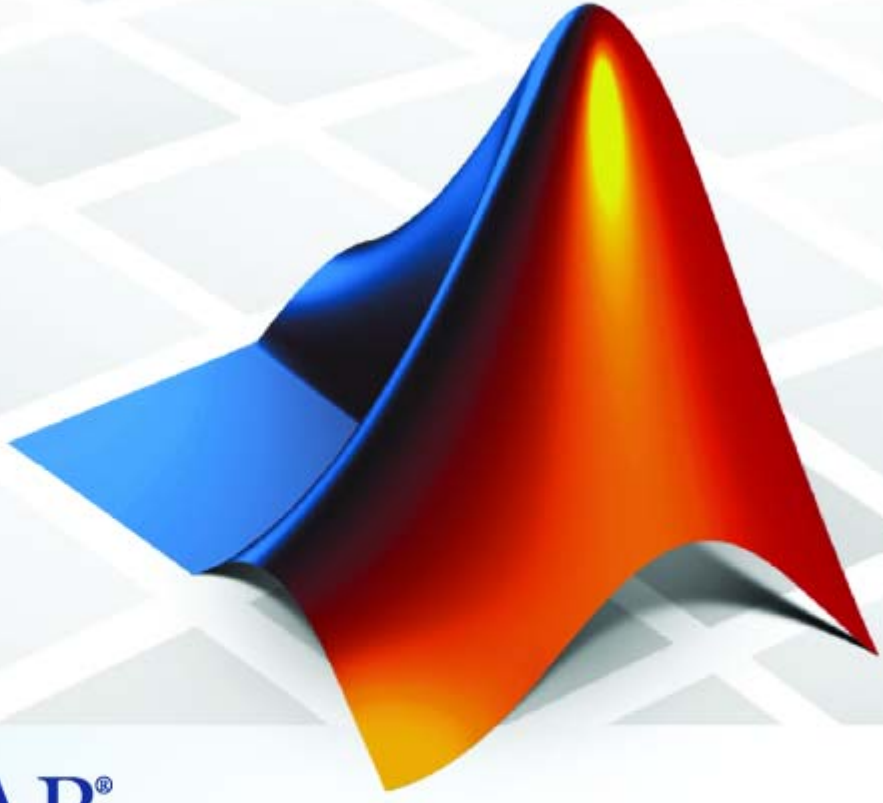
Getting Started with MATLAB® 7



MATLAB®

The MathWorks™

Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

Getting Started with MATLAB

Revision History

December 1996	First printing	For MATLAB 5
May 1997	Second printing	For MATLAB 5.1
September 1998	Third printing	For MATLAB 5.3
September 2000	Fourth printing	Revised for MATLAB 6 (Release 12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
August 2002	Fifth printing	Revised for MATLAB 6.5
June 2004	Sixth printing	Revised for MATLAB 7.0 (Release 14)
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	Seventh printing	Minor revision for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Minor revision for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Minor revision for MATLAB 7.2 (Release 2006a)
September 2006	Eighth printing	Minor revision for MATLAB 7.3 (Release 2006b)
March 2007	Ninth printing	Minor revision for MATLAB 7.4 (Release 2007a)
September 2007	Tenth printing	Minor revision for MATLAB 7.5 (Release 2007b)

Contents

1 Introduction

2 Matrices and Arrays

3 Graphics

4 Programming

5 Data Analysis

8 External Interfaces

1

Introduction

What Is MATLAB?

In this section...

Overview of MATLAB

MATLAB is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical uses include

- Math and computation
- Algorithm development
- Data acquisition
- Modeling, simulation, and prototyping
- Data analysis, exploration, and visualization
- Scientific and engineering graphics
- Application development, including graphical user interface building

MATLAB is an interactive system whose basic data element is an array that does not require dimensioning. This allows you to solve many technical computing problems, especially those with matrix and vector formulations, in a fraction of the time it would take to write a program in a scalar noninteractive language such as C or Fortran.

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects. Today, MATLAB engines incorporate the LAPACK and BLAS libraries, embedding the state of the art in software for matrix computation.

MATLAB has evolved over a period of years with input from many users. In university environments, it is the standard instructional tool for introductory

and advanced courses in mathematics, engineering, and science. In industry, MATLAB is the tool of choice for high-productivity research, development, and analysis.

MATLAB features a family of add-on application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

The MATLAB System

The MATLAB system consists of these main parts:

Desktop Tools and Development Environment

This is the set of tools and facilities that help you use MATLAB functions and files. Many of these tools are graphical user interfaces. It includes the MATLAB desktop and Command Window, a command history, an editor and debugger, a code analyzer and other reports, and browsers for viewing help, the workspace, files, and the search path.

The MATLAB Mathematical Function Library

This is a vast collection of computational algorithms ranging from elementary functions, like sum, sine, cosine, and complex arithmetic, to more sophisticated functions like matrix inverse, matrix eigenvalues, Bessel functions, and fast Fourier transforms.

The MATLAB Language

This is a high-level matrix/array language with control flow statements, functions, data structures, input/output, and object-oriented programming features. It allows both “programming in the small” to rapidly create quick and dirty throw-away programs, and “programming in the large” to create large and complex application programs.

Graphics

MATLAB has extensive facilities for displaying vectors and matrices as graphs, as well as annotating and printing these graphs. It includes high-level functions for two-dimensional and three-dimensional data visualization, image processing, animation, and presentation graphics. It also includes low-level functions that allow you to fully customize the appearance of graphics as well as to build complete graphical user interfaces on your MATLAB applications.

MATLAB External Interfaces

This is a library that allows you to write C and Fortran programs that interact with MATLAB. It includes facilities for calling routines from MATLAB (dynamic linking), calling MATLAB as a computational engine, and for reading and writing MAT-files.

MATLAB Documentation

MATLAB provides extensive documentation, in both printable and HTML format, to help you learn about and use all of its features. If you are a new user, start with this Getting Started book. It covers all the primary MATLAB features at a high level, including many examples.

To view the online documentation, select **MATLAB Help** from the **Help** menu in MATLAB. Online help appears in the Help browser, providing task-oriented and reference information about MATLAB features. For more information about using the Help browser, including typographical conventions used in the documentation, see “Help” on page 7-8.

The MATLAB documentation is organized into these main topics:

- Desktop Tools and Development Environment — Startup and shutdown, the desktop, and other tools that help you use MATLAB
- Mathematics — Mathematical operations
- Data Analysis — Data analysis, including data fitting, Fourier analysis, and time-series tools
- Programming — The MATLAB language and how to develop MATLAB applications
- Graphics — Tools and techniques for plotting, graph annotation, printing, and programming with Handle Graphics®
- 3-D Visualization — Visualizing surface and volume data, transparency, and viewing and lighting techniques
- Creating Graphical User Interfaces — GUI-building tools and how to write callback functions
- External Interfaces — MEX-files, the MATLAB engine, and interfacing to Java, COM, and the serial port

MATLAB also includes reference documentation for all MATLAB functions:

- “Functions — By Category” — Lists all MATLAB functions grouped into categories
- Handle Graphics Property Browser — Provides easy access to descriptions of graphics object properties
- C and Fortran API Reference — Covers those functions used by the MATLAB external interfaces, providing information on syntax in the calling language, description, arguments, return values, and examples

The MATLAB online documentation also includes

- Examples — An index of examples included in the documentation
- Release Notes — New features, compatibility considerations, and bug reports
- Printable Documentation — PDF versions of the documentation suitable for printing

In addition to the documentation, you can access demos from the Help browser by clicking the **Demos** tab. Run demos to learn about key functionality of MathWorks products and tools.

Starting and Quitting MATLAB

In this section...

Starting MATLAB

On Windows platforms, start MATLAB by double-clicking the MATLAB shortcut icon on your Windows desktop.

On UNIX platforms, start MATLAB by typing `matlab` at the operating system prompt.

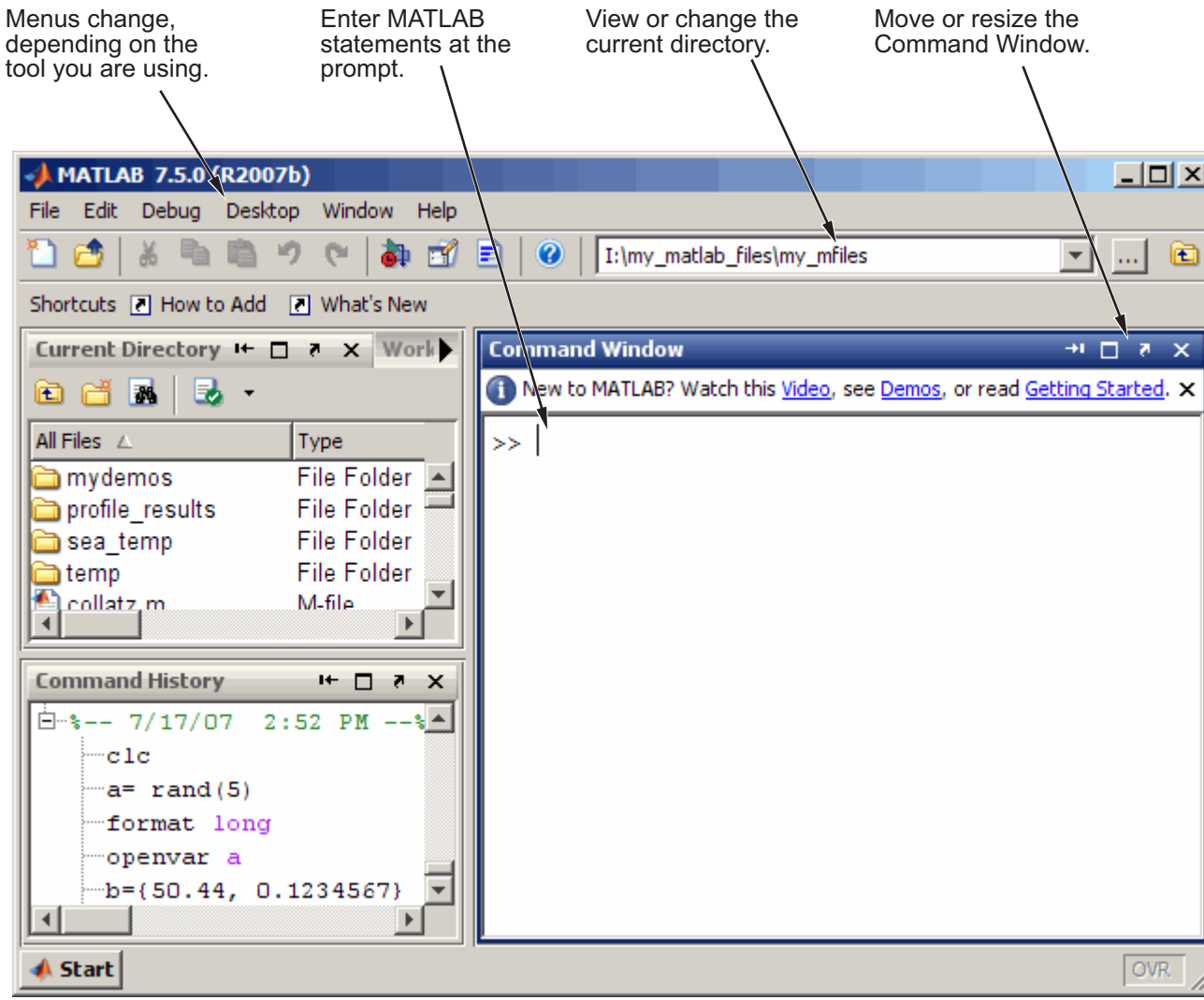
You can customize MATLAB startup. For example, you can change the directory in which MATLAB starts or automatically execute MATLAB statements in a script file named `startup.m`.

For More Information See "Starting MATLAB on Windows Platforms" and "Starting MATLAB on UNIX Platforms" in the Desktop Tools and Development Environment documentation.

MATLAB Desktop

When you start MATLAB, the MATLAB desktop appears, containing tools (graphical user interfaces) for managing files, variables, and applications associated with MATLAB.

The following illustration shows the default desktop. You can customize the arrangement of tools and documents to suit your needs. For more information about the desktop tools, see Chapter 7, "Desktop Tools and Development Environment".



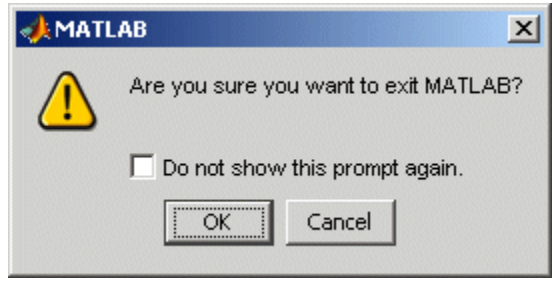
Quitting MATLAB

To end your MATLAB session, select **File > Exit MATLAB** in the desktop, or type `quit` in the Command Window. You can run a script file named

`finish.m` each time MATLAB quits that, for example, executes functions to save the workspace.

Confirm Quitting

MATLAB can display a confirmation dialog box before quitting. To set this option, select **File > Preferences > General > Confirmation Dialogs**, and select the check box for **Confirm before exiting MATLAB**.



For More Information See “Quitting MATLAB” in the Desktop Tools and Development Environment documentation.

2

Matrices and Arrays

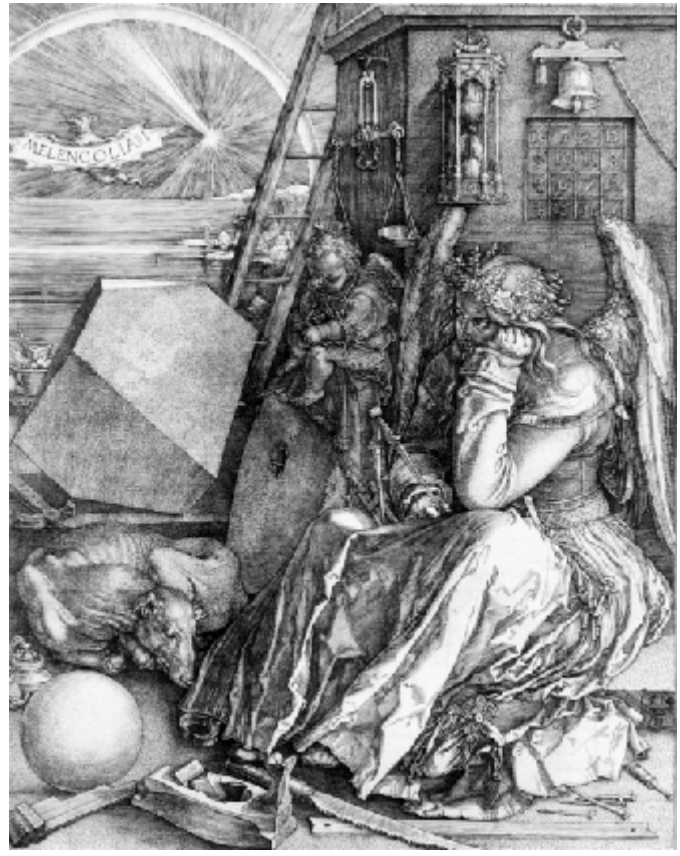
You can watch the Getting Started with MATLAB video demo for an overview of the major functionality.

Matrices and Magic Squares

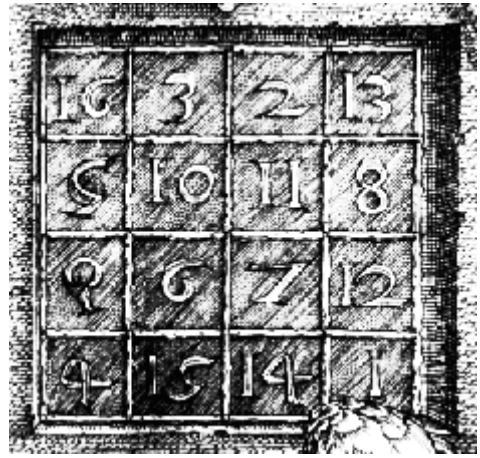
In this section...

About Matrices

In MATLAB, a matrix is a rectangular array of numbers. Special meaning is sometimes attached to 1-by-1 matrices, which are scalars, and to matrices with only one row or column, which are vectors. MATLAB has other ways of storing both numeric and nonnumeric data, but in the beginning, it is usually best to think of everything as a matrix. The operations in MATLAB are designed to be as natural as possible. Where other programming languages work with numbers one at a time, MATLAB allows you to work with entire matrices quickly and easily. A good example matrix, used throughout this book, appears in the Renaissance engraving Melencolia I by the German artist and amateur mathematician Albrecht Dürer.



This image is filled with mathematical symbolism, and if you look carefully, you will see a matrix in the upper right corner. This matrix is known as a magic square and was believed by many in Dürer's time to have genuinely magical properties. It does turn out to have some fascinating characteristics worth exploring.



Entering Matrices

The best way for you to get started with MATLAB is to learn how to handle matrices. Start MATLAB and follow along with each example.

You can enter matrices into MATLAB in several different ways:

- Enter an explicit list of elements.
- Load matrices from external data files.
- Generate matrices using built-in functions.
- Create matrices with your own functions in M-files.

Start by entering Dürer's matrix as a list of its elements. You only have to follow a few basic conventions:

- Separate the elements of a row with blanks or commas.
- Use a semicolon, `;` , to indicate the end of each row.
- Surround the entire list of elements with square brackets, `[ ]`.

To enter Dürer's matrix, simply type in the Command Window

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

MATLAB displays the matrix you just entered:

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

This matrix matches the numbers in the engraving. Once you have entered the matrix, it is automatically remembered in the MATLAB workspace. You can refer to it simply as `A`. Now that you have `A` in the workspace, take a look at what makes it so interesting. Why is it magic?

sum, transpose, and diag

You are probably already aware that the special properties of a magic square have to do with the various ways of summing its elements. If you take the sum along any row or column, or along either of the two main diagonals, you will always get the same number. Let us verify that using MATLAB. The first statement to try is

```
sum(A)
```

MATLAB replies with

```
ans =
    34    34    34    34
```

When you do not specify an output variable, MATLAB uses the variable `ans`, short for *answer*, to store the results of a calculation. You have computed a row vector containing the sums of the columns of `A`. Sure enough, each of the columns has the same sum, the *magic* sum, 34.

How about the row sums? MATLAB has a preference for working with the columns of a matrix, so one way to get the row sums is to transpose the matrix, compute the column sums of the transpose, and then transpose the result. For an additional way that avoids the double transpose use the dimension argument for the `sum` function.

MATLAB has two transpose operators. The apostrophe operator (e.g., `A'`) performs a complex conjugate transposition. It flips a matrix about its main

diagonal, and also changes the sign of the imaginary component of any complex elements of the matrix. The dot-apostrophe operator (e.g., `A.'`), transposes without affecting the sign of complex elements. For matrices containing all real elements, the two operators return the same result.

So

```
A'
```

produces

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

and

```
sum(A')'
```

produces a column vector containing the row sums

```
ans =
    34
    34
    34
    34
```

The sum of the elements on the main diagonal is obtained with the `sum` and the `diag` functions:

```
diag(A)
```

produces

```
ans =
    16
    10
     7
     1
```

and

```
sum(diag(A))
```

produces

```
ans =
    34
```

The other diagonal, the so-called *antidiagonal,* is not so important mathematically, so MATLAB does not have a ready-made function for it. But a function originally intended for use in graphics, `fliplr`, flips a matrix from left to right:

```
sum(diag(fliplr(A)))
ans =
    34
```

You have verified that the matrix in Dürer's engraving is indeed a magic square and, in the process, have sampled a few MATLAB matrix operations. The following sections continue to use this matrix to illustrate additional MATLAB capabilities.

Subscripts

The element in row `i` and column `j` of `A` is denoted by `A(i,j)`. For example, `A(4,2)` is the number in the fourth row and second column. For our magic square, `A(4,2)` is `15`. So to compute the sum of the elements in the fourth column of `A`, type

```
A(1,4) + A(2,4) + A(3,4) + A(4,4)
```

This produces

```
ans =
     34
```

but is not the most elegant way of summing a single column.

It is also possible to refer to the elements of a matrix with a single subscript, `A(k)`. This is the usual way of referencing row and column vectors. But it can also apply to a fully two-dimensional matrix, in which case the array is

regarded as one long column vector formed from the columns of the original matrix. So, for our magic square, `A(8)` is another way of referring to the value `15` stored in `A(4,2)`.

If you try to use the value of an element outside of the matrix, it is an error:

```
t = A(4,5)
Index exceeds matrix dimensions.
```

On the other hand, if you store a value in an element outside of the matrix, the size increases to accommodate the newcomer:

```
X = A;
X(4,5) = 17

X =
    16     3     2    13     0
     5    10    11     8     0
     9     6     7    12     0
     4    15    14     1    17
```

The Colon Operator

The colon, `:`, is one of the most important MATLAB operators. It occurs in several different forms. The expression

```
1:10
```

is a row vector containing the integers from 1 to 10:

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

To obtain nonunit spacing, specify an increment. For example,

```
100:-7:50
```

is

```
100    93    86    79    72    65    58    51
```

and

```
0:pi/4:pi
```

is

```
0    0.7854    1.5708    2.3562    3.1416
```

Subscript expressions involving colons refer to portions of a matrix:

```
A(1:k,j)
```

is the first `k` elements of the `j`th column of `A`. So

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

computes the sum of the fourth column. But there is a better way. The colon by itself refers to *all* the elements in a row or column of a matrix and the keyword `end` refers to the *last* row or column. So

```
sum(A(:,end))
```

computes the sum of the elements in the last column of `A`:

```
ans =
     34
```

Why is the magic sum for a 4-by-4 square equal to 34? If the integers from 1 to 16 are sorted into four groups with equal sums, that sum must be

```
sum(1:16)/4
```

which, of course, is

```
ans =
     34
```

The magic Function

MATLAB actually has a built-in function that creates magic squares of almost any size. Not surprisingly, this function is named `magic`:

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

This matrix is almost the same as the one in the Dürer engraving and has all the same "magic" properties; the only difference is that the two middle columns are exchanged.

To make this `B` into Dürer's `A`, swap the two middle columns:

```
A = B(:,[1 3 2 4])
```

This says, for each of the rows of matrix `B`, reorder the elements in the order 1, 3, 2, 4. It produces

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

Why would Dürer go to the trouble of rearranging the columns when he could have used MATLAB ordering? No doubt he wanted to include the date of the engraving, 1514, at the bottom of his magic square.

Expressions

In this section...

Variables

Like most other programming languages, MATLAB provides mathematical *expressions*, but unlike most programming languages, these expressions involve entire matrices.

MATLAB does not require any type declarations or dimension statements. When MATLAB encounters a new variable name, it automatically creates the variable and allocates the appropriate amount of storage. If the variable already exists, MATLAB changes its contents and, if necessary, allocates new storage. For example,

```
num_students = 25
```

creates a 1-by-1 matrix named `num_students` and stores the value 25 in its single element. To view the matrix assigned to any variable, simply enter the variable name.

Variable names consist of a letter, followed by any number of letters, digits, or underscores. MATLAB is case sensitive; it distinguishes between uppercase and lowercase letters. `A` and `a` are *not* the same variable.

Although variable names can be of any length, MATLAB uses only the first `N` characters of the name, (where `N` is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first `N` characters to enable MATLAB to distinguish variables.

```
N = namelengthmax
```

```
N =
    63
```

The `genvarname` function can be useful in creating variable names that are both valid and unique.

Numbers

MATLAB uses conventional decimal notation, with an optional decimal point and leading plus or minus sign, for numbers. *Scientific notation* uses the letter `e` to specify a power-of-ten scale factor. *Imaginary numbers* use either `i` or `j` as a suffix. Some examples of legal numbers are

```
3              -99            0.0001
9.6397238      1.60210e-20    6.02252e23
1i             -3.14159j      3e5i
```

All numbers are stored internally using the *long* format specified by the IEEE floating-point standard. Floating-point numbers have a finite *precision* of roughly 16 significant decimal digits and a finite *range* of roughly 10^{-308} to 10^{+308}.

The section "Avoiding Common Problems with Floating-Point Arithmetic" gives a few of the examples showing how IEEE floating-point arithmetic affects computations in MATLAB. For more examples and information, see Technical Note 1108 — Common Problems with Floating-Point Arithmetic.

Operators

Expressions use familiar arithmetic operators and precedence rules.

`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`\`	Left division (described in "Matrices and Linear Algebra" in the MATLAB documentation)
`^`	Power

'	Complex conjugate transpose
( )	Specify evaluation order

Functions

MATLAB provides a large number of standard elementary mathematical functions, including `abs`, `sqrt`, `exp`, and `sin`. Taking the square root or logarithm of a negative number is not an error; the appropriate complex result is produced automatically. MATLAB also provides many more advanced mathematical functions, including Bessel and gamma functions. Most of these functions accept complex arguments. For a list of the elementary mathematical functions, type

```
help elfun
```

For a list of more advanced mathematical and matrix functions, type

```
help specfun
help elmat
```

Some of the functions, like `sqrt` and `sin`, are *built in*. Built-in functions are part of the MATLAB core so they are very efficient, but the computational details are not readily accessible. Other functions, like `gamma` and `sinh`, are implemented in M-files.

There are some differences between built-in functions and other functions. For example, for built-in functions, you cannot see the code. For other functions, you can see the code and even modify it if you want.

Several special functions provide values of useful constants.

`pi`	3.14159265...
`i`	Imaginary unit, $\sqrt{-1}$
`j`	Same as `i`
`eps`	Floating-point relative precision, $\varepsilon = 2^{-52}$
`realmin`	Smallest floating-point number, 2^{-1022}

`realmax`	Largest floating-point number, $(2-\varepsilon)2^{1023}$
`Inf`	Infinity
`NaN`	Not-a-number

Infinity is generated by dividing a nonzero value by zero, or by evaluating well defined mathematical expressions that *overflow*, i.e., exceed `realmax`. Not-a-number is generated by trying to evaluate expressions like `0/0` or `Inf-Inf` that do not have well defined mathematical values.

The function names are not reserved. It is possible to overwrite any of them with a new variable, such as

```
eps = 1.e-6
```

and then use that value in subsequent calculations. The original function can be restored with

```
clear eps
```

Examples of Expressions

You have already seen several examples of MATLAB expressions. Here are a few more examples, and the resulting values:

```
rho = (1+sqrt(5))/2
rho =
    1.6180

a = abs(3+4i)
a =
     5

z = sqrt(besselk(4/3,rho-i))
z =
   0.3730+ 0.3214i

huge = exp(log(realmax))
huge =
  1.7977e+308
```

```
toobig = pi*huge
toobig =
   Inf
```

Working with Matrices

In this section...

Generating Matrices

MATLAB provides four functions that generate basic matrices.

`zeros`	All zeros
`ones`	All ones
`rand`	Uniformly distributed random elements
`randn`	Normally distributed random elements

Here are some examples:

```
Z = zeros(2,4)
Z =
     0     0     0     0
     0     0     0     0

F = 5*ones(3,3)
F =
     5     5     5
     5     5     5
     5     5     5

N = fix(10*rand(1,10))
N =
     9     2     6     4     8     7     4     0     8     4
```

```
R = randn(4,4)
R =
    0.6353    0.0860   -0.3210   -1.2316
   -0.6014   -2.0046    1.2366    1.0556
    0.5512   -0.4931   -0.6313   -0.1132
   -1.0998    0.4620   -2.3252    0.3792
```

The load Function

The `load` function reads binary files containing matrices generated by earlier MATLAB sessions, or reads text files containing numeric data. The text file should be organized as a rectangular table of numbers, separated by blanks, with one row per line, and an equal number of elements in each row. For example, outside of MATLAB, create a text file containing these four lines:

```
16.0     3.0     2.0    13.0
 5.0    10.0    11.0     8.0
 9.0     6.0     7.0    12.0
 4.0    15.0    14.0     1.0
```

Save the file as `magik.dat` in the current directory. The statement

```
load magik.dat
```

reads the file and creates a variable, `magik`, containing the example matrix.

An easy way to read data into MATLAB in many text or binary formats is to use the Import Wizard.

M-Files

You can create your own matrices using *M-files*, which are text files containing MATLAB code. Use the MATLAB Editor or another text editor to create a file containing the same statements you would type at the MATLAB command line. Save the file under a name that ends in `.m`.

For example, create a file in the current directory named `magik.m` containing these five lines:

```
A = [16.0     3.0     2.0    13.0
      5.0    10.0    11.0     8.0
```

```
     9.0     6.0     7.0    12.0
     4.0    15.0    14.0     1.0 ];
```

The statement

```
magik
```

reads the file and creates a variable, `A`, containing the example matrix.

Concatenation

Concatenation is the process of joining small matrices to make bigger ones. In fact, you made your first matrix by concatenating its individual elements. The pair of square brackets, `[]`, is the concatenation operator. For an example, start with the 4-by-4 magic square, `A`, and form

```
B = [A  A+32; A+48  A+16]
```

The result is an 8-by-8 matrix, obtained by joining the four submatrices:

```
B =

    16     3     2    13    48    35    34    45
     5    10    11     8    37    42    43    40
     9     6     7    12    41    38    39    44
     4    15    14     1    36    47    46    33
    64    51    50    61    32    19    18    29
    53    58    59    56    21    26    27    24
    57    54    55    60    25    22    23    28
    52    63    62    49    20    31    30    17
```

This matrix is halfway to being another magic square. Its elements are a rearrangement of the integers `1:64`. Its column sums are the correct value for an 8-by-8 magic square:

```
sum(B)

ans =
   260   260   260   260   260   260   260   260
```

But its row sums, `sum(B')'`, are not all the same. Further manipulation is necessary to make this a valid 8-by-8 magic square.

Deleting Rows and Columns

You can delete rows and columns from a matrix using just a pair of square brackets. Start with

```
X = A;
```

Then, to delete the second column of `X`, use

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

This changes `X` to

```
X =
    16     2    13
     5    11     8
     9     7    12
     4    14     1
```

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

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

result in an error. However, using a single subscript deletes a single element, or sequence of elements, and reshapes the remaining elements into a row vector. So

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

results in

```
X =
    16     9     2     7    13    12     1
```

More About Matrices and Arrays

In this section...

Linear Algebra

Informally, the terms *matrix* and *array* are often used interchangeably. More precisely, a *matrix* is a two-dimensional numeric array that represents a *linear transformation*. The mathematical operations defined on matrices are the subject of *linear algebra*.

Dürer's magic square

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

provides several examples that give a taste of MATLAB matrix operations. You have already seen the matrix transpose, `A'`. Adding a matrix to its transpose produces a *symmetric* matrix:

```
A + A'

ans =
    32     8    11    17
     8    20    17    23
    11    17    14    26
    17    23    26     2
```

The multiplication symbol, `*`, denotes the *matrix* multiplication involving inner products between rows and columns. Multiplying the transpose of a matrix by the original matrix also produces a symmetric matrix:

```
A'*A

ans =
   378   212   206   360
   212   370   368   206
   206   368   370   212
   360   206   212   378
```

The determinant of this particular matrix happens to be zero, indicating that the matrix is *singular*:

```
d = det(A)

d =
     0
```

The reduced row echelon form of `A` is not the identity:

```
R = rref(A)

R =
     1     0     0     1
     0     1     0    -3
     0     0     1     3
     0     0     0     0
```

Since the matrix is singular, it does not have an inverse. If you try to compute the inverse with

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

you will get a warning message:

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 9.796086e-018.
```

Roundoff error has prevented the matrix inversion algorithm from detecting exact singularity. But the value of `rcond`, which stands for *reciprocal*

condition estimate, is on the order of `eps`, the floating-point relative precision, so the computed inverse is unlikely to be of much use.

The eigenvalues of the magic square are interesting:

```
e = eig(A)

e =
   34.0000
    8.0000
    0.0000
   -8.0000
```

One of the eigenvalues is zero, which is another consequence of singularity. The largest eigenvalue is 34, the magic sum. That is because the vector of all ones is an eigenvector:

```
v = ones(4,1)

v =
     1
     1
     1
     1

A*v

ans =
    34
    34
    34
    34
```

When a magic square is scaled by its magic sum,

```
P = A/34
```

the result is a *doubly stochastic* matrix whose row and column sums are all 1:

```
P =
    0.4706    0.0882    0.0588    0.3824
```

```
    0.1471    0.2941    0.3235    0.2353
    0.2647    0.1765    0.2059    0.3529
    0.1176    0.4412    0.4118    0.0294
```

Such matrices represent the transition probabilities in a *Markov process*. Repeated powers of the matrix represent repeated steps of the process. For our example, the fifth power

```
P^5
```

is

```
    0.2507    0.2495    0.2494    0.2504
    0.2497    0.2501    0.2502    0.2500
    0.2500    0.2498    0.2499    0.2503
    0.2496    0.2506    0.2505    0.2493
```

This shows that as k approaches infinity, all the elements in the kth power, P^k, approach $1/4$.

Finally, the coefficients in the characteristic polynomial

```
poly(A)
```

are

```
     1   -34   -64  2176     0
```

This indicates that the characteristic polynomial

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

is

$$\lambda^4 - 34\lambda^3 - 64\lambda^2 + 2176\lambda$$

The constant term is zero, because the matrix is singular, and the coefficient of the cubic term is -34, because the matrix is magic!

Arrays

When they are taken away from the world of linear algebra, matrices become two-dimensional numeric arrays. Arithmetic operations on arrays are done element by element. This means that addition and subtraction are the same for arrays and matrices, but that multiplicative operations are different. MATLAB uses a dot, or decimal point, as part of the notation for multiplicative array operations.

The list of operators includes

`+`	Addition
`-`	Subtraction
`.*`	Element-by-element multiplication
`./`	Element-by-element division
`.\`	Element-by-element left division
`.^`	Element-by-element power
`.'`	Unconjugated array transpose

If the Dürer magic square is multiplied by itself with array multiplication

```
A.*A
```

the result is an array containing the squares of the integers from 1 to 16, in an unusual order:

```
ans =
   256     9     4   169
    25   100   121    64
    81    36    49   144
    16   225   196     1
```

Building Tables

Array operations are useful for building tables. Suppose `n` is the column vector

```
n = (0:9)';
```

Then

```
pows = [n  n.^2  2.^n]
```

builds a table of squares and powers of 2:

```
pows =
     0     0     1
     1     1     2
     2     4     4
     3     9     8
     4    16    16
     5    25    32
     6    36    64
     7    49   128
     8    64   256
     9    81   512
```

The elementary math functions operate on arrays element by element. So

```
format short g
x = (1:0.1:2)';
logs = [x log10(x)]
```

builds a table of logarithms.

```
 logs =
     1.0           0
     1.1     0.04139
     1.2     0.07918
     1.3     0.11394
     1.4     0.14613
     1.5     0.17609
     1.6     0.20412
     1.7     0.23045
     1.8     0.25527
     1.9     0.27875
     2.0     0.30103
```

Multivariate Data

MATLAB uses column-oriented analysis for multivariate statistical data. Each column in a data set represents a variable and each row an observation. The `(i,j)`th element is the `i`th observation of the `j`th variable.

As an example, consider a data set with three variables:

- Heart rate
- Weight
- Hours of exercise per week

For five observations, the resulting array might look like

```
D = [ 72          134          3.2
      81          201          3.5
      69          156          7.1
      82          148          2.4
      75          170          1.2 ]
```

The first row contains the heart rate, weight, and exercise hours for patient 1, the second row contains the data for patient 2, and so on. Now you can apply many MATLAB data analysis functions to this data set. For example, to obtain the mean and standard deviation of each column, use

```
mu = mean(D), sigma = std(D)

mu =
      75.8       161.8         3.48

sigma =
    5.6303      25.499       2.2107
```

For a list of the data analysis functions available in MATLAB, type

```
help datafun
```

If you have access to Statistics Toolbox, type

```
help stats
```

Scalar Expansion

Matrices and scalars can be combined in several different ways. For example, a scalar is subtracted from a matrix by subtracting it from each element. The average value of the elements in our magic square is 8.5, so

```
B = A - 8.5
```

forms a matrix whose column sums are zero:

```
B =
      7.5     -5.5     -6.5      4.5
     -3.5      1.5      2.5     -0.5
      0.5     -2.5     -1.5      3.5
     -4.5      6.5      5.5     -7.5

sum(B)

ans =
     0     0     0     0
```

With scalar expansion, MATLAB assigns a specified scalar to all indices in a range. For example,

```
B(1:2,2:3) = 0
```

zeroes out a portion of `B`:

```
B =
      7.5        0        0      4.5
     -3.5        0        0     -0.5
      0.5     -2.5     -1.5      3.5
     -4.5      6.5      5.5     -7.5
```

Logical Subscripting

The logical vectors created from logical and relational operations can be used to reference subarrays. Suppose `X` is an ordinary matrix and `L` is a matrix of the same size that is the result of some logical operation. Then `X(L)` specifies the elements of `X` where the elements of `L` are nonzero.

This kind of subscripting can be done in one step by specifying the logical operation as the subscripting expression. Suppose you have the following set of data:

```
x = [2.1 1.7 1.6 1.5 NaN 1.9 1.8 1.5 5.1 1.8 1.4 2.2 1.6 1.8];
```

The `NaN` is a marker for a missing observation, such as a failure to respond to an item on a questionnaire. To remove the missing data with logical indexing, use `isfinite(x)`, which is true for all finite numerical values and false for `NaN` and `Inf`:

```
x = x(isfinite(x))
x =
  2.1 1.7 1.6 1.5 1.9 1.8 1.5 5.1 1.8 1.4 2.2 1.6 1.8
```

Now there is one observation, `5.1`, which seems to be very different from the others. It is an *outlier*. The following statement removes outliers, in this case those elements more than three standard deviations from the mean:

```
x = x(abs(x-mean(x)) <= 3*std(x))
x =
  2.1 1.7 1.6 1.5 1.9 1.8 1.5 1.8 1.4 2.2 1.6 1.8
```

For another example, highlight the location of the prime numbers in Dürer's magic square by using logical indexing and scalar expansion to set the nonprimes to 0. (See "The magic Function" on page 2-9.)

```
A(~isprime(A)) = 0

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

The find Function

The `find` function determines the indices of array elements that meet a given logical condition. In its simplest form, `find` returns a column vector of indices. Transpose that vector to obtain a row vector of indices. For example, start again with Dürer's magic square. (See "The magic Function" on page 2-9.)

```
k = find(isprime(A))'
```

picks out the locations, using one-dimensional indexing, of the primes in the magic square:

```
k =
     2     5     9    10    11    13
```

Display those primes, as a row vector in the order determined by `k`, with

```
A(k)

ans =
     5     3     2    11     7    13
```

When you use `k` as a left-hand-side index in an assignment statement, the matrix structure is preserved:

```
A(k) = NaN

A =
    16   NaN   NaN   NaN
   NaN    10   NaN     8
     9     6   NaN    12
     4    15    14     1
```

Controlling Command Window Input and Output

In this section...

The format Function

The `format` function controls the numeric format of the values displayed by MATLAB. The function affects only how numbers are displayed, not how MATLAB computes or saves them. Here are the different formats, together with the resulting output produced from a vector `x` with components of different magnitudes.

Note To ensure proper spacing, use a fixed-width font, such as Courier.

```
x = [4/3 1.2345e-6]

format short

   1.3333    0.0000

format short e

   1.3333e+000  1.2345e-006

format short g

   1.3333  1.2345e-006

format long

   1.33333333333333   0.00000123450000
```

```
format long e

   1.333333333333333e+000    1.234500000000000e-006

format long g

   1.33333333333333               1.2345e-006

format bank

   1.33          0.00

format rat

   4/3          1/810045

format hex

   3ff5555555555555   3eb4b6231abfd271
```

If the largest element of a matrix is larger than 10^3 or smaller than 10^{-3}, MATLAB applies a common scale factor for the short and long formats.

In addition to the `format` functions shown above

```
format compact
```

suppresses many of the blank lines that appear in the output. This lets you view more information on a screen or window. If you want more control over the output format, use the `sprintf` and `fprintf` functions.

Suppressing Output

If you simply type a statement and press **Return** or **Enter**, MATLAB automatically displays the results on screen. However, if you end the line with a semicolon, MATLAB performs the computation but does not display any output. This is particularly useful when you generate large matrices. For example,

```
A = magic(100);
```

Entering Long Statements

If a statement does not fit on one line, use an ellipsis (three periods), `...`, followed by **Return** or **Enter** to indicate that the statement continues on the next line. For example,

```
s = 1 -1/2 + 1/3 -1/4 + 1/5 - 1/6 + 1/7 ...
      - 1/8 + 1/9 - 1/10 + 1/11 - 1/12;
```

Blank spaces around the =, +, and - signs are optional, but they improve readability.

Command Line Editing

Various arrow and control keys on your keyboard allow you to recall, edit, and reuse statements you have typed earlier. For example, suppose you mistakenly enter

```
rho = (1 + sqt(5))/2
```

You have misspelled `sqrt`. MATLAB responds with

```
Undefined function or variable 'sqt'.
```

Instead of retyping the entire line, simply press the ↑ key. The statement you typed is redisplayed. Use the ← key to move the cursor over and insert the missing `r`. Repeated use of the ↑ key recalls earlier lines. Typing a few characters and then the ↑ key finds a previous line that begins with those characters. You can also copy previously executed statements from the Command History. For more information, see “Command History” on page 7-7.

Following is the list of arrow and control keys you can use in the Command Window. If the preference you select for “Command Window Key Bindings” is `MATLAB standard (Emacs)`, you can also use the **Ctrl**+key combinations shown. See also general keyboard shortcuts for desktop tools in the MATLAB Desktop Tools and Development Environment documentation.

Key	Control Key for MATLAB Standard (Emacs) Preference	Operation
↑	**Ctrl+P**	Recall *p*revious line. Works only at command line.
↓	**Ctrl+N**	Recall *n*ext line. Works only at the prompt if you previously used the up arrow or **Ctrl+P**.
←	**Ctrl+B**	Move *b*ack one character.
→	**Ctrl+F**	Move *f*orward one character.
Ctrl+→	None	Move *r*ight one word.
Ctrl+←	None	Move *l*eft one word.
Home	**Ctrl+A**	Move to beginning of current statement.
End	**Ctrl+E**	Move to *e*nd of current statement.
Ctrl+Home	None	Move to top of Command Window.
Ctrl+End	None	Move to end of Command Window.
Esc	**Ctrl+U**	Clear command line when cursor is at the prompt. Otherwise, move cursor to the prompt.
Delete	**Ctrl+D**	Delete character after cursor.
Backspace	**Ctrl+H**	Delete character before cursor.
None	**Ctrl+K**	Cut contents (*k*ill) to end of command line.
Shift+Home	None	Select from cursor to beginning of statement.
Shift+End	None	Select from cursor to end of statement.

3

Graphics

Overview of MATLAB Plotting

In this section...

For More Information MATLAB Graphics and 3-D Visualization in the MATLAB documentation provide in-depth coverage of MATLAB graphics and visualization tools. Access these topics from the Help browser.

Plotting Process

MATLAB provides a wide variety of techniques to display data graphically. Interactive tools enable you to manipulate graphs to achieve results that reveal the most information about your data. You can also annotate and print graphs for presentations, or export graphs to standard graphics formats for presentation in web browsers or other media.

The process of visualizing data typically involves a series of operations. This section provides a "big picture" view of the plotting process and contains links to sections that have examples and specific details about performing each operation.

Creating a Graph

The type of graph you choose to create depends on the nature of your data and what you want to reveal about the data. MATLAB predefines many graph types, such as line, bar, histogram, and pie graphs. There are also 3-D graphs, such as surfaces, slice planes, and streamlines.

There are two basic ways to create graphs in MATLAB:

- Use plotting tools to create graphs interactively.

See “Some Ways to Use MATLAB Plotting Tools” on page 3-23.

- Use the command interface to enter commands in the Command Window or create plotting programs.

  See “Using Basic Plotting Functions” on page 3-49.

You might find it useful to combine both approaches. For example, you might issue a plotting command to create a graph and then modify the graph using one of the interactive tools.

Exploring Data

Once you create a graph, you can extract specific information about the data, such as the numeric value of a peak in a plot, the average value of a series of data, or you can perform data fitting.

For More Information See “Data Exploration Tools” in the MATLAB Graphics documentation and “Opening the Basic Fitting GUI” in the MATLAB Data Analysis documentation.

Editing the Graph Components

Graphs are composed of objects, which have properties you can change. These properties affect the way the various graph components look and behave.

For example, the axes used to define the coordinate system of the graph has properties that define the limits of each axis, the scale, color, etc. The line used to create a line graph has properties such as color, type of marker used at each data point (if any), line style, etc.

Note that the data used to create a line graph are properties of the line. You can, therefore, change the data without actually creating a new graph.

See “Editing Plots” on page 3-17.

Annotating Graphs

Annotations are the text, arrows, callouts, and other labels added to graphs to help viewers see what is important about the data. You typically add

annotations to graphs when you want to show them to other people or when you want to save them for later reference.

For More Information See "Annotating Graphs" in the MATLAB Graphics documentation or select **Annotating Graphs** from the figure **Help** menu.

Printing and Exporting Graphs

You can print your graph on any printer connected to your computer. The print previewer enables you to view how your graph will look when printed. It enables you to add headers, footers, a date, and so on. The print preview dialog lets you control the size, layout, and other characteristics of the graph (select **Print Preview** from the figure **File** menu).

Exporting a graph means creating a copy of it in a standard graphics file format, such as TIFF, JPEG, or EPS. You can then import the file into a word processor, include it in an HTML document, or edit it in a drawing package (select **Export Setup** from the figure **File** menu).

Adding and Removing Figure Content

By default, when you create a new graph in the same figure window, its data replaces that of the graph that is currently displayed, if any. You can add new data to a graph in several ways; see "Adding More Data to the Graph" on page 3-27 for how to do this using a GUI. You can manually remove all data, graphics and annotations from the current figure by typing `CLF` in the Command Window or by selecting **Clear Figure** from the figure's **Edit** menu.

For More Information See the `print` command reference page and "Printing and Exporting" in the MATLAB Graphics documentation or select **Printing and Exporting** from the figure **Help** menu.

Saving Graphs to Reload into MATLAB

There are two ways to save graphs that enable you to save the work you have invested in their preparation:

- Save the graph as a FIG-file (select **Save** from the figure **File** menu).
- Generate MATLAB code that can recreate the graph (select **Generate M-File** from the figure **File** menu).

FIG-Files. FIG-files are a binary format that saves a figure in its current state. This means that all graphics objects and property settings are stored in the file when you create it. You can reload the file into a different MATLAB session, even if you are running MATLAB on a different type of computer. When you load a FIG-file, MATLAB creates a new figure in the same state as the one you saved.

Note that the states of any figure tools (i.e., any items on the toolbars) are not saved in a FIG-file; only the contents of the graph are saved.

Generated Code. You can use the MATLAB M-code generator to create code that recreates the graph. Unlike a FIG-file, the generated code does not contain any data. You must pass appropriate data to the generated function when you run the code.

Studying the generated code for a graph is a good way to learn how to program with MATLAB.

For More Information See the `print` command reference page and "Saving Your Work" in the MATLAB Graphics documentation.

Graph Components

MATLAB displays graphs in a special window known as a figure. To create a graph, you need to define a coordinate system. Therefore every graph is placed within axes, which are contained by the figure.

The actual visual representation of the data is achieved with graphics objects like lines and surfaces. These objects are drawn within the coordinate system defined by the axes, which MATLAB automatically creates specifically to accommodate the range of the data. The actual data is stored as properties of the graphics objects.

See “Handle Graphics” on page 3-74 for more information about graphics object properties.

The following picture shows the basic components of a typical graph. You can find commands for plotting this graph in “Preparing Graphs for Presentation” on page 3-37.

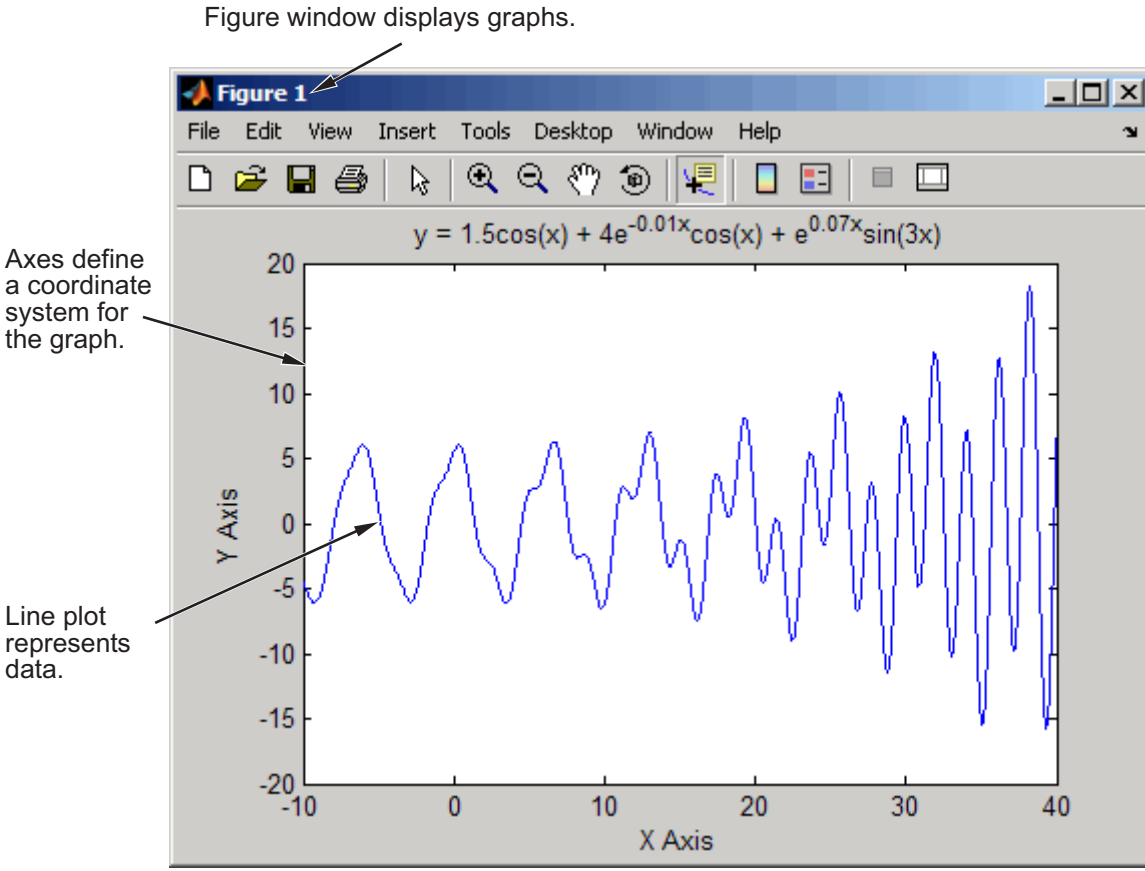
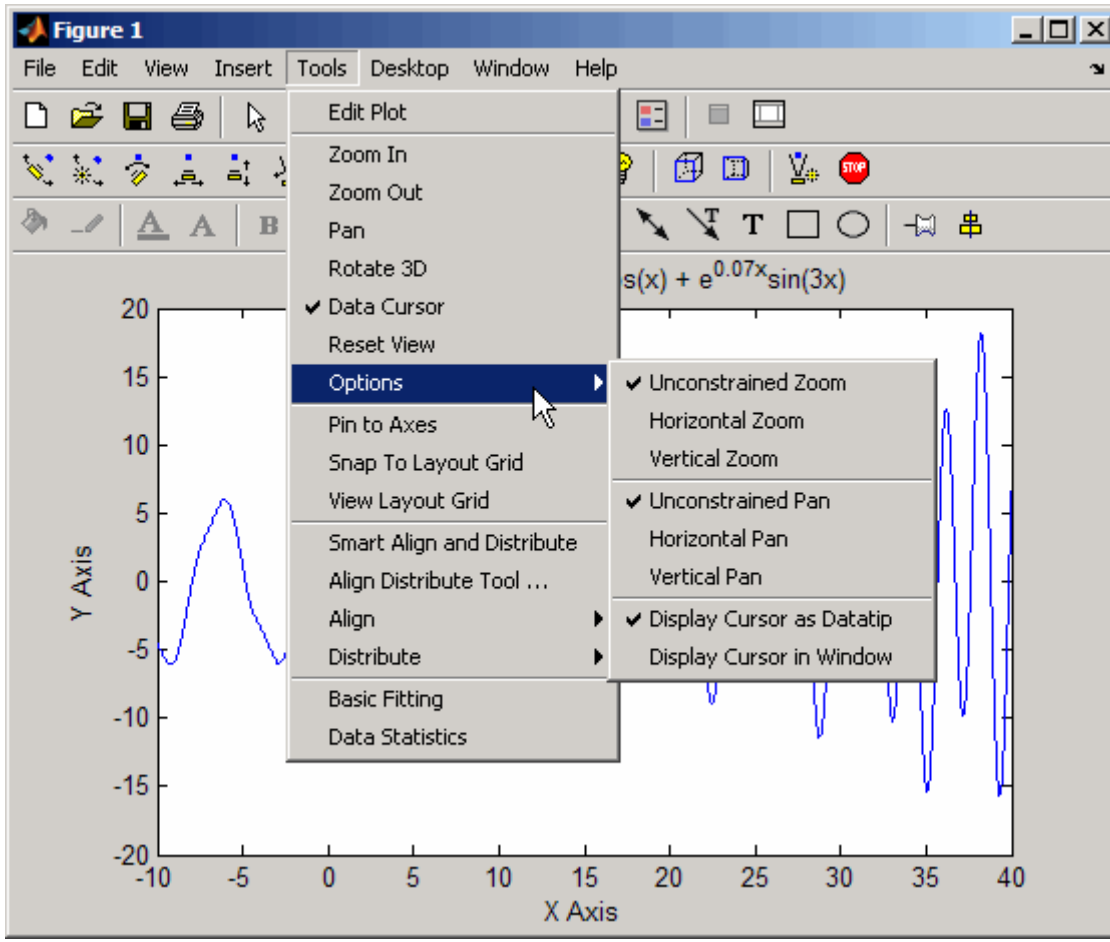


Figure Tools

The figure is equipped with sets of tools that operate on graphs. The figure **Tools** menu provides access to many graph tools, as this view of the **Options** submenu illustrates. Many of the options shown here are also present as

context menu items for individual tools such as zoom and pan. The figure also shows three figure *toolbars*, discussed in “Figure Toolbars” on page 3-8.



For More Information See “Plots and Plotting Tools” in the MATLAB Graphics documentation or select **Plotting Tools** from the figure **Help** menu.

Accessing the Tools

You can access or remove the figure toolbars and the plotting tools from the **View** menu, as shown in the following picture. Toggle on and off the toolbars you need. Adding a toolbar stacks it beneath the lowest one.

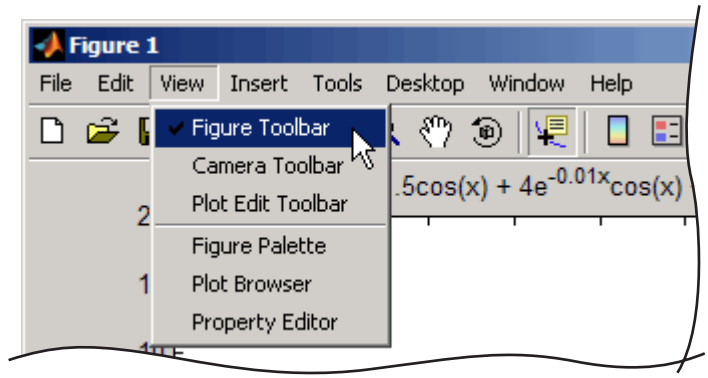
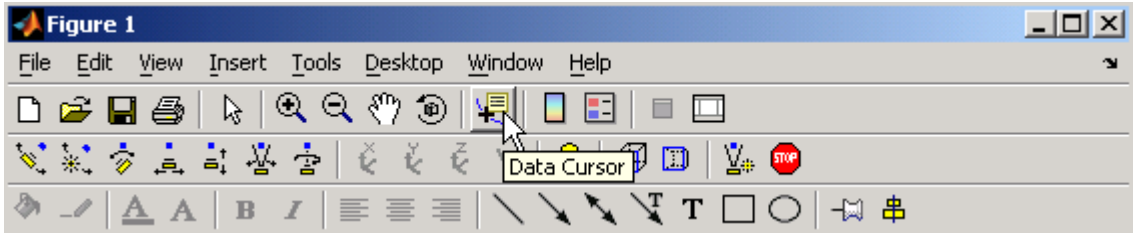


Figure Toolbars

Figure toolbars provide easy access to many graph modification features. There are three toolbars. When you place the cursor over a particular tool, a text box pops up with the tool name. The following picture shows the three toolbars displayed with the cursor over the **Data Cursor** tool.



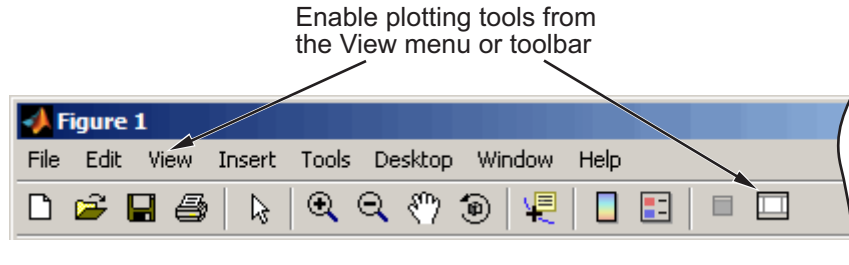
For More Information See “Anatomy of a Graph” in the MATLAB Graphics documentation.

Plotting Tools

Plotting tools are attached to figures and create an environment for creating graphs. These tools enable you to do the following:

- Select from a wide variety of graph types.
- Change the type of graph that represents a variable.
- See and set the properties of graphics objects.
- Annotate graphs with text, arrows, etc.
- Create and arrange subplots in the figure.
- Drag and drop data into graphs.

Display the plotting tools from the **View** menu or by clicking the **Show Plot Tools** icon in the figure toolbar, as shown in the following picture.



