alternatively, you can pass any user-defined quadrature function name to `dblquad` as long as the quadrature function has the same calling and return arguments as `quad`.

11

Fourier Transforms

- "Discrete Fourier Transform (DFT)" on page 11-2
- "Fast Fourier Transform (FFT)" on page 11-8
- "Function Summary" on page 11-28

Discrete Fourier Transform (DFT)

In this section...

"Introduction" on page 11-2

"Visualizing the DFT" on page 11-3

Introduction

Spectral analysis is the process of identifying component frequencies in data. For discrete data, the computational basis of spectral analysis is the *discrete Fourier transform (DFT)*. The DFT transforms time- or space-based data into frequency-based data.

The DFT of a vector x of length n is another vector y of length n:

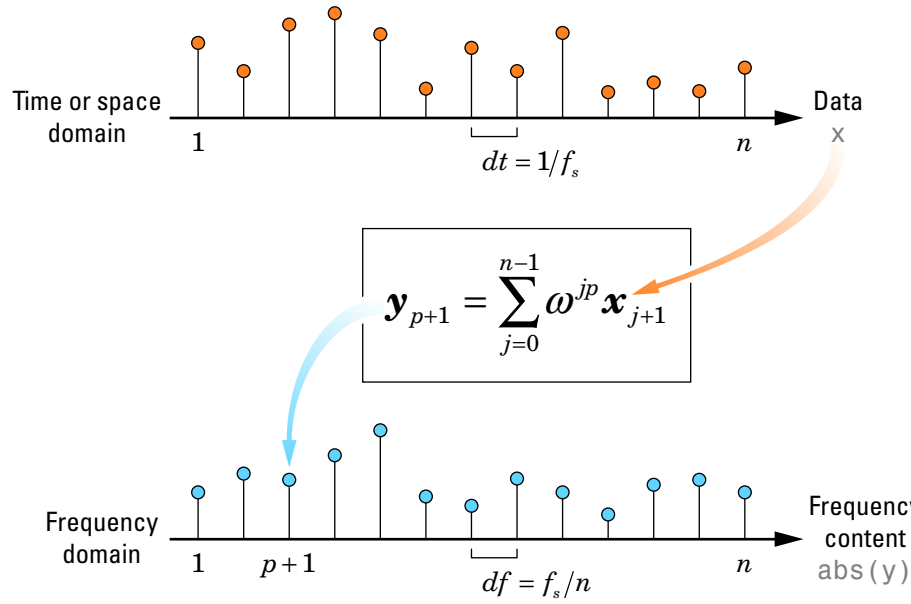
$$y_{p+1} = \sum_{j=0}^{n-1} \omega^{jp} x_{j+1}$$

where ω is a complex nth root of unity:

$$\omega = e^{-2\pi i / n}$$

This notation uses i for the imaginary unit, and p and j for indices that run from 0 to $n-1$. The indices $p+1$ and $j+1$ run from 1 to n, corresponding to ranges associated with MATLAB vectors.

Data in the vector x are assumed to be separated by a constant interval in time or space, $dt = 1/f_s$ or $ds = 1/f_s$, where f_s is the *sampling frequency*. The DFT y is complex-valued. The absolute value of y at index $p+1$ measures the amount of the frequency $f = p(f_s / n)$ present in the data.



The first element of *y*, corresponding to zero frequency, is the sum of the data in *x*. This *DC component* is often removed from *y* so that it does not obscure the positive frequency content of the data.

Visualizing the DFT

The documentation example function `fftgui` opens a graphical user interface that allows you to explore the real and imaginary parts of a data vector and its DFT. (Here `fft` refers to efficient computation of the DFT by the MATLAB `fft` function, as discussed in "Fast Fourier Transform (FFT)" on page 11-8.)

Note Documentation example files for MATLAB mathematics are located in the `\help\techdoc\math\examples` subfolder of your MATLAB root folder (`matlabroot`). This suibfolder is not on the MATLAB path at installation. To use the MATLAB files in this subfolder, either add the subfolder to the MATLAB path (`addpath`) or make the subfolder your current working folder (`cd`).

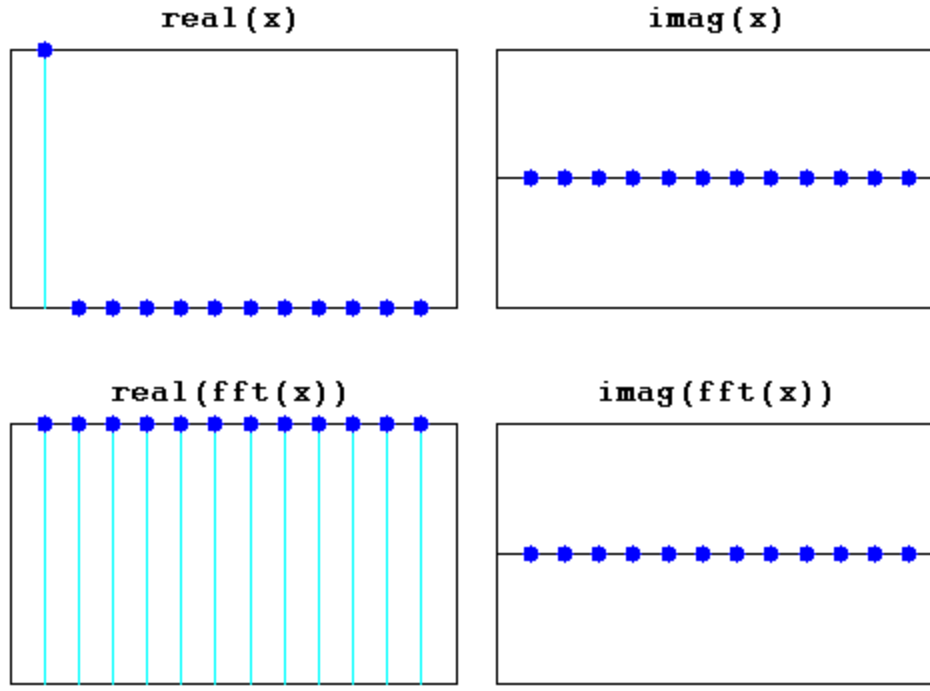
If `x` is a vector,

```
fftgui(x)
```

plots the real and imaginary parts of both `x` and its DFT (`fft(x)`). Use your mouse to drag points in any of the plots and the points in the other plots respond to the changes.

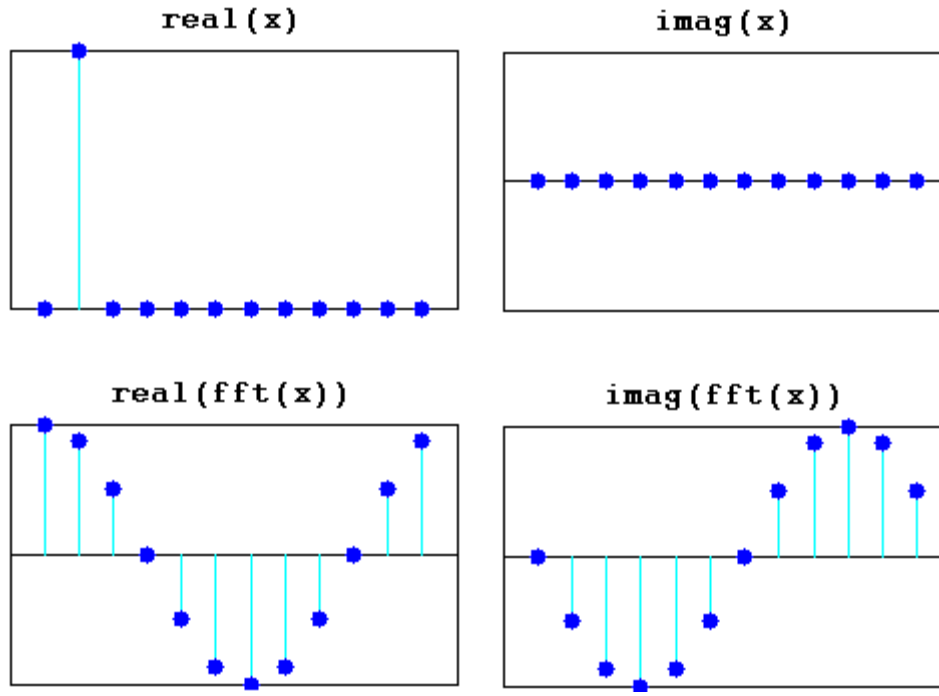
For example, use `fftgui` to display the DFT of a unit impulse at `x(1)`:

```
x = [1 zeros(1,11)];
fftgui(x)
```



