MATLAB®

The Language of Technical Computing

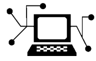
- Computation
- Visualization
- Programming

Mathematics

Version 7

The MathWorks

How to Contact The MathWorks:



`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup

`support@mathworks.com`	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. Mail
3 Apple Hill Drive
Natick, MA 01760-2098

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

MATLAB Mathematics

© COPYRIGHT 1984 — 2005 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, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc. Other product or brand names are 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	Online only	Revised for MATLAB 7.1 (Release 14SP3)

Contents

1 Matrices and Linear Algebra

Function Summary **1-2**

Matrices in MATLAB **1-4**
Creating Matrices **1-4**
Adding and Subtracting Matrices **1-6**
Vector Products and Transpose **1-7**
Multiplying Matrices **1-8**
The Identity Matrix **1-10**
The Kronecker Tensor Product **1-11**
Vector and Matrix Norms **1-12**

Solving Linear Systems of Equations **1-13**
Computational Considerations **1-13**
General Solution **1-15**
Square Systems **1-15**
Overdetermined Systems **1-18**
Underdetermined Systems **1-20**

Inverses and Determinants **1-22**
Overview **1-22**
Pseudoinverses **1-23**

Cholesky, LU, and QR Factorizations **1-27**
Cholesky Factorization **1-27**
LU Factorization **1-29**
QR Factorization **1-30**

Matrix Powers and Exponentials **1-34**

Eigenvalues **1-38**

Singular Value Decomposition **1-42**

2 Polynomials and Interpolation

Polynomials **2-2**
Polynomial Function Summary **2-2**
Representing Polynomials **2-3**
Polynomial Roots **2-3**
Characteristic Polynomials **2-4**
Polynomial Evaluation **2-4**
Convolution and Deconvolution **2-5**
Polynomial Derivatives **2-5**
Polynomial Curve Fitting **2-6**
Partial Fraction Expansion **2-7**

Interpolation **2-9**
Interpolation Function Summary **2-9**
One-Dimensional Interpolation **2-10**
Two-Dimensional Interpolation **2-12**
Comparing Interpolation Methods **2-13**
Interpolation and Multidimensional Arrays **2-15**
Triangulation and Interpolation of Scattered Data **2-18**
Tessellation and Interpolation of Scattered Data in Higher Dimensions **2-26**

Selected Bibliography **2-37**

3 Fast Fourier Transform (FFT)

Introduction **3-2**
Finding an FFT **3-2**
Example: Using FFT to Calculate Sunspot Periodicity **3-3**

Magnitude and Phase of Transformed Data **3-7**

FFT Length Versus Speed **3-9**

Function Summary 3-10

4 Function Functions

Function Summary 4-2

Representing Functions in MATLAB 4-3

Plotting Mathematical Functions 4-5

Minimizing Functions and Finding Zeros 4-8
- Minimizing Functions of One Variable 4-8
- Minimizing Functions of Several Variables 4-9
- Fitting a Curve to Data 4-10
- Setting Minimization Options 4-13
- Output Functions 4-14
- Finding Zeros of Functions 4-21
- Tips 4-25
- Troubleshooting 4-26

Numerical Integration (Quadrature) 4-27
- Example: Computing the Length of a Curve 4-27
- Example: Double Integration 4-28

Parameterizing Functions Called by Function Functions 4-30
- Providing Parameter Values Using Nested Functions 4-30
- Providing Parameter Values to Anonymous Functions 4-31

5 Differential Equations

Initial Value Problems for ODEs and DAEs 5-2
- ODE Function Summary 5-2

Introduction to Initial Value ODE Problems **5-4**
Solvers for Explicit and Linearly Implicit ODEs **5-5**
Examples: Solving Explicit ODE Problems **5-9**
Solver for Fully Implicit ODEs **5-15**
Example: Solving a Fully Implicit ODE Problem **5-16**
Changing ODE Integration Properties **5-17**
Examples: Applying the ODE Initial Value Problem Solvers . **5-18**
Questions and Answers, and Troubleshooting **5-43**

Initial Value Problems for DDEs **5-49**
DDE Function Summary **5-49**
Introduction to Initial Value DDE Problems **5-50**
DDE Solver **5-51**
Solving DDE Problems **5-53**
Discontinuities **5-57**
Changing DDE Integration Properties **5-60**

Boundary Value Problems for ODEs **5-61**
BVP Function Summary **5-62**
Introduction to Boundary Value ODE Problems **5-63**
Boundary Value Problem Solver **5-64**
Changing BVP Integration Properties **5-67**
Solving BVP Problems **5-68**
Using Continuation to Make a Good Initial Guess **5-72**
Solving Singular BVPs **5-80**
Solving Multipoint BVPs **5-84**

Partial Differential Equations **5-89**
PDE Function Summary **5-89**
Introduction to PDE Problems **5-90**
MATLAB Partial Differential Equation Solver **5-91**
Solving PDE Problems **5-94**
Evaluating the Solution at Specific Points **5-99**
Changing PDE Integration Properties **5-100**
Example: Electrodynamics Problem **5-100**

Selected Bibliography **5-106**

6 Sparse Matrices

Function Summary **6-2**

Introduction **6-5**
Sparse Matrix Storage **6-5**
General Storage Information **6-6**
Creating Sparse Matrices **6-7**
Importing Sparse Matrices from Outside MATLAB **6-12**

Viewing Sparse Matrices **6-13**
Information About Nonzero Elements **6-13**
Viewing Sparse Matrices Graphically **6-15**
The find Function and Sparse Matrices **6-16**

Adjacency Matrices and Graphs **6-17**
Introduction to Adjacency Matrices **6-17**
Graphing Using Adjacency Matrices **6-18**
The Bucky Ball **6-18**
An Airflow Model **6-23**

Sparse Matrix Operations **6-25**
Computational Considerations **6-25**
Standard Mathematical Operations **6-25**
Permutation and Reordering **6-26**
Factorization **6-30**
Simultaneous Linear Equations **6-36**
Eigenvalues and Singular Values **6-39**
Performance Limitations **6-41**

Selected Bibliography **6-44**

Index

1

Matrices and Linear Algebra

Function Summary (p. 1-2)	Summarizes the MATLAB® linear algebra functions
Matrices in MATLAB (p. 1-4)	Explains the use of matrices and basic matrix operations in MATLAB
Solving Linear Systems of Equations (p. 1-13)	Discusses the solution of simultaneous linear equations in MATLAB, including square systems, overdetermined systems, and underdetermined systems
Inverses and Determinants (p. 1-22)	Explains the use in MATLAB of inverses, determinants, and pseudoinverses in the solution of systems of linear equations
Cholesky, LU, and QR Factorizations (p. 1-27)	Discusses the solution in MATLAB of systems of linear equations that involve triangular matrices, using Cholesky factorization, Gaussian elimination, and orthogonalization
Matrix Powers and Exponentials (p. 1-34)	Explains the use of MATLAB notation to obtain various matrix powers and exponentials
Eigenvalues (p. 1-38)	Explains eigenvalues and describes eigenvalue decomposition in MATLAB
Singular Value Decomposition (p. 1-42)	Describes singular value decomposition of a rectangular matrix in MATLAB

Function Summary

The linear algebra functions are located in the MATLAB `matfun` directory.

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.
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.
	`luinc`	Incomplete LU factorization.
	`qr`	Orthogonal-triangular decomposition.

Function Summary (Continued)

Category	Function	Description
	`lsqnonneg`	Nonnegative least-squares.
	`pinv`	Pseudoinverse.
	`lscov`	Least squares with known covariance.
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 MATLAB

A *matrix* is a two-dimensional array of real or complex numbers. *Linear algebra* defines many matrix operations that are directly supported by MATLAB. Linear algebra includes matrix arithmetic, linear equations, eigenvalues, singular values, and matrix factorizations.

For more information about creating and working with matrices, see Data Structures in the MATLAB Programming documentation.

This section describes the following topics:

- “Creating Matrices” on page 1-4
- “Adding and Subtracting Matrices” on page 1-6
- “Vector Products and Transpose” on page 1-7
- “Vector Products and Transpose” on page 1-7
- “Multiplying Matrices” on page 1-8
- “The Identity Matrix” on page 1-10
- “The Kronecker Tensor Product” on page 1-11
- “Vector and Matrix Norms” on page 1-12

Creating Matrices

Informally, the terms matrix and array are often used interchangeably. More precisely, a matrix is a two-dimensional rectangular array of real or complex numbers that represents a linear transformation. The linear algebraic operations defined on matrices have found applications in a wide variety of technical fields. (The optional Symbolic Math Toolbox extends the capabilities of MATLAB to operations on various types of nonnumeric matrices.)

MATLAB has dozens of functions that create different kinds of matrices. Two of them can be used 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
```

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 (or single quote) to denote transpose. 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' denotes the *complex conjugate transpose*, where the sign of the complex part of each element is reversed. The unconjugated complex transpose, where the complex part of each element retains its sign, is denoted by z.'. So if

```
z = [1+2i 3+4i]
```

then z' is

```
1-2i
3-4i
```

while z.' is

```
1+2i
3+4i
```

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
```

The 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 upper and lowercase letters and i already served double duty 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.

The 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
```

Solving Linear Systems of Equations

This section describes

- Computational considerations
- The general solution to a system

It also discusses particular solutions to

- Square systems
- Overdetermined systems
- Underdetermined systems

Computational Considerations

One of the most important problems in technical computing is the solution of simultaneous linear equations. In matrix notation, this problem can be stated as follows.

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.

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 $7^{-1} = 0.142857\ldots$, and then multiplying 7^{-1} by 21. This would be more work and, if 7^{-1} is represented to a finite number of digits, less accurate. Similar considerations apply to sets of linear equations with more than one unknown; MATLAB solves such equations without computing the inverse of the matrix.

Although it is not standard mathematical notation, MATLAB uses the division terminology familiar in the scalar case to describe the solution of a general system of simultaneous equations. The two division symbols, *slash*, /, and

backslash, \, are used for the two situations where the unknown matrix appears on the left or right of the coefficient matrix:

`X = A\B`	Denotes the solution to the matrix equation $AX = B$.
`X = B/A`	Denotes the solution to the matrix equation $XA = B$.

You can 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, backslash is used far more frequently than 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 backslash operator employs different algorithms to handle different kinds of coefficient matrices. The various cases, which are diagnosed automatically by examining the coefficient matrix, include

- Permutations of triangular matrices
- Symmetric, positive definite matrices
- Square, nonsingular matrices
- Rectangular, overdetermined systems
- Rectangular, underdetermined systems

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 non-homogeneous 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, then 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 one. A later section considers the effects of roundoff error inherent in more realistic computations.

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 1-23.

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
```

You can 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]`, then $AX = b$ 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:

$$y(t) \approx c_1 + c_2 e^{-t}$$

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
```

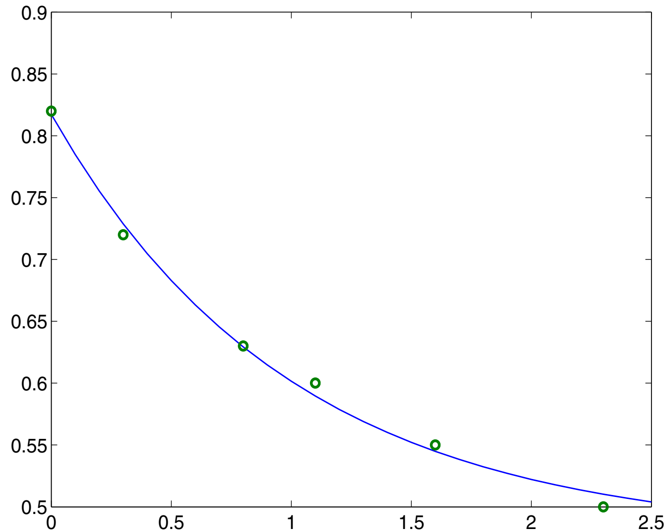
In other words, the least squares fit to the data is

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

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')
```

You can see that `E*c` is not exactly equal to `y`, but that 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 squares solution that has at most `rank(A)` nonzeros.

Underdetermined Systems

Underdetermined linear systems involve more unknowns than equations. The solution to such underdetermined systems is not unique. The matrix left division operation in MATLAB finds a basic solution, which has at most `m` nonzero components.

Here is a small, random example:

```
R = [6 8 7 3; 3 5 4 1]
R =
     6     8     7     3
     3     5     4     1

rand('state', 0);
```

```
b = fix(10*rand(2,1))
b =
       9
       2
```

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

```
format rat
p = R\b
p =
       0
      -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`.

Inverses and Determinants

This section provides

- An overview of the use of inverses and determinants for solving square nonsingular systems of linear equations
- A discussion of the Moore-Penrose pseudoinverse for solving rectangular systems of linear equations

Overview

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` would theoretically be the same as `X = A\B` and `Y = B*inv(A)` would theoretically be 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
rand('state', 0);
C = fix(10*rand(3,2));
```

```
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, then 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 singular to working precision.

z =
     Inf
     Inf
     Inf
```

Cholesky, LU, and QR Factorizations

The MATLAB linear equation capabilities are based on three basic matrix factorizations:

- Cholesky factorization for symmetric, positive definite matrices
- LU factorization (Gaussian elimination) for general square matrices
- QR factorization (orthogonal) for rectangular matrices

These three factorizations are available through the `chol`, `lu`, and `qr` functions.

All three of these factorizations 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 which 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 MATLAB 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:

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

Q =
    -0.8182    0.3999   -0.4131
    -0.1818   -0.8616   -0.4739
    -0.5455   -0.3126    0.7777

R =
  -11.0000   -8.5455
         0   -7.4817
         0         0
```

In many cases, the last $m - n$ columns of Q are not needed because they are multiplied by the zeros in the bottom portion of R. So the *economy* size QR factorization produces a rectangular, m-by-n Q with orthonormal columns and a square n-by-n upper triangular R. For the 3-by-2 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 =
    -0.8182    0.3999
    -0.1818   -0.8616
    -0.5455   -0.3126

R =
  -11.0000   -8.5455
         0   -7.4817
```

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)
```

is equal to

```
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.

Matrix Powers and Exponentials

This section tells you how to obtain the following matrix powers and exponentials in MATLAB:

- Positive integer
- Inverse and fractional
- Element-by-element
- Exponentials

Positive Integer Powers

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

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

A =

     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, then `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
```

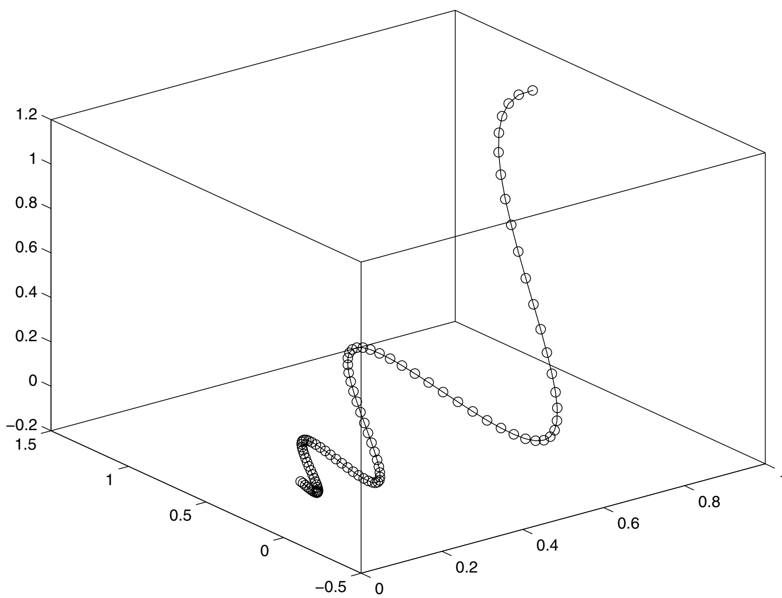
The matrix exponential is used to compute the solution, $x(t)$, to the differential equation at 101 points on the interval $0 \le t \le 1$ with

```
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

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

$$Av = \lambda v$$

This section explains:

- Eigenvalue decomposition
- Problems associated with defective (not diagonalizable) matrices
- The use of Schur decomposition to avoid problems associated with eigenvalue decomposition

Eigenvalue Decomposition

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`.

Defective Matrices

Some matrices do not have an eigenvector decomposition. These matrices are *defective*, or *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.

The optional Symbolic Math Toolbox extends the capabilities of MATLAB by connecting to Maple, a powerful computer algebra system. One of the functions provided by the toolbox computes the Jordan Canonical Form. This is appropriate for matrices like the example given here, which is 3-by-3 and has exactly known, integer elements:

```
[X,J] = jordan(A)

X =
   -1.7500    1.5000    2.7500
    3.0000   -3.0000   -3.0000
   -1.2500    1.5000    1.2500

J =
     -1     0     0
      0     1     1
      0     0     1
```

The Jordan Canonical Form is an important theoretical concept, but it is not a reliable computational tool for larger matrices, or for matrices whose elements are subject to roundoff errors and other uncertainties.

Schur Decomposition in MATLAB Matrix Computations

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

$$A = U\,S\,U^T$$

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 Value Decomposition

A *singular value* and corresponding *singular vectors* of a rectangular matrix A are 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 roundoff 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 roundoff error.

2

Polynomials and Interpolation

Polynomials (p. 2-2)	Functions for standard polynomial operations. Additional topics include curve fitting and partial fraction expansion.
Interpolation (p. 2-9)	Two- and multi-dimensional interpolation techniques, taking into account speed, memory, and smoothness considerations.
Selected Bibliography (p. 2-37)	Published materials that support concepts implemented in "Polynomials and Interpolation"

Polynomials

This section provides

- A summary of the MATLAB polynomial functions
- Instructions for representing polynomials in MATLAB

It also describes the MATLAB polynomial functions that

- Calculate the roots of a polynomial
- Calculate the coefficients of the characteristic polynomial of a matrix
- Evaluate a polynomial at a specified value
- Convolve (multiply) and deconvolve (divide) polynomials
- Compute the derivative of a polynomial
- Fit a polynomial to a set of data
- Convert between partial fraction expansion and polynomial coefficients

Polynomial Function Summary

MATLAB provides functions for standard polynomial operations, such as polynomial roots, evaluation, and differentiation. In addition, there are functions for more advanced applications, such as curve fitting and partial fraction expansion.

The polynomial functions reside in the MATLAB `polyfun` directory.

Polynomial Function Summary

Function	Description
`conv`	Multiply polynomials.
`deconv`	Divide polynomials.
`poly`	Polynomial with specified roots.
`polyder`	Polynomial derivative.
`polyfit`	Polynomial curve fitting.
`polyval`	Polynomial evaluation.

Polynomial Function Summary (Continued)

Function	Description
`polyvalm`	Matrix polynomial evaluation.
`residue`	Partial-fraction expansion (residues).
`roots`	Find polynomial roots.

The Symbolic Math Toolbox contains additional specialized support for polynomial operations.

Representing Polynomials

MATLAB 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];
```

Polynomial 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, MATLAB 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.

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.)

Polynomial Evaluation

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
```

Convolution and Deconvolution

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
```

Polynomial 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.

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

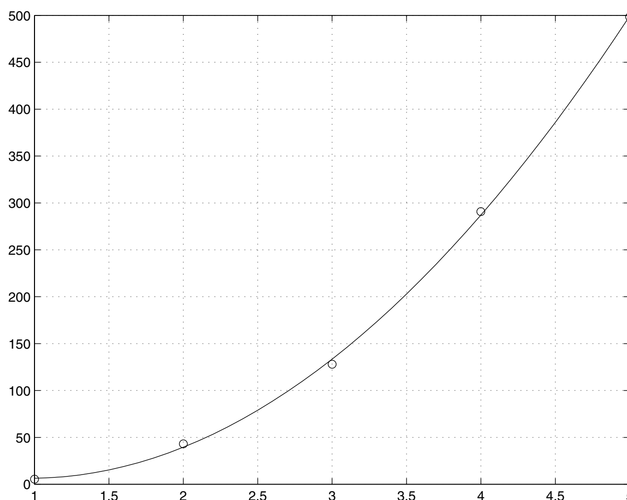
```
x = [1 2 3 4 5]; y = [5.5 43.1 128 290.7 498.4];
```

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
```



To use these functions in an application example, see “Data Fitting Using Linear Regression” in the MATLAB Data Analysis book.

Partial Fraction Expansion

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
```

Interpolation

Interpolation is a process for estimating values that lie between known data points. It has important applications in areas such as signal and image processing.

This section

- Provides a summary of the MATLAB interpolation functions
- Discusses one-dimensional interpolation
- Discusses two-dimensional interpolation
- Uses an example to compare nearest neighbor, bilinear, and bicubic interpolation methods
- Discusses interpolation of multidimensional data
- Discusses triangulation and interpolation of scattered data

Interpolation Function Summary

MATLAB provides a number of interpolation techniques that let you balance the smoothness of the data fit with speed of execution and memory usage.

The interpolation functions reside in the MATLAB `polyfun` directory.

Interpolation Function Summary

Function	Description
`griddata`	Data gridding and surface fitting.
`griddata3`	Data gridding and hypersurface fitting for three-dimensional data.
`griddatan`	Data gridding and hypersurface fitting (dimension >= 3).
`interp1`	One-dimensional interpolation (table lookup).
`interp2`	Two-dimensional interpolation (table lookup).
`interp3`	Three-dimensional interpolation (table lookup).
`interpft`	One-dimensional interpolation using FFT method.

Interpolation Function Summary (Continued)

Function	Description
`interpn`	N-dimensional interpolation (table lookup).
`mkpp`	Make a piecewise polynomial
`pchip`	Piecewise Cubic Hermite Interpolating Polynomial (PCHIP).
`ppval`	Piecewise polynomial evaluation
`spline`	Cubic spline data interpolation
`unmkpp`	Piecewise polynomial details

One-Dimensional Interpolation

There are two kinds of one-dimensional interpolation in MATLAB:

- Polynomial interpolation
- FFT-based interpolation

Polynomial Interpolation

The function `interp1` performs one-dimensional interpolation, an important operation for data analysis and curve fitting. 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 non-uniform 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.

The relative performance of each method holds true even for interpolation of two-dimensional or multidimensional data. For a graphical comparison of

interpolation methods, see the section "Comparing Interpolation Methods" on page 2-13.

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 Interpolation

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.

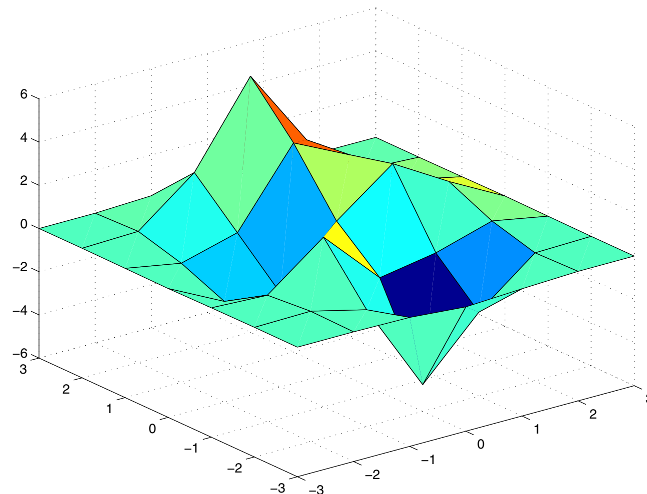
All of these methods require that `X` and `Y` be monotonic, that is, either always increasing or always decreasing from point to point. You should prepare these matrices using the `meshgrid` function, or else be sure that the “pattern” of the points emulates the output of `meshgrid`. In addition, each 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'`.

Comparing Interpolation 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');
```

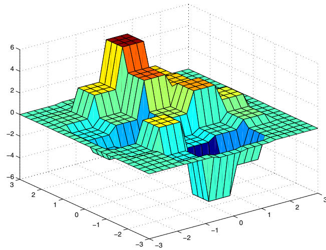
4 Interpolate using bilinear interpolation:

```
zi2 = interp2(x,y,z,xi,yi,'bilinear');
```

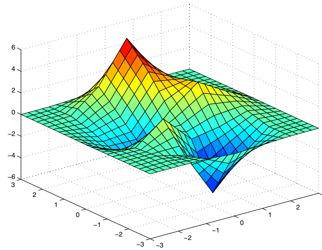
5 Interpolate using bicubic interpolation:

```
zi3 = interp2(x,y,z,xi,yi,'bicubic');
```

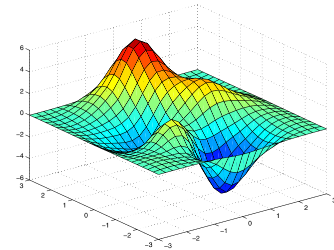
6 Compare the surface plots for the different interpolation methods.



```
surf(xi,yi,zi1)
% nearest
```

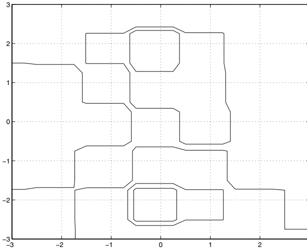


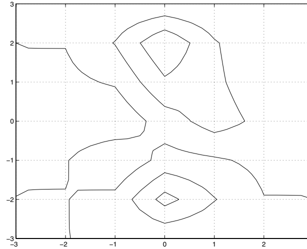
```
surf(xi,yi,zi2)
% bilinear
```

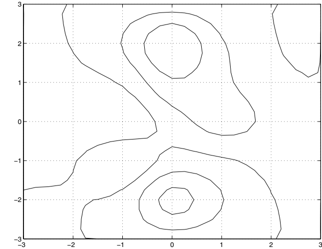


```
surf(xi,yi,zi3)
% bicubic
```

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.