You can also start the plotting tools from the MATLAB prompt:

```
plottools
```

The plotting tools are made up of three independent GUI components:

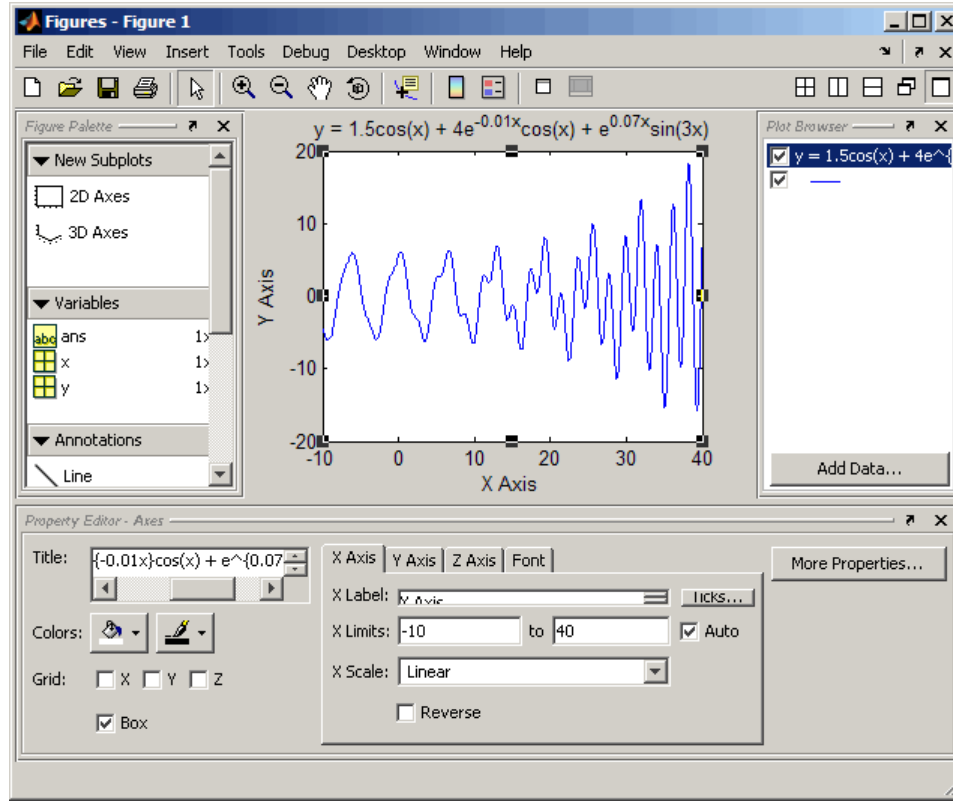
- Figure Palette — Specify and arrange subplots, access workspace variables for plotting or editing, and add annotations.
- Plot Browser — Select objects in the graphics hierarchy, control visibility, and add data to axes.
- Property Editor — Change key properties of the selected object. Click **More Properties** to access all object properties with the Property Inspector.

You can also control these components from the MATLAB Command Window, by typing the following:

```
figurepalette
plotbrowser
propertyeditor
```

See the reference pages for `plottools`, `figurepalette`, `plotbrowser`, and `propertyeditor` for information on syntax and options.

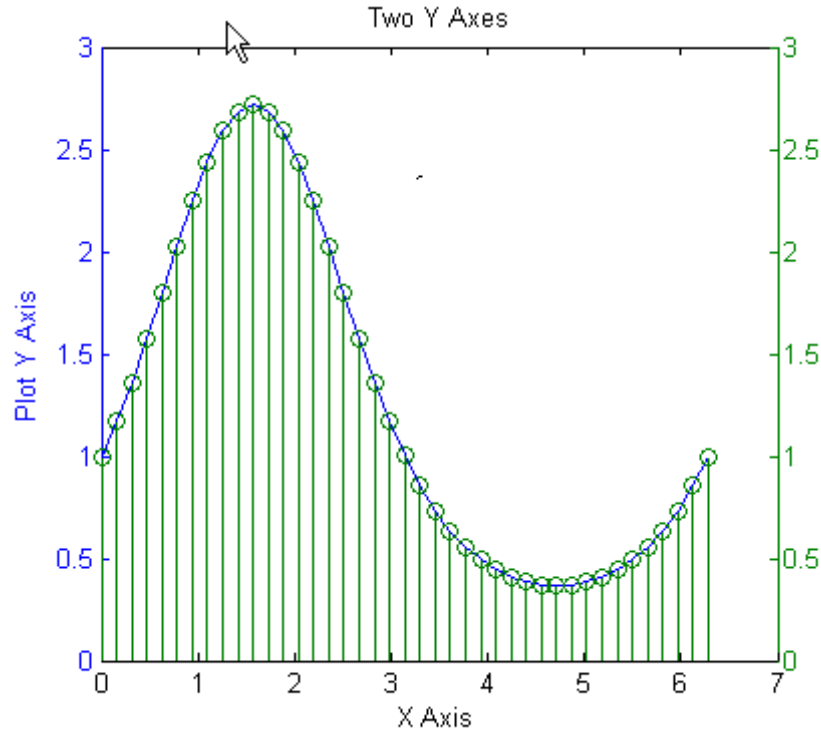
The following picture shows a figure with all three plotting tools enabled.



Using Plotting Tools and MATLAB Code

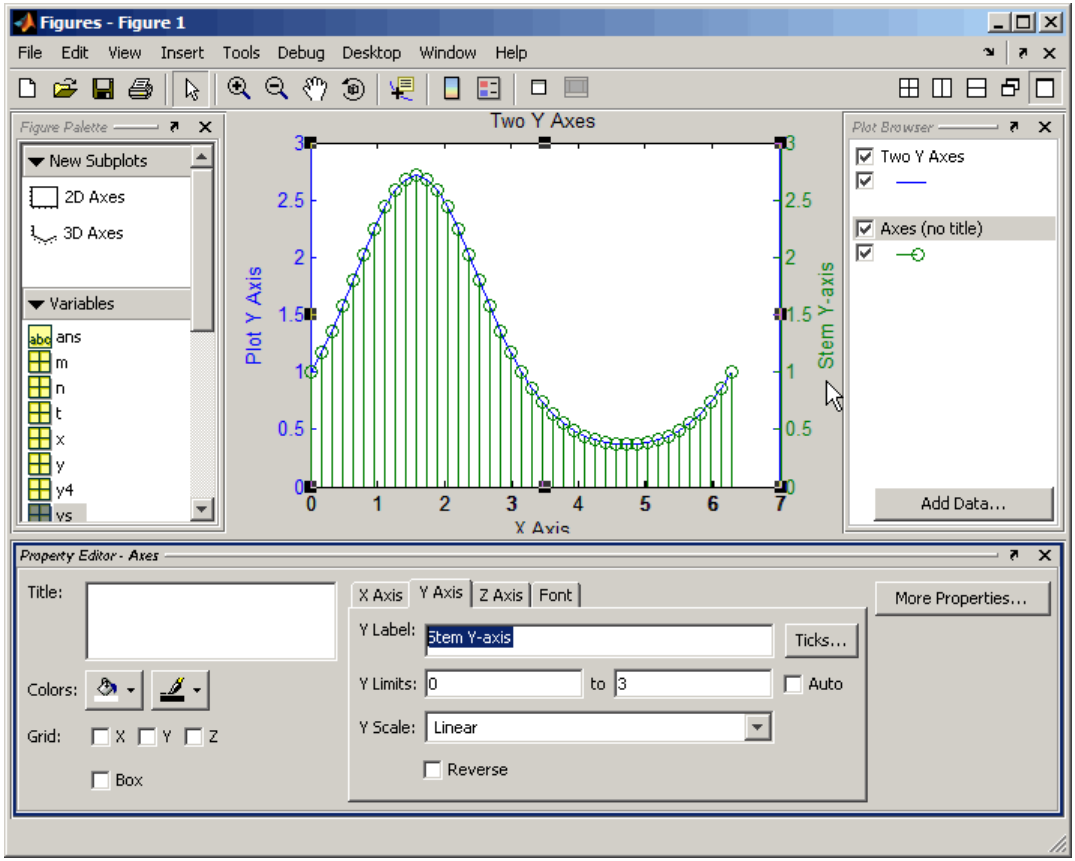
You can enable the plotting tools for any graph, even one created using MATLAB commands. For example, suppose you type the following code to create a graph:

```
t = 0:pi/20:2*pi;
y = exp(sin(t));
plotyy(t,y,t,y,'plot','stem')
xlabel('X Axis')
ylabel('Plot Y Axis')
title('Two Y Axes')
```



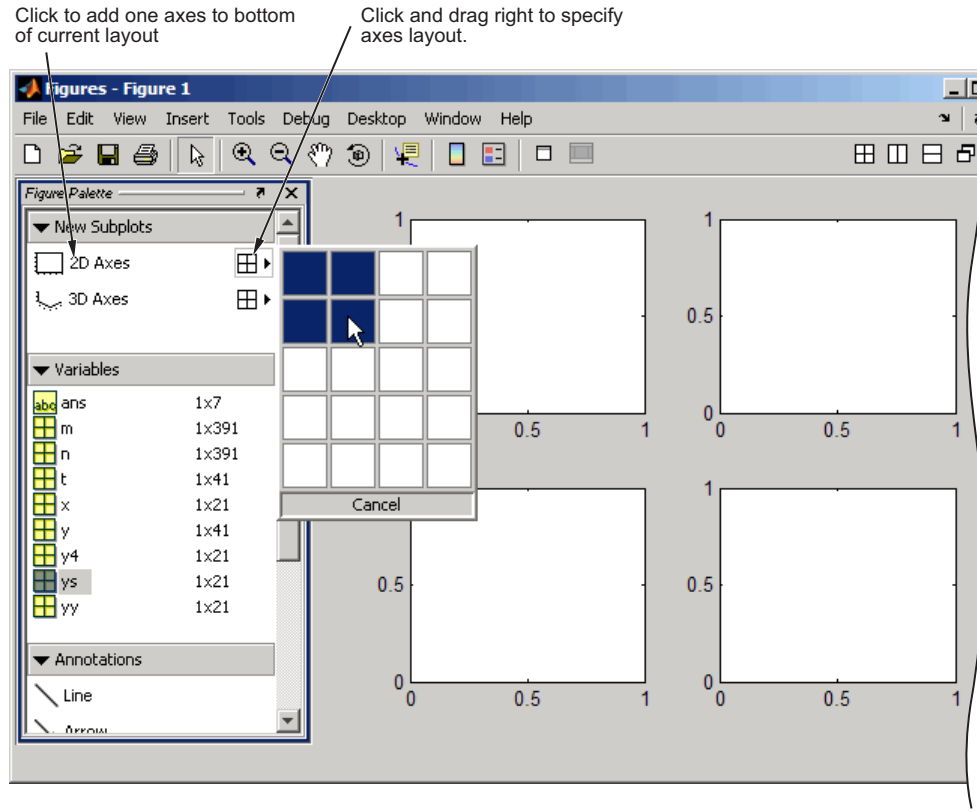
This graph contains two *y*-axes, one for each plot type (a lineseries and a stemseries). The plotting tools make it easy to select any of the objects that the graph contains and modify their properties.

For example, adding a label for the *y*-axis that corresponds to the stem plot is easily accomplished by selecting that axes in the Plot Browser and setting the **Y Label** property in the Property Editor (if you do not see that text field, stretch the Figures window to make it taller).



Arranging Graphs Within a Figure

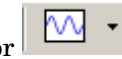
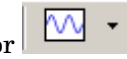
You can place a number of axes within a figure by selecting the layout you want from the Figure Palette. For example, the following picture shows how to specify four 2-D axes in the figure.



Select the axes you want to target for plotting. You can also use the `subplot` function to create multiple axes.

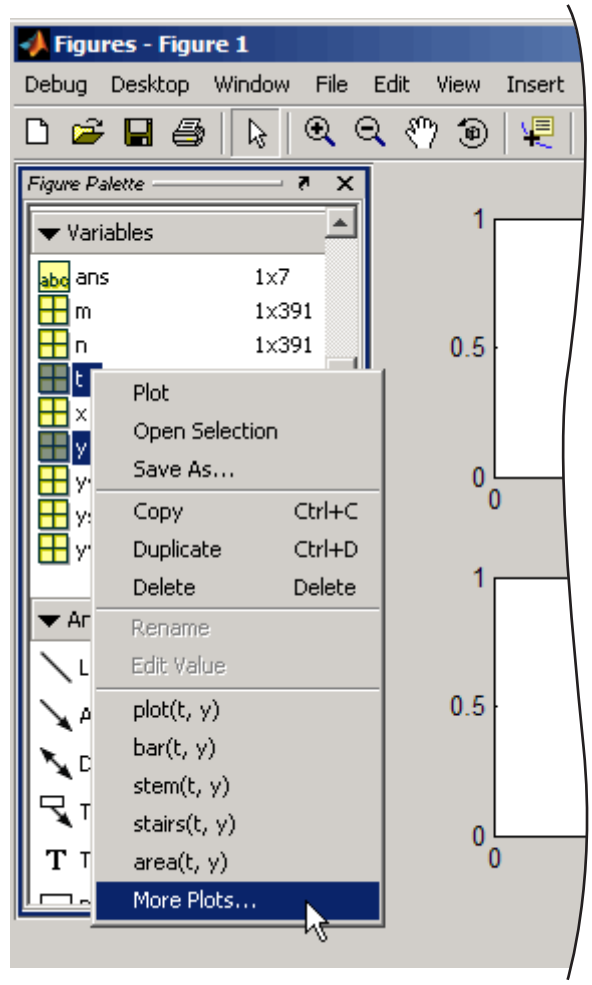
Choosing a Type of Graph to Plot

The many kinds of 2-D and 3-D graphs that MATLAB can make are described in “Types of Plots Available in MATLAB” in the MATLAB Graphics documentation. Almost all plot types are itemized, described, and illustrated by a tool called the Plot Catalog. You can use the Plot Catalog to browse graph types, choose one to visualize your selected variables, and then create it in the current or a new figure window. You can access the Plot Catalog by selecting one or more variables, as follows:

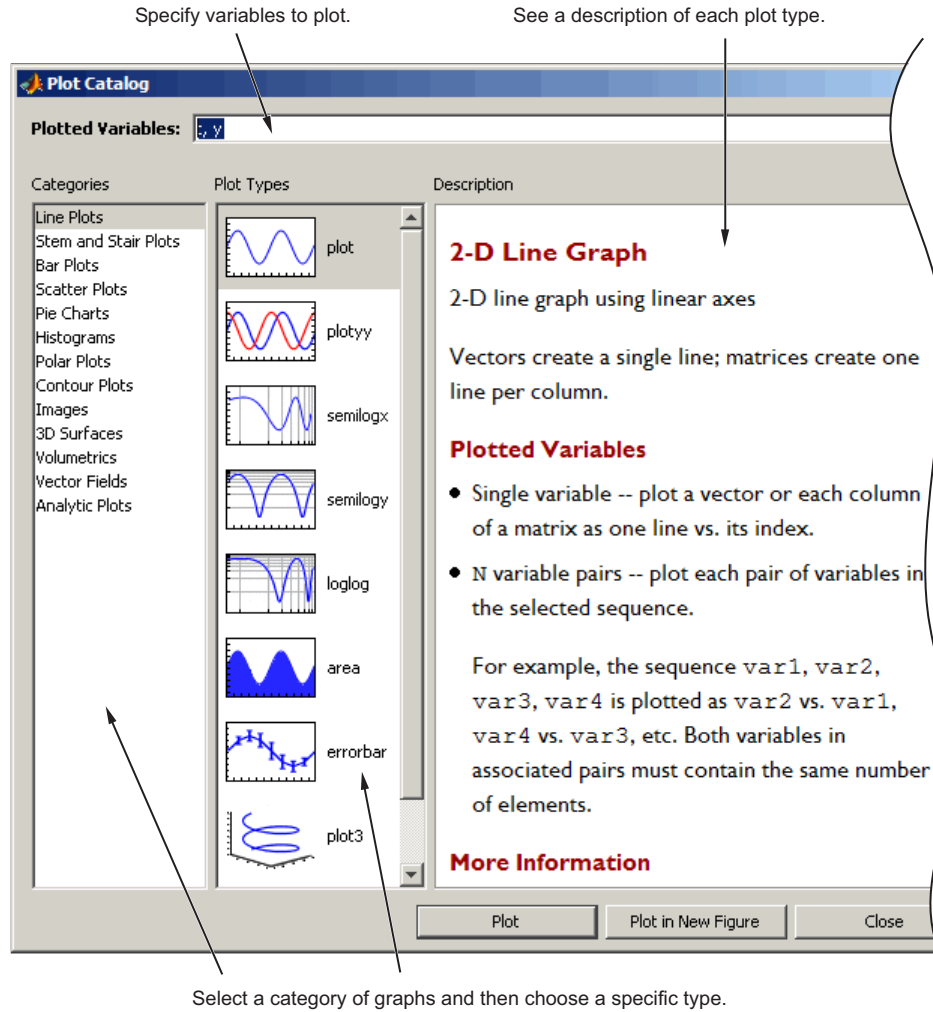
- In the Figure Palette, right-click a selected variable and choose **More Plots** from the context menu
- In the Workspace Browser, right-click a selected variable and choose **More Plots** from the context menu, or click the plot selector  tool and choose **More Plots** from its menu
- In the Array Editor, select the values you want to graph, click the plot selector  tool and choose **More Plots** from its menu

The icon on the plot selector tool represents a graph type, and changes depending on the type and dimensionality of the data you select. It is disabled if no data or non-numeric data is selected.

The following illustration shows how you can open the plot catalog from the Figure Palette:



MATLAB displays the Plot Catalog in a new, undocked window with the selected variables ready to plot, after you select a plot type and click **Plot** or **Plot in New Figure**. You can override the selected variables by typing other variable names or MATLAB expressions in the **Plotted Variables** edit field.


Specify variables to plot.
See a description of each plot type.
Plot Catalog
Plotted Variables:
Categories
Plot Types
Description
Line Plots
Stem and Stair Plots
Bar Plots
Scatter Plots
Pie Charts
Histograms
Polar Plots
Contour Plots
Images
3D Surfaces
Volumetrics
Vector Fields
Analytic Plots
plot
plotyy
semilogx
semilogy
loglog
area
errorbar
plot3
2-D Line Graph
2-D line graph using linear axes
Vectors create a single line; matrices create one line per column.
Plotted Variables
Single variable -- plot a vector or each column of a matrix as one line vs. its index.
N variable pairs -- plot each pair of variables in the selected sequence.
For example, the sequence var1, var2, var3, var4 is plotted as var2 vs. var1, var4 vs. var3, etc. Both variables in associated pairs must contain the same number of elements.
More Information
Plot
Plot in New Figure
Close
Select a category of graphs and then choose a specific type.

Editing Plots

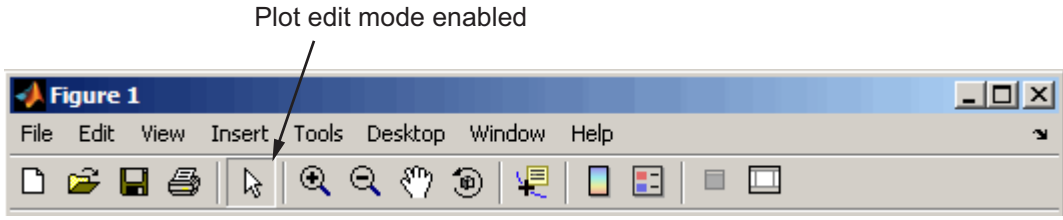
In this section...

Plot Edit Mode

Plot edit mode lets you select specific objects in a graph and enables you to perform point-and-click editing of most of them.

Enabling Plot Edit Mode

To enable plot edit mode, click the arrowhead in the figure toolbar:

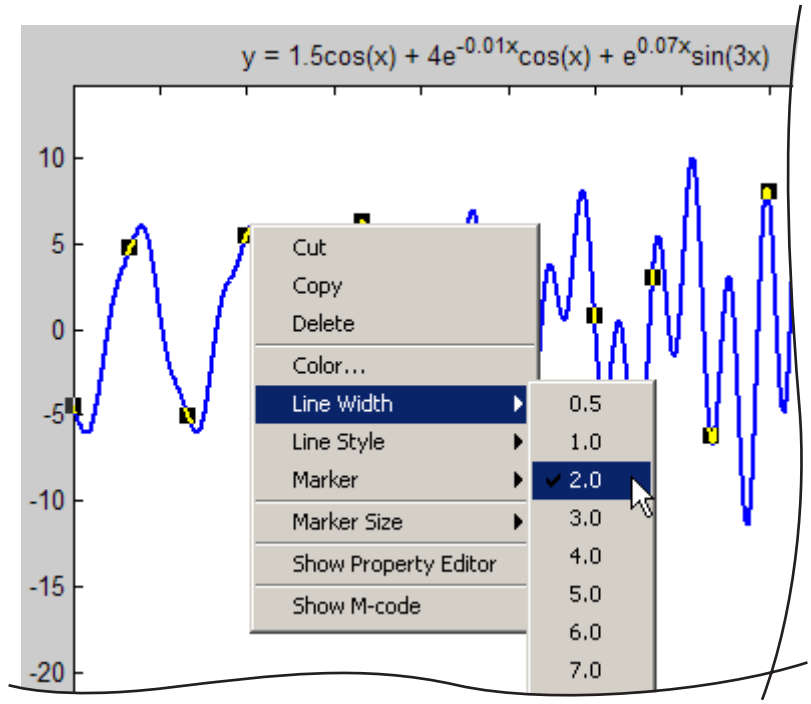


You can also select **Edit Plot** from the figure **Tools** menu.

Setting Object Properties

After you have enabled plot edit mode, you can select objects by clicking them in the graph. Selection handles appear and indicate that the object is selected. Select multiple objects using **Shift**+click.

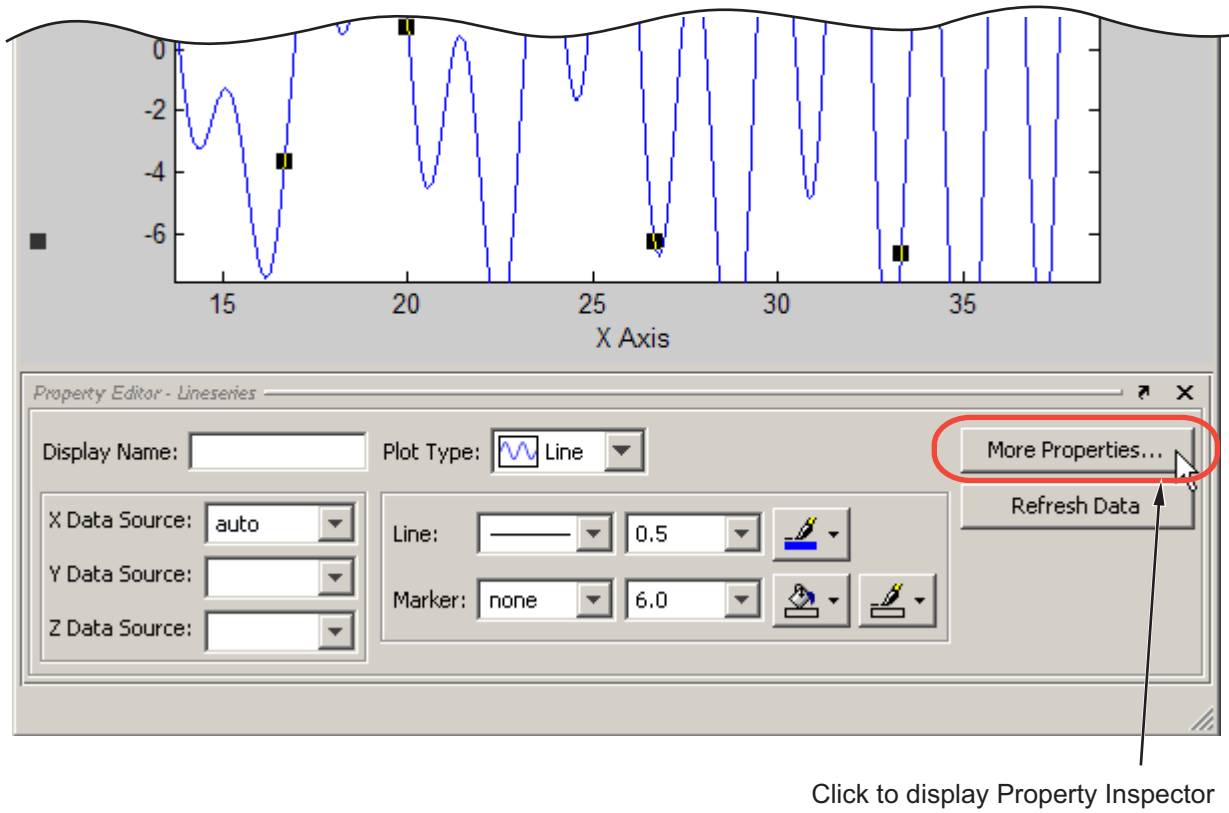
Right-click with the pointer over the selected object to display the object’s context menu:



The context menu provides quick access to the most commonly used operations and properties.

Using the Property Editor

In plot edit mode, double-clicking an object in a graph opens the Property Editor GUI with that object's major properties displayed. The Property Editor provides access to the most used object properties. It is updated to display the properties of whatever object you select.



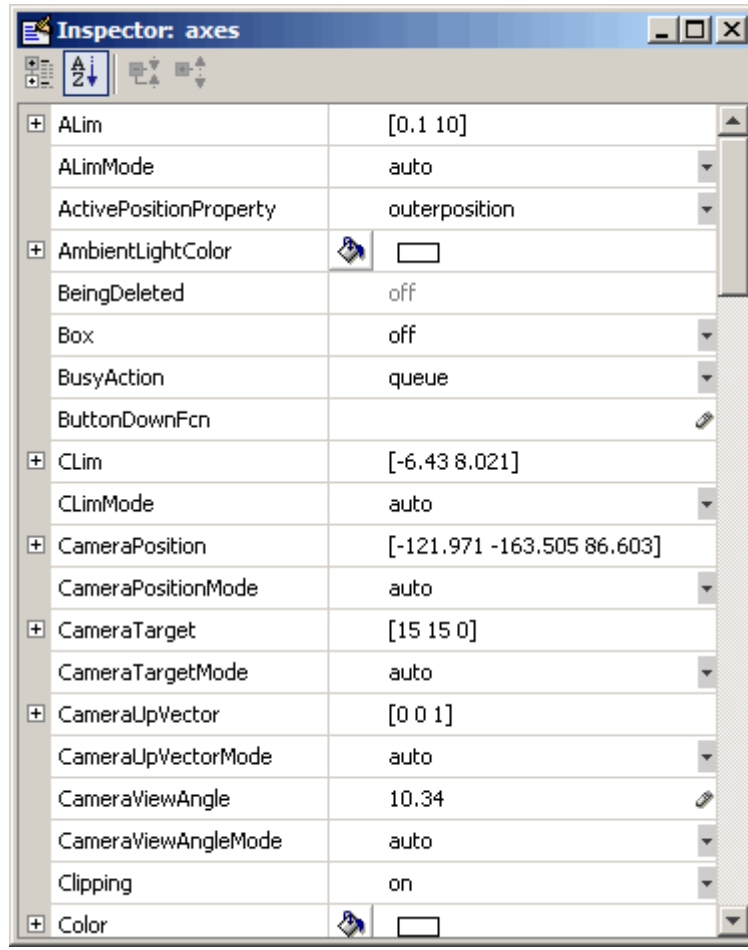
Accessing Properties with the Property Inspector

The Property Inspector is a tool that enables you to access most of the properties of Handle Graphics and other MATLAB objects. If you do not find the property you want to set in the Property Editor, click the **More Properties** button to display the Property Inspector. You can also use the `inspect` command to start the Property Inspector. For example, to inspect the properties of the current axes, type

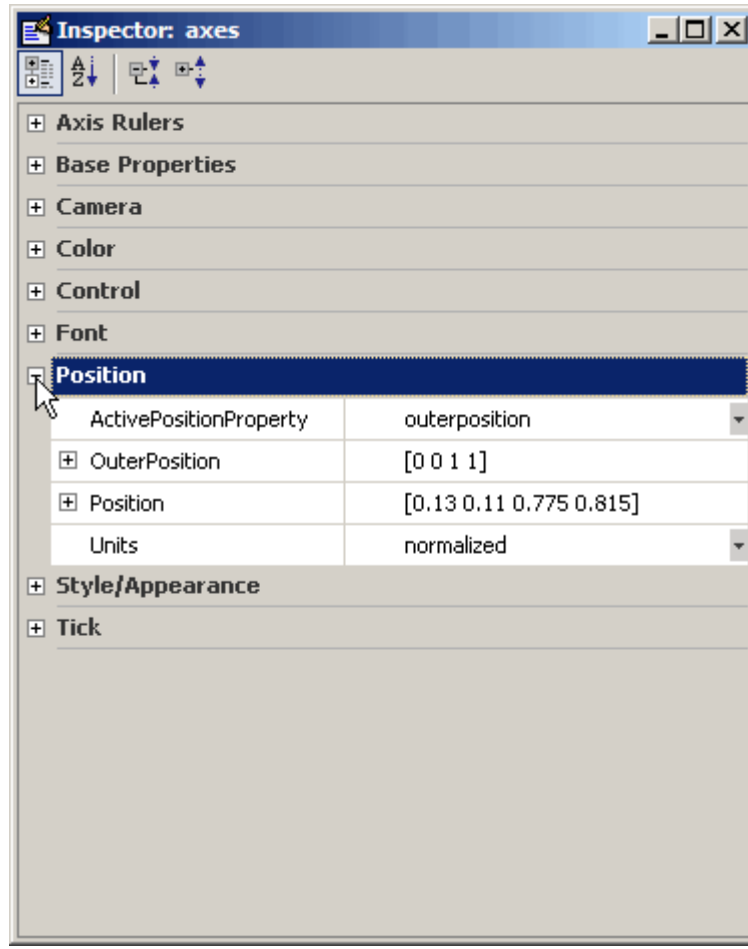
```
inspect(gca)
```

The following picture shows the Property Inspector displaying the properties of a graph's axes. It lists each property and provides a text field or other appropriate device (such as a color picker) from which you can set the value of the property.

As you select different objects, the Property Inspector is updated to display the properties of the current object.



The Property Inspector lists properties alphabetically by default. However, you can group Handle Graphics objects, such as axes, by categories which you can reveal or close in the Property Inspector. To do this, click the  icon at the upper left, then click the + next to the category you want to expand. For example, to see the position-related properties, click the + to the left of the **Position** category.



The **Position** category opens and the + changes to a - to indicate that you can collapse the category by clicking it.

Using Functions to Edit Graphs

If you prefer to work from the MATLAB command line, or if you are creating an M-file, you can use MATLAB commands to edit the graphs you create. You can use the `set` and `get` commands to change the properties of the objects in a graph. For more information about using graphics commands, see "Handle Graphics" on page 3-74.

Some Ways to Use MATLAB Plotting Tools

In this section...

Plotting Two Variables with Plotting Tools

Suppose you want to graph the function $y = x^3$ over the x domain -1 to 1. The first step is to generate the data to plot.

It is simple to evaluate a function because MATLAB can distribute arithmetic operations over all elements of a multivalued variable.

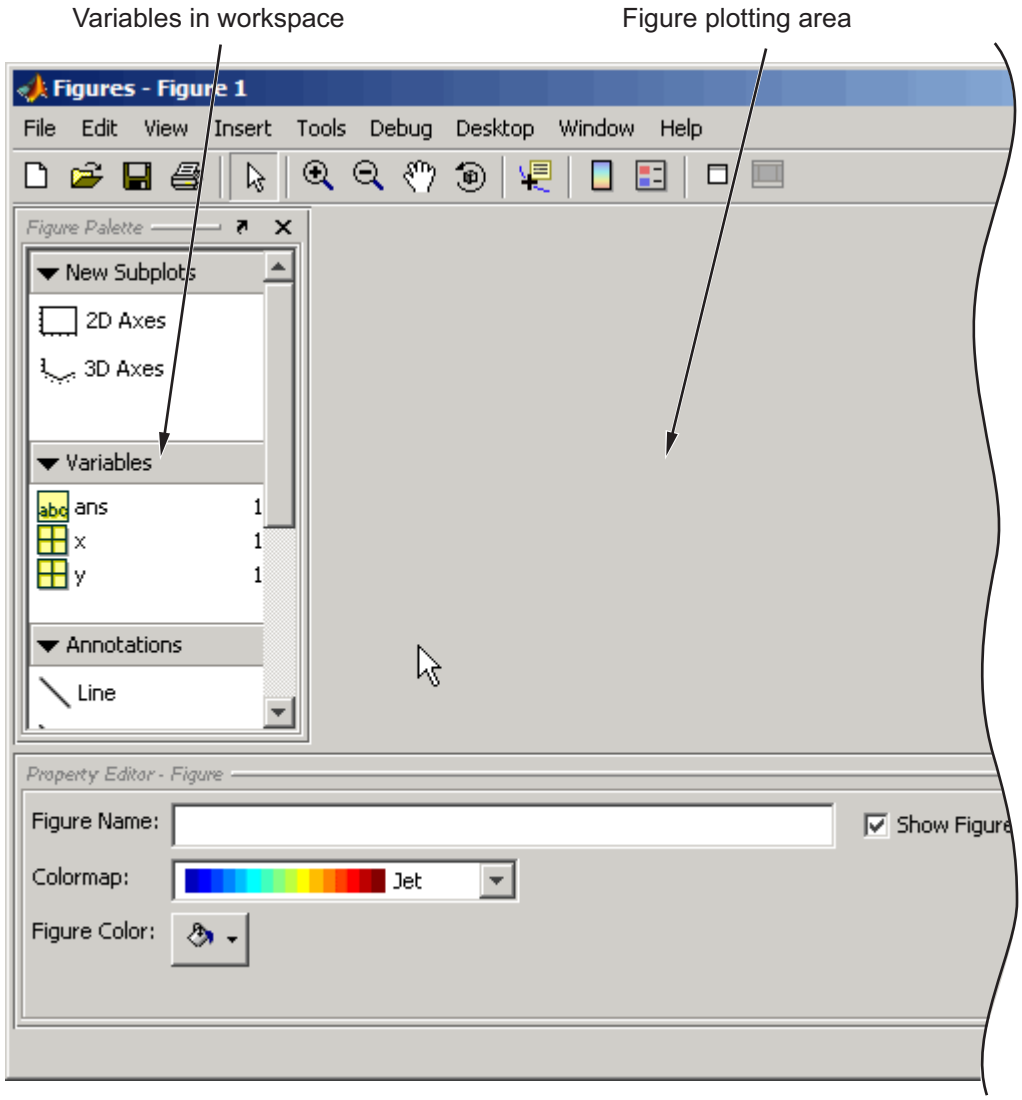
For example, the following statement creates a variable `x` that contains values ranging from -1 to 1 in increments of 0.1 (you could also use the `linspace` function to generate data for `x`). The second statement raises each value in `x` to the third power and stores these values in `y`:

```
x = -1:.1:1; % Define the range of x
y = x.^3;    % Raise each element in x to the third power
```

Now that you have generated some data, you can plot it using the MATLAB plotting tools. To start the plotting tools, type

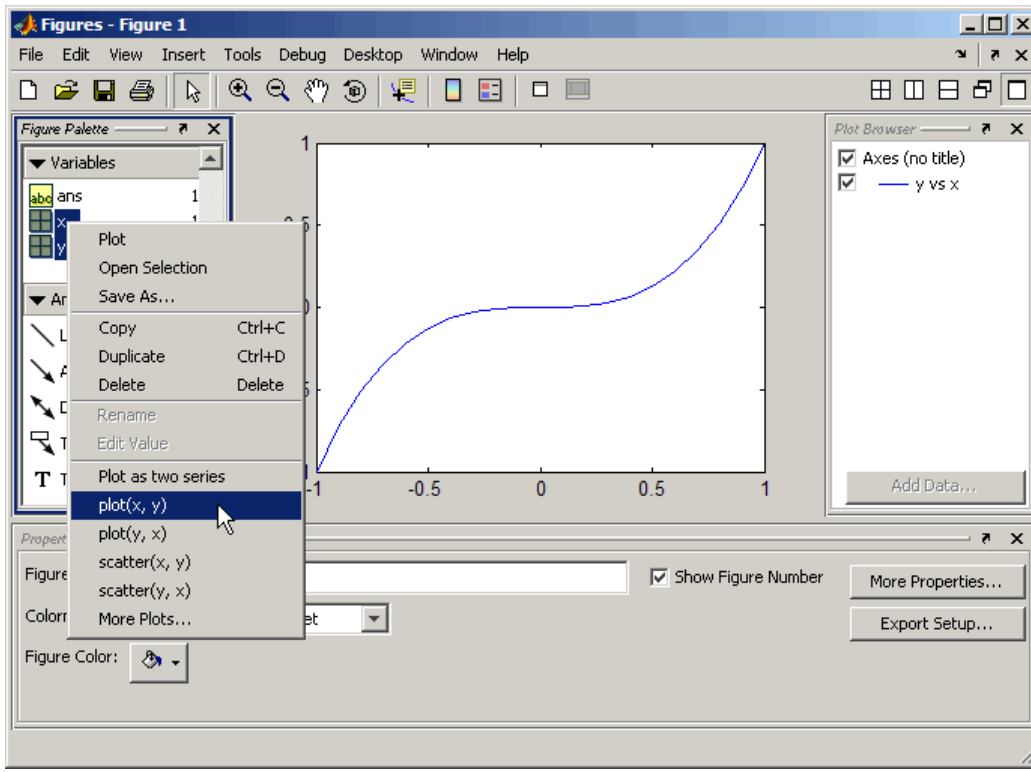
```
plottools
```

MATLAB displays a figure with plotting tools attached.


Variables in workspace
Figure plotting area
Figures - Figure 1
File Edit View Insert Tools Debug Desktop Window Help
Figure Palette
New Subplots
2D Axes
3D Axes
Variables
ans 1
x 1
y 1
Annotations
Line
Property Editor - Figure
Figure Name:
Show Figure
Colormap:
Jet
Figure Color:

Note When you invoke `plottools`, the set of plotting tools you see and their relative positions depend on how they were configured the last time you used them. Also, sometimes when you dock and undock figures with plotting tools attached, the size or proportions of the various components can change, and you may need to resize one or more of the tool panes.

A simple line graph is a suitable way to display x as the independent variable and y as the dependent variable. To do this, select both variables (click to select, and then **Shift**+click to select again), and then right-click to display the context menu.

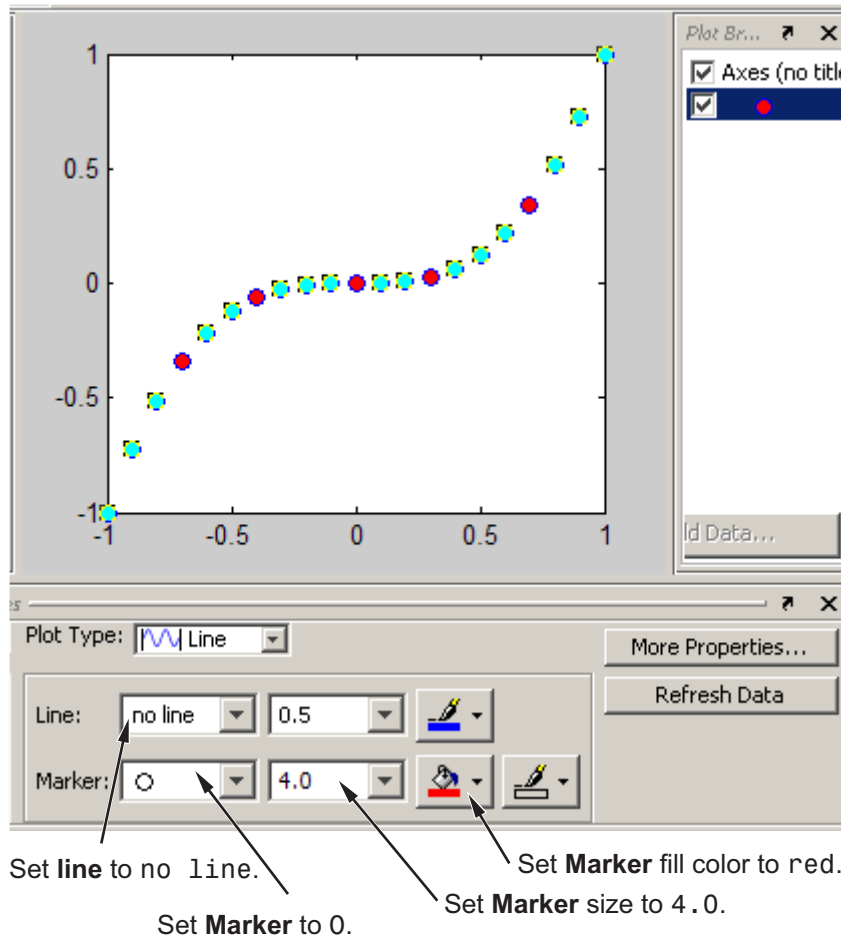


Select **plot(x, y)** from the menu. MATLAB creates the line graph in the figure area. The black squares indicate that the line is selected and you can edit its properties with the Property Editor.

Changing the Appearance of Lines and Markers

Next change the line properties so that the graph displays only the data point. Use the Property Editor to set following properties:

- Line to no line
- Marker to o (circle)
- Marker size to 4.0
- Marker fill color to red



Adding More Data to the Graph

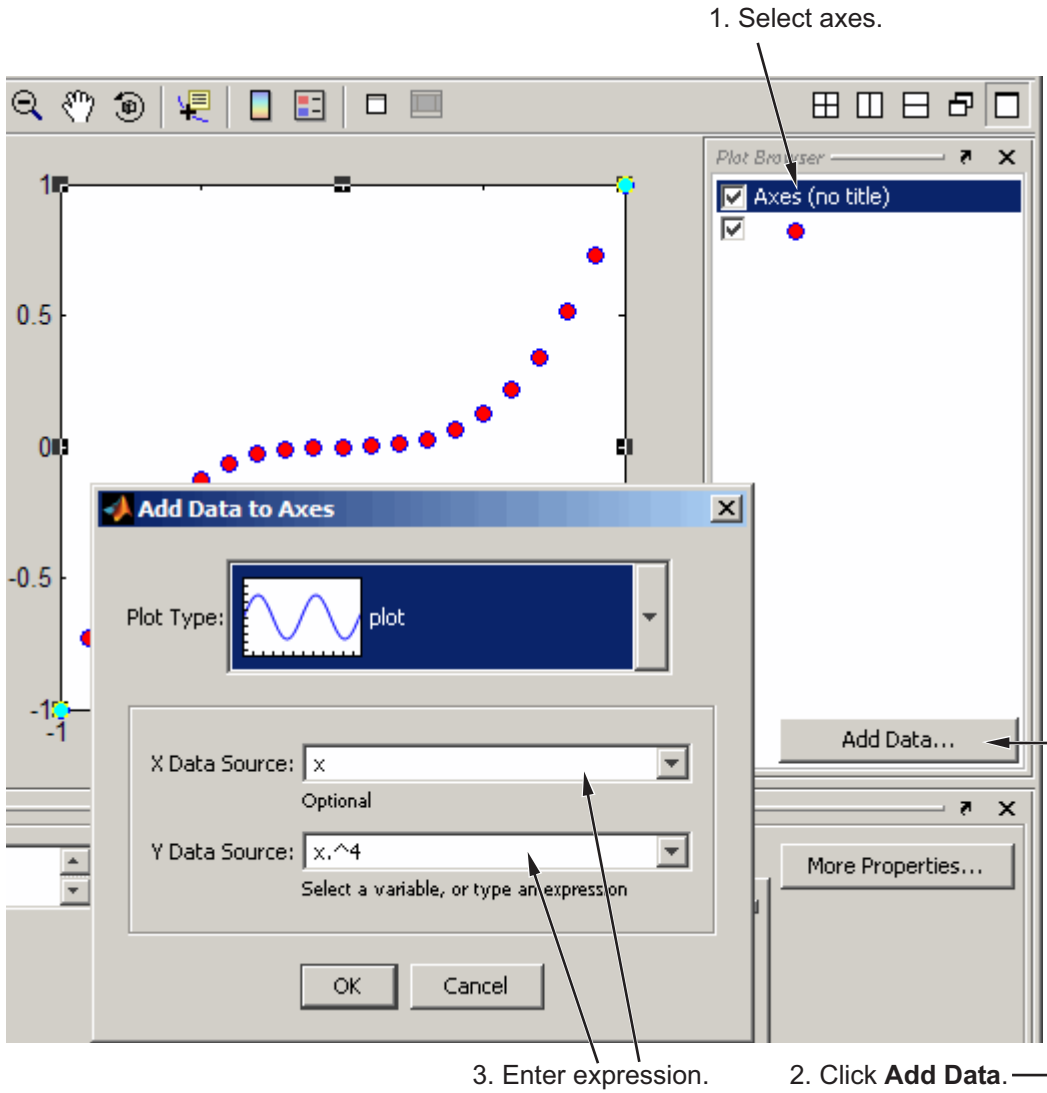
You can add more data to the graph by defining more variables or by specifying an expression that MATLAB uses to generate data for the plot. This second approach makes it easy to explore variations of the data already plotted.

To add data to the graph, select the axes in the Plot Browser and click the **Add Data** button. When you are using the plotting tools, MATLAB always adds data to the existing graph, instead of replacing the graph, as it would

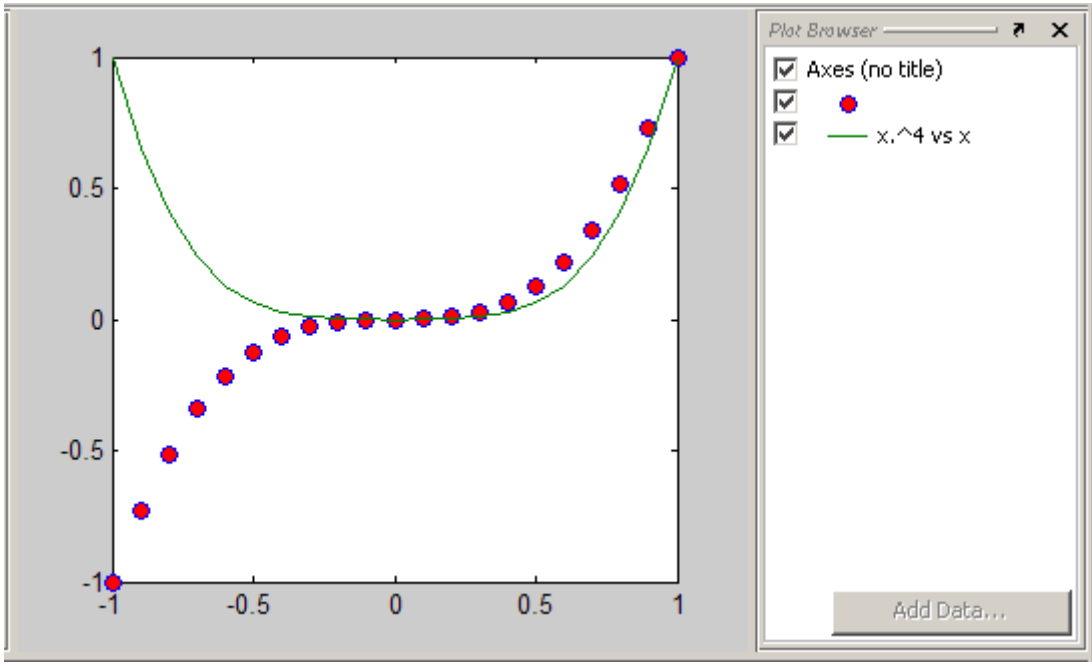
if you issued repeated plotting commands. That is, the plotting tools are in a `hold on` state.

To add data using the Plot Browser:

1. Click the **Edit Plot** tool  .
2. Select the axes to which you wish to add data; handles appear around it.
3. Click the **Add Data** button in the Plot Browser; the Add Data to Axes dialog box opens.
4. Select a plot type from the **Plot Type** drop-down menu.
5. Select a variable or type an expression for **X Data Source**.
6. Select a variable or type an expression for **Y Data Source**.
7. Click **OK**; a plot of the data you specified is added to the axes.



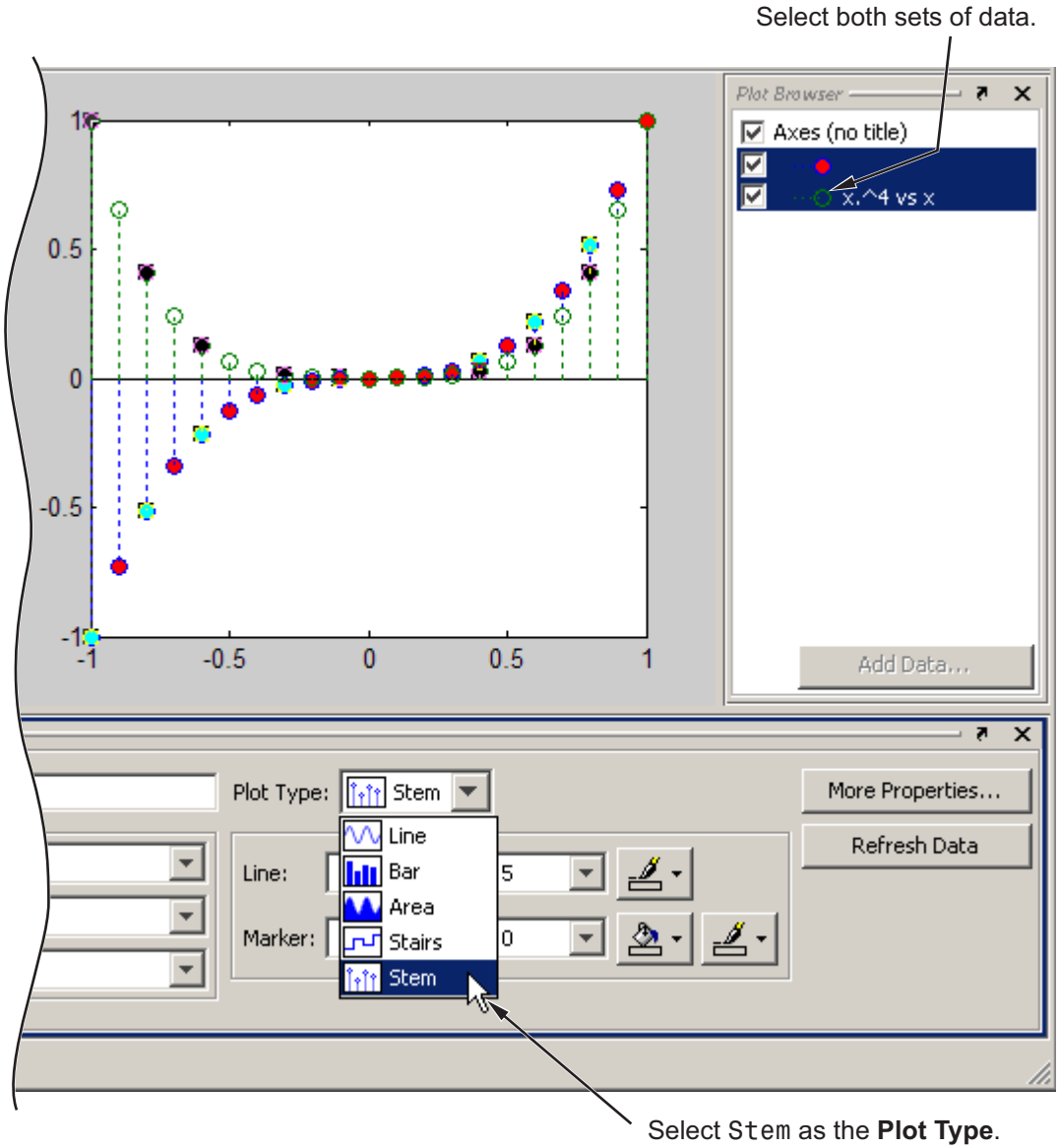
The picture above shows how to use the Add Data to Axes dialog box to create a line plot of $y = x^4$, which is added to the existing plot of $y = x^3$. The resulting plot is shown as follows with the Plot Browser:



Changing the Type of Graph

The plotting tools enable you to easily view your data with a variety of plot types. The following picture shows the same data as above converted to stem plots. To change the plot type,

1. Select both plotted series in the Plot Browser or **Shift**+click to select them in the plot itself.
2. Select short dashes from the **Line** drop-down menu in the Property Inspector; the line type of both series changes.
3. Select **Stem** from the **Plot Type** menu.


Select both sets of data.
Plot Browser
Axes (no title)
x.^4 vs x
Add Data...
Plot Type:
Stem
Line
Bar
Area
Stairs
Stem
More Properties...
Refresh Data
Line:
Marker:
Select Stem as the Plot Type.

Modifying the Graph Data Source

You can link graph data to variables in your workspace. When you change the values contained in the variables, you can then update the graph to use the new data without having to create a new graph. (See also the `refresh` function.)

1. Define 50 points between -3π and 3π and compute their sines and cosines:

```
x = linspace(-3*pi,3*pi,50);
ys = sin(x);
yc = cos(x);
```

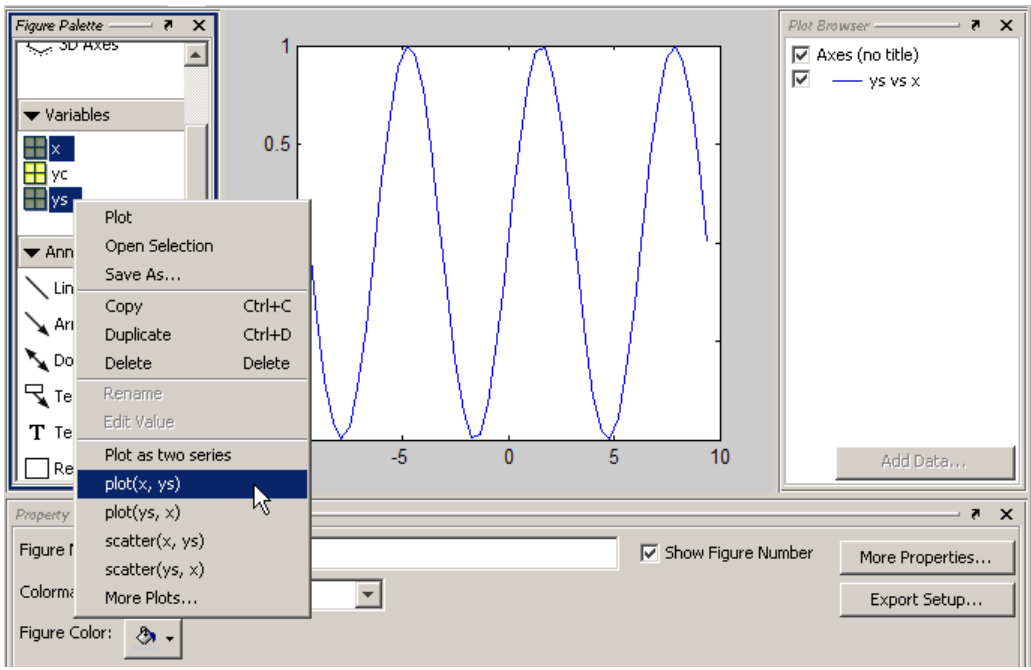
2. Using the plotting tools, create a graph of `ys = sin(x)`:

```
figure
plottools
```

3. In the Figure Palette, alternate-click to select *x* and *ys* in the **Variable** pane.

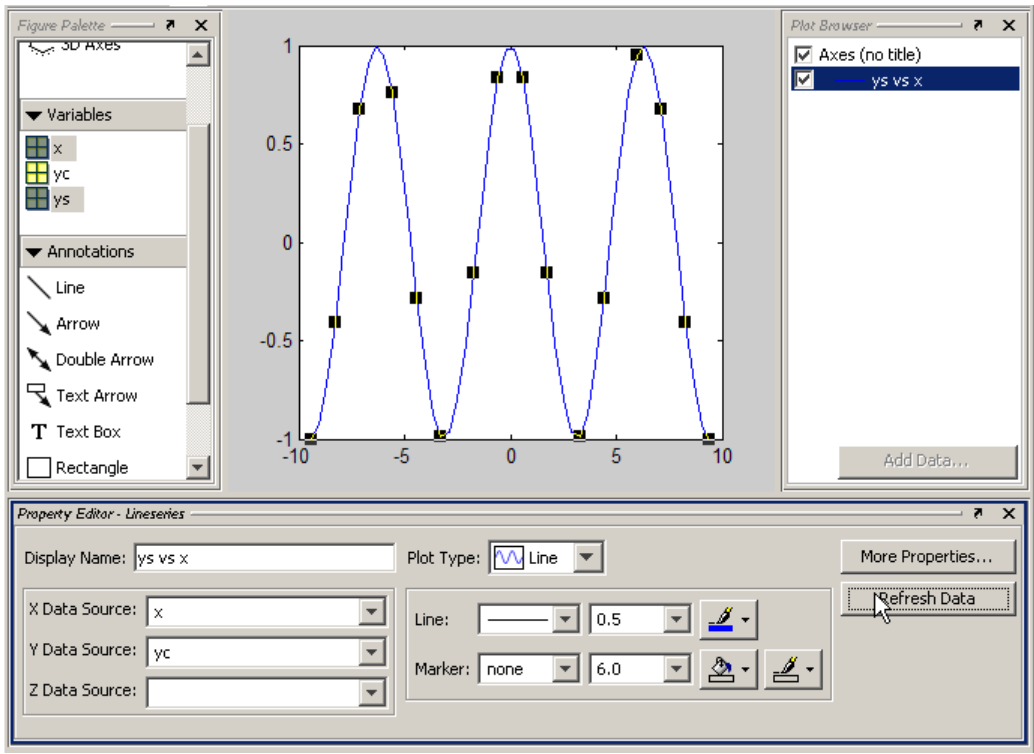
4. Right-click either selected variable and choose **plot(x, ys)** from the context menu, as shown below.

The resulting plot looks like this.