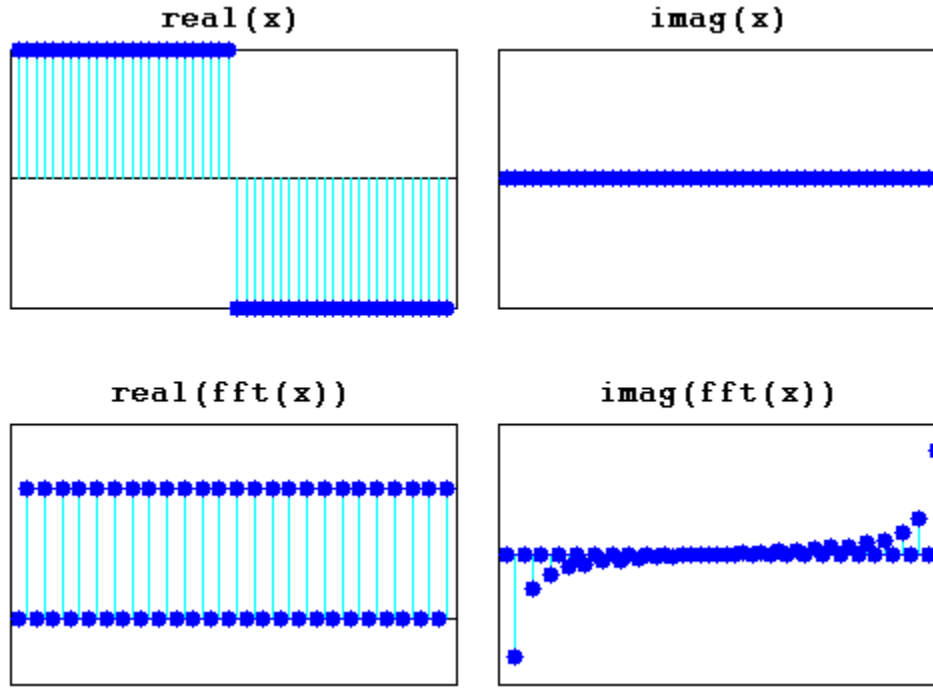
The transform is quite different if the unit impulse is at `x(2)`:

```
x = [0 1 zeros(1,10)];
fftgui(x)
```

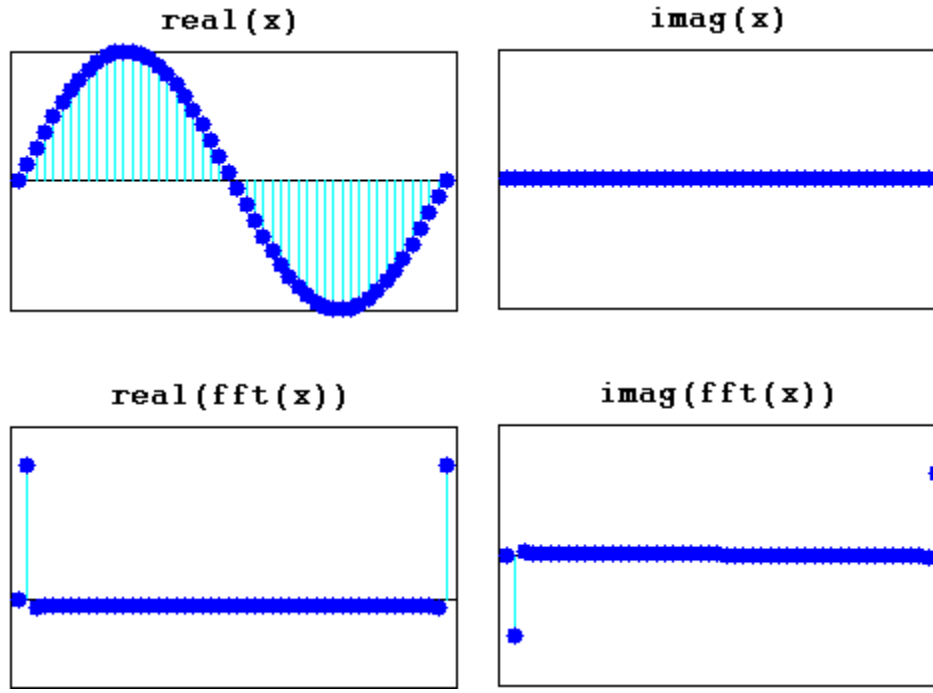


The following commands display DFTs of square and sine waves, respectively:

```
x = [ones(1,25),-ones(1,25)];
fftgui(x)
```



```
t = linspace(0,1,50);
x = sin(2*pi*t);
fftgui(x)
```



The midpoint of the DFT (or the point just to the right of the midpoint if the length is even), corresponding to half the sampling frequency of the data, is the *Nyquist point*. For real `x`, the real part of the DFT is symmetric about the Nyquist point, and the imaginary part is antisymmetric about the Nyquist point.

Fast Fourier Transform (FFT)

In this section...

"Introduction" on page 11-8

"The FFT in One Dimension" on page 11-9

"The FFT in Multiple Dimensions" on page 11-23

Introduction

DFTs with a million points are common in many applications. Modern signal and image processing applications would be impossible without an efficient method for computing the DFT.

Direct application of the definition of the DFT (see "Discrete Fourier Transform (DFT)" on page 11-2) to a data vector of length n requires n multiplications and n additions—a total of $2n^2$ floating-point operations. This does not include the generation of the powers of the complex nth root of unity ω. To compute a million-point DFT, a computer capable of doing one multiplication and addition every microsecond requires a million seconds, or about 11.5 days.

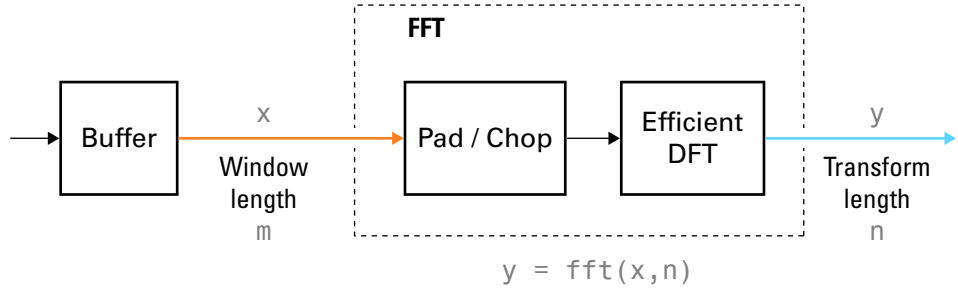
Fast Fourier Transform (FFT) algorithms have computational complexity $O(n \log n)$ instead of $O(n^2)$. If n is a power of 2, a one-dimensional FFT of length n requires less than $3n \log_2 n$ floating-point operations (times a proportionality constant). For $n = 220$, that is a factor of almost 35,000 faster than $2n^2$.

The MATLAB functions `fft`, `fft2`, and `fftn` (and their inverses `ifft`, `ifft2`, and `ifftn`, respectively) all use fast Fourier transform algorithms to compute the DFT.

Note MATLAB FFT algorithms are based on FFTW, "The Fastest Fourier Transform in the West" (`http://www.fftw.org`). See `fft` and `fftw` for details.

When using FFT algorithms, a distinction is made between the *window length* and the *transform length*. The window length is the length of the input data vector. It is determined by, for example, the size of an external buffer. The

transform length is the length of the output, the computed DFT. An FFT algorithm pads or chops the input to achieve the desired transform length. The following figure illustrates the two lengths.



The execution time of an FFT algorithm depends on the transform length. It is fastest when the transform length is a power of two, and almost as fast when the transform length has only small prime factors. It is typically slower for transform lengths that are prime or have large prime factors. Time differences, however, are reduced to insignificance by modern FFT algorithms such as those used in MATLAB. Adjusting the transform length for efficiency is usually unnecessary in practice.

The FFT in One Dimension

- "Introduction" on page 11-9
- "Example: Basic Spectral Analysis" on page 11-10
- "Example: Spectral Analysis of a Whale Call" on page 11-14
- "Example: Data Interpolation" on page 11-18

Introduction

The MATLAB `fft` function returns the DFT `y` of an input vector `x` using a fast Fourier transform algorithm:

```
y = fft(x);
```

In this call to `fft`, the window length `m = length(x)` and the transform length `n = length(y)` are the same.

The transform length is specified by an optional second argument:

```
y = fft(x,n);
```

In this call to `fft`, the transform length is `n`. If the length of `x` is less than `n`, `x` is padded with trailing zeros to increase its length to `n` before computing the DFT. If the length of `x` is greater than `n`, only the first `n` elements of `x` are used to compute the transform.

Example: Basic Spectral Analysis

The FFT allows you to efficiently estimate component frequencies in data from a discrete set of values sampled at a fixed rate. Relevant quantities in a spectral analysis are listed in the following table. For space-based data, replace references to time with references to space.