Interpolation and Multidimensional Arrays

Several interpolation functions 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).

This section discusses

- Interpolation of three-dimensional data
- Interpolation of higher dimensional data
- Multidimensional data gridding

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 ith dimension of these output arrays are copies of the elements of input vector x_i.

The syntax for ndgrid is

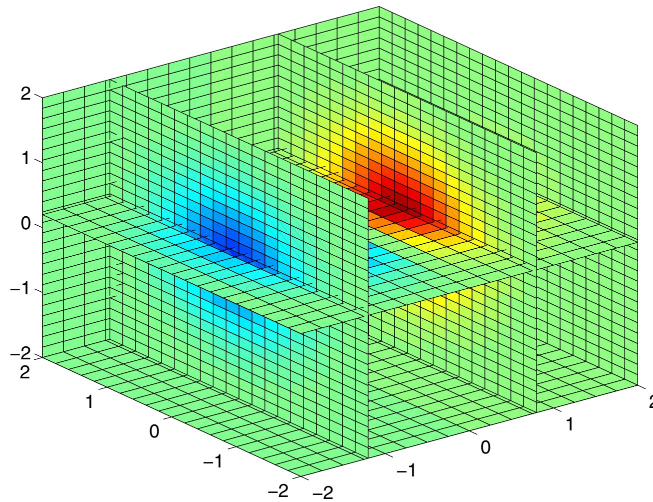
```
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)
```

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])
```



Triangulation and Interpolation of Scattered Data

MATLAB provides routines that aid in the analysis of closest-point problems and geometric analysis.

Functions for Analysis of Closest-Point Problems and Geometric Analysis

Function	Description
`convhull`	Convex hull.
`delaunay`	Delaunay triangulation.

Functions for Analysis of Closest-Point Problems and Geometric Analysis

Function	Description
`delaunay3`	3-D Delaunay tessellation.
`dsearch`	Nearest point search of Delaunay triangulation.
`inpolygon`	True for points inside polygonal region.
`polyarea`	Area of polygon.
`rectint`	Area of intersection for two or more rectangles.
`tsearch`	Closest triangle search.
`voronoi`	Voronoi diagram.

This section applies the following techniques to the `seamount` data set supplied with MATLAB:

- Convex hulls
- Delaunay triangulation
- Voronoi diagrams

See also “Tessellation and Interpolation of Scattered Data in Higher Dimensions” on page 2-26.

Note Examples in this section use the MATLAB `seamount` data set. Seamounts are underwater mountains. They are valuable sources of information about marine geology. The `seamount` data set represents the surface, in 1984, of the seamount designated LR148.8W located at 48.2°S, 148.8°W on the Louisville Ridge in the South Pacific. For more information about the data and its use, see Parker [2].

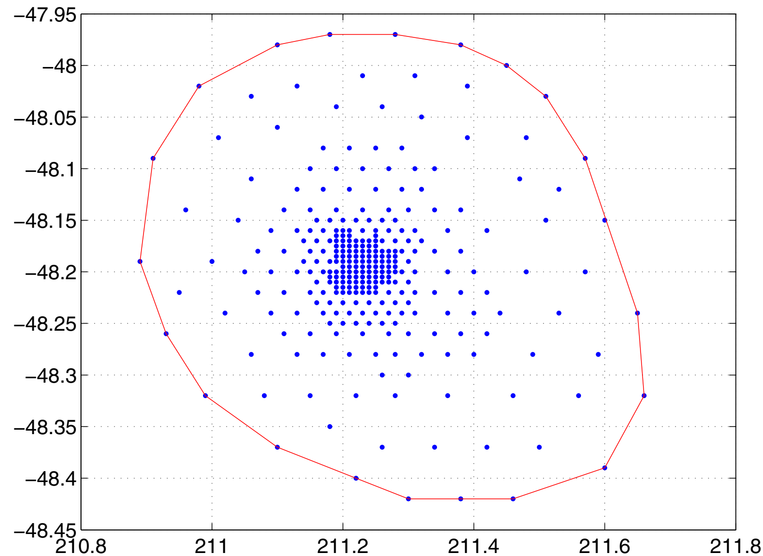
The `seamount` data set provides longitude (`x`), latitude (`y`) and depth-in-feet (`z`) data for 294 points on the seamount LR148.8W.

Convex Hulls

The `convhull` function returns the indices of the points in a data set that comprise the convex hull for the set. Use the `plot` function to plot the output of `convhull`.

This example loads the `seamount` data and plots the longitudinal (`x`) and latitudinal (`y`) data as a scatter plot. It then generates the convex hull and uses `plot` to plot the convex hull:

```
load seamount
plot(x,y,'.','markersize',10)
k = convhull(x,y);
hold on, plot(x(k),y(k),'-r'), hold off
grid on
```



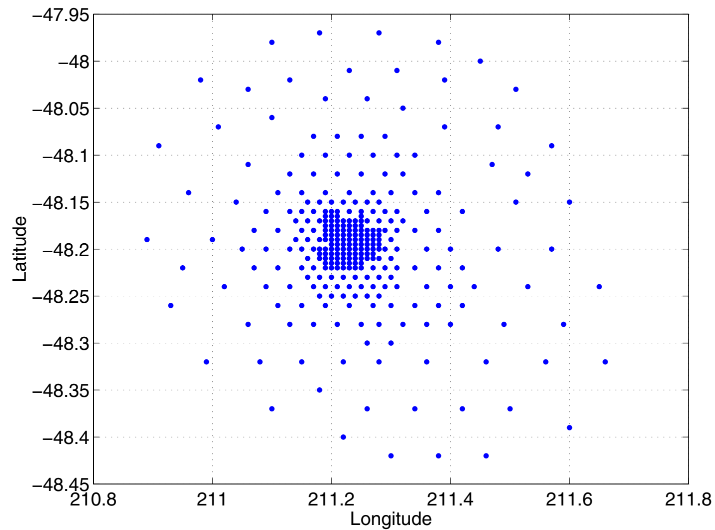
Delaunay Triangulation

Given a set of coplanar data points, *Delaunay triangulation* is a set of lines connecting each point to its natural neighbors. The `delaunay` function returns a Delaunay triangulation as a set of triangles having the property that, for each triangle, the unique circle circumscribed about the triangle contains no data points.

You can use triplot to print the resulting triangles in two-dimensional space. You can also add data for a third dimension to the output of delaunay and plot the result as a surface with trisurf, or as a mesh with trimesh.

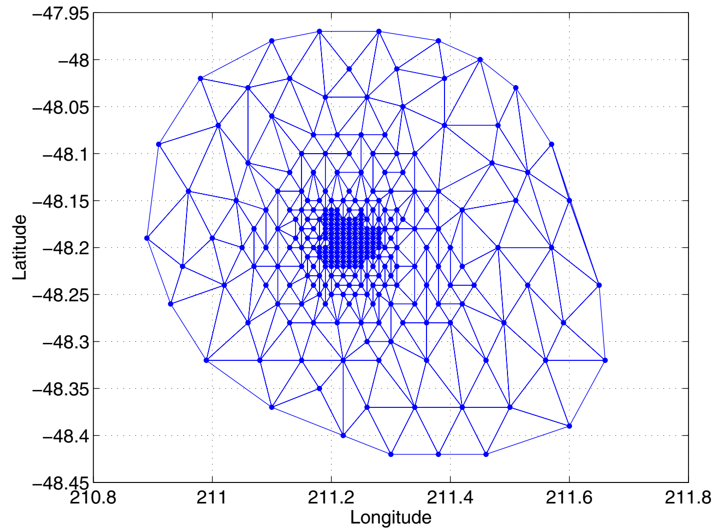
Plotting a Delaunay Triangulation. To try delaunay, load the seamount data set and view the longitude (x) and latitude (y) data as a scatter plot:

```
load seamount
plot(x,y,'.','markersize',12)
xlabel('Longitude'), ylabel('Latitude')
grid on
```



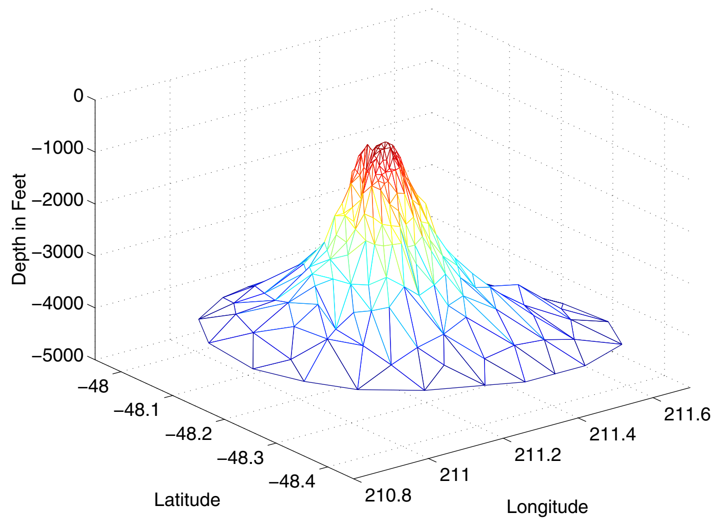
Apply Delaunay triangulation and use triplot to overplot the resulting triangles on the scatter plot:

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



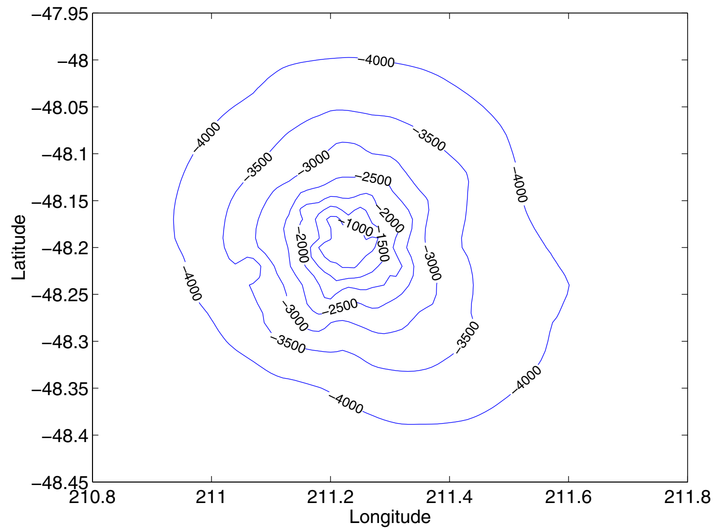
Mesh and Surface Plots. Add the depth data (`z`) from `seamount`, to the Delaunay triangulation, and use `trimesh` to produce a mesh in three-dimensional space. Similarly, you can use `trisurf` to produce a surface:

```
figure
hidden on
trimesh(tri,x,y,z)
grid on
xlabel('Longitude'); ylabel('Latitude'); zlabel('Depth in Feet')
```



Contour Plots. This code uses `meshgrid`, `griddata`, and `contour` to produce a contour plot of the `seamount` data:

```
figure
[xi,yi] = meshgrid(210.8:.01:211.8,-48.5:.01:-47.9);
zi = griddata(x,y,z,xi,yi,'cubic');
[c,h] = contour(xi,yi,zi,'b-');
clabel(c,h)
xlabel('Longitude'), ylabel('Latitude')
```



The arguments for meshgrid encompass the largest and smallest `x` and `y` values in the original `seamount` data. To obtain these values, use `min(x)`, `max(x)`, `min(y)`, and `max(y)`.

Closest-Point Searches. You can search through the Delaunay triangulation data with two functions:

- `dsearch` finds the indices of the `(x,y)` points in a Delaunay triangulation closest to the points you specify. This code searches for the point closest to (211.32, -48.35) in the triangulation of the `seamount` data.

  ```
  xi = 211.32; yi = -48.35;
  p = dsearch(x,y,tri,xi,yi);
  [x(p), y(p)]

  ans =
      211.3400  -48.3700
  ```

- `tsearch` finds the indices into the `delaunay` output that specify the enclosing triangles of the points you specify. This example uses the index of the

enclosing triangle for the point (211.32, -48.35) to obtain the coordinates of the vertices of the triangle:

```
xi = 211.32; yi = -48.35;
t = tsearch(x,y,tri,xi,yi);
r = tri(t,:);
A = [x(r) y(r)]

A =
  211.3000  -48.3000
  211.3400  -48.3700
  211.2800  -48.3200
```

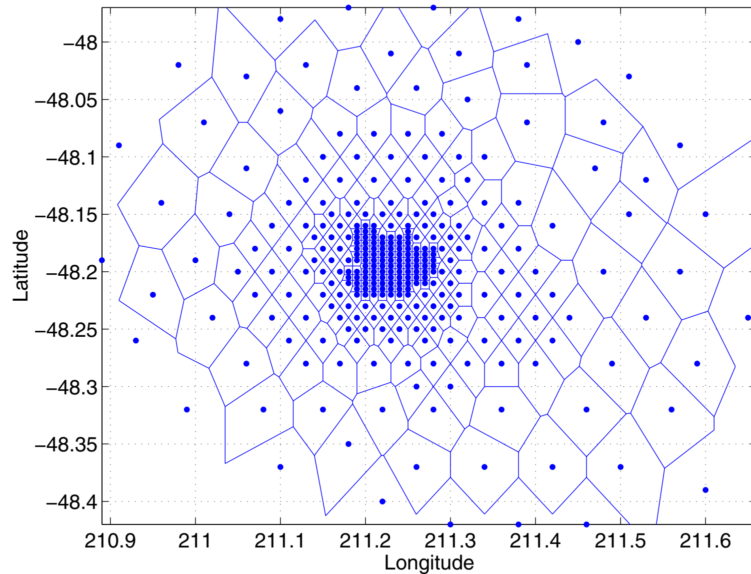
Voronoi Diagrams

Voronoi diagrams are a closest-point plotting technique related to Delaunay triangulation.

For each point in a set of coplanar points, you can draw a polygon that encloses all the intermediate points that are closer to that point than to any other point in the set. Such a polygon is called a *Voronoi polygon*, and the set of all Voronoi polygons for a given point set is called a *Voronoi diagram*.

The `voronoi` function can plot the cells of the Voronoi diagram, or return the vertices of the edges of the diagram. This example loads the `seamount` data, then uses the `voronoi` function to produce the Voronoi diagram for the longitudinal (`x`) and latitudinal (`y`) dimensions. Note that `voronoi` plots only the bounded cells of the Voronoi diagram:

```
load seamount
voronoi(x,y)
grid on
xlabel('Longitude'), ylabel('Latitude')
```



Note See the `voronoi` function for an example that uses the vertices of the edges to plot a Voronoi diagram.

Tessellation and Interpolation of Scattered Data in Higher Dimensions

Many applications in science, engineering, statistics, and mathematics require structures like convex hulls, Voronoi diagrams, and Delaunay tessellations. Using Qhull [1], MATLAB functions enable you to geometrically analyze data sets in any dimension.

Functions for Multidimensional Geometrical Analysis

Function	Description
`convhulln`	N-dimensional convex hull.
`delaunayn`	N-dimensional Delaunay tessellation.
`dsearchn`	N-dimensional nearest point search.
`griddatan`	N-dimensional data gridding and hypersurface fitting.
`tsearchn`	N-dimensional closest simplex search.
`voronoin`	N-dimensional Voronoi diagrams.

This section demonstrates these geometric analysis techniques:

- Convex hulls
- Delaunay triangulations
- Voronoi diagrams
- Interpolation of scattered multidimensional data

Convex Hulls

The convex hull of a data set in n-dimensional space is defined as the smallest convex region that contains the data set.

Computing a Convex Hull. The `convhulln` function returns the indices of the points in a data set that comprise the facets of the convex hull for the set. For example, suppose `X` is an 8-by-3 matrix that consists of the 8 vertices of a cube. The convex hull of `X` then consists of 12 facets:

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)];       % 8 corner points of a cube
C = convhulln(X)

C =
     4     2     1
     3     4     1
```

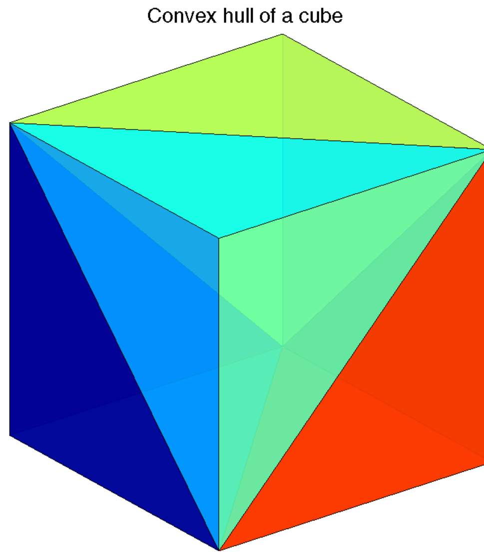
```
    7     3     1
    5     7     1
    7     4     3
    4     7     8
    2     6     1
    6     5     1
    4     6     2
    6     4     8
    6     7     5
    7     6     8
```

Because the data is three-dimensional, the facets that make up the convex hull are triangles. The 12 rows of `C` represent 12 triangles. The elements of C are indices of points in `X`. For example, the first row, 3 1 5, means that the first triangle has `X(3,:)`, `X(1,:)`, and `X(5,:)` as its vertices.

For three-dimensional convex hulls, you can use `trisurf` to plot the output. However, using `patch` to plot the output gives you more control over the color of the facets. Note that you cannot plot `convhulln` output for `n > 3`.

This code plots the convex hull by drawing the triangles as three-dimensional patches:

```
figure, hold on
d = [1 2 3 1];      % Index into C column.
for i = 1:size(C,1) % Draw each triangle.
    j= C(i,d);      % Get the ith C to make a patch.
    h(i)=patch(X(j,1),X(j,2),X(j,3),i,'FaceAlpha',0.9);
end                 % 'FaceAlpha' is used to make it transparent.
hold off
view(3), axis equal, axis off
camorbit(90,-5);    % To view it from another angle
title('Convex hull of a cube')
```



Delaunay Tessellations

A Delaunay tessellation is a set of simplices with the property that, for each simplex, the unique sphere circumscribed about the simplex contains no data points. In two-dimensional space, a simplex is a triangle. In three-dimensional space, a simplex is a tetrahedron.

Computing a Delaunay Tessellation. The `delaunayn` function returns the indices of the points in a data set that comprise the simplices of an n-dimensional Delaunay tessellation of the data set.