You can use the Property Editor to change the data that this plot displays:

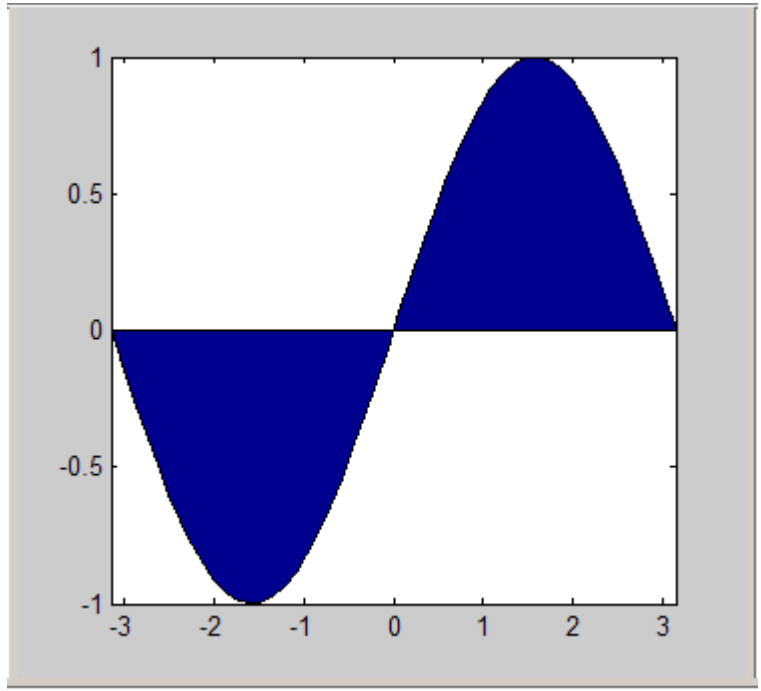
1. Select the line `ys vs x` in the Plot Browser or by clicking it.
2. In the Property Editor, select `yc` in the **Y Data Source** drop-down menu.
3. Click the **Refresh Data** button; the plot will change to display a plot of `yc vs x`.



Providing New Values for the Data Source

MATLAB copies the data that defines the graph from variables in the base workspace (for example, `x` and `y`) to the `XData` and `YData` properties of the plot object (for example, a lineseries). Therefore, in addition to being able to choose new data sources, you can assign new values to workspace variables in the Command Window and click the **Refresh Data** button to update a graph to use the new data.

```
x = linspace(-pi,pi,50); % Define 50 points between -π and π
y = sin(x);
area(x,y)  % Make an area plot of x and y
```



Now recalculate `y` at the command line:

```
y = cos(x)
```

Select the blue line on the plot. Select, `x` as the **X Data Source**, `y` as the **Y Data Source**, and click **Refresh Data**. The graph's `XData` and `YData` are replaced, making the plot look like this.

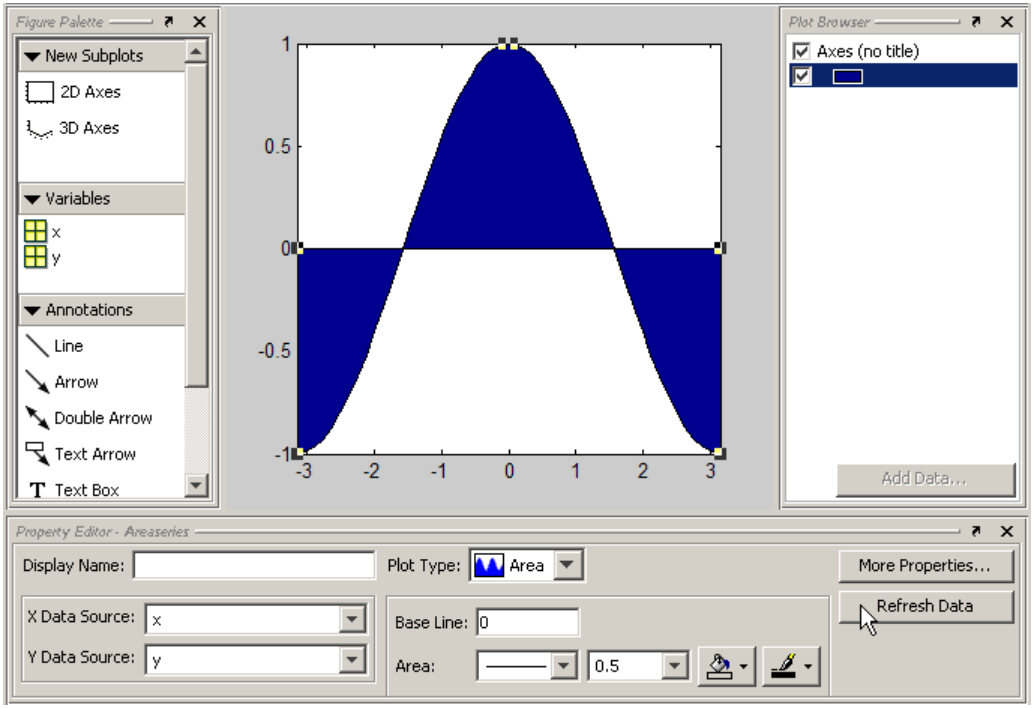


Figure Palette
New Subplots
2D Axes
3D Axes
Variables
x
y
Annotations
Line
Arrow
Double Arrow
Text Arrow
Text Box
1
0.5
0
-0.5
-1
-3
-2
-1
0
1
2
3
Plot Browser
Axes (no title)
Add Data...
Property Editor - Areaseries
Display Name:
Plot Type:
Area
More Properties...
Refresh Data
X Data Source:
x
Y Data Source:
y
Base Line:
0
Area:
0.5

Preparing Graphs for Presentation

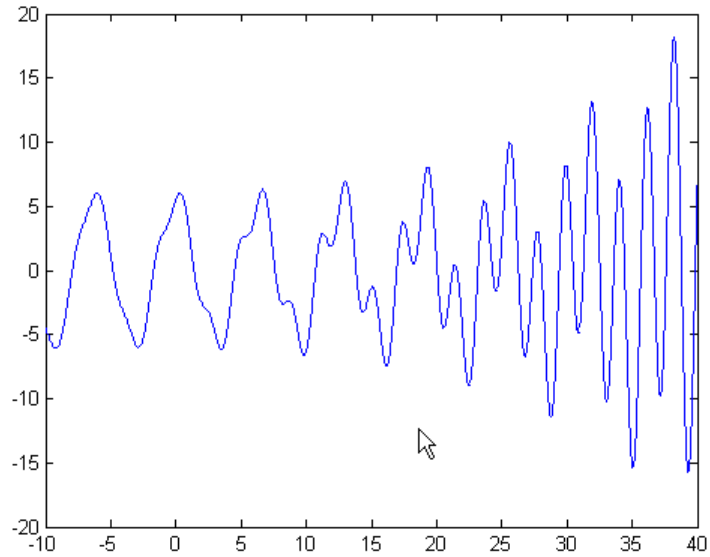
In this section...

Annotating Graphs for Presentation

Suppose you plot the following data and want to create a graph that presents certain information about the data:

```
x = -10:.005:40;
y = [1.5*cos(x)+4*exp(-.01*x).*cos(x)+exp(.07*x).*sin(3*x)];
plot(x,y)
```

This picture shows the graph created by the previous code.

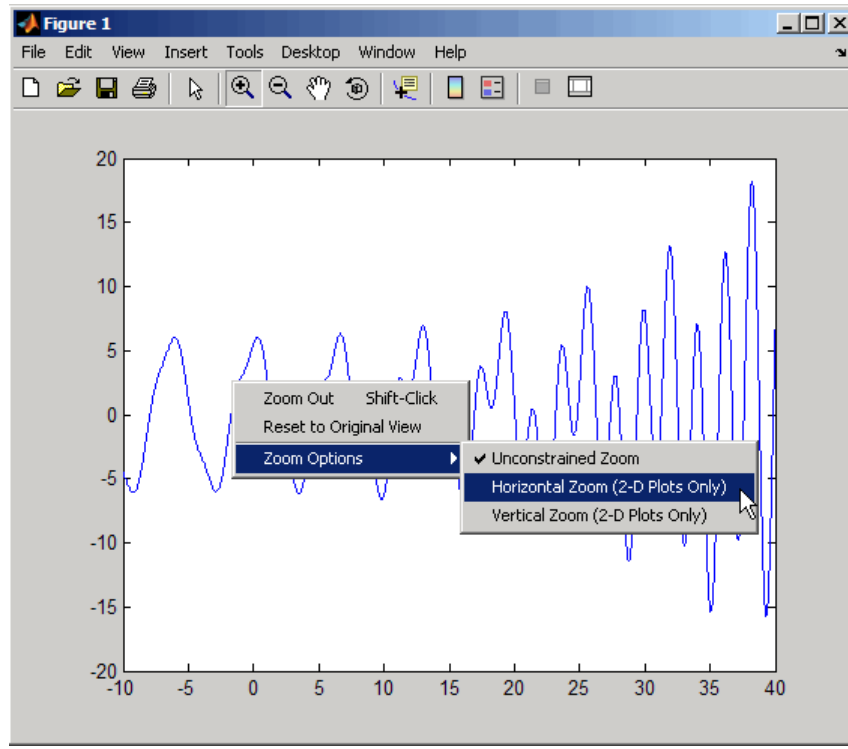


Now suppose you want to save copies of the graph by

- Printing the graph on a local printer so you have a copy for your notebook
- Exporting the graph to an Encapsulated PostScript (EPS) file to incorporate into a word processor document

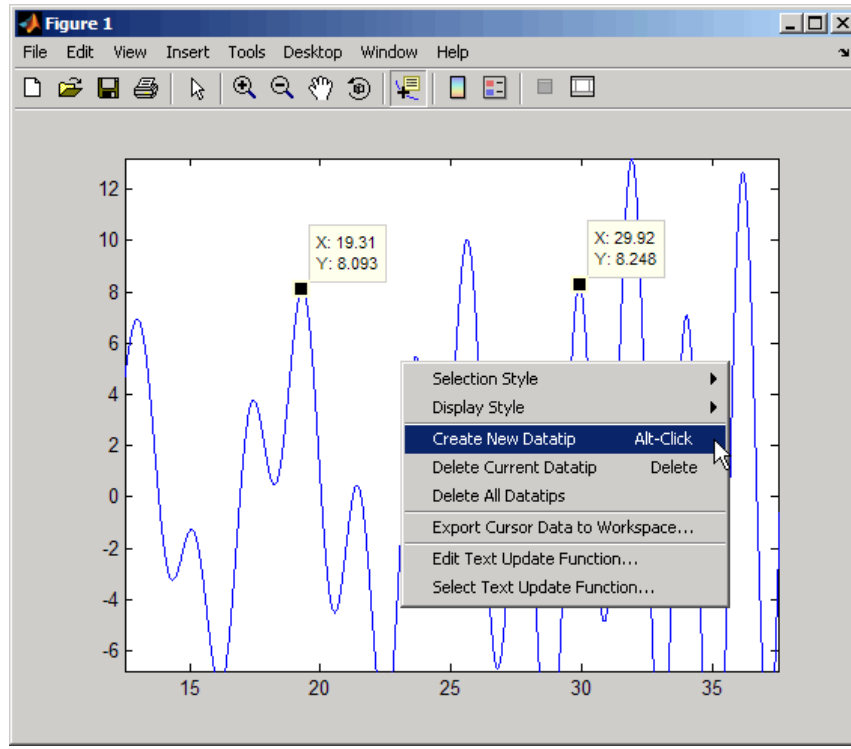
To obtain a better view, zoom in on the graph using horizontal zoom.

Enable zoom mode by clicking the **Zoom** tool on the figure toolbar, and then right-click to display the context menu. Select **Horizontal Zoom (2-D Plots Only)** from **Zoom Options**. Notice that you can reverse your zoom direction by **Shift**+left-clicking, or using the context menu.

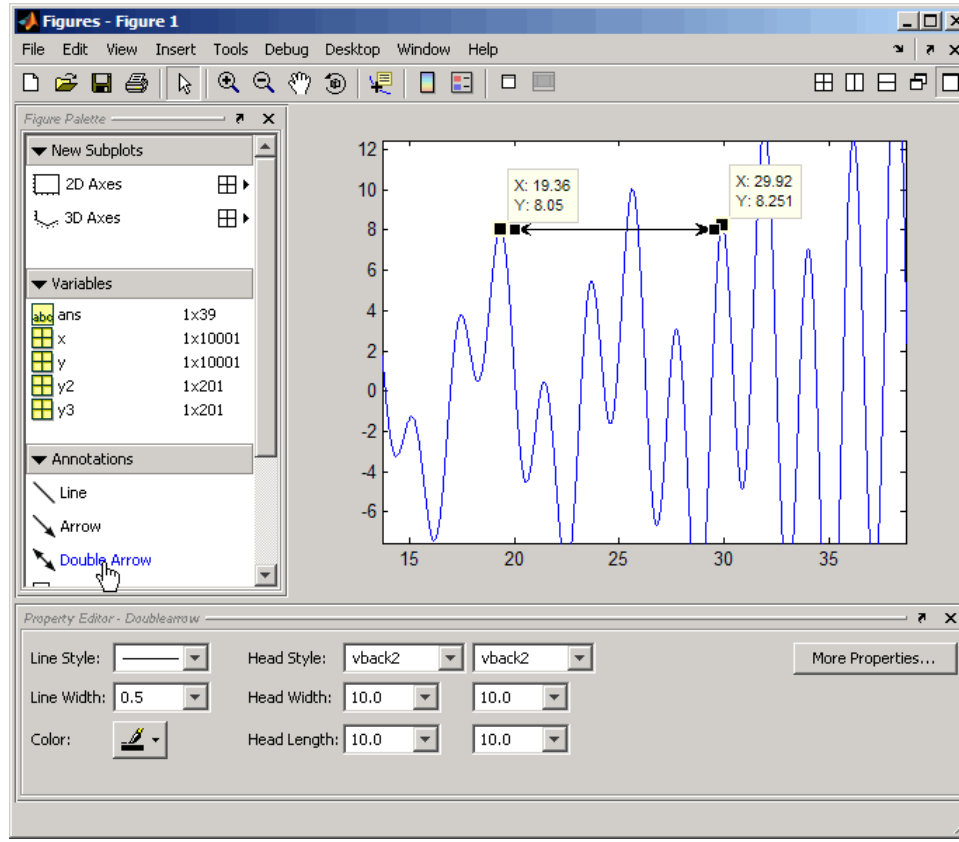


Left-click to zoom in on a region of the graph and use the **Pan** tool to position the points of interest where you want them on the graph.

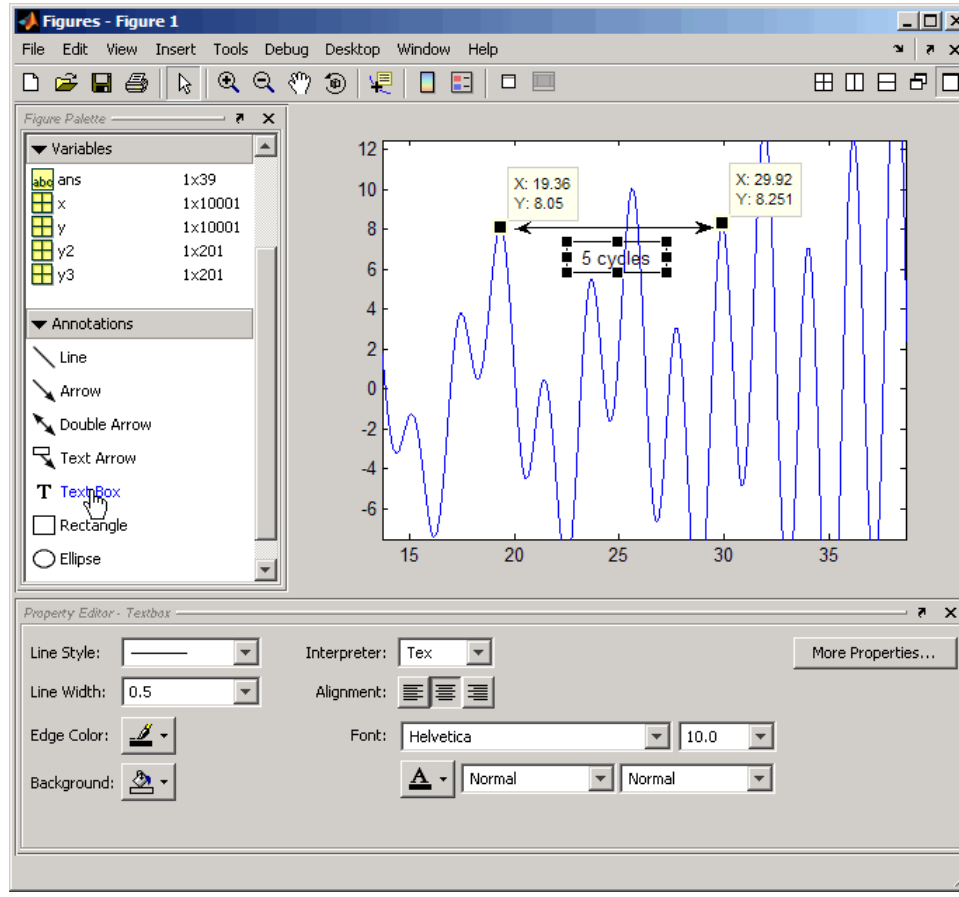
Label some key points with data tips using the **Data Cursor** tool  . Notice that left-clicking the line moves the last datatip you created to where you just clicked. To create a new datatip, press **Alt**+click or use the tool's context menu. See "Data Cursor — Displaying Data Values Interactively" in the MATLAB Graphics documentation for more information on using datatips.



Next use the Figure Palette to annotate the plot. Choose the Double arrow tool in the Annotations section to draw a line between two datatips, as shown below:



Now add a text box, also using the Figure Palette. You may have to scroll to see the text box icon. Drag out a box, and then type into it. You can stretch or shrink the box with its handles, and center the text with the Property Editor while the text box is selected. You can also use the Property Editor to change the text font, size, style, and color, as well as the text box line and background colors.

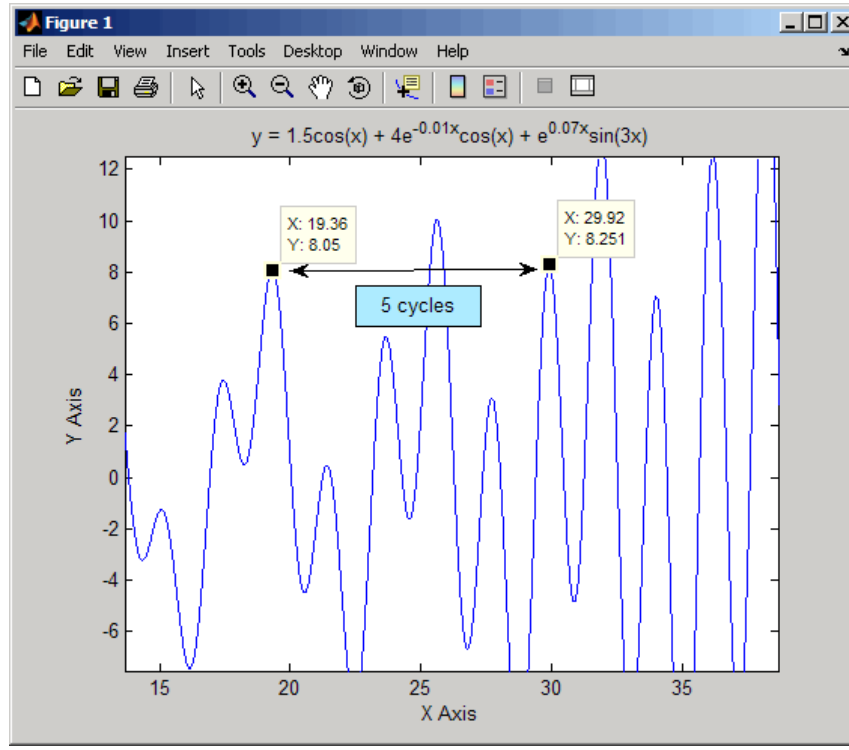


Finally, add text annotations, axis labels, and a title. You can add the title and axis labels using the following commands:

```
title ('y = 1.5cos(x) + 4e^{-0.01x}cos(x) + e^{0.07x}sin(3x)')
xlabel('X Axis')
ylabel('Y Axis')
```

Note that the text string passed to the `title` command uses T_EX syntax to produce the exponents. See “Information About Using T_EX” in the Text Properties page in the MATLAB Function Reference documentation about using T_EX syntax to produce mathematical symbols.

You can also add these annotations by selecting the axes and typing the above strings into their respective fields in the Property Editor. The graph is now ready to print and export.

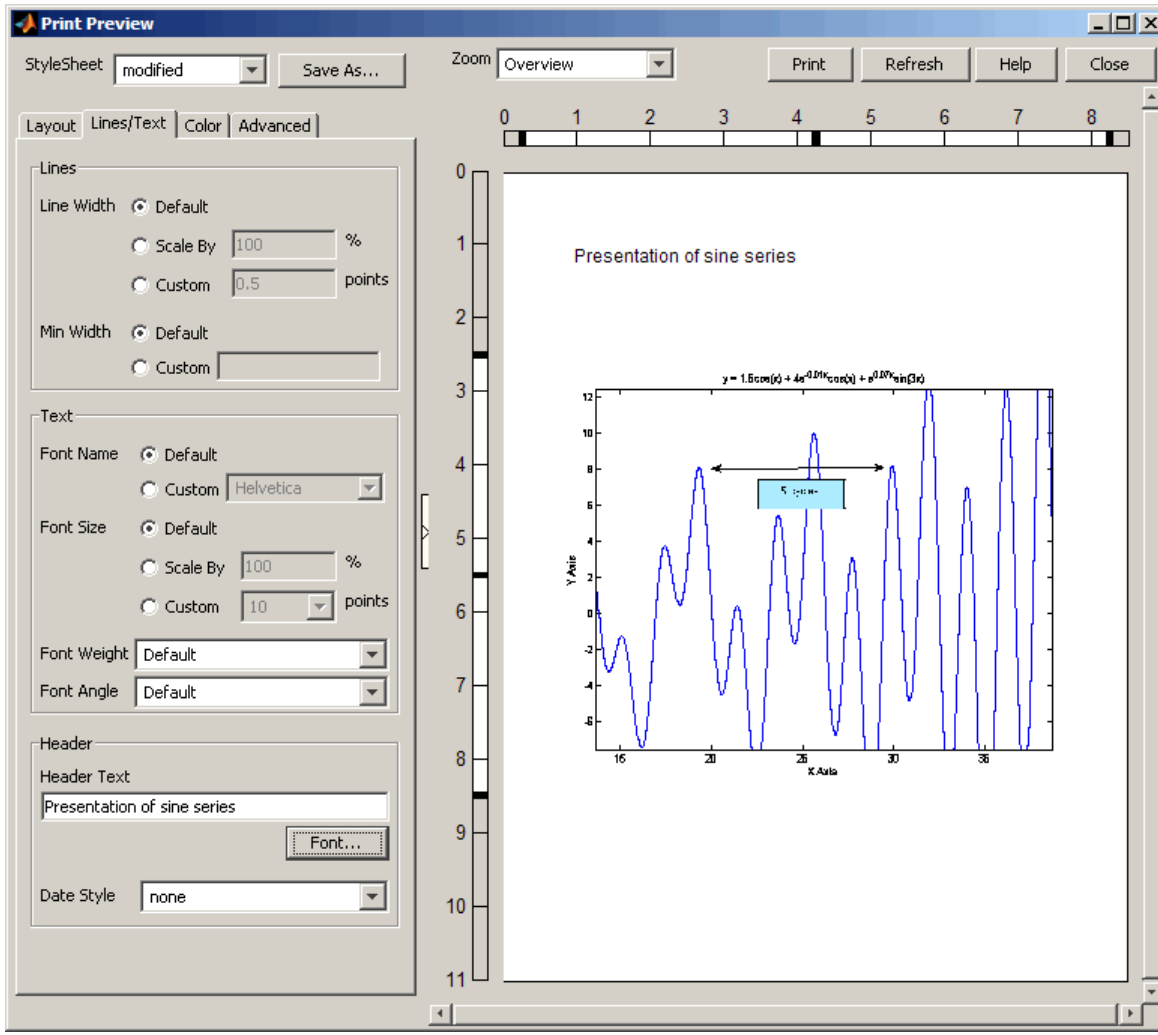


Printing the Graph

Before printing the graph, select **Print Preview** from the figure **File** menu to see and modify how the graph will be laid out on the page. The Print Preview window opens, containing a tabbed control panel on its left side and a page image on its right side.

- Click the **Lines/Text** tab, and enter a line of text in the **Header Text** edit field that you want to place at the top of the page. You can change the font, style, and size of the header by clicking the **Font** button beneath the text

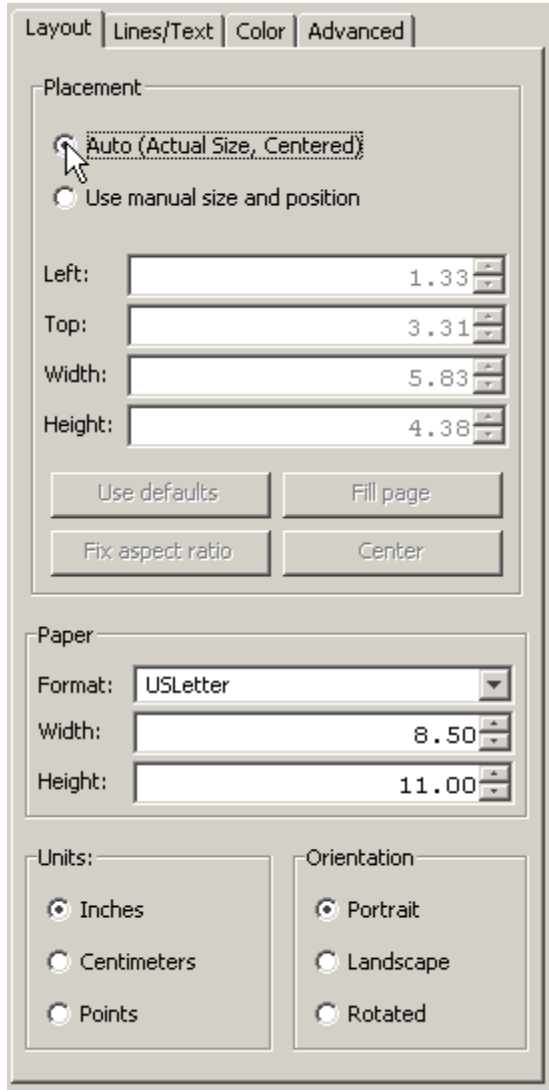
field, and also use the **Date Style** drop-down list to specify a date format to add the current date/time to the header.



- Notice the three black handlebars in the rulers along the left and top sides of the preview pane. The outside handlebars let you stretch one edge of the plot, leaving the other edges in place. The inner handlebars let you move

the plot up and down or left and right without stretching it. Using them does not affect the figure itself, only the printed version of it.

- You can also change the size and position of the plot on the page using the buttons and edit boxes on the **Layout** tab. You can revert to the original configuration by clicking the **Auto (Actual Size, Centered)** option button, and correct stretching and shrinking by clicking **Fix Aspect Ratio**. The following picture shows the **Layout** tab in Auto configuration.



- By default, MATLAB recalculates the locations of the axes tick marks because a printed graph is normally larger than the one displayed on your monitor. However, you can keep your graph's tick marks and limits when printing it by clicking the **Advanced** tab and selecting **Keep screen limits and ticks**.

- When you are ready to print your plot, click **Print** in the right pane. You can also click **Close** to accept the settings and dismiss the dialog box. Later, you can print the figure as you previewed it using **Print** on the figure's **File** menu. Both methods will open a standard Print dialog box, and will produce the same printed results.

Note There is no way to cancel a print preview; any changes you make will take effect if you print the figure. If you want to revert to a default page layout, you can generally accomplish this by selecting either the **Use Defaults** button or the **Auto (Actual Size, Centered)** option button on the **Layout** tab, although this will not affect every setting you can make.

The Print Preview dialog box provides many other options for controlling how printed graphs look. Click its **Help** button for more information.

Exporting the Graph

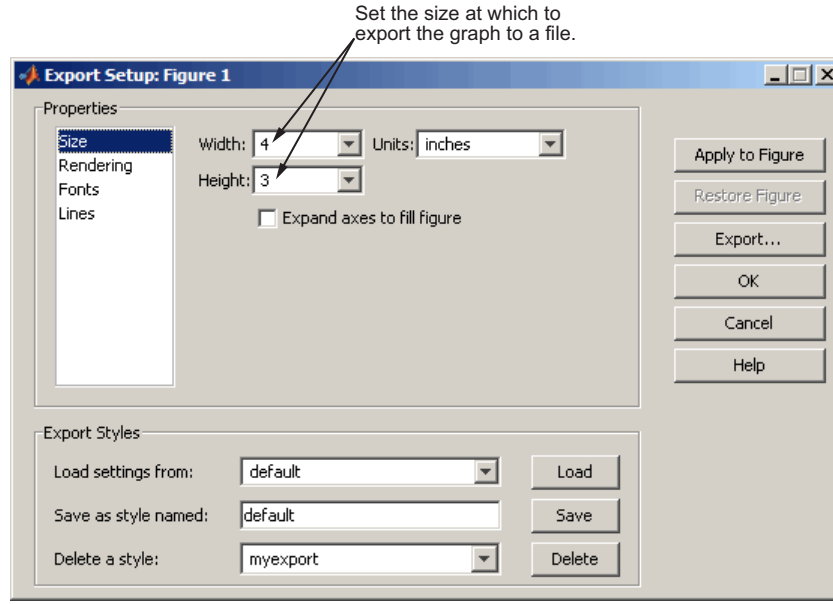
Exporting a graph is the process of creating a standard graphics file format of the graph (such as EPS or TIFF), which you can then import into other applications like word processors, drawing packages, etc.

This example exports the graph as an EPS file with the following requirements:

- The size of the picture when imported into the word processor document should be 4 inches wide and 3 inches high.
- All the text in the figure should have a size of 8 points.

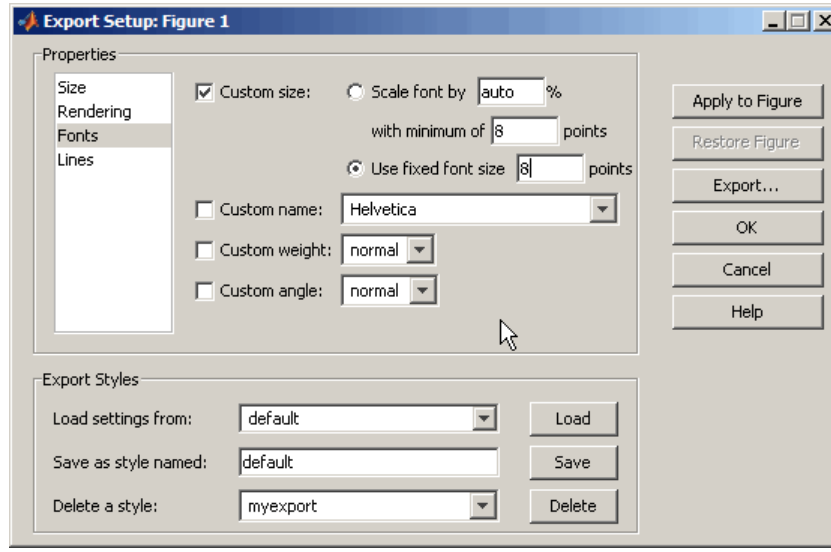
Specifying the Size of the Graph

To set the size, use the Export Setup dialog box (select **Export Setup** from the figure **File** menu). Then select 4 from the **Width** list and 3 from the **Height** list.



Specifying the Font Size

To set the font size of all the text in the graph, select `Fonts` in the Export Setup dialog box **Properties** selector. Then click **Use fixed font size** and enter `8` in the text box.



Selecting the File Format

After you finish setting options for the exported graph, click the **Export** button. MATLAB displays a standard Save As dialog box that enables you to specify a name for the file as well as select the type of file format you want to use.

The **Save as type** drop-down menu lists a number of other options for file formats. For this example, select `EPS (*.eps)` from the **Save as type** menu.

You can import the saved file into any application that supports EPS files.

You can also use the `print` command to print figures on your local printer or to export graphs to standard file types.

For More Information See the `print` command reference page and "Printing and Exporting" in the MATLAB Graphics documentation or select **Printing and Exporting** from the figure **Help** menu.

Using Basic Plotting Functions

In this section...

Creating a Plot

The `plot` function has different forms, depending on the input arguments. If `y` is a vector, `plot(y)` produces a piecewise linear graph of the elements of `y` versus the index of the elements of `y`. If you specify two vectors as arguments, `plot(x,y)` produces a graph of `y` versus `x`.

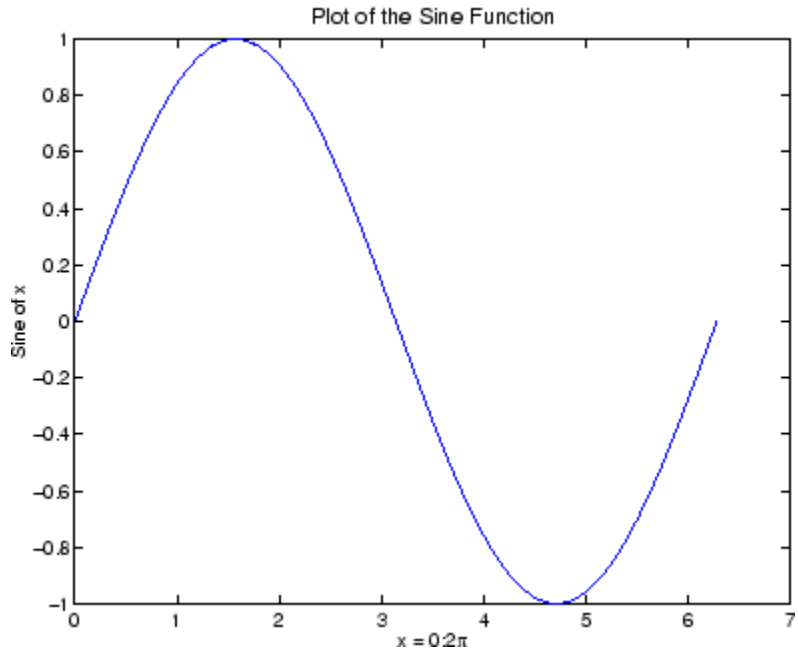
For example, these statements use the `colon` operator to create a vector of `x` values ranging from 0 to 2π, compute the sine of these values, and plot the result:

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)
```

Now label the axes and add a title. The characters `\pi` create the symbol π. See "text strings" in the MATLAB Reference documentation for more symbols:

```
xlabel('x = 0:2\pi')
ylabel('Sine of x')
```

```
title('Plot of the Sine Function','FontSize',12)
```



Plotting Multiple Data Sets in One Graph

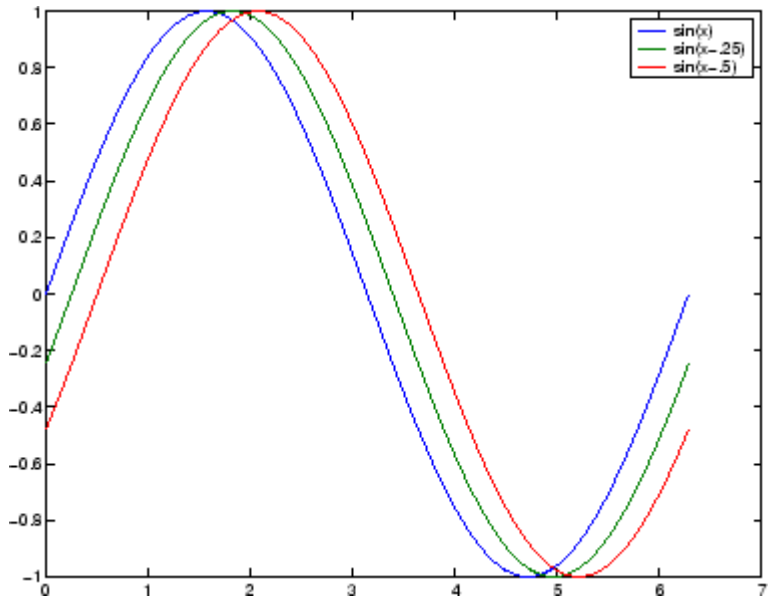
Multiple x-y pair arguments create multiple graphs with a single call to `plot`. MATLAB automatically cycles through a predefined (but user settable) list of colors to allow discrimination among sets of data. See the axes `ColorOrder` and `LineStyleOrder` properties.

For example, these statements plot three related functions of `x`, with each curve in a separate distinguishing color:

```
x = 0:pi/100:2*pi;
y = sin(x);
y2 = sin(x-.25);
y3 = sin(x-.5);
plot(x,y,x,y2,x,y3)
```

The `legend` command provides an easy way to identify the individual plots:

```
legend('sin(x)','sin(x-.25)','sin(x-.5)')
```



For More Information See “Defining the Color of Lines for Plotting” in the MATLAB Graphics documentation.

Specifying Line Styles and Colors

It is possible to specify color, line styles, and markers (such as plus signs or circles) when you plot your data using the `plot` command:

```
plot(x,y,'color_style_marker')
```

`color_style_marker` is a string containing from one to four characters (enclosed in single quotation marks) constructed from a color, a line style, and a marker type:

- Color strings are `'c'`, `'m'`, `'y'`, `'r'`, `'g'`, `'b'`, `'w'`, and `'k'`. These correspond to cyan, magenta, yellow, red, green, blue, white, and black.

- Line style strings are `'-'` for solid, `'--'` for dashed, `':'` for dotted, and `'-.'` for dash-dot. Omit the line style for no line.
- The marker types are `'+'`, `'o'`, `'*'`, and `'x'`, and the filled marker types are `'s'` for square, `'d'` for diamond, `'^'` for up triangle, `'v'` for down triangle, `'>'` for right triangle, `'<'` for left triangle, `'p'` for pentagram, `'h'` for hexagram, and `none` for no marker.

You can also edit color, line style, and markers interactively. See "Editing Plots" on page 3-17 for more information.

Plotting Lines and Markers

If you specify a marker type but not a line style, MATLAB draws only the marker. For example,

```
plot(x,y,'ks')
```

plots black squares at each data point, but does not connect the markers with a line.

The statement

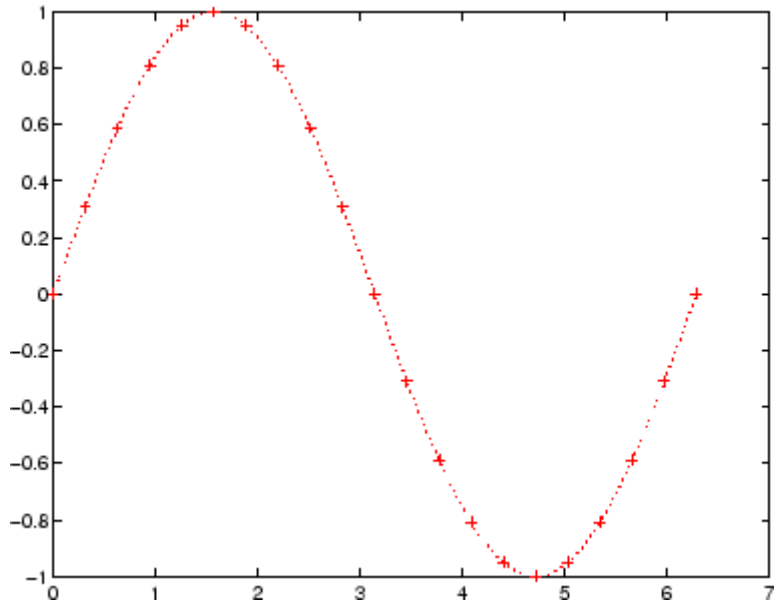
```
plot(x,y,'r:+')
```

plots a red dotted line and places plus sign markers at each data point.

Placing Markers at Every Tenth Data Point

You might want to use fewer data points to plot the markers than you use to plot the lines. This example plots the data twice using a different number of points for the dotted line and marker plots:

```
x1 = 0:pi/100:2*pi;
x2 = 0:pi/10:2*pi;
plot(x1,sin(x1),'r:',x2,sin(x2),'r+')
```



Graphing Imaginary and Complex Data

When the arguments to `plot` are complex, the imaginary part is ignored *except* when you pass `plot` a single complex argument. For this special case, the command is a shortcut for a graph of the real part versus the imaginary part. Therefore,

```
plot(Z)
```

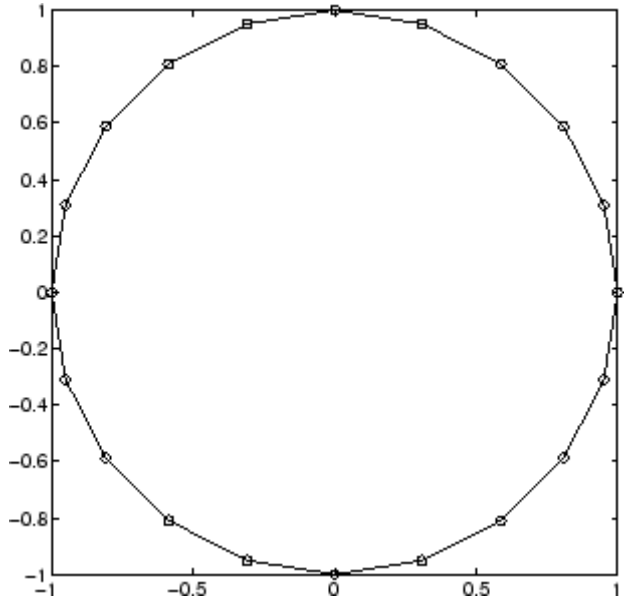
where `Z` is a complex vector or matrix, is equivalent to

```
plot(real(Z),imag(Z))
```

For example,

```
t = 0:pi/10:2*pi;
plot(exp(i*t),'-o')
axis equal
```

draws a 20-sided polygon with little circles at the vertices. The `axis equal` command makes the individual tick-mark increments on the *x*- and *y*-axes the same length, which makes this plot more circular in appearance.



Adding Plots to an Existing Graph

The `hold` command enables you to add plots to an existing graph. When you type

```
hold on
```

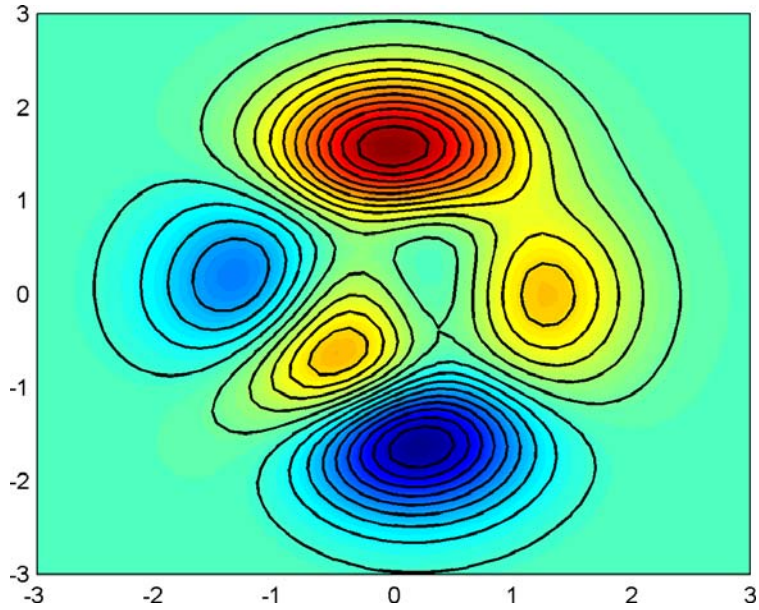
MATLAB does not replace the existing graph when you issue another plotting command; it adds the new data to the current graph, rescaling the axes if necessary.

For example, these statements first create a contour plot of the `peaks` function, then superimpose a pseudocolor plot of the same function:

```
[x,y,z] = peaks;
pcolor(x,y,z)
shading interp
```

```
hold on
contour(x,y,z,20,'k')
hold off
```

The `hold on` command combines the `pcolor` plot with the `contour` plot in one figure.



For More Information See “Creating Specialized Plots” in the MATLAB Graphics documentation for details about a variety of graph types.

Figure Windows

Graphing functions automatically open a new figure window if there are no figure windows already on the screen. If a figure window exists, MATLAB uses that window for graphics output. If there are multiple figure windows open, MATLAB targets the one that is designated the “current figure” (the last figure used or clicked in).

To make an existing figure window the current figure, you can click the mouse while the pointer is in that window or you can type

```
figure(n)
```

where `n` is the number in the figure title bar. The results of subsequent graphics commands are displayed in this window.

To open a new figure window and make it the current figure, type

```
figure
```

Clearing the Figure for a New Plot

When a figure already exists, most plotting commands clear the axes and use this figure to create the new plot. However, these commands do not reset figure properties, such as the background color or the colormap. If you have set any figure properties in the previous plot, you might want to use the `clf` command with the `reset` option,

```
clf reset
```

before creating your new plot to restore the figure's properties to their defaults.

For More Information See "Figure Properties" and "Graphics Windows — the Figure" in the MATLAB Graphics documentation for details about figures.

Displaying Multiple Plots in One Figure

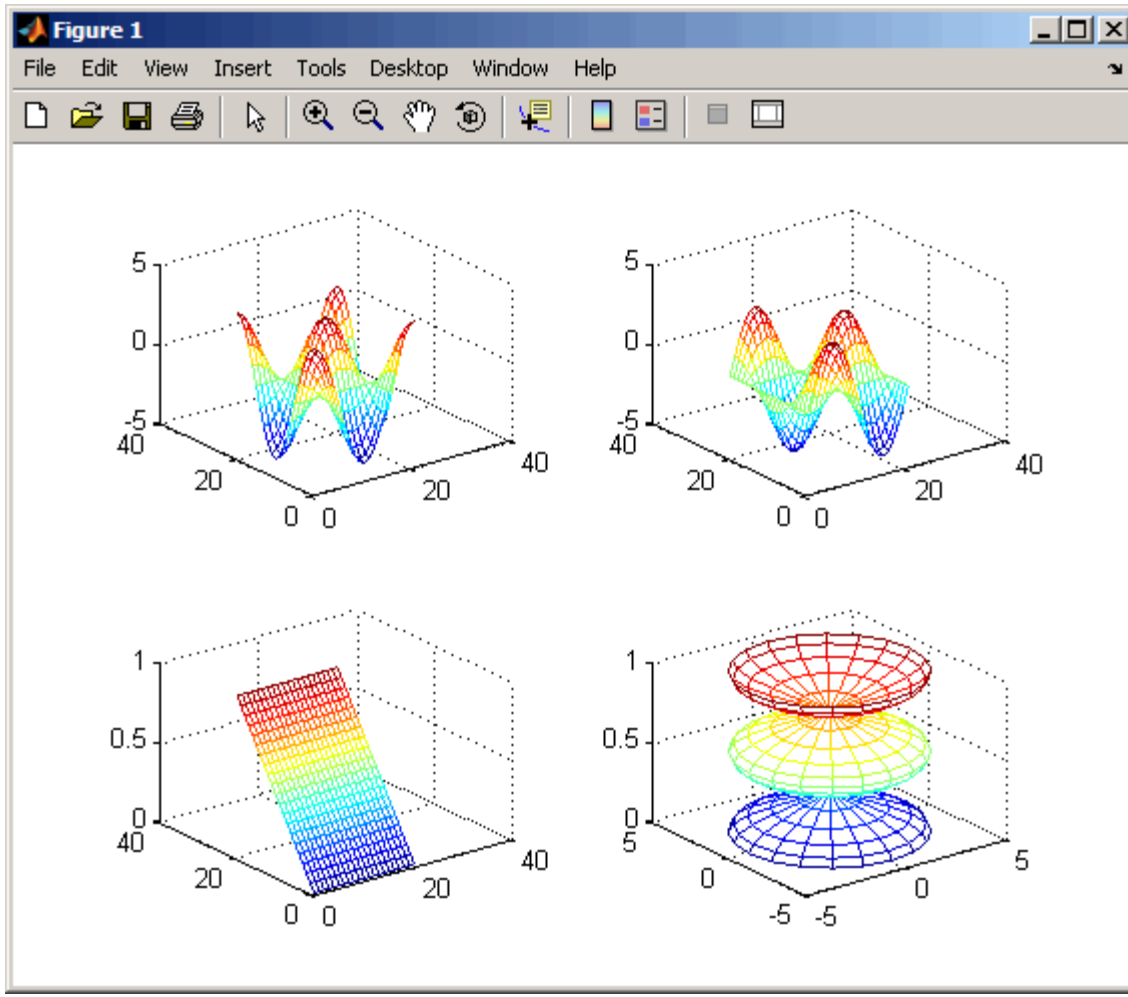
The `subplot` command enables you to display multiple plots in the same window or print them on the same piece of paper. Typing

```
subplot(m,n,p)
```

partitions the figure window into an `m`-by-`n` matrix of small subplots and selects the `p`th subplot for the current plot. The plots are numbered along the first row of the figure window, then the second row, and so on. For example, these statements plot data in four different subregions of the figure window:

```
t = 0:pi/10:2*pi;
```

```
[X,Y,Z] = cylinder(4*cos(t));
subplot(2,2,1); mesh(X)
subplot(2,2,2); mesh(Y)
subplot(2,2,3); mesh(Z)
subplot(2,2,4); mesh(X,Y,Z)
```



You can add subplots to GUIs as well as to figures. For details about creating subplots in a GUIDE-generated GUI, see "Creating Subplots" in the MATLAB Creating Graphical User Interfaces documentation.

Controlling the Axes

The `axis` command provides a number of options for setting the scaling, orientation, and aspect ratio of graphs. You can also set these options interactively. See "Editing Plots" on page 3-17 for more information.

Setting Axis Limits

By default, MATLAB finds the maxima and minima of the data and chooses the axis limits to span this range. The `axis` command enables you to specify your own limits:

```
axis([xmin xmax ymin ymax])
```

or for three-dimensional graphs,

```
axis([xmin xmax ymin ymax zmin zmax])
```

Use the command

```
axis auto
```

to reenable MATLAB automatic limit selection.

Setting the Axis Aspect Ratio

The `axis` command also enables you to specify a number of predefined modes. For example,

```
axis square
```

makes the *x*-axis and *y*-axis the same length.

```
axis equal
```

makes the individual tick mark increments on the *x*-axes and *y*-axes the same length. This means

```
plot(exp(i*[0:pi/10:2*pi]))
```

followed by either `axis square` or `axis equal` turns the oval into a proper circle:

```
axis auto normal
```

returns the axis scaling to its default automatic mode.

Setting Axis Visibility

You can use the `axis` command to make the axis visible or invisible.

```
axis on
```

makes the axes visible. This is the default.