Quantity	Description
`x`	Sampled data
`m = length(x)`	Window length (number of samples)
`fs`	Samples/unit time
`dt = 1/fs`	Time increment per sample
`t = (0:m-1)/fs`	Time range for data
`y = fft(x,n)`	Discrete Fourier transform (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

For example, consider the following data `x` with two component frequencies of differing amplitude and phase buried in noise:

```
fs = 100;                           % Sample frequency (Hz)
t = 0:1/fs:10-1/fs;                 % 10 sec sample
x = (1.3)*sin(2*pi*15*t) ...        % 15 Hz component
  + (1.7)*sin(2*pi*40*(t-2)) ...    % 40 Hz component
  + (2.5)*randn(size(t));           % Gaussian noise;
```

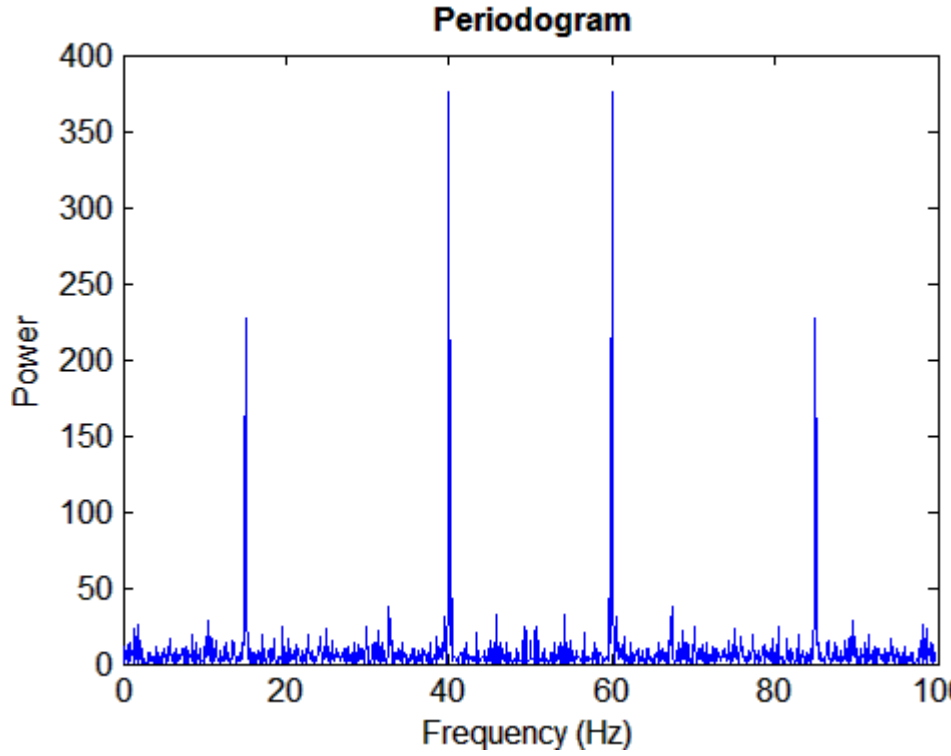
Use `fft` to compute the DFT `y` and its power:

```
m = length(x);          % Window length
n = pow2(nextpow2(m));  % Transform length
y = fft(x,n);           % DFT
f = (0:n-1)*(fs/n);     % Frequency range
power = y.*conj(y)/n;   % Power of the DFT
```

`nextpow2` finds the exponent of the next power of two greater than or equal to the window length (`ceil(log2(m))`), and `pow2` computes the power. Using a power of two for the transform length optimizes the FFT algorithm, though in practice there is usually little difference in execution time from using `n = m`.

To visualize the DFT, plots of `abs(y)`, `abs(y).^2`, and `log(abs(y))` are all common. A plot of power versus frequency is called a *periodogram*:

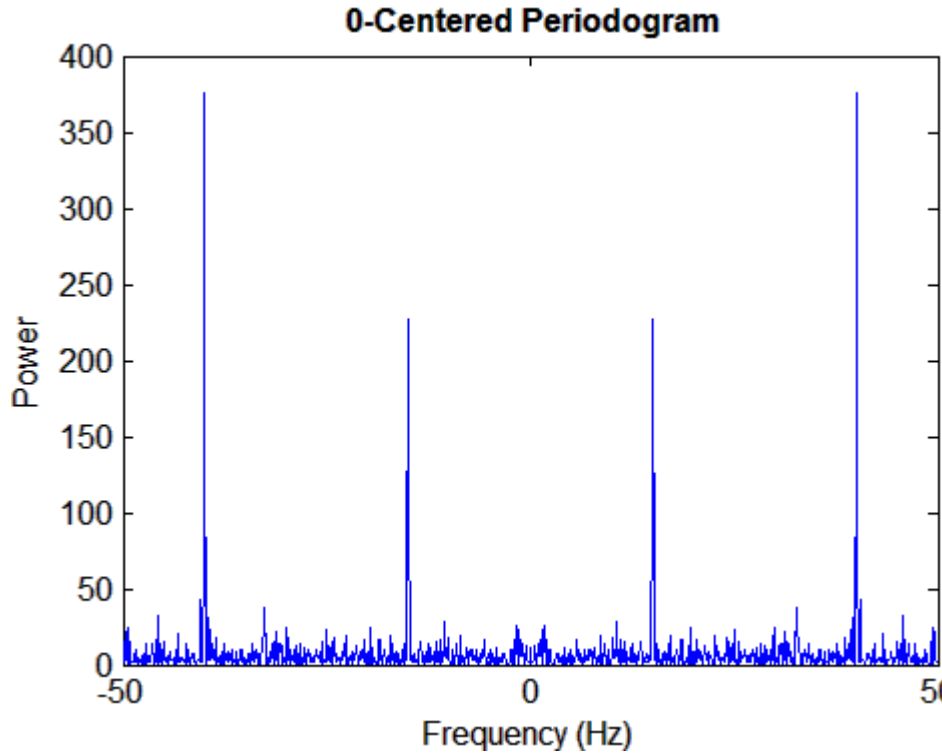
```
plot(f,power)
xlabel('Frequency (Hz)')
ylabel('Power')
title('{\bf Periodogram}')
```



The first half of the frequency range (from 0 to the Nyquist frequency `fs/2`) is sufficient to identify the component frequencies in the data, since the second half is just a reflection of the first half.

In many applications it is traditional to center the periodogram at 0. The `fftshift` function rearranges the output from `fft` with a circular shift to produce a 0-centered periodogram:

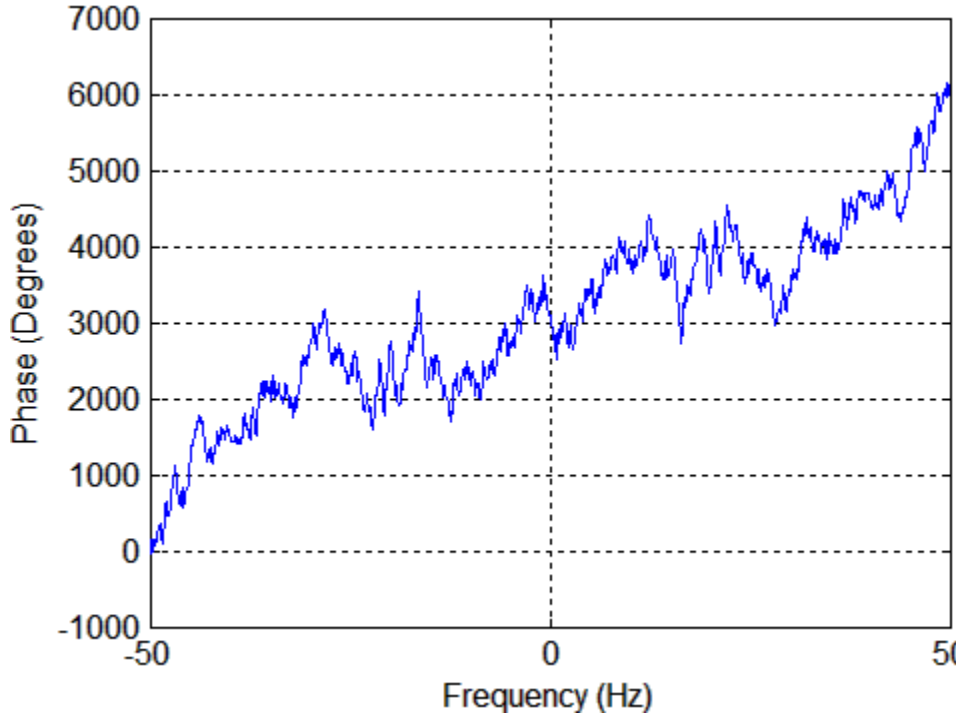
```
y0 = fftshift(y);           % Rearrange y values
f0 = (-n/2:n/2-1)*(fs/n);   % 0-centered frequency range
power0 = y0.*conj(y0)/n;    % 0-centered power

plot(f0,power0)
xlabel('Frequency (Hz)')
ylabel('Power')
title('{\bf 0-Centered Periodogram}')
```



The rearrangement makes use of the periodicity in the definition of the DFT (see “Discrete Fourier Transform (DFT)” on page 11-2).

Use the MATLAB `angle` and `unwrap` functions to create a phase plot of the DFT:

```
phase = unwrap(angle(y0));

plot(f0,phase*180/pi)
xlabel('Frequency (Hz)')
ylabel('Phase (Degrees)')
grid on
```



Component frequencies are mostly hidden by the randomness in phase at adjacent values. The upward trend in the plot is due to the `unwrap` function, which in this case adds 2π to the phase more often than it subtracts it.