This example uses the same `X` as in the convex hull example, i.e. the 8 corner points of a cube, with the addition of a center point:

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)];   % 8 corner points of a cube
X(9,:) = [0 0 0];       % Add center to the vertex list.
T = delaunayn(X)      % Generate Delaunay tessellation.
```

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

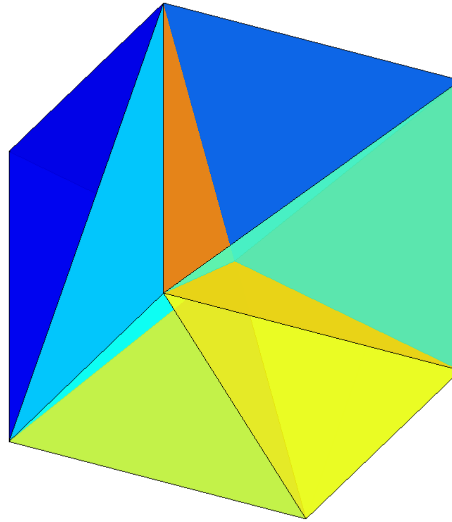
The 12 rows of `T` represent the 12 simplices, in this case irregular tetrahedrons, that partition the cube. Each row represents one tetrahedron, and the row elements are indices of points in `X`.

For three-dimensional tessellations, you can use `tetramesh` to plot the output. However, using `patch` to plot the output gives you more control over the color of the facets. Note that you cannot plot `delaunayn` output for `n > 3`.

This code plots the tessellation `T` by drawing the tetrahedrons using three-dimensional patches:

```
figure, hold on
d = [1 1 1 2; 2 2 3 3; 3 4 4 4];  % Index into T
for i = 1:size(T,1)      % Draw each tetrahedron.
   y = T(i,d);           % Get the ith T to make a patch.
   x1 = reshape(X(y,1),3,4);
   x2 = reshape(X(y,2),3,4);
   x3 = reshape(X(y,3),3,4);
   h(i)=patch(x1,x2,x3,(1:4)*i,'FaceAlpha',0.9);
end
hold off
view(3), axis equal
axis off
camorbit(65,120)         % To view it from another angle
title('Delaunay tessellation of a cube with a center point')
```

You can use cameramenu to rotate the figure in any direction.



Voronoi Diagrams

Given m data points in n-dimensional space, a *Voronoi diagram* is the partition of n-dimensional space into m polyhedral regions, one region for each data point. Such a region is called a *Voronoi cell*. A Voronoi cell satisfies the condition that it contains all points that are closer to its data point than any other data point in the set.

Computing a Voronoi Diagram. The voronoin function returns two outputs:

- V is an m-by-n matrix of m points in n-space. Each row of V represents a Voronoi vertex.
- C is a cell array of vectors. Each vector in the cell array C represents a Voronoi cell. The vector contains indices of the points in V that are the vertices of the Voronoi cell. Each Voronoi cell may have a different number of points.

Because a Voronoi cell can be unbounded, the first row of V is a point at infinity. Then any unbounded Voronoi cell in C includes the point at infinity, i.e., the first point in V.

This example uses the same X as in the Delaunay example, i.e., the 8 corner points of a cube and its center. Random noise is added to make the cube less regular. The resulting Voronoi diagram has 9 Voronoi cells:

```
d = [-1 1];
[x,y,z] = meshgrid(d,d,d);
X = [x(:),y(:),z(:)];       % 8 corner points of a cube
X(9,:) = [0 0 0];           % Add center to the vertex list.
X = X+0.01*rand(size(X));  % Make the cube less regular.
[V,C] = voronoin(X);

V =
       Inf       Inf       Inf
    0.0055    1.5054    0.0004
    0.0037    0.0101   -1.4990
    0.0052    0.0087   -1.4990
    0.0030    1.5054    0.0030
    0.0072    0.0072    1.4971
   -1.7912    0.0000    0.0044
   -1.4886    0.0011    0.0036
   -1.4886    0.0002    0.0045
    0.0101    0.0044    1.4971
    1.5115    0.0074    0.0033
    1.5115    0.0081    0.0040
    0.0104   -1.4846   -0.0007
    0.0026   -1.4846    0.0071

C =
     [1x8 double]
     [1x6 double]
     [1x4 double]
     [1x6 double]
     [1x6 double]
     [1x6 double]
     [1x6 double]
     [1x6 double]
     [1x12 double]
```

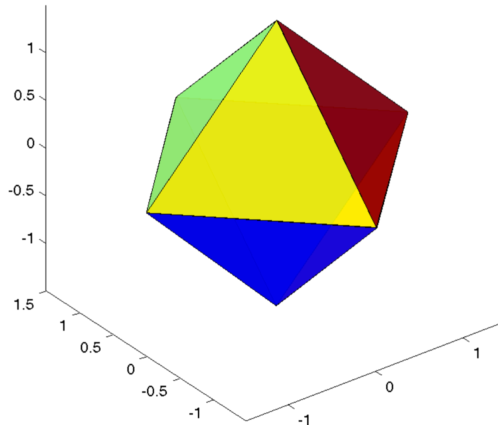
In this example, `V` is a 13-by-3 matrix, the 13 rows are the coordinates of the 13 Voronoi vertices. The first row of `V` is a point at infinity. `C` is a 9-by-1 cell array, where each cell in the array contains an index vector into `V` corresponding to one of the 9 Voronoi cells. For example, the 9th cell of the Voronoi diagram is

```
C{9} = 2 3 4 5 6 7 8 9 10 11 12 13
```

If any index in a cell of the cell array is `1`, then the corresponding Voronoi cell contains the first point in `V`, a point at infinity. This means the Voronoi cell is unbounded.

To view a *bounded* Voronoi cell, i.e., one that does not contain a point at infinity, use the `convhulln` function to compute the vertices of the facets that make up the Voronoi cell. Then use `patch` and other plot functions to generate the figure. For example, this code plots the Voronoi cell defined by the 9th cell in `C`:

```
X = V(C{9},:);       % View 9th Voronoi cell.
K = convhulln(X);
figure
hold on
d = [1 2 3 1];       % Index into K
for i = 1:size(K,1)
   j = K(i,d);
   h(i)=patch(X(j,1),X(j,2),X(j,3),i,'FaceAlpha',0.9);
end
hold off
view(3)
axis equal
title('One cell of a Voronoi diagram')
```



Interpolating N-Dimensional Data

Use the `griddatan` function to interpolate multidimensional data, particularly scattered data. `griddatan` uses the `delaunayn` function to tessellate the data, and then interpolates based on the tessellation.

Suppose you want to visualize a function that you have evaluated at a set of `n` scattered points. In this example, `X` is an `n`-by-3 matrix of points, each row containing the `(x,y,z)` coordinates for one of the points. The vector `v` contains the `n` function values at these points. The function for this example is the squared distance from the origin, `v = x.^2 + y.^2 + z.^2`.

Start by generating `n` = 5000 points at random in three-dimensional space, and computing the value of a function on those points:

```
n = 5000;
X = 2*rand(n,3)-1;
v = sum(X.^2,2);
```

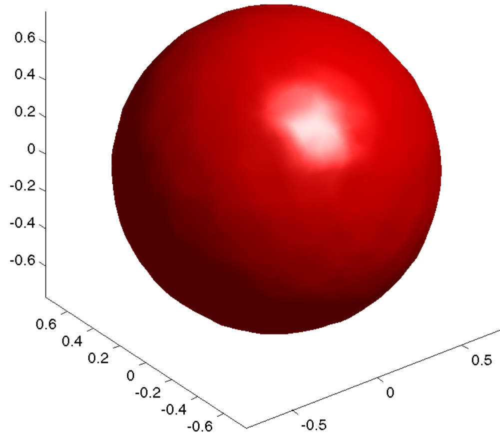
The next step is to use interpolation to compute function values over a grid. Use `meshgrid` to create the grid, and `griddatan` to do the interpolation:

```
delta = 0.05;
d = -1:delta:1;
[x0,y0,z0] = meshgrid(d,d,d);
X0 = [x0(:), y0(:), z0(:)];
v0 = griddatan(X,v,X0);
v0 = reshape(v0, size(x0));
```

Then use `isosurface` and related functions to visualize the surface that consists of the (x,y,z) values for which the function takes a constant value. You could pick any value, but the example uses the value 0.6. Since the function is the squared distance from the origin, the surface at a constant value is a sphere:

```
p = patch(isosurface(x0,y0,z0,v0,0.6));
isonormals(x0,y0,z0,v0,p);
set(p,'FaceColor','red','EdgeColor','none');
view(3);
camlight;
lighting phong
axis equal
title('Interpolated sphere from scattered data')
```

Note A smaller `delta` produces a smoother sphere, but increases the compute time.


Interpolated sphere from scattered data
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.5
0
0.5

Selected Bibliography

[1] National Science and Technology Research Center for Computation and Visualization of Geometric Structures (The Geometry Center), University of Minnesota. 1993. For information about `qhull`, see `http://www.qhull.org`.

[2] Parker, Robert. L., Loren Shure, & John A. Hildebrand, "The Application of Inverse Theory to Seamount Magnetism." *Reviews of Geophysics*. Vol. 25, No. 1, 1987.

3

Fast Fourier Transform (FFT)

Introduction (p. 3-2)	Introduces Fourier transform analysis with an example about sunspot activity
Magnitude and Phase of Transformed Data (p. 3-7)	Calculates magnitude and phase of transformed data
FFT Length Versus Speed (p. 3-9)	Discusses the dependence of execution time on length of the transform
Function Summary (p. 3-10)	Summarizes the Fourier transform functions

The fast Fourier transform (FFT) is an efficient algorithm for computing the discrete Fourier transform (DFT) of a sequence; it is not a separate transform. It is particularly useful in areas such as signal and image processing, where its uses range from filtering, convolution, and frequency analysis to power spectrum estimation.

Introduction

For length N input sequence x, the DFT is a length N vector, X. `fft` and `ifft` implement the relationships

$$X(k) = \sum_{n=1}^{N} x(n) e^{-j2\pi(k-1)\left(\frac{n-1}{N}\right)} \qquad 1 \le k \le N$$

$$x(n) = \frac{1}{N} \sum_{k=1}^{N} X(k) e^{j2\pi(k-1)\left(\frac{n-1}{N}\right)} \qquad 1 \le n \le N$$

Note Since the first element of a MATLAB vector has an index 1, the summations in the equations above are from 1 to N. These produce identical results as traditional Fourier equations with summations from 0 to N-1.

If $x(n)$ is real, you can rewrite the above equation in terms of a summation of sine and cosine functions with real coefficients:

$$x(n) = \frac{1}{N} \sum_{k=1}^{N} a(k) \cos\left(\frac{2\pi(k-1)(n-1)}{N}\right) + b(k) \sin\left(\frac{2\pi(k-1)(n-1)}{N}\right)$$

where

$$a(k) = \mathrm{real}(X(k)), \; b(k) = -\mathrm{imag}(X(k)), \; 1 \le n \le N$$

Finding an FFT

The FFT of a column vector x

```
x = [4 3 7 -9 1 0 0 0]' ;
```

is found with

```
y = fft(x)
```

which results in

```
y =
6.0000
11.4853 - 2.7574i
-2.0000 -12.0000i
-5.4853 +11.2426i
18.0000
-5.4853 -11.2426i
-2.0000 +12.0000i
11.4853 + 2.7574i
```

Notice that although the sequence x is real, y is complex. The first component of the transformed data is the constant contribution and the fifth element corresponds to the Nyquist frequency. The last three values of y correspond to negative frequencies and, for the real sequence x, they are complex conjugates of three components in the first half of y.

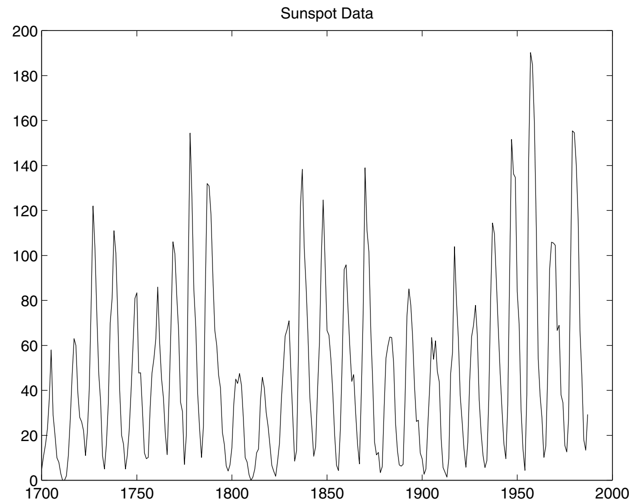
Example: Using FFT to Calculate Sunspot Periodicity

Suppose, you want to analyze the variations in sunspot activity over the last 300 years. You are probably aware that sunspot activity is cyclical, reaching a maximum about every 11 years. This example confirms that.

Astronomers have tabulated a quantity called the Wolfer number for almost 300 years. This quantity measures both number and size of sunspots.

Load and plot the sunspot data:

```
load sunspot.dat
year = sunspot(:,1);
wolfer = sunspot(:,2);
plot(year,wolfer)
title('Sunspot Data')
```

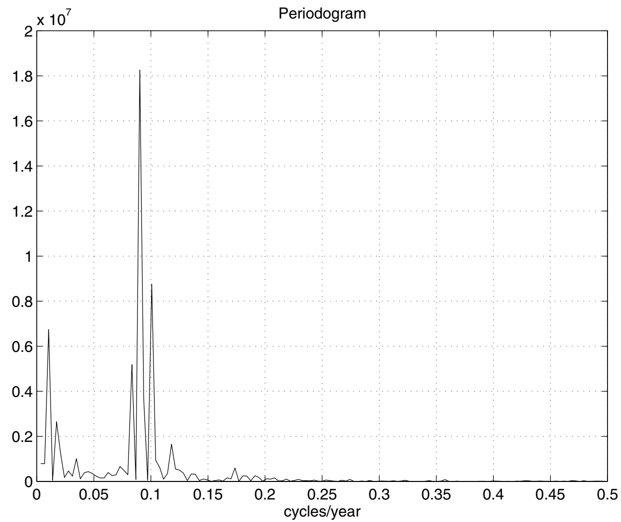


Now take the FFT of the sunspot data:

```
Y = fft(wolfer);
```

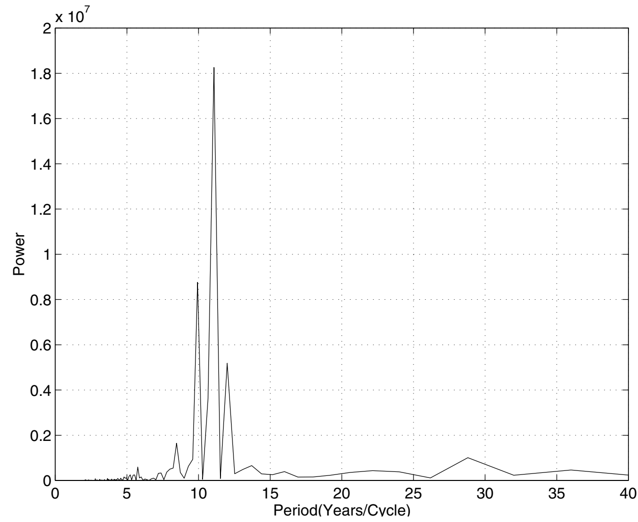
The result of this transform is the complex vector, Y. The magnitude of Y squared is called the power and a plot of power versus frequency is a "periodogram." Remove the first component of Y, which is simply the sum of the data, and plot the results:

```
N = length(Y);
Y(1) = [];
power = abs(Y(1:N/2)).^2;
nyquist = 1/2;
freq = (1:N/2)/(N/2)*nyquist;
plot(freq,power), grid on
xlabel('cycles/year')
title('Periodogram')
```



The scale in cycles/year is somewhat inconvenient. You can plot in years/cycle and estimate what one cycle is. For convenience, plot the power versus period (where `period = 1./freq`) from 0 to 40 years/cycle:

```
period = 1./freq;
plot(period,power), axis([0 40 0 2e7]), grid on
ylabel('Power')
xlabel('Period(Years/Cycle)')
```



In order to determine the cycle more precisely,

```
[mp,index] = max(power);
period(index)

ans =
      11.0769
```

Magnitude and Phase of Transformed Data

Important information about a transformed sequence includes its magnitude and phase. The MATLAB functions `abs` and `angle` calculate this information.

To try this, create a time vector `t`, and use this vector to create a sequence `x` consisting of two sinusoids at different frequencies:

```
t = 0:1/100:10-1/100;
x = sin(2*pi*15*t) + sin(2*pi*40*t);
```

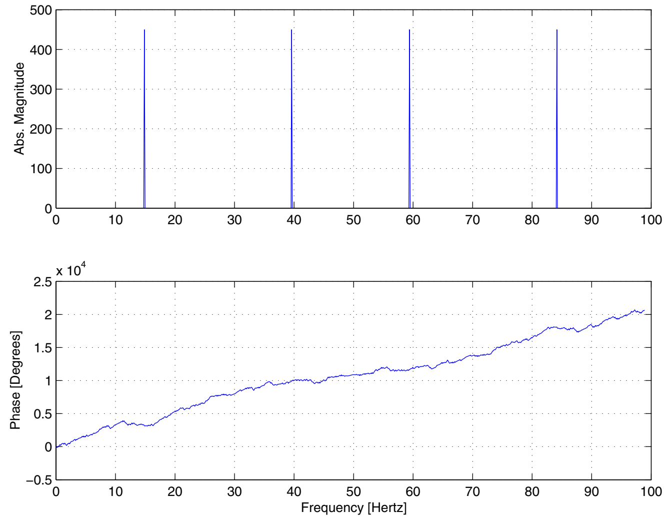
Now use the `fft` function to compute the DFT of the sequence. The code below calculates the magnitude and phase of the transformed sequence. It uses the `abs` function to obtain the magnitude of the data, the `angle` function to obtain the phase information, and `unwrap` to remove phase jumps greater than `pi` to their `2*pi` complement:

```
y = fft(x);
m = abs(y);
p = unwrap(angle(y));
```

Now create a frequency vector for the x-axis and plot the magnitude and phase:

```
f = (0:length(y)-1)'*100/length(y);
subplot(2,1,1), plot(f,m),
ylabel('Abs. Magnitude'), grid on
subplot(2,1,2), plot(f,p*180/pi)
ylabel('Phase [Degrees]'), grid on
xlabel('Frequency [Hertz]')
```

The magnitude plot is perfectly symmetrical about the Nyquist frequency of 50 hertz. The useful information in the signal is found in the range 0 to 50 hertz.


Abs. Magnitude
Phase [Degrees]
x 10^4
Frequency [Hertz]

FFT Length Versus Speed

You can add a second argument to `fft` to specify a number of points `n` for the transform:

```
y = fft(x,n)
```

With this syntax, `fft` pads `x` with zeros if it is shorter than `n`, or truncates it if it is longer than `n`. If you do not specify `n`, `fft` defaults to the length of the input sequence.

The execution time for `fft` depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors.

The inverse FFT function `ifft` also accepts a transform length argument.

For practical application of the FFT, the Signal Processing Toolbox includes numerous functions for spectral analysis.

Function Summary

MATLAB provides a collection of functions for computing and working with Fourier transforms.

FFT Function Summary

Function	Description
`fft`	Discrete Fourier transform.
`fft2`	Two-dimensional discrete Fourier transform.
`fftn`	N-dimensional discrete Fourier transform.
`ifft`	Inverse discrete Fourier transform.
`ifft2`	Two-dimensional inverse discrete Fourier transform.
`ifftn`	N-dimensional inverse discrete Fourier transform.
`abs`	Magnitude.
`angle`	Phase angle.
`unwrap`	Unwrap phase angle in radians.
`fftshift`	Move zeroth lag to center of spectrum.
`cplxpair`	Sort numbers into complex conjugate pairs.
`nextpow2`	Next higher power of two.

4

Function Functions

Function Summary (p. 4-2)	A summary of some function functions
Representing Functions in MATLAB (p. 4-3)	Some guidelines for representing functions in MATLAB
Plotting Mathematical Functions (p. 4-5)	A discussion about using `fplot` to plot mathematical functions
Minimizing Functions and Finding Zeros (p. 4-8)	A discussion of high-level function functions that perform optimization-related tasks
Numerical Integration (Quadrature) (p. 4-27)	A discussion of the MATLAB quadrature functions
Parameterizing Functions Called by Function Functions (p. 4-30)	Explains how to pass additional arguments to user-defined functions that are called by a function function.

See the “Differential Equations” and “Sparse Matrices” chapters for information about the use of other function functions.

For information about function handles, see the `function_handle` (@), `func2str`, and `str2func` reference pages, and the “Function Handles” section of “Programming and Data Types” in the MATLAB documentation.

Function Summary

Function functions are functions that call other functions as input arguments. An example of a function function is `fplot`, which plots the graphs of functions. You can call the function `fplot` with the syntax

```
fplot(@fun, [-pi pi])
```

where the input argument `@fun` is a handle to the function you want to plot. The function `fun` is referred to as the *called* function.

The function functions are located in the MATLAB `funfun` directory.

This table provides a brief description of the functions discussed in this chapter. Related functions are grouped by category.

Function Summary

Category	Function	Description
Plotting	`fplot`	Plot function
Optimization and zero finding	`fminbnd`	Minimize function of one variable with bound constraints.
	`fminsearch`	Minimize function of several variables.
	`fzero`	Find zero of function of one variable.
Numerical integration	`quad`	Numerically evaluate integral, adaptive Simpson quadrature.
	`quadl`	Numerically evaluate integral, adaptive Lobatto quadrature.
	`quadv`	Vectorized quadrature
	`dblquad`	Numerically evaluate double integral.
	`triplequad`	Numerically evaluate triple integral.

Representing Functions in MATLAB

MATLAB can represent mathematical functions by expressing them as MATLAB functions in M-files or as anonymous functions. For example, consider the function

$$f(x) = \frac{1}{(x-0.3)^2 + 0.01} + \frac{1}{(x-0.9)^2 + 0.04} - 6$$

This function can be used as input to any of the function functions.

MATLAB Functions

You can find the function above in the M-file named `humps.m`.

```
function y = humps(x)
y = 1./((x - 0.3).^2 + 0.01) + 1./((x - 0.9).^2 + 0.04) - 6;
```

To evaluate the function `humps` at 2.0, use `@` to obtain a function handle for `humps`, and then use the function handle in the same way you would use a function name to call the function:

```
fh = @humps;
fh(2.0)

ans =
   -4.8552
```

Anonymous Functions

A second way to represent a mathematical function at the command line is by creating an anonymous function from a string expression. For example, you can create an anonymous function of the `humps` function. The value returned, `fh`, is a function handle:

```
fh = @(x)1./((x-0.3).^2 + 0.01) + 1./((x-0.9).^2 + 0.04)-6;
```

You can then evaluate `fh` at `2.0` in the same way that you can with a function handle for a MATLAB function:

```
fh(2.0)
ans =
   -4.8552
```

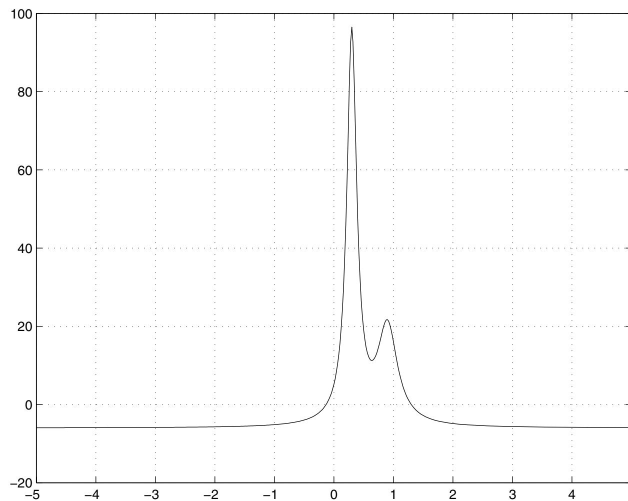
You can also create anonymous functions of more than one argument. The following function has two input arguments `x` and `y`.

```
fh = @(x,y)y*sin(x)+x*cos(y);
fh(pi,2*pi)
ans =
    3.1416
```

Plotting Mathematical Functions

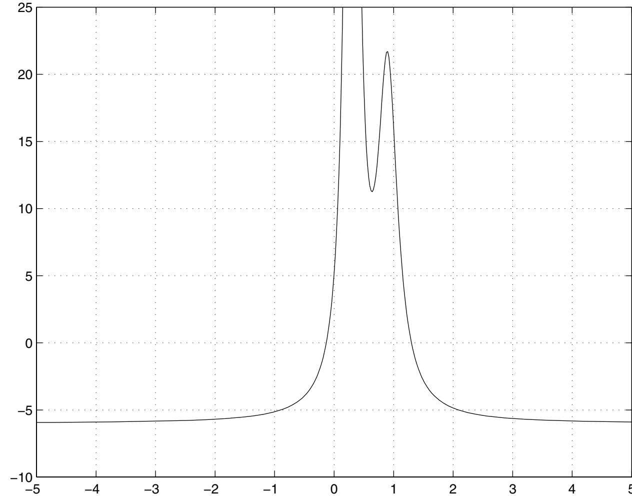
The fplot function plots a mathematical function between a given set of axes limits. You can control the *x*-axis limits only, or both the *x*- and *y*-axis limits. For example, to plot the humps function over the *x*-axis range [-5 5], use

```
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

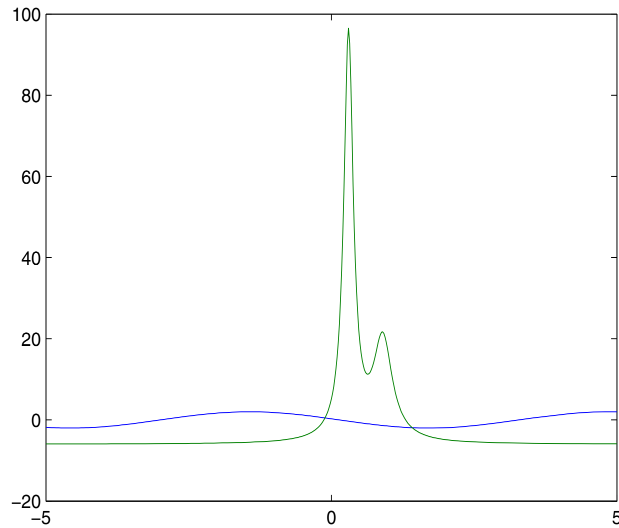
```
fplot(@(x)2*sin(x+3),[-1 1]);
```

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
```

Minimizing Functions and Finding Zeros

MATLAB provides a number of high-level function functions that perform optimization-related tasks. This section describes the following topics:

- "Minimizing Functions of One Variable" on page 4-8
- "Minimizing Functions of Several Variables" on page 4-9
- "Fitting a Curve to Data" on page 4-10
- "Setting Minimization Options" on page 4-13
- "Output Functions" on page 4-14
- "Finding Zeros of Functions" on page 4-21

The MATLAB optimization functions are:

`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.

Minimizing Functions of One Variable

Given a mathematical function of a single variable coded in an M-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` 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
```

Fitting a Curve to Data

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.