```
axis off
```

makes the axes invisible.

Setting Grid Lines

The `grid` command toggles grid lines on and off. The statement

```
grid on
```

turns the grid lines on, and

```
grid off
```

turns them back off again.

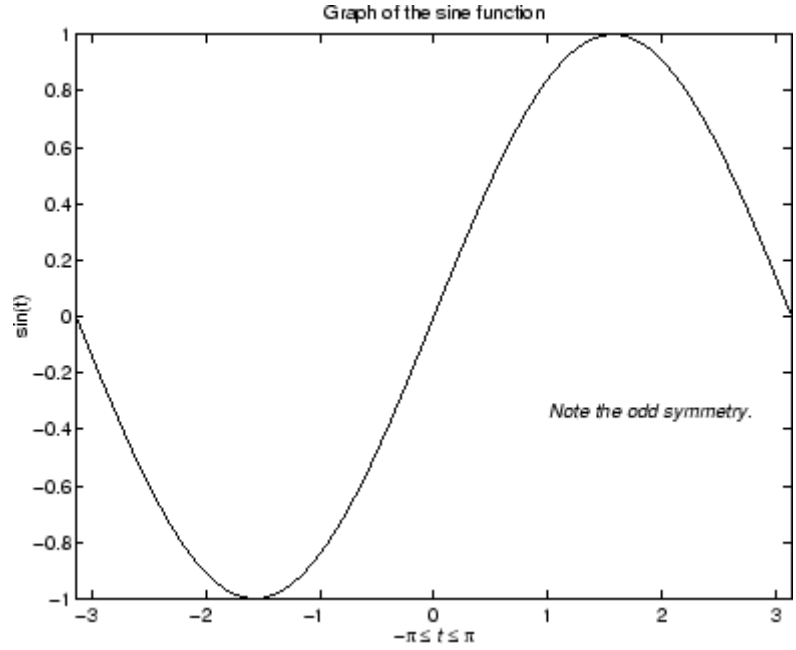
For More Information See the `axis` and `axes` reference pages and "Axes Properties" in the MATLAB Graphics documentation.

Adding Axis Labels and Titles

The `xlabel`, `ylabel`, and `zlabel` commands add *x*-, *y*-, and *z*-axis labels. The `title` command adds a title at the top of the figure and the `text` function inserts text anywhere in the figure.

You can produce mathematical symbols using LaTeX notation in the text string, as the following example illustrates:

```
t = -pi:pi/100:pi;
y = sin(t);
plot(t,y)
axis([-pi pi -1 1])
xlabel('-\pi \leq {\itt} \leq \pi')
ylabel('sin(t)')
title('Graph of the sine function')
text(1,-1/3,'{\itNote the odd symmetry.}')
```



You can also set these options interactively. See “Editing Plots” on page 3-17 for more information.

Note that the location of the text string is defined in axes units (i.e., the same units as the data). See the `annotation` function for a way to place text in normalized figure units.

Saving Figures

Save a figure by selecting **Save** from the **File** menu to display a Save dialog box. MATLAB saves the data it needs to recreate the figure and its contents (i.e., the entire graph) in a file with a `.fig` extension.

To save a figure using a standard graphics format, such as TIFF, for use with other applications, select **Export Setup** from the **File** menu. You can also save from the command line—use the `saveas` command, including any options to save the figure in a different format. The more restricted `hgexport` command, which saves figures to either bitmap or metafile files, depending on the rendering method in effect, is also available.

See “Exporting the Graph” on page 3-46 for an example.

Saving Workspace Data

You can save the variables in your workspace by selecting **Save Workspace As** from the figure **File** menu. You can reload saved data using the **Import Data** item in the figure **File** menu. MATLAB supports a variety of data file formats, including MATLAB data files, which have a `.mat` extension.

Generating M-Code to Recreate a Figure

You can generate MATLAB code that recreates a figure and the graph it contains by selecting **Generate M-File** from the figure **File** menu. This option is particularly useful if you have developed a graph using plotting tools and want to create a similar graph using the same or different data.

Saving Figures That Are Compatible with the Previous Version of MATLAB

Create backward-compatible FIG-files by following these two steps:

1. Ensure that any plotting functions used to create the contents of the figure are called with the `'v6'` argument, where applicable.
2. Use the `'-v6'` option with the `hgsave` command.

Note The `v6` option enables users of Version 7.x of MATLAB to create FIG-files that previous versions can open. It is obsolete and will be removed in a future version of MATLAB. For more information, see "Plot Objects and Backward Compatibility" in the MATLAB Graphics documentation.

Creating Mesh and Surface Plots

In this section...

About Mesh and Surface Plots

MATLAB defines a surface by the *z*-coordinates of points above a grid in the *x*-*y* plane, using straight lines to connect adjacent points. The `mesh` and `surf` plotting functions display surfaces in three dimensions. `mesh` produces wireframe surfaces that color only the lines connecting the defining points. `surf` displays both the connecting lines and the faces of the surface in color.

The figure colormap and figure properties determine how MATLAB colors the surface.

Visualizing Functions of Two Variables

To display a function of two variables, $z = f(x,y)$,

1. Generate `X` and `Y` matrices consisting of repeated rows and columns, respectively, over the domain of the function.
2. Use `X` and `Y` to evaluate and graph the function.

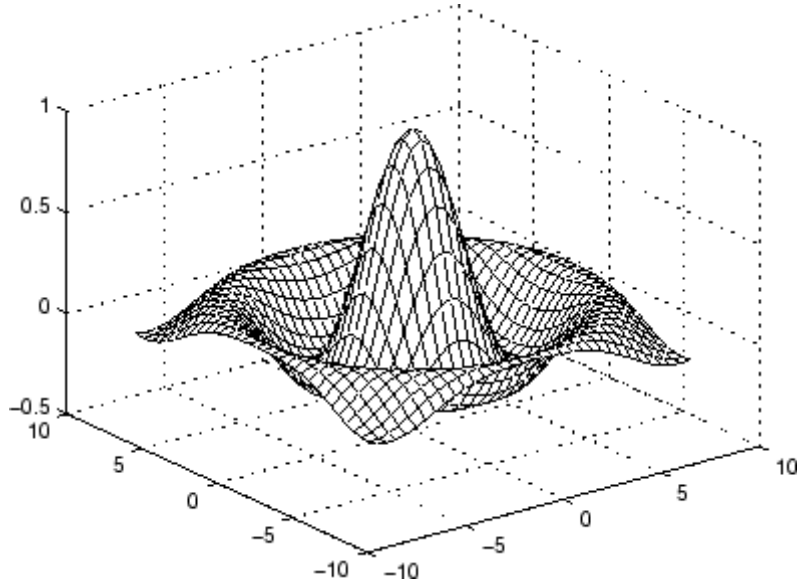
The `meshgrid` function transforms the domain specified by a single vector or two vectors `x` and `y` into matrices `X` and `Y` for use in evaluating functions of two variables. The rows of `X` are copies of the vector `x` and the columns of `Y` are copies of the vector `y`.

Example — Graphing the sinc Function

This example evaluates and graphs the two-dimensional `sinc` function, $\sin(r)/r$, between the *x* and *y* directions. `R` is the distance from the origin, which is at the center of the matrix. Adding `eps` (a MATLAB command that returns a small floating-point number) avoids the indeterminate 0/0 at the origin:

```
[X,Y] = meshgrid(-8:.5:8);
```

```
R = sqrt(X.^2 + Y.^2) + eps;
Z = sin(R)./R;
mesh(X,Y,Z,'EdgeColor','black')
```



By default, MATLAB colors the mesh using the current colormap. However, this example uses a single-colored mesh by specifying the `EdgeColor` surface property. See the `surface` reference page for a list of all surface properties.

You can create a mesh with see-through faces by disabling hidden line removal:

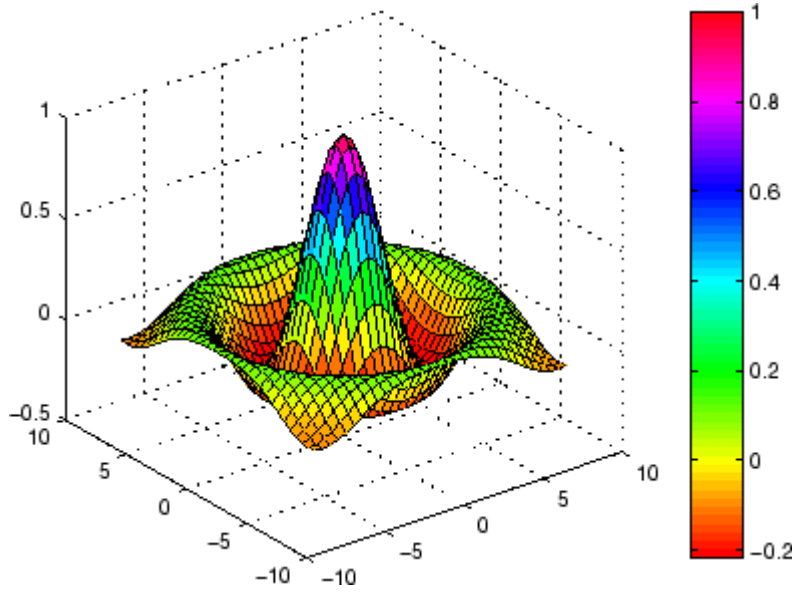
```
hidden off
```

See the `hidden` reference page for more information on this option.

Example — Colored Surface Plots

A surface plot is similar to a mesh plot except that MATLAB colors the rectangular faces of the surface. The color of each face is determined by the values of `Z` and the colormap (a colormap is an ordered list of colors). These statements graph the `sinc` function as a surface plot, specify a colormap, and add a color bar to show the mapping of data to color:

```
surf(X,Y,Z)
colormap hsv
colorbar
```



See the `colormap` reference page for information on colormaps.

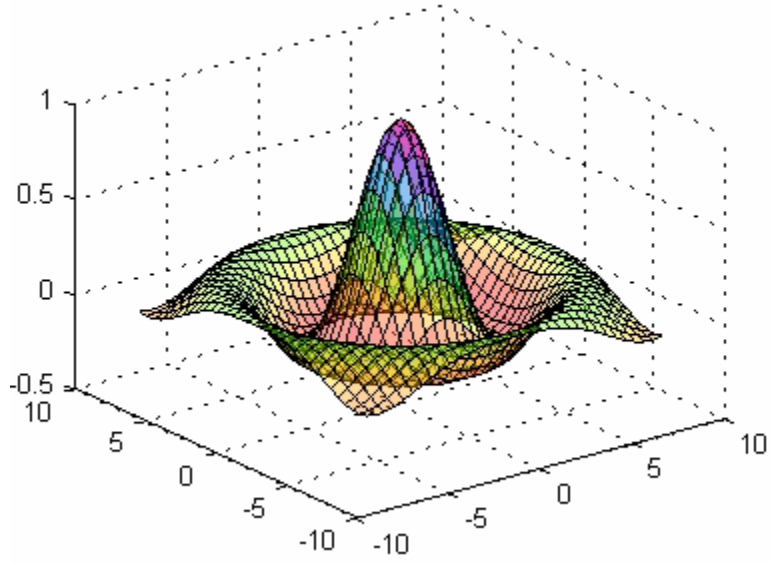
For More Information See "Creating 3-D Graphs" in the MATLAB 3-D Visualization documentation for more information on surface plots.

Making Surfaces Transparent

You can make the faces of a surface transparent to a varying degree. Transparency (referred to as the alpha value) can be specified for the whole object or can be based on an `alphamap`, which behaves similarly to colormaps. For example,

```
surf(X,Y,Z)
colormap hsv
alpha(.4)
```

produces a surface with a face alpha value of 0.4. Alpha values range from 0 (completely transparent) to 1 (not transparent).



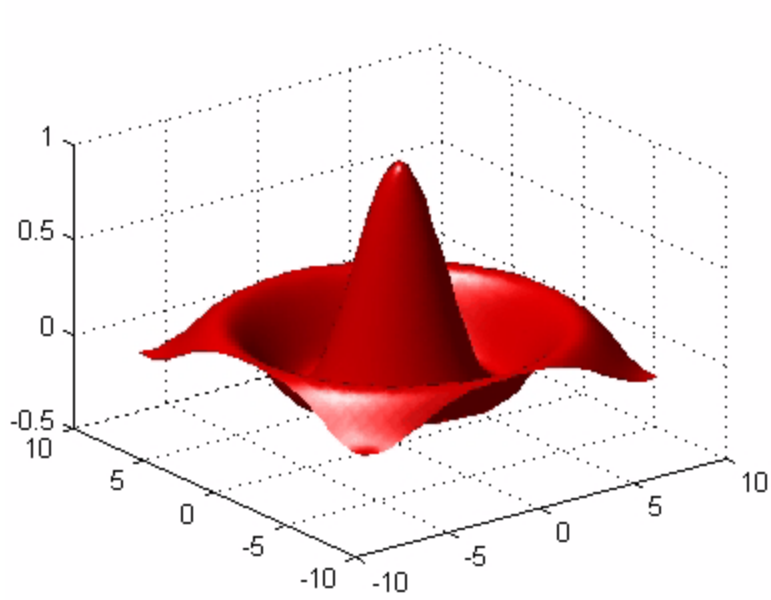
For More Information See "Transparency" in the MATLAB 3-D Visualization documentation for details about using this feature.

Illuminating Surface Plots with Lights

Lighting is the technique of illuminating an object with a directional light source. In certain cases, this technique can make subtle differences in surface shape easier to see. Lighting can also be used to add realism to three-dimensional graphs.

This example uses the same surface as the previous examples, but colors it red and removes the mesh lines. A light object is then added to the left of the "camera" (the camera is the location in space from where you are viewing the surface):

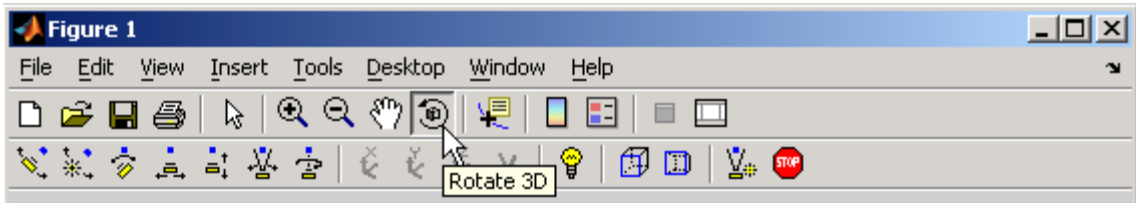
```
surf(X,Y,Z,'FaceColor','red','EdgeColor','none')
camlight left; lighting phong
```



Manipulating the Surface

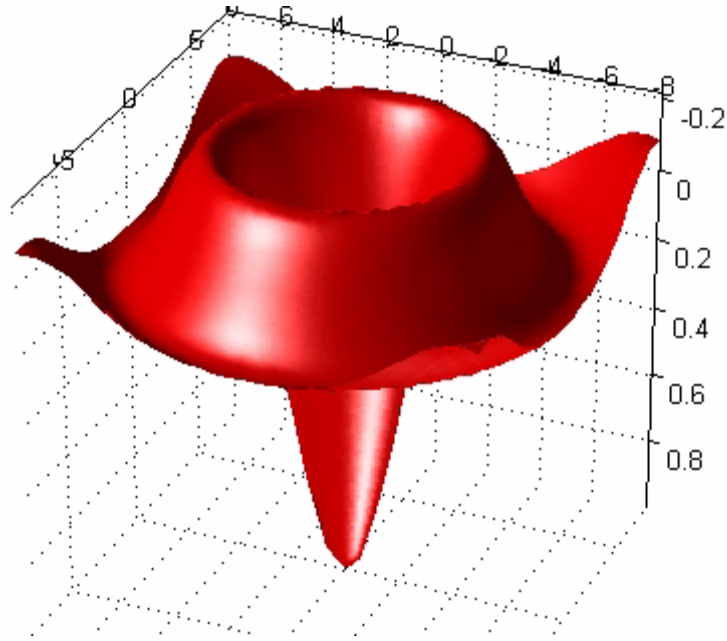
The figure toolbar and the camera toolbar provide ways to explore 3-D graphics interactively. Display the camera toolbar by selecting **Camera Toolbar** from the figure **View** menu.

The following picture shows both toolbars with the **Rotate 3D** tool selected.



These tools enable you to move the camera around the surface object, zoom, add lighting, and perform other viewing operations without issuing commands.

The following picture shows the surface viewed by orbiting the camera toward the bottom using **Rotate 3D**. A scene light has been added to illuminate the underside of the surface, which is not lit by the light added in the previous section.



For More Information See "Lighting as a Visualization Tool" and "View Control with the Camera Toolbar" in the MATLAB 3-D Visualization documentation for details about these techniques.

Plotting Image Data

In this section...

About Plotting Image Data

Two-dimensional arrays can be displayed as *images*, where the array elements determine brightness or color of the images. For example, the statements

```
load durer
whos
Name          Size         Bytes  Class

  X           648x509      2638656  double array
  caption     2x28             112  char array
  map         128x3           3072  double array
```

load the file `durer.mat`, adding three variables to the workspace. The matrix `X` is a 648-by-509 matrix and `map` is a 128-by-3 matrix that is the colormap for this image.

MAT-files, such as `durer.mat`, are binary files that can be created on one platform and later read by MATLAB on a different platform.

The elements of `X` are integers between 1 and 128, which serve as indices into the colormap, `map`. Then

```
image(X)
colormap(map)
axis image
```

reproduces Albrecht Dürer's etching shown in "Matrices and Magic Squares" on page 2-2. A high-resolution scan of the magic square in the upper-right corner is available in another file. Type

```
load detail
```

and then use the up arrow key on your keyboard to reexecute the `image`, `colormap`, and `axis` commands. The statement

```
colormap(hot)
```

adds some 21st century colorization to the 16th century etching. The function `hot` generates a colormap containing shades of reds, oranges, and yellows. Typically, a given image matrix has a specific colormap associated with it. See the `colormap` reference page for a list of other predefined colormaps.

Reading and Writing Images

You can read standard image files (TIFF, JPEG, BMP, etc.) into MATLAB using the `imread` function. The type of data returned by `imread` depends on the type of image you are reading.

You can write MATLAB data to a variety of standard image formats using the `imwrite` function. See the MATLAB reference pages for these functions for more information and examples.

For More Information See "Displaying Bit-Mapped Images" in the MATLAB Graphics documentation for details about the image processing capabilities of MATLAB.

Printing Graphics

In this section...

Overview of Printing

You can print a MATLAB figure directly on a printer connected to your computer or you can export the figure to one of the standard graphics file formats supported by MATLAB. There are two ways to print and export figures:

- Use the **Print**, **Print Preview**, or **Export Setup** GUI options under the **File** menu; see "Preparing Graphs for Presentation" on page 3-37 for an example.
- Use the `print` command to print or export the figure from the command line.

The `print` command provides greater control over drivers and file formats. The Print Preview dialog box gives you greater control over figure size, proportions, placement, and page headers. You can sometimes notice differences in output as printed from a GUI and as printed from the command line.

Printing from the File Menu

There are two menu options under the **File** menu that pertain to printing:

- The **Print Preview** option displays a dialog box that lets you lay out and style figures for printing while previewing the output page, and from which you can print the figure. It includes options that formerly were part of the Page Setup dialog box.

- The **Print** option displays a dialog box that lets you choose a printer, select standard printing options, and print the figure.

Use **Print Preview** to determine whether the printed output is what you want. Click the Print Preview dialog box **Help** button to display information on how to set up the page.

See "Printing the Graph" on page 3-42 for an example of printing from the Print Preview dialog. For details on printing from GUIs and from the Command Window, see "Printing and Exporting" in the MATLAB Graphics documentation.

Exporting the Figure to a Graphics File

The **Export Setup** option in the **File** menu opens a GUI that enables you to set graphic characteristics, such as text size, font, and style, for figures you save as graphics files. The Export Setup GUI lets you define and apply templates to customize and standardize output. After setup, you can export the figure to a number of standard graphics file formats such as EPS, PNG, and TIFF.

See "Exporting the Graph" on page 3-46 for an example of exporting a figure to a graphics file.

Using the Print Command

The `print` command provides more flexibility in the type of output sent to the printer and allows you to control printing from M-files. The result can be sent directly to your default printer or stored in a specified file. A wide variety of output formats, including TIFF, JPEG, and PostScript, is available.

For example, this statement saves the contents of the current figure window as color Encapsulated Level 2 PostScript in the file called `magicsquare.eps`. It also includes a TIFF preview, which enables most word processors to display the picture.

```
print -depsc2 -tiff magicsquare.eps
```

To save the same figure as a TIFF file with a resolution of 200 dpi, use the following command:

```
print -dtiff -r200 magicsquare.tiff
```

If you type `print` on the command line,

```
print
```

MATLAB prints the current figure on your default printer.

For More Information See the `print` reference page and "Printing and Exporting" in the MATLAB Graphics documentation for details about printing.

Handle Graphics

In this section...

Handle Graphics refers to a system of graphics objects that MATLAB uses to implement graphing and visualization functions. Each object created has a fixed set of properties. You can use these properties to control the behavior and appearance of your graph.

When you call a plotting function, MATLAB creates the graph using various graphics objects, such as a figure window, axes, lines, text, and so on. MATLAB enables you to query the value of each property and set the values of most properties.

For example, the following statement creates a figure with a white background color and without displaying the figure toolbar:

```
figure('Color','white','Toolbar','none')
```

Using the Handle

Whenever MATLAB creates a graphics object, it assigns an identifier (called a *handle*) to the object. You can use this handle to access the object's properties with the `set` and `get` functions. For example, the following statements create a graph and return a handle to a lineseries object in `h`:

```
x = 1:10;
y = x.^3;
h = plot(x,y);
```

You can use the handle `h` to set the properties of the lineseries object. For example, you can set its `Color` property:

```
set(h,'Color','red')
```

You can also specify properties when you call the plotting function:

```
h = plot(x,y,'Color','red');
```

When you query the lineseries properties,

```
get(h,'LineWidth')
```

MATLAB returns the answer:

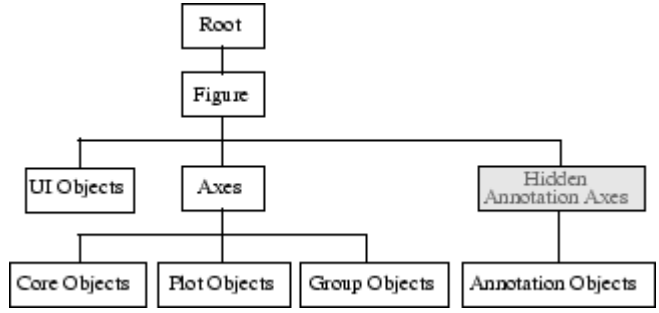
```
ans =
 0.5000
```

Use the handle to see what properties a particular object contains:

```
get(h)
```

Graphics Objects

Graphics objects are the basic elements used to display graphs and user interface components. These objects are organized into a hierarchy, as shown by the following diagram.



Key Graphics Objects

When you call a function to create a graph, MATLAB creates a hierarchy of graphics objects. For example, calling the `plot` function creates the following graphics objects:

- Lineseries plot objects — Represent the data passed to the `plot` function.

- Axes — Provide a frame of reference and scaling for the plotted lineseries.
- Text — Label the axes tick marks and are used for titles and annotations.
- Figures — Are the windows that contain axes toolbars, menus, etc.

Different types of graphs use different objects to represent data; however, all data objects are contained in axes and all objects (except root) are contained in figures.

The root is an abstract object that primarily stores information about your computer or MATLAB state. You cannot create an instance of the root object.

For More Information See "Handle Graphics Objects" in the MATLAB Graphics documentation for details about graphics objects.

User interface objects are used to create graphical user interfaces (GUIs). These objects include components like push buttons, editable text boxes, and list boxes.

For More Information See Chapter 6, "Creating Graphical User Interfaces" for details about user interface objects.

Creating Objects

Plotting functions (like `plot` and `surf`) call the appropriate low-level function to draw their respective graph. For information about an object's properties, you can use the Handle Graphics Property Browser in the MATLAB online Graphics documentation.

Functions for Working with Objects

This table lists functions commonly used when working with objects.

Function	Purpose
`allchild`	Find all children of specified objects.
`ancestor`	Find ancestor of graphics object.
`copyobj`	Copy graphics object.
`delete`	Delete an object.
`findall`	Find all graphics objects (including hidden handles).
`findobj`	Find the handles of objects having specified property values.
`gca`	Return the handle of the current axes.
`gcf`	Return the handle of the current figure.
`gco`	Return the handle of the current object.
`get`	Query the values of an object's properties.
`ishandle`	True if the value is a valid object handle.
`set`	Set the values of an object's properties.

Setting Object Properties

All object properties have default values. However, you might find it useful to change the settings of some properties to customize your graph. There are two ways to set object properties:

- Specify values for properties when you create the object.
- Set the property value on an object that already exists.

Setting Properties from Plotting Commands

You can specify object property value pairs as arguments to many plotting functions, such as `plot`, `mesh`, and `surf`.

For example, plotting commands that create lineseries or surfaceplot objects enable you to specify property name/property value pairs as arguments. The command

```
surf(x,y,z,'FaceColor','interp',...
   'FaceLighting','gouraud')
```

plots the data in the variables `x`, `y`, and `z` using a surfaceplot object with interpolated face color and employing the Gouraud face light technique. You can set any of the object's properties this way.

Setting Properties of Existing Objects

To modify the property values of existing objects, you can use the `set` command or the Property Editor. This section describes how to use the `set` command. See "Using the Property Editor" on page 3-18 for more information.

Most plotting functions return the handles of the objects that they create so you can modify the objects using the `set` command. For example, these statements plot a 5-by-5 matrix (creating five lineseries, one per column), and then set the `Marker` property to a square and the `MarkerFaceColor` property to green:

```
h = plot(magic(5));
set(h,'Marker','s','MarkerFaceColor','g')
```

In this case, `h` is a vector containing five handles, one for each of the five lineseries in the graph. The `set` statement sets the `Marker` and `MarkerFaceColor` properties of all lineseries to the same values.

Setting Multiple Property Values

If you want to set the properties of each lineseries to a different value, you can use cell arrays to store all the data and pass it to the `set` command. For example, create a plot and save the lineseries handles:

```
h = plot(magic(5));
```

Suppose you want to add different markers to each lineseries and color the marker's face color the same color as the lineseries. You need to define two cell arrays—one containing the property names and the other containing the desired values of the properties.

The `prop_name` cell array contains two elements:

```
prop_name(1) = {'Marker'};
```

```
prop_name(2) = {'MarkerFaceColor'};
```

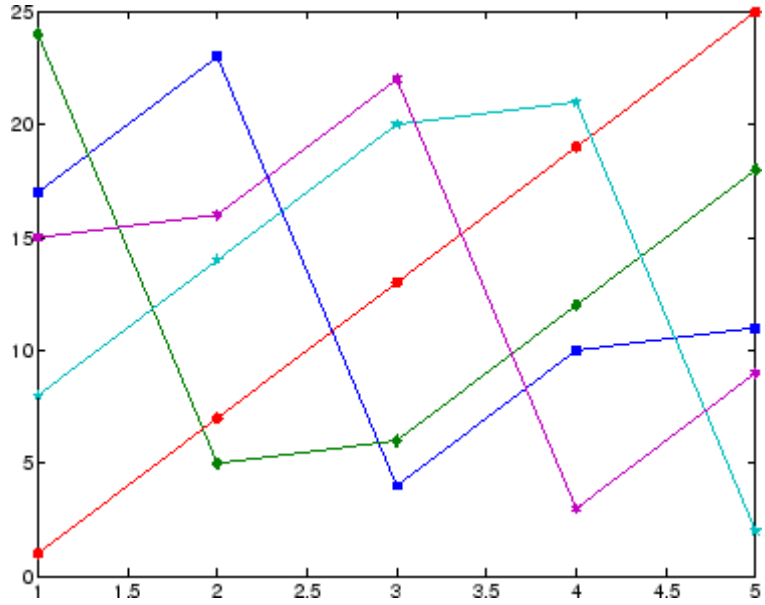
The `prop_values` cell array contains 10 values: five values for the `Marker` property and five values for the `MarkerFaceColor` property. Notice that `prop_values` is a two-dimensional cell array. The first dimension indicates which handle in `h` the values apply to and the second dimension indicates which property the value is assigned to:

```
prop_values(1,1) = {'s'};
prop_values(1,2) = {get(h(1),'Color')};
prop_values(2,1) = {'d'};
prop_values(2,2) = {get(h(2),'Color')};
prop_values(3,1) = {'o'};
prop_values(3,2) = {get(h(3),'Color')};
prop_values(4,1) = {'p'};
prop_values(4,2) = {get(h(4),'Color')};
prop_values(5,1) = {'h'};
prop_values(5,2) = {get(h(5),'Color')};
```

The `MarkerFaceColor` is always assigned the value of the corresponding line's color (obtained by getting the lineseries `Color` property with the `get` command).

After defining the cell arrays, call `set` to specify the new property values:

```
set(h,prop_name,prop_values)
```



Specifying the Axes or Figure

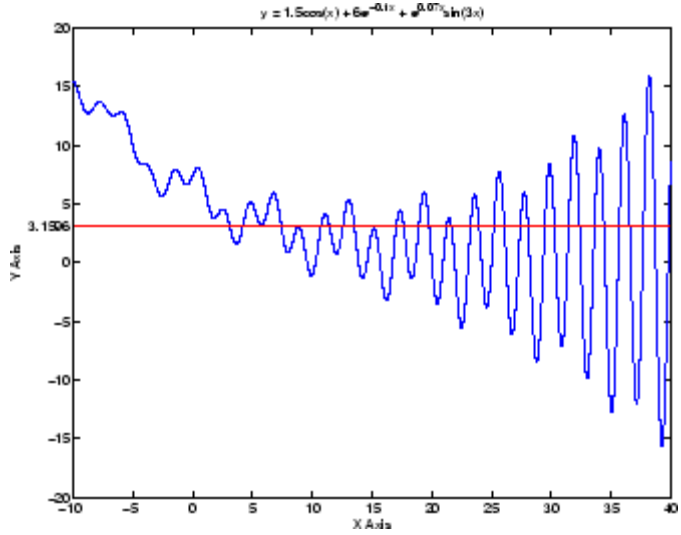
MATLAB always creates an axes or figure if one does not exist when you issue a plotting command. However, when you are creating a graphics M-file, it is good practice to explicitly create and specify the parent axes and figure, particularly if others will use your program. Specifying the parent prevents the following problems:

- Your M-file overwrites the graph in the current figure. Note that a figure becomes the current figure whenever a user clicks it.
- The current figure might be in an unexpected state and not behave as your program expects.

The following example shows a simple M-file that plots a function and the mean of the function over the specified range:

```
function myfunc(x)
% x = -10:.005:40; Here's a value you can use for x
y = [1.5*cos(x) + 6*exp(-.1*x) + exp(.07*x).*sin(3*x)];
ym = mean(y);
```

```
hfig = figure('Name','Function and Mean',...
  'Pointer','fullcrosshair');
hax = axes('Parent',hfig);
plot(hax,x,y)
hold on
plot(hax,[min(x) max(x)],[ym ym],'Color','red')
hold off
ylab = get(hax,'YTick');
set(hax,'YTick',sort([ylab ym]))
title ('y = 1.5cos(x) + 6e^{-0.1x} + e^{0.07x}sin(3x)')
xlabel('X Axis'); ylabel('Y Axis')
```



Finding the Handles of Existing Objects

The `findobj` function enables you to obtain the handles of graphics objects by searching for objects with particular property values. With `findobj` you can specify the values of any combination of properties, which makes it easy to pick one object out of many. `findobj` also recognizes regular expressions (`regexp`).

For example, you might want to find the blue line with square marker having blue face color. You can also specify which figures or axes to search, if there

are more than one. The following four sections provide examples illustrating how to use `findobj`.

Finding All Objects of a Certain Type

Because all objects have a `Type` property that identifies the type of object, you can find the handles of all occurrences of a particular type of object. For example,

```
h = findobj('Type','patch');
```

finds the handles of all patch objects.

Finding Objects with a Particular Property

You can specify multiple properties to narrow the search. For example,

```
h = findobj('Type','line','Color','r','LineStyle',':');
```

finds the handles of all red dotted lines.

Limiting the Scope of the Search

You can specify the starting point in the object hierarchy by passing the handle of the starting figure or axes as the first argument. For example,

```
h = findobj(gca,'Type','text','String','\pi/2');
```

finds the string π/2 only within the current axes.

Using findobj as an Argument

Because `findobj` returns the handles it finds, you can use it in place of the handle argument. For example,

```
set(findobj('Type','line','Color','red'),'LineStyle',':')
```

finds all red lines and sets their line style to dotted.

4

Programming

If you have an active Internet connection, you can watch the Writing a MATLAB Program video demo for an overview of the major functionality.

Flow Control

In this section...

Conditional Control – if, else, switch

This section covers those MATLAB functions that provide conditional program control.

if, else, and elseif

The `if` statement evaluates a logical expression and executes a group of statements when the expression is *true*. The optional `elseif` and `else` keywords provide for the execution of alternate groups of statements. An `end` keyword, which matches the `if`, terminates the last group of statements. The groups of statements are delineated by the four keywords—no braces or brackets are involved.

The MATLAB algorithm for generating a magic square of order *n* involves three different cases: when *n* is odd, when *n* is even but not divisible by 4, or when *n* is divisible by 4. This is described by

```
if rem(n,2) ~= 0
   M = odd_magic(n)
elseif rem(n,4) ~= 0
   M = single_even_magic(n)
else
   M = double_even_magic(n)
end
```

For most values of `n` in this example, the three cases are mutually exclusive. For values that are not mutually exclusive, such as `n=5`, the first *true* condition is executed.

It is important to understand how relational operators and `if` statements work with matrices. When you want to check for equality between two variables, you might use

```
if A == B, ...
```

This is valid MATLAB code, and does what you expect when `A` and `B` are scalars. But when `A` and `B` are matrices, `A == B` does not test *if* they are equal, it tests *where* they are equal; the result is another matrix of 0's and 1's showing element-by-element equality. (In fact, if `A` and `B` are not the same size, then `A == B` is an error.)

```
A = magic(4);     B = A;     B(1,1) = 0;

A == B
ans =
     0     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

The proper way to check for equality between two variables is to use the `isequal` function:

```
if isequal(A, B), ...
```

`isequal` returns a *scalar* logical value of `1` (representing `true`) or `0` (`false`), instead of a matrix, as the expression to be evaluated by the `if` function. Using the `A` and `B` matrices from above, you get

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

Here is another example to emphasize this point. If `A` and `B` are scalars, the following program will never reach the "unexpected situation". But for most pairs of matrices, including our magic squares with interchanged columns, none of the matrix conditions `A > B`, `A < B`, or `A == B` is true for *all* elements and so the `else` clause is executed:

```
if A > B
   'greater'
```

```
elseif A < B
   'less'
elseif A == B
   'equal'
else
   error('Unexpected situation')
end
```

Several functions are helpful for reducing the results of matrix comparisons to scalar conditions for use with `if`, including

```
isequal
isempty
all
any
```

switch and case

The `switch` statement executes groups of statements based on the value of a variable or expression. The keywords `case` and `otherwise` delineate the groups. Only the first matching case is executed. There must always be an `end` to match the `switch`.

The logic of the magic squares algorithm can also be described by

```
switch (rem(n,4)==0) + (rem(n,2)==0)
   case 0
      M = odd_magic(n)
   case 1
      M = single_even_magic(n)
   case 2
      M = double_even_magic(n)
   otherwise
      error('This is impossible')
end
```

Note Unlike the C language `switch` statement, MATLAB `switch` does not fall through. If the first `case` statement is `true`, the other `case` statements do not execute. So, `break` statements are not required.

Loop Control – for, while, continue, break

This section covers those MATLAB functions that provide control over program loops.

for

The `for` loop repeats a group of statements a fixed, predetermined number of times. A matching `end` delineates the statements:

```
for n = 3:32
   r(n) = rank(magic(n));
end
r
```

The semicolon terminating the inner statement suppresses repeated printing, and the `r` after the loop displays the final result.

It is a good idea to indent the loops for readability, especially when they are nested:

```
for i = 1:m
   for j = 1:n
      H(i,j) = 1/(i+j);
   end
end
```

while

The `while` loop repeats a group of statements an indefinite number of times under control of a logical condition. A matching `end` delineates the statements.

Here is a complete program, illustrating `while`, `if`, `else`, and `end`, that uses interval bisection to find a zero of a polynomial:

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3-2*x-5;
   if sign(fx) == sign(fa)
      a = x; fa = fx;
```

```
   else
      b = x; fb = fx;
   end
end
x
```

The result is a root of the polynomial $x^3 - 2x - 5$, namely

```
x =
   2.09455148154233
```

The cautions involving matrix comparisons that are discussed in the section on the `if` statement also apply to the `while` statement.

continue

The `continue` statement passes control to the next iteration of the `for` loop or `while` loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for `continue` statements in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

The example below shows a `continue` loop that counts the lines of code in the file `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered:

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line,'%',1)
        continue
    end
    count = count + 1;
end
disp(sprintf('%d lines',count));
```

break

The `break` statement lets you exit early from a `for` loop or `while` loop. In nested loops, `break` exits from the innermost loop only.

Here is an improvement on the example from the previous section. Why is this use of `break` a good idea?

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
   x = (a+b)/2;
   fx = x^3-2*x-5;
   if fx == 0
      break
   elseif sign(fx) == sign(fa)
      a = x; fa = fx;
   else
      b = x; fb = fx;
   end
end
x
```

Error Control – try, catch

This section covers those MATLAB functions that provide error handling control.

try

The general form of a `try-catch` statement sequence is

```
try
   statement
   ...
   statement
catch
   statement
   ...
   statement
end
```

In this sequence the statements between `try` and `catch` are executed until an error occurs. The statements between `catch` and `end` are then executed. Use `lasterr` to see the cause of the error. If an error occurs between `catch` and `end`, MATLAB terminates execution unless another `try-catch` sequence has been established.

Program Termination – return

This section covers the MATLAB `return` function that enables you to terminate your program before it runs to completion.

return

`return` terminates the current sequence of commands and returns control to the invoking function or to the keyboard. `return` is also used to terminate `keyboard` mode. A called function normally transfers control to the function that invoked it when it reaches the end of the function. You can insert a `return` statement within the called function to force an early termination and to transfer control to the invoking function.

Other Data Structures

In this section...

Multidimensional Arrays

Multidimensional arrays in MATLAB are arrays with more than two subscripts. One way of creating a multidimensional array is by calling `zeros`, `ones`, `rand`, or `randn` with more than two arguments. For example,

```
R = randn(3,4,5);
```

creates a 3-by-4-by-5 array with a total of `3*4*5 = 60` normally distributed random elements.

A three-dimensional array might represent three-dimensional physical data, say the temperature in a room, sampled on a rectangular grid. Or it might represent a sequence of matrices, $A^{(k)}$, or samples of a time-dependent matrix, $A(t)$. In these latter cases, the (i, j)th element of the kth matrix, or the t_kth matrix, is denoted by `A(i,j,k)`.

MATLAB and Dürer's versions of the magic square of order 4 differ by an interchange of two columns. Many different magic squares can be generated by interchanging columns. The statement

```
p = perms(1:4);
```

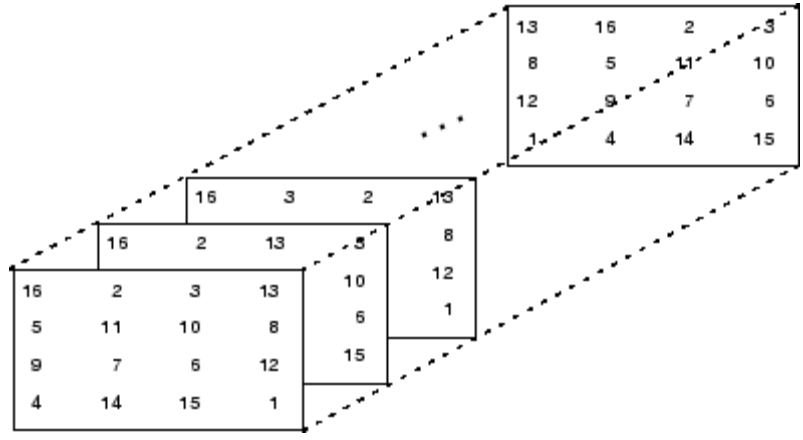
generates the 4! = 24 permutations of `1:4`. The `k`th permutation is the row vector `p(k,:)`. Then

```
A = magic(4);
M = zeros(4,4,24);

for k = 1:24
```

```
    M(:,:,k) = A(:,p(k,:));
end
```

stores the sequence of 24 magic squares in a three-dimensional array, `M`. The size of `M` is

```
size(M)

ans =
     4     4    24
```



Note The order of the matrices shown in this illustration might differ from your results. The `perms` function always returns all permutations of the input vector, but the order of the permutations might be different for different MATLAB versions.

The statement

```
sum(M,d)
```

computes sums by varying the `d`th subscript. So

```
sum(M,1)
```

is a 1-by-4-by-24 array containing 24 copies of the row vector

```
   34    34    34    34
```

and

```
sum(M,2)
```

is a 4-by-1-by-24 array containing 24 copies of the column vector

```
34
34
34
34
```

Finally,

```
S = sum(M,3)
```

adds the 24 matrices in the sequence. The result has size 4-by-4-by-1, so it looks like a 4-by-4 array:

```
S =
   204   204   204   204
   204   204   204   204
   204   204   204   204
   204   204   204   204
```

Cell Arrays

Cell arrays in MATLAB are multidimensional arrays whose elements are copies of other arrays. A cell array of empty matrices can be created with the `cell` function. But, more often, cell arrays are created by enclosing a miscellaneous collection of things in curly braces, `{}`. The curly braces are also used with subscripts to access the contents of various cells. For example,

```
C = {A sum(A) prod(prod(A))}
```

produces a 1-by-3 cell array. The three cells contain the magic square, the row vector of column sums, and the product of all its elements. When `C` is displayed, you see

```
C =
    [4x4 double]    [1x4 double]    [20922789888000]
```

This is because the first two cells are too large to print in this limited space, but the third cell contains only a single number, 16!, so there is room to print it.

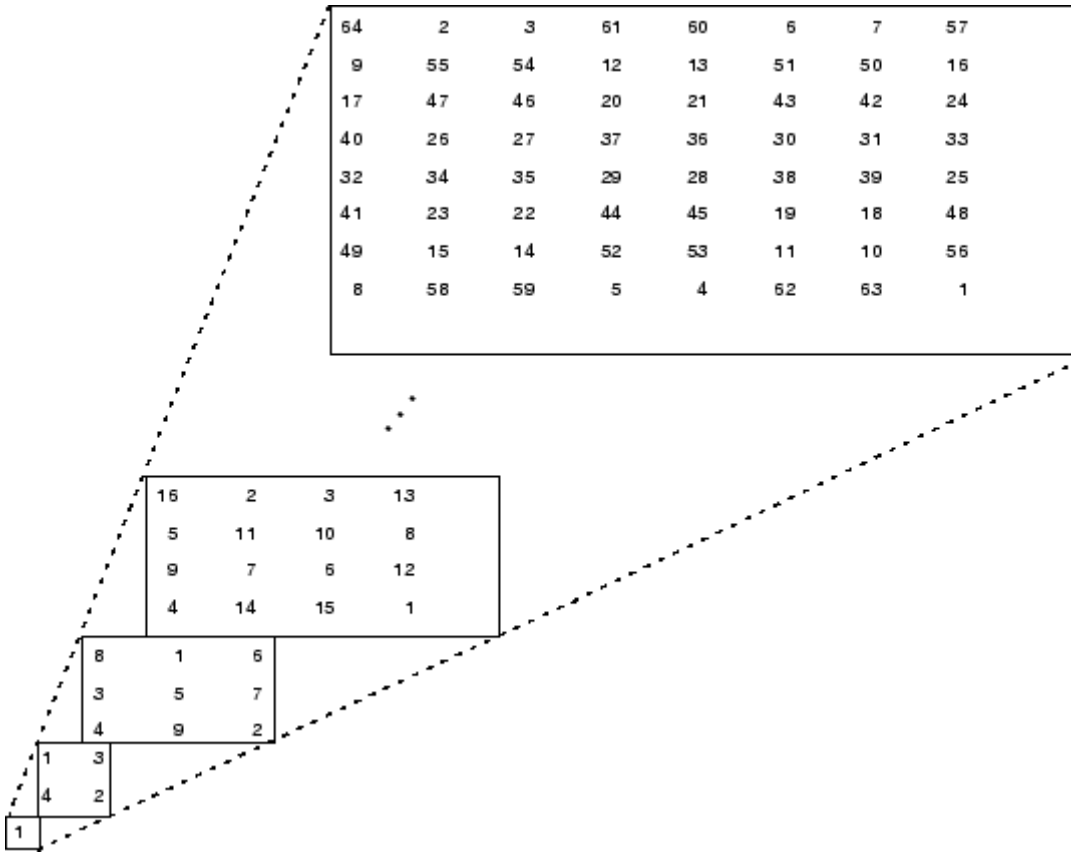
Here are two important points to remember. First, to retrieve the contents of one of the cells, use subscripts in curly braces. For example, `C{1}` retrieves the magic square and `C{3}` is 16!. Second, cell arrays contain *copies* of other arrays, not *pointers* to those arrays. If you subsequently change `A`, nothing happens to `C`.

You can use three-dimensional arrays to store a sequence of matrices of the *same* size. Cell arrays can be used to store a sequence of matrices of *different* sizes. For example,

```
M = cell(8,1);
for n = 1:8
   M{n} = magic(n);
end
M
```

produces a sequence of magic squares of different order:

```
M =
    [           1]
    [ 2x2  double]
    [ 3x3  double]
    [ 4x4  double]
    [ 5x5  double]
    [ 6x6  double]
    [ 7x7  double]
    [ 8x8  double]
```



You can retrieve the 4-by-4 magic square matrix with

```
M{4}
```

Characters and Text

Enter text into MATLAB using single quotes. For example,

```
s = 'Hello'
```

The result is not the same kind of numeric matrix or array you have been dealing with up to now. It is a 1-by-5 character array.

Internally, the characters are stored as numbers, but not in floating-point format. The statement

```
a = double(s)
```

converts the character array to a numeric matrix containing floating-point representations of the ASCII codes for each character. The result is

```
a =
    72    101    108    108    111
```

The statement

```
s = char(a)
```

reverses the conversion.

Converting numbers to characters makes it possible to investigate the various fonts available on your computer. The printable characters in the basic ASCII character set are represented by the integers `32:127`. (The integers less than 32 represent nonprinting control characters.) These integers are arranged in an appropriate 6-by-16 array with

```
F = reshape(32:127,16,6)';
```

The printable characters in the extended ASCII character set are represented by `F+128`. When these integers are interpreted as characters, the result depends on the font currently being used. Type the statements

```
char(F)
char(F+128)
```

and then vary the font being used for the Command Window. Select **Preferences** from the **File** menu to change the font. If you include tabs in lines of code, use a fixed-width font, such as Monospaced, to align the tab positions on different lines.

Concatenation with square brackets joins text variables together into larger strings. The statement

```
h = [s, ' world']
```

joins the strings horizontally and produces

```
h =
   Hello world
```

The statement

```
v = [s; 'world']
```

joins the strings vertically and produces

```
v =
   Hello
   world
```

Note that a blank has to be inserted before the `'w'` in `h` and that both words in `v` have to have the same length. The resulting arrays are both character arrays; `h` is 1-by-11 and `v` is 2-by-5.

To manipulate a body of text containing lines of different lengths, you have two choices—a padded character array or a cell array of strings. When creating a character array, you must make each row of the array the same length. (Pad the ends of the shorter rows with spaces.) The `char` function does this padding for you. For example,

```
S = char('A','rolling','stone','gathers','momentum.')
```

produces a 5-by-9 character array:

```
S =
A
rolling
stone
gathers
momentum.
```

Alternatively, you can store the text in a cell array. For example,

```
C = {'A';'rolling';'stone';'gathers';'momentum.'}
```

creates a 5-by-1 cell array that requires no padding because each row of the array can have a different length:

```
C =
    'A'
    'rolling'
    'stone'
    'gathers'
    'momentum.'
```

You can convert a padded character array to a cell array of strings with

```
C = cellstr(S)
```

and reverse the process with

```
S = char(C)
```

Structures

Structures are multidimensional MATLAB arrays with elements accessed by textual *field designators*. For example,

```
S.name = 'Ed Plum';
S.score = 83;
S.grade = 'B+'
```

creates a scalar structure with three fields:

```
S =
     name: 'Ed Plum'
    score: 83
    grade: 'B+'
```

Like everything else in MATLAB, structures are arrays, so you can insert additional elements. In this case, each element of the array is a structure with several fields. The fields can be added one at a time,

```
S(2).name = 'Toni Miller';
S(2).score = 91;
S(2).grade = 'A-';
```

or an entire element can be added with a single statement:

```
S(3) = struct('name','Jerry Garcia',...
              'score',70,'grade','C')
```

Now the structure is large enough that only a summary is printed:

```
S =
1x3 struct array with fields:
    name
    score
    grade
```

There are several ways to reassemble the various fields into other MATLAB arrays. They are mostly based on the notation of a *comma-separated list*. If you type

```
S.score
```

it is the same as typing

```
S(1).score, S(2).score, S(3).score
```

which is a comma-separated list.

If you enclose the expression that generates such a list within square brackets, MATLAB stores each item from the list in an array. In this example, MATLAB creates a numeric row vector containing the `score` field of each element of structure array `S`:

```
scores = [S.score]
scores =
    83    91    70

avg_score = sum(scores)/length(scores)
avg_score =
   81.3333
```

To create a character array from one of the text fields (`name`, for example), call the `char` function on the comma-separated list produced by `S.name`:

```
names = char(S.name)
names =
   Ed Plum
   Toni Miller
   Jerry Garcia
```

Similarly, you can create a cell array from the `name` fields by enclosing the list-generating expression within curly braces:

```
names = {S.name}
names =
    'Ed Plum'    'Toni Miller'    'Jerry Garcia'
```

To assign the fields of each element of a structure array to separate variables outside of the structure, specify each output to the left of the equals sign, enclosing them all within square brackets:

```
[N1 N2 N3] = S.name
N1 =
   Ed Plum
N2 =
   Toni Miller
N3 =
   Jerry Garcia
```

Dynamic Field Names

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time. The dot-parentheses syntax shown here makes `expression` a dynamic field name:

```
structName.(expression)
```

Index into this field using the standard MATLAB indexing syntax. For example, to evaluate `expression` into a field name and obtain the values of that field at columns `1` through `25` of row `7`, use

```
structName.(expression)(7,1:25)
```

Dynamic Field Names Example. The `avgscore` function shown below computes an average test score, retrieving information from the `testscores` structure using dynamic field names:

```
function avg = avgscore(testscores, student, first, last)
for k = first:last
   scores(k) = testscores.(student).week(k);
```

```
end
avg = sum(scores)/(last - first + 1);
```

You can run this function using different values for the dynamic field `student`. First, initialize the structure that contains scores for a 25-week period:

```
testscores.Ann_Lane.week(1:25) = ...
  [95 89 76 82 79 92 94 92 89 81 75 93 ...
   85 84 83 86 85 90 82 82 84 79 96 88 98];

testscores.William_King.week(1:25) = ...
  [87 80 91 84 99 87 93 87 97 87 82 89 ...
   86 82 90 98 75 79 92 84 90 93 84 78 81];
```

Now run `avgscore`, supplying the students name fields for the `testscores` structure at runtime using dynamic field names:

```
avgscore(testscores, 'Ann_Lane', 7, 22)
ans =
   85.2500

avgscore(testscores, 'William_King', 7, 22)
ans =
   87.7500
```

Scripts and Functions

In this section...

Overview

MATLAB is a powerful programming language as well as an interactive computational environment. Files that contain code in the MATLAB language are called M-files. You create M-files using a text editor, then use them as you would any other MATLAB function or command.

There are two kinds of M-files:

- Scripts, which do not accept input arguments or return output arguments. They operate on data in the workspace.
- Functions, which can accept input arguments and return output arguments. Internal variables are local to the function.

If you're a new MATLAB programmer, just create the M-files that you want to try out in the current directory. As you develop more of your own M-files, you will want to organize them into other directories and personal toolboxes that you can add to your MATLAB search path.

If you duplicate function names, MATLAB executes the one that occurs first in the search path.

To view the contents of an M-file, for example, `myfunction.m`, use

```
type myfunction
```

Scripts

When you invoke a *script*, MATLAB simply executes the commands found in the file. Scripts can operate on existing data in the workspace, or they can create new data on which to operate. Although scripts do not return output arguments, any variables that they create remain in the workspace, to be used in subsequent computations. In addition, scripts can produce graphical output using functions like `plot`.

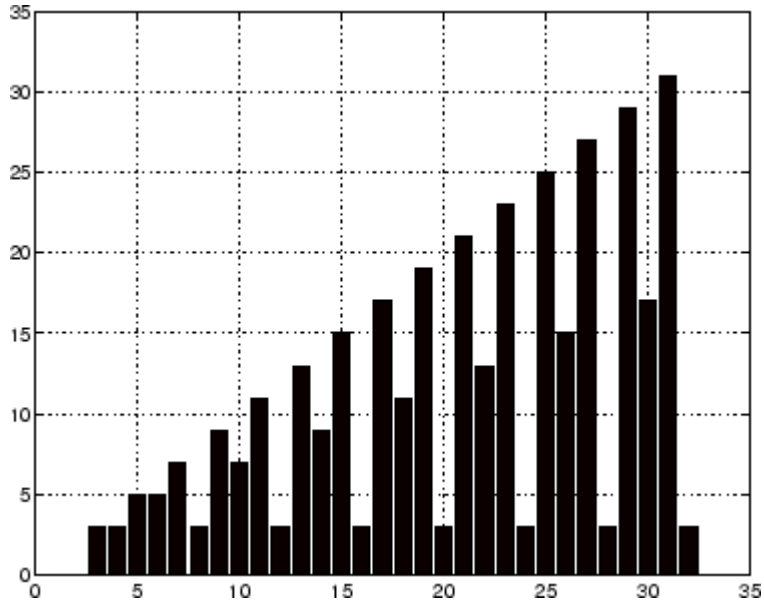
For example, create a file called `magicrank.m` that contains these MATLAB commands:

```
% Investigate the rank of magic squares
r = zeros(1,32);
for n = 3:32
   r(n) = rank(magic(n));
end
r
bar(r)
```

Typing the statement

```
magicrank
```

causes MATLAB to execute the commands, compute the rank of the first 30 magic squares, and plot a bar graph of the result. After execution of the file is complete, the variables `n` and `r` remain in the workspace.



Functions

Functions are M-files that can accept input arguments and return output arguments. The names of the M-file and of the function should be the same. Functions operate on variables within their own workspace, separate from the workspace you access at the MATLAB command prompt.

A good example is provided by `rank`. The M-file `rank.m` is available in the directory

```
toolbox/matlab/matfun
```

You can see the file with

```
type rank
```

Here is the file:

```
function r = rank(A,tol)
%   RANK Matrix rank.
%   RANK(A) provides an estimate of the number of linearly
```

```
%   independent rows or columns of a matrix A.
%   RANK(A,tol) is the number of singular values of A
%   that are larger than tol.
%   RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

s = svd(A);
if nargin==1
   tol = max(size(A)') * max(s) * eps;
end
r = sum(s > tol);
```

The first line of a function M-file starts with the keyword `function`. It gives the function name and order of arguments. In this case, there are up to two input arguments and one output argument.

The next several lines, up to the first blank or executable line, are comment lines that provide the help text. These lines are printed when you type

```
help rank
```

The first line of the help text is the H1 line, which MATLAB displays when you use the `lookfor` command or request `help` on a directory.

The rest of the file is the executable MATLAB code defining the function. The variable `s` introduced in the body of the function, as well as the variables on the first line, `r`, `A` and `tol`, are all *local* to the function; they are separate from any variables in the MATLAB workspace.

This example illustrates one aspect of MATLAB functions that is not ordinarily found in other programming languages—a variable number of arguments. The `rank` function can be used in several different ways:

```
rank(A)
r = rank(A)
r = rank(A,1.e-6)
```

Many M-files work this way. If no output argument is supplied, the result is stored in `ans`. If the second input argument is not supplied, the function computes a default value. Within the body of the function, two quantities named `nargin` and `nargout` are available that tell you the number of input

and output arguments involved in each particular use of the function. The `rank` function uses `nargin`, but does not need to use `nargout`.

Types of Functions

MATLAB offers several different types of functions to use in your programming.

Anonymous Functions

An *anonymous function* is a simple form of MATLAB function that does not require an M-file. It consists of a single MATLAB expression and any number of input and output arguments. You can define an anonymous function right at the MATLAB command line, or within an M-file function or script. This gives you a quick means of creating simple functions without having to create M-files each time.

The syntax for creating an anonymous function from an expression is

```
f = @(arglist)expression
```

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Primary and Subfunctions

All functions that are not anonymous must be defined within an M-file. Each M-file has a required *primary function* that appears first in the file, and any number of *subfunctions* that follow the primary. Primary functions have a wider scope than subfunctions. That is, primary functions can be invoked from outside of their M-file (from the MATLAB command line or from functions in

other M-files) while subfunctions cannot. Subfunctions are visible only to the primary function and other subfunctions within their own M-file.

The `rank` function shown in the section on "Functions" on page 4-22 is an example of a primary function.

Private Functions

A *private function* is a type of primary M-file function. Its unique characteristic is that it is visible only to a limited group of other functions. This type of function can be useful if you want to limit access to a function, or when you choose not to expose the implementation of a function.

Private functions reside in subdirectories with the special name `private`. They are visible only to functions in the parent directory. For example, assume the directory `newmath` is on the MATLAB search path. A subdirectory of `newmath` called `private` can contain functions that only the functions in `newmath` can call.

Because private functions are invisible outside the parent directory, they can use the same names as functions in other directories. This is useful if you want to create your own version of a particular function while retaining the original in another directory. Because MATLAB looks for private functions before standard M-file functions, it will find a private function named `test.m` before a nonprivate M-file named `test.m`.

Nested Functions

You can define functions within the body of any MATLAB M-file function. These are said to be *nested* within the outer function. A nested function contains any or all of the components of any other M-file function. In this example, function `B` is nested in function `A`:

```
function x = A(p1, p2)
...
B(p2)
   function y = B(p3)
   ...
   end
...
end
```

Like other functions, a nested function has its own workspace where variables used by the function are stored. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

Function Overloading

Overloaded functions act the same way as overloaded functions in most computer languages. Overloaded functions are useful when you need to create a function that responds to different types of inputs accordingly. For instance, you might want one of your functions to accept both double-precision and integer input, but to handle each type somewhat differently. You can make this difference invisible to the user by creating two separate functions having the same name, and designating one to handle `double` types and one to handle integers. When you call the function, MATLAB chooses which M-file to dispatch to based on the type of the input arguments.

Global Variables

If you want more than one function to share a single copy of a variable, simply declare the variable as `global` in all the functions. Do the same thing at the command line if you want the base workspace to access the variable. The global declaration must occur before the variable is actually used in a function. Although it is not required, using capital letters for the names of global variables helps distinguish them from other variables. For example, create an M-file called `falling.m`:

```
function h = falling(t)
global GRAVITY
h = 1/2*GRAVITY*t.^2;
```

Then interactively enter the statements

```
global GRAVITY
GRAVITY = 32;
y = falling((0:.1:5)');
```

The two global statements make the value assigned to `GRAVITY` at the command prompt available inside the function. You can then modify `GRAVITY` interactively and obtain new solutions without editing any files.

Passing String Arguments to Functions

You can write MATLAB functions that accept string arguments without the parentheses and quotes. That is, MATLAB interprets

```
foo a b c
```

as

```
foo('a','b','c')
```

However, when you use the unquoted form, MATLAB cannot return output arguments. For example,

```
legend apples oranges
```

creates a legend on a plot using the strings `apples` and `oranges` as labels. If you want the `legend` command to return its output arguments, then you must use the quoted form:

```
[legh,objh] = legend('apples','oranges');
```

In addition, you must use the quoted form if any of the arguments is not a string.

Caution While the unquoted syntax is convenient, in some cases it can be used incorrectly without causing MATLAB to generate an error.

Constructing String Arguments in Code

The quoted form enables you to construct string arguments within the code. The following example processes multiple data files, `August1.dat`, `August2.dat`, and so on. It uses the function `int2str`, which converts an integer to a character, to build the filename:

```
for d = 1:31
```

```
   s = ['August' int2str(d) '.dat'];
   load(s)
   % Code to process the contents of the d-th file
end
```

The eval Function

The `eval` function works with text variables to implement a powerful text macro facility. The expression or statement

```
eval(s)
```

uses the MATLAB interpreter to evaluate the expression or execute the statement contained in the text string `s`.

The example of the previous section could also be done with the following code, although this would be somewhat less efficient because it involves the full interpreter, not just a function call:

```
for d = 1:31
   s = ['load August' int2str(d) '.dat'];
   eval(s)
   % Process the contents of the d-th file
end
```

Function Handles

You can create a handle to any MATLAB function and then use that handle as a means of referencing the function. A function handle is typically passed in an argument list to other functions, which can then execute, or *evaluate*, the function using the handle.

Construct a function handle in MATLAB using the *at* sign, `@`, before the function name. The following example creates a function handle for the `sin` function and assigns it to the variable `fhandle`:

```
fhandle = @sin;
```

You can call a function by means of its handle in the same way that you would call the function using its name. The syntax is

```
fhandle(arg1, arg2, ...);
```

The function `plot_fhandle`, shown below, receives a function handle and data, generates y-axis data using the function handle, and plots it:

```
function x = plot_fhandle(fhandle, data)
plot(data, fhandle(data))
```

When you call `plot_fhandle` with a handle to the `sin` function and the argument shown below, the resulting evaluation produces a sine wave plot:

```
plot_fhandle(@sin, -pi:0.01:pi)
```

Function Functions

A class of functions called "function functions" works with nonlinear functions of a scalar variable. That is, one function works on another function. The function functions include

- Zero finding
- Optimization
- Quadrature
- Ordinary differential equations

MATLAB represents the nonlinear function by a function M-file. For example, here is a simplified version of the function `humps` from the `matlab/demos` directory:

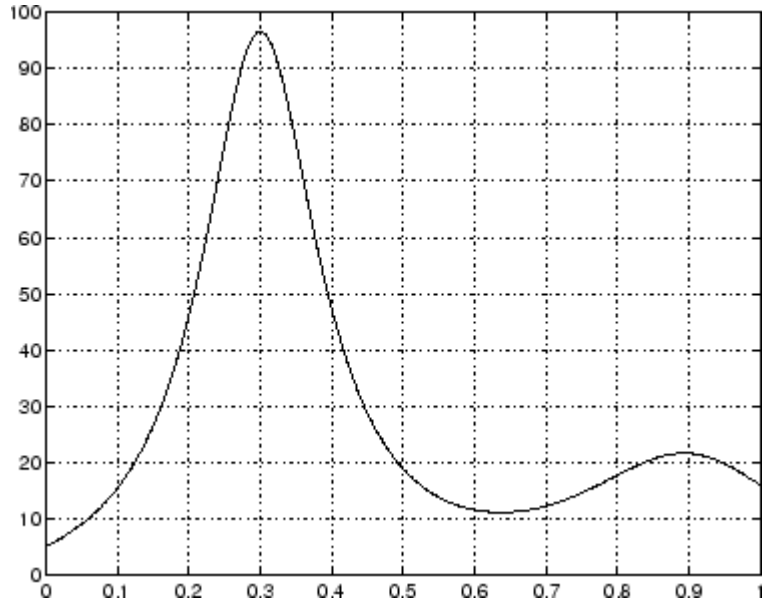
```
function y = humps(x)
y = 1./((x-.3).^2 + .01) + 1./((x-.9).^2 + .04) - 6;
```

Evaluate this function at a set of points in the interval $0 \le x \le 1$ with

```
x = 0:.002:1;
y = humps(x);
```

Then plot the function with

```
plot(x,y)
```



The graph shows that the function has a local minimum near $x = 0.6$. The function `fminsearch` finds the *minimizer*, the value of x where the function takes on this minimum. The first argument to `fminsearch` is a function handle to the function being minimized and the second argument is a rough guess at the location of the minimum:

```
p = fminsearch(@humps,.5)
p =
    0.6370
```

To evaluate the function at the minimizer,

```
humps(p)

ans =
   11.2528
```

Numerical analysts use the terms *quadrature* and *integration* to distinguish between numerical approximation of definite integrals and numerical integration of ordinary differential equations. MATLAB quadrature routines are `quad` and `quadl`. The statement

```
Q = quadl(@humps,0,1)
```

computes the area under the curve in the graph and produces

```
Q =
   29.8583
```

Finally, the graph shows that the function is never zero on this interval. So, if you search for a zero with

```
z = fzero(@humps,.5)
```

you will find one outside the interval

```
z =
   -0.1316
```

Vectorization

One way to make your MATLAB programs run faster is to vectorize the algorithms you use in constructing the programs. Where other programming languages might use `for` loops or `DO` loops, MATLAB can use vector or matrix operations. A simple example involves creating a table of logarithms:

```
x = .01;
for k = 1:1001
   y(k) = log10(x);
   x = x + .01;
end
```

A vectorized version of the same code is

```
x = .01:.01:10;
y = log10(x);
```

For more complicated code, vectorization options are not always so obvious.

For More Information See “Improving Performance and Memory Usage” in the MATLAB Programming documentation for other techniques that you can use.

Preallocation

If you cannot vectorize a piece of code, you can make your `for` loops go faster by preallocating any vectors or arrays in which output results are stored. For example, this code uses the function `zeros` to preallocate the vector created in the `for` loop. This makes the `for` loop execute significantly faster:

```
r = zeros(32,1);
for n = 1:32
    r(n) = rank(magic(n));
end
```

Without the preallocation in the previous example, the MATLAB interpreter enlarges the `r` vector by one element each time through the loop. Vector preallocation eliminates this step and results in faster execution.

5

Data Analysis

Introduction

Every data analysis has some standard components:

- Preprocessing — Consider outliers and missing values, and smooth data to identify possible models.
- Summarizing — Compute basic statistics to describe the overall location, scale, and shape of the data.
- Visualizing — Plot data to identify patterns and trends.
- Modeling — Give data trends fuller descriptions, suitable for predicting new values.

Data analysis moves among these components with two basic goals in mind:

1. Describe the patterns in the data with simple models that lead to accurate predictions.
2. Understand the relationships among variables that lead to the model.

This section of the Getting Started guide explains how to use MATLAB to carry out a basic data analysis.

Preprocessing Data

In this section...

Overview

Begin a data analysis by loading data into suitable MATLAB container variables and sorting out the "good" data from the "bad." This is a preliminary step that assures meaningful conclusions in subsequent parts of the analysis.

Note This section begins a data analysis that is continued in "Summarizing Data" on page 5-10, "Visualizing Data" on page 5-14, and "Modeling Data" on page 5-19.

Loading the Data

Begin by loading the data in `count.dat`:

```
load count.dat
```

The 24-by-3 array `count` contains hourly traffic counts (the rows) at three intersections (the columns) for a single day.

See "MATLAB for Data Analysis" and "Importing and Exporting Data" in the MATLAB Data Analysis documentation for more information on storing data in MATLAB variables for analysis.

Missing Data

In MATLAB, `NaN` (Not a Number) values represent missing data. `NaN` values allow variables with missing data to maintain their structure—in this case, 24-by-1 vectors with consistent indexing across all three intersections.

Check the data at the third intersection for `NaN` values using the MATLAB `isnan` function:

```
c3 = count(:,3); % Data at intersection 3
c3NaNCount = sum(isnan(c3))
c3NaNCount =
     0
```

`isnan` returns a logical vector the same size as `c3`, with entries indicating the presence (`1`) or absence (`0`) of `NaN` values for each of the 24 elements in the data. In this case, the logical values sum to `0`, so there are no `NaN` values in the data.

`NaN` values are introduced into the data in the section on “Outliers” on page 5-4.

See “Removing and Interpolating Missing Values” in the MATLAB Data Analysis documentation for more information on handling missing data in MATLAB.

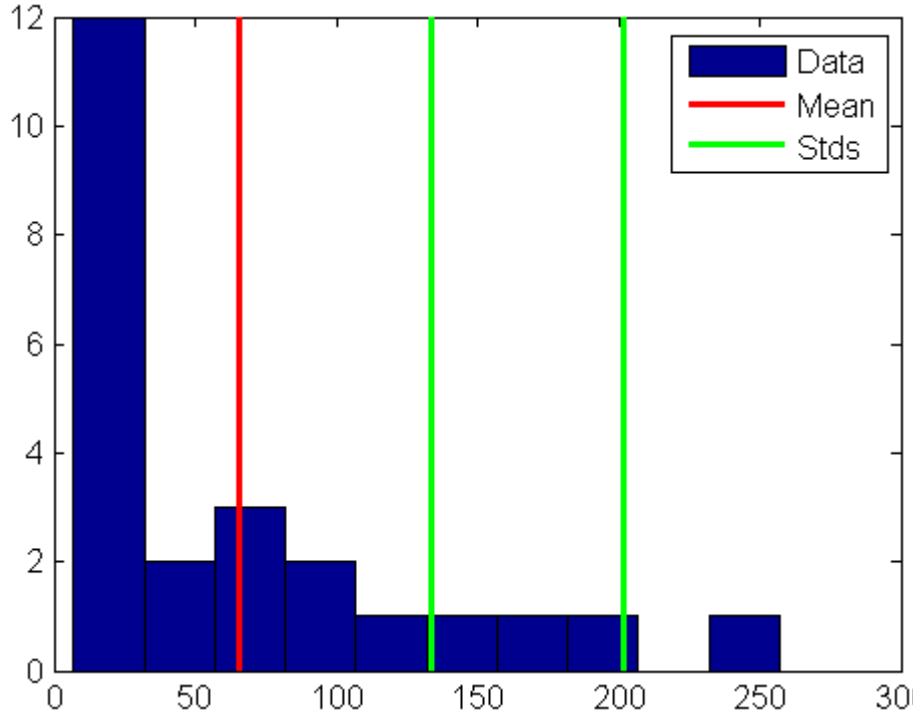
Outliers

Outliers are data values that are dramatically different from patterns in the rest of the data. They may be due to measurement error, or they may represent significant features in the data. Identifying outliers, and deciding what to do with them, depends on an understanding of the data and its source.

One common method for identifying outliers is to look for values more than a certain number of standard deviations σ from the mean μ. The following code plots a histogram of the data at the third intersection together with lines at μ and $\mu + n\sigma$, for $n = 1, 2$:

```
bin_counts = hist(c3); % Histogram bin counts
N = max(bin_counts); % Maximum bin count
mu3 = mean(c3); % Data mean
```

```
sigma3 = std(c3); % Data standard deviation

hist(c3) % Plot histogram
hold on
plot([mu3 mu3],[0 N],'r','LineWidth',2) % Mean
X = repmat(mu3+(1:2)*sigma3,2,1);
Y = repmat([0;N],1,2);
plot(X,Y,'g','LineWidth',2) % Standard deviations
legend('Data','Mean','Stds')
hold off
```



The plot shows that some of the data are more than two standard deviations above the mean. If you identify these data as errors (not features), replace them with `NaN` values as follows:

```
outliers = (c3 - mu3) > 2*sigma3;
c3m = c3; % Copy c3 to c3m
```

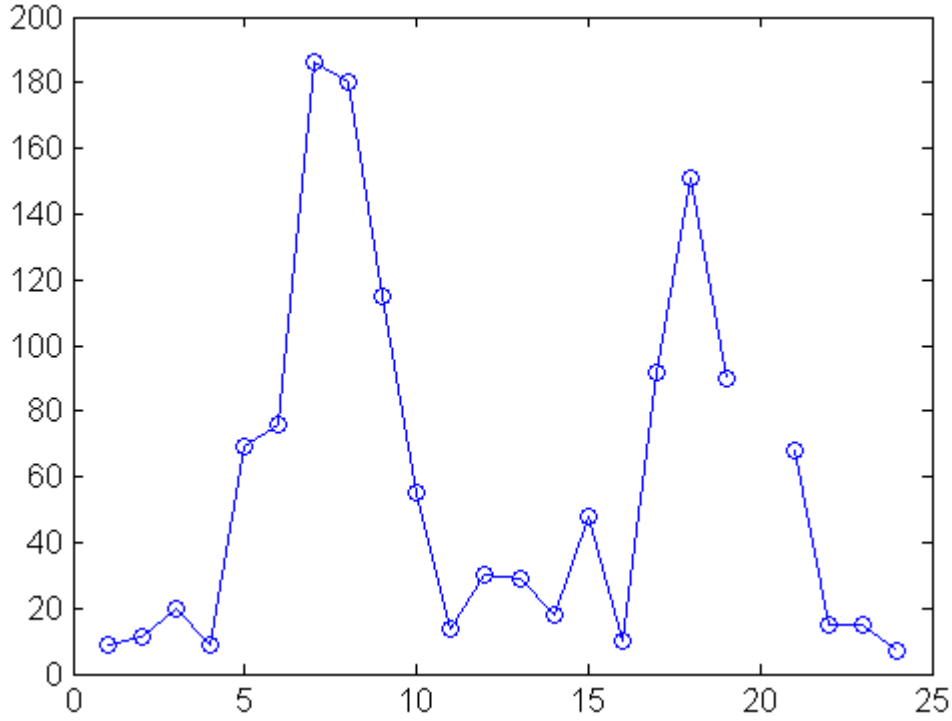
```
c3m(outliers) = NaN; % Add NaN values
```

See “Removing Outliers” in the MATLAB Data Analysis documentation for more information on handling outliers in MATLAB.

Smoothing and Filtering

A time-series plot of the data at the third intersection (with the outlier removed in “Outliers” on page 5-4) looks like this:

```
plot(c3m,'o-')
hold on
```



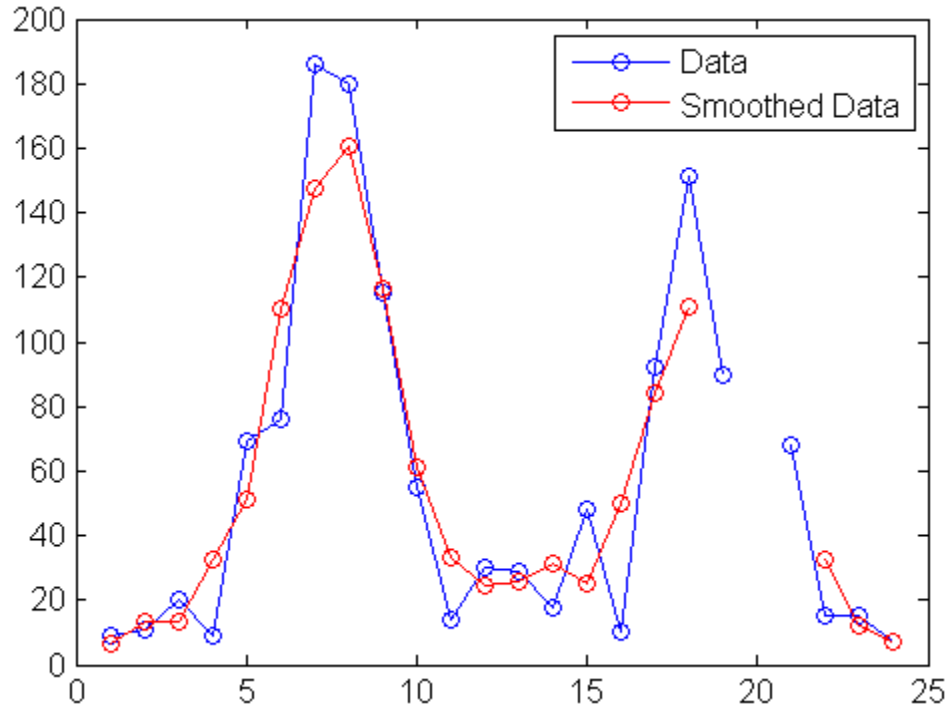
The `NaN` value at hour 20 appears as a gap in the plot. This handling of `NaN` values is typical of MATLAB plotting functions.

Noisy data shows random variations about expected values. You may want to smooth the data to reveal its main features before building a model. Two basic assumptions underlie smoothing:

- The relationship between the predictor (time) and the response (traffic volume) is smooth.
- The smoothing algorithm results in values that are better estimates of expected values because the noise has been reduced.

Apply a simple moving average smoother to the data using the MATLAB `convn` function:

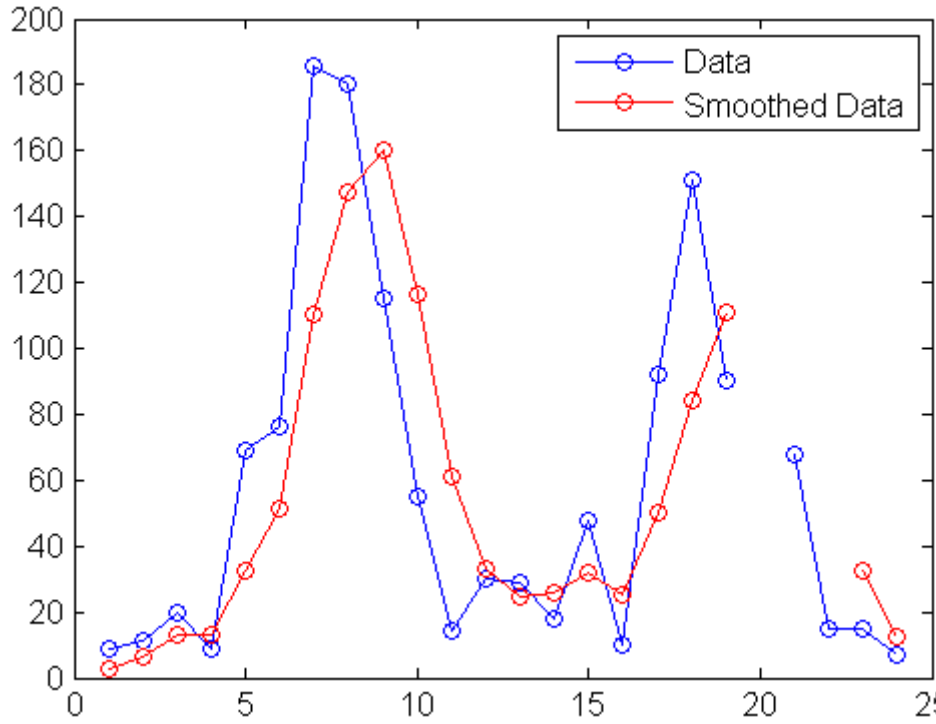
```
span = 3; % Size of the averaging window
window = ones(span,1)/span;
smoothed_c3m = convn(c3m,window,'same');

h = plot(smoothed_c3m,'ro-');
legend('Data','Smoothed Data')
```



The extent of the smoothing is controlled with the variable `span`. The averaging calculation returns `NaN` values whenever the smoothing window includes the `NaN` value in the data, thus increasing the size of the gap in the smoothed data.

The MATLAB `filter` function is also used for smoothing data:

```
smoothed2_c3m = filter(window,1,c3m);

delete(h)
plot(smoothed2_c3m,'ro-');
```



The smoothed data are shifted from the previous plot. `convn` with the `'same'` parameter returns the central part of the convolution, the same length as the data. `filter` returns the initial part of the convolution, the same length as the data. Otherwise, the algorithms are identical.

Smoothing estimates the center of the distribution of response values at each value of the predictor. It invalidates a basic assumption of many fitting algorithms, namely, that the errors at each value of the predictor are normally distributed. Accordingly, smoothed data should not be used to fit a model. Use smoothed data to identify a model.

See “Filtering Data” in the MATLAB Data Analysis documentation for more information on smoothing and filtering.

Summarizing Data

In this section...

Overview

MATLAB includes many functions for summarizing the overall location, scale, and shape of a data sample.

One of the advantages of working in MATLAB is that functions operate on entire arrays of data, not just on single scalar values. The functions are said to be *vectorized*. Vectorization allows for both efficient problem formulation, using array-based data, and efficient computation, using vectorized statistical functions.

Note This section continues the data analysis from "Preprocessing Data" on page 5-3.

Measures of Location

Summarize the location of a data sample by finding a "typical" value. Common measures of location or "central tendency" are computed by the MATLAB functions `mean`, `median`, and `mode`:

```
x1 = mean(count)
x1 =
   32.0000   46.5417   65.5833

x2 = median(count)
x2 =
   23.5000   36.0000   39.0000
```

```
x3 = mode(count)
x3 =
    11     9     9
```

Like all of the statistical functions in MATLAB, the functions above summarize data across observations (rows) while preserving variables (columns). The functions compute the location of the data at each of the three intersections in a single call.

Measures of Scale

There are many ways to measure the scale or "dispersion" of a data sample. The MATLAB functions `max`, `min`, `std`, and `var` compute some common measures:

```
dx1 = max(count)-min(count)
dx1 =
   107   136   250

dx2 = std(count)
dx2 =
   25.3703   41.4057   68.0281

dx3 = var(count)
dx3 =
  1.0e+003 *
    0.6437    1.7144    4.6278
```

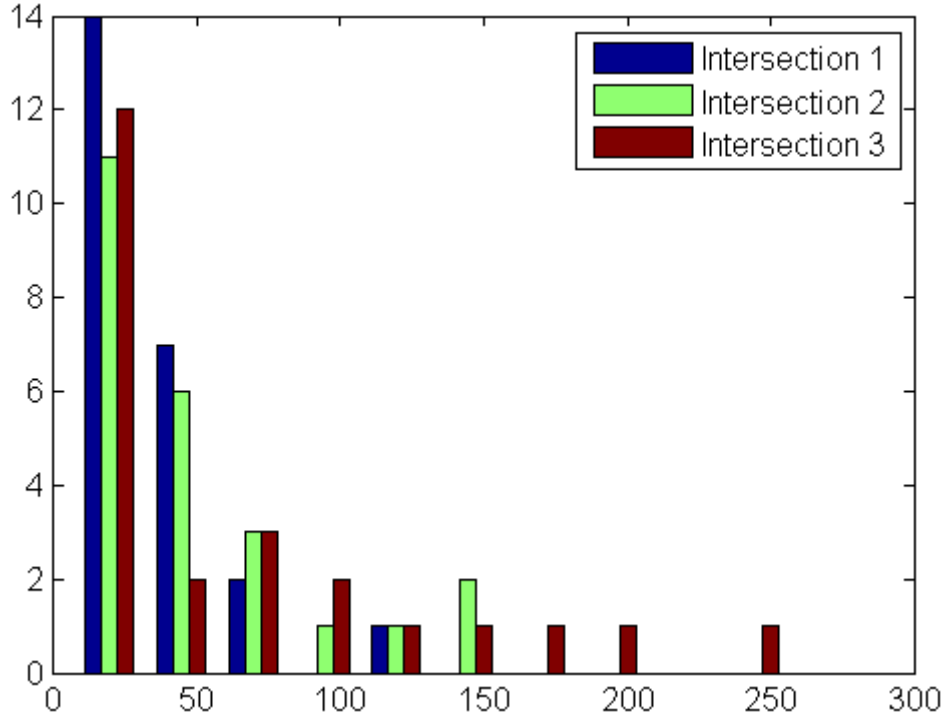
Like all of the statistical functions in MATLAB, the functions above summarize data across observations (rows) while preserving variables (columns). The functions compute the scale of the data at each of the three intersections in a single call.

Shape of a Distribution

The shape of a distribution is harder to summarize than its location or scale. The MATLAB `hist` function plots a histogram that provides a visual summary:

```
figure
hist(count)
```

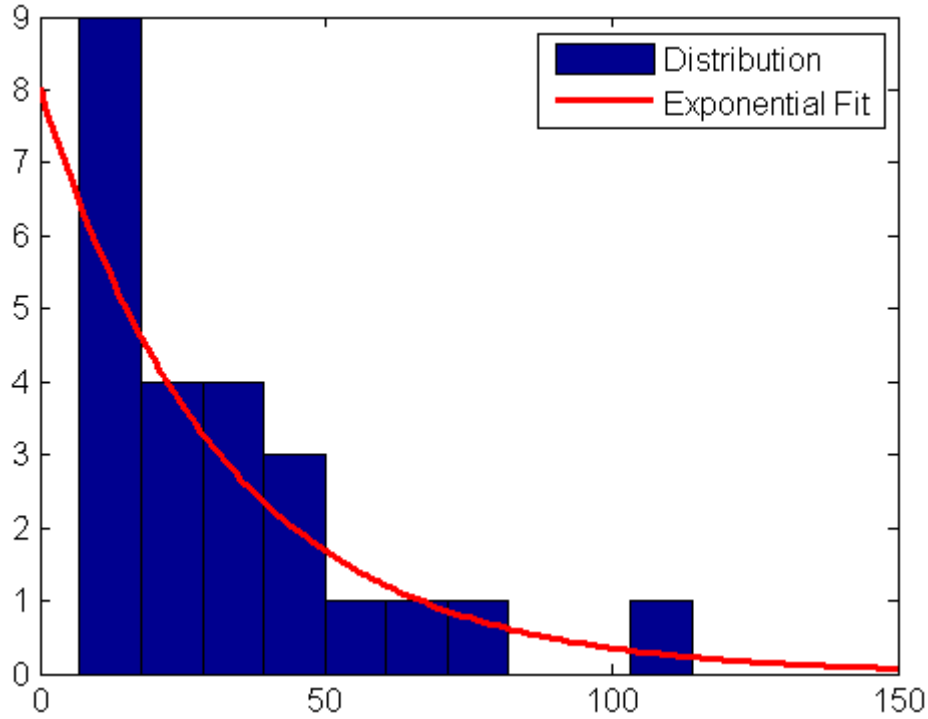
```
legend('Intersection 1',...
       'Intersection 2',...
       'Intersection 3')
```



Parametric models give analytic summaries of distribution shapes. Exponential distributions, with parameter `mu` given by the data mean, are a good choice for the traffic data:

```
c1 = count(:,1); % Data at intersection 1
[bin_counts,bin_locations] = hist(c1);
bin_width = bin_locations(2) - bin_locations(1);
hist_area = (bin_width)*(sum(bin_counts));

figure
hist(c1)
hold on
```

```
mu1 = mean(c1);
exp_pdf = @(t)(1/mu1)*exp(-t/mu1); % Integrates
                                   % to 1
t = 0:150;
y = exp_pdf(t);
plot(t,(hist_area)*y,'r','LineWidth',2)
legend('Distribution','Exponential Fit')
```



Methods for fitting general parametric models to data distributions are beyond the scope of this Getting Started guide. Functions for computing maximum likelihood estimates of distribution parameters are available in Statistics Toolbox.

See “Descriptive Statistics” in the MATLAB Data Analysis documentation for more information on summarizing data samples.

Visualizing Data

In this section...

Overview

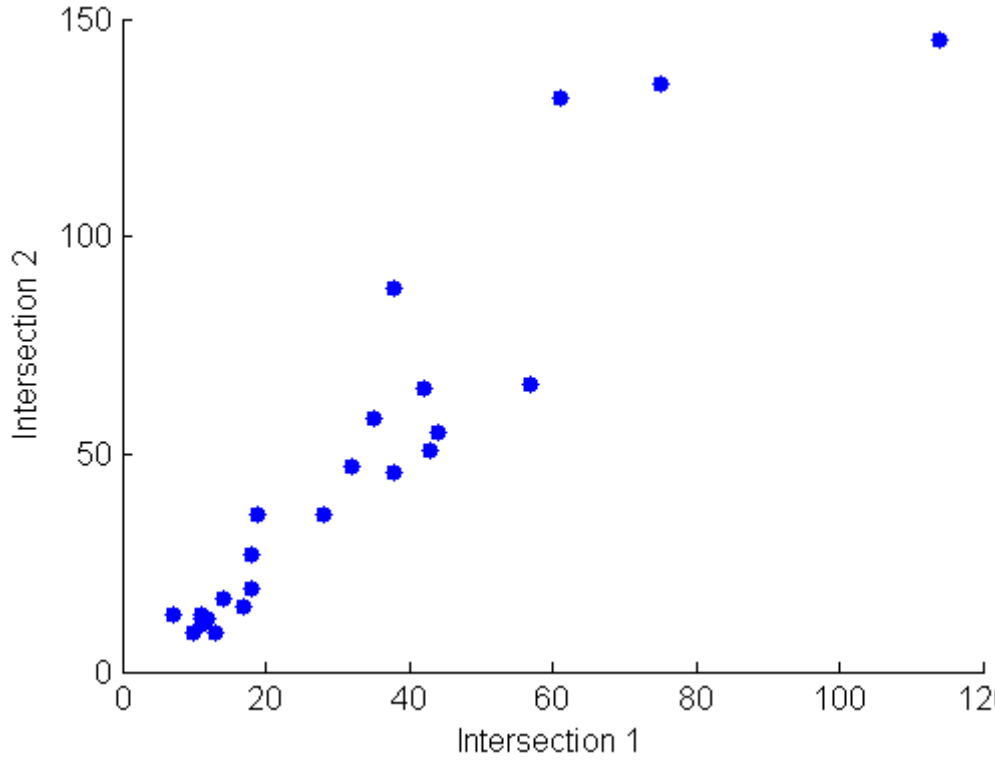
MATLAB provides many plots for visualizing data patterns and trends. Histograms and time-series plots of the traffic data are described in the sections on "Preprocessing Data" on page 5-3 and "Summarizing Data" on page 5-10. Scatter plots, described in this section, help to visualize relationships among the traffic data at different intersections.

Note This section continues the data analysis from "Summarizing Data" on page 5-10.

2-D Scatter Plots

A 2-D scatter plot, created with the MATLAB `scatter` function, shows the relationship between the traffic volume at the first two intersections:

```
c1 = count(:,1); % Data at intersection 1
c2 = count(:,2); % Data at intersection 2

figure
scatter(c1,c2,'filled')
xlabel('Intersection 1')
ylabel('Intersection 2')
```



The strength of the linear relationship between the two variables (how tightly the data lies along a least-squares line through the scatter) is measured by the *covariance*, computed by the MATLAB `cov` function:

```
C12 = cov([c1 c2])
C12 =
  1.0e+003 *
    0.6437    0.9802
    0.9802    1.7144
```

The results are displayed in a symmetric square matrix, with the covariance of the i th and j th variables in the (i, j)th position. The i th diagonal element is the variance of the i th variable.

Covariances have the disadvantage of depending on the units used to measure the individual variables. They are often divided by the standard deviations

of the variables to normalize values between +1 and –1. The results are *correlation coefficients*, computed by the MATLAB `corrcoef` function:

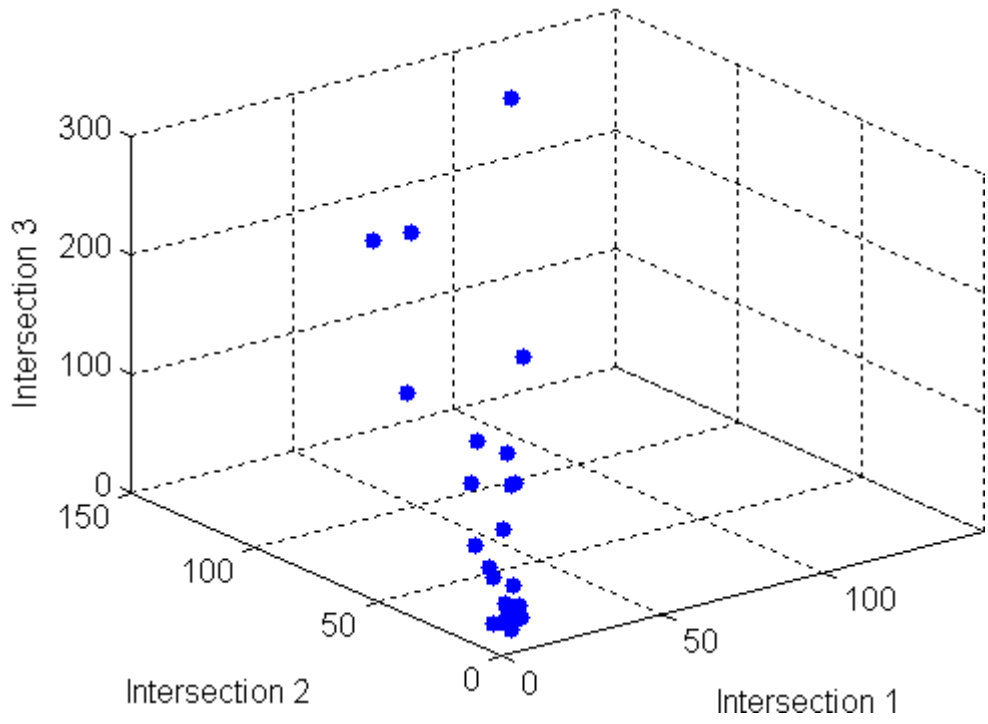
```
R12 = corrcoef([c1 c2])
R12 =
    1.0000    0.9331
    0.9331    1.0000

r12 = R12(1,2) % Correlation coefficient
r12 =
    0.9331

r12sq = r12^2 % Coefficient of determination
r12sq =
    0.8707
```

Because it is normalized, the value of the correlation coefficient is readily comparable to values for other pairs of intersections. Its square, the *coefficient of determination*, is the variance about the least-squares line divided by the variance about the mean. That is, it is the proportion of variation in the response (in this case, the traffic volume at intersection 2) that is eliminated or non-causally "explained" by a least-squares line through the scatter.

3-D Scatter Plots

A 3-D scatter plot, created with the MATLAB `scatter3` function, shows the relationship between the traffic volume at all three intersections:

```
figure
scatter3(c1,c2,c3,'filled')
xlabel('Intersection 1')
ylabel('Intersection 2')
zlabel('Intersection 3')
```



The strength of the linear relationship among the variables in the 3-D scatter is measured by computing eigenvalues of the covariance matrix with the MATLAB `eig` function:

```
vars = eig(cov([c1 c2 c3]))
vars =
  1.0e+003 *
    0.0442
    0.1118
    6.8300

explained = max(vars)/sum(vars)
explained =
    0.9777
```

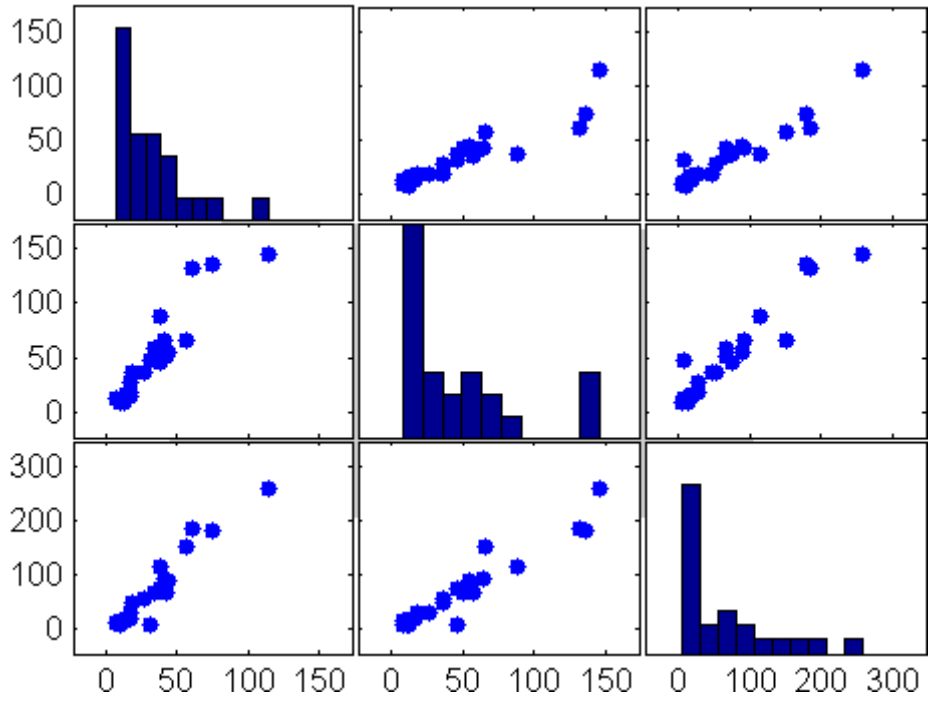
The eigenvalues are the variances along the *principal components* of the data. The variable `explained` measures the proportion of variation “explained” by

the first principal component, along the axis of the data. Unlike the coefficient of determination for 2-D scatters, this measure does distinguish predictor and response variables.

Scatter Plot Arrays

Use the MATLAB `plotmatrix` function to make comparisons of the relationships between multiple pairs of intersections:

```
figure
plotmatrix(count)
```



The plot in the (i, j)th position of the array is a scatter with the i th variable on the vertical axis and the j th variable on the horizontal axis. The plot in the i th diagonal position is a histogram of the i th variable.

See “Plotting Data” in the MATLAB Data Analysis documentation for more information on statistical visualization.

Modeling Data

In this section...

Overview

Parametric models translate an understanding of data relationships into analytic tools with predictive power. Polynomial and sinusoidal models are simple choices for the up and down trends in the traffic data.

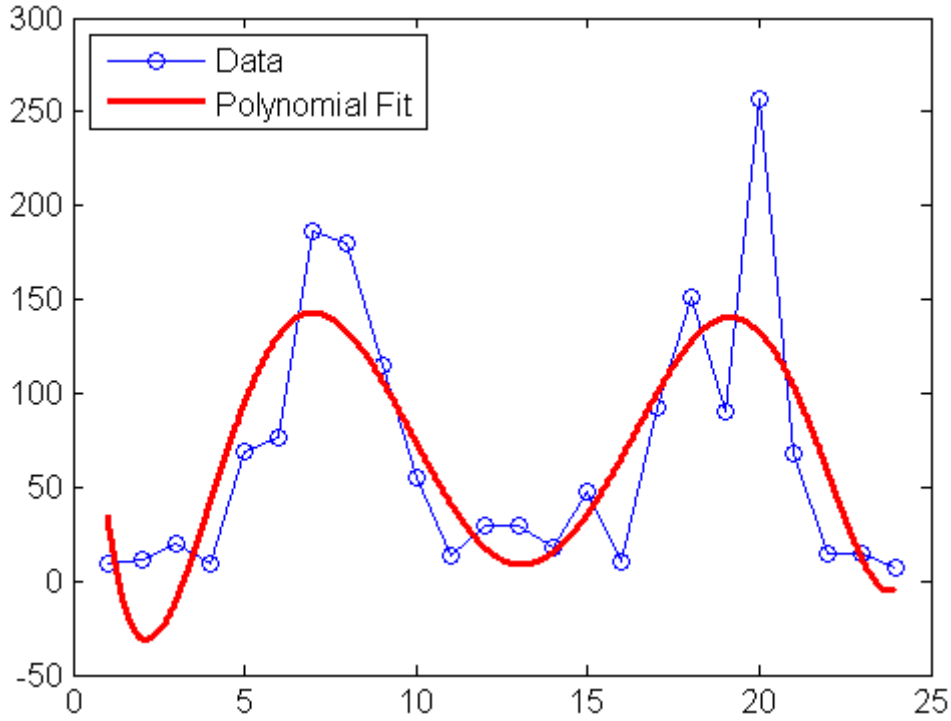
Note This section continues the data analysis from "Visualizing Data" on page 5-14.

Polynomial Regression

Use the MATLAB `polyfit` function to estimate coefficients of polynomial models, then use the MATLAB `polyval` function to evaluate the model at arbitrary values of the predictor.

The following code fits the traffic data at the third intersection with a polynomial model of degree six:

```
c3 = count(:,3); % Data at intersection 3
tdata = (1:24)';
p_coeffs = polyfit(tdata,c3,6);

figure
plot(c3,'o-')
hold on
tfit = (1:0.01:24)';
yfit = polyval(p_coeffs,tfit);
plot(tfit,yfit,'r-','LineWidth',2)
legend('Data','Polynomial Fit','Location','NW')
```



The model has the advantage of being simple while following the up-and-down trend. The accuracy of its predictive power, however, is questionable, especially at the ends of the data.

General Linear Regression

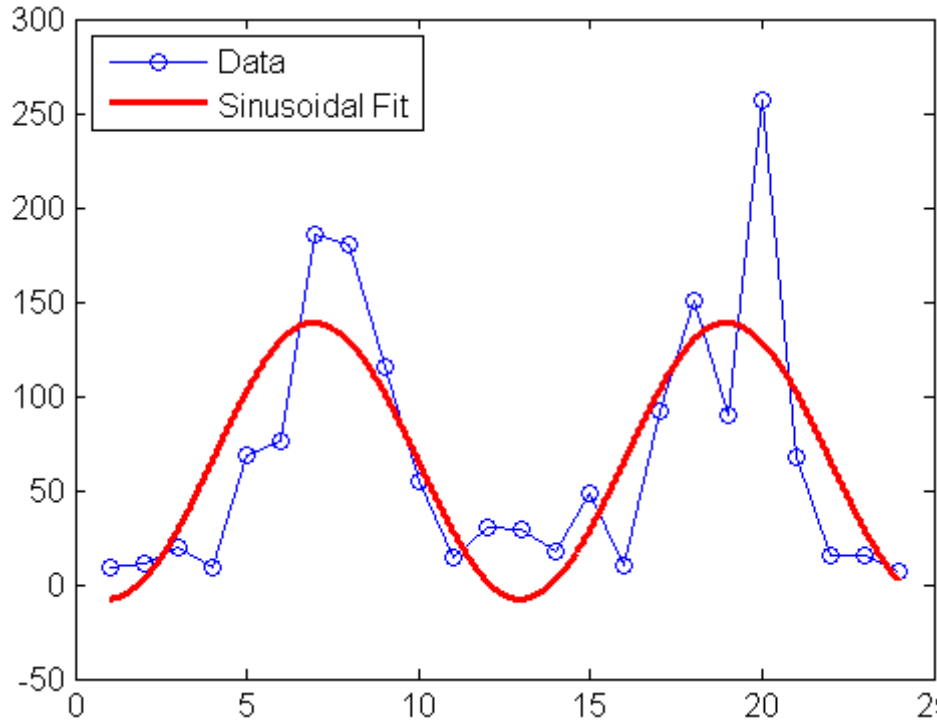
Assuming that the data are periodic with a 12-hour period and a peak around hour 7, it is reasonable to fit a sinusoidal model of the form:

$$y = a + b\cos((2\pi/12)(t-7))$$

The coefficients a and b appear linearly. Use the MATLAB `mldivide` (backslash) operator to fit general linear models:

```
c3 = count(:,3); % Data at intersection 3
tdata = (1:24)';
```

```
X = [ones(size(tdata)) cos((2*pi/12)*(tdata-7))];
s_coeffs = X\c3;

figure
plot(c3,'o-')
hold on
tfit = (1:0.01:24)';
yfit = [ones(size(tfit)) cos((2*pi/12)*(tfit-7))]*s_coeffs;
plot(tfit,yfit,'r-','LineWidth',2)
legend('Data','Sinusoidal Fit','Location','NW')
```



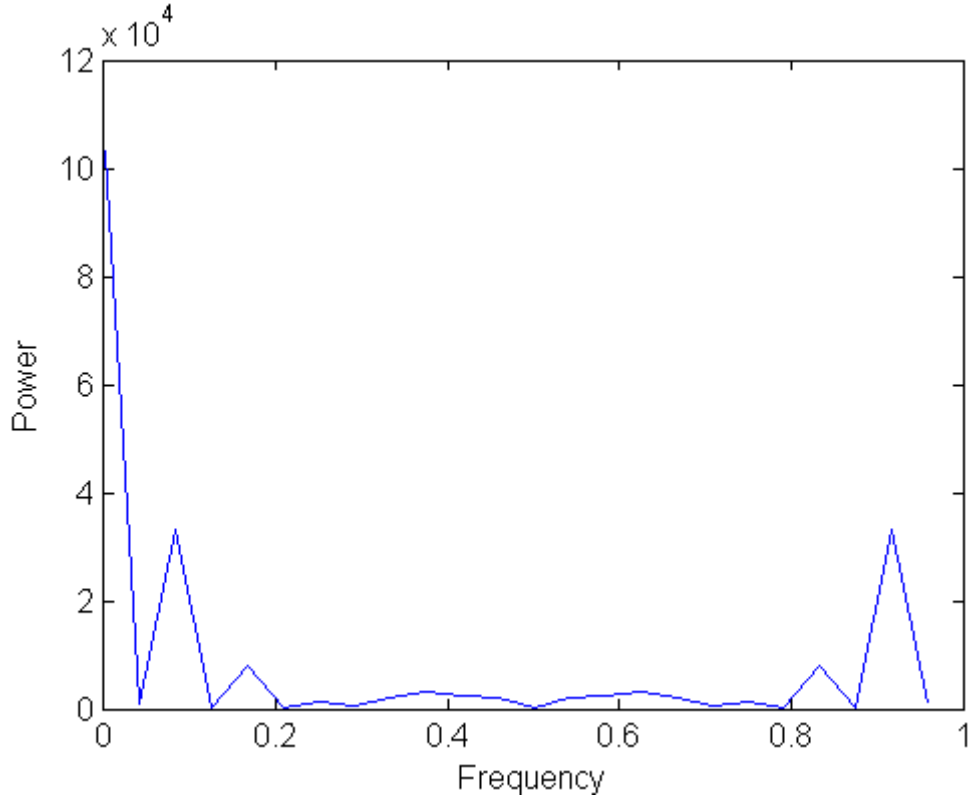
Use the MATLAB `lscov` function to compute statistics on the fit, such as estimated standard errors of the coefficients and the mean squared error:

```
[s_coeffs,stdx,mse] = lscov(X,c3)
s_coeffs =
   65.5833
```

```
   73.2819
stdx =
    8.9185
   12.6127
mse =
  1.9090e+003
```

Check the assumption of a 12-hour period in the data with a *periodogram*, computed using the MATLAB `fft` function:

```
Fs = 1; % Sample frequency (per hour)
n = length(c3); % Window length
Y = fft(c3); % DFT of data
f = (0:n-1)*(Fs/n); % Frequency range
P = Y.*conj(Y)/n; % Power of the DFT

figure
plot(f,P)
xlabel('Frequency')
ylabel('Power')