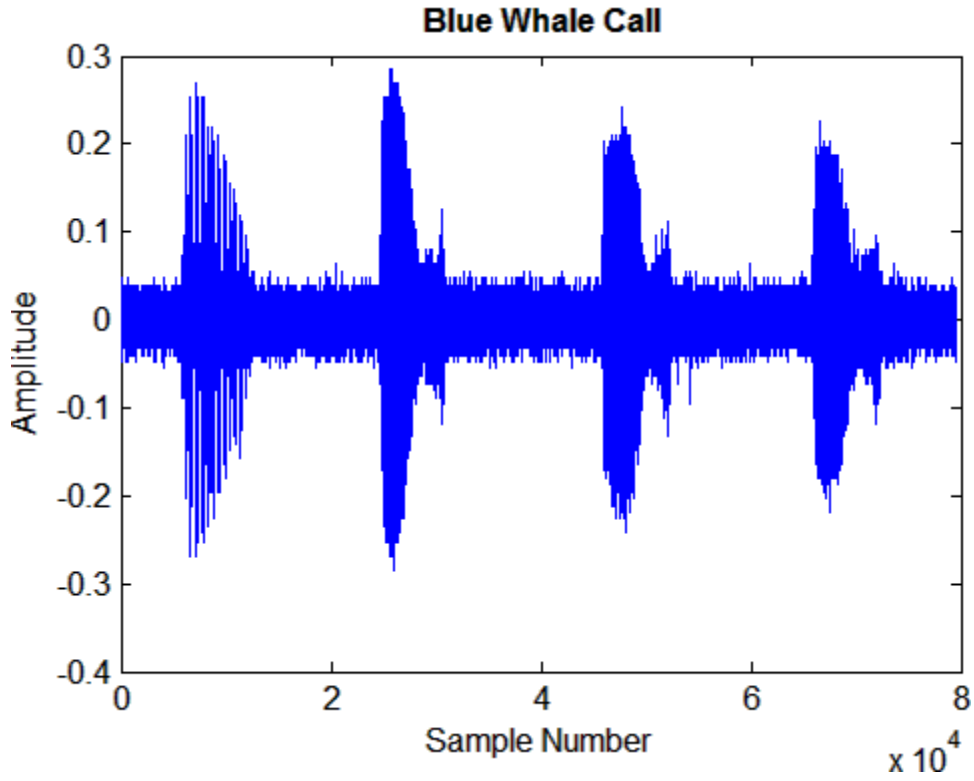
Example: Spectral Analysis of a Whale Call

The documentation example 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.

Note Documentation example files for MATLAB mathematics are located in the `\help\techdoc\math\examples` subfolder of your MATLAB root folder (`matlabroot`). This subfolder is not on the MATLAB path at installation. To use the MATLAB files in this subfolder, either add the subfolder to the MATLAB path (`addpath`) or make the subfolder your current working folder (`cd`).

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. The following reads, plots, and plays the data:

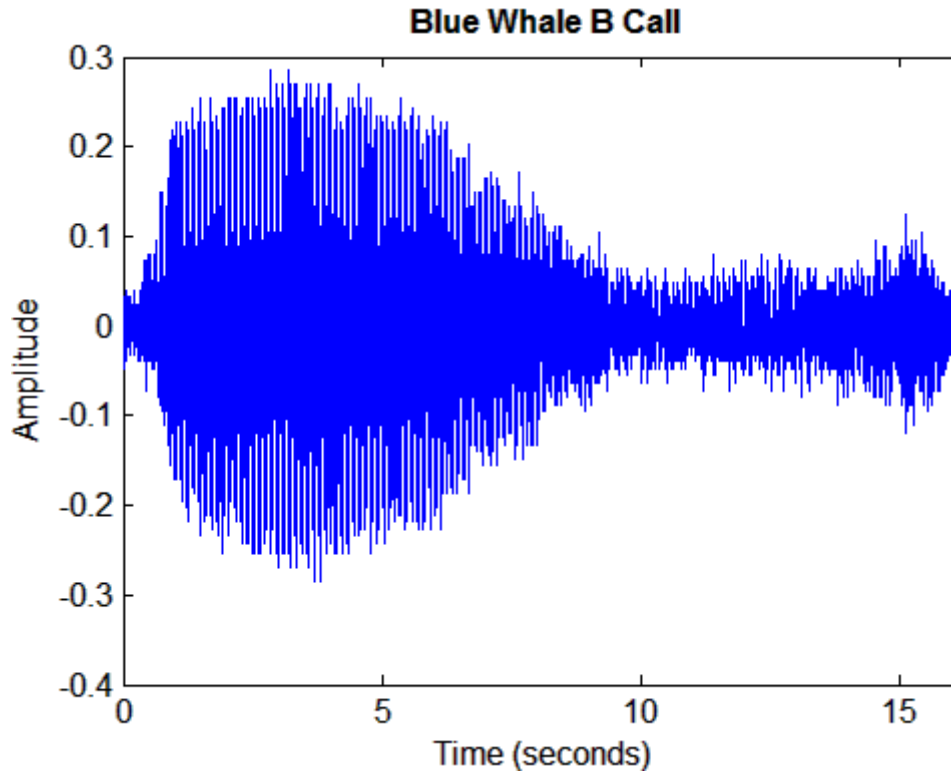
```
[x,fs] = auread('bluewhale.au');

plot(x)
xlabel('Sample Number')
ylabel('Amplitude')
title('{\bf Blue Whale Call}')

sound(x,fs)
```



An A “trill” is followed by a series of B “moans.”

The B call is simpler and easier to analyze. Use the previous plot to determine approximate indices for the beginning and end of the first B call. Correct the time base for the factor of 10 speed-up in the data:

```
bCall = x(2.45e4:3.10e4);
tb = 10*(0:1/fs:(length(bCall)-1)/fs); % Time base

plot(tb,bCall)
xlim([0 tb(end)])
xlabel('Time (seconds)')
ylabel('Amplitude')
title('{\bf Blue Whale B Call}')
```

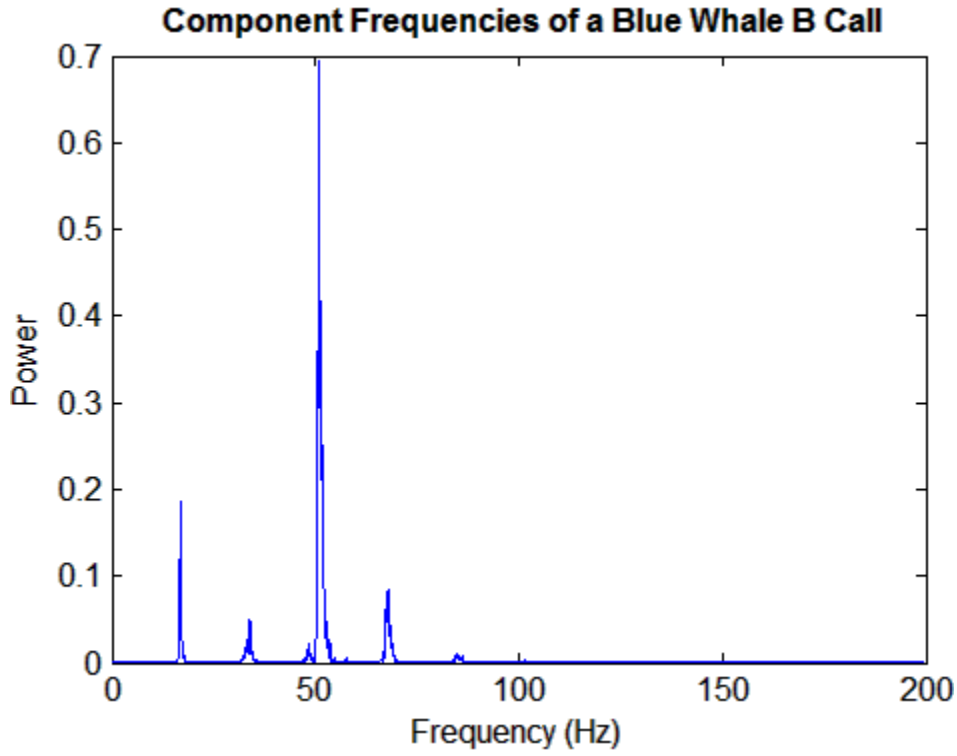


Use `fft` to compute the DFT of the signal. Correct the frequency range for the factor of 10 speed-up in the data:

```
m = length(bCall);      % Window length
n = pow2(nextpow2(m));  % Transform length
y = fft(bCall);         % DFT of signal
f = (0:n-1)*(fs/n)/10;  % Frequency range
p = y.*conj(y)/n;       % Power of the DFT
```

Plot the first half of the periodogram, up to the Nyquist frequency:

```
plot(f(1:floor(n/2)),p(1:floor(n/2)))
xlabel('Frequency (Hz)')
ylabel('Power')
title('{\bf Component Frequencies of a Blue Whale B Call}')
```



The B call is composed of a fundamental frequency around 17 Hz and a sequence of harmonics, with the second harmonic emphasized.

Example: Data Interpolation

This example demonstrates the FFT in a context other than spectral analysis—estimating coefficients of a trigonometric polynomial that interpolates a set of regularly-spaced data. This approach to data interpolation is described in [1].