- "Creating an M-file for the Example" on page 4-10
- "Running the Example" on page 4-11
- "Plotting the Results" on page 4-12

Creating an M-file for the Example

To run the example, first create an M-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 an M-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 M-file calls the function `fminsearch`, which find 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
```

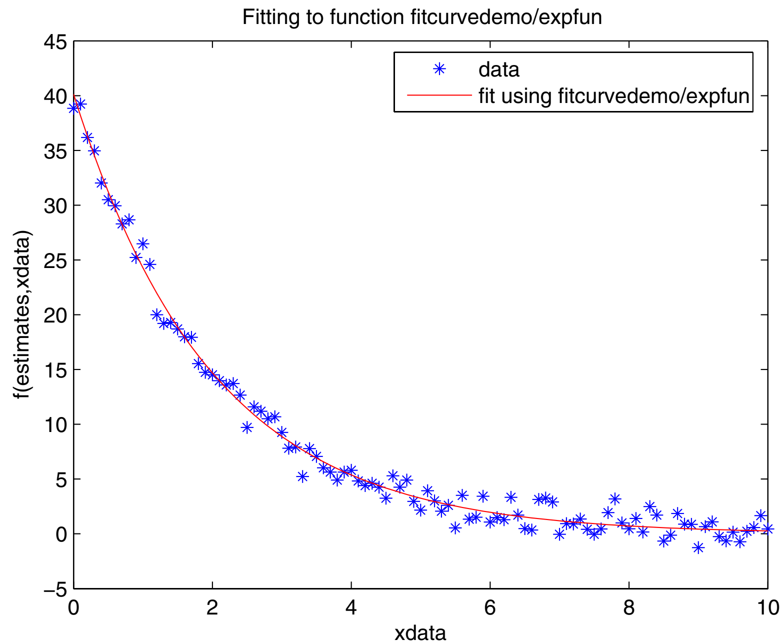
and a function handle, `model`, to the function that computes the exponential function `A*exp(-lambda*t)`.

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 Minimization Options

You can specify control options that set some minimization parameters using an options structure that you create using the function optimset. You then pass options as in 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)
```

Use optimset to set the values of the options structure. For example, to set the 'Display' option to 'iter', in order to display output from the algorithm at each iteration, enter

```
options = optimset('Display','iter');
```

fminbnd and fminsearch use only the options parameters shown in the following table.

options.Display	A flag that determines if intermediate steps in the minimization appear on the screen. If set to 'iter', intermediate steps are displayed; if set to 'off', no intermediate solutions are displayed, if set to final, displays just the final output.
options.TolX	The termination tolerance for x. Its default value is 1.e-4.
options.TolFun	The termination tolerance for the function value. The default value is 1.e-4. This parameter is used by fminsearch, but not fminbnd.
options.MaxIter	Maximum number of iterations allowed.
options.MaxFunEvals	The maximum number of function evaluations allowed. The default value is 500 for fminbnd and 200*length(x0) for fminsearch.

The number of function evaluations, the number of iterations, and the algorithm are returned in the structure `output` 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

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 an M-file function, a subfunction, or a nested function.

You can use the `OutputFcn` option with the following MATLAB optimization functions:

- `fminbnd`
- `fminsearch`
- `fzero`

This section covers the following topics:

- "Creating and Using an Output Function" on page 4-15
- "Structure of the Output Function" on page 4-16
- "Example of a Nested Output Function" on page 4-17
- "Fields in optimValues" on page 4-19
- "States of the Algorithm" on page 4-20
- "Stop Flag" on page 4-20

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

$$\underset{x}{\text{minimize}}\ f(x) = e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1)$$

To do so,

1. Create an M-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`.

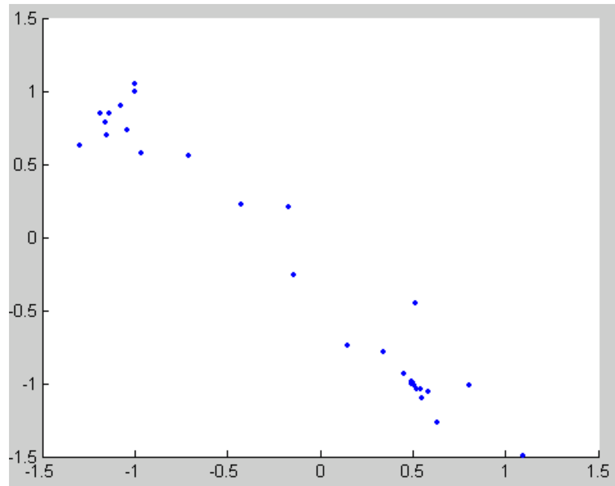
3. Enter the following commands:

   ```
   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 4-20.
- `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 4-19 describes the structure in detail.
- `state` is the current state of the algorithm. "States of the Algorithm" on page 4-20 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 4-15 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 an M-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 M-file that does the following:

- Contains the output function as a nested function — see Nested Functions in the online MATLAB documentation for more information.
- Calls the optimization function.

In the following example, the M-file also contains the objective function as a subfunction, although you could also write the objective function as a separate M-file or as an anonymous function.

Since the nested function has access to variables in the M-file function 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

$$\underset{x}{\text{minimize}}\ f(x) = e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1)$$

and returns the sequence of points as a matrix called `history`.

To run the example, do the following steps:

1. Open a new M-file in the MATLAB editor.
2. Copy and paste the following code into the M-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,:)

ans =

   -1.0000    1.0000
   -1.0000    1.0500
   -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 function `fzero` also provides additional fields that are described in its reference page.

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
`funcount`	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
```

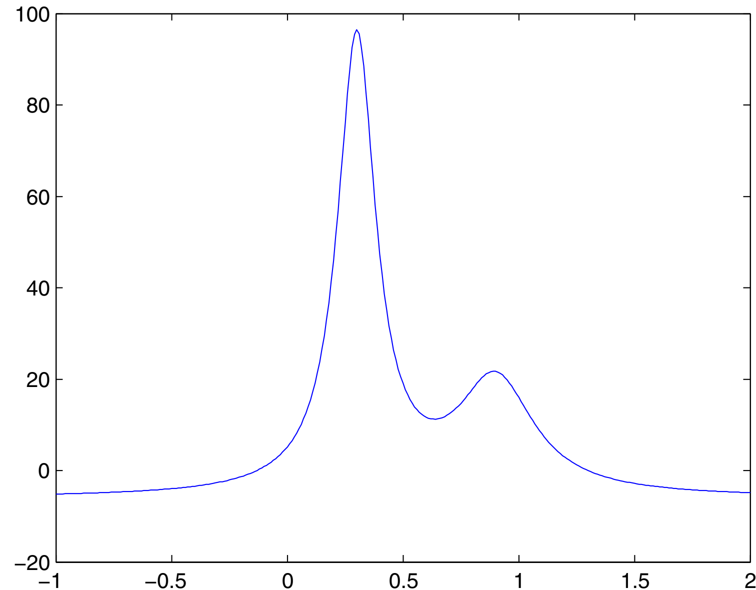
Stopping an Optimization Based on GUI Input. If you design a GUI to perform optimizations, you can make the output function stop an optimization when a user clicks a **Stop** button on the GUI. The following code shows how to do this, assuming that the **Stop** button callback stores the value true in 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');
```

Finding Zeros of Functions

The fzero function attempts to find a zero 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, 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 a 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, if you set the starting point to `-0.2`, the `Display` option to `Iter`, and call `fzero` by

```
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 are listed under 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.

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.

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 a warning when the objective function returns `NaN` or complex values.

Numerical Integration (Quadrature)

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
`quadv`	Vectorized quadrature
`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 a relative error tolerance for the integration. If a nonzero fifth argument is passed to `quad` or `quadl`, the function evaluations are traced.

Two examples illustrate use of these functions:

- Computing the length of a curve
- Double integration

Example: Computing the Length of a Curve

You can use `quad` or `quadl` to compute the length of a curve. Consider the curve parameterized by the equations

$$x(t) = \sin(2t), \qquad y(t) = \cos(t), \qquad z(t) = t$$

where $t \in [0, 3\pi]$.

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.

Parameterizing Functions Called by Function Functions

At times, you might want use a function function that calls a function with several parameters. For example, if you want to use `fzero` to find zeros of the cubic polynomial $x^3 + bx + c$ for different values of the coefficients b and c, you would like the function that computes the polynomial to accept the additional parameters b and c. When you invoke `fzero`, you must also provide values for these additional parameters to the polynomial function. This section describes two ways to do this:

- "Providing Parameter Values Using Nested Functions" on page 4-30
- "Providing Parameter Values to Anonymous Functions" on page 4-31

Providing Parameter Values Using Nested Functions

One way to provide parameters to the polynomial is to write a single M-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 an M-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
```

Providing Parameter Values to Anonymous Functions

Suppose you have already written a standalone M-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, as described in "Providing Parameter Values Using Nested Functions" on page 4-30, 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
```

"Anonymous Functions" on page 4-3 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, as described in "Providing Parameter Values Using Nested Functions" on page 4-30.

5

Differential Equations

Initial Value Problems for ODEs and DAEs (p. 5-2)	Describes the solution of ordinary differential equations (ODEs) and differential-algebraic equations (DAEs), where the solution of interest satisfies initial conditions at a given initial value of the independent variable.
Initial Value Problems for DDEs (p. 5-49)	Describes the solution of delay differential equations (DDEs) where the solution of interest is determined by a history function.
Boundary Value Problems for ODEs (p. 5-61)	Describes the solution of ODEs, where the solution of interest satisfies certain boundary conditions. The boundary conditions specify a relationship between the values of the solution at the initial and final values of the independent variable.
Partial Differential Equations (p. 5-89)	Describes the solution of initial-boundary value problems for systems of parabolic and elliptic partial differential equations (PDEs) in one spatial variable and time.
Selected Bibliography (p. 5-106)	Lists published materials that support concepts described in this chapter.

Note In function tables, commonly used functions are listed first, followed by more advanced functions. The same is true of property tables.

Initial Value Problems for ODEs and DAEs

This section describes how to use MATLAB to solve initial value problems (IVPs) of ordinary differential equations (ODEs) and differential-algebraic equations (DAEs). This section covers the following topics:

- "ODE Function Summary" on page 5-2
- "Introduction to Initial Value ODE Problems" on page 5-4
- "Solvers for Explicit and Linearly Implicit ODEs" on page 5-5
- "Examples: Solving Explicit ODE Problems" on page 5-9
- "Solver for Fully Implicit ODEs" on page 5-15
- "Example: Solving a Fully Implicit ODE Problem" on page 5-16
- "Changing ODE Integration Properties" on page 5-17
- "Examples: Applying the ODE Initial Value Problem Solvers" on page 5-18
- "Questions and Answers, and Troubleshooting" on page 5-43

ODE Function Summary

ODE Initial Value Problem 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
`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

ODE Solution 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

ODE Solvers Properties Handling

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`.

ODE Solver 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

Introduction to Initial Value ODE Problems

What Is an Ordinary Differential Equation?

The ODE solvers are designed to handle *ordinary differential equations*. An ordinary differential equation 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 $y_1, y_2, \ldots, y_n$.

Types of Problems Handled by the ODE Solvers

The ODE solvers handle the following types of first-order ODEs:

- Explicit ODEs of the form $y' = f(t, y)$
- Linearly implicit ODEs of the form $M(t, y) \cdot y' = f(t, y)$, where $M(t,y)$ is a matrix
- Fully implicit ODEs of the form $f(t, y, y') = 0$ (`ode15i` only)

Using Initial Conditions to Specify the Solution of Interest

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

$$\begin{aligned} y' &= f(t, y) \\ y(t_0) &= y_0 \end{aligned} \tag{5-1}$$

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, such as using one of the ODE solvers.

Working with Higher Order ODEs

The ODE solvers accept only first-order differential equations. However, ODEs often involve a number of dependent variables, as well as derivatives of order higher than one. To use the ODE solvers, you must rewrite such equations 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.

$$\begin{aligned} y'_1 &= y_2 \\ y'_2 &= y_3 \\ &\vdots \\ y'_n &= f(t, y_1, y_2, \ldots, y_n) \end{aligned}$$

"Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)" on page 5-9 rewrites 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.

Solvers for Explicit and Linearly Implicit ODEs

This section describes the ODE solver functions for explicit or linearly implicit ODEs, as described in "Types of Problems Handled by the ODE Solvers" on page 5-4. The solver functions implement numerical integration methods for solving initial value problems for ODEs. Beginning at the initial time with initial conditions, they step through the time interval, computing a solution at each time step. If the solution for a time step satisfies the solver's error tolerance criteria, it is a successful step. Otherwise, it is a failed attempt; the solver shrinks the step size and tries again.

This section describes:

- Solvers for nonstiff ODE problems
- Solvers for stiff ODE problems
- Basic ODE solver syntax

"Mass Matrix and DAE Properties," in the reference page for `odeset`, explains how to set options to solve more general linearly implicit problems.

The function `ode15i`, which solves implicit ODEs, is described in "Solver for Fully Implicit ODEs" on page 5-15.

Solvers for 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.

Solvers for 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 "Changing ODE Integration Properties" on page 5-17.)

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.

Basic ODE 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 the MATLAB Programming documentation 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 "Introduction to Initial Value ODE Problems" on page 5-4.

`options` Structure of optional parameters that change the default integration properties.

"Changing ODE Integration Properties" on page 5-17 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.

Note See "Evaluating the Solution at Specific Points" on page 5-72 for more information about solver syntax where a continuous solution is returned.

Examples: Solving Explicit ODE Problems

This section uses the van der Pol equation

$$y_1'' - \mu(1 - y_1^2)y'_1 + y_1 = 0$$

to describe the process for solving initial value ODE problems using the ODE solvers.

- "Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)" on page 5-9 describes each step of the process. Because the van der Pol equation is a second-order equation, the example must first rewrite it as a system of first order equations.
- "Example: The van der Pol Equation, m = 1000 (Stiff)" on page 5-12 demonstrates the solution of a stiff problem.
- "Evaluating the Solution at Specific Points" on page 5-15 tells you how to evaluate the solution at specific points.

Note See "Basic ODE Solver Syntax" on page 5-7 for more information.

Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)

This example explains and illustrates the steps you need to solve 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

$$y'_1 = y_2$$

$$y'_2 = \mu(1 - y_1^{\ 2})y_2 - y_1$$

See "Working with Higher Order ODEs" on page 5-5 for more information.

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. See "Examples: Solving Explicit ODE Problems" on page 5-9 and the ODE solver reference page for descriptions of the ODE solvers.

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]);
```

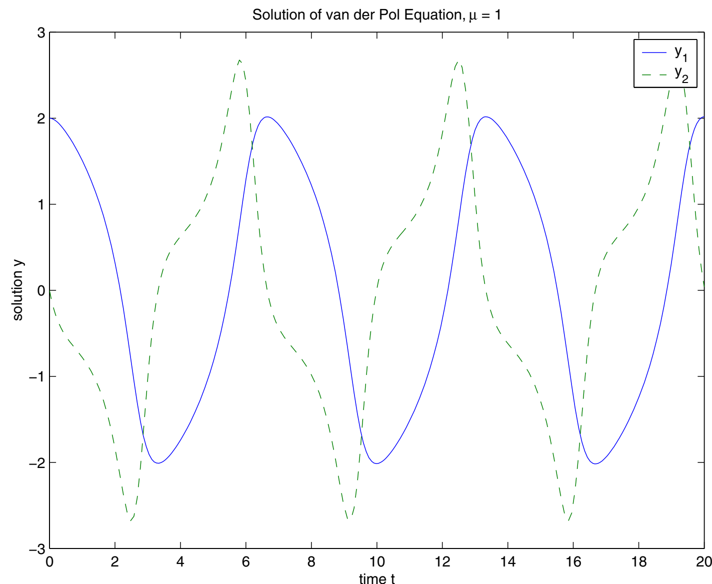
This example uses @ to pass vdp1 as a function handle to ode45. The resulting output is a column vector of time points t and a solution array y. Each row in y corresponds to a time returned in the corresponding row of t. The first column of y corresponds to y_1, and the second column to y_2.

Note For information on function handles, see the `function_handle` (@), `func2str`, and `str2func` reference pages, and the Function Handles chapter of "Programming and Data Types" in the MATLAB documentation.

---

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.

Example: The van der Pol Equation, μ = 1000 (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.

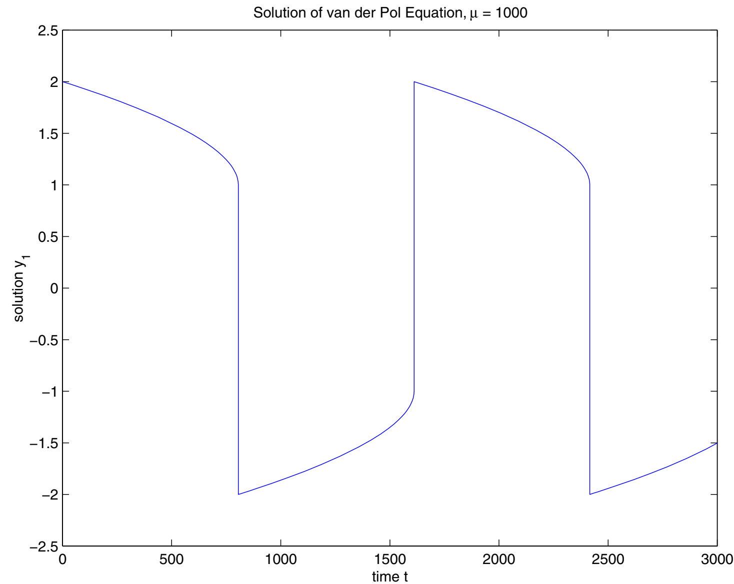
The vdp1000 function evaluates the van der Pol system from the previous example, but with μ = 1000.

```
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');
```



Note For detailed instructions for solving an initial value ODE problem, see "Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)" on page 5-9.

Parameterizing an ODE Function

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 $\mu = 1$ and $\mu = 1000$ 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 an M-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 M-file for `mu = 1`, as in “Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)” on page 5-9, enter

```
[t,y] = solve_vdp(1);
```

To run the code for μ = 1000, as in “Example: The van der Pol Equation, m = 1000 (Stiff)” on page 5-12, enter

```
[t,y] = solve_vdp(1000);
```

See the `vdpode` code for a complete example based on these functions.

Evaluating the Solution at Specific Points

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 "Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)" on page 5-9 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 $y(\mathrm{xint(i)})$.

Solver for 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`. "Example: Solving a Fully Implicit ODE Problem" on page 5-16 shows how to use the function `decic` to compute consistent initial conditions.
`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.

Note See "Evaluating the Solution at Specific Points" on page 5-72 for more information about solver syntax where a continuous solution is returned.

Example: Solving a Fully Implicit ODE Problem

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

$$y(t) = \sqrt{t^2 + 0.5}$$

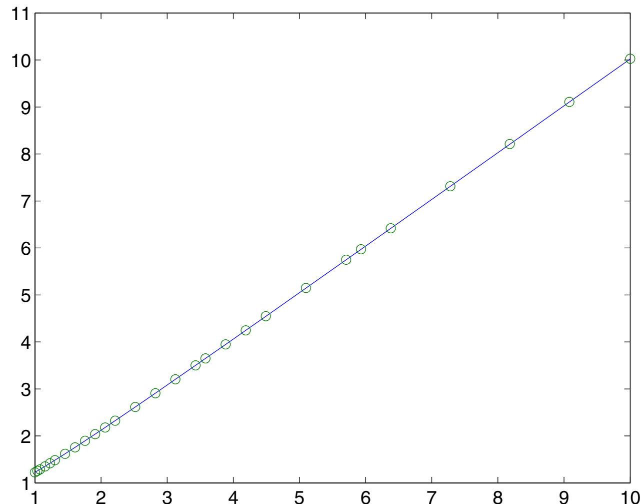
The example uses the function `weissinger`, which is provided with MATLAB, to compute the left-hand side of the equation.

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');
```



Changing ODE Integration Properties

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 an example that uses the `options` structure, see “Example: Stiff Problem (van der Pol Equation)” on page 5-20. For a complete description of the available options, see the reference page for `odeset`.

Examples: Applying the ODE Initial Value Problem Solvers

This section contains several examples that illustrate the kinds of problems you can solve. For each example, there is a corresponding M-file, included in MATLAB. You can

- View the M-file code in an editor by entering `edit` followed by the name of the M-file at the MATLAB prompt. For example, to view the code for the simple nonstiff problem example, enter

  ```
  edit rigidode
  ```

  Alternatively, if you are reading this in the MATLAB Help Browser, you can click the name of the M-file in the list below.
- Run the example by entering the name of the M-file at the MATLAB prompt.

This section presents the following examples:

- Simple nonstiff problem (`rigidode`)
- Stiff problem (`vdpode`)
- Finite element discretization (`fem1ode`)

- Large, stiff, sparse problem (brussode)
- Simple event location (ballode)
- Advanced event location (orbitode)
- Differential-algebraic problem (hb1dae)
- Computing nonnegative solutions (kneeode)
- "Summary of Code Examples" on page 5-42

Example: Simple Nonstiff Problem

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.

$$y'_1 = y_2 y_3$$

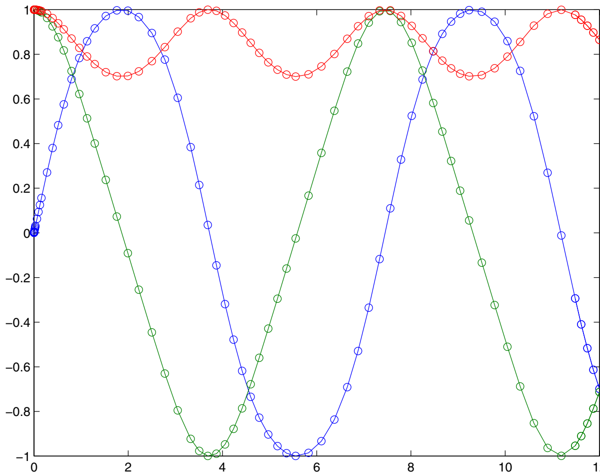
$$y'_2 = -y_1 y_3$$

$$y'_3 = -0.51 y_1 y_2$$

For your convenience, the entire problem is defined and solved in a single M-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 on the example name, or type rigidode at the command line.

```
function rigidode
%RIGIDODE  Euler equations of a 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) ];
```



Example: Stiff Problem (van der Pol Equation)

`vdpode` illustrates the solution of the van der Pol problem described in "Example: The van der Pol Equation, m = 1000 (Stiff)" on page 5-12. The differential equations

$$y_1' = y_2$$

$$y_2' = \mu(1 - y_1^2)y_2 - y_1$$

involve a constant parameter μ.