predicted_f = 1/12
predicted_f =
    0.0833
```



The peak near `0.0833` supports the assumption, although it occurs at a slightly higher frequency. The model can be adjusted accordingly.

See “Linear Regression Analysis” and “Fourier Analysis” in the MATLAB Data Analysis documentation for more information on data modeling.

6

Creating Graphical User Interfaces

What Is GUIDE?

GUIDE, the MATLAB graphical user interface development environment, provides a set of tools for creating graphical user interfaces (GUIs). These tools greatly simplify the process of designing and building GUIs. You can use the GUIDE tools to

- Lay out the GUI.

  Using the GUIDE Layout Editor, you can lay out a GUI easily by clicking and dragging GUI components—such as panels, buttons, text fields, sliders, menus, and so on—into the layout area. GUIDE stores the GUI layout in a FIG-file.

- Program the GUI.

  GUIDE automatically generates an M-file that controls how the GUI operates. The M-file initializes the GUI and contains a framework for the most commonly used callbacks for each component—the commands that execute when a user clicks a GUI component. Using the M-file editor, you can add code to the callbacks to perform the functions you want.

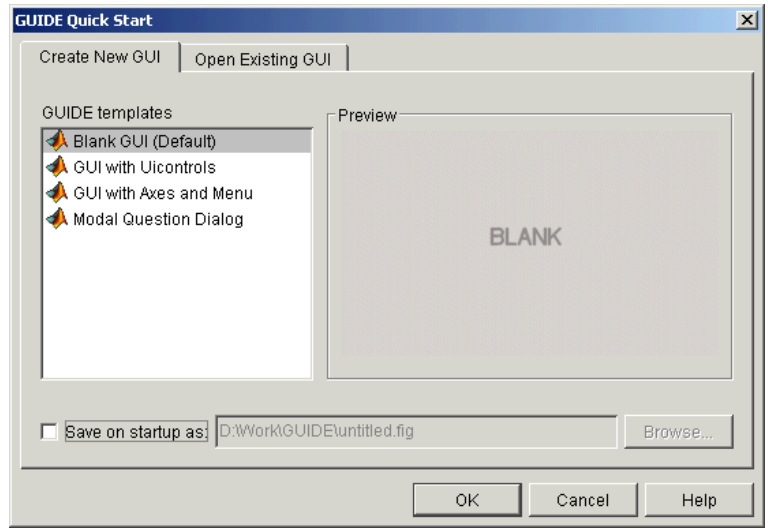
Note You can also create GUIs programmatically. For information on how to get started, see "Creating a Simple GUI Programmatically" in the MATLAB Creating Graphical User interfaces documentation.

Laying Out a GUI

In this section...

Starting GUIDE

Start GUIDE by typing `guide` at the MATLAB command prompt. This displays the GUIDE Quick Start dialog box, as shown in the following figure.

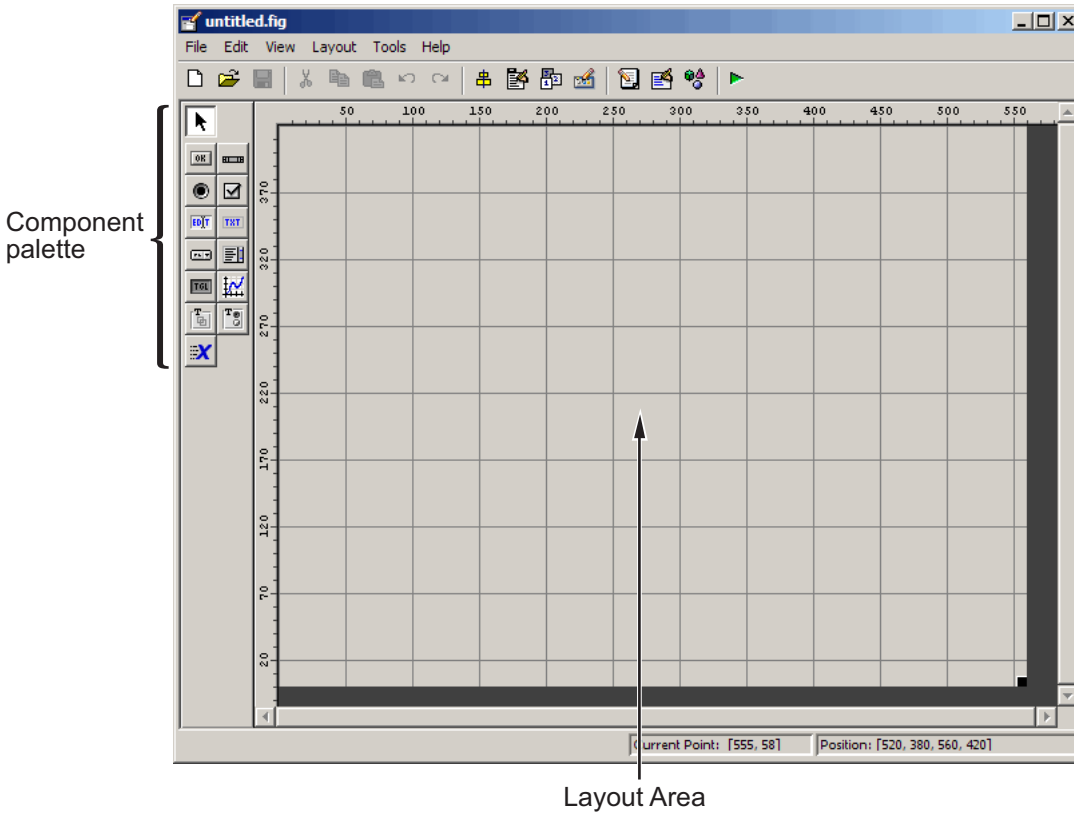


From the GUIDE Quick Start dialog box, you can

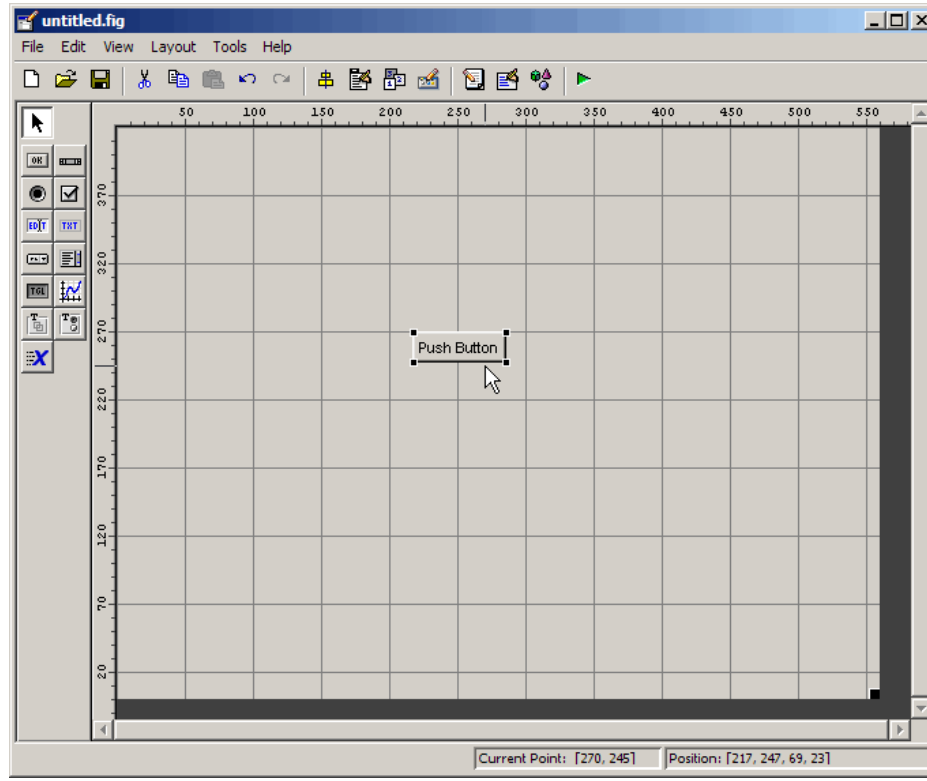
- Create a new GUI from one of the GUIDE templates—prebuilt GUIs that you can modify for your own purposes.
- Open an existing GUI.

The Layout Editor

When you open a GUI in GUIDE, it is displayed in the Layout Editor, which is the control panel for all of the GUIDE tools. The following figure shows the Layout Editor with a blank GUI template.



You can lay out your GUI by dragging components, such as panels, push buttons, pop-up menus, or axes, from the component palette, at the left side of the Layout Editor, into the layout area. For example, if you drag a push button into the layout area, it appears as in the following figure.



You can also use the Layout Editor (along with the Toolbar Editor and Icon Editor) to create menus and toolbars, create and modify tool icons, and set basic properties of the GUI components.

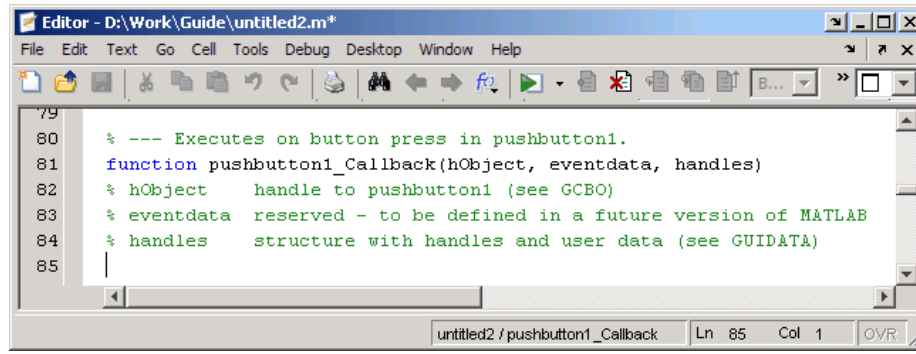
To get started using the Layout Editor and setting property values, see "Creating a Simple GUI with GUIDE" in the MATLAB Creating Graphical User Interfaces documentation. "Examples of GUIDE GUIs" in the same documentation illustrates the variety of GUIs that you can create with GUIDE.

Programming a GUI

After laying out the GUI and setting component properties, the next step is to program the GUI. You program the GUI by coding one or more callbacks for each of its components. Callbacks are functions that execute in response to some action by the user. A typical action is clicking a push button.

A GUI's callbacks are found in an M-file that GUIDE generates automatically. GUIDE adds templates for the most commonly used callbacks to this M-file, but you may want to add others. Use the M-file Editor to edit this file.

The following figure shows the Callback template for a push button.



To learn more about programming a GUI, see "Creating a Simple GUI with GUIDE" in the MATLAB Creating GUIs documentation.

7

Desktop Tools and Development Environment

If you have an active Internet connection, you can watch the Working in The Development Environment video demo for an overview of the major functionality.

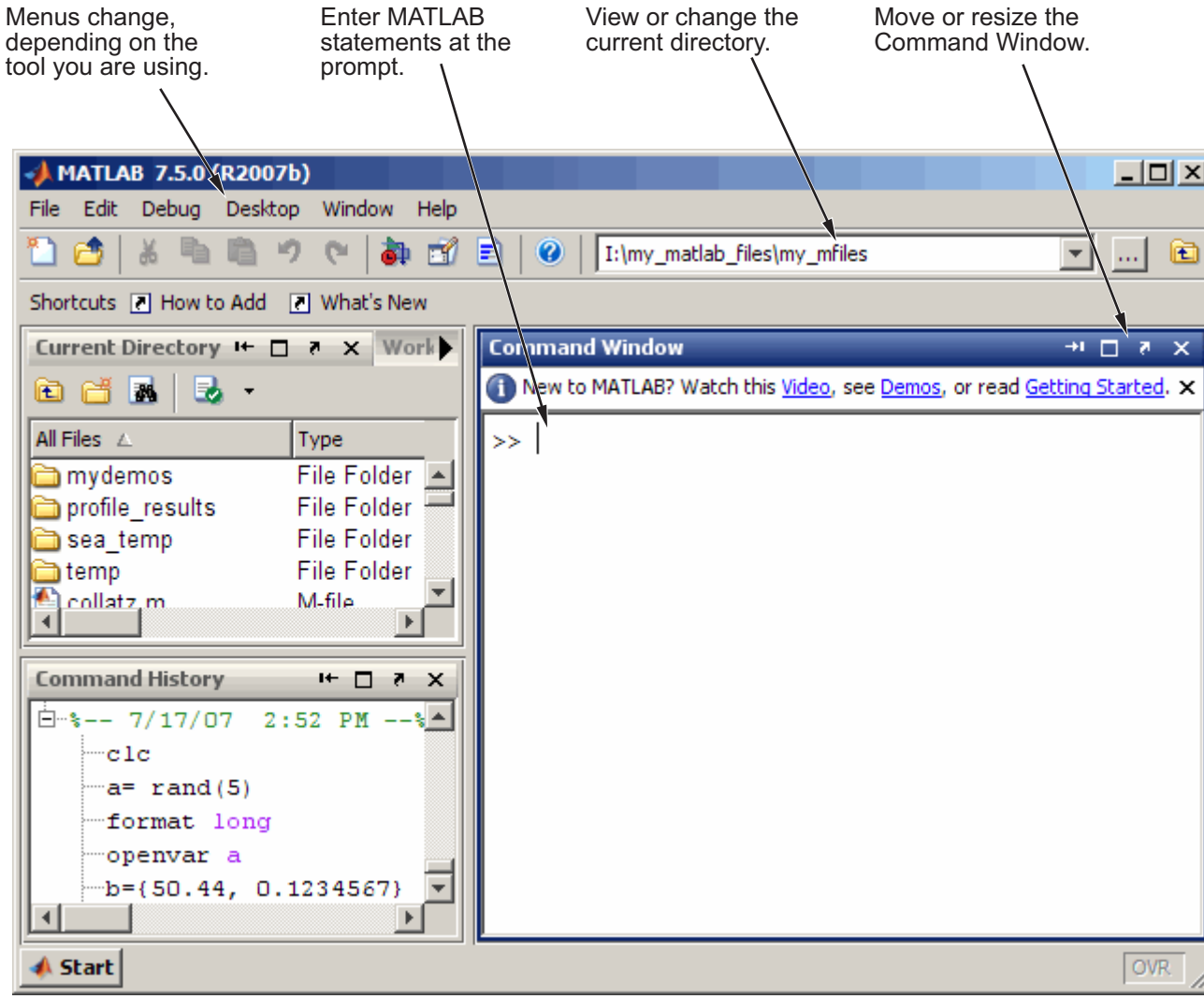
Desktop Overview

In this section...

Introduction to the Desktop

Use desktop tools to manage your work in MATLAB. You can also use MATLAB functions to perform the equivalent of most of the features found in the desktop tools.

The following illustration shows the default configuration of the MATLAB desktop. You can modify the setup to meet your needs.


Menus change, depending on the tool you are using.
Enter MATLAB statements at the prompt.
View or change the current directory.
Move or resize the Command Window.
MATLAB 7.5.0 (R2007b)
File Edit Debug Desktop Window Help
I:\my_matlab_files\my_mfiles
Shortcuts How to Add What's New
Current Directory
All Files Type
mydemos File Folder
profile_results File Folder
sea_temp File Folder
temp File Folder
collatz.m M-file
Command History
%-- 7/17/07 2:52 PM --%
clc
a= rand(5)
format long
openvar a
b={50.44, 0.1234567}
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
>>
Start
OVR

For More Information For an overview of the desktop tools, watch the video tutorials, accessible by typing `demo matlab desktop` (requires an Internet connection). For complete details, see the MATLAB Desktop Tools and Development Environment documentation.

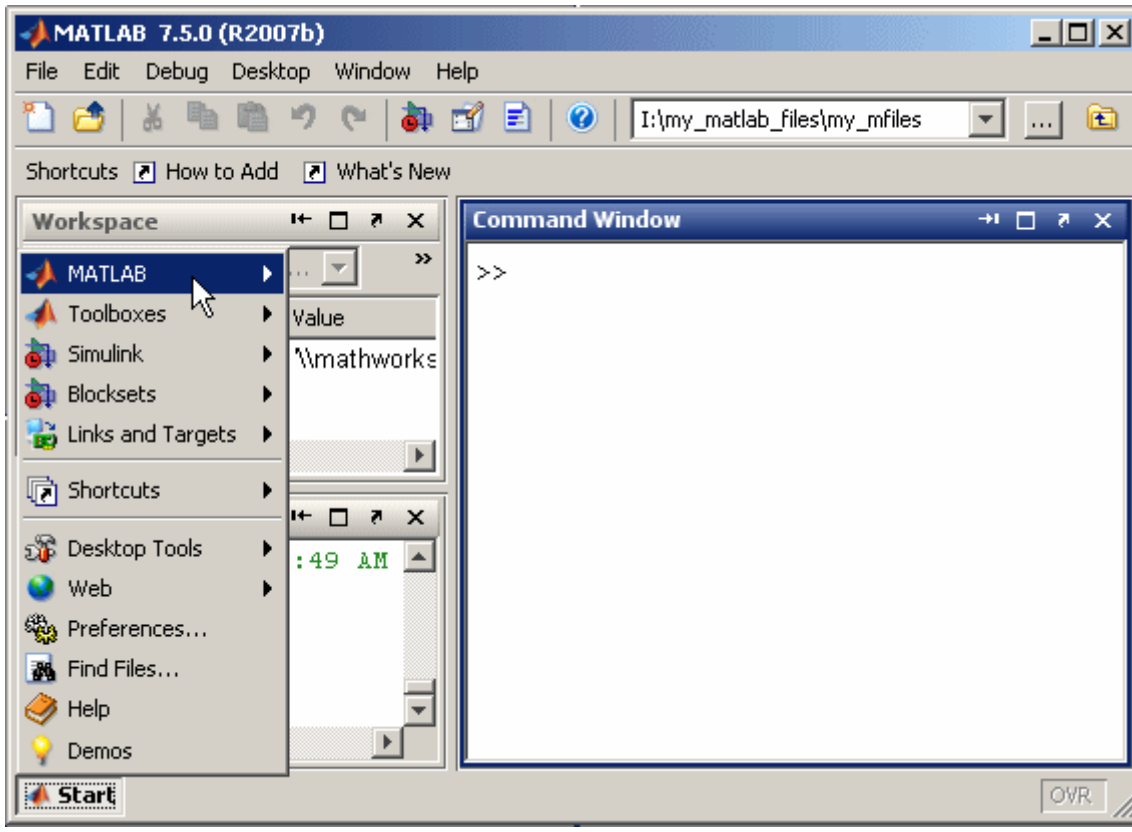
Arranging the Desktop

These are some common ways to customize the desktop:

- Show or hide desktop tools via the **Desktop** menu.
- Resize any tool by dragging one of its edges.
- Move a tool outside of the desktop by clicking the undock button in the tool's title bar.
- Reposition a tool within the desktop by dragging its title bar to the new location. Tools can occupy the same position, as shown for the Current Directory and Workspace browser in the preceding illustration, in which case, you access a tool via its name in the title bar.
- Maximize or minimize (temporarily hide) a tool within the desktop via the **Desktop** menu.
- Change fonts and other options by using **File > Preferences**.

Start Button

The MATLAB **Start** button provides easy access to tools, demos, shortcuts, and documentation. Click the **Start** button to see the options.



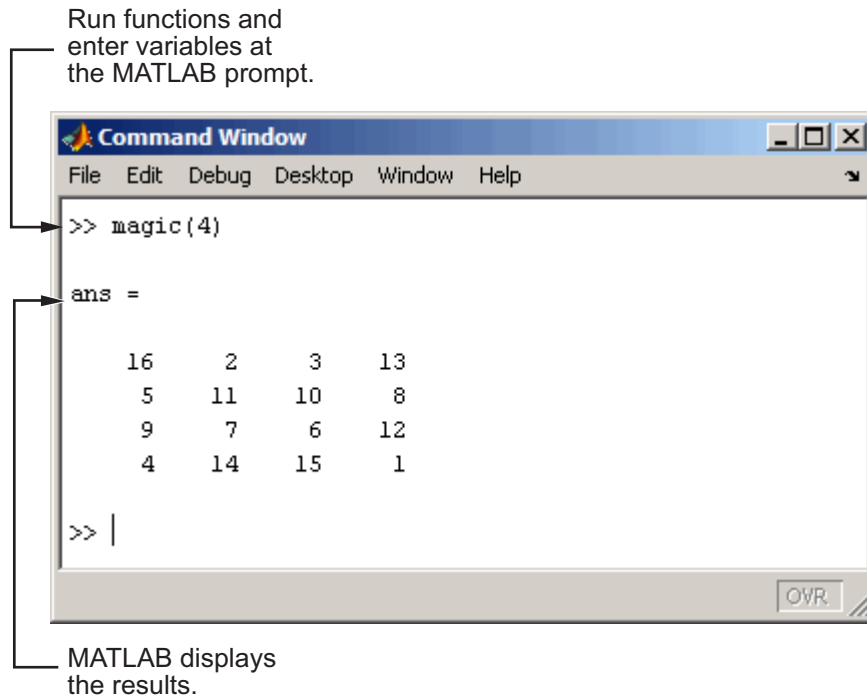
For More Information See "Desktop" in the MATLAB Desktop Tools and Development Environment documentation.

Command Window and Command History

In this section...

Command Window

Use the Command Window to enter variables and to run functions and M-file scripts.

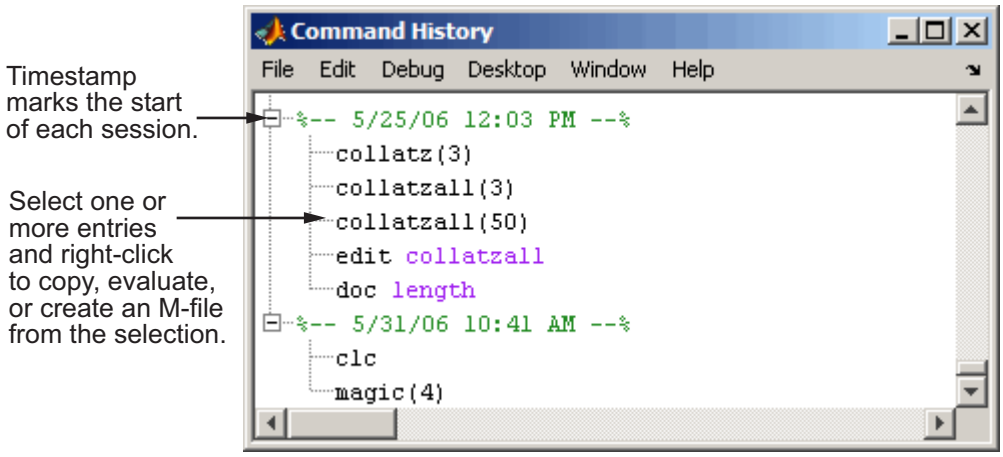


Press the up arrow key ↑ to recall a statement you previously typed. Edit the statement as needed and then press **Enter** to run it. For more information about entering statements in the Command Window, see “Controlling Command Window Input and Output” on page 2-30.

For More Information See "Running Functions — Command Window and History" in the MATLAB Desktop Tools and Development Environment documentation for complete details.

Command History

Statements you enter in the Command Window are logged in the Command History. From the Command History, you can view and search for previously run statements, as well as copy and execute selected statements. You can also create an M-file from selected statements.



To save the input and output from a MATLAB session to a file, use the `diary` function.

For More Information See "Command History Window" in the MATLAB Desktop Tools and Development Environment documentation, and the reference page for the `diary` function.

Help

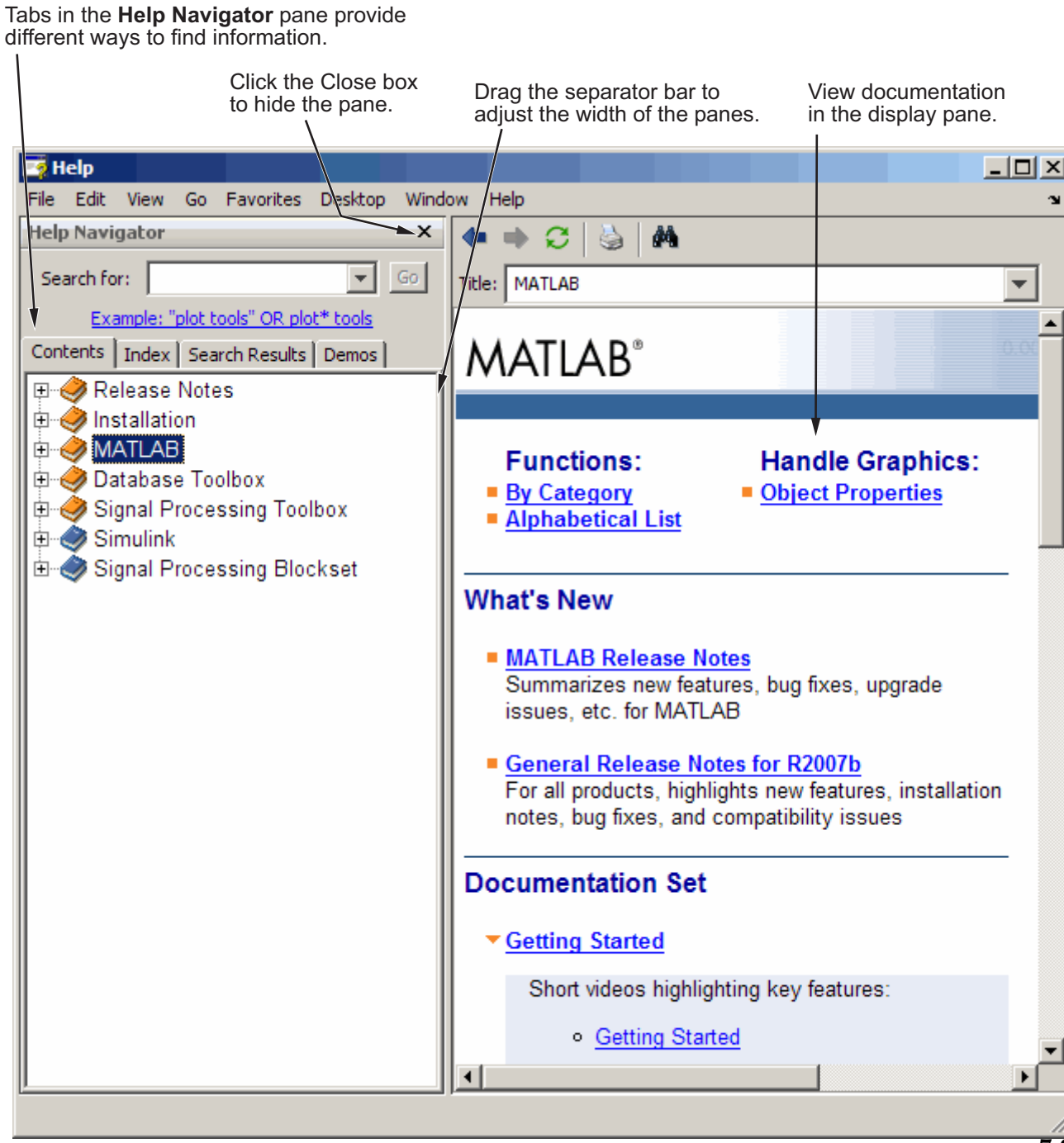
In this section...

Help Browser

Use the Help browser to search for and view documentation and demos for all your MathWorks products. The Help browser is an HTML viewer integrated into the MATLAB desktop.

To open the Help browser, click the Help button  in the desktop toolbar.

The Help browser consists of two panes, the **Help Navigator**, which you use to find information, and the display pane, where you view the information.


Tabs in the **Help Navigator** pane provide different ways to find information.
Click the Close box to hide the pane.
Drag the separator bar to adjust the width of the panes.
View documentation in the display pane.
Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search for:
Go
Example: "plot tools" OR plot* tools
Contents Index Search Results Demos
Release Notes
Installation
MATLAB
Database Toolbox
Signal Processing Toolbox
Simulink
Signal Processing Blockset
Title: MATLAB
MATLAB®
Functions:
By Category
Alphabetical List
Handle Graphics:
Object Properties
What's New
MATLAB Release Notes
Summarizes new features, bug fixes, upgrade issues, etc. for MATLAB
General Release Notes for R2007b
For all products, highlights new features, installation notes, bug fixes, and compatibility issues
Documentation Set
Getting Started
Short videos highlighting key features:
Getting Started

These are the key features:

- **Search for** field — Look for specific words in the documentation and demos. You can
  - Specify an exact phrase by enclosing words in double quotation marks, such as `"word1 word2"`
  - Use a wildcard symbol (`*`) in place of letters, such as `wo*d1`
  - Include Boolean operators between words, such as `word1 NOT word2`
- **Contents** tab — View the titles and tables of contents of the documentation. By default, the contents synchronizes to the displayed page. If you get to a page from a search or by following a link, click the **Contents** tab if you want to see the context within the overall documentation for the page you are viewing.
- **Index** tab — Find specific index entries (selected keywords) in the documentation.
- **Search Results** tab — Displays results from **Search for**, separating the results in Documentation from the results in Demos.
- **Demos** tab — View and run demonstrations for your MathWorks products. Demos include code that you can use as a basis for creating your own M-files.

While viewing the documentation, you can

- Browse to other pages — Use the arrows at the tops and bottoms of the pages to move through the document, or use the back and forward buttons in the toolbar to go to previously viewed pages.
- Bookmark pages — Use the **Favorites** menu.
- Print a page — Click the print button in the toolbar.
- Find a term in the page — Click the find icon  in the toolbar.
- Copy or evaluate a selection — Select text, such as code from an example, then right-click and use a context menu item to copy the selection or evaluate (run) it.

Other Forms of Help

In addition to the Help browser, you can use help functions. To get help for a specific function, use the `doc` function. For example, `doc format` displays documentation for the `format` function in the Help browser.

To see a briefer form of the documentation for a function, type `help` followed by the function name. The resulting help text appears in the Command Window. It shows function names in all capital letters to distinguish them from the surrounding text. When you use the function names, type them in lowercase or they will not run. Some functions actually consist of both uppercase and lowercase letters, and the help text clearly indicates that. For those functions, match the case used in the `help` function.

Other means for getting help include contacting Technical Support (`www.mathworks.com/support`) and participating in the Usenet newsgroup for MATLAB users, `comp.soft-sys.matlab`.

Typographical Conventions

These conventions are used in the Help browser and PDF documentation.

Item	Convention	Example
Buttons and keys	**Boldface**	Press the **Enter** key.
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, user input, items in drop-down lists	`Monospace` font	The `cos` function finds the cosine of each array element.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** font	`f = freqspace(n,'whole')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace` font	MATLAB responds with
		`A =`
		`5`

Menu titles and items	**Boldface**	Select **File > Save**.

Item	Convention	Example
New terms and for emphasis	*Italics*	In MATLAB, a *matrix* is a rectangular array of numbers.
Omitted input arguments	`(...)` ellipsis denotes all of the input/output arguments from preceding syntaxes	`[c, ia, ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`format('type')`

For More Information See "Help for Using MATLAB" in the MATLAB Desktop Tools and Development Environment documentation, and the reference pages for the `doc` and `help` functions.

Current Directory Browser and Search Path

In this section...

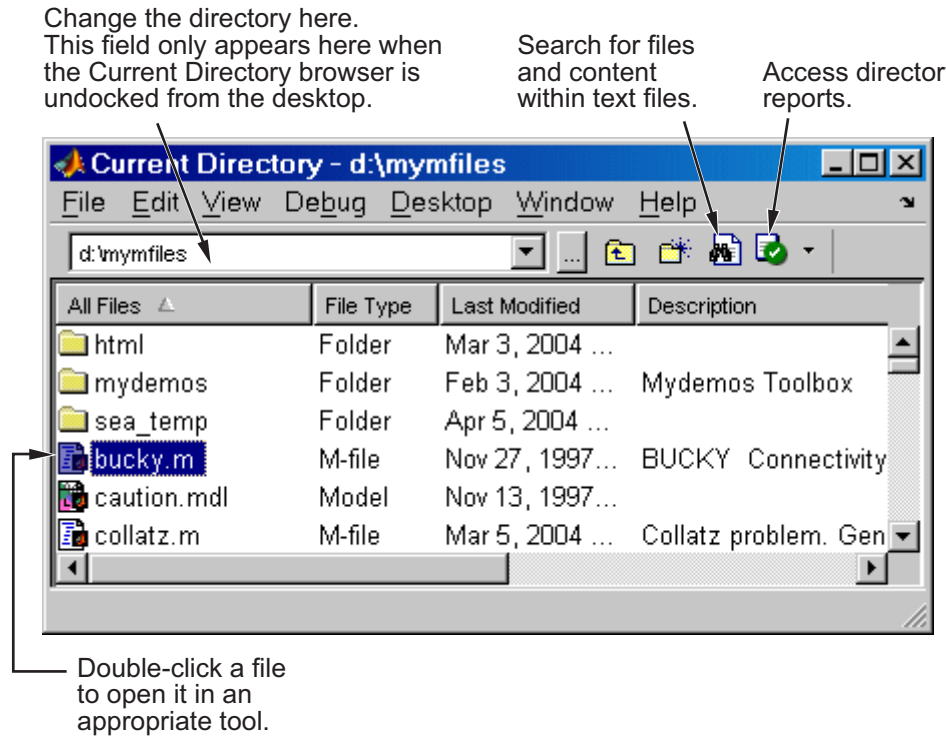
Running Files

MATLAB file operations use the current directory and the search path as reference points. Any file you want to run must either be in the current directory or on the search path.

Current Directory

A quick way to view or change the current directory is by using the current directory field in the desktop toolbar, shown here.

To search for, view, open, and make changes to MATLAB related directories and files, use the MATLAB Current Directory browser. Alternatively, you can use the functions `dir`, `cd`, and `delete`. Use “Directory Reports in Current Directory Browser” to help you tune and manage M-files.



For More Information See "File Management Operations" in the MATLAB Desktop Tools and Development Environment documentation, and the reference pages for the `dir`, `cd`, and `delete` functions.

Search Path

MATLAB uses a *search path* to find M-files and other MATLAB related files, which are organized in directories on your file system. Any file you want to run in MATLAB must reside in the current directory or in a directory that is on the search path. When you create M-files and related files for MATLAB, add the directories in which they are located to the MATLAB search path. By default, the files supplied with MATLAB and other MathWorks products are included in the search path.

To see which directories are on the search path or to change the search path, select **File > Set Path** and use the resulting Set Path dialog box. Alternatively, you can use the `path` function to view the search path, `addpath` to add directories to the path, and `rmpath` to remove directories from the path.

For More Information See "Search Path" in the MATLAB Desktop Tools and Development Environment documentation, and the reference pages for the `path`, `addpath`, and `rmpath` functions.

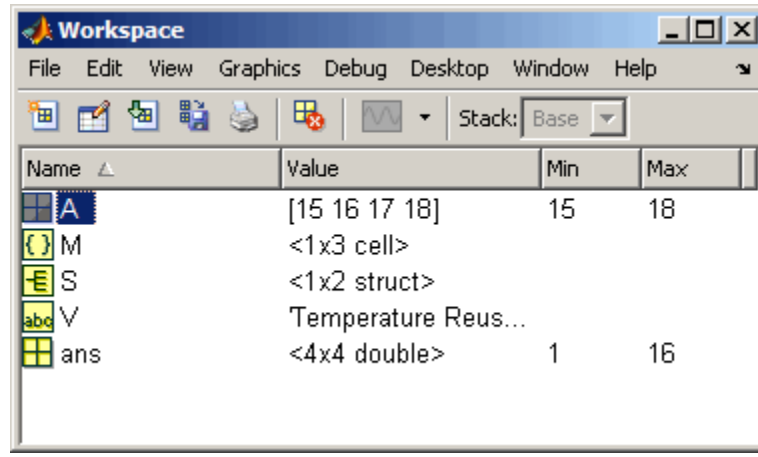
Workspace Browser and Array Editor

In this section...

Workspace Browser

The MATLAB workspace consists of the set of variables (named arrays) built up during a MATLAB session and stored in memory. You add variables to the workspace by using functions, running M-files, and loading saved workspaces.

To view the workspace and information about each variable, use the Workspace browser, or use the functions `who` and `whos`.



To delete variables from the workspace, select the variables and select **Edit > Delete**. Alternatively, use the `clear` function.

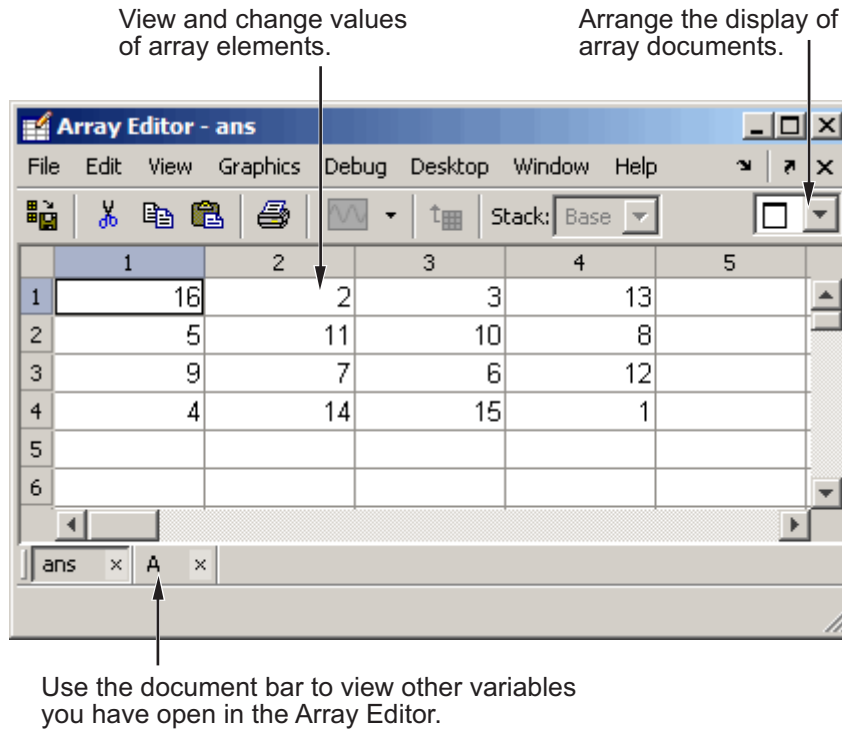
The workspace is not maintained after you end the MATLAB session. To save the workspace to a file that can be read during a later MATLAB session, select **File > Save**, or use the `save` function. This saves the workspace to a binary file called a MAT-file, which has a `.mat` extension. You can use options

to save to different formats. To read in a MAT-file, select **File > Import Data**, or use the `load` function.

For More Information See "MATLAB Workspace" in the MATLAB Desktop Tools and Development Environment documentation, and the reference pages for the `who`, `clear`, `save`, and `load` functions.

Array Editor

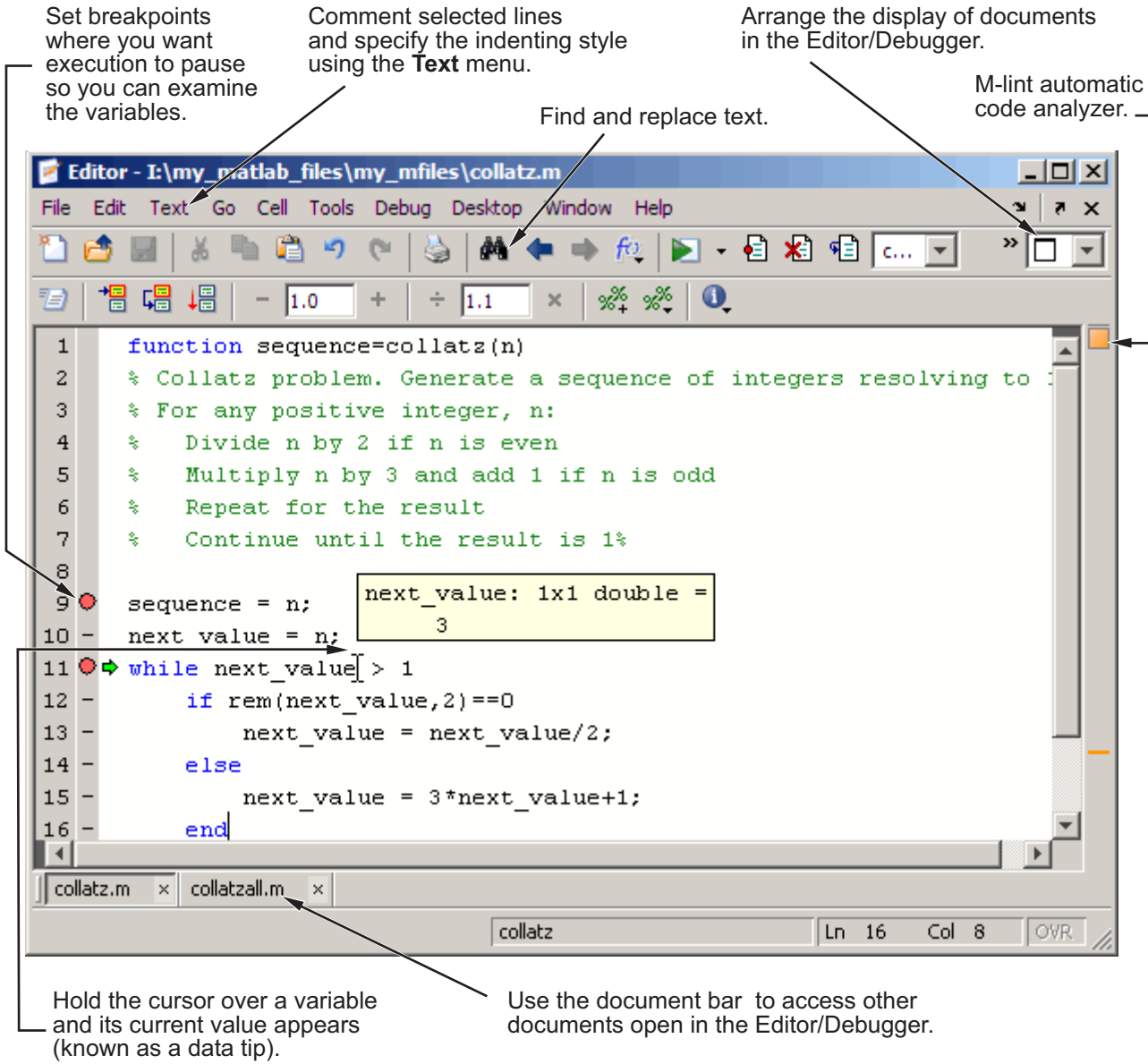
Double-click a variable in the Workspace browser, or use `openvar variablename`, to see it in the Array Editor. Use the Array Editor to view and edit a visual representation of variables in the workspace.



For More Information See “Viewing and Editing Workspace Variables with the Array Editor” in the MATLAB Desktop Tools and Development Environment documentation, and the reference page for the `openvar` function.

Editor/Debugger

Use the Editor/Debugger to create and debug M-files, which are programs you write to run MATLAB functions. The Editor/Debugger provides a graphical user interface for text editing, as well as for M-file debugging. To create or edit an M-file use **File > New** or **File > Open**, or use the `edit` function.



You can use any text editor to create M-files, such as Emacs. Use preferences (accessible from the desktop **File** menu) to specify that editor as the default.

If you use another editor, you can still use the MATLAB Editor/Debugger for debugging, or you can use debugging functions, such as `dbstop`, which sets a breakpoint.

If you just need to view the contents of an M-file, you can display the contents in the Command Window using the `type` function.

Use the M-Lint automatic code analyzer to help you identify problems and potential improvements in your code. For details, see "M-Lint Code Check and Profiler Reports" on page 7-23.

You can evaluate your code in sections, called cells, and can publish your code, including results, to popular output formats like HTML. For more information, see "Using Cells for Rapid Code Iteration and Publishing Results" in the MATLAB Desktop Tools and Development Environment documentation.

For More Information See "Editing and Debugging M-Files" in the MATLAB Desktop Tools and Development Environment documentation, and the function reference pages for `edit`, `type`, and `debug`.

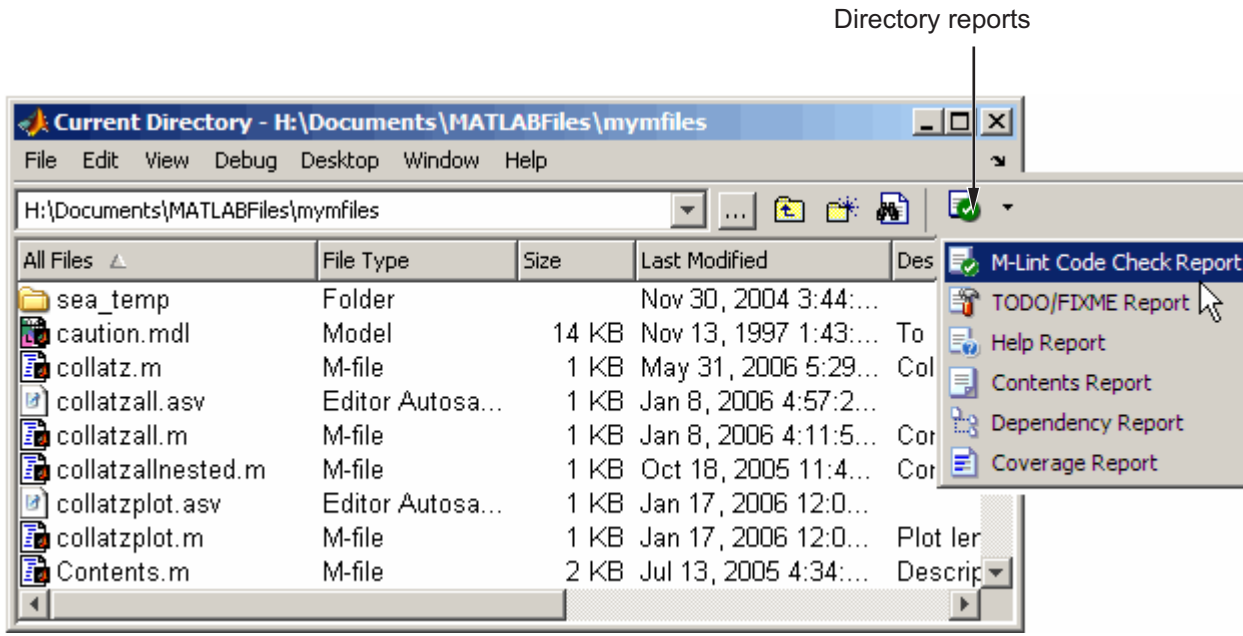
M-Lint Code Check and Profiler Reports

In this section...

M-Lint Code Check Report

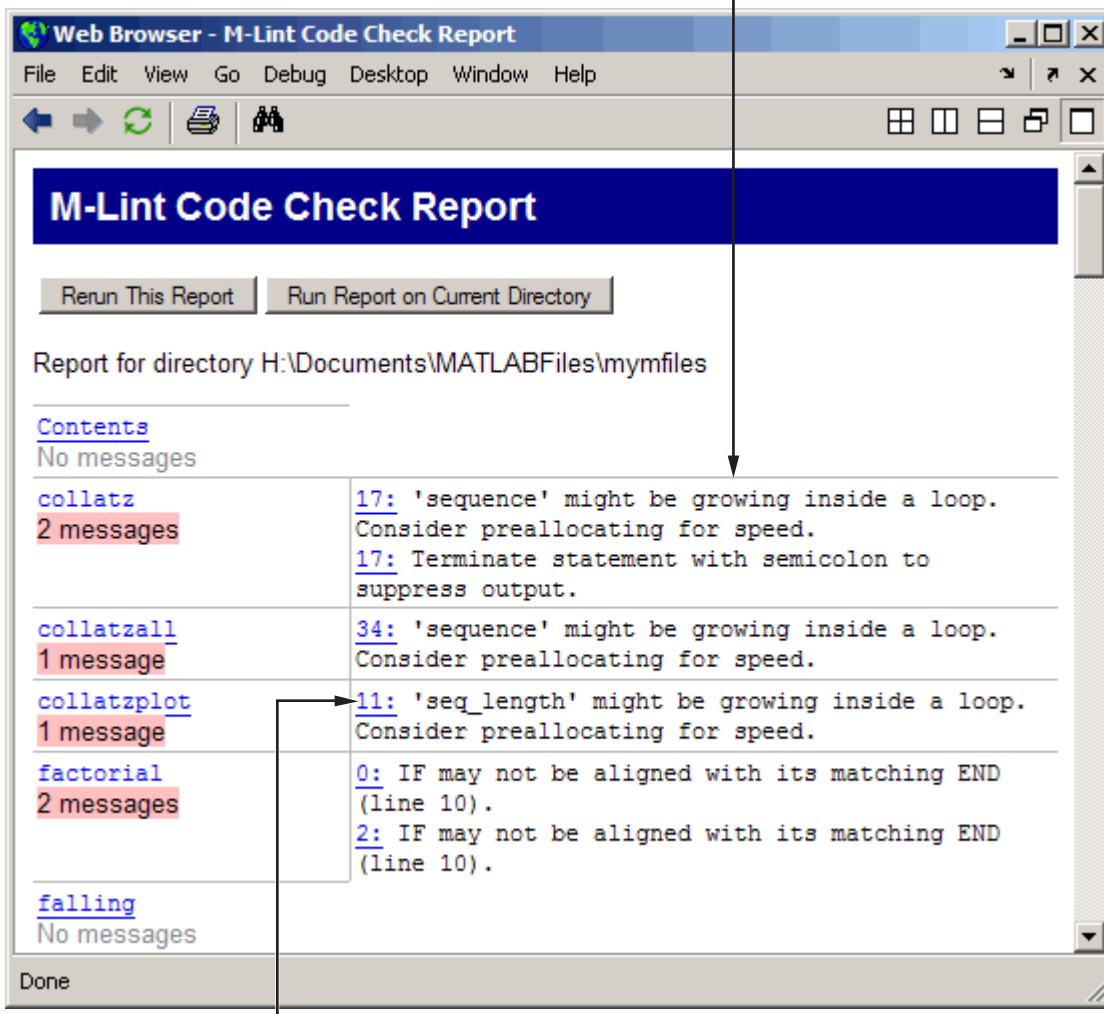
The M-Lint Code Check Report displays potential errors and problems, as well as opportunities for improvement in your M-files. The term *lint* is used by similar tools in other programming languages such as C.

Access the M-Lint Code Check Report and other directory reports from the Current Directory browser. You run a report for all files in the current directory.



In MATLAB, the M-Lint Code Check Report displays a message for each line of an M-file it determines might be improved. For example, a common M-Lint message is that a variable is defined but never used in the M-file.

The report displays a line number and message for each potential problem or improvement opportunity.



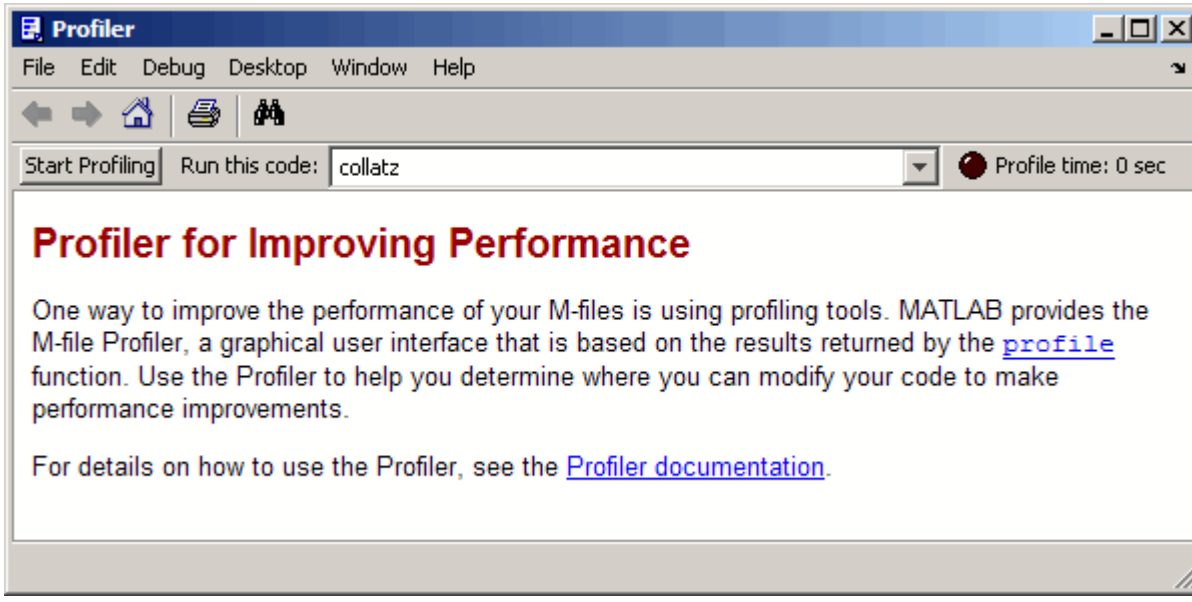
Click a line number to open the M-file in the Editor/Debugger at that line.

Alternatively, you can use automatic M-Lint code checking to view M-Lint messages while you work on a file in the Editor/Debugger. You can also use the `mlint` function to get results for a single M-file.

For More Information See "Tuning and Managing M-Files" and "M-Lint Code Analyzer" in the MATLAB Desktop Tools and Development Environment documentation, and the reference page for the `mlint` function.

Profiler

MATLAB includes the Profiler to help you improve the performance of your M-files. Run a MATLAB statement or an M-file in the Profiler and it produces a report of where the time is being spent. Access the Profiler from the **Desktop** menu, or use the `profile` function.



For More Information See “Tuning and Managing M-Files” in the MATLAB Desktop Tools and Development Environment documentation, and the reference page for the `profile` function.

Other Development Environment Features

Additional development environment features include

- Source Control — Access your source control system from within MATLAB.
- Publishing Results — Use the Editor/Debugger's cell features to publish M-files and results to popular output formats including HTML and Microsoft Word. You can also use MATLAB Notebook to access MATLAB functions from within Microsoft Word.

For More Information See "Source Control Interface" and "Publishing Results" in the MATLAB Desktop Tools and Development Environment documentation.

8

External Interfaces

Use MATLAB External Interfaces to connect MATLAB to programs, devices and data. Application developers use external interfaces to integrate MATLAB functionality with their applications. External interfaces also facilitate data collection, such as from peripheral devices like an oscilloscope or a remote network server.

Programming Interfaces

In this section...

Call MATLAB from C and Fortran Programs

Use the MATLAB engine library to call MATLAB from C and Fortran programs. When you call MATLAB from your own programs, MATLAB acts as a computation engine. For example, you can:

- Use MATLAB as a programmable mathematical subroutine library.
- Build an application with a front end (GUI) programmed in C and a back end (analysis) programmed in MATLAB.

Call C and Fortran Programs from MATLAB

Use MEX-files to call your own C or Fortran subroutines from MATLAB as if they were built-in functions. For example, you can:

- Call preexisting C and Fortran programs from MATLAB without having to rewrite them as M-files.
- Code bottleneck computations that do not run fast enough in MATLAB in C or Fortran for efficiency.

The mxArray access library creates and manipulates MATLAB arrays. The mex library performs operations in the MATLAB environment.

Call Java from MATLAB

MATLAB includes a Java Virtual Machine (JVM). This allows you to use the Java interpreter with MATLAB commands and to create and access Java objects. For example, you can:

- Access Java API class packages that support I/O and networking.
- Access third-party Java classes.
- Construct Java objects in MATLAB.
- Call Java methods, using either Java or MATLAB syntax.
- Pass data between MATLAB variables and Java objects.

Call Functions in Shared Libraries

Use the MATLAB interface to generic DLLs to interact with functions in a dynamic link library (`.dll`) on Windows, a shared object file (`.so`) on UNIX and Linux, or a dynamic shared library (`.dylib`) on Intel-based Macintosh platforms.

MATLAB supports any shared library written in C, or in any language that can provide a C interface.

Import and Export Data

MAT-files and the MAT-file access library provide a convenient mechanism for moving MATLAB binary data between platforms, and for importing and exporting data to stand-alone MATLAB applications.

Component Object Model Interface

With Component Object Model (COM) tools and technologies, you can integrate application-specific components from different vendors into your own applications. With COM, MATLAB can include ActiveX controls or OLE server processes, or you can configure MATLAB as a computational server controlled by your client application programs.

For example, you can:

- Include ActiveX components, like a calendar, in your MATLAB program.
- Access existing applications that expose objects via Automation, like Microsoft Excel.
- Access MATLAB as an Automation server from an application written in Visual Basic or C.

COM support in MATLAB is only available on the Microsoft Windows platform.

Web Services

Web services are XML-based technologies for making remote procedure calls over a network. They enable communication between applications running on disparate operating systems and development platforms. Web service technologies available in MATLAB are:

- Simple Object Access Protocol (SOAP)
- Web Services Description Language (WSDL)

Serial Port Interface

The MATLAB serial port interface provides direct access to peripheral devices that you connect to your computer's serial port, such as modems, printers, and scientific instruments. For example, you can:

- Configure serial port communications.
- Use serial port control pins.
- Write and read data.
- Use events and callbacks.
- Record information to disk.

Index

D

E

F

G

N

O

P

Q

R

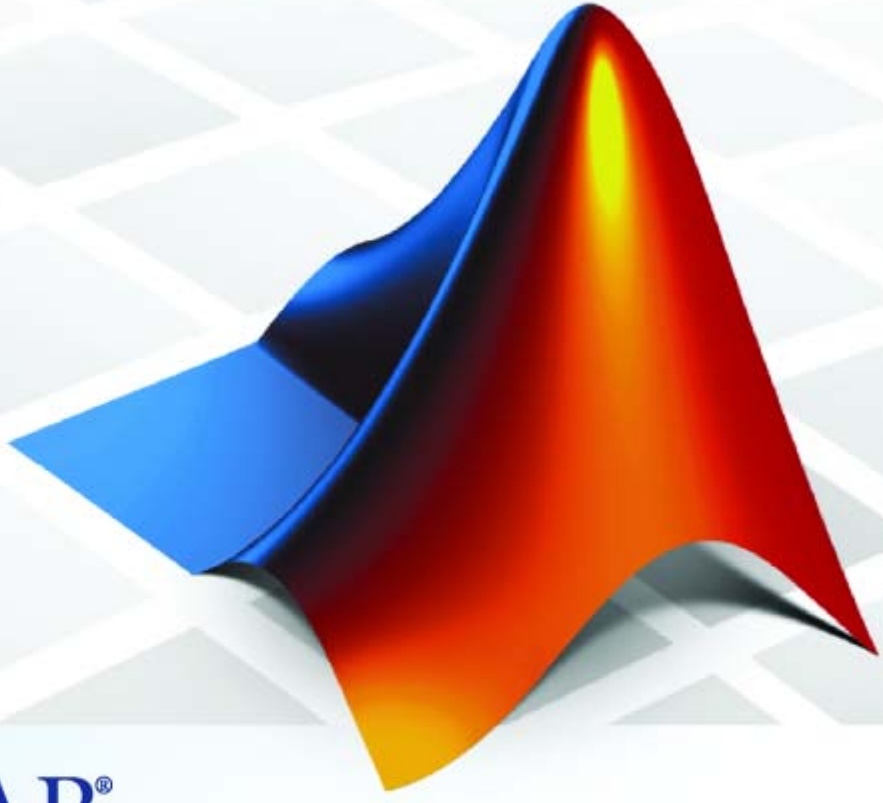
S

T

V

W

MATLAB® 7
Programming



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Programming

Revision History

June 2004	First printing	New for MATLAB 7.0 (Release 14)
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)
March 2006	Online only	Revised for MATLAB 7.2 (Release R2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release R2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release R2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release R2007b)

Contents

1 Data Structures

3 Basic Program Components

4 M-File Programming

5 Types of Functions

6 Data Import and Export

7 Working with Scientific Data Formats

8 Error Handling

9 Classes and Objects

10 Scheduling Program Execution with Timers

11 Improving Performance and Memory Usage

12 Programming Tips

Data Structures

Multidimensional Arrays (p. 1-56)	Create and work with arrays that have more than two dimensions.
Summary of Matrix and Array Functions (p. 1-76)	Quick reference to the functions commonly used in working with matrices.

Creating and Concatenating Matrices

In this section...

Overview

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

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

```
A = 100;

whos A
  Name      Size                    Bytes  Class

  A         1x1                         8  double array
```

Regardless of the data type being used, whether it is numeric, character, or logical `true` or `false` data, MATLAB stores this data in matrix (or array) form. For example, the string `'Hello World'` is a 1-by-11 matrix of individual character elements in MATLAB. You can also build matrices composed of more complex data types, such as MATLAB structures and cell arrays.

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

- "Constructing a Simple Matrix" on page 1-4
- "Specialized Matrix Functions" on page 1-5

To build a matrix composed of other matrices, see

- "Concatenating Matrices" on page 1-8
- "Matrix Concatenation Functions" on page 1-9

This section also describes

- "Generating a Numeric Sequence" on page 1-11
- "Combining Unlike Data Types" on page 1-13

Constructing a Simple Matrix

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

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

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

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

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

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

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

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

The square brackets operator constructs two-dimensional matrices only, (including 0-by-0, 1-by-1, and 1-by-n matrices). To construct arrays of more than two dimensions, see “Creating Multidimensional Arrays” on page 1-58.

For instructions on how to read or overwrite any matrix element, see “Matrix Indexing” on page 1-18.

Entering Signed Numbers

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

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

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

the next two are *not*:

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

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

Specialized Matrix Functions

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

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

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

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

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

Examples

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

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

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

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

Creating a Random Matrix. The `rand` function creates a matrix or array with elements uniformly distributed between zero and one. This example multiplies each element by 20:

```
A = rand(5) * 20
A =
   19.0026   15.2419   12.3086    8.1141    1.1578
    4.6228    9.1294   15.8387   18.7094    7.0574
   12.1369    0.3701   18.4363   18.3381   16.2633
    9.7196   16.4281   14.7641    8.2054    0.1972
   17.8260    8.8941    3.5253   17.8730    2.7778
```

The sequence of numbers produced by `rand` is determined by the internal state of the generator. Setting the generator to the same fixed state enables you to repeat computations. Examples in this documentation that use the `rand` function are initialized to a state of 0 to make the output consistent each time they are run:

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

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

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

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

Concatenating Matrices

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

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

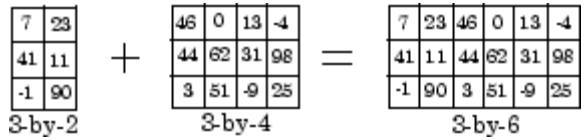
```
A = ones(2, 5) * 6;        % 2-by-5 matrix of 6's
B = rand(3, 5);            % 3-by-5 matrix of random values

C = [A; B]                 % Vertically concatenate A and B
C =
    6.0000    6.0000    6.0000    6.0000    6.0000
    6.0000    6.0000    6.0000    6.0000    6.0000
    0.9501    0.4860    0.4565    0.4447    0.9218
    0.2311    0.8913    0.0185    0.6154    0.7382
    0.6068    0.7621    0.8214    0.7919    0.1763
```

Keeping Matrices Rectangular

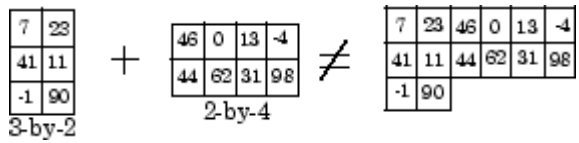
You can construct matrices, or even multidimensional arrays, using concatenation as long as the resulting matrix does not have an irregular shape (as in the second illustration shown below). If you are building a matrix horizontally, then each component matrix must have the same number of

rows. When building vertically, each component must have the same number of columns.

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



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



Matrix Concatenation Functions

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

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

Examples

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

Concatenating Matrices and Arrays. An alternative to using the `[]` operator for concatenation are the three functions `cat`, `horzcat`, and `vertcat`. With these functions, you can construct matrices (or multidimensional arrays) along a specified dimension. Either of the following commands accomplish the same task as the command `C = [A; B]` used in the section on "Concatenating Matrices" on page 1-8:

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

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

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

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

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

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

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

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

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

Generating a Numeric Sequence

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

This section covers

The Colon Operator

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

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

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

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

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

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

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

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

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

Using the Colon Operator with a Step Value

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

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

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

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

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

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

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

Combining Unlike Data Types

Matrices and arrays can be composed of elements of most any MATLAB data type as long as all elements in the matrix are of the same type. If you do include elements of unlike data types when constructing a matrix, MATLAB converts some elements so that all elements of the resulting matrix are of the same type. (See Chapter 2, "Data Types" for information on any of the MATLAB data types discussed here.)

Data type conversion is done with respect to a preset precedence of data types. The following table shows the five data types you can concatenate with an unlike type without generating an error (that is, with the exception of character and logical).

TYPE	character	integer	single	double	logical
character	character	character	character	character	invalid
integer	character	integer	integer	integer	integer
single	character	integer	single	single	single
double	character	integer	single	double	double
logical	invalid	integer	single	double	logical

For example, concatenating a `double` and `single` matrix always yields a matrix of type `single`. MATLAB converts the `double` element to `single` to accomplish this.

Combining Unlike Integer Types

If you combine different integer types in a matrix (e.g., signed with unsigned, or 8-bit integers with 16-bit integers), MATLAB returns a matrix in which all elements are of one common type. MATLAB sets all elements of the resulting matrix to the data type of the left-most element in the input matrix. For example, the result of the following concatenation is a vector of three 16-bit signed integers:

```
A = [int16(450) uint8(250) int32(1000000)]
```

MATLAB also displays a warning to inform you that the result may not be what you had expected:

```
A = [int16(450) uint8(250) int32(1000000)];
Warning: Concatenation with dominant (left-most) integer class
may overflow other operands on conversion to return class.
```

You can disable this warning by entering the following two commands directly after the operation that caused the warning. The first command retrieves the message identifier associated with the most recent warning issued by MATLAB. The second command uses this identifier to disable any further warnings of that type from being issued:

```
[msg, intcat_msgid] = lastwarn;
warning('off', intcat_msgid);
```

To reenable the warning so that it will now be displayed, use

```
warning('on', intcat_msgid);
```

You can use these commands to disable or enable the display of any MATLAB warning.

Example of Combining Unlike Integer Sizes. After disabling the integer concatenation warnings as shown above, concatenate the following two numbers once, and then switch their order. The return value depends on the order in which the integers are concatenated. The left-most type determines the data type for all elements in the vector:

```
A = [int16(5000) int8(50)]
A =
   5000   50

B = [int8(50) int16(5000)]
B =
   50   127
```

The first operation returns a vector of 16-bit integers. The second returns a vector of 8-bit integers. The element `int16(5000)` is set to `127`, the maximum value for an 8-bit signed integer.

The same rules apply to vertical concatenation:

```
C = [int8(50); int16(5000)]
```

```
C =
    50
   127
```

Note You can find the maximum or minimum values for any MATLAB integer type using the `intmax` and `intmin` functions. For floating-point types, use `realmax` and `realmin`.

Example of Combining Signed with Unsigned. Now do the same exercise with signed and unsigned integers. Again, the left-most element determines the data type for all elements in the resulting matrix:

```
A = [int8(-100) uint8(100)]
A =
   -100   100

B = [uint8(100) int8(-100)]
B =
   100   0
```

The element `int8(-100)` is set to zero because it is no longer signed.

MATLAB evaluates each element *prior to* concatenating them into a combined array. In other words, the following statement evaluates to an 8-bit signed integer (equal to 50) and an 8-bit unsigned integer (unsigned -50 is set to zero) before the two elements are combined. Following the concatenation, the second element retains its zero value but takes on the unsigned `int8` type:

```
A = [int8(50), uint8(-50)]
A =
  50    0
```

Combining Integer and Noninteger Data

If you combine integers with `double`, `single`, or `logical` data types, all elements of the resulting matrix are given the data type of the left-most integer. For example, all elements of the following vector are set to `int32`:

```
A = [true pi int32(1000000) single(17.32) uint8(250)]
```

Empty Matrices

If you construct a matrix using empty matrix elements, the empty matrices are ignored in the resulting matrix:

```
A = [5.36; 7.01; []; 9.44]
A =
    5.3600
    7.0100
    9.4400
```

Concatenation Examples

Here are some examples of data type conversion during matrix construction.

Combining Single and Double Types. Combining `single` values with `double` values yields a `single` matrix. Note that `5.73*10^300` is too big to be stored as a `single`, thus the conversion from `double` to `single` sets it to infinity. (The `class` function used in this example returns the data type for the input value):

```
x = [single(4.5) single(-2.8) pi 5.73*10^300]
x =
    4.5000   -2.8000    3.1416       Inf

class(x)              % Display the data type of x
ans =
   single
```

Combining Integer and Double Types. Combining integer values with `double` values yields an integer matrix. Note that the fractional part of `pi` is rounded to the nearest integer. (The `int8` function used in this example converts its numeric argument to an 8-bit integer):

```
x = [int8(21) int8(-22) int8(23) pi 45/6]
x =
    21   -22    23     3     7
```

```
class(x)
ans =
   int8
```

Combining Character and Double Types. Combining `character` values with `double` values yields a `character` matrix. MATLAB converts the `double` elements in this example to their `character` equivalents:

```
x = ['A' 'B' 'C' 68 69 70]
x =
   ABCDEF

class(x)
ans =
   char
```

Combining Logical and Double Types. Combining `logical` values with `double` values yields a `double` matrix. MATLAB converts the `logical` `true` and `false` elements in this example to `double`:

```
x = [true false false pi sqrt(7)]
x =
    1.0000         0         0    3.1416    2.6458

class(x)
ans =
   double
```

Matrix Indexing

In this section...

Accessing Single Elements

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

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

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

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

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

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

For arrays with more than two dimensions, specify additional indices following the `row` and `column` indices. See the section on "Multidimensional Arrays" on page 1-56.

Linear Indexing

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

So, matrix `A`

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

is actually stored in memory as the sequence

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

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

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

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

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

Functions That Control Indexing Style

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

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

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

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

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

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

Accessing Multiple Elements

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

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

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

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

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

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

Nonconsecutive Elements

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

```
B = A;

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

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

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

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

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

The end Keyword

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

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

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

Specifying All Elements of a Row or Column

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

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

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

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

Using Logicals in Array Indexing

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

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

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

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

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

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

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

Logical Indexing – Example 1

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

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

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

A simpler way to express this is

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

Logical Indexing – Example 2

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

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

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

A(~B) = 0;                          % Logical indexing

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

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

Logical Indexing with a Smaller Array

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

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

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

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

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

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

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

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

A(C)
ans =
     4
```

```
     7
     8
```

Single-Colon Indexing with Different Array Types

When you index into a standard MATLAB array using a single colon, MATLAB returns a column vector (see variable `n`, below). When you index into a structure or cell array using a single colon, you get a comma-separated list "Comma-Separated Lists" on page 3-79 (see variables `c` and `s`, below).

Create three types of arrays:

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

Use single-colon indexing on each:

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

Indexing on Assignment

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

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

- The number of subscripts specified for `B`, not including trailing subscripts equal to 1, does not exceed `ndims(B)`.
- The number of nonscalar subscripts specified for `A` equals the number of nonscalar subscripts specified for `B`. For example, `A(5, 1:4, 1, 2)`

`= B(5:8)` is valid because both sides of the equation use one nonscalar subscript.

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

Getting Information About a Matrix

In this section...

Dimensions of the Matrix

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

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

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

```
rand('state', 0);     % Initialize random number generator
A = rand(5) * 10;
A(4:5, :) = []
A =
    9.5013    7.6210    6.1543    4.0571    0.5789
    2.3114    4.5647    7.9194    9.3547    3.5287
    6.0684    0.1850    9.2181    9.1690    8.1317
```

Example Using numel

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

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

```
    5.8909
```

Example Using ndims, numel, and size

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

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

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

The code returns the following:

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

Data Types Used in the Matrix

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

Function	Description
`isa`	Detect if input is of a given data type.
`iscell`	Determine if input is a cell array.
`iscellstr`	Determine if input is a cell array of strings.
`ischar`	Determine if input is a character array.
`isfloat`	Determine if input is a floating-point array.
`isinteger`	Determine if input is an integer array.

Function	Description
islogical	Determine if input is a logical array.
isnumeric	Determine if input is a numeric array.
isreal	Determine if input is an array of real numbers.
isstruct	Determine if input is a MATLAB structure array.

Example Using isnumeric and isreal

Pick out the real numeric elements from this vector:

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

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

The values returned are

```
    1.1429
    4.2300
    3.1416
```

Data Structures Used in the Matrix

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

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

Resizing and Reshaping Matrices

In this section...

Expanding the Size of a Matrix

You can expand the size of any existing matrix as long as doing so does not give the resulting matrix an irregular shape. (See "Keeping Matrices Rectangular" on page 1-8). For example, you can vertically combine a 4-by-3 matrix and 7-by-3 matrix because all rows of the resulting matrix have the same number of columns (3).

Two ways of expanding the size of an existing matrix are

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

Note If you intend to expand the size of a matrix repeatedly over time as it requires more room (usually done in a programming loop), it is advisable to preallocate space for the matrix when you initially create it. See "Preallocating Memory" on page 1-38.

Concatenating Onto the Matrix

Concatenation is most useful when you want to expand a matrix by adding new elements or blocks that are compatible in size with the original matrix. This means that the size of all matrices being joined along a specific dimension must be equal along that dimension. See "Concatenating Matrices" on page 1-8.

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

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

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

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

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

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

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

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

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

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

Adding Smaller Blocks to a Matrix

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

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

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

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

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

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

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

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

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

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

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

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

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

The original array is 2-by-3:

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

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

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

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

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

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

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

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

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

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

Diminishing the Size of a Matrix

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

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

Then, delete the second column of `A` using

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

This changes matrix A to

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

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

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

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

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

results in

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

Reshaping a Matrix

The following functions change the shape of a matrix.

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

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

Examples

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

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

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

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

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

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

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

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

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

Rotating a Matrix. Rotate the matrix by 90 degrees:

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

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

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

Preallocating Memory

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

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

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

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

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

Building a Preallocated Array

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

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

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

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

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

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

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

   % Update row pointer
```

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

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

Shifting and Sorting Matrices

In this section...

Shift and Sort Functions

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

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

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

Shifting the Location of Matrix Elements

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

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

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

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

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

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

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

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

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

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

Sorting the Data in Each Column

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

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

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

This example first constructs a 6-by-7 random matrix:

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

Sort each column of `A` in ascending order:

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

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

Sorting the Data in Each Row

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

```
rand('state', 0);     A = floor(rand(6,7) * 100);

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

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

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

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

Sorting Row Vectors

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

The next example creates a random matrix `A`:

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

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

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

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

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

Operating on Diagonal Matrices

In this section...

Diagonal Matrix Functions

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

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

Constructing a Matrix from a Diagonal Vector

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

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

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

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

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

Returning a Triangular Portion of a Matrix

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

```
A = magic(6);

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

Concatenating Matrices Diagonally

You can diagonally concatenate matrices to form a composite matrix using the `blkdiag` function. See “Creating a Block Diagonal Matrix” on page 1-10 for more information on how this works.

Empty Matrices, Scalars, and Vectors

In this section...

Overview

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

- "The Empty Matrix" on page 1-49

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

- "Scalars" on page 1-51

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

- "Vectors" on page 1-52

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

```
Column Vector          Row Vector

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

The Empty Matrix

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

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

```
A = [];

whos A
  Name      Size         Bytes  Class

  A         0x0              0  double array
```

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

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

Operating on an Empty Matrix

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

For example, horizontal concatenation

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

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

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

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

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

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

These operations on an empty scalar array return a nonzero value:

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

Using Empty Matrices in Relational Operations

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

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

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

Using Empty Matrices in Logical Operations

MATLAB has two distinct types of logical operators:

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

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

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

```
true || []
ans =
   1

false && []
ans =
   0
```

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

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

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

•

Scalars

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

```
A = 5;

ndims(A)        % Check number of dimensions in A
ans =
     2

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

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

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

Vectors

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

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

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

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

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

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

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

```
A = [5.36; 7.01; []; 9.44]
A =
    5.3600
    7.0100
    9.4400
```

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

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

Full and Sparse Matrices

In this section...