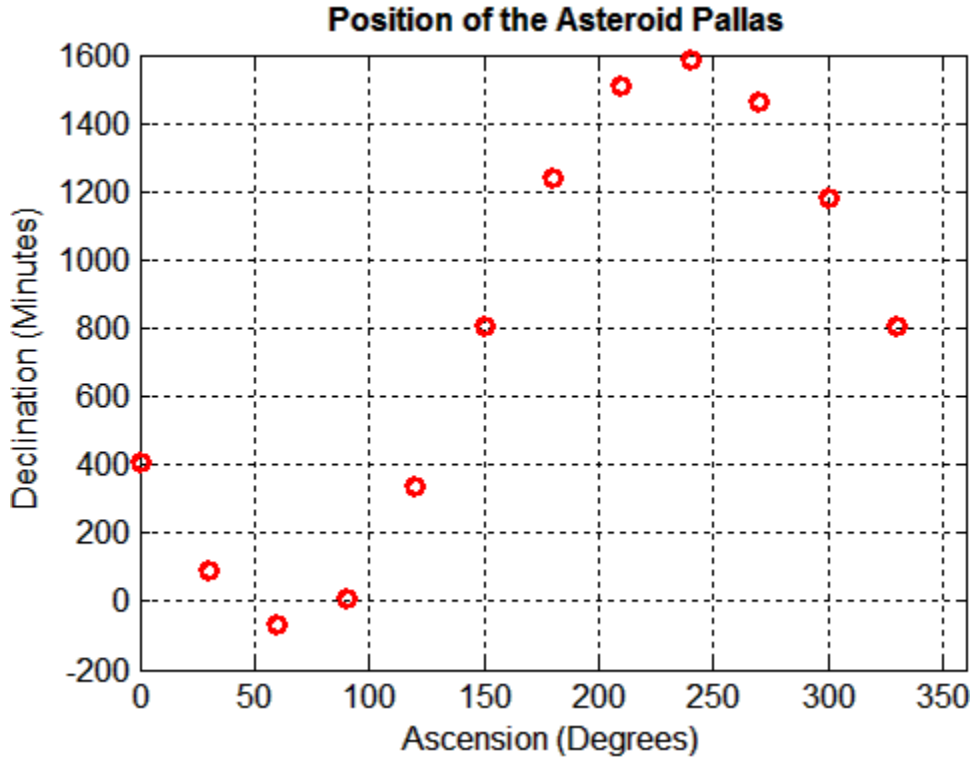
Several people discovered fast DFT algorithms independently, and many people have contributed to their development. A 1965 paper by John Tukey and John Cooley [2] is generally credited as the starting point for modern usage of the FFT. However, a paper by Gauss published posthumously in 1866

[3] (and dated to 1805) contains indisputable use of the splitting technique that forms the basis of modern FFT algorithms.

Gauss was interested in the problem of computing accurate asteroid orbits from observations of their positions. His paper contains 12 data points on the position of the asteroid Pallas, through which he wished to interpolate a trigonometric polynomial with 12 coefficients. Instead of solving the resulting 12-by-12 system of linear equations by hand, Gauss looked for a shortcut. He discovered how to separate the equations into three subproblems that were much easier to solve, and then how to recombine the solutions to obtain the desired result. The solution is equivalent to estimating the DFT of the data with an FFT algorithm.

Here is the data that appears in Gauss' paper:

```
asc = 0:30:330;
dec = [408 89 -66 10 338 807 1238 1511 1583 1462 1183 804];

plot(asc,dec,'ro','Linewidth',2)
xlim([0 360])
xlabel('Ascension (Degrees)')
ylabel('Declination (Minutes)')
title('{\bf Position of the Asteroid Pallas}')
grid on
```



Gauss wished to interpolate a trigonometric polynomial of the form:

$$
\begin{aligned}
y = a_0 &+ a_1 \cos(2\pi(x/360)) + b_1 \sin(2\pi(x/360)) \\
&a_2 \cos(2\pi(2x/360)) + b_2 \sin(2\pi(2x/360)) \\
&\cdots \\
&a_5 \cos(2\pi(5x/360)) + b_5 \sin(2\pi(5x/360)) \\
&a_6 \cos(2\pi(6x/360))
\end{aligned}
$$

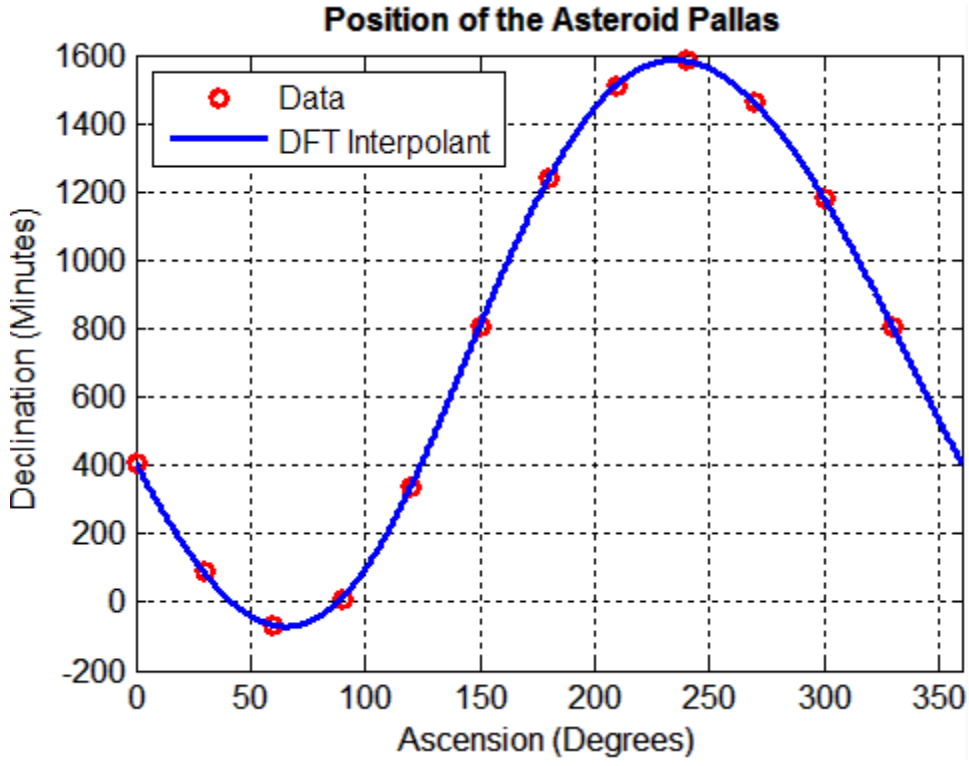
The following uses `fft` to perform an equivalent of Gauss' calculation:

```
d = fft(dec);
m = length(dec);
M = floor((m+1)/2);
```

```
a0 = d(1)/m;
an = 2*real(d(2:M))/m;
a6 = d(M+1)/m;
bn = -2*imag(d(2:M))/m;
```

Plot the interpolant with the data:

```
hold on

x = 0:0.01:360;
n = 1:length(an);
y = a0 + an*cos(2*pi*n'*x/360) ...
       + bn*sin(2*pi*n'*x/360) ...
       + a6*cos(2*pi*6*x/360);

plot(x,y,'Linewidth',2)
legend('Data','DFT Interpolant','Location','NW')
```



References.

[1] Briggs, W. and V.E. Henson. *The DFT: An Owner's Manual for the Discrete Fourier Transform*. Philadelphia: SIAM, 1995.

[2] Cooley, J.W. and J.W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation*. Vol. 19. 1965, pp. 297–301.

[3] Gauss, C. F. "Theoria interpolationis methodo nova tractata." *Carl Friedrich Gauss Werke*. Band 3. Göttingen: Königlichen Gesellschaft der Wissenschaften, 1866.

[4] 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.

[5] Goldstine, H. H. *A History of Numerical Analysis from the 16th through the 19th Century*. Berlin: Springer-Verlag, 1977.

The FFT in Multiple Dimensions

- “Introduction” on page 11-23
- “Example: Diffraction Patterns” on page 11-24

Introduction

This section discusses generalizations of the DFT in one dimension (see “Discrete Fourier Transform (DFT)” on page 11-2).

In two dimensions, the DFT of an m-by-n array X is another m-by-n array Y:

$$Y_{p+1,q+1} = \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} \omega_m^{jp} \omega_n^{kq} X_{j+1,k+1}$$

where ω_m and ω_n are complex roots of unity:

$$\omega_m = e^{-2\pi i/m}$$
$$\omega_n = e^{-2\pi i/n}$$

This notation uses i for the imaginary unit, p and j for indices that run from 0 to m–1, and q and k for indices that run from 0 to n–1. The indices p+1 and j+1 run from 1 to m and the indices q+1 and k+1 run from 1 to n, corresponding to ranges associated with MATLAB arrays.

The MATLAB function `fft2` computes two-dimensional DFTs using a fast Fourier transform algorithm. `Y = fft2(X)` is equivalent to `Y = fft(fft(X).').'`, that is, to computing the one-dimensional DFT of

each column `X` followed by the one-dimensional DFT of each row of the result. The inverse transform of the two-dimensional DFT is computed by `ifft2`.