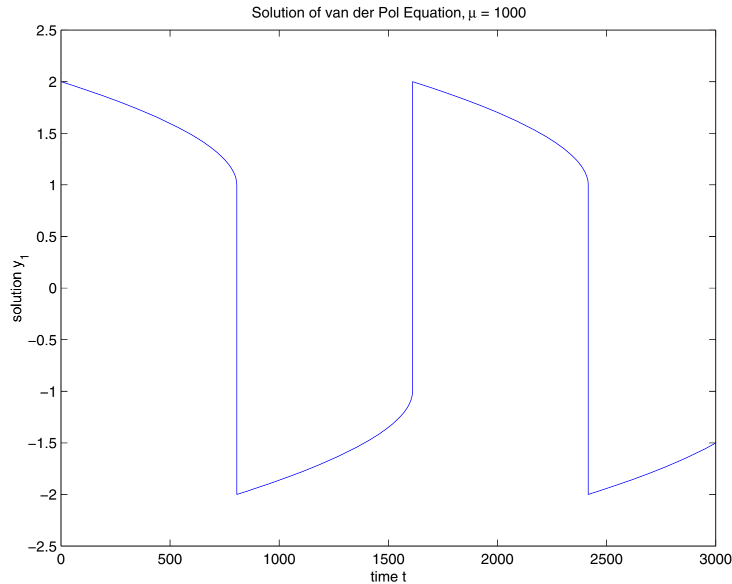
As μ increases, the problem becomes more stiff, and the period of oscillation becomes larger. When μ is `1000` the equation is in relaxation oscillation and the problem is very stiff. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff (quasi-discontinuities).

By default, the solvers in the ODE suite that are intended for stiff problems approximate Jacobian matrices numerically. However, this example provides a nested function `J(t,y)` to evaluate the Jacobian matrix $\partial f / \partial y$ analytically at

(t,y) for μ = MU. The use of an analytic Jacobian can improve the reliability and efficiency of integration.

To run this example, click on the example name, or type vdpode at the command line. From the command line, you can specify a value of μ as an argument to vdpode. The default is $\mu = 1000$.

```
function vdpode(MU)
%VDPODE  Parameterizable van der Pol equation (stiff for large MU)
if nargin < 1
  MU = 1000;     % default
end

tspan = [0; max(20,3*MU)];             % Several periods
y0 = [2; 0];
options = odeset('Jacobian',@J);

[t,y] = ode15s(@f,tspan,y0,options);

plot(t,y(:,1));
title(['Solution of van der Pol Equation, \mu = ' num2str(MU)]);
xlabel('time t');
ylabel('solution y_1');

axis([tspan(1) tspan(end) -2.5 2.5]);
---------------------------------------------------------------
function dydt = f(t,y)
dydt = [            y(2)
         MU*(1-y(1)^2)*y(2)-y(1) ];
end   % End nested function f
---------------------------------------------------------------
function dfdy = J(t,y)
dfdy = [         0                  1
         -2*MU*y(1)*y(2)-1    MU*(1-y(1)^2) ];
end   % End nested function J
end
```



Example: 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

$$u(t,x_k) \approx \sum_{k=1}^{N} c_k(t)\phi_k(x)$$

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

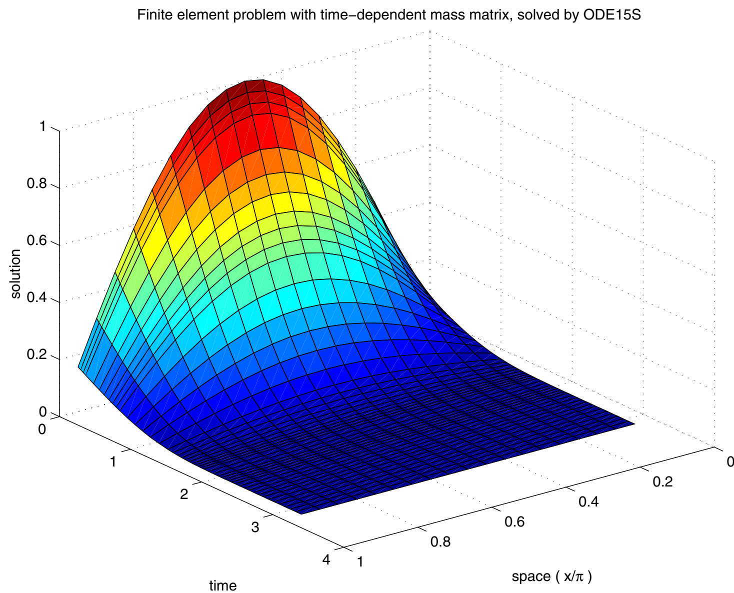
```
options = odeset('Mass',@mass,'MStateDep','none','Jacobian',J)
```

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 on 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 is 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
```



Example: 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

$$\left.\begin{array}{l} u_i(0) = 1 + \sin(2\pi x_i) \\ v_i(0) = 3 \end{array}\right\} \quad \text{with } x_i = i/(N+1), \text{ for } i = 1, ..., N$$

There are $2N$ equations in the system, but the Jacobian is banded with a constant width 5 if the equations are ordered as $u_1, v_1, u_2, v_2, \ldots$

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 1s and 0s 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.

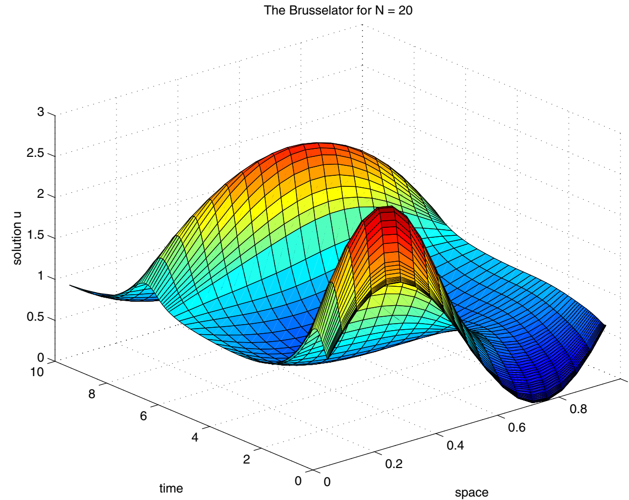
For the Brusselator problem, if the sparsity pattern is not supplied, 2N evaluations of the function are needed to compute the 2N-by-2N Jacobian matrix. If the sparsity pattern is supplied, only four evaluations are needed, regardless of the value of N.

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;
```



Example: Simple 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` (“Example: Advanced Event Location” on page 5-32).

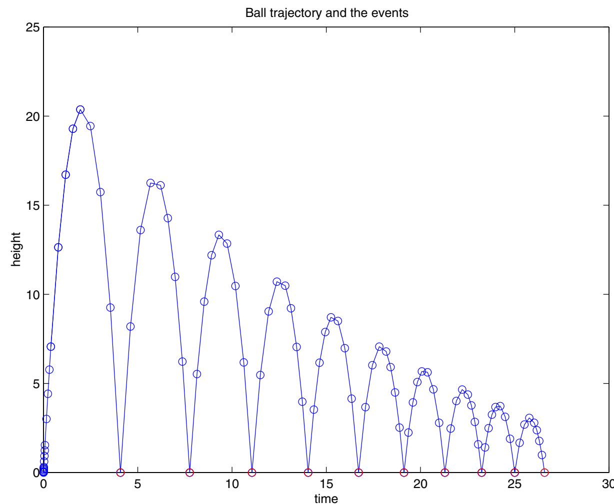
In `ballode`, setting the `Events` property to `@events` causes the solver to stop the integration (`isterminal = 1`) when the ball hits the ground (the height `y(1)` is `0`) during its fall (`direction = -1`). The example then restarts the integration with initial conditions corresponding to a ball 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
```



Example: 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 shown:

$$y'_1 = y_3$$

$$y'_2 = y_4$$

$$y'_3 = 2y_4 + y_1 - \frac{\mu^*(y_1 + \mu)}{r_1^3} - \frac{\mu(y_1 - \mu^*)}{r_2^3}$$

$$y'_4 = -2y_3 + y_2 - \frac{\mu^* y_2}{r_1^3} - \frac{\mu y_2}{r_2^3}$$

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

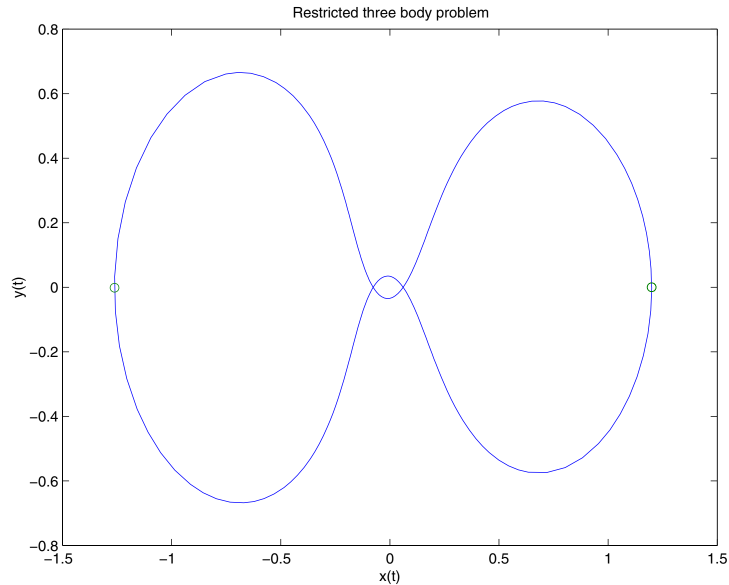
events. In this example, the ability to specify the direction of the zero crossing is critical. Both the point of return to the initial point and the point of maximum distance have the same event function value, and the direction of the crossing is used to distinguish them.

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, and 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.  We can 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
```



Example: Differential-Algebraic Problem

`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

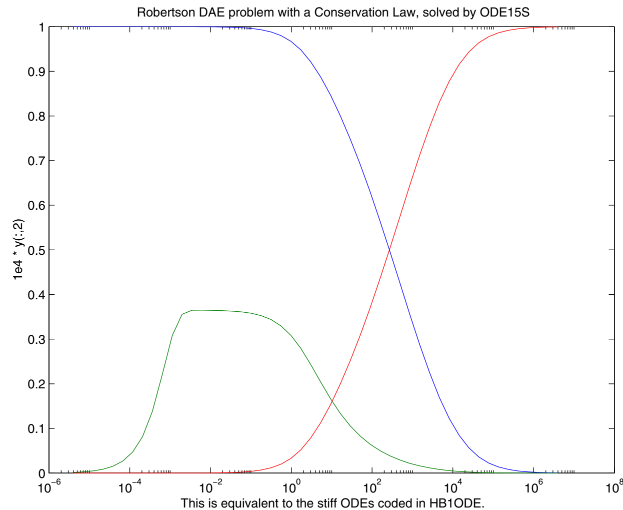
$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

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 an inconsistent initial condition to test initialization.
y0 = [1; 0; 1e-3];
tspan = [0 4*logspace(-6,6)];

% Use the LSODI example tolerances. The 'MassSingular' property
% 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 coded 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 ];
```



Example: Computing 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`, and 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.

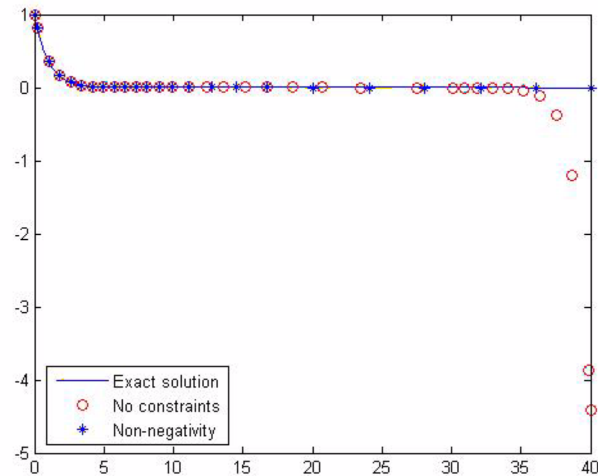
In this example, the first call to ode45 uses the defaults for the solver parameters:

```
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 MATLAB kneeode Demo. The MATLAB kneeode demo solves the “knee problem” by imposing a nonnegativity constraint on the numerical solution. The initial value problem is

```
ε*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 comprises 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')
```

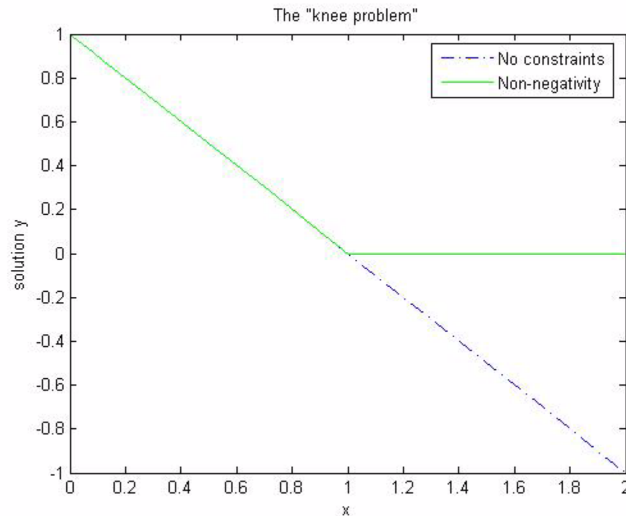
```
      function yp = odefcn(x,y)
         yp = ((1 - x)*y - y^2)/epsilon;
      end
   end  % kneeode
```

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.



Summary of Code Examples

The following table lists the M-files for all the ODE initial value problem examples. Click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the ODE examples and DAE examples. You can also run the examples from the browser. Click on these links to invoke the browser, or type `odeexamples('ode')`or `odeexamples('dae')`at the command line.

Example	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
`brussode`	Stiff large problem — diffusion in a chemical reaction (the Brusselator)
`burgersode`	ODE with strongly state-dependent mass matrix — Burger's 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

Example	Description
`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)

Questions and Answers, and Troubleshooting

This section contains a number of tables that answer questions about the use and operation of the ODE solvers:

- General ODE solver questions
- Problem size, memory use, and computation speed
- Time steps for integration
- Error tolerance and other options
- Solving different kinds of problems
- Troubleshooting

General ODE Solver 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.

Problem Size, Memory Use, and Computation Speed

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.
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.

Error Tolerance and Other Options (Continued)

Question	Answer
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.

Solving Different Kinds of Problems

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 "Example: Large, Stiff, Sparse Problem" on page 5-25. 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 5-89.
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`.

Solving Different Kinds of Problems (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`.

Troubleshooting

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 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`.

Troubleshooting (Continued)

Question	Answer
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 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.

Initial Value Problems for DDEs

This section describes how to use MATLAB to solve initial value problems (IVPs) for delay differential equations (DDEs). It provides:

- A summary of the DDE functions and examples
- An introduction to DDEs
- A description of the DDE solver and its syntax
- General instructions for representing a DDE
- A discussion and example about discontinuities and restarting
- A discussion about changing default integration properties

DDE Function Summary

DDE Initial Value Problem Solver

Solver	Description
`dde23`	Solve initial value problems for delay differential equations with constant delays.

DDE Helper Functions

Function	Description
`deval`	Evaluate the numerical solution using the output of `dde23`.

DDE Solver Properties Handling

An options structure contains named properties whose values are passed to `dde23`, and which affect problem solution. Use these functions to create, alter, or access an options structure.

Function	Description
ddeset	Create/alter the DDE options structure.
ddeget	Extract properties from options structure created with ddeset.

DDE Initial Value Problem Examples

These examples illustrate the kind of problems you can solve using dde23. Click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the DDE examples, and also run them. Click on the link to invoke the browser, or type odeexamples('dde')at the command line.

Example	Description
ddex1	Straightforward example
ddex2	Cardiovascular model with discontinuities

Additional examples are provided by "Tutorial on Solving DDEs with DDE23," available at http://www.mathworks.com/dde_tutorial.

Introduction to Initial Value DDE Problems

The DDE solver can solve systems of ordinary differential equations

$$y'(t) = f(t, y(t), y(t-\tau_1), \ldots, y(t-\tau_k))$$

where t is the independent variable, y is the dependent variable, and y' represents dy/dt. The delays (lags) $\tau_1, \ldots, \tau_k$ are positive constants.

Using a History to Specify the Solution of Interest

In an *initial value problem*, we 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.

$$S'(t_0^-) \neq y'(t_0^+) = f(t_0, y(t_0), S(t_0 - \tau_1), \ldots, S(t_0 - \tau_k))$$

A discontinuity in any derivative propagates into the future at spacings of $\tau_1, \tau_2, \ldots, \tau_k$.

For reliable and efficient integration of DDEs, a solver must track discontinuities in low order derivatives and deal with them. For DDEs with constant lags, the solution gets smoother as the integration progresses, so after a while the solver can stop tracking a discontinuity. See “Discontinuities” on page 5-57 for more information.

DDE Solver

This section describes:

- The DDE solver, `dde23`
- DDE solver basic syntax

The DDE Solver

The function `dde23` solves initial value problems for delay differential equations (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 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 Basic Syntax

The basic syntax of the DDE solver is

```
sol = dde23(ddefun,lags,history,tspan,options)
```

The input arguments are

ddefun	Handle to a function that evaluates the right side of the differential equations. The function must have the form `dydt = ddefun(t,y,Z)` where the scalar t is the independent variable, the column vector y is the dependent variable, and Z(:,j) is $y(t-\tau_j)$ for τ_j = lags(j). See "Function Handles" in the MATLAB Programming documentation for more information.
lags	A vector of constant positive delays $\tau_1, \ldots, \tau_k$.
history	Handle to a function of t that evaluates the solution $y(t)$ for $t \le t_0$. The function must be of the form `S = history(t)` where S is a column vector. Alternatively, if $y(t)$ is constant, you can specify history as this constant vector. If the current call to dde23 continues a previous integration to t0, use the solution sol from that call as the history.
tspan	The interval of integration as a two-element vector [t0,tf] with t0 < tf.

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. You can create the structure `options` using `odeset`. The `odeset` reference page describes the properties you can specify.

The output argument `sol` is a structure created by the solver. It has fields:

`sol.x`	Nodes of the mesh selected by `dde23`
`sol.y`	Approximation to $y(t)$ at the mesh points of `sol.x`
`sol.yp`	Approximation to $y'(t)$ at the mesh points of `sol.x`
`sol.solver`	`'dde23'`

To evaluate the numerical solution at any point from `[t0,tf]`, use `deval` with the output structure `sol` as its input.

Solving DDE Problems

This section uses an example to describe:

- Using `dde23` to solve initial value problems (IVPs) for delay differential equations (DDEs)
- Evaluating the solution at specific points

Example: A Straightforward Problem

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. See “DDE Solver Basic Syntax” on page 5-52 for more information.

---

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 (see “Examples: Solving Explicit ODE Problems” on page 5-9). 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`.

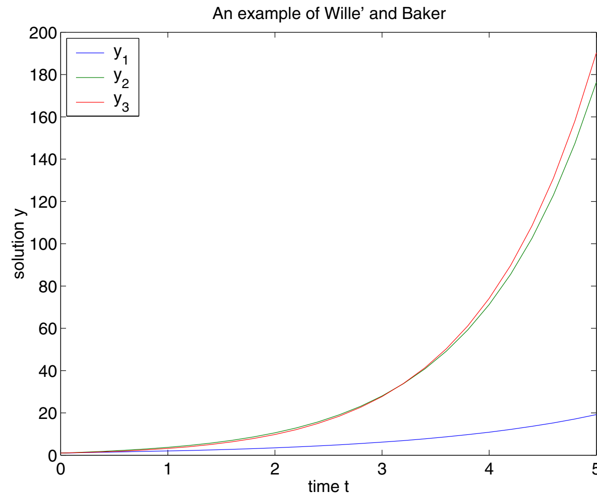
```
sol = dde23(@ddex1de,lags,@ddex1hist,[0,5]);
```

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 at Specific Points

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 ith 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);
```

Discontinuities

dde23 can solve problems with discontinuities in the history or discontinuities in coefficients of the equations. It provides properties that enable you to supply locations of known discontinuities and a different initial value.

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.
State-dependent	Events	dde23 uses the events function you supply to locate these discontinuities. When dde23 finds such a discontinuity, restart the integration to continue. Specify the solution structure for the current integration as the history for the new integration. dde23 extends each element of the solution structure after each restart so that the final structure provides the solution for the whole interval of integration. If the new problem involves a change in the solution, use the InitialY property to specify the initial value for the new integration.

Example: 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. See “DDE Solver Basic Syntax” on page 5-52 for more information.

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
.
.
.
```

Solve Using the `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`.

Solve by Restarting. The example could have solved this problem by breaking it into two pieces

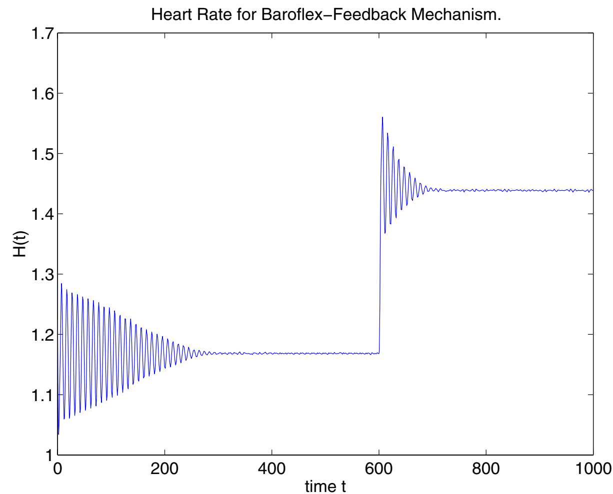
```
sol = dde23(@ddex2de,tau,history,[0, 600]);
sol = dde23(@ddex2de,tau,sol,[600, 1000]);
```

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. When you use event functions to locate such discontinuities, you must restart the integration at discontinuities.



Changing DDE Integration Properties

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`.

Boundary Value Problems for ODEs

This section describes how to use MATLAB to solve boundary value problems (BVPs) of ordinary differential equations (ODEs). It provides:

- A summary of the BVP functions and examples
- An introduction to BVPs
- A description of the BVP solver and its syntax
- A discussion about changing default integration properties
- General instructions for solving a BVP
- Examples that use continuation to solve a difficult problem
- Instructions for solving singular BVPs
- Instructions for solving multipoint BVPs

BVP Function Summary

ODE Boundary Value Problem Solver

Solver	Description
`bvp4c`	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 Properties Handling

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`.

ODE Boundary Value Problem Examples

These examples illustrate the kind of problems you can solve using the BVP solver. Click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the BVP examples, and also run them. Click on the link to invoke the browser, or type `odeexamples('bvp')` at the command line.

Example	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`.

Introduction to Boundary Value ODE 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 dy/dx .

See “What Is an Ordinary Differential Equation?” on page 5-4 for general information about ODEs.

Using Boundary Conditions to Specify the Solution of Interest

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.

Boundary Value Problem Solver

This section describes:

- The BVP solver, `bvp4c`
- BVP solver basic syntax
- BVP solver options

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. For more information, see “Finding Unknown Parameters” on page 5-72.

`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 Basic 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 the MATLAB Programming documentation for more information. `odefun` can also accept a vector of unknown parameters and a variable number of known parameters.