Overview

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

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

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

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

Sparse Matrix Functions

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

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

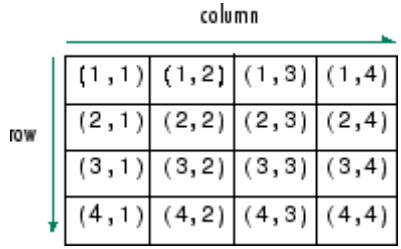
Function	Description
`nnz`	Return the number of nonzero matrix elements.
`nonzeros`	Return the nonzero elements of a matrix.
`nzmax`	Return the amount of storage allocated for nonzero elements.
`spalloc`	Allocate space for a sparse matrix.
`sparse`	Create a sparse matrix or convert full to sparse.
`speye`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

Multidimensional Arrays

In this section...

Overview

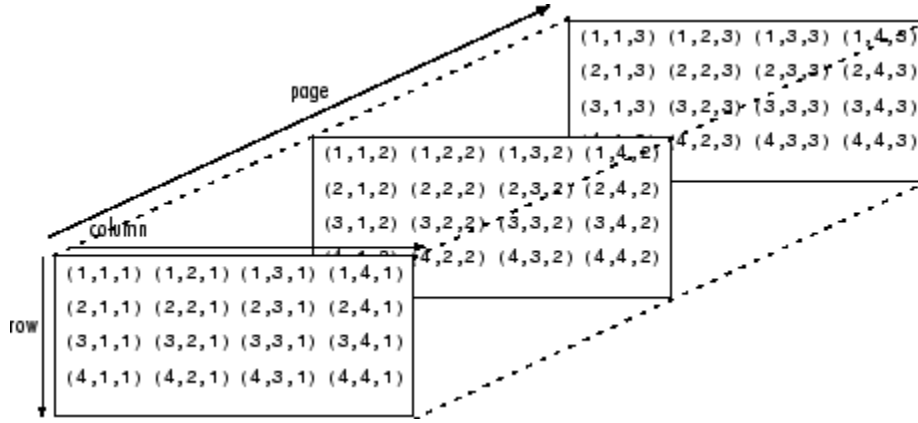
An array having more than two dimensions is called a multidimensional array in MATLAB. Multidimensional arrays in MATLAB are an extension of the normal two-dimensional matrix. Matrices have two dimensions: the row dimension and the column dimension.



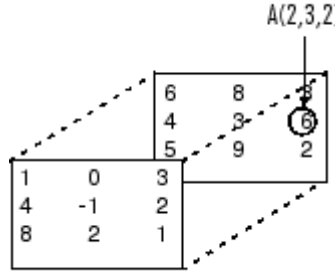
You can access a two-dimensional matrix element with two subscripts: the first representing the row index, and the second representing the column index.

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

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



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




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

     1     0     3
     4    -1     2
     8     2     1

A(:,:,2) =

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

As you add dimensions to an array, you also add subscripts. A four-dimensional array, for example, has four subscripts. The first two reference a row-column pair; the second two access the third and fourth dimensions of data.

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

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

Creating Multidimensional Arrays

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

This section discusses

Generating Arrays Using Indexing

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

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

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

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

MATLAB responds with

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

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

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

Extending Multidimensional Arrays

To extend `A` in any dimension:

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

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

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

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

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

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

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

Generating Arrays Using MATLAB Functions

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

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

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

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

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

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

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

Building Multidimensional Arrays with the cat Function

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

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

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

For example, to create a new array with cat:

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

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

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

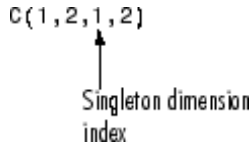
The cat function accepts any combination of existing and new data. In addition, you can nest calls to cat. The lines below, for example, create a four-dimensional array.

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

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

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

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



Accessing Multidimensional Array Properties

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

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

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

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

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

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

```
whos
Name     Size       Bytes   Class

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

Grand total is 48 elements using 384 bytes
```

Indexing Multidimensional Arrays

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

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

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

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

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

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

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

The Colon and Multidimensional Array Indexing

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

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

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

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

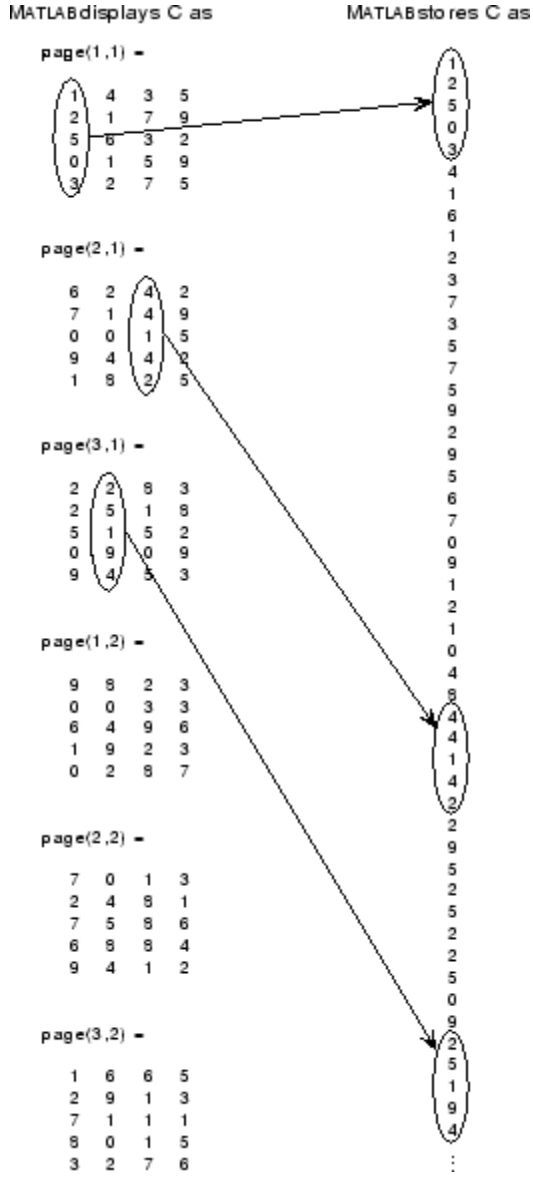
Now assign a 2-by-2 subset of array `nddata` to the four elements in the center of `C`.

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

Linear Indexing with Multidimensional Arrays

MATLAB linear indexing also extends to multidimensional arrays. In this case, MATLAB operates on a page-by-page basis to create the storage column, again appending elements columnwise. See “Linear Indexing” on page 1-19 for an introduction to this topic.

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



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

```
ans =
     0
```

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

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

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

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

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

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

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

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

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

is equivalent to

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

Avoiding Ambiguity in Multidimensional Indexing

Some assignment statements, such as

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

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

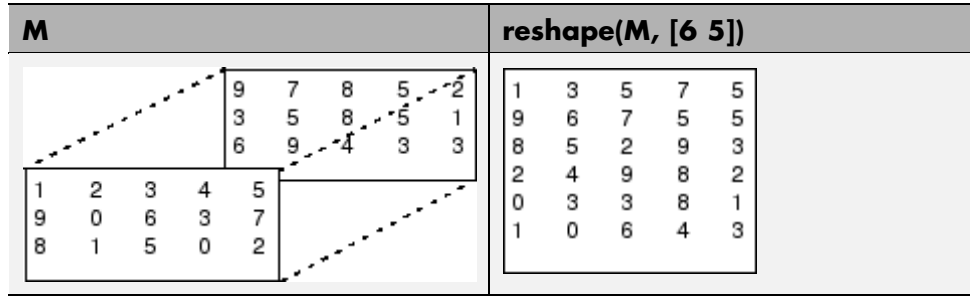
```
A(1,:,2) = 1:10;
```

Reshaping Multidimensional Arrays

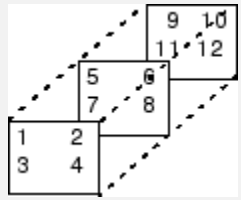
Unless you change its shape or size, a MATLAB array retains the dimensions specified at its creation. You change array size by adding or deleting elements. You change array shape by respecifying the array's row, column, or page dimensions while retaining the same elements. The `reshape` function performs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

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

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



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

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

Here are several new arrays from reshaping `nddata`:

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

Removing Singleton Dimensions

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

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

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

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

```
C = squeeze(B);

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

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

Permuting Array Dimensions

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

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

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

A

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

     1     2     3
     4     5     6
     7     8     9

A(:,:,2) =

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

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

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

     1     4     7
     2     5     8
     3     6     9

B(:,:,2) =

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

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

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

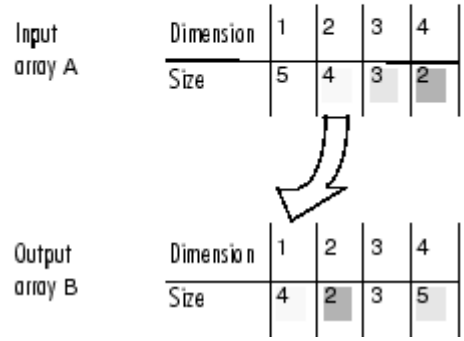
```
C(:,:,1) =

     1     2     3
     0     5     4

C(:,:,2) =

     4     5     6
     2     7     6

C(:,:,3) =

     7     8     9
     9     3     1
```

Row and page subscripts are reversed.

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

```
B = permute(A,[2 4 3 1])
```

Move dimension 2 of A to first subscript position of B, dimension 4 to second subscript position, and so on.



The order of dimensions in `permute`'s argument list determines the size and shape of the output array. In this example, the second dimension moves to the first position. Because the second dimension of the original array had size 4, the output array's first dimension also has size 4.

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

Inverse Permutation

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

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

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

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

Computing with Multidimensional Arrays

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

Operating on Vectors

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

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

Operating Element-by-Element

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

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

Operating on Planes and Matrices

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

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

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

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

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

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

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

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

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

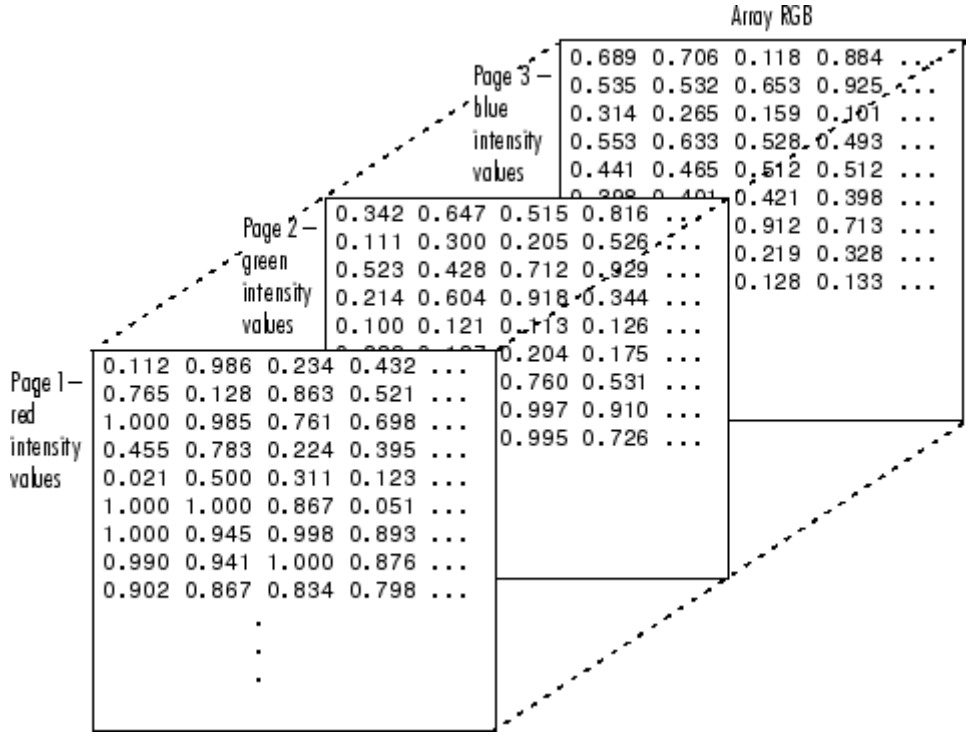
Organizing Data in Multidimensional Arrays

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

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

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

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



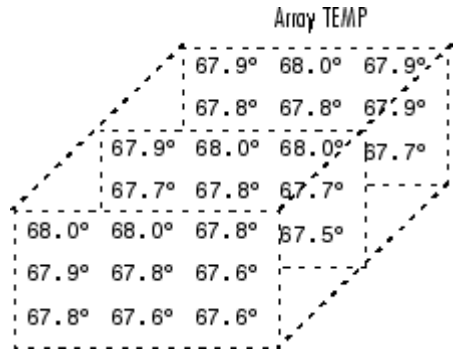
To access an entire plane of the image, use

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

To access a subimage, use

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

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



Now to find the average of all the measurements, use

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

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

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

Multidimensional Cell Arrays

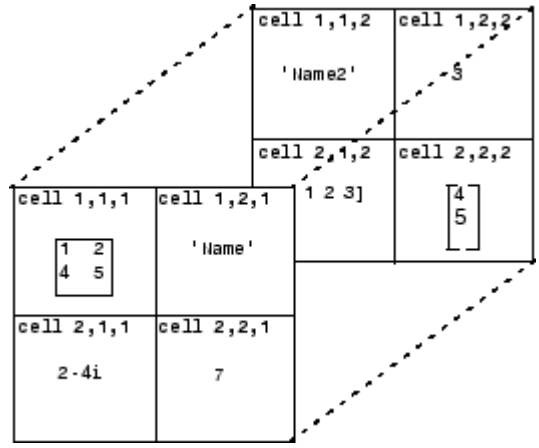
Like numeric arrays, the framework for multidimensional cell arrays in MATLAB is an extension of the two-dimensional cell array model. You can use the `cat` function to build multidimensional cell arrays, just as you use it for numeric arrays.

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

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

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

The subscripts for the cells of `C` look like

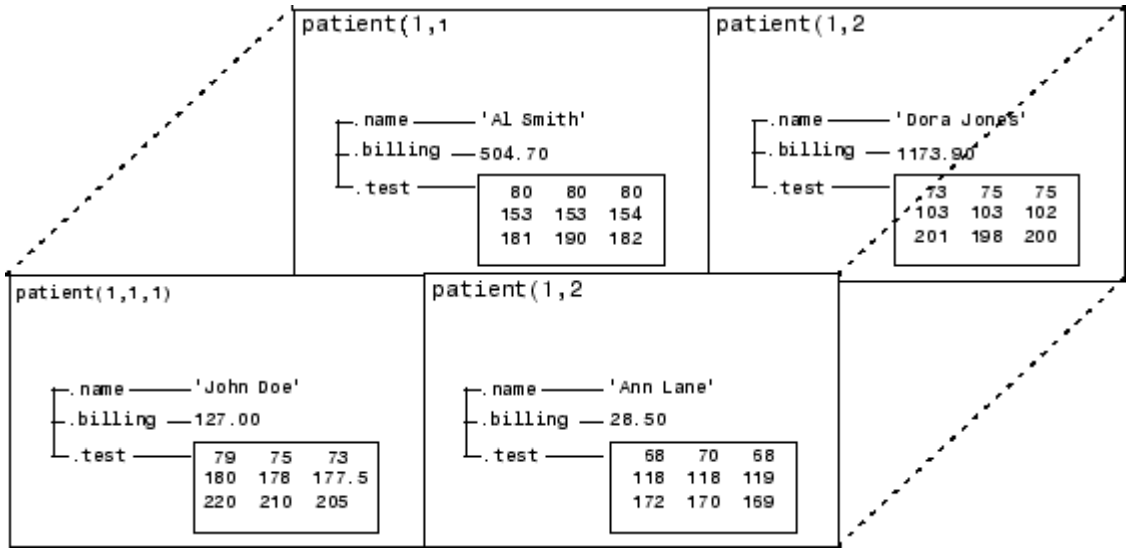


Multidimensional Structure Arrays

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

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

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



Applying Functions to Multidimensional Structure Arrays

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

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

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

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

Summary of Matrix and Array Functions

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

Functions to Create a Matrix

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

Functions to Modify the Shape of a Matrix

Function	Description
ctranspose	Flip a matrix about the main diagonal and replace each element with its complex conjugate.
`flipdim`	Flip a matrix along the specified dimension.
`fliplr`	Flip a matrix about a vertical axis.

Functions to Modify the Shape of a Matrix (Continued)

Function	Description
`flipud`	Flip a matrix about a horizontal axis.
`reshape`	Change the dimensions of a matrix.
`rot90`	Rotate a matrix by 90 degrees.
transpose	Flip a matrix about the main diagonal.

Functions to Find the Structure or Shape of a Matrix

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

Functions to Determine Data Type

Function	Description
`iscell`	Return `true` if the matrix is a cell array.
`ischar`	Return `true` if matrix elements are characters or strings.
`isfloat`	Determine if input is a floating point array.
`isinteger`	Determine if input is an integer array.
`islogical`	Return `true` if matrix elements are logicals.
`isnumeric`	Return `true` if matrix elements are numeric.

Functions to Determine Data Type (Continued)

Function	Description
`isreal`	Return `true` if matrix elements are real numbers.
`isstruct`	Return `true` if matrix elements are MATLAB structures.

Functions to Sort and Shift Matrix Elements

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

Functions That Work on Diagonals of a Matrix

Function	Description
`blkdiag`	Construct a block diagonal matrix.
`diag`	Return the diagonals of a matrix.
`trace`	Compute the sum of the elements on the main diagonal.

Functions That Work on Diagonals of a Matrix (Continued)

Function	Description
`tril`	Return the lower triangular part of a matrix.
`triu`	Return the upper triangular part of a matrix.

Functions to Change the Indexing Style

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

Functions for Working with Multidimensional Arrays

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

2

Data Types

Overview of MATLAB Data Types

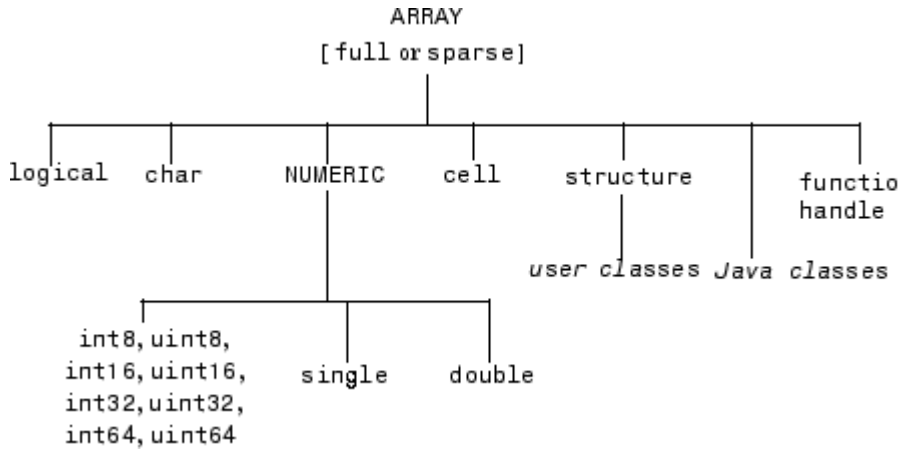
In this section...

Fundamental Data Types

There are many different types of data that you can work with in MATLAB. You can build matrices and arrays of floating-point and integer data, characters and strings, logical `true` and `false` states, etc. Two of the MATLAB data types, structures and cell arrays, provide a way to store dissimilar types of data in the same array. You can also develop your own data types using MATLAB classes.

There are 15 fundamental data types in MATLAB. Each of these data types is in the form of a matrix or array. This matrix or array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size.

All of the fundamental data types are shown in lowercase, plain nonitalic text in the diagram below.



The two data types shown in italic text are user-defined, object-oriented *`user classes`* and *`Java classes`*. You can use the latter with the MATLAB interface to Java (see "Calling Java from MATLAB" in the MATLAB External Interfaces documentation).

You can create two-dimensional `double` and `logical` matrices using one of two storage formats: `full` or `sparse`. For matrices with mostly zero-valued elements, a sparse matrix requires a fraction of the storage space required for an equivalent full matrix. Sparse matrices invoke methods especially tailored to solve sparse problems

These data types require different amounts of storage, the smallest being a `logical` value or 8–bit integer which requires only 1 byte. It is important to keep this minimum size in mind if you work on data in files that were written using a precision smaller than 8 bits.

How to Use the Different Types

The following table describes these data types in more detail.

Data Type	Example	Description
`int8, uint8, int16, uint16, int32, uint32, int64, uint64`	`uint16(65000)`	Array of signed (`int`) and unsigned (`uint`) integers. Some integer types require less storage space than `single` or `double`. All integer types except for `int64` and `uint64` can be used in mathematical operations.
`single`	`single(3 * 10^38)`	Array of single-precision numbers. Requires less storage space than `double`, but has less precision and a smaller range.
`double`	`3 * 10^300` `5 + 6i`	Array of double-precision numbers. Two-dimensional arrays can be sparse. The default numeric type in MATLAB.
`logical`	`magic(4) > 10`	Array of logical values of `1` or `0` to represent `true` and `false` respectively. Two-dimensional arrays can be sparse.

Data Type	Example	Description
`char`	`'Hello'`	Array of characters. Strings are represented as vectors of characters. For arrays containing more than one string, it is best to use cell arrays.
`cell array`	`a{1,1} = 12; a{1,2} = 'Red'; a{1,3} = magic(4);`	Array of indexed cells, each capable of storing an array of a different dimension and data type.
`structure`	`a.day = 12; a.color = 'Red'; a.mat = magic(3);`	Array of C-like structures, each structure having named fields capable of storing an array of a different dimension and data type.
`function handle`	`@sin`	Pointer to a function. You can pass function handles to other functions.
`user class`	`polynom([0 -2 -5])`	Objects constructed from a user-defined class. See “MATLAB Classes” on page 2-117
`Java class`	`java.awt.Frame`	Objects constructed from a Java class. See “Java Classes” on page 2-118.

Numeric Types

In this section...

Overview

Numeric data types in MATLAB include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting and reshaping. All numeric types except for `int64` and `uint64` can be used in mathematical operations.

Integers

MATLAB has four signed and four unsigned integer data types. Signed types enable you to work with negative integers as well as positive, but cannot represent as wide a range of numbers as the unsigned types because one bit is used to designate a positive or negative sign for the number. Unsigned types give you a wider range of numbers, but these numbers can only be zero or positive.

This section covers:

MATLAB supports 1-, 2-, 4-, and 8-byte storage for integer data. You can save memory and execution time for your programs if you use the smallest integer type that accommodates your data. For example, you don't need a 32-bit integer to store the value `100`.

Here are the eight integer data types, the range of values you can store with each type, and the MATLAB conversion function required to create that type:

Data Type	Range of Values	Conversion Function
Signed 8-bit integer	-2^7 to 2^7-1	`int8`
Signed 16-bit integer	-2^{15} to $2^{15}-1$	`int16`
Signed 32-bit integer	-2^{31} to $2^{31}-1$	`int32`
Signed 64-bit integer	-2^{63} to $2^{63}-1$	`int64`
Unsigned 8-bit integer	0 to 2^8-1	`uint8`
Unsigned 16-bit integer	0 to $2^{16}-1$	`uint16`
Unsigned 32-bit integer	0 to $2^{32}-1$	`uint32`
Unsigned 64-bit integer	0 to $2^{64}-1$	`uint64`

Creating Integer Data

MATLAB stores numeric data as double-precision floating point (`double`) by default. To store data as an integer, you need to convert from `double` to the desired integer type. Use one of the conversion functions shown in the table above.

For example, to store `325` as a 16-bit signed integer assigned to variable `x`, type

```
x = int16(325);
```

If the number being converted to an integer has a fractional part, MATLAB rounds to the nearest integer. If the fractional part is exactly `0.5`, then from the two equally nearby integers, MATLAB chooses the one for which the absolute value is larger in magnitude:

```
x = 325.499;                    x = x + .001;

int16(x)                        int16(x)
ans =                           ans =
     325                             326
```

If you need to round a number using a rounding scheme other than the default, MATLAB provides four rounding functions: `round`, `fix`, `floor`, and `ceil`. In this example, the `fix` function enables you to override the default and round *towards zero* when the fractional part of a number is `.5`:

```
x = 325.5;

int16(fix(x))
ans =
     325
```

Arithmetic operations that involve both integers and floating-point always result in an integer data type. MATLAB rounds the result, when necessary, according to the default rounding algorithm. The example below yields an exact answer of `1426.75` which MATLAB then rounds to the next highest integer:

```
int16(325) * 4.39
ans =
     1427
```

The integer conversion functions are also useful when converting other data types, such as strings, to integers:

```
str = 'Hello World';

int8(str)
ans =
   72  101  108  108  111   32   87  111  114  108  100
```

Arithmetic Operations on Integer Data Types

MATLAB can perform integer arithmetic on the following types of data:

- Integers or integer arrays of the same integer data type. This yields a result that has the same data type as the operands:

```
x = uint32([132 347 528]) .* uint32(75);
```

- Integers or integer arrays and scalar double-precision floating-point numbers. This yields a result that has the same data type as the integer operands:

```
x = uint32([132 347 528]) .* 75.49;
```

For all binary operations in which one operand is an array of integer data type and the other is a scalar `double`, MATLAB computes the operation using elementwise double-precision arithmetic, and then converts the result back to the original integer data type.

For a list of the operations that support integer data types, see Nondouble Data Type Support in the arithmetic operators reference page.

Largest and Smallest Values for Integer Data Types

For each integer data type, there is a largest and smallest number that you can represent with that type. The table shown under “Integers” on page 2-6 lists the largest and smallest values for each integer data type in the “Range of Values” column.

You can also obtain these values with the `intmax` and `intmin` functions:

```
intmax('int8')                intmin('int8')
ans =                         ans =
   127                           -128
```

If you convert a number that is larger than the maximum value of an integer data type to that type, MATLAB sets it to the maximum value. Similarly, if

you convert a number that is smaller than the minimum value of the integer data type, MATLAB sets it to the minimum value. For example,

```
x = int8(300)
x =
   127
```

```
x = int8(-300)
x =
   -128
```

Also, when the result of an arithmetic operation involving integers exceeds the maximum (or minimum) value of the data type, MATLAB sets it to the maximum (or minimum) value:

```
x = int8(100) * 3
x =
   127
```

```
x = int8(-100) * 3
x =
   -128
```

You can make MATLAB return a warning when your input is outside the range an integer data type. This is described in the next section.

Warnings for Integer Data Types

Use the `intwarning` function to make MATLAB return a warning message when it converts a number outside the range of an integer data type to that type, or when the result of an arithmetic operation overflows. There are four possible warning messages that you can turn on using `intwarning`:

Message Identifier	Reason for Warning
`MATLAB:intConvertOverflow`	Overflow when attempting to convert from a numeric class to an integer class
`MATLAB:intMathOverflow`	Overflow when attempting an integer arithmetic operation
`MATLAB:intConvertNonIntVal`	Attempt to convert a noninteger value to an integer
`MATLAB:intConvertNaN`	Attempt to convert NaN (Not a Number) to an integer

Querying the Present Warning State. Use the following command to display the state of all integer warnings:

```
intwarning('query')
   The state of warning 'MATLAB:intConvertNaN' is 'off'.
   The state of warning 'MATLAB:intConvertNonIntVal' is 'off'.
   The state of warning 'MATLAB:intConvertOverflow' is 'off'.
   The state of warning 'MATLAB:intMathOverflow' is 'off'.
```

To display the state of one or more selected warnings, index into the structure returned by `intwarning`. This example shows the current state of the `intConvertOverflow` warning:

```
iwState = intwarning('query');
iwState(3)
ans =
    identifier: 'MATLAB:intConvertOverflow'
         state: 'off'
```

Turning the Warning On. To enable all four integer warnings, use `intwarning` with the string `'on'`:

```
intwarning('on');
intwarning('query')
   The state of warning 'MATLAB:intConvertNaN' is 'on'.
   The state of warning 'MATLAB:intConvertNonIntVal' is 'on'.
   The state of warning 'MATLAB:intConvertOverflow' is 'on'.
   The state of warning 'MATLAB:intMathOverflow' is 'on'.
```

To enable one or more selected integer warnings, first make sure that all integer warnings are disabled:

```
intwarning('off');
```

Note that, in this state, the following conversion to a 16-bit integer overflows, but does not issue a warning:

```
x = int16(50000)
x =
  32767
```

Find which of the four warnings covers integer conversion. In this case, it is the third in the structure array:

```
iwState = intwarning('query');
iwState(3).identifier
ans =
   MATLAB:intConvertOverflow
```

Set the warning state to `'on'` in the structure, and then call `intwarning` using the structure as input:

```
iwState(3).state = 'on';
intwarning(iwState);
```

With the warning enabled, the overflow on conversion does issue the warning message:

```
x = int16(50000)
Warning: Out of range value converted to intmin('int16') or
intmax('int16').
x =
  32767
```

You can also use the following `for` loop to enable integer warnings selectively:

```
maxintwarn = 4;

for k = 1:maxintwarn
   if strcmp(iwState(k).identifier, 'MATLAB:intConvertOverflow')
      iwState(k).state = 'on';
      intwarning(iwState);
   end
end
```

Turning the Warning Off. To turn all integer warnings off (their default state when you start MATLAB), enter

```
intwarning('off')
```

To disable selected integer warnings, follow the steps shown for enabling warnings, but with the `state` field of the structure set to `'off'`:

```
iwState(3).identifier
ans =
```

```
   MATLAB:intConvertOverflow

iwState(3).state = 'off';
intwarning(iwState);
```

Turning Warnings On or Off Temporarily. When writing M-files that contain integer data types, it is sometimes convenient to temporarily turn integer warnings on, and then return the states of the warnings (`'on'` or `'off'`) to their previous settings. The following commands illustrate how to do this:

```
oldState = intwarning('on');

int8(200);
Warning: Out of range value converted to intmin('int8') or
intmax('int8').

intwarning(oldState)
```

To temporarily turn the warnings off, change the first line of the preceding code to

```
oldState = intwarning('off');
```

Recommended Usage of Math Overflow Warning. Enabling the `MATLAB:intMathOverflow` warning slows down integer arithmetic. It is recommended that you enable this particular warning only when you need to diagnose unusual behavior in your code, and disable it during normal program operation. The other integer warnings listed above do not affect program performance.

Note that calling `intwarning('on')` enables all four integer warnings, including the `intMathOverflow` warning that can have an effect on integer arithmetic.

Integer Functions

See Integer Functions on page 2-30 for a list of functions most commonly used with integers in MATLAB.

Floating-Point Numbers

MATLAB represents floating-point numbers in either double-precision or single-precision format. The default is double precision, but you can make any number single precision with a simple conversion function.

This section covers:

Double-Precision Floating Point

MATLAB constructs the double-precision (or `double`) data type according to IEEE Standard 754 for double precision. Any value stored as a `double` requires 64 bits, formatted as shown in the table below:

Bits	Usage
63	Sign (`0` = positive, `1` = negative)
62 to 52	Exponent, biased by `1023`
51 to 0	Fraction `f` of the number `1.f`

Single-Precision Floating Point

MATLAB constructs the single-precision (or `single`) data type according to IEEE Standard 754 for single precision. Any value stored as a `single` requires 32 bits, formatted as shown in the table below:

Bits	Usage
31	Sign (0 = positive, 1 = negative)
30 to 23	Exponent, biased by 127
22 to 0	Fraction `f` of the number `1.f`

Because MATLAB stores numbers of type `single` using 32 bits, they require less memory than numbers of type `double`, which use 64 bits. However, because they are stored with fewer bits, numbers of type `single` are represented to less precision than numbers of type `double`.

Creating Floating-Point Data

Use double-precision to store values greater than approximately 3.4×10^{38} or less than approximately -3.4×10^{38}. For numbers that lie between these two limits, you can use either double- or single-precision, but single requires less memory.

Double-Precision. Because the default numeric type for MATLAB is `double`, you can create a `double` with a simple assignment statement:

```
x = 25.783;
```

The `whos` function shows that MATLAB has created a 1-by-1 array of type `double` for the value you just stored in `x`:

```
whos(x)
  Name      Size                    Bytes  Class

  x         1x1                         8  double
```

Use `isfloat` if you just want to verify that `x` is a floating-point number. This function returns logical 1 (`true`) if the input is a floating-point number, and logical 0 (`false`) otherwise:

```
isfloat(x)
ans =
     1
```

You can convert other numeric data, characters or strings, and logical data to double precision using the MATLAB function, `double`. This example converts a signed integer to double-precision floating point:

```
y = int64(-589324077574);          % Create a 64-bit integer

x = double(y)                      % Convert to double
x =
   -5.8932e+011
```

Single-Precision. Because MATLAB stores numeric data as a `double` by default, you need to use the `single` conversion function to create a single-precision number:

```
x = single(25.783);
```

The `whos` function returns the attributes of variable `x` in a structure. The `bytes` field of this structure shows that when `x` is stored as a single, it requires just 4 bytes compared with the 8 bytes to store it as a `double`:

```
xAttrib = whos('x');
xAttrib.bytes
ans =
     4
```

You can convert other numeric data, characters or strings, and logical data to single precision using the `single` function. This example converts a signed integer to single-precision floating point:

```
y = int64(-589324077574);          % Create a 64-bit integer

x = single(y)                      % Convert to single
x =
   -5.8932e+011
```

Arithmetic Operations on Floating-Point Numbers

This section describes which data types you can use in arithmetic operations with floating-point numbers.

Double-Precision. You can perform basic arithmetic operations with `double` and any of the following other data types. When one or more operands is an integer (scalar or array), the `double` operand must be a scalar. The result is of type `double`, except where noted otherwise:

- `single` — The result is of type `single`
- `double`
- `int*` or `uint*` — The result has the same data type as the integer operand
- `char`
- `logical`

This example performs arithmetic on data of types `char` and `double`. The result is of type `double`:

```
c = 'uppercase' - 32;

class(c)
ans =
   double

char(c)
ans =
   UPPERCASE
```

Single-Precision. You can perform basic arithmetic operations with `single` and any of the following other data types. The result is always `single`:

- `single`
- `double`
- `char`
- `logical`

In this example, 7.5 defaults to type `double`, and the result is of type `single`:

```
x = single([1.32 3.47 5.28]) .* 7.5;

class(x)
ans =
   single
```

Largest and Smallest Values for Floating-Point Data Types

For the `double` and `single` data types, there is a largest and smallest number that you can represent with that type.

Double-Precision. The MATLAB functions `realmax` and `realmin` return the maximum and minimum values that you can represent with the `double` data type:

```
str = 'The range for double is:\n\t%g to %g and\n\t %g to  %g';
sprintf(str, -realmax, -realmin, realmin, realmax)

ans =
The range for double is:
   -1.79769e+308 to -2.22507e-308 and
    2.22507e-308 to  1.79769e+308
```

Numbers larger than `realmax` or smaller than `-realmax` are assigned the values of positive and negative infinity, respectively:

```
realmax + .0001e+308
ans =
   Inf

-realmax - .0001e+308
ans =
  -Inf
```

Single-Precision. The MATLAB functions `realmax` and `realmin`, when called with the argument `'single'`, return the maximum and minimum values that you can represent with the `single` data type:

```
str = 'The range for single is:\n\t%g to %g and\n\t %g to  %g';
sprintf(str, -realmax('single'), -realmin('single'), ...
   realmin('single'), realmax('single'))

ans =
The range for single is:
   -3.40282e+038 to -1.17549e-038 and
    1.17549e-038 to  3.40282e+038
```

Numbers larger than `realmax`('single') or smaller than `-realmax` ('single') are assigned the values of positive and negative infinity, respectively:

```
realmax('single') + .0001e+038
ans =
   Inf

-realmax('single') - .0001e+038
ans =
  -Inf
```

Accuracy of Floating-Point Data

If the result of a floating-point arithmetic computation is not as precise as you had expected, it is likely caused by the limitations of your computer's hardware. Probably, your result was a little less exact because the hardware had insufficient bits to represent the result with perfect accuracy; therefore, it truncated the resulting value.

Double-Precision. Because there are only a finite number of double-precision numbers, you cannot represent all numbers in double-precision storage. On any computer, there is a small gap between each double-precision number and the next larger double-precision number. You can determine the size of this gap, which limits the precision of your results, using the `eps` function. For example, to find the distance between `5` and the next larger double-precision number, enter

```
format long

eps(5)
ans =
```

```
    8.881784197001252e-016
```

This tells you that there are no double-precision numbers between 5 and `5 + eps(5)`. If a double-precision computation returns the answer 5, the result is only accurate to within `eps(5)`.

The value of `eps(x)` depends on `x`. This example shows that, as `x` gets larger, so does `eps(x)`:

```
eps(50)
ans =
    7.105427357601002e-015
```

If you enter `eps` with no input argument, MATLAB returns the value of `eps(1)`, the distance from `1` to the next larger double-precision number.

Single-Precision. Similarly, there are gaps between any two single-precision numbers. If `x` has type `single`, `eps(x)` returns the distance between `x` and the next larger single-precision number. For example,

```
x = single(5);
eps(x)
```

returns

```
ans =
  4.7684e-007
```

Note that this result is larger than `eps(5)`. Because there are fewer single-precision numbers than double-precision numbers, the gaps between the single-precision numbers are larger than the gaps between double-precision numbers. This means that results in single-precision arithmetic are less precise than in double-precision arithmetic.

For a number `x` of type `double`, `eps(single(x))` gives you an upper bound for the amount that `x` is rounded when you convert it from `double` to `single`. For example, when you convert the double-precision number `3.14` to `single`, it is rounded by

```
double(single(3.14) - 3.14)
ans =
```

```
  1.0490e-007
```

The amount that `3.14` is rounded is less than

```
eps(single(3.14))
ans =
  2.3842e-007
```

Avoiding Common Problems with Floating-Point Arithmetic

Almost all operations in MATLAB are performed in double-precision arithmetic conforming to the IEEE standard 754. Because computers only represent numbers to a finite precision (double precision calls for 52 mantissa bits), computations sometimes yield mathematically nonintuitive results. It is important to note that these results are not bugs in MATLAB.

Use the following examples to help you identify these cases:

Example 1 — Round-Off or What You Get Is Not What You Expect.
The decimal number `4/3` is not exactly representable as a binary fraction. For this reason, the following calculation does not give zero, but rather reveals the quantity `eps`.

```
e = 1 - 3*(4/3 - 1)

e =
  2.2204e-016
```

Similarly, `0.1` is not exactly representable as a binary number. Thus, you get the following nonintuitive behavior:

```
a = 0.0;
for i = 1:10
  a = a + 0.1;
end
a == 1

ans =
     0
```

Note that the order of operations can matter in the computation:

```
b = 1e-16 + 1 - 1e-16;
c = 1e-16 - 1e-16 + 1;
b == c

ans =
     0
```

There are gaps between floating-point numbers. As the numbers get larger, so do the gaps, as evidenced by:

```
(2^53 + 1) - 2^53

ans =
     0
```

Since `pi` is not really `pi`, it is not surprising that `sin(pi)` is not exactly zero:

```
sin(pi)

ans =
   1.224646799147353e-016
```

Example 2 — Catastrophic Cancellation. When subtractions are performed with nearly equal operands, sometimes cancellation can occur unexpectedly. The following is an example of a cancellation caused by swamping (loss of precision that makes the addition insignificant):

```
sqrt(1e-16 + 1) - 1

ans =
     0
```

Some functions in MATLAB, such as `expm1` and `log1p`, may be used to compensate for the effects of catastrophic cancellation.

Example 3 — Floating-Point Operations and Linear Algebra.
Round-off, cancellation, and other traits of floating-point arithmetic combine to produce startling computations when solving the problems of linear algebra. MATLAB warns that the following matrix `A` is ill-conditioned, and therefore the system `Ax = b` may be sensitive to small perturbations. Although the computation differs from what you expect in exact arithmetic, the result is correct.

```
A = [2 eps -eps; eps 1 1; -eps 1 1];
b = [2; eps + 2; -eps + 2];
x = A\b

x =
  1.0e+015 *
   0.000000000000001
   2.251799813685249
  -2.251799813685247
```

These are only a few of the examples showing how IEEE floating-point arithmetic affects computations in MATLAB. Note that all computations performed in IEEE 754 arithmetic are affected, this includes applications written in C or FORTRAN, as well as MATLAB. For more examples and information, see `Technical Note 1108  Common Problems with Floating-Point Arithmetic`.

Floating-Point Functions

See Floating-Point Functions on page 2-30 for a list of functions most commonly used with floating-point numbers in MATLAB.

References

The following references provide more information about floating-point arithmetic.

[1] Moler, Cleve, "Floating Points," *MATLAB News and Notes*, Fall, 1996. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf

[2] Moler, Cleve, *Numerical Computing with MATLAB*, S.I.A.M. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/moler/.

Complex Numbers

Complex numbers consist of two separate parts: a real part and an imaginary part. The basic imaginary unit is equal to the square root of `-1`. This is represented in MATLAB by either of two letters: `i` or `j`.

Creating Complex Numbers

The following statement shows one way of creating a complex value in MATLAB. The variable `x` is assigned a complex number with a real part of `2` and an imaginary part of `3`:

```
x = 2 + 3i;
```

Another way to create a complex number is using the `complex` function. This function combines two numeric inputs into a complex output, making the first input real and the second imaginary:

```
x = rand(3) * 5;
y = rand(3) * -8;

z = complex(x, y)
z =
   4.7842 -1.0921i   0.8648 -1.5931i   1.2616 -2.2753i
   2.6130 -0.0941i   4.8987 -2.3898i   4.3787 -3.7538i
   4.4007 -7.1512i   1.3572 -5.2915i   3.6865 -0.5182i
```

You can separate a complex number into its real and imaginary parts using the `real` and `imag` functions:

```
zr = real(z)
zr =
    4.7842    0.8648    1.2616
    2.6130    4.8987    4.3787
    4.4007    1.3572    3.6865

zi = imag(z)
zi =
   -1.0921   -1.5931   -2.2753
   -0.0941   -2.3898   -3.7538
   -7.1512   -5.2915   -0.5182
```

Complex Number Functions

See Complex Number Functions on page 2-31 for a list of functions most commonly used with MATLAB complex numbers in MATLAB.

Infinity and NaN

MATLAB uses the special values `inf`, `-inf`, and `NaN` to represent values that are positive and negative infinity, and not a number respectively.

Infinity

MATLAB represents infinity by the special value `inf`. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values. MATLAB also provides a function called `inf` that returns the IEEE arithmetic representation for positive infinity as a `double` scalar value.

Several examples of statements that return positive or negative infinity in MATLAB are shown here.

`x = 1/0`
`x =`
`Inf`
	`x = 1.e1000`
	`x =`
	`Inf`

`x = exp(1000)`
`x =`
`Inf`
	`x = log(0)`
	`x =`
	`-Inf`

Use the `isinf` function to verify that `x` is positive or negative infinity:

```
x = log(0);

isinf(x)
ans =
     1
```

NaN

MATLAB represents values that are not real or complex numbers with a special value called `NaN`, which stands for Not a Number. Expressions like `0/0` and `inf/inf` result in `NaN`, as do any arithmetic operations involving a `NaN`.

For example, the statement `n/0`, where `n` is complex, returns `NaN` for the real part of the result:

```
x = 7i/0
x =
      NaN +    Infi
```

Use the `isnan` function to verify that the real part of `x` is NaN:

```
isnan(real(x))
ans =
     1
```

MATLAB also provides a function called `NaN` that returns the IEEE arithmetic representation for `NaN` as a `double` scalar value:

```
x = NaN;

whos x
  Name      Size                    Bytes  Class

  x         1x1                         8  double
```

Logical Operations on NaN. Because two `NaN`s are not equal to each other, logical operations involving `NaN` always return false, except for a test for inequality, (`NaN ~= NaN`):

```
NaN > NaN
ans =
     0

NaN ~= NaN
ans =
     1
```

Infinity and NaN Functions

See Infinity and NaN Functions on page 2-31 for a list of functions most commonly used with `inf` and `NaN` in MATLAB.

Identifying Numeric Types

You can check the data type of a variable `x` using any of these commands.

Command	Operation
`whos x`	Display the data type of `x`.
`xType = class(x);`	Assign the data type of `x` to a variable.
`isnumeric(x)`	Determine if `x` is a numeric type.

`isa(x, 'integer')`
`isa(x, 'uint64')`
`isa(x, 'float')`
`isa(x, 'double')`
`isa(x, 'single')`
	Determine if `x` is the specified numeric type. (Examples for any integer, unsigned 64-bit integer, any floating point, double precision, and single precision are shown here).

`isreal(x)`	Determine if `x` is real or complex.
`isnan(x)`	Determine if `x` is Not a Number (`NaN`).
`isinf(x)`	Determine if `x` is infinite.
`isfinite(x)`	Determine if `x` is finite.

Display Format for Numeric Values

By default, MATLAB displays numeric output as 5-digit scaled, fixed-point values. You can change the way numeric values are displayed to any of the following:

- 5-digit scaled fixed point, floating point, or the best of the two
- 15-digit scaled fixed point, floating point, or the best of the two
- A ratio of small integers
- Hexadecimal (base 16)
- Bank notation

All available formats are listed on the `format` reference page.

To change the numeric display setting, use either the `format` function or the **Preferences** dialog box (accessible from the MATLAB **File** menu). The `format` function changes the display of numeric values for the duration of a single MATLAB session, while your Preferences settings remain active from one session to the next. These settings affect only how numbers are displayed, not how MATLAB computes or saves them.

Display Format Examples

Here are a few examples of the various formats and the output produced from the following two-element vector `x`, with components of different magnitudes.

Check the current format setting:

```
get(0, 'format')
ans =
   short
```

Set the value for `x` and display in 5-digit scaled fixed point:

```
x = [4/3 1.2345e-6]
x =
    1.3333    0.0000
```

Set the format to 5-digit floating point:

```
format short e
x
x =
  1.3333e+000  1.2345e-006
```

Set the format to 15-digit scaled fixed point:

```
format long
x
x =
   1.33333333333333   0.00000123450000
```

Set the format to `'rational'` for small integer ratio output:

```
format rational
x
x =
     4/3          1/810045
```

Set an integer value for `x` and display it in hexadecimal (base 16) format:

```
format hex
x = uint32(876543210)
x =
   343efcea
```

Setting Numeric Format in a Program

To temporarily change the numeric format inside a program, get the original format using the `get` function and save it in a variable. When you finish working with the new format, you can restore the original format setting using the `set` function as shown here:

```
origFormat = get(0, 'format');
format('rational');

  -- Work in rational format --

set(0,'format', origFormat);
```

Function Summary

MATLAB provides these functions for working with numeric data types:

- Integer Functions on page 2-30
- Floating-Point Functions on page 2-30
- Complex Number Functions on page 2-31
- Infinity and NaN Functions on page 2-31
- Type Identification Functions on page 2-32
- Output Formatting Functions on page 2-32

Integer Functions

Function	Description
`int8, int16, int32, int64`	Convert to signed 1-, 2-, 4-, or 8-byte integer.
`uint8, uint16, uint32, uint64`	Convert to unsigned 1-, 2-, 4-, or 8-byte integer.
`ceil`	Round towards plus infinity to nearest integer
`class`	Return the data type of an object.
`fix`	Round towards zero to nearest integer
`floor`	Round towards minus infinity to nearest integer
`isa`	Determine if input value has the specified data type.
`isinteger`	Determine if input value is an integer array.
`isnumeric`	Determine if input value is a numeric array.
`round`	Round towards the nearest integer

Floating-Point Functions

Function	Description
`double`	Convert to double precision.
`single`	Convert to single precision.
`class`	Return the data type of an object.
`isa`	Determine if input value has the specified data type.
`isfloat`	Determine if input value is a floating-point array.
`isnumeric`	Determine if input value is a numeric array.
`eps`	Return the floating-point relative accuracy. This value is the tolerance MATLAB uses in its calculations.

Floating-Point Functions (Continued)

Function	Description
`realmax`	Return the largest floating-point number your computer can represent.
`realmin`	Return the smallest floating-point number your computer can represent.

Complex Number Functions

Function	Description
`complex`	Construct complex data from real and imaginary components.
`i` or `j`	Return the imaginary unit used in constructing complex data.
`real`	Return the real part of a complex number.
`imag`	Return the imaginary part of a complex number.
`isreal`	Determine if a number is real or imaginary.

Infinity and NaN Functions

Function	Description
`inf`	Return the IEEE value for infinity.
`isnan`	Detect `NaN` elements of an array.
`isinf`	Detect infinite elements of an array.

Infinity and NaN Functions (Continued)

Function	Description
`isfinite`	Detect finite elements of an array.
`nan`	Return the IEEE value for Not a Number.

Type Identification Functions

Function	Description
`class`	Return data type (or class).
`isa`	Determine if input value is of the specified data type.
`isfloat`	Determine if input value is a floating-point array.
`isinteger`	Determine if input value is an integer array.
`isnumeric`	Determine if input value is a numeric array.
`isreal`	Determine if input value is real.
`whos`	Display the data type of input.

Output Formatting Functions

Function	Description
`format`	Control display format for output.

Logical Types

In this section...

Overview

The `logical` data type represents a logical `true` or `false` state using the numbers `1` and `0`, respectively. Certain MATLAB functions and operators return logical `true` or `false` to indicate whether a certain condition was found to be true or not. For example, the statement `(5 * 10) > 40` returns a logical `true` value.

Logical data does not have to be scalar; MATLAB supports arrays of logical values as well. For example, the following statement returns a vector of logicals indicating `false` for the first two elements and `true` for the last three:

```
[30 40 50 60 70] > 40
ans =
     0     0     1     1     1
```

Creating a Logical Array

One way of creating an array of logicals is to just enter a `true` or `false` value for each element. The `true` function returns logical one; the `false` function returns logical zero:

```
x = [true, true, false, true, false];
```

Logical Operations on an Array

You can also perform some logical operation on an array that yields an array of logicals:

```
x = magic(4) >= 9
```

```
x =
     1     0     0     1
     0     1     1     0
     1     0     0     1
     0     1     1     0
```

The MATLAB functions that have names beginning with `is` (e.g., `ischar`, `issparse`) also return a logical value or array:

```
a = [2.5 6.7 9.2 inf 4.8];

isfinite(a)
ans =
     1     1     1     0     1
```

This table shows some of the MATLAB operations that return a logical `true` or `false`.

Function	Operation
`true`, `false`	Setting value to `true` or `false`
`logical`	Numeric to logical conversion
`&` (`and`), `|` (`or`), `~` (`not`), `xor`, `any`, `all`	Logical operations
`&&`, `||`	Short-circuit AND and OR
`==` (`eq`), `~=` (`ne`), `<` (`lt`), `>` (`gt`), `<=` (`le`), `>=` (`ge`)	Relational operations
All `is*` functions, `cellfun`	Test operations
`strcmp`, `strncmp`, `strcmpi`, `strncmpi`	String comparisons

Sparse Logical Arrays

Logical arrays can also be sparse as long as they have no more than two dimensions:

```
x = sparse(magic(20) > 395)
x =
   (1,1)        1
   (1,4)        1
```

```
 (1,5)        1
(20,18)       1
(20,19)       1
```

How Logical Arrays Are Used

MATLAB has two primary uses for logical arrays:

- "Using Logicals in Conditional Statements" on page 2-35
- "Logical Indexing" on page 2-35

Most mathematics operations are not supported on logical values.

Using Logicals in Conditional Statements

Conditional statements are useful when you want to execute a block of code only when a certain condition is met. For example, the `sprintf` command shown below is valid only if `str` is a nonempty string. The statement

```
if ~isempty(str) && ischar(str)
```

checks for this condition and allows the `sprintf` to execute only if it is `true`:

```
str = 'Hello';

if ~isempty(str) && ischar(str)
   sprintf('Input string is ''%s''', str)
   end

ans =
   Input string is 'Hello'
```

Logical Indexing

A logical matrix provides a different type of array indexing in MATLAB. While most indices are numeric, indicating a certain row or column number, logical indices are positional. That is, it is the *position* of each 1 in the logical matrix that determines which array element is being referred to.

See “Using Logicals in Array Indexing” on page 1-22 for more information on this subject.

Identifying Logical Arrays

This table shows the commands you can use to determine whether or not an array `x` is logical. The last function listed, `cellfun`, operates on cell arrays, which you can read about in the section “Cell Arrays” on page 2-93.

Command	Operation
`whos(x)`	Display value and data type for `x`.
`islogical(x)`	Return `true` if array is logical.
`isa(x, 'logical')`	Return `true` if array is logical.
`class(x)`	Return string with data type name.
`cellfun('islogical', x)`	Check each cell array element for logical.

Characters and Strings

In this section...

Overview

In MATLAB, the term *string* refers to an array of Unicode characters. MATLAB represents each character internally as its corresponding numeric value. Unless you want to access these values, you can simply work with the characters as they display on screen.

You can use `char` to hold an `m`-by-`n` array of strings as long as each string in the array has the same length. (This is because MATLAB arrays must be rectangular.) To hold an array of strings of unequal length, use a cell array.

The string is actually a vector whose components are the numeric codes for the characters. The actual characters displayed depend on the character set encoding for a given font.

Creating Character Arrays

Specify character data by placing characters inside a pair of single quotes. For example, this line creates a 1-by-13 character array called `name`:

```
name = 'Thomas R. Lee';
```

In the workspace, the output of `whos` shows

```
Name        Size          Bytes  Class

name        1x13             26  char
```

You can see that each character uses 2 bytes of storage internally.

The `class` and `ischar` functions show that `name` is a character array:

```
class(name)
ans =
   char

ischar(name)
ans =
    1
```

You also can join two or more character arrays together to create a new character array. To do this, use either the string concatenation function, `strcat`, or the MATLAB concatenation operator, `[]`. The latter preserves any trailing spaces found in the input arrays:

```
name = 'Thomas R. Lee';
title = ' Sr. Developer';

strcat(name,',',title)
ans =
   Thomas R. Lee, Sr. Developer
```

To concatenate strings vertically, use `strvcat`.

Creating Two-Dimensional Character Arrays

When creating a two-dimensional character array, be sure that each row has the same length. For example, this line is legal because both input rows have exactly 13 characters:

```
name = ['Thomas R. Lee' ; 'Sr. Developer']
name =
   Thomas R. Lee
```

```
    Sr. Developer
```

When creating character arrays from strings of different lengths, you can pad the shorter strings with blanks to force rows of equal length:

```
name = ['Thomas R. Lee   '; 'Senior Developer'];
```

A simpler way to create string arrays is to use the `char` function. `char` automatically pads all strings to the length of the longest input string. In the following example, `char` pads the 13-character input string `'Thomas R. Lee'` with three trailing blanks so that it will be as long as the second string:

```
name = char('Thomas R. Lee','Senior Developer')
name =
   Thomas R. Lee
   Senior Developer
```

When extracting strings from an array, use the `deblank` function to remove any trailing blanks:

```
trimname = deblank(name(1,:))
trimname =
   Thomas R. Lee

size(trimname)
ans =
    1    13
```

Expanding Character Arrays

Expanding the size of an existing character array by assigning additional characters to indices beyond the bounds of the array such that part of the array becomes padded with zeros, is generally not recommended. See the documentation on “Expanding a Character Array” on page 1-35 in the MATLAB Programming documentation.

Cell Arrays of Strings

Creating strings in a regular MATLAB array requires that all strings in the array be of the same length. This often means that you have to pad blanks at

the end of strings to equalize their length. However, another type of MATLAB array, the cell array, can hold different sizes and types of data in an array without padding. Cell arrays provide a more flexible way to store strings of varying length.

For details on cell arrays, see "Cell Arrays" on page 2-93.

Converting to a Cell Array of Strings

The `cellstr` function converts a character array into a cell array of strings. Consider the character array

```
data = ['Allison Jones';'Development  ';'Phoenix      '];
```

Each row of the matrix is padded so that all have equal length (in this case, 13 characters).

Now use `cellstr` to create a column vector of cells, each cell containing one of the strings from the `data` array:

```
celldata = cellstr(data)
celldata =
    'Allison Jones'
    'Development'
    'Phoenix'
```

Note that the `cellstr` function strips off the blanks that pad the rows of the input string matrix:

```
length(celldata{3})
ans =
    7
```

The `iscellstr` function determines if the input argument is a cell array of strings. It returns a logical 1 (`true`) in the case of `celldata`:

```
iscellstr(celldata)
ans =
     1
```

Use `char` to convert back to a standard padded character array:

```
strings = char(celldata)
strings =
   Allison Jones
   Development
   Phoenix

length(strings(3,:))
ans =
    13
```

Functions for Cell Arrays of Strings

This table describes the MATLAB functions for working with cell arrays.

Function	Description
`cellstr`	Convert a character array to a cell array of strings.
`char`	Convert a cell array of strings to a character array.
`deblank`	Remove trailing blanks from a string.
`iscellstr`	Return `true` for a cell array of strings.
`sort`	Sort elements in ascending or descending order.
`strcat`	Concatenate strings.
`strcmp`	Compare strings.
`strmatch`	Find possible matches for a string.

You can also use the following `set` functions with cell arrays of strings.

Function	Description
`intersect`	Set the intersection of two vectors.
`ismember`	Detect members of a set.
`setdiff`	Return the set difference of two vectors.
`setxor`	Set the exclusive OR of two vectors.

Function	Description
union	Set the union of two vectors.
unique	Set the unique elements of a vector.

Formatting Strings

The following MATLAB functions offer the capability to compose a string that includes ordinary text and data formatted to your specification:

- sprintf — Write formatted data to an output string
- fprintf — Write formatted data to an output file or the command window
- warning — Display formatted data in a warning message
- error — Display formatted data in an error message and abort
- assert — Generate an error when a condition is violated

The syntax of each of these functions includes formatting operators similar to those used by the printf function in the C programming language. For example, %s tells MATLAB to interpret an input value as a string, and %d means to format an integer using decimal notation.

The general formatting syntax for these functions is

```
functionname(..., format_string, value1, value2, ..., valueN)
```

where the format_string argument expresses the basic content and formatting of the final output string, and the value arguments that follow supply data values to be inserted into the string.

Here is a sample sprintf statement, also showing the resulting output string:

```
sprintf('The price of %s on %d/%d/%d was $%.2f.', ...
        'bread', 7, 1, 2006, 2.49)
ans =
    The price of bread on 7/1/2006 was $2.49.
```

The following sections cover

Note The examples in this section of the documentation use only the `sprintf` function to demonstrate how string formatting works. However, you can run the examples using the `fprintf`, `warning`, and `error` functions as well.

The Format String

The first input argument in the `sprintf` statement shown above is the format string:

```
'The price of %s on %d/%d/%d was $%.2f.'
```

The `string` argument can include ordinary text, formatting operators and, in some cases, special characters. The formatting operators for this particular string are: `%s`, `%d`, `%d`, `%d`, and `%.2f`.

Following the string argument are five additional input arguments, one for each of the formatting operators in the string:

```
'bread', 7, 1, 2006, 2.49
```

When MATLAB processes the format string, it replaces each operator with one of these input values.

Special Characters. Special characters are a part of the text in the string. But, because they cannot be entered as ordinary text, they require a unique character sequence to represent them. Use any of the following character sequences to insert special characters into the output string.

To Insert . . .	Use . . .
Backspace	\b
Form feed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t
Backslash	\\
Percent character	%%

Input Value Arguments

In the syntax

```
functionname(..., format_string, value1, value2, ..., valueN)
```

The `value` arguments must immediately follow `string` in the argument list. In most instances, you supply one of these `value` arguments for each formatting operator used in `string`. Scalars, vectors, and numeric and character arrays are valid value arguments. You cannot use cell arrays or structures.

If you include fewer formatting operators than there are values to insert, MATLAB reuses the operators on the additional values. This example shows two formatting operators and six values to insert into the string:

```
sprintf('%s = %d\n', 'A', 479, 'B', 352, 'C', 651)
ans =
    A = 479
    B = 352
    C = 651
```

Sequential and Numbered Argument Specification.

You can place `value` arguments in the argument list either sequentially (that is, in the same order in which their formatting operators appear in the string), or by identifier (adding a number to each operator that identifies which `value` argument to replace it with). By default, MATLAB uses sequential ordering.

To specify arguments by a numeric identifier, add a positive integer followed by a $ sign immediately after the % sign in the operator. Numbered argument specification is explained in more detail under the topic “Value Identifiers” on page 2-51.

Ordered Sequentially	Ordered By Identifier

`sprintf('%s %s %s', ...`
`    '1st', '2nd', '3rd')`
`ans =`
`    1st 2nd 3rd`
	`sprintf('%3$s %2$s %1$s', ...`
	`    '1st', '2nd', '3rd')`
	`ans =`
	`     3rd 2nd 1st`

Vectorizing. Instead of using individual value arguments, you can use a vector or matrix as the source of data input values, as shown here:

```
sprintf('%d ', magic(4))
ans =
    16 5 9 4 2 11 7 14 3 10 6 15 13 8 12 1
```

When using the %s operator, MATLAB interprets integers as characters:

```
mvec = [77 65 84 76 65 66];

sprintf('%s ', mvec)
ans =
    MATLAB
```

The Formatting Operator

Formatting operators tell MATLAB how to format the numeric or character value arguments and where to insert them into the string. These operators control the notation, alignment, significant digits, field width, and other aspects of the output string.

A formatting operator begins with a % character, which may be followed by a series of one or more numbers, characters, or symbols, each playing a role in further defining the format of the insertion value. The final entry in this series is a single *conversion character* that MATLAB uses to define the notation style for the inserted data. Conversion characters used in MATLAB are based on those used by the `printf` function in the C programming language.

Here is a simple example showing five formatting variations for a common value:

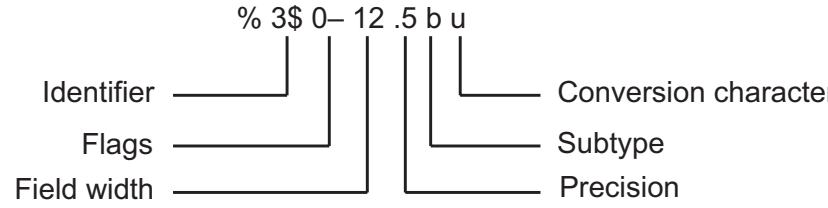
```
A = pi*100*ones(1,5);

sprintf(' %f \n %.2f \n %+.2f \n %12.2f \n %012.2f \n', A)
ans =
    314.159265       % Display in fixed-point notation (%f)
    314.16           % Display 2 decimal digits (%.2f)
    +314.16          % Display + for positive numbers (%+.2f)
          314.16     % Set width to 12 characters (%12.2f)
    000000314.16     % Replace leading spaces with 0 (%012.2f)
```

Constructing the Formatting Operator

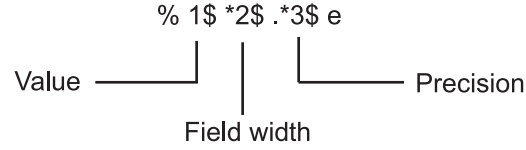
The fields that make up a formatting operator in MATLAB are as shown here, in the order they appear from right to left in the operator. The rightmost field, the conversion character, is required; the five to the left of that are optional. Each of these fields is explained in a section below:

- Conversion Character — Specifies the notation of the output.
- Subtype — Further specifies any nonstandard types.
- Precision — Sets the number of digits to display to the right of the decimal point.
- Field Width — Sets the minimum number of digits to display.
- Flags — Controls the alignment, padding, and inclusion of plus or minus signs.
- Value Identifiers — Map formatting operators to value input arguments. Use the identifier field when value arguments are not in a sequential order in the argument list.

Here is an example of a formatting operator that uses all six fields. (Space characters are not allowed in the operator. They are shown here only to improve readability of the figure).



An alternate syntax, that enables you to supply values for the field width and precision fields from values in the argument list, is shown below. See the section "Specifying Field Width and Precision Outside the format String" on page 2-52 for information on when and how to use this syntax. (Again, space characters are shown only to improve readability of the figure.)



Each field of the formatting operator is described in the following sections. These fields are covered as they appear going from right to left in the formatting operator, starting with the `Conversion Character` and ending with the `Identifier` field.

Conversion Character. The conversion character specifies the notation of the output. It consists of a single character and appears last in the format specifier. It is the only required field of the format specifier other than the leading `%` character.

Specifier	Description
`c`	Single character
`d`	Decimal notation (signed)
`e`	Exponential notation (using a lowercase `e` as in `3.1415e+00`)
`E`	Exponential notation (using an uppercase `E` as in `3.1415E+00`)
`f`	Fixed-point notation

Specifier	Description
g	The more compact of %e or %f. (Insignificant zeros do not print.)
G	Same as %g, but using an uppercase E
o	Octal notation (unsigned)
s	String of characters
u	Decimal notation (unsigned)
x	Hexadecimal notation (using lowercase letters a–f)
X	Hexadecimal notation (using uppercase letters A–F)

This example uses conversion characters to display the number 46 in decimal, fixed-point, exponential, and hexadecimal formats:

```
A = 46*ones(1,4);

sprintf('%d   %f   %e   %X', A)
ans =
    46   46.000000   4.600000e+001   2E
```

Subtype. The subtype field is a single alphabetic character that immediately precedes the conversion character. The following nonstandard subtype specifiers are supported for the conversion characters %o, %u, %x, and %X.

b	The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like '%bx'.
t	The underlying C data type is a float rather than an unsigned integer.

Precision. precision in a formatting operator is a nonnegative integer that tells MATLAB how many digits to the right of the decimal point to use when formatting the corresponding input value. The precision field consists of a nonnegative integer that immediately follows a period and extends to the first alphabetic character after that period. For example, the specifier %7.3f, has a precision of 3.

Here are some examples of how the `precision` field affects different types of notation:

```
sprintf('%g   %.2g   %f   %.2f', pi*50*ones(1,4))
ans =
    157.08   1.6e+002   157.079633   157.08
```

Precision is not usually used in format specifiers for strings (i.e., `%s`). If you do use it on a string and if the value `p` in the precision field is less than the number of characters in the string, MATLAB displays only `p` characters of the string and truncates the rest.

You can also supply the value for a precision field from outside of the format specifier. See the section "Specifying Field Width and Precision Outside the format String" on page 2-52 for more information on this.

For more information on the use of `precision` in formatting, see "Setting Field Width and Precision" on page 2-51.

Field Width. `Field width` in a formatting operator is a nonnegative integer that tells MATLAB the minimum number of digits or characters to use when formatting the corresponding input value. For example, the specifier `%7.3f`, has a width of 7.

Here are some examples of how the `width` field affects different types of notation:

```
sprintf('|%e|%15e|%f|%15f|', pi*50*ones(1,4))
ans =
    |1.570796e+002|  1.570796e+002|157.079633|     157.079633|
```

When used on a string, the `field width` can determine whether MATLAB pads the string with spaces. If `width` is less than or equal to the number of characters in the string, it has no effect.