The MATLAB function `fftn` generalizes `fft2` to *N*-dimensional arrays. `Y = fftn(X)` is equivalent to:

```
Y = X;
for p = 1:length(size(X))
    Y = fft(Y,[],p);
end
```

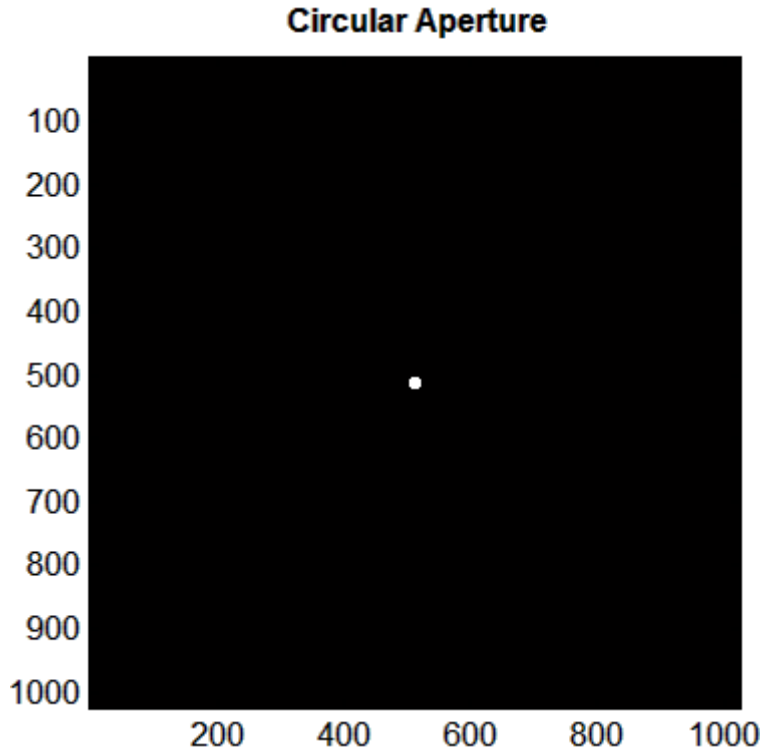
That is, to computing in place the one-dimensional DFT along each dimension of `X`. The inverse transform of the *N*-dimensional DFT is computed by `ifftn`.

Example: Diffraction Patterns

The theory of optics predicts that the diffraction pattern produced by a plane wave incident on an optical mask with a small aperture is described, at a distance, by the Fourier transform of the mask. See, for example, [1].

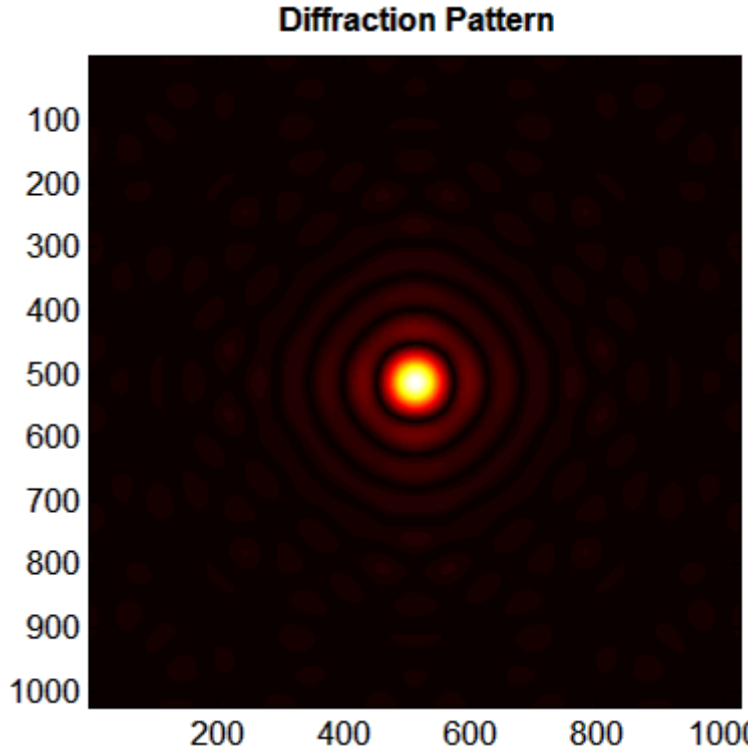
The following creates a logical array describing an optical mask `M` with a circular aperture `A` of radius `R`:

```
n = 2^10;
M = zeros(n);

I = 1:n;
x = I-n/2;
y = n/2-I;
[X,Y] = meshgrid(x,y);
R = 10;
A = (X.^2 + Y.^2 <= R^2);
M(A) = 1;

imagesc(M)
colormap([0 0 0; 1 1 1])
axis image
title('{\bf Circular Aperture}')
```



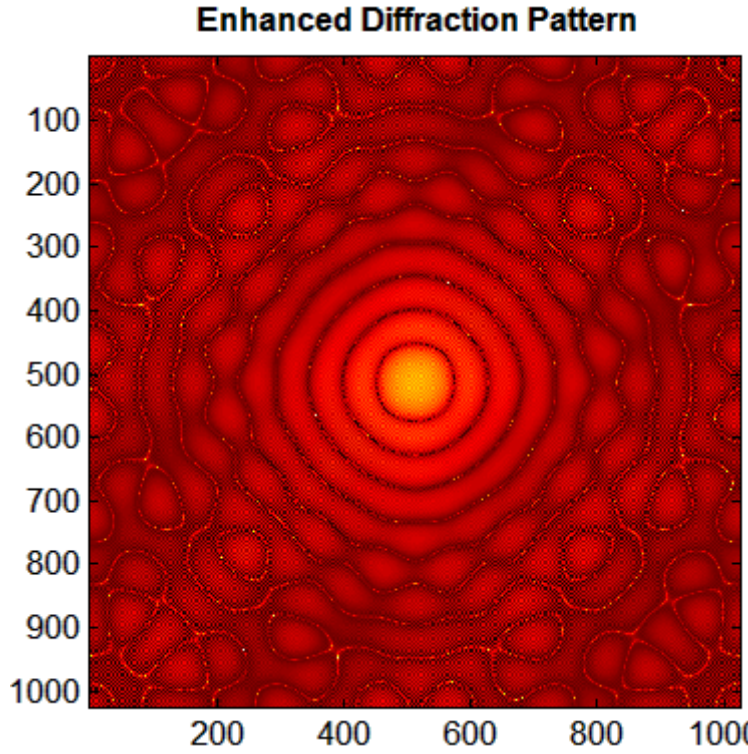
Use `fft2` to compute the two-dimensional DFT of the mask and `fftshift` to rearrange the output so that the zero-frequency component is at the center:

```
D1 = fft2(M);
D2 = fftshift(D1);

imagesc(abs(D2))
axis image
colormap(hot)
title('{\bf Diffraction Pattern}')
```



The logarithm helps to bring out details of the DFT in regions where the amplitude is small:

```
D3 = log2(D2);

imagesc(abs(D3))
axis image
colormap(hot)
title('{\bf Enhanced Diffraction Pattern}')
```



Very small amplitudes are affected by numerical round-off. The lack of radial symmetry is an artifact of the rectangular arrangement of data.

Reference.

[1] Fowles, G. R. *Introduction to Modern Optics*. New York: Dover, 1989.

Function Summary

MATLAB functions related to Fourier transforms include:

Function	Description
`fft`	One-dimensional fast Fourier transform
`ifft`	One-dimensional inverse fast Fourier transform
`fft2`	Two-dimensional fast Fourier transform
`ifft2`	Two-dimensional inverse fast Fourier transform
`fftn`	*N*-dimensional fast Fourier transform
`ifftn`	*N*-dimensional inverse fast Fourier transform
`fftshift`	Rearrange DFT data to center zero-frequency component
`fftw`	Interface to FFTW run-time algorithm
`abs`	Amplitude of the DFT
`angle`	Phase of the DFT
`unwrap`	Correct phase angles with jumps greater than or equal to `pi`
`nextpow2`	Next power of two greater than or equal to a given length
`pow2`	Compute powers of two

The MATLAB software also includes demos that use these functions in combination for Fourier analysis:

- FFT for Spectral Analysis — Reviews basic spectral analysis with the FFT
- Using FFT in MATLAB — Example of time series analysis with the FFT that looks for periodicity in historical data on sunspot activity

Index

Symbols and Numerics

: symbol
  generating a numeric sequence 1-9

A

additional parameters
  BVP example 10-70 10-73
amp1dae demo 10-38
anonymous functions
  representing mathematical functions 9-4
arc length 10-107
arrays
  concatenating diagonally 1-41
  deleting rows and columns 1-29
  diagonal 1-40
  dimensions
    inverse permutation 1-63
  empty 1-43
  expanding 1-25
  flipping 1-32
  functions
    changing indexing style 1-72
    creating a matrix 1-70
    determining data type 1-71
    finding matrix structure or shape 1-71
    modifying matrix shape 1-70
    multidimensional arrays 1-73
    sorting and shifting 1-72
  functions for diagonals 1-72
  getting dimensions of 1-22
  linear indexing 1-13
  multidimensional 1-50
  reshaping 1-30
  rotating 1-32
  shifting 1-35
  sorting column data 1-37
  sorting row data 1-37
  sorting row vectors 1-38
  storage 1-13
  transposing 1-31