`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.

`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]`. For information about using `deval`, see "Evaluating the Solution at Specific Points" on page 5-56.

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.

Changing BVP Integration Properties

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`.

Note For other ways to improve solver efficiency, check “Using Continuation to Make a Good Initial Guess” on page 8-72 and the tutorial, “Solving Boundary Value Problems for Ordinary Differential Equations in MATLAB with bvp4c,” available at `http://www.mathworks.com/bvp_tutorial`.

Solving BVP Problems

This section describes:

- The process for solving boundary value problems using `bvp4c`
- Finding unknown parameters
- Evaluating the solution at specific points

Example: Mathieu’s Equation

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 λ of Mathieu's equation

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

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

$$y(0) = 1$$
$$y'(0) = 0$$
$$y'(\pi) = 0$$

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 M-file. To run this example type mat4bvp at the command line. See "BVP Solver Basic Syntax" on page 5-65 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
```

See "Finding Unknown Parameters" on page 5-72 for more information about using unknown parameters with bvp4c.

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 the “Function Handles” chapter of “Programming and Data Types” in the MATLAB documentation for information about function handles.

---

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`.

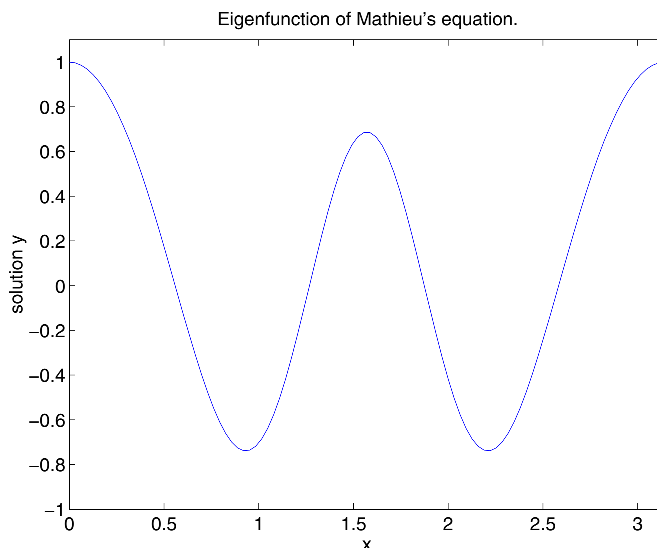
```
fprintf('The fourth eigenvalue is approximately %7.3f.\n',...
        sol.parameters)
```

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')
```

See "Evaluating the Solution at Specific Points" on page 5-72 for information about using `deval`.

The following plot shows the eigenfunction associated with the final eigenvalue $\lambda = 17.097$.



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. See "Example: Mathieu's Equation" on page 5-68.

Evaluating the Solution at Specific Points

The collocation method implemented in bvp4c produces a C^1-continuous solution over the whole interval of integration $[a, b]$. You can evaluate the approximate solution, $S(x)$, at any point in $[a, b]$ using the helper function deval and the structure sol returned by bvp4c.

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

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

Using Continuation to Make a Good Initial Guess

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 section provides examples that illustrate how to use continuation to:

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

Example: Using Continuation to Solve a Difficult 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 M-file. To run this example type `shockbvp` at the command line. See "BVP Solver Basic Syntax" on page 5-65 and "Solving BVP Problems" on page 5-68 for more information.

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.

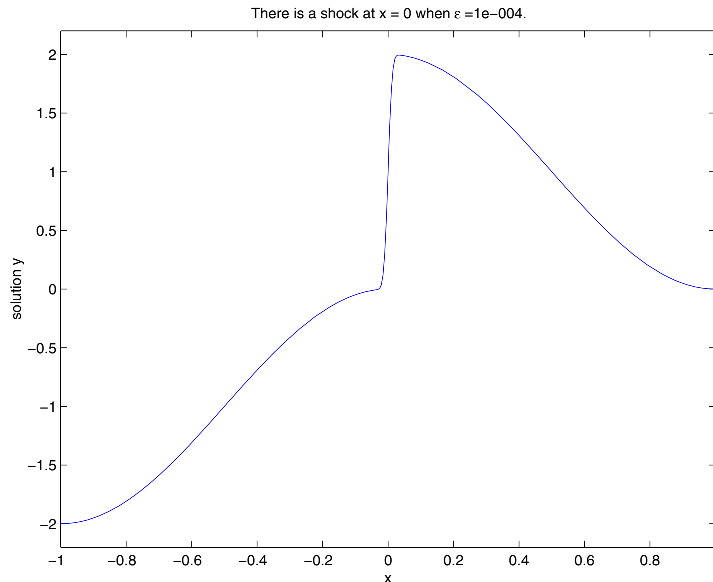
```
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
```

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')
```



Example: Using Continuation to Verify a Solution's Consistent Behavior

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 M-file. To run this example type `fsbvp` at the command line. See "BVP Solver Basic Syntax" on page 5-65 and "Solving BVP Problems" on page 5-68 for more information.

---

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]:

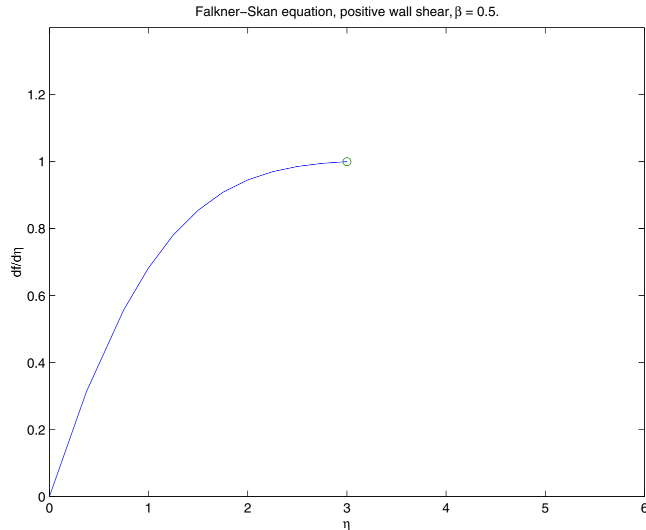
```
sol = bvp4c(@fsode,@fsbc,solinit);
eta = sol.x;
f = sol.y;

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.

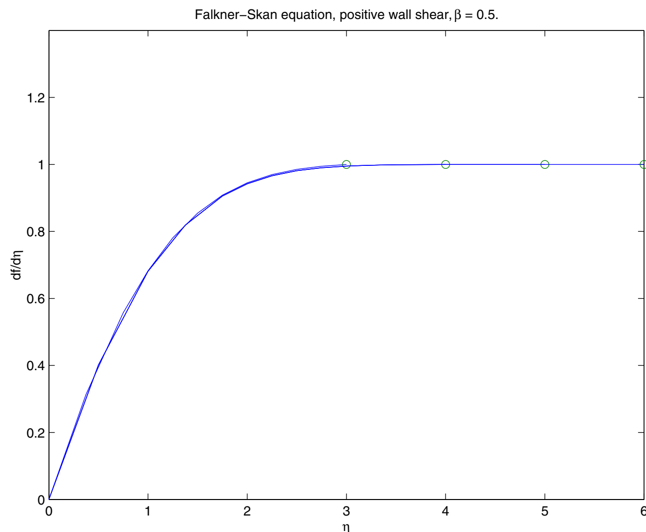
```
for Bnew = infinity+1:maxinfinity

  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.



Solving Singular BVPs

The function `bvp4c` solves a class of singular BVPs of the form

$$\begin{aligned} y' &= \frac{1}{x}Sy + f(x, y) \\ 0 &= g(y(0), y(b)) \end{aligned} \qquad \textbf{(5-2)}$$

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.

Example: Solving a BVP that Has a Singular Term

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 M-file. To run this example type `emdenbvp` at the command line. See "BVP Solver Basic Syntax" on page 5-65 and "Solving BVP Problems" on page 5-68 for more information.

---

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);
```

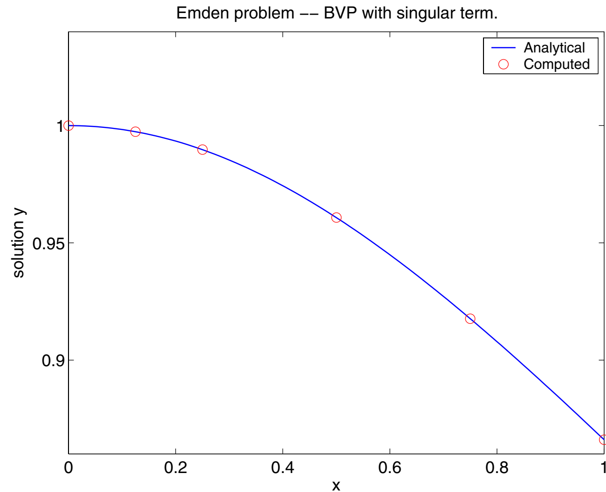
6 View the results. This problem has an analytical solution

$$y(x) = \left(1 + \frac{x^2}{3}\right)^{-1/2}$$

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');
```



Solving 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 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 M-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 kth column YL(:,k) represents the solution at the left boundary of the kth region. Similarly, YR(:,k) represents the solution at the right boundary of the kth region.

In the example, y(0) is approximated by YL(:,1), while y(λ) is approximated by YR(:,end). Continuity of the solution at the internal interface requires that YR(:,1) = YL(:,2). The residual in the boundary conditions is computed by nested function bc:

```
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

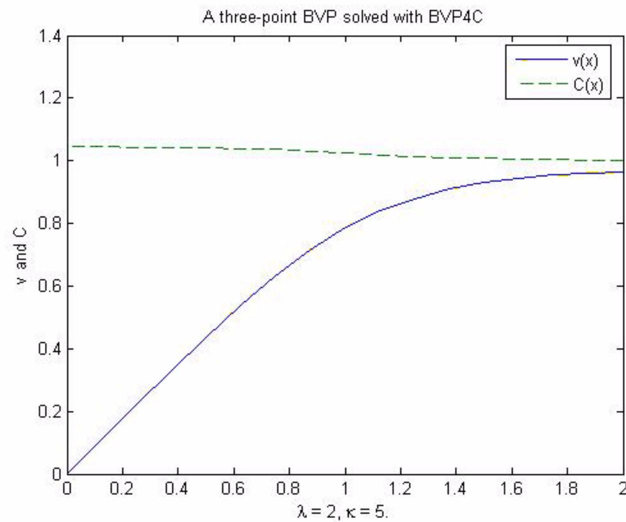
Using `deval`, the solution can be evaluated at any point in the interval of integration.

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(['\λ = ',num2str(λ), ...
        ', \κ = ',num2str(κ),'.'])
ylabel('v and C')
```



Partial Differential Equations

This section describes how to use MATLAB to solve initial-boundary value problems for partial differential equations (PDEs). It provides:

- A summary of the MATLAB PDE functions and examples
- An introduction to PDEs
- A description of the PDE solver and its syntax
- General instructions for representing a PDE in MATLAB, including an example
- Instructions on evaluating the solution at specific points
- A discussion about changing default integration properties
- An example of solving a real-life problem

PDE Function Summary

MATLAB PDE Solver

This is the MATLAB PDE solver.

PDE Initial-Boundary Value Problem Solver	
`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	
`pdeval`	Evaluate the numerical solution of a PDE using the output of `pdepe`.

PDE Examples

These examples illustrate some problems you can solve using the MATLAB PDE solver. Click the example name to see the code in an editor. Type the example name at the command line to run it.

Note The Differential Equations Examples browser enables you to view the code for the PDE examples, and also run them. Click on the link to invoke the browser, or type `odeexamples('pde')` at the command line.

Example	Description
`pdex1`	Simple PDE that illustrates the straightforward formulation, computation, and plotting of the solution
`pdex2`	Problem that involves discontinuities
`pdex3`	Problem that requires computing values of the partial derivative
`pdex4`	System of two PDEs whose solution has boundary layers at both ends of the interval and changes rapidly for small t
`pdex5`	System of PDEs with step functions as initial conditions

Introduction to PDE Problems

`pdepe` solves systems of parabolic and elliptic PDEs in one spatial variable x and time t, of the form

$$c\left(x, t, u, \frac{\partial u}{\partial x}\right)\frac{\partial u}{\partial t} = x^{-m}\frac{\partial}{\partial x}\left(x^m f\left(x, t, u, \frac{\partial u}{\partial x}\right)\right) + s\left(x, t, u, \frac{\partial u}{\partial x}\right) \quad \textbf{(5-3)}$$

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 5-3, $f(x, t, u, \partial u/\partial x)$ 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

$$u(x, t_0) = u_0(x) \tag{5-4}$$

At the boundary $x = a$ or $x = b$, for all t the solution components satisfy a boundary condition of the form

$$p(x, t, u) + q(x, t)f\left(x, t, u, \frac{\partial u}{\partial x}\right) = 0 \tag{5-5}$$

$q(x, t)$ 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 flux f rather than $\partial u / \partial x$. Also, of the two coefficients, only p can depend on u.

MATLAB Partial Differential Equation Solver

This section describes:

- The PDE solver, `pdepe`
- PDE solver basic syntax
- Additional PDE solver arguments

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 nodes specified by the user. 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 5-3 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 Basic Syntax

The basic syntax of the solver is

```
sol = pdepe(m,pdefun,icfun,bcfun,xmesh,tspan)
```

Note Correspondences given are to terms used in "Introduction to PDE Problems" on page 5-90.

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 5-3.

`pdefun` Function that defines the components of the PDE. It computes the terms c, f, and s in Equation 5-3, 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 flux 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)`.

Additional PDE Solver Arguments

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 "Changing PDE Integration Properties" on page 5-100 for more information.

Solving PDE Problems

This section describes:

- The process for solving PDE problems using the MATLAB solver, `pdepe`
- Evaluating the solution at specific points

Example: A Single PDE

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 $t \ge 0$. 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 M-file. To run the demo type `pdex1` at the command line. See "PDE Solver Basic Syntax" on page 5-92 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 5-3 and expected by `pdepe`. See "Introduction to PDE Problems" on page 5-90 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 5-3) 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 5-5, 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 “MATLAB Partial Differential Equation Solver” on page 5-91). In particular, the cost and the accuracy of the solution depend strongly on the length of the vector `x`. However, the computation is much less sensitive to the values in the vector `t`.

This example requests the solution on the mesh produced by 20 equally spaced points from the spatial interval [0,1] and five values of `t` from the time interval [0,2].

```
x = linspace(0,1,20);
t = linspace(0,2,5);
```

6 **Apply the PDE solver.** The example calls `pdepe` with `m = 0`, the functions `pdex1pde`, `pdex1ic`, and `pdex1bc`, and the mesh defined by `x` and `t` at which `pdepe` is to evaluate the solution. The `pdepe` function returns the numerical solution in a three-dimensional array `sol`, where `sol(i,j,k)` approximates the `k`th component of the solution, u_k, evaluated at `t(i)` and `x(j)`.

```
m = 0;
sol = pdepe(m,@pdex1pde,@pdex1ic,@pdex1bc,x,t);
```

This example uses `@` to pass `pdex1pde`, `pdex1ic`, and `pdex1bc` as function handles to `pdepe`.

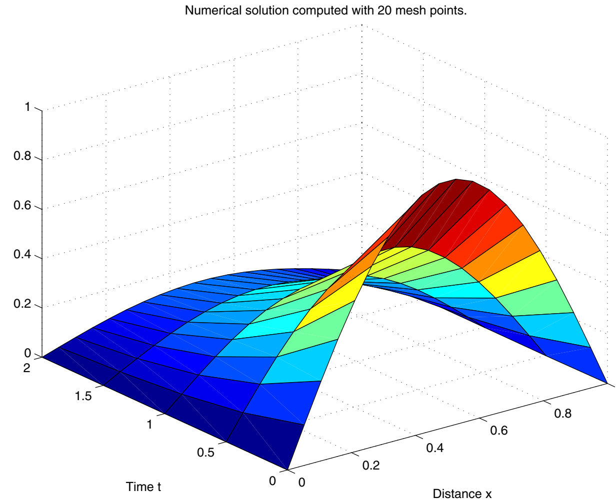
Note See the `function_handle` (@), `func2str`, and `str2func` reference pages, and the "Function Handles" chapter of "Programming and Data Types" in the MATLAB documentation for information about function handles.

---

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.

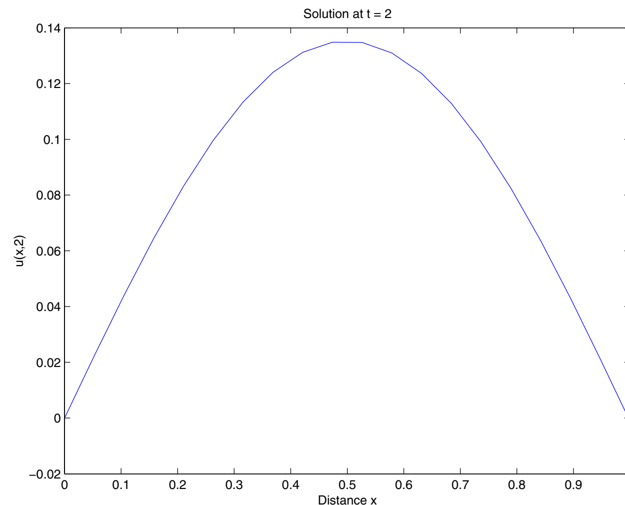
```
u = sol(:,:,1);

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 =

c 2. See "Evaluating the Solution at Specific Points" on page 5-99 for more information.

```
figure
plot(x,u(end,:))
title('Solution at t = 2')
xlabel('Distance x')
ylabel('u(x,2)')
```



Evaluating the Solution at Specific Points

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.

Changing PDE Integration Properties

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. "Changing ODE Integration Properties" on page 5-17 tells you how to create the structure and describes the properties.

PDE Property Categories

Properties Category	Property Name
Error control	RelTol, AbsTol, NormControl
Step-size	InitialStep, MaxStep

Example: Electrodynamics Problem

This example illustrates the solution of a system of partial differential equations. The problem is taken from electrodynamics. It has boundary layers at both ends of the interval, and the solution changes rapidly for small t.

The PDEs are

$$\frac{\partial u_1}{\partial t} = 0.024\frac{\partial^2 u_1}{\partial x^2} - F(u_1 - u_2)$$

$$\frac{\partial u_2}{\partial t} = 0.170\frac{\partial^2 u_2}{\partial x^2} + F(u_1 - u_2)$$

where $F(y) = \exp(5.73y) - \exp(-11.46y)$. The equations hold on an interval $0 \le x \le 1$ for times $t \ge 0$.

The solution u satisfies the initial conditions

$$u_1(x, 0) \equiv 1$$
$$u_2(x, 0) \equiv 0$$

and boundary conditions

$$\frac{\partial u_1}{\partial x}(0, t) \equiv 0$$
$$u_2(0, t) \equiv 0$$
$$u_1(1, t) \equiv 1$$
$$\frac{\partial u_2}{\partial x}(1, t) \equiv 0$$

Note The demo `pdex4` contains the complete code for this example. The demo uses subfunctions to place all required functions in a single M-file. To run this example type `pdex4` at the command line. See "PDE Solver Basic Syntax" on page 5-92 and "Solving PDE Problems" on page 5-94 for more information.

---

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 5-3.

```
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 5-5) 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.

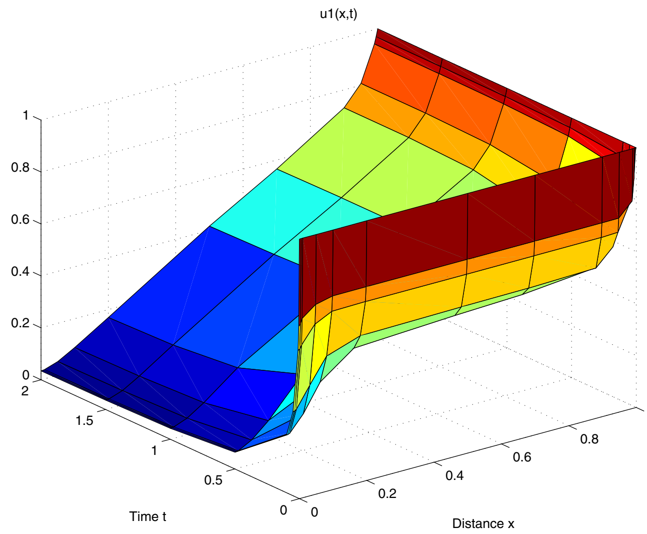
```
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];
```

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 `k`th component of the solution, u_k , evaluated at `t(i)` and `x(j)`.

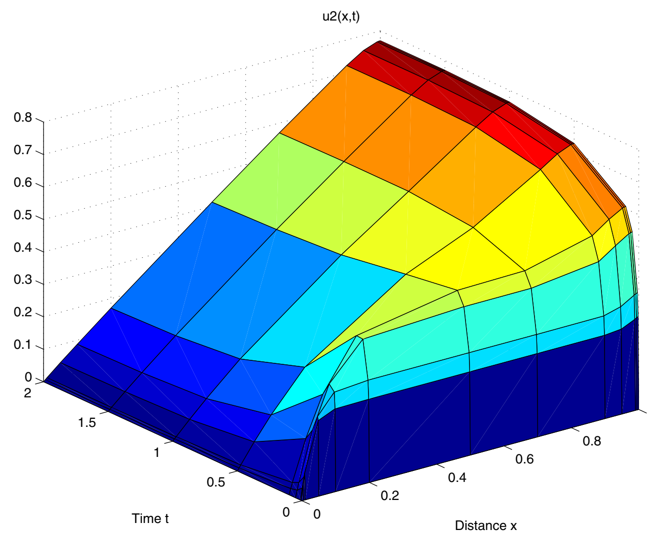
```
m = 0;
sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
```

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')
```


u2(x,t)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
2
1.5
1
0.5
0
Time t
0
0.2
0.4
0.6
0.8
1
Distance x

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.

Sparse Matrices

Function Summary (p. 6-2)	A summary of the sparse matrix functions
Introduction (p. 6-5)	An introduction to sparse matrices in MATLAB
Viewing Sparse Matrices (p. 6-13)	How to obtain quantitative and graphical information about sparse matrices
Adjacency Matrices and Graphs (p. 6-17)	Using adjacency matrices to illustrate sparse matrices
Sparse Matrix Operations (p. 6-25)	A discussion of functions that perform operations specific to sparse matrices
Selected Bibliography (p. 6-44)	Published materials that support concepts described in this chapter

Function Summary

The sparse matrix functions are located in the MATLAB `sparfun` directory.

Function Summary

Category	Function	Description
Elementary sparse matrices	`speye`	Sparse identity matrix.
	`sprand`	Sparse uniformly distributed random matrix.
	`sprandn`	Sparse normally distributed random matrix.
	`sprandsym`	Sparse random symmetric matrix.
	`spdiags`	Sparse matrix formed from diagonals.
Full to sparse conversion	`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	`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.

Function Summary (Continued)

Category	Function	Description
Graph theory	`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	`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	`eigs`	A few eigenvalues.
	`svds`	A few singular values.
	`luinc`	Incomplete LU factorization.
	`cholinc`	Incomplete Cholesky factorization.
	`normest`	Estimate the matrix 2-norm.
	`condest`	1-norm condition number estimate.
	`sprank`	Structural rank.
Linear equations (iterative methods)	`bicg`	BiConjugate Gradients Method.
	`bicgstab`	BiConjugate Gradients Stabilized Method.
	`cgs`	Conjugate Gradients Squared Method.