B

ballode demo 10-25
bandwidth of sparse matrix, reducing 4-29
batonode demo 10-38
bicubic interpolation 7-4
bilinear interpolation 7-4
boundary conditions
  BVP 10-60
  BVP example 10-66
  PDE 10-88
  PDE example 10-95
boundary value problems. *See* BVP
Brusselator system (ODE example) 10-22
brussode demo 10-22
burgersode demo 10-39
BVP 10-59
  defined 10-60
  initial guess 10-69
  multipoint terms 10-81
  rewriting as first-order system 10-66
  singular terms 10-77
BVP solver 10-61
  basic syntax 10-62
  evaluate solution at specific points 10-69
  examples
    boundary condition at infinity (shockbvp) 10-73
    rapid solution changes (shockbvp) 10-70
  performance 10-64
  unknown parameters 10-69

C

cat 1-54
  sparse operands 4-26
cell arrays
  growing 1-26 1-28

multidimensional 1-67
character arrays
expanding 1-29
characteristic polynomial of matrix 5-11
characteristic roots of matrix 5-11
chol
sparse matrices 4-26
Cholesky factorization 2-31
sparse matrices 4-33
colamd
minimum degree ordering 4-30
colmmd
column permutation 4-31
colon operator 1-9
for multidimensional array subscripting 1-57
scalar expansion with 1-53
colperm 4-29
computational functions
applying to multidimensional arrays 1-64
applying to sparse matrices 4-25
concatenation 1-6
functions 1-7
of diagonal matrices 1-41
of matrices 1-6
contents of sparse matrix 4-19
convolution 5-7
creating
multidimensional array 1-52
sparse matrix 4-14
cross 1-64
cubic interpolation
multidimensional 7-9
one-dimensional 7-3
spline 7-2
curve fitting
polynomial 5-9
curves
computing length 10-107
Cuthill-McKee
reverse ordering 4-29

D

DAE 10-2
data gridding
multidimensional 7-9
data organization
multidimensional arrays 1-65
DDE 10-47
rewriting as first-order system 10-52
DDE solver 10-49
discontinuities 10-50
evaluating solution at specific points 10-54
examples
cardiovascular model (ddex2) 10-55
ddex1 10-52
performance 10-51
ddex1 demo 10-52
ddex2 demo 10-55
decomposition
eigenvalue 2-44
Schur 2-46
singular value 2-48
deconvolution 5-7
delay differential equations. *See* DDE
deleting
matrix rows and columns 1-29
deleting array elements 1-29
deletion operator 1-29
density
sparse matrix 4-13
derivatives
polynomial 5-6
determinant of matrix 2-26
DFT 11-2
diag 4-26
diagonal
creating sparse matrix from 4-16
diagonal matrices 1-40
differential equations 10-1 to 10-2
differential-algebraic equations 10-2
dim argument for cat 1-54

dimensions
  permuting 1-62
  removing singleton 1-61
direct methods
  systems of sparse equations 4-37
discontinuities
  DDE solver 10-50
discrete Fourier transform (DFT) 11-2
displaying
  sparse matrices 4-24
dot product 2-9

E

`eig` 1-65
eigenvalues 2-44
  of sparse matrix 4-40
eigenvectors 2-44
electrical circuits
  DAE example 10-38
Emden's equation
  example 10-78
empty matrices 1-43
`end` 1-15
error tolerance
  effects of too large (ODE) 10-45
  machine precision 10-42
event location (ODE)
  advanced example 10-28
  simple example 10-25
expanding
  character arrays 1-29
expanding cell arrays 1-26 1-28
expanding structure arrays 1-26 1-28
`eye`
  derivation of the name 2-11
  sparse matrices 4-26

F

factorization 4-30
  Cholesky 2-31
  Hermitian positive definite 2-32
  incomplete 4-35
  LU 2-33
  partial pivoting 2-33
  positive definite 2-31
  QR 2-34
  sparse matrices 4-30
    Cholesky 4-33
    LU 4-30
    triangular 4-30
fast Fourier transform (FFT) 11-8
`fem1ode` demo 10-19
`fem2ode` demo 10-39
FFT 11-8
  DC component 11-3
  history 11-18
  transform length 11-8
  window length 11-8
`find` function
  sparse matrices 4-21
finite element discretization (ODE
  example) 10-19
first-order differential equations
  representation for BVP solver 10-66
  representation for DDE solver 10-52
flipping matrices 1-32
Fourier transforms
  FFT-based interpolation 7-3
`fsbvp` demo 10-73
`full` 4-26 4-29
function functions 9-1
functional operators 9-1
functions
  applying
    to multidimensional structure
      arrays 1-69
  changing indexing style 1-72

creating a matrix 1-70
creating arrays with 1-54
creating matrices 1-4
determining data type 1-71
finding matrix structure or shape 1-71
for diagonal matrices 1-72
mathematical. *See* mathematical functions
matrix concatenation 1-7
modifying matrix shape 1-70
multidimensional arrays 1-73
optimizing 8-1
sorting and shifting 1-72
sparse matrix 1-48

G

Gaussian elimination 2-33
global minimum 8-30
growing an array 1-25
growing cell array 1-26 1-28
growing structure arrays 1-26 1-28

H

`hb1dae` demo 10-31
`hb1ode` demo 10-39
Hermitian positive definite matrix 2-32
higher-order ODEs
rewriting as first-order ODEs 10-5

I

`iburgersode` demo 10-39
identity matrix 2-11
`ihb1dae` demo 10-39
importing
sparse matrix 4-18
incomplete factorization 4-35
indexing
multidimensional arrays 1-56
indices, how MATLAB calculates 1-59
infeasible optimization problems 8-30
initial conditions
ODE 10-5
ODE example 10-11
PDE 10-88
PDE example 10-94
initial guess (BVP)
example 10-67
initial value problems
defined 10-5
inner product 2-7
integration 10-106
double 10-107
numerical 10-106
triple 10-106
See also differential equations
integration interval
PDE (MATLAB) 10-91
interpolation
comparing methods graphically 7-5
FFT-based 7-3
multidimensional 7-8
one-dimensional 7-2
speed, memory, smoothness 7-3
three-dimensional 7-7
two-dimensional 7-4
using FFT 11-18
inverse of matrix 2-26
inverse permutation of array dimensions 1-63
`ipermute` 1-63
iterative methods
sparse matrices 4-38
sparse systems of equations 4-37

K

Kronecker tensor matrix product 2-12

L

least squares 4-35
length of curve, computing 10-107
linear equations
  minimal norm solution 2-28
  overdetermined systems 2-21
  rectangular systems 2-27
linear interpolation
  multidimensional 7-9
  one-dimensional 7-2
linear systems of equations
  direct methods (sparse) 4-37
  full 2-15
  iterative methods (sparse) 4-37
  sparse 4-37
linear transformation 2-5
`load`
  sparse matrices 4-18
Lobatto IIIa BVP solver 10-62
LU factorization 2-33
  sparse matrices and reordering 4-30

M

`mat4bvp` demo 10-86
mathematical functions
  as function input arguments 9-1
  finding roots 8-25
  minimizing 8-3
  numerical integration 10-106
mathematical operations
  sparse matrices 4-25
Mathieu's equation (BVP example) 10-65
matrices 2-5
  accessing multiple elements 1-14
  accessing single elements 1-12
  as linear transformation 2-5
  characteristic polynomial 5-11
  characteristic roots 5-11
  concatenating 1-6
  concatenating diagonally 1-41
  constructing a matrix operations
    constructing 1-3
  creating 1-2
  creation 2-5
  data structure query 1-24
  data type query 1-23
  deleting rows and columns 1-29
  determinant 2-26
  diagonal 1-40
  empty 1-43
  expanding 1-25
  flipping 1-32
  full to sparse conversion 4-13
  functions
    changing indexing style 1-72
    creating a matrix 1-70
    determining data type 1-71
    finding matrix structure or shape 1-71
    modifying matrix shape 1-70
    sorting and shifting 1-72
  functions for creating 1-4
  functions for diagonals 1-72
  getting dimensions of 1-22
  identity 2-11
  inverse 2-26
  iterative methods (sparse) 4-38
  linear indexing 1-13
  orthogonal 2-34
  pseudoinverse 2-27
  rank deficiency 2-23
  reshaping 1-30
  rotating 1-32
  scalar 1-45
  shifting 1-35
  sorting column data 1-37
  sorting row data 1-37
  sorting row vectors 1-38
  symmetric 2-8
  transposing 1-31

triangular 2-31
vectors 1-46
matrix operations
addition and subtraction 2-7
concatenating matrices 1-6
creating matrices 1-2
division 2-16
exponentials 2-41
multiplication 2-9
powers 2-40
transpose 2-8
matrix products
Kronecker tensor 2-12
`max` 4-26
`mean` 1-64
minimizing mathematical functions
of one variable 8-3
of several variables 8-5
options 8-12
minimum degree ordering 4-29
Moore-Penrose pseudoinverse 2-27
multidimensional
data gridding 7-9
interpolation 7-8
multidimensional arrays
applying functions 1-64
element-by-element functions 1-64
matrix functions 1-64
vector functions 1-64
cell arrays 1-67
computations on 1-64
creating 1-52
at the command line 1-52
with functions 1-54
with the `cat` function 1-54
extending 1-53
format 1-56
indexing 1-56
avoiding ambiguity 1-60
with the colon operator 1-57
number of dimensions 1-56
organizing data 1-65
permuting dimensions 1-62
removing singleton dimensions 1-61
reshaping 1-60
size of 1-56
storage 1-56
structure arrays 1-68
applying functions 1-69
subscripts 1-51
multistep solver (ODE) 10-6