Function Summary (Continued)

Category	Function	Description
	`gmres`	Generalized Minimum Residual Method.
	`lsqr`	LSQR implementation of Conjugate Gradients on the Normal Equations.
	`minres`	Minimum Residual Method.
	`pcg`	Preconditioned Conjugate Gradients Method.
	`qmr`	Quasi-Minimal Residual Method.
	`symmlq`	Symmetric LQ method
Miscellaneous	`spaugment`	Form least squares augmented system.
	`spparms`	Set parameters for sparse matrix routines.
	`symbfact`	Symbolic factorization analysis.

Introduction

Sparse matrices are a special class of matrices that contain a significant number of zero-valued elements. This property allows MATLAB to:

- Store only the nonzero elements of the matrix, together with their indices.
- Reduce computation time by eliminating operations on zero elements.

This section provides information about:

- Sparse matrix storage
- General storage information
- Creating sparse matrices
- Importing sparse matrices

Sparse Matrix Storage

For full matrices, MATLAB stores internally every matrix element. 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.

MATLAB uses a compressed column, or Harwell-Boeing, format for storing matrices. This method uses three arrays internally to store sparse matrices with real elements. Consider an `m`-by-`n` sparse matrix with `nnz` nonzero entries stored in arrays of length `nzmax`:

- The first array contains all the nonzero elements of the array in floating-point format. The length of this array is equal to `nzmax`.
- The second array contains the corresponding integer row indices for the nonzero elements stored in the first `nnz` entries. This array also has length equal to `nzmax`.
- The third array contains `n` integer pointers to the start of each column in the other arrays and an additional pointer that marks the end of those arrays. The length of the third array is `n+1`.

This matrix requires storage for nzmax floating-point numbers and nzmax+n+1 integers. At 8 bytes per floating-point number and 4 bytes per integer, the total number of bytes required to store a sparse matrix is

```
8*nzmax + 4*(nzmax+n+1)
```

Note that the storage requirement depends upon nzmax and the number of columns, n. The memory required to store a sparse matrix containing a large number of rows but having few columns is much less that the memory required to store the transpose of this matrix:

```
S1 = spalloc(2^20,2,1);
S2 = spalloc(2,2^20,1);

whos
  Name      Size                   Bytes  Class

  S1        1048576x2                 24  double array (sparse)
  S2        2x1048576            4194320  double array (sparse)

Grand total is 2 elements using 4194344 bytes
```

Sparse matrices with complex elements are also possible. In this case, MATLAB uses a fourth array with nnz floating-point elements to store the imaginary parts of the nonzero elements. An element is considered nonzero if either its real or imaginary part is nonzero.

General Storage Information

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

  M_full      1100x1100          9680000  double array
  M_sparse    1100x1100             5004  double array (sparse)


Grand total is 1210050 elements using 9685004 bytes
```

Notice that the number of bytes used is much less in the sparse case, because zero-valued elements are not stored.

Creating Sparse Matrices

MATLAB never creates sparse matrices automatically. Instead, you must determine if a matrix contains a large enough percentage of zeros to benefit from sparse techniques.

The *density* of a matrix is the number of non-zero 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.

Example: Generating a Second Difference Operator

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 from Outside MATLAB

You can import sparse matrices from computations outside MATLAB. 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.

Viewing Sparse Matrices

MATLAB provides a number of functions that let you get quantitative or graphical information about sparse matrices.

This section provides information about:

- Obtaining information about nonzero elements
- Viewing graphs of sparse matrices
- Finding indices and values of nonzero elements

Information About 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

  west0479     479x479        24576  sparse array
```

This matrix models an eight-stage chemical distillation column.

Try these commands.

```
nnz(west0479)

ans =
1887

format short e
```

```
west0479

west0479 =

   (25,1)      1.0000e+00
   (31,1)     -3.7648e-02
   (87,1)     -3.4424e-01
   (26,2)      1.0000e+00
   (31,2)     -2.4523e-02
   (88,2)     -3.7371e-01
   (27,3)      1.0000e+00
   (31,3)     -3.6613e-02
   (89,3)     -8.3694e-01
   (28,4)      1.3000e+02
      .
      .
      .

nonzeros(west0479);
     ans =

    1.0000e+00
   -3.7648e-02
   -3.4424e-01
    1.0000e+00
   -2.4523e-02
   -3.7371e-01
    1.0000e+00
   -3.6613e-02
   -8.3694e-01
    1.3000e+02
      .
      .
      .
```

Note Use **Ctrl+C** to stop the `nonzeros` listing at any time.

Note that initially nnz has the same value as nzmax by default. That is, the number of nonzero elements is equivalent to the number of storage locations allocated for nonzeros. However, MATLAB does not dynamically release memory if you zero out additional array elements. Changing the value of some matrix elements to zero changes the value of nnz, but not that of nzmax.

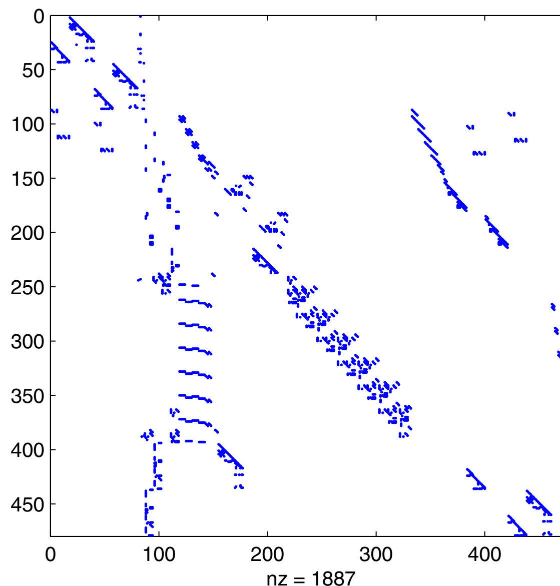
However, you can add as many nonzero elements to the matrix as desired. You are not constrained by the original value of nzmax.

Viewing Sparse Matrices Graphically

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)
```



The find Function and Sparse Matrices

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)
```

Adjacency Matrices and Graphs

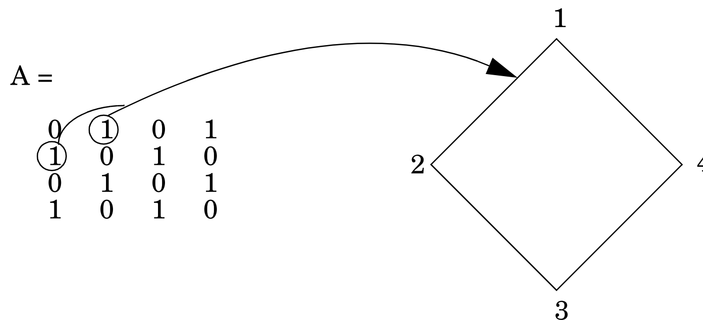
This section includes:

- An introduction to adjacency matrices
- Instructions for graphing adjacency matrices with `gplot`
- A Bucky ball example, including information about using spy plots to illustrate fill-in and distance
- An airflow model example

Introduction to Adjacency Matrices

The formal mathematical definition of a *graph* is a set of points, or nodes, with specified connections between them. An economic model, for example, is a graph with different industries as the nodes and direct economic ties as the connections. The computer software industry is connected to the computer hardware industry, which, in turn, is connected to the semiconductor industry, and so on.

This definition of a graph lends itself to matrix representation. The *adjacency matrix* of an *undirected* graph is a matrix whose `(i,j)`th and `(j,i)`th entries are 1 if node `i` is connected to node `j`, and 0 otherwise. For example, the adjacency matrix for a diamond-shaped graph looks like



Since most graphs have relatively few connections per node, most adjacency matrices are sparse. The actual locations of the nonzero elements depend on how the nodes are numbered. A change in the numbering leads to permutation

of the rows and columns of the adjacency matrix, which can have a significant effect on both the time and storage requirements for sparse matrix computations.

Graphing Using Adjacency Matrices

The MATLAB `gplot` function creates a graph based on an adjacency matrix and a related array of coordinates. To try `gplot`, create the adjacency matrix shown above by entering

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

The columns of `gplot`'s coordinate array contain the Cartesian coordinates for the corresponding node. For the diamond example, create the array by entering

```
xy = [1 3; 2 1; 3 3; 2 5];
```

This places the first node at location `(1,3)`, the second at location `(2,1)`, the third at location `(3,3)`, and the fourth at location `(2,5)`. To view the resulting graph, enter

```
gplot(A,xy)
```

The Bucky Ball

One interesting construction for graph analysis is the *Bucky ball*. This is composed of 60 points distributed on the surface of a sphere in such a way that the distance from any point to its nearest neighbors is the same for all the points. Each point has exactly three neighbors. The Bucky ball models four different physical objects:

- The geodesic dome popularized by Buckminster Fuller
- The C_{60} molecule, a form of pure carbon with 60 atoms in a nearly spherical configuration
- In geometry, the truncated icosahedron
- In sports, the seams in a soccer ball

The Bucky ball adjacency matrix is a 60-by-60 symmetric matrix `B`. `B` has three nonzero elements in each row and column, for a total of 180 nonzero values. This matrix has important applications related to the physical objects listed earlier. For example, the eigenvalues of `B` are involved in studying the chemical properties of C_{60}.

To obtain the Bucky ball adjacency matrix, enter

```
B = bucky;
```

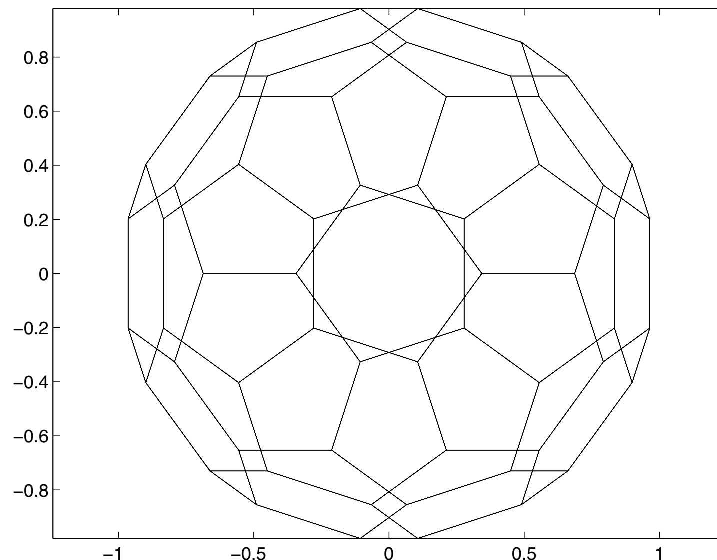
At order 60, and with a density of 5%, this matrix does not require sparse techniques, but it does provide an interesting example.

You can also obtain the coordinates of the Bucky ball graph using

```
[B,v] = bucky;
```

This statement generates `v`, a list of *xyz*-coordinates of the 60 points in 3-space equidistributed on the unit sphere. The function `gplot` uses these points to plot the Bucky ball graph.

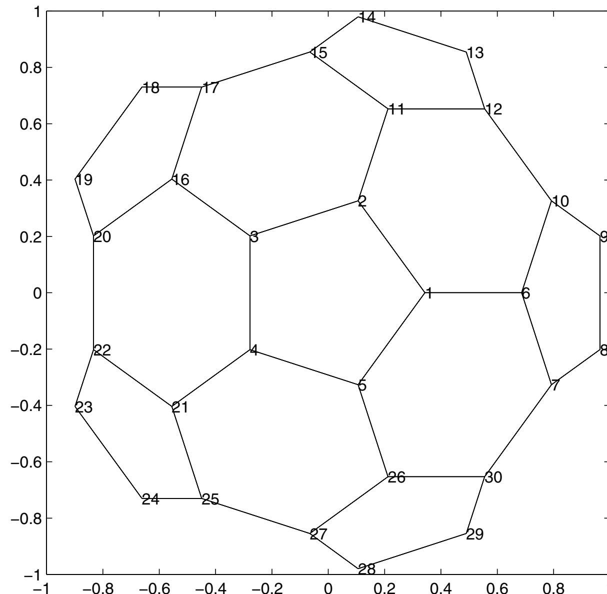
```
gplot(B,v)
axis equal
```



It is not obvious how to number the nodes in the Bucky ball so that the resulting adjacency matrix reflects the spherical and combinatorial symmetries of the graph. The numbering used by `bucky.m` is based on the pentagons inherent in the ball's structure.

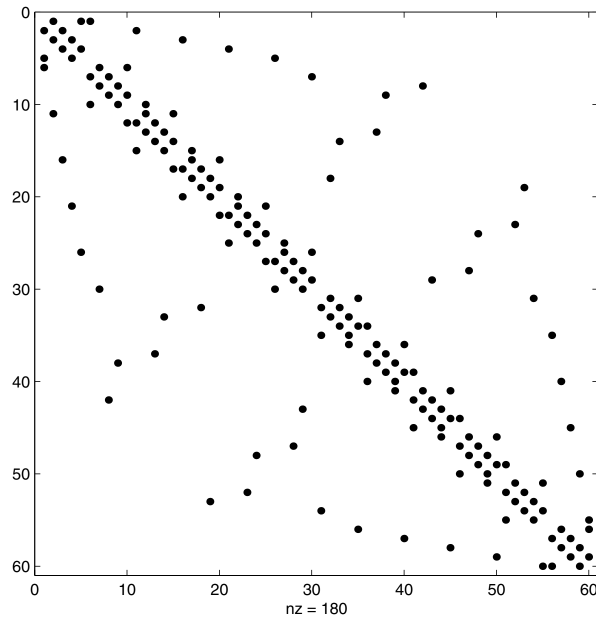
The vertices of one pentagon are numbered 1 through 5, the vertices of an adjacent pentagon are numbered 6 through 10, and so on. The picture on the following page shows the numbering of half of the nodes (one hemisphere); the numbering of the other hemisphere is obtained by a reflection about the equator. Use `gplot` to produce a graph showing half the nodes. You can add the node numbers using a `for` loop.

```
k = 1:30;
gplot(B(k,k),v);
axis square
for j = 1:30, text(v(j,1),v(j,2), int2str(j)); end
```



To view a template of the nonzero locations in the Bucky ball's adjacency matrix, use the `spy` function:

```
spy(B)
```

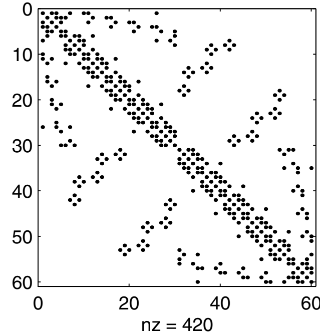


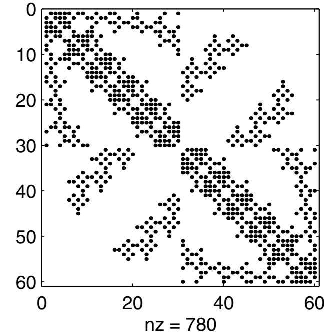
The node numbering that this model uses generates a spy plot with 12 groups of five elements, corresponding to the 12 pentagons in the structure. Each node is connected to two other nodes within its pentagon and one node in some other pentagon. Since the nodes within each pentagon have consecutive numbers, most of the elements in the first super- and sub-diagonals of `B` are nonzero. In addition, the symmetry of the numbering about the equator is apparent in the symmetry of the spy plot about the antidiagonal.

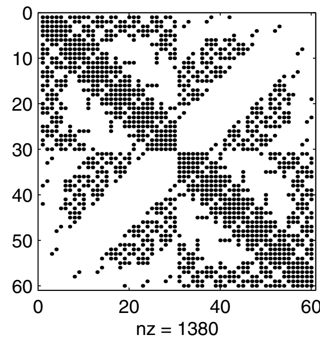
Graphs and Characteristics of Sparse Matrices

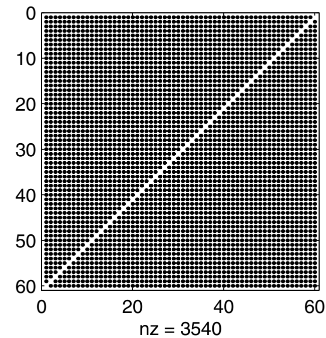
Spy plots of the matrix powers of `B` illustrate two important concepts related to sparse matrix operations, fill-in and distance. `spy` plots help illustrate these concepts.

```
spy(B^2)
spy(B^3)
spy(B^4)
spy(B^8)
```







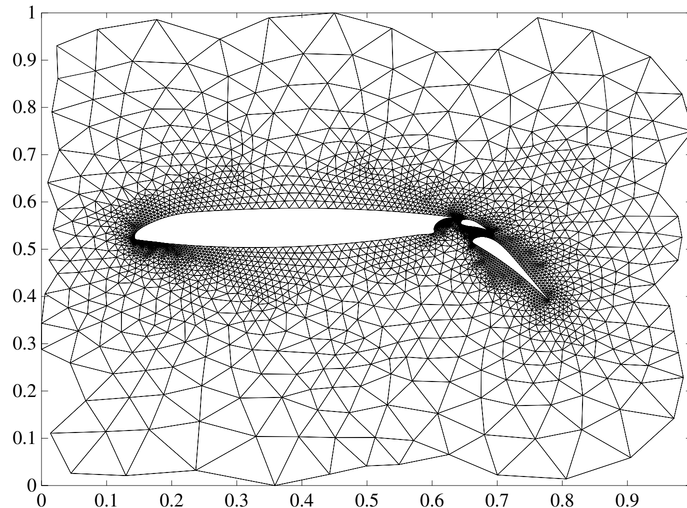


Fill-in is generated by operations like matrix multiplication. The product of two or more matrices usually has more nonzero entries than the individual terms, and so requires more storage. As `p` increases, `B^p` fills in and `spy(B^p)` gets more dense.

The *distance* between two nodes in a graph is the number of steps on the graph necessary to get from one node to the other. The spy plot of the `p`-th power of `B` shows the nodes that are a distance `p` apart. As `p` increases, it is possible to get to more and more nodes in `p` steps. For the Bucky ball, `B^8` is almost completely full. Only the antidiagonal is zero, indicating that it is possible to get from any node to any other node, except the one directly opposite it on the sphere, in eight steps.

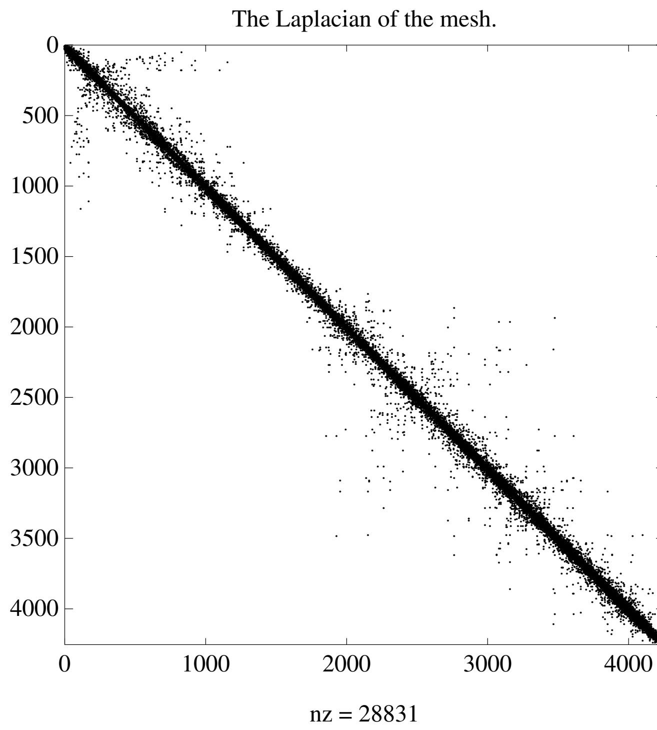
An Airflow Model

A calculation performed at NASA's Research Institute for Applications of Computer Science involves modeling the flow over an airplane wing with two trailing flaps.



In a two-dimensional model, a triangular grid surrounds a cross section of the wing and flaps. The partial differential equations are nonlinear and involve several unknowns, including hydrodynamic pressure and two components of velocity. Each step of the nonlinear iteration requires the solution of a sparse linear system of equations. Since both the connectivity and the geometric location of the grid points are known, the `gplot` function can produce the graph shown above.

In this example, there are 4253 grid points, each of which is connected to between 3 and 9 others, for a total of 28831 nonzeros in the matrix, and a density equal to 0.0016. This spy plot shows that the node numbering yields a definite band structure.


The Laplacian of the mesh.
0
500
1000
1500
2000
2500
3000
3500
4000
0
1000
2000
3000
4000
nz = 28831

Sparse Matrix Operations

Most of the MATLAB standard mathematical functions work on sparse matrices just as they do on full matrices. In addition, MATLAB provides a number of functions that perform operations specific to sparse matrices. This section discusses:

- "Computational Considerations" on page 6-25
- "Standard Mathematical Operations" on page 6-25
- "Permutation and Reordering" on page 6-26
- "Factorization" on page 6-30
- "Simultaneous Linear Equations" on page 6-36
- "Eigenvalues and Singular Values" on page 6-39
- "Performance Limitations" on page 6-41

Computational Considerations

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.

Standard Mathematical Operations

Sparse matrices propagate through computations according to these rules:

- Functions that accept a matrix and return a scalar or 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 may 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.
- Multiplication and division are performed on only the nonzero elements of sparse matrices. Dividing a sparse matrix by zero returns a sparse matrix with `Inf` at each nonzero location. Because the zero-valued elements are not operated on, these elements are not returned as `NaN`. Similarly, multiplying a sparse matrix by `Inf` or `NaN` returns `Inf` or `NaN`, respectively, for the nonzero elements, but does not fill in `NaN` for the zero-valued elements.

Permutation and Reordering

A permutation of the rows and columns of a sparse matrix `S` can be represented in two ways:

- A permutation matrix `P` acts on the rows of `S` as `P*S` or on the columns as `S*P'`.
- A permutation vector `p`, which is a full vector containing a permutation of `1:n`, acts on the rows of `S` as `S(p,:)`, or on the columns as `S(:,p)`.

For example, 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` M-file 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 6-31 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].

Factorization

This section discusses four important factorization techniques for sparse matrices:

- LU, or triangular, factorization
- Cholesky factorization
- QR, or orthogonal, factorization
- Incomplete factorizations

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).

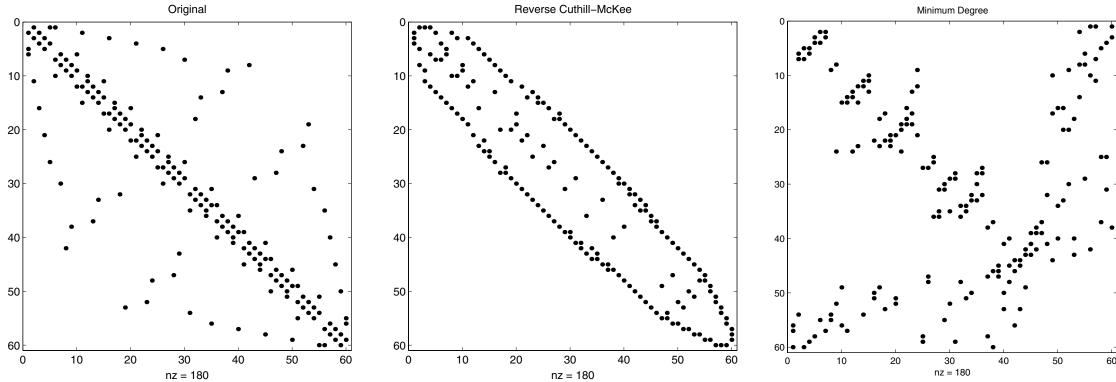
When you call lu with three or less outputs, MATLAB automatically allocates the memory necessary to hold the sparse L and U factors during the factorization. Except for the four-output syntax, MATLAB does not use any symbolic LU prefactorization to determine the memory requirements and set up the data 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`.

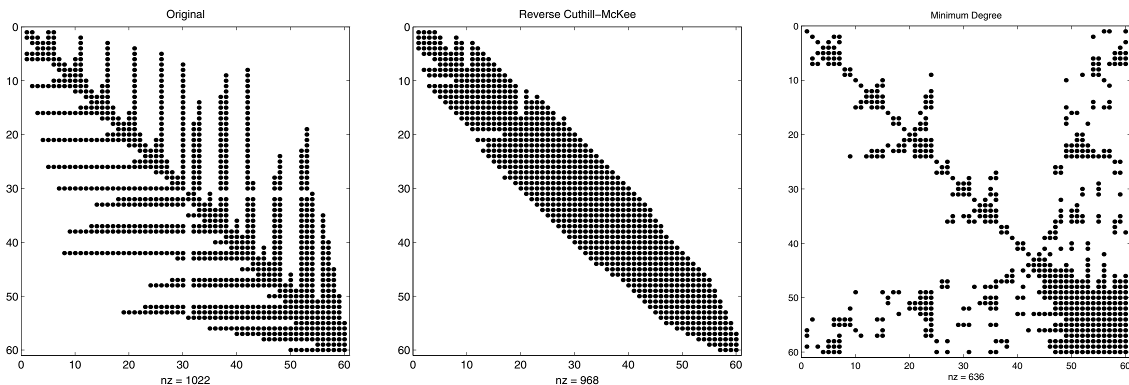
```
B = B - 3*speye(n,n);
```

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)
```

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

With one sparse input argument and one output argument

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

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

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

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

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

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

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

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

$$\text{minimize}\|Ax - b\|$$

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 roundoff error

```
r = b - A*x
e = R\(R'\(A'*r))
x = x + e
```

Incomplete Factorizations

The `luinc` 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 factorizations, one involving a drop tolerance and one involving fill-in level. If `A` is a sparse matrix, and `tol` is a small tolerance, then

```
[L,U] = luinc(A,tol)
```

computes an approximate LU factorization where all elements less than `tol` times the norm of the relevant column are set to zero. Alternatively,

```
[L,U] = luinc(A,'0')
```

computes an approximate LU factorization where the sparsity pattern of `L+U` is a permutation of the sparsity pattern of `A`.

For example,

```
load west0479
A = west0479;
nnz(A)
nnz(lu(A))
nnz(luinc(A,1e-6))
nnz(luinc(A,'0'))
```

shows that `A` has 1887 nonzeros, its complete LU factorization has 16777 nonzeros, its incomplete LU factorization with a drop tolerance of `1e-6` has 10311 nonzeros, and its `lu('0')` factorization has 1886 nonzeros.

The `luinc` function has a few other options. See the `luinc` reference page for 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.

Simultaneous 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 factorizations 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 upon properties like diagonal dominance or the existence of an underlying differential operator. Direct methods are implemented in the core of MATLAB and are made as efficient as possible for general classes of matrices. Iterative methods are usually implemented in MATLAB M-files and may make use of 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
gmres	Generalized minimum residual
lsqr	Least squares
minres	Minimum residual
pcg	Preconditioned conjugate gradient

Functions for Iterative Methods for Sparse Systems (Continued)

Function	Method
`qmr`	Quasiminimal residual
`symmlq`	Symmetric LQ

These methods are designed to solve $Ax = b$ or $\min\|b - Ax\|$. 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 may be approximated by one with constant coefficients, for example. Incomplete matrix factorizations may 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 which compute a few specified eigenvalues or singular values. `svds` is based on `eigs` which 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 by M-files.

The statement

```
[V,lambda] = eigs(A,k,sigma)
```

finds the `k` eigenvalues and corresponding eigenvectors of the matrix `A` which 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`, may be included for the generalized eigenvalue problem

$$Av = \lambda Bv$$

The statement

```
[U,S,V] = svds(A,k)
```

finds the `k` largest singular values of `A` and

```
[U,S,V] = svds(A,k,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.

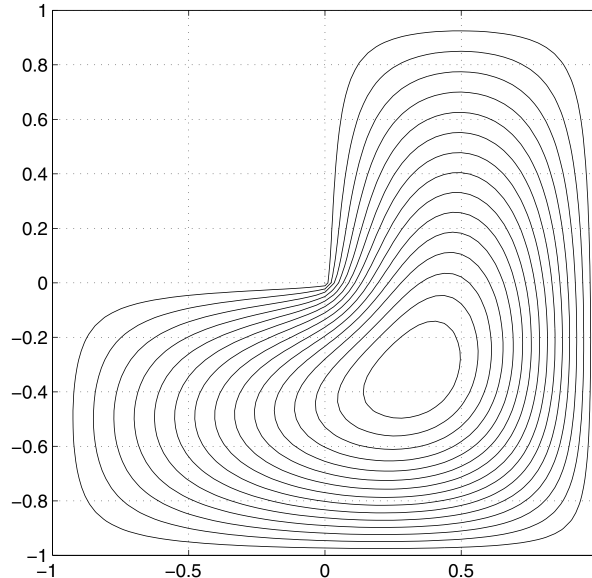
The statement

```
[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

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

Because sparse matrices are stored in a column-major format, accessing the matrix by columns is more efficient than by rows. Compare the time required for adding rows of a matrix 1000 times

```
S = sparse(10000,10000,1);
tic;
for n = 1:1000
   A = S(100,:) + S(200,:);
end;
toc

Elapsed time is 1.208162 seconds.
```

versus the time required for adding columns

```
S = sparse(10000,10000,1);
tic;
for n = 1:1000
   B = S(:,100) + S(:,200);
end;
toc

Elapsed time is 0.088747 seconds.
```

When possible, you can transpose the matrix, perform operations on the columns, and then retranspose the result:

```
S = sparse(10000,10000,1);
tic;
for n = 1:1000
   A = S(100,:)' + S(200,:)';
   A = A';
end;
toc

Elapsed time is 0.597142 seconds.
```

The time required to transpose the matrix is negligible. Note that the sparse matrix memory requirements may 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/tech_reports/`

[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.

Index

A

additional parameters
  BVP example 5-73, 5-76
adjacency matrix
  and graphing 6-17
  Bucky ball 6-18
  defined 6-17
  distance between nodes 6-22
  node 6-17
  numbering nodes 6-19
airflow modeling 6-23
`amp1dae` demo 5-42
anonymous functions
  representing mathematical functions 4-3
arguments, additional 4-30

B

`ballode` demo 5-28
bandwidth of sparse matrix, reducing 6-29
`batonode` demo 5-42
bicubic interpolation 2-12
bilinear interpolation 2-12
boundary conditions
  BVP 5-63
  BVP example 5-70
  PDE 5-91
  PDE example 5-96
Boundary Value Problems. *See* BVP
Brusselator system (ODE example) 5-25
`brussode` demo 5-25
Buckminster Fuller dome 6-18
Bucky ball 6-18
`burgersode` demo 5-42
BVP 5-61
  defined 5-63
  rewriting as first-order system 5-69
BVP solver 5-64
  basic syntax 5-65
  evaluate solution at specific points 5-72
  examples
    boundary condition at infinity (`shockbvp`) 5-76
    Mathieu's Equation (`mat4bvp`) 5-68
    multipoint terms 5-84
    rapid solution changes (`shockbvp`) 5-73
    singular terms 5-80
  initial guess 5-72
  multipoint terms 5-84
  performance 5-67
  representing problems 5-68
  singular terms 5-80
  unknown parameters 5-72
BVP solver properties
  querying property structure 5-89

C

`cat`
  sparse operands 6-26
characteristic polynomial of matrix 2-4
characteristic roots of matrix 2-4
`chol`
  sparse matrices 6-26
Cholesky factorization 1-27
  sparse matrices 6-33
closest point searches
  Delaunay triangulation 2-24
`colamd`
  minimum degree ordering 6-30
`colmmd`
  column permutation 6-31
`colperm` 6-29

comparing
  sparse and full matrix storage 6-6
complex values in sparse matrix 6-6
computational functions
  applying to sparse matrices 6-25
computational geometry
  multidimensional 2-26
  two-dimensional 2-18
contents of sparse matrix 6-13
convex hulls
  multidimensional 2-27
  two-dimensional 2-20
convolution 2-5
creating
  sparse matrix 6-8
cubic interpolation
  multidimensional 2-17
  one-dimensional 2-11
  spline 2-11
curve fitting
  polynomial 2-6
curves
  computing length 4-27
Cuthill-McKee
  reverse ordering 6-29

D

DAE
  solution of 5-2
data gridding
  multidimensional 2-17
DDE 5-49
  rewriting as first-order system 5-54
DDE solver 5-51
  basic syntax 5-52
  discontinuities 5-57
  evaluating solution at specific points 5-56
  examples
    cardiovascular model (`ddex2`) 5-58
    straightforward example (`ddex1`) 5-53
  performance 5-60
  representing problems 5-53
`ddex1` demo 5-53
`ddex2` demo 5-58
decomposition
  eigenvalue 1-38
  Schur 1-41
  singular value 1-42
deconvolution 2-5
Delaunay tessellations 2-29
Delaunay triangulation 2-20
  closest point searches 2-24
Delay Differential Equations. *See* DDE
density
  sparse matrix 6-7
derivatives
  polynomial 2-5
determinant of matrix 1-22
`diag` 6-26
diagonal
  creating sparse matrix from 6-10
differential equations 5-1
  boundary value problems for ODEs 5-61
  initial value problems for DAEs 5-2
  initial value problems for DDEs 5-49
  initial value problems for ODEs 5-2
  partial differential equations 5-89
differential-algebraic equations. *See* DAE
direct methods
  systems of sparse equations 6-36
discontinuities
  DDE solver 5-57

displaying
  sparse matrices 6-15
distance between nodes 6-22
dot product 1-8

E

eigenvalues 1-38
  of sparse matrix 6-39
eigenvectors 1-38
electrical circuits
  DAE example 5-42
Emden's equation
  example 5-81
error tolerance
  effects of too large (ODE) 5-47
  machine precision 5-45
event location (ODE)
  advanced example 5-32
  simple example 5-28
`eye`
  derivation of the name 1-10
  sparse matrices 6-25

F

factorization 6-30
  Cholesky 1-27
  Hermitian positive definite 1-28
  incomplete 6-35
  LU 1-29
  partial pivoting 1-29
  positive definite 1-27
  QR 1-30
  sparse matrices 6-30
    Cholesky 6-33
    LU 6-30
    triangular 6-30
`fem1ode` demo 5-22
`fem2ode` demo 5-42
fill-in of sparse matrix 6-22
`find` function
  sparse matrices 6-16
finite element discretization (ODE example) 5-22
first-order differential equations
  representation for BVP solver 5-69
  representation for DDE solver 5-54
Fourier analysis
  concepts 3-2
Fourier transforms
  calculating sunspot periodicity 3-3
  FFT-based interpolation 2-12
  length vs. speed 3-9
  phase and magnitude of transformed data 3-7
`fsbvp` demo 5-76
`full` 6-26, 6-29
function functions 4-1
functions
  mathematical. *See* mathematical functions
  optimizing 4-8

G

Gaussian elimination 1-29
geodesic dome 6-18
geometric analysis
  multidimensional 2-26
  two-dimensional 2-18
global minimum 4-26
global variables 4-30
`gplot` 6-18
graph
  characteristics 6-21
  defined 6-17

H

`hb1dae` demo 5-35
`hb1ode` demo 5-42
Hermitian positive definite matrix 1-28
higher-order ODEs
  rewriting as first-order ODEs 5-5

I

`iburgersode` demo 5-43
identity matrix 1-10
`ihb1dae` demo 5-42
importing
  sparse matrix 6-12
incomplete factorization 6-35
infeasible optimization problems 4-26
initial conditions
  ODE 5-4
  ODE example 5-10
  PDE 5-91
  PDE example 5-96
initial guess (BVP)
  example 5-70
  quality of 5-72
initial value problems
  DDE 5-49
  defined 5-4
  ODE and DAE 5-2
initial-boundary value PDE problems 5-89
inner product 1-7
integration
  double 4-28
  numerical 4-27
  triple 4-27
  See also differential equations
integration interval
  DDE 5-52
  PDE (MATLAB) 5-93
interpolation 2-9
  comparing methods graphically 2-13
  FFT-based 2-12
  multidimensional 2-16
    scattered data 2-34
  one-dimensional 2-10
  speed, memory, smoothness 2-11
  three-dimensional 2-16
  two-dimensional 2-12
inverse of matrix 1-22
iterative methods
  sparse matrices 6-37
  sparse systems of equations 6-36

K

Kronecker tensor matrix product 1-11

L

least squares 6-34
length of curve, computing 4-27
linear algebra 1-4
linear equations
  minimal norm solution 1-25
  overdetermined systems 1-18
  rectangular systems 1-23
  underdetermined systems 1-20
linear interpolation
  multidimensional 2-17
  one-dimensional 2-10
linear systems of equations
  direct methods (sparse) 6-36
  full 1-13
  iterative methods (sparse) 6-36
  sparse 6-36

linear transformation 1-4
`load`
  sparse matrices 6-12
Lobatto IIIa BVP solver 5-65
LU factorization 1-29
  sparse matrices and reordering 6-30

M

`mat4bvp` demo 5-63
`mat4bvp` demo 5-68
mathematical functions
  as function input arguments 4-1
  finding zeros 4-21
  minimizing 4-8
  numerical integration 4-27
  plotting 4-5
  representing in MATLAB 4-3
mathematical operations
  sparse matrices 6-25
Mathieu's equation (BVP example) 5-68
matrices 1-4
  as linear transformation 1-4
  characteristic polynomial 2-4
  characteristic roots 2-4
  creation 1-4
  determinant 1-22
  full to sparse conversion 6-7
  identity 1-10
  inverse 1-22
  iterative methods (sparse) 6-37
  orthogonal 1-30
  pseudoinverse 1-23
  rank deficiency 1-20
  symmetric 1-7
  triangular 1-27
matrix operations
  addition and subtraction 1-6
  division 1-13
  exponentials 1-35
  multiplication 1-8
  powers 1-34
  transpose 1-7
matrix products
  Kronecker tensor 1-11
`max` 6-26
M-files
  representing mathematical functions 4-3
minimizing mathematical functions
  of one variable 4-8
  of several variables 4-9
  options 4-13
minimum degree ordering 6-29
Moore-Penrose pseudoinverse 1-23
multidimensional
  data gridding 2-17
  interpolation 2-16
multidimensional interpolation 2-16
  scattered data 2-26
multistep solver (ODE) 5-6

N

nearest neighbor interpolation
  multidimensional 2-17
  one-dimensional 2-10
  three-dimensional 2-16
  two-dimensional 2-12
`nnz` 6-13
nodes 6-17
  distance between 6-22
  numbering 6-19

nonstiff ODE examples
  rigid body (`rigidode`) 5-19
nonzero elements
  maximum number in sparse matrix 6-9
  number in sparse matrix 6-13
  sparse matrix 6-13
  storage for sparse matrices 6-5
  values for sparse matrices 6-13
  visualizing for sparse matrices 6-21
`nonzeros` 6-13
norms
  vector and matrix 1-12
numerical integration 4-27
  computing length of curve 4-27
  double 4-28
  triple 4-27
`nzmax` 6-13, 6-15

O

objective function 4-1
  return values 4-26
ODE
  coding in MATLAB 5-10
  defined 5-4
  overspecified systems 5-43
  solution of 5-2
ODE solver
  evaluate solution at specific points 5-15
ODE solver properties
  fixed step sizes 5-45
ODE solvers 5-5
  algorithms
    Adams-Bashworth-Moulton PECE 5-6
    Bogacki-Shampine 5-6
    Dormand-Prince 5-6
    modified Rosenbrock formula 5-7
    numerical differentiation formulas 5-7
  backwards in time 5-47
  basic example
    stiff problem 5-12
  basic syntax 5-7
  calling 5-10
  examples 5-18
  minimizing output storage 5-44
  minimizing startup cost 5-44
  multistep solver 5-6
  nonstiff problem example 5-9
  nonstiff problems 5-6
  one-step solver 5-6
  overview 5-5
  performance 5-17
  problem size 5-44
  representing problems 5-9
  sampled data 5-47
  stiff problems 5-6, 5-12
  troubleshooting 5-43
one-dimensional interpolation 2-10
`ones`
  sparse matrices 6-25
one-step solver (ODE) 5-6
optimization 4-8
  helpful hints 4-25
  options parameters 4-13
  troubleshooting 4-26
  See also minimizing mathematical functions
`orbitode` demo 5-32
Ordinary Differential Equations. *See* ODE
orthogonal matrix 1-30
outer product 1-7
output functions 4-14
overdetermined
  rectangular matrices 1-18
overspecified ODE systems 5-43

P

Partial Differential Equations. *See* PDE
partial fraction expansion 2-7
PDE 5-89
  defined 5-90
  discretized 5-46
PDE examples (MATLAB) 5-89
PDE solver (MATLAB) 5-91
  basic syntax 5-92
  evaluate solution at specific points 5-99
  examples
    electrodynamics problem 5-100
    simple PDE 5-94
  performance 5-100
  representing problems 5-94
PDE solver (MATLAB) properties 5-100
`pdex1` demo 5-94
`pdex2` demo 5-90
`pdex3` demo 5-90
`pdex4` demo 5-100
`pdex5` demo 5-90
performance
  de-emphasizing an ODE solution component 5-46
  improving for BVP solver 5-67
  improving for DDE solver 5-60
  improving for ODE solvers 5-17
  improving for PDE solver 5-100
permutations 6-26
plotting
  mathematical functions 4-5
polynomial interpolation 2-10
polynomials
  basic operations 2-2
  calculating coefficients from roots 2-3
  calculating roots 2-3
  curve fitting 2-6
  derivatives 2-5
  evaluating 2-4
  multiplying and dividing 2-5
  partial fraction expansion 2-7
  representing as vectors 2-3
preconditioner
  sparse matrices 6-35
property structure (BVP)
  querying 5-89
pseudoinverse
  of matrix 1-23

Q

QR factorization 1-30, 6-34
`quad`, `quadl` functions
  differ from ODE solvers 5-43
quadrature. *See* numerical integration

R

`rand`
  sparse matrices 6-25
rank deficiency
  detecting 1-32
  rectangular matrices 1-20
  sparse matrices 6-34
rectangular matrices
  identity 1-10
  overdetermined systems 1-18
  pseudoinverse 1-23
  QR factorization 1-30
  rank deficient 1-20
  singular value decomposition 1-42
  underdetermined systems 1-20
reorderings 6-26
  for sparser factorizations 6-28

LU factorization 6-30
minimum degree ordering 6-29
reducing bandwidth 6-29
representing
mathematical functions 4-3
rigid body (ODE example) 5-19
`rigidode` demo 5-19
Robertson problem
DAE example 5-35
ODE example 5-42
roots
polynomial 2-3

S

sampled data
with ODE solvers 5-47
`save` 6-12
scalar
as a matrix 1-5
scalar product 1-8
scattered data
multidimensional interpolation 2-34
multidimensional tessellation 2-26
triangulation and interpolation 2-18
Schur decomposition 1-41
`seamount` data set 2-19
second difference operator
example 6-9
`shockbvp` demo 5-73
singular value matrix decomposition 1-42
`size`
sparse matrices 6-25
solution changes, rapid
making initial guess 5-73
verifying consistent behavior 5-76
solving linear systems of equations
full 1-13
sparse 6-36
`sort` 6-29
`sparse` function
converting full to sparse 6-7
sparse matrix
advantages 6-5
and complex values 6-6
Cholesky factorization 6-33
computational considerations 6-25
contents 6-13
conversion from full 6-7
creating 6-7
directly 6-8
from diagonal elements 6-10
density 6-7
distance between nodes 6-22
eigenvalues 6-39
fill-in 6-22
importing 6-12
linear systems of equations 6-36
LU factorization 6-30
and reordering 6-30
mathematical operations 6-25
nonzero elements 6-13
maximum number 6-9
specifying when creating matrix 6-8
storage 6-5, 6-13
values 6-13
nonzero elements of sparse matrix
number of 6-13
operations 6-25
permutation 6-26
preconditioner 6-35
propagation through computations 6-25
QR factorization 6-34

reordering 6-26
storage 6-5
for various permutations 6-28
viewing 6-13
triangular factorization 6-30
viewing contents graphically 6-15
viewing storage 6-13
visualizing 6-21
sparse ODE examples
Brusselator system (`brussode`) 5-25
`spconvert` 6-12
`spdiags` 6-10
`speye` 6-26
`spones` 6-29
`spparms` 6-37
`sprand` 6-26
`spy` 6-15
spy plot 6-21
startup cost
minimizing for ODE solvers 5-44
stiff ODE examples
Brusselator system (`brussode`) 5-25
differential-algebraic problem (`hb1dae`) 5-35
finite element discretization (`fem1ode`) 5-22
van der Pol (`vdpode`) 5-20
stiffness (ODE), defined 5-12
storage
minimizing for ODE problems 5-44
permutations of sparse matrices 6-28
sparse and full, comparison 6-6
sparse matrix 6-5
viewing for sparse matrix 6-13
`sum`
counting nonzeros in sparse matrix 6-29
sparse matrices 6-26
sunspot periodicity
calculating using Fourier transforms 3-3
`symamd`
minimum degree ordering 6-30
symmetric matrix
transpose 1-7
`symrcm`
column permutation 6-31
reducing sparse matrix bandwidth 6-29
systems of equations. *See* linear systems of equations

T

tessellations, multidimensional
Delaunay 2-29
Voronoi diagrams 2-31
theoretical graph 6-17
example 6-18
node 6-17
`threebvp` demo 5-63
three-dimensional interpolation 2-16
transfer functions
using partial fraction expansion 2-7
transpose
complex conjugate 1-8
unconjugated complex 1-8
triangular factorization
sparse matrices 6-30
triangular matrix 1-27
triangulation
closest point searches 2-24
Delaunay 2-20
scattered data 2-18
Voronoi diagrams 2-25
See also tessellation
tricubic interpolation 2-16
trilinear interpolation 2-16
troubleshooting (ODE) 5-43

`twobvp` demo 5-63
two-dimensional interpolation 2-12
  comparing methods graphically 2-13

U
underdetermined
  rectangular matrices 1-20
unitary matrices
  QR factorization 1-30
unknown parameters (BVP) 5-72
  example 5-68

V
van der Pol example 5-20
  simple, nonstiff 5-9
  simple, stiff 5-12
`vdpode` demo 5-20
vector products
  dot or scalar 1-8
  outer and inner 1-7
vectors
  column and row 1-5
  multiplication 1-7
visualizing
  sparse matrix 6-21
visualizing solver results
  BVP 5-71
  DDE 5-55
  ODE 5-11
  PDE 5-98
Voronoi diagrams
  multidimensional 2-31
  two-dimensional 2-25

Z
zeros
  of mathematical functions 4-21
`zeros`
  sparse matrices 6-25