N

`ndgrid` 1-73
`ndims` 1-56
nearest neighbor interpolation
multidimensional 7-9
one-dimensional 7-2
three-dimensional 7-8
two-dimensional 7-4
`nnz` 4-19
nonstiff ODE examples
rigid body (`rigidode`) 10-16
nonzero elements
maximum number in sparse matrix 4-15
number in sparse matrix 4-19
sparse matrix 4-19
values for sparse matrices 4-19
`nonzeros` 4-19
norms
vector and matrix 2-13
numerical integration 10-106
computing length of curve 10-107
double 10-107
triple 10-106
Nyquist point 11-7
`nzmax` 4-19 4-21

O

objective function
  return values 8-30
ODE 10-2
  coding in MATLAB 10-11
  first order 10-4
  overspecified systems 10-40
ODE solver properties
  fixed step sizes 10-41
ODE solvers
  algorithms
    Adams-Bashworth-Moulton PECE 10-6
    Bogacki-Shampine 10-6
    Dormand-Prince 10-6
    modified Rosenbrock formula 10-7
    numerical differentiation formulas 10-7
  backwards in time 10-44
  basic example
    stiff problem 10-13
  basic syntax 10-8
  calling 10-11
  evaluate solution at specific points 10-16
  examples 10-10
  minimizing output storage 10-41
  minimizing startup cost 10-41
  multistep solver 10-6
  nonstiff problem example 10-11
  nonstiff problems 10-6
  one-step solver 10-6
  performance 10-9
  problem size 10-40
  sampled data 10-44
  stiff problems 10-7 10-13
  troubleshooting 10-39
offsets for indexing 1-59
one-dimensional interpolation 7-2
one-step solver (ODE) 10-6
`ones`
  sparse matrices 4-26
operators
  deletion 1-29
optimization 8-1
  helpful hints 8-31
  options parameters 8-12
  troubleshooting 8-30
  See also minimizing mathematical functions
`orbitode` demo 10-28
organizing data
  multidimensional arrays 1-65
orthogonal matrix 2-34
outer product 2-7
output functions 8-14
overdetermined
  rectangular matrices 2-21
overspecified ODE systems 10-40

P

page subscripts 1-51
partial differential equations. *See* PDE
partial fraction expansion 5-8
PDE 10-87
  defined 10-88
  discretized 10-43
PDE solver (MATLAB) 10-89
  basic syntax 10-89
  performance 10-92
  properties 10-92
`pdex2` demo 10-103
`pdex3` demo 10-103
`pdex5` demo 10-104
performance
  de-emphasizing an ODE solution component 10-42
  improving for BVP solver 10-64
  improving for DDE solver 10-51
  improving for ODE solvers 10-9
  improving for PDE solver 10-92
periodogram 11-11
  0-centered 11-12

permutations 4-26
permute 1-62
permuting array dimensions 1-62
  inverse 1-63
polynomial interpolation 7-2
polynomials
  calculating coefficients from roots 5-5
  calculating roots 5-5
  curve fitting 5-9
  derivatives 5-6
  evaluating 5-4
  multiplying and dividing 5-7
  partial fraction expansion 5-8
  representing as vectors 5-3
preconditioner
  sparse matrices 4-35
pseudoinverse
  of matrix 2-27
pseudorandom 3-1

Q

QR factorization 2-34 4-34
quad, quadl functions
  differ from ODE solvers 10-40
quadrature. *See* numerical integration

R

rand
  sparse matrices 4-26
randn 1-54
random number generators 3-1
rank deficiency
  detecting 2-36
  rectangular matrices 2-23
  sparse matrices 4-35
rectangular matrices
  identity 2-11
  overdetermined systems 2-21
  pseudoinverse 2-27
  QR factorization 2-34
  rank deficient 2-23
  singular value decomposition 2-48
removing
  singleton dimensions 1-61
reorderings 4-26
  for sparser factorizations 4-29
  LU factorization 4-30
  minimum degree ordering 4-29
  reducing bandwidth 4-29
repmap 1-54
reshape 1-60
reshaping
  multidimensional arrays 1-60
reshaping matrices 1-30
rigid body (ODE example) 10-16
rigidode demo 10-16
Robertson problem
  DAE example 10-31
  ODE example 10-39
roots
  of mathematical functions 8-25
  polynomial 5-5
rotating matrices 1-32

S

sampled data
  with ODE solvers 10-44
sampling frequency 11-2
save 4-18
scalar
  as a matrix 2-6
scalar product 2-9
scalars 1-45
Schur decomposition 2-46
second difference operator 4-15
shiftdim 1-73
shifting matrix elements 1-35

shockbvp demo 10-70
sin 1-64
singular value matrix decomposition 2-48
size
  sparse matrices 4-25
solution changes, rapid
  making initial guess 10-70
  verifying consistent behavior 10-73
solving linear systems of equations
  full 2-15
  sparse 4-37
sort 4-29
sorting matrix column data 1-37
sorting matrix row data 1-37
sorting matrix row vectors 1-38
sparse function
  converting full to sparse 4-13
sparse matrix
  advantages 4-11
  Cholesky factorization 4-33
  computational considerations 4-25
  contents 4-19
  conversion from full 4-13
  creating 4-13
    directly 4-14
    from diagonal elements 4-16
  density 4-13
  eigenvalues 4-40
  importing 4-18
  linear systems of equations 4-37
  LU factorization 4-30
    and reordering 4-30
  mathematical operations 4-25
  nonzero elements 4-19
    maximum number 4-15
    specifying when creating matrix 4-14
    storage 4-19
    values 4-19
  nonzero elements of sparse matrix
    number of 4-19
  operations 4-25
  permutation 4-26
  preconditioner 4-35
  propagation through computations 4-25
  QR factorization 4-34
  reordering 4-26
  storage 4-10
    for various permutations 4-28
    viewing 4-19
  triangular factorization 4-30
  viewing contents graphically 4-24
  viewing storage 4-19
sparse matrix functions 1-48
sparse ODE examples
  Brusselator system (brussode) 10-22
spconvert 4-18
spdiags 4-16
spectral analysis 11-2
  audio data 11-14
  basic example 11-10
speye 4-26
spones 4-29
spparms 4-38
sprand 4-26
spy 4-24
squeeze 1-61
  with multidimensional arguments 1-65
startup cost
  minimizing for ODE solvers 10-41
stiff ODE examples
  Brusselator system (brussode) 10-22
  differential-algebraic problem (hb1dae) 10-31
  finite element discretization (fem1ode) 10-19
stiffness (ODE), defined 10-13
storage
  minimizing for ODE problems 10-41
  permutations of sparse matrices 4-28
  sparse matrix 4-10
  viewing for sparse matrix 4-19

structure arrays
growing 1-26 1-28
multidimensional 1-68
applying functions 1-69
subscripting
how MATLAB calculates indices 1-59
multidimensional arrays 1-51
page 1-51
sum 1-64
counting nonzeros in sparse matrix 4-29
sparse matrices 4-26
symamd
minimum degree ordering 4-30
symmetric matrix
transpose 2-8
symrcm
column permutation 4-31
reducing sparse matrix bandwidth 4-29
systems of equations.. *See* linear systems of equations

T

three-dimensional interpolation 7-7
threebvp demo 10-86
transfer functions
using partial fraction expansion 5-8
transpose 1-63
complex conjugate 2-9
unconjugated complex 2-9
transposing matrices 1-31
triangular factorization
sparse matrices 4-30
triangular matrix 2-31
tricubic interpolation 7-8
trigonometric functions 1-64
trilinear interpolation 7-8
troubleshooting (ODE) 10-39
two-dimensional interpolation 7-4
comparing methods graphically 7-5
twobvp demo 10-86

U

unitary matrices
QR factorization 2-34
unknown parameters (BVP) 10-69

V

van der Pol example
simple, nonstiff 10-11
simple, stiff 10-13
vector products
dot or scalar 2-9
outer and inner 2-7
vectors 1-46
column and row 2-6
multiplication 2-7
visualizing solver results
BVP 10-68
DDE 10-53
ODE 10-12
PDE 10-96

W

whos 1-56

Z

zeros 1-54
sparse matrices 4-26