```
sprintf('%30s', 'Pad left with spaces')
ans =
          Pad left with spaces
```

You can also supply a value for `field width` from outside of the format specifier. See the section "Specifying Field Width and Precision Outside the format String" on page 2-52 for more information on this.

For more information on the use of `field width` in formatting, see "Setting Field Width and Precision" on page 2-51.

Flags. You can control the alignment of the output using any of these optional flags:

Character	Description	Example
A minus sign (`-`)	Left-justifies the converted argument in its field	`%-5.2d`
A plus sign (`+`)	Always prints a sign character (+ or –)	`%+5.2d`
Zero (`0`)	Pad with zeros rather than spaces.	`%05.2f`

Right- and left-justify the output. The default is to right-justify:

```
sprintf('right-justify: %12.2f\nleft-justify: %-12.2f', ...
        12.3, 12.3)
ans =
    right-justify:        12.30
    left-justify: 12.30
```

Display a + sign for positive numbers. The default is to omit the + sign:

```
sprintf('no sign: %12.2f\nsign: %+12.2f', ...
        12.3, 12.3)
ans =
    no sign:        12.30
    sign:       +12.30
```

Pad to the left with spaces or zeros. The default is to use space-padding:

```
sprintf('space-padded: %12.2f\nzero-padded: %012.2f', ...
```

```
        5.2, 5.2)
ans =
    space-padded:         5.20
    zero-padded: 000000005.20
```

Note You can specify more than one flag in a formatting operator.

Value Identifiers. By default, MATLAB inserts data values from the argument list into the string in a sequential order. If you have a need to use the value arguments in a nonsequential order, you can override the default by using a numeric identifier in each format specifier. Specify nonsequential arguments with an integer immediately following the % sign, followed by a $ sign.

Ordered Sequentially	Ordered By Identifier

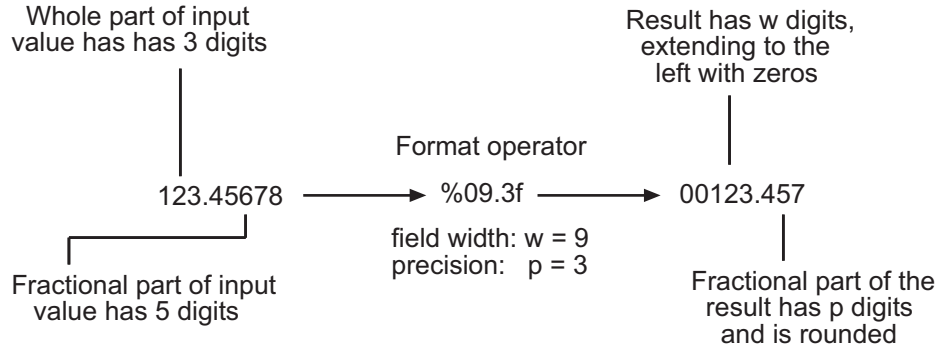
`sprintf('%s %s %s', ...`
`    '1st', '2nd', '3rd')`
`ans =`
`    1st 2nd 3rd`
	`sprintf('%3$s %2$s %1$s', ...`
	`    '1st', '2nd', '3rd')`
	`ans =`
	`     3rd 2nd 1st`

Setting Field Width and Precision

This section provides further information on the use of the field width and precision fields of the formatting operator:

- "Effect on the Output String" on page 2-51
- "Specifying Field Width and Precision Outside the format String" on page 2-52
- "Using Identifiers In the Width and Precision Fields" on page 2-53

Effect on the Output String. The figure below illustrates the way in which the field width and precision settings affect the output of the string formatting functions. In this figure, the zero following the % sign in the formatting operator means to add leading zeros to the output string rather than space characters:



General rules for formatting

- If precision is not specified, it defaults to 6.
- If precision (`p`) is less than the number of digits in the fractional part of the input value (`f`), then only `p` digits are shown to the right of the decimal point in the output, and that fractional value is rounded.
- If precision (`p`) is greater than the number of digits in the fractional part of the input value (`f`), then `p` digits are shown to the right of the decimal point in the output, and the fractional part is extended to the right with `p-f` zeros.
- If field width is not specified, it defaults to `precision + 1` + the number of digits in the whole part of the input value.
- If field width (`w`) is greater than `p+1` plus the number of digits in the whole part of the input value (`n`), then the whole part of the output value is extended to the left with `w-(n+1+p)` space characters or zeros, depending on whether or not the `zero` flag is set in the `Flags` field. The default is to extend the whole part of the output with space characters.

Specifying Field Width and Precision Outside the format String. To specify field width or precision using values from a sequential argument list, use an asterisk (`*`) in place of the `field width` or `precision` field of the formatting operator.

This example formats and displays three numbers. The formatting operator for the first, `%*f`, has an asterisk in the field width location of the formatting

operator, specifying that just the field width, 15, is to be taken from the argument list. The second operator, `%.*f` puts the asterisk after the decimal point meaning, that it is the precision that is to take its value from the argument list. And the third operator, `%*.*f`, specifies both field width and precision in the argument list:

```
 sprintf('%*f   %.*f   %*.*f', ...
        15, 123.45678, ...     % Width for 123.45678 is 15
        3, 16.42837, ...       % Precision for rand*20 is .3
        6, 4, pi)              % Width & Precision for pi is 6.4
ans =
     123.456780   16.428   3.1416
```

You can mix the two styles. For example, this statement gets the field width from the argument list and the precision from the format string:

```
sprintf('%*.2f', 5, 123.45678)
ans =
    123.46
```

Using Identifiers In the Width and Precision Fields. You can also derive field width and precision values from a nonsequential (i.e., numbered) argument list. Inside the formatting operator, specify `field width` and/or `precision` with an asterisk followed by an identifier number, followed by a `$` sign.

This example from the previous section shows how to obtain field width and precision from a sequential argument list:

```
sprintf('%*f   %.*f   %*.*f', ...
        15, 123.45678, ...
        3, 16.42837, ...
        6, 4, pi)

ans =
     123.456780   16.428   3.1416
```

Here is an example of how to do the same thing using numbered ordering. Field width for the first output value is 15, precision for the second value is 3, and field width and precision for the third value is 6 and 4, respectively.

If you specify field width or precision with identifiers, then you must specify the value with an identifier as well:

```
sprintf('%1$*4$f   %2$.*5$f   %3$*6$.*7$f', ...
123.45678, 16.42837, pi, 15, 3, 6, 4)

ans =
     123.456780   16.428   3.1416
```

Restrictions for Using Identifiers

If any of the formatting operators in a string include an identifier field, then all of the operators in that string must do the same; you cannot use both sequential and nonsequential ordering in the same function call.

Valid Syntax	Invalid Syntax

`sprintf('%d %d %d %d', ...`
`        1, 2, 3, 4)`
`ans =`
`    1 2 3 4`
	`sprintf('%d %3$d %d %d', ...`
	`        1, 2, 3, 4)`
	`ans =`
	`    1`

If your command provides more value arguments than there are formatting operators in the format string, MATLAB reuses the operators. However, MATLAB allows this only for commands that use sequential ordering. You cannot reuse formatting operators when making a function call with numbered ordering of the value arguments.

Valid Syntax	Invalid Syntax

`sprintf('%d', 1, 2, 3, 4)`
`ans =`
`    1234`
	`sprintf('%1$d', 1, 2, 3, 4)`
	`ans =`
	`    1`

Also, do not use identifiers when the value arguments are in the form of a vector or array:

Valid Syntax	Invalid Syntax

`v = [1.4 2.7 3.1];`
 	
`sprintf('%.4f %.4f %.4f', v)`
`ans =`
`    1.4000 2.7000 3.1000`
	`v = [1.4 2.7 3.1];`
	
	`sprintf('%3$.4f %1$.4f %2$.4f', v)`
	`ans =`
	`   Empty string: 1-by-0`

String Comparisons

There are several ways to compare strings and substrings:

- You can compare two strings, or parts of two strings, for equality.
- You can compare individual characters in two strings for equality.
- You can categorize every element within a string, determining whether each element is a character or white space.

These functions work for both character arrays and cell arrays of strings.

Comparing Strings for Equality

You can use any of four functions to determine if two input strings are identical:

- `strcmp` determines if two strings are identical.
- `strncmp` determines if the first `n` characters of two strings are identical.
- `strcmpi` and `strncmpi` are the same as `strcmp` and `strncmp`, except that they ignore case.

Consider the two strings

```
str1 = 'hello';
str2 = 'help';
```

Strings `str1` and `str2` are not identical, so invoking `strcmp` returns logical `0` (`false`). For example,

```
C = strcmp(str1,str2)
```

```
C =
     0
```

Note For C programmers, this is an important difference between the MATLAB `strcmp` and C `strcmp()`functions, where the latter returns 0 if the two strings are the same.

The first three characters of `str1` and `str2` are identical, so invoking `strncmp` with any value up to 3 returns `1`:

```
C = strncmp(str1, str2, 2)
C =
     1
```

These functions work cell-by-cell on a cell array of strings. Consider the two cell arrays of strings

```
A = {'pizza'; 'chips'; 'candy'};
B = {'pizza'; 'chocolate'; 'pretzels'};
```

Now apply the string comparison functions:

```
strcmp(A,B)
ans =
     1
     0
     0
strncmp(A,B,1)
ans =
     1
     1
     0
```

Comparing for Equality Using Operators

You can use MATLAB relational operators on character arrays, as long as the arrays you are comparing have equal dimensions, or one is a scalar. For example, you can use the equality operator (==) to determine where the matching characters are in two strings:

```
A = 'fate';
B = 'cake';

A == B
ans =
    0   1   0   1
```

All of the relational operators (>, >=, <, <=, ==, ~=) compare the values of corresponding characters.

Categorizing Characters Within a String

There are three functions for categorizing characters inside a string:

1. `isletter` determines if a character is a letter.
2. `isspace` determines if a character is white space (blank, tab, or new line).
3. `isstrprop` checks characters in a string to see if they match a category you specify, such as
   - Alphabetic
   - Alphanumeric
   - Lowercase or uppercase
   - Decimal digits
   - Hexadecimal digits
   - Control characters
   - Graphic characters
   - Punctuation characters
   - White-space characters

For example, create a string named `mystring`:

```
mystring = 'Room 401';
```

`isletter` examines each character in the string, producing an output vector of the same length as `mystring`:

```
A = isletter(mystring)
A =
    1   1   1   1   0   0   0   0
```

The first four elements in `A` are logical `1` (`true`) because the first four characters of `mystring` are letters.

Searching and Replacing

MATLAB provides several functions for searching and replacing characters in a string. (MATLAB also supports search and replace operations using regular expressions. See "Regular Expressions" on page 3-30.)

Consider a string named `label`:

```
label = 'Sample 1, 10/28/95';
```

The `strrep` function performs the standard search-and-replace operation. Use `strrep` to change the date from `'10/28'` to `'10/30'`:

```
newlabel = strrep(label, '28', '30')
newlabel =
   Sample 1, 10/30/95
```

`findstr` returns the starting position of a substring within a longer string. To find all occurrences of the string `'amp'` inside `label`, use

```
position = findstr('amp', label)
position =
    2
```

The position within `label` where the only occurrence of `'amp'` begins is the second character.

The `strtok` function returns the characters before the first occurrence of a delimiting character in an input string. The default delimiting characters are the set of white-space characters. You can use the `strtok` function to parse a sentence into words. For example,

```
function allWords = words(inputString)
remainder = inputString;
allWords = '';
```

```
while (any(remainder))
  [chopped,remainder] = strtok(remainder);
  allWords = strvcat(allWords, chopped);
end
```

The `strmatch` function looks through the rows of a character array or cell array of strings to find strings that begin with a given series of characters. It returns the indices of the rows that begin with these characters:

```
maxstrings = strvcat('max', 'minimax', 'maximum')
maxstrings =
   max
   minimax
   maximum

strmatch('max', maxstrings)
ans =
     1
     3
```

Converting from Numeric to String

The functions listed in this table provide a number of ways to convert numeric data to character strings.

Function	Description	Example
`char`	Convert a positive integer to an equivalent character. (Truncates any fractional parts.)	`[72 105]` → `'Hi'`
`int2str`	Convert a positive or negative integer to a character type. (Rounds any fractional parts.)	`[72 105]` → `'72 105'`
`num2str`	Convert a numeric type to a character type of the specified precision and format.	`[72 105]` → `'72/105/'` (format set to `%1d/`)
`mat2str`	Convert a numeric type to a character type of the specified precision, returning a string MATLAB can evaluate.	`[72 105]` → `'[72 105]'`

Function	Description	Example
dec2hex	Convert a positive integer to a character type of hexadecimal base.	[72 105] → '48 69'
dec2bin	Convert a positive integer to a character type of binary base.	[72 105] → '1001000 1101001'
dec2base	Convert a positive integer to a character type of any base from 2 through 36.	[72 105] → '110 151' (base set to 8)

Converting to a Character Equivalent

The char function converts integers to Unicode character codes and returns a string composed of the equivalent characters:

```
x = [77 65 84 76 65 66];
char(x)
ans =
   MATLAB
```

Converting to a String of Numbers

The int2str, num2str, and mat2str functions convert numeric values to strings where each character represents a separate digit of the input value. The int2str and num2str functions are often useful for labeling plots. For example, the following lines use num2str to prepare automated labels for the *x*-axis of a plot:

```
function plotlabel(x, y)
plot(x, y)
str1 = num2str(min(x));
str2 = num2str(max(x));
out = ['Value of f from ' str1 ' to ' str2];
xlabel(out);
```

Converting to a Specific Radix

Another class of conversion functions changes numeric values into strings representing a decimal value in another base, such as binary or hexadecimal representation. This includes dec2hex, dec2bin, and dec2base.

Converting from String to Numeric

The functions listed in this table provide a number of ways to convert character strings to numeric data.

Function	Description	Example
`uintN` (e.g., `uint8`)	Convert a character to an integer code that represents that character.	`'Hi'` → `72 105`
`str2num`	Convert a character type to a numeric type.	`'72 105'` → `[72 105]`
`str2double`	Similar to `str2num`, but offers better performance and works with cell arrays of strings.	`{'72' '105'}` → `[72 105]`
`hex2num`	Convert a numeric type to a character type of specified precision, returning a string that MATLAB can evaluate.	`'A'` → `'-1.4917e-154'`
`hex2dec`	Convert a character type of hexadecimal base to a positive integer.	`'A'` → `10`
`bin2dec`	Convert a positive integer to a character type of binary base.	`'1010'` → `10`
`base2dec`	Convert a positive integer to a character type of any base from 2 through 36.	`'12'` → `10` (if `base == 8`)

Converting from a Character Equivalent

Character arrays store each character as a 16-bit numeric value. Use one of the integer conversion functions (e.g., `uint8`) or the `double` function to convert strings to their numeric values, and `char` to revert to character representation:

```
name = 'Thomas R. Lee';

name = double(name)
name =
    84  104  111  109  97  115  32  82  46  32  76  101  101

name = char(name)
name =
   Thomas R. Lee
```

Converting from a Numeric String

Use `str2num` to convert a character array to the numeric value represented by that string:

```
str = '37.294e-1';

val = str2num(str)
val =
    3.7294
```

The `str2double` function converts a cell array of strings to the double-precision values represented by the strings:

```
c = {'37.294e-1'; '-58.375'; '13.796'};

d = str2double(c)
d =
    3.7294
  -58.3750
   13.7960

whos
  Name      Size                    Bytes  Class

  c         3x1                       224  cell
  d         3x1                        24  double
```

Converting from a Specific Radix

To convert from a character representation of a nondecimal number to the value of that number, use one of these functions: `hex2num`, `hex2dec`, `bin2dec`, or `base2dec`.

The `hex2num` and `hex2dec` functions both take hexadecimal (base 16) inputs, but `hex2num` returns the IEEE double-precision floating-point number it represents, while `hex2dec` converts to a decimal integer.

Function Summary

MATLAB provides these functions for working with character arrays:

Functions to Create Character Arrays

Function	Description
`'str'`	Create the string specified between quotes.
`blanks`	Create a string of blanks.
`sprintf`	Write formatted data to a string.
`strcat`	Concatenate strings.
`strvcat`	Concatenate strings vertically.

Functions to Modify Character Arrays

Function	Description
`deblank`	Remove trailing blanks.
`lower`	Make all letters lowercase.
`sort`	Sort elements in ascending or descending order.
`strjust`	Justify a string.
`strrep`	Replace one string with another.

Functions to Modify Character Arrays (Continued)

Function	Description
`strtrim`	Remove leading and trailing white space.
`upper`	Make all letters uppercase.

Functions to Read and Operate on Character Arrays

Function	Description
`eval`	Execute a string with MATLAB expression.
`sscanf`	Read a string under format control.

Functions to Search or Compare Character Arrays

Function	Description
`findstr`	Find one string within another.
`strcmp`	Compare strings.
`strcmpi`	Compare strings, ignoring case.
`strmatch`	Find matches for a string.
`strncmp`	Compare the first N characters of strings.
`strncmpi`	Compare the first N characters, ignoring case.
`strtok`	Find a token in a string.

Functions to Determine Data Type or Content

Function	Description
`iscellstr`	Return `true` for a cell array of strings.
`ischar`	Return `true` for a character array.
`isletter`	Return `true` for letters of the alphabet.

Functions to Determine Data Type or Content (Continued)

Function	Description
`isstrprop`	Determine if a string is of the specified category.
`isspace`	Return `true` for white-space characters.

Functions to Convert Between Numeric and String Data Types

Function	Description
`char`	Convert to a character or string.
`cellstr`	Convert a character array to a cell array of strings.
`double`	Convert a string to numeric codes.
`int2str`	Convert an integer to a string.
`mat2str`	Convert a matrix to a string you can run `eval` on.
`num2str`	Convert a number to a string.
`str2num`	Convert a string to a number.
`str2double`	Convert a string to a double-precision value.

Functions to Work with Cell Arrays of Strings as Sets

Function	Description
`intersect`	Set the intersection of two vectors.
`ismember`	Detect members of a set.
`setdiff`	Return the set difference of two vectors.
`setxor`	Set the exclusive OR of two vectors.
`union`	Set the union of two vectors.
`unique`	Set the unique elements of a vector.

Dates and Times

In this section...

Overview

MATLAB represents date and time information in either of three formats: date strings, serial date numbers, or date vectors. You have the choice of using any of these formats. If you work with more than one date and time format, MATLAB provides functions to help you easily convert from one format to another, (e.g., from a string to a serial date number).

When using date strings, you have an additional option of choosing from 19 different string styles to express date and/or time information.

Types of Date Formats

The three MATLAB date and time formats are

- "Date Strings" on page 2-67
- "Serial Date Numbers" on page 2-67
- "Date Vectors" on page 2-68

This table shows examples of the three formats.

Date Format	Example
Date string	`02-Oct-1996`
Serial date number	`729300`
Date vector	`1996 10 2 0 0 0`

Date Strings

There are a number of different styles in which to express date and time information as a date string. For example, several possibilities for October 31, 2003 at 3:45:17 in the afternoon are

```
31-Oct-2003 15:45:17
10/31/03
15:45:17
03:45:17 PM
```

If you are working with a small number of dates at the MATLAB command line, then date strings are often the most convenient format to use.

Note The MATLAB `date` function returns the current date as a string.

Serial Date Numbers

A serial date number represents a calendar date as the number of days that has passed since a fixed base date. In MATLAB, serial date number 1 is January 1, 0000. MATLAB also uses serial time to represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So the string '31-Oct-2003, 6:00 pm' in MATLAB is date number 731885.75.

MATLAB works internally with serial date numbers. If you are using functions that handle large numbers of dates or doing extensive calculations with dates, you get better performance if you use date numbers.

Note The MATLAB `now` function returns the current date and time as a serial date number.

Date Vectors

Date vectors are an internal format for some MATLAB functions; you do not typically use them in calculations. A date vector contains the elements `[year month day hour minute second]`.

Note The MATLAB `clock` function returns the current date and time as a serial vector.

Conversions Between Date Formats

Functions that convert between date formats are shown below.

Function	Description
`datenum`	Convert a date string to a serial date number.
`datestr`	Convert a serial date number to a date string.
`datevec`	Split a date number or date string into individual date elements.

Here are some examples of conversions from one date format to another:

```
d1 = datenum('02-Oct-1996')
d1 =
    729300

d2 = datestr(d1 + 10)
d2 =
   12-Oct-1996

dv1 = datevec(d1)
dv1 =
    1996      10       2       0       0       0

dv2 = datevec(d2)
dv2 =
    1996      10      12       0       0       0
```

Date String Formats

The `datenum` function is important for doing date calculations efficiently. `datenum` takes an input string in any of several formats, with `'dd-mmm-yyyy'`, `'mm/dd/yyyy'`, or `'dd-mmm-yyyy, hh:mm:ss.ss'` most common. You can form up to six fields from letters and digits separated by any other characters:

- The day field is an integer from 1 to 31.
- The month field is either an integer from 1 to 12 or an alphabetic string with at least three characters.
- The year field is a nonnegative integer: if only two digits are specified, then a year 19yy is assumed; if the year is omitted, then the current year is used as a default.
- The hours, minutes, and seconds fields are optional. They are integers separated by colons or followed by `'AM'` or `'PM'`.

For example, if the current year is 1996, then these are all equivalent:

```
'17-May-1996'
'17-May-96'
'17-May'
'May 17, 1996'
'5/17/96'
'5/17'
```

and both of these represent the same time:

```
'17-May-1996, 18:30'
'5/17/96/6:30 pm'
```

Note that the default format for numbers-only input follows the American convention. Thus `3/6` is March 6, not June 3.

If you create a vector of input date strings, use a column vector and be sure all strings are the same length. Fill in with spaces or zeros.

Output Formats

The command `datestr(D, dateform)` converts a serial date `D` to one of 19 different date string output formats showing date, time, or both. The default output for dates is a day-month-year string: 01-Mar-1996. You select an alternative output format by using the optional integer argument `dateform`.

This table shows the date string formats that correspond to each `dateform` value.

dateform	Format	Description
0	`01-Mar-1996 15:45:17`	day-month-year hour:minute:second
1	`01-Mar-1996`	day-month-year
2	`03/01/96`	month/day/year
3	`Mar`	month, three letters
4	`M`	month, single letter
5	`3`	month
6	`03/01`	month/day
7	`1`	day of month
8	`Wed`	day of week, three letters
9	`W`	day of week, single letter
10	`1996`	year, four digits
11	`96`	year, two digits
12	`Mar96`	month year
13	`15:45:17`	hour:minute:second
14	`03:45:17 PM`	hour:minute:second AM or PM
15	`15:45`	hour:minute
16	`03:45 PM`	hour:minute AM or PM
17	`Q1-96`	calendar quarter-year
18	`Q1`	calendar quarter

Converting Output Format with datestr

Here are some examples of converting the date March 1, 1996 to various forms using the `datestr` function:

```
d = '01-Mar-1999'
d =
   01-Mar-1999

datestr(d)
ans =
     01-Mar-1999

datestr(d, 2)
ans =
     03/01/99

datestr(d, 17)
ans =
     Q1-99
```

Current Date and Time

The function `date` returns a string for today's date:

```
date
ans =
     02-Oct-1996
```

The function `now` returns the serial date number for the current date and time:

```
now
ans =
     729300.71

datestr(now)
ans =
     02-Oct-1996 16:56:16

datestr(floor(now))
ans =
     02-Oct-1996
```

Function Summary

MATLAB provides the following functions for time and date handling:

Current Date and Time Functions

Function	Description
`clock`	Return the current date and time as a date vector.
`date`	Return the current date as date string.
`now`	Return the current date and time as serial date number.

Conversion Functions

Function	Description
`datenum`	Convert to a serial date number.
`datestr`	Convert to a string representation of the date.
`datevec`	Convert to a date vector.

Utility Functions

Function	Description
`addtodate`	Modify a date number by field.
`calendar`	Return a matrix representing a calendar.
`datetick`	Label axis tick lines with dates.

Utility Functions (Continued)

Function	Description
`eomday`	Return the last day of a year and month.
`weekday`	Return the current day of the week.

Timing Measurement Functions

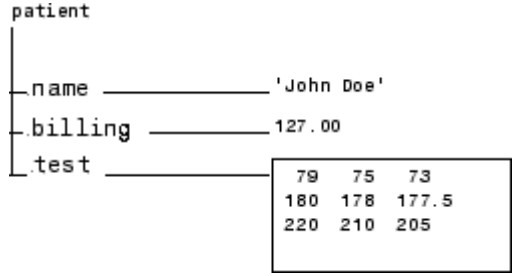
Function	Description
`cputime`	Return the total CPU time used by MATLAB since it was started.
`etime`	Return the time elapsed between two date vectors.
`tic, toc`	Measure the time elapsed between invoking `tic` and `toc`.

Structures

In this section...

Overview

Structures are MATLAB arrays with named “data containers” called *fields*. The fields of a structure can contain any kind of data. For example, one field might contain a text string representing a name, another might contain a scalar representing a billing amount, a third might hold a matrix of medical test results, and so on.



Like standard arrays, structures are inherently array oriented. A single structure is a 1-by-1 structure array, just as the value 5 is a 1-by-1 numeric array. You can build structure arrays with any valid size or shape, including multidimensional structure arrays.

Note The examples in this section focus on two-dimensional structure arrays. For examples of higher-dimension structure arrays, see "Multidimensional Arrays" on page 1-56.

Building Structure Arrays

You can build structures in two ways:

- Using assignment statements
- Using the `struct` function

Building Structure Arrays Using Assignment Statements

You can build a simple 1-by-1 structure array by assigning data to individual fields. MATLAB automatically builds the structure as you go along. For example, create the 1-by-1 `patient` structure array shown at the beginning of this section:

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

Now entering

```
patient
```

at the command line results in

```
name: 'John Doe'
billing: 127
test: [3x3 double]
```

`patient` is an array containing a structure with three fields. To expand the structure array, add subscripts after the structure name:

```
patient(2).name = 'Ann Lane';
patient(2).billing = 28.50;
patient(2).test = [68 70 68; 118 118 119; 172 170 169];
```

The `patient` structure array now has size `[1 2]`. Note that once a structure array contains more than a single element, MATLAB does not display individual field contents when you type the array name. Instead, it shows a summary of the kind of information the structure contains:

```
patient
patient =
1x2 struct array with fields:
    name
    billing
    test
```

You can also use the `fieldnames` function to obtain this information. `fieldnames` returns a cell array of strings containing field names.

As you expand the structure, MATLAB fills in unspecified fields with empty matrices so that

- All structures in the array have the same number of fields.
- All fields have the same field names.

For example, entering `patient(3).name = 'Alan Johnson'` expands the `patient` array to size `[1 3]`. Now both `patient(3).billing` and `patient(3).test` contain empty matrices.

Note Field sizes do not have to conform for every element in an array. In the `patient` example, the `name` fields can have different lengths, the `test` fields can be arrays of different sizes, and so on.

Building Structure Arrays Using the struct Function

You can preallocate an array of structures with the `struct` function. Its basic form is

```
strArray = struct('field1',val1,'field2',val2, ...)
```

where the arguments are field names and their corresponding values. A field value can be a single value, represented by any MATLAB data construct, or a cell array of values. All field values in the argument list must be of the same scale (single value or cell array).

You can use different methods for preallocating structure arrays. These methods differ in the way in which the structure fields are initialized. As an example, consider the allocation of a 1-by-3 structure array, `weather`, with the structure fields `temp` and `rainfall`. Three different methods for allocating such an array are shown in this table.

Method	Syntax	Initialization
`struct`	`weather(3) = struct('temp', 72, ... 'rainfall', 0.0);`	`weather(3)` is initialized with the field values shown. The fields for the other structures in the array, `weather(1)` and `weather(2)`, are initialized to the empty matrix.
`struct` with `repmat`	`weather = repmat(struct('temp', ... 72, 'rainfall', 0.0), 1, 3);`	All structures in the `weather` array are initialized using one set of field values.
`struct` with cell array syntax	`weather = ...` `struct('temp', {68, 80, 72}, ... 'rainfall', {0.2, 0.4, 0.0});`	The structures in the `weather` array are initialized with distinct field values specified with cell arrays.

Naming conventions for Structure Field Names

MATLAB structure field names are required to follow the same rules as standard MATLAB variables:

1. Field names must begin with a letter, which may be followed by any combination of letters, digits, and underscores. The following statements are all invalid:

```
w = setfield(w, 'My.Score', 3);
```

```
w = setfield(w, '1stScore', 3);
w = setfield(w, '1+1=3', 3);
w = setfield(w, '@MyScore', 3);
```

2. Although field names can be of any length, MATLAB uses only the first N characters of the field name, (where N is the number returned by the function `namelengthmax`), and ignores the rest.

```
N= namelengthmax
N=
   63
```

3. MATLAB distinguishes between uppercase and lowercase characters. Field name `length` is not the same as field name `Length`.

4. In most cases, you should refrain from using the names of functions and variables as field names.

See "Adding Fields to Structures" on page 2-82 and "Deleting Fields from Structures" on page 2-83 for more information on working with field names.

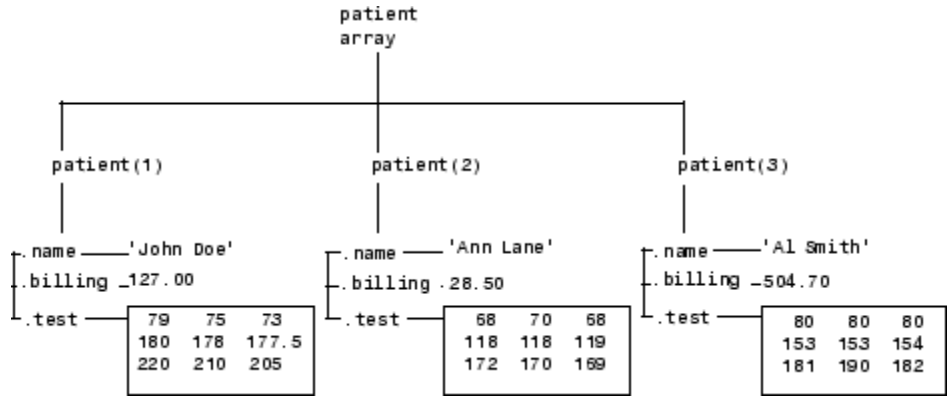
Memory Requirements for Structures

You do not necessarily need a contiguous block of memory to store a structure. The memory for each field in the structure needs to be contiguous, but not the entire structure itself.

Accessing Data in Structure Arrays

Using structure array indexing, you can access the value of any field or field element in a structure array. Likewise, you can assign a value to any field or field element. You can also access the fields of an array of structures in the form of a comma-separated list.

For the examples in this section, consider this structure array.



You can access subarrays by appending standard subscripts to a structure array name. For example, the line below results in a 1-by-2 structure array:

```
mypatients = patient(1:2)
1x2 struct array with fields:
    name
    billing
    test
```

The first structure in the `mypatients` array is the same as the first structure in the `patient` array:

```
mypatients(1)
ans =
       name: 'John Doe'
    billing: 127
       test: [3x3 double]
```

To access a field of a particular structure, include a period (.) after the structure name followed by the field name:

```
str = patient(2).name
str =
   Ann Lane
```

To access elements within fields, append the appropriate indexing mechanism to the field name. That is, if the field contains an array, use array subscripting; if the field contains a cell array, use cell array subscripting, and so on:

```
test2b = patient(3).test(2,2)
test2b =
   153
```

Use the same notations to assign values to structure fields, for example,

```
patient(3).test(2,2) = 7;
```

You can extract field values for multiple structures at a time. For example, the line below creates a 1-by-3 vector containing all of the `billing` fields:

```
bills = [patient.billing]
bills =
  127.0000   28.5000  504.7000
```

Similarly, you can create a cell array containing the `test` data for the first two structures:

```
tests = {patient(1:2).test}
tests =
    [3x3 double]    [3x3 double]
```

Using Dynamic Field Names

The most common way to access the data in a structure is by specifying the name of the field that you want to reference. Another means of accessing structure data is to use dynamic field names. These names express the field as a variable expression that MATLAB evaluates at run-time. The dot-parentheses syntax shown here makes `expression` a dynamic field name:

```
structName.(expression)
```

Index into this field using the standard MATLAB indexing syntax. For example, to evaluate *`expression`* into a field name and obtain the values of that field at columns `1` through `25` of row `7`, use

```
structName.(expression)(7,1:25)
```

Dynamic Field Names Example

The `avgscore` function shown below computes an average test score, retrieving information from the `testscores` structure using dynamic field names:

```
function avg = avgscore(student, first, last)
for k = first:last
   scores(k) = testscores.(student).week(k);
end
avg = sum(scores)/(last - first + 1);
```

You can run this function using different values for the dynamic field `student`. First, initialize the structure that contains scores for a 25 week period:

```
testscores.Ann_Lane.week(1:25) = ...
  [95 89 76 82 79 92 94 92 89 81 75 93 ...
   85 84 83 86 85 90 82 82 84 79 96 88 98];

testscores.William_King.week(1:25) = ...
  [87 80 91 84 99 87 93 87 97 87 82 89 ...
   86 82 90 98 75 79 92 84 90 93 84 78 81];
```

Now run `avgscore`, supplying the students name fields for the `testscores` structure at runtime using dynamic field names:

```
avgscore(testscores, 'Ann_Lane', 7, 22)
ans =
   85.2500

avgscore(testscores, 'William_King', 7, 22)
ans =
   87.7500
```

Finding the Size of Structure Arrays

Use the `size` function to obtain the size of a structure array, or of any structure field. Given a structure array name as an argument, `size` returns a vector of array dimensions. Given an argument in the form `array(n).field`, the `size` function returns a vector containing the size of the field contents.

For example, for the 1-by-3 structure array `patient`, `size(patient)` returns the vector `[1 3]`. The statement `size(patient(1,2).name)` returns the length of the `name` string for element `(1,2)` of `patient`.

Adding Fields to Structures

You can add a field to every structure in an array by adding the field to a single structure. For example, to add a social security number field to the `patient` array, use an assignment like

```
patient(2).ssn = '000-00-0000';
```

Now `patient(2).ssn` has the assigned value. Every other structure in the array also has the `ssn` field, but these fields contain the empty matrix until you explicitly assign a value to them.

See "Naming conventions for Structure Field Names" on page 2-77 for guidelines to creating valid field names.

Adding or Modifying Fields With the setfield Function

The `setfield` function offers another way to add or modify fields of a structure. Given the structure

```
mystr(1,1).name = 'alice';
mystr(1,1).ID = 0;
mystr(2,1).name = 'gertrude';
mystr(2,1).ID = 1;
```

You can change the name field of `mystr(2,1)` using

```
mystr = setfield(mystr, {2,1}, 'name', 'ted');

mystr(2,1).name
ans =
   ted
```

Adding New Fields Dynamically

To add new fields to a structure, specifying the names for these fields at run-time, see the section on "Using Dynamic Field Names" on page 2-80.

Deleting Fields from Structures

You can remove a given field from every structure within a structure array using the `rmfield` function. Its most basic form is

```
struc2 = rmfield(array, 'field')
```

where `array` is a structure array and `'field'` is the name of a field to remove from it. To remove the `name` field from the `patient` array, for example, enter

```
patient = rmfield(patient, 'name');
```

Applying Functions and Operators

Operate on fields and field elements the same way you operate on any other MATLAB array. Use indexing to access the data on which to operate.

For example, this statement finds the mean across the rows of the `test` array in `patient(2)`:

```
mean((patient(2).test)');
```

There are sometimes multiple ways to apply functions or operators across fields in a structure array. One way to add all the `billing` fields in the `patient` array is

```
total = 0;
for k = 1:length(patient)
      total = total + patient(k).billing;
end
```

To simplify operations like this, MATLAB enables you to operate on all like-named fields in a structure array. Simply enclose the `array.field` expression in square brackets within the function call. For example, you can sum all the `billing` fields in the `patient` array using

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

This is equivalent to using the comma-separated list:

```
total = sum ([patient(1).billing, patient(2).billing, ...]);
```

This syntax is most useful in cases where the operand field is a scalar field:

Writing Functions to Operate on Structures

You can write functions that work on structures with specific field architectures. Such functions can access structure fields and elements for processing.

Note When writing M-file functions to operate on structures, you must perform your own error checking. That is, you must ensure that the code checks for the expected fields.

As an example, consider a collection of data that describes measurements, at different times, of the levels of various toxins in a water source. The data consists of fifteen separate observations, where each observation contains three separate measurements.

You can organize this data into an array of 15 structures, where each structure has three fields, one for each of the three measurements taken.

The function `concen`, shown below, operates on an array of structures with specific characteristics. Its arguments must contain the fields `lead`, `mercury`, and `chromium`:

```
function [r1, r2] = concen(toxtest);
% Create two vectors:
%  r1 contains the ratio of mercury to lead at each observation.
%  r2 contains the ratio of lead to chromium.
r1 = [toxtest.mercury] ./ [toxtest.lead];
r2 = [toxtest.lead] ./ [toxtest.chromium];

% Plot the concentrations of lead, mercury, and chromium
% on the same plot, using different colors for each.
lead = [toxtest.lead];
mercury = [toxtest.mercury];
chromium = [toxtest.chromium];

plot(lead, 'r');   hold on
```

```
plot(mercury, 'b')
plot(chromium, 'y');   hold off
```

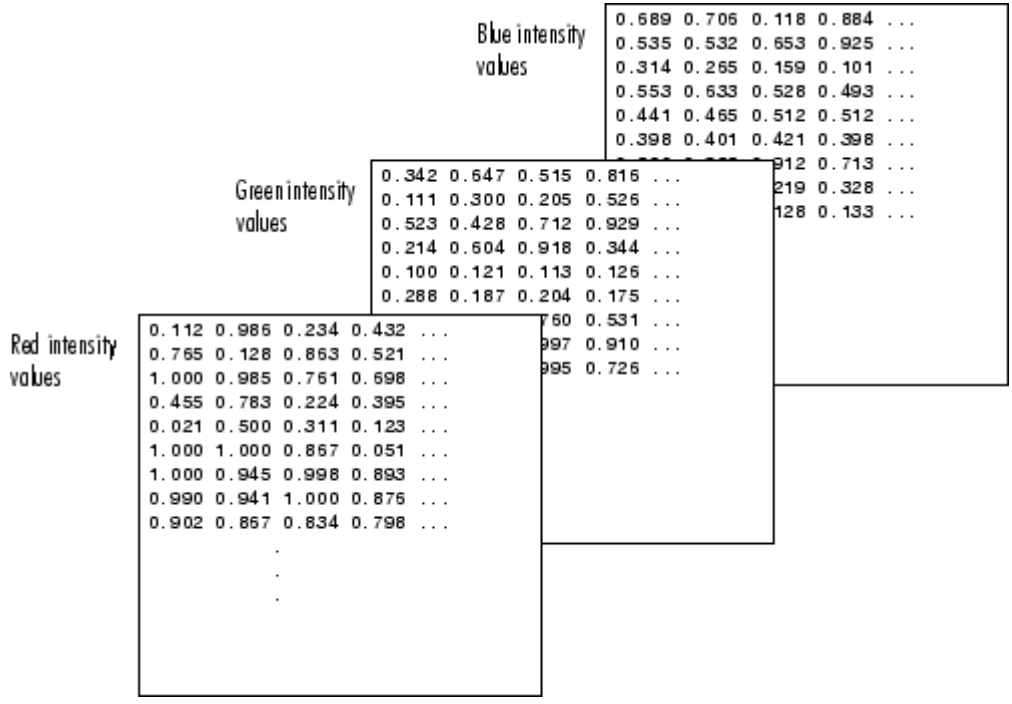
Try this function with a sample structure array like `test`:

```
test(1).lead = .007;
test(2).lead = .031;
test(3).lead = .019;

test(1).mercury = .0021;
test(2).mercury = .0009;
test(3).mercury = .0013;

test(1).chromium = .025;
test(2).chromium = .017;
test(3).chromium = .10;
```

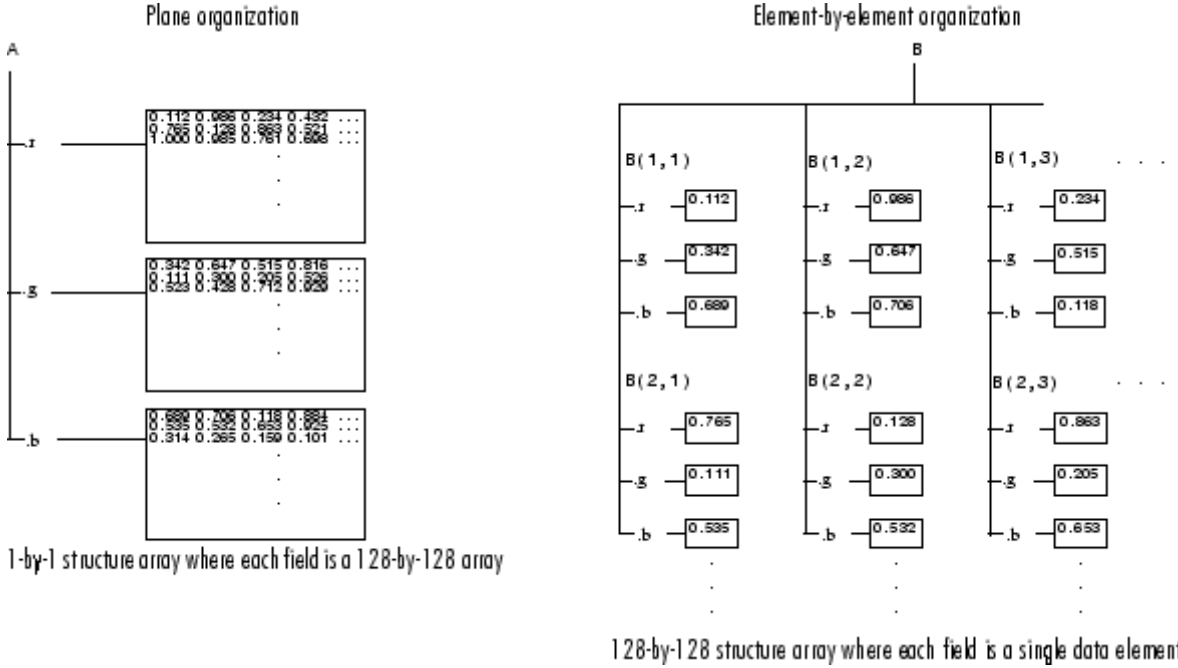
Organizing Data in Structure Arrays

The key to organizing structure arrays is to decide how you want to access subsets of the information. This, in turn, determines how you build the array that holds the structures, and how you break up the structure fields.

For example, consider a 128-by-128 RGB image stored in three separate arrays; `RED`, `GREEN`, and `BLUE`.


Blue intensity values
0.689 0.706 0.118 0.884 ...
0.535 0.532 0.653 0.925 ...
0.314 0.265 0.159 0.101 ...
0.553 0.633 0.528 0.493 ...
0.441 0.465 0.512 0.512 ...
0.398 0.401 0.421 0.398 ...
Green intensity values
0.342 0.647 0.515 0.816 ...
0.111 0.300 0.205 0.526 ...
0.523 0.428 0.712 0.929 ...
0.214 0.604 0.918 0.344 ...
0.100 0.121 0.113 0.126 ...
0.288 0.187 0.204 0.175 ...
Red intensity values
0.112 0.986 0.234 0.432 ...
0.765 0.128 0.863 0.521 ...
1.000 0.985 0.761 0.698 ...
0.455 0.783 0.224 0.395 ...
0.021 0.500 0.311 0.123 ...
1.000 1.000 0.867 0.051 ...
1.000 0.945 0.998 0.893 ...
0.990 0.941 1.000 0.876 ...
0.902 0.867 0.834 0.798 ...

There are at least two ways you can organize such data into a structure array.



Plane Organization

In the plane organization, shown to the left in the figure above, each field of the structure is an entire plane of the image. You can create this structure using

```
A.r = RED;
A.g = GREEN;
A.b = BLUE;
```

This approach allows you to easily extract entire image planes for display, filtering, or other tasks that work on the entire image at once. To access the entire red plane, for example, use

```
redPlane = A.r;
```

Plane organization has the additional advantage of being extensible to multiple images in this case. If you have a number of images, you can store them as `A(2)`, `A(3)`, and so on, each containing an entire image.

The disadvantage of plane organization is evident when you need to access subsets of the planes. To access a subimage, for example, you need to access each field separately:

```
redSub = A.r(2:12,13:30);
greenSub = A.g(2:12,13:30);
blueSub = A.b(2:12,13:30);
```

Element-by-Element Organization

The element-by-element organization, shown to the right in the figure above, has the advantage of allowing easy access to subsets of data. To set up the data in this organization, use

```
for m = 1:size(RED,1)
   for n = 1:size(RED,2)
      B(m,n).r = RED(m,n);
      B(m,n).g = GREEN(m,n);
      B(m,n).b = BLUE(m,n);
   end
end
```

With element-by-element organization, you can access a subset of data with a single statement:

```
Bsub = B(1:10,1:10);
```

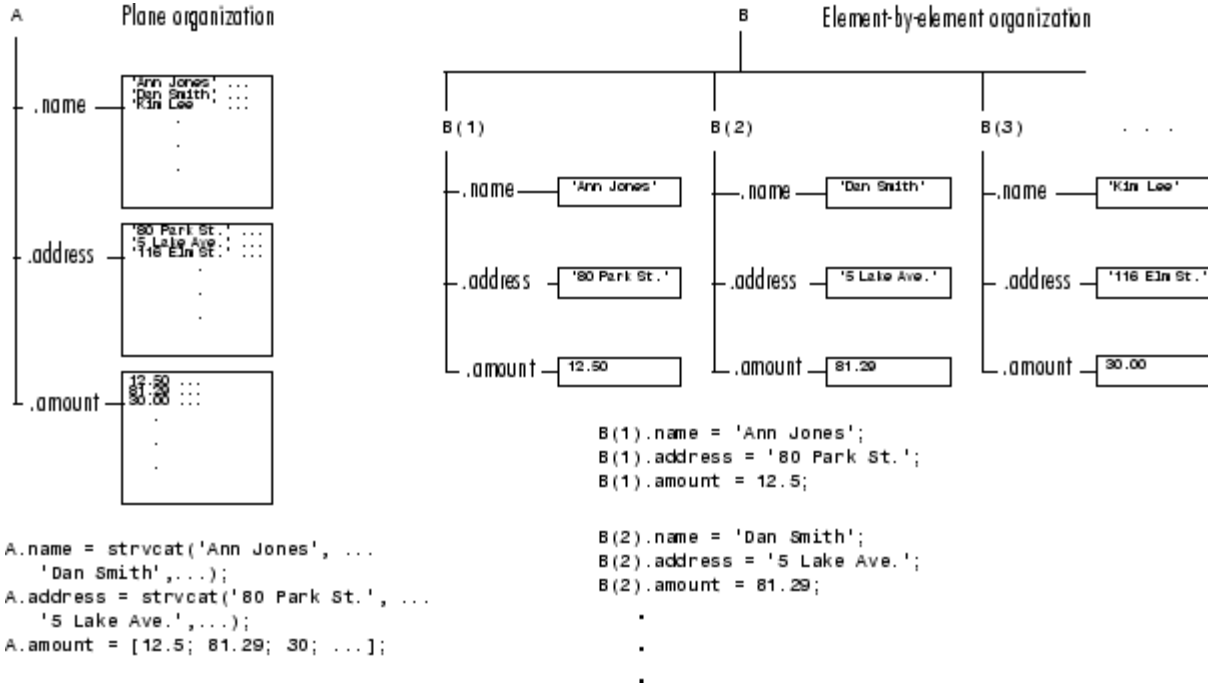
To access an entire plane of the image using the element-by-element method, however, requires a loop:

```
redPlane = zeros(128, 128);
for k = 1:(128 * 128)
    redPlane(k) = B(k).r;
end
```

Element-by-element organization is not the best structure array choice for most image processing applications; however, it can be the best for other applications wherein you will routinely need to access corresponding subsets of structure fields. The example in the following section demonstrates this type of application.

Example — A Simple Database

Consider organizing a simple database.




```
A.name = strvcat('Ann Jones', ...
   'Dan Smith',...);
A.address = strvcat('80 Park St.', ...
   '5 Lake Ave.',...);
A.amount = [12.5; 81.29; 30; ...];
```

```
B(1).name = 'Ann Jones';
B(1).address = '80 Park St.';
B(1).amount = 12.5;

B(2).name = 'Dan Smith';
B(2).address = '5 Lake Ave.';
B(2).amount = 81.29;
```

Each of the possible organizations has advantages depending on how you want to access the data:

- Plane organization makes it easier to operate on all field values at once. For example, to find the average of all the values in the `amount` field,
  - Using plane organization

```
avg = mean(A.amount);
```

- Using element-by-element organization

```
avg = mean([B.amount]);
```

- Element-by-element organization makes it easier to access all the information related to a single client. Consider an M-file, `client.m`, which displays the name and address of a given client on screen.

  Using plane organization, pass individual fields.

```
function client(name,address)
disp(name)
disp(address)
```

  To call the `client` function,

```
client(A.name(2,:),A.address(2,:))
```

  Using element-by-element organization, pass an entire structure.

```
function client(B)
disp(B)
```

  To call the `client` function,

```
client(B(2))
```

- Element-by-element organization makes it easier to expand the string array fields. If you do not know the maximum string length ahead of time for plane organization, you may need to frequently recreate the `name` or `address` field to accommodate longer strings.

Typically, your data does not dictate the organization scheme you choose. Rather, you must consider how you want to access and operate on the data.

Nesting Structures

A structure field can contain another structure, or even an array of structures. Once you have created a structure, you can use the `struct` function or direct assignment statements to nest structures within existing structure fields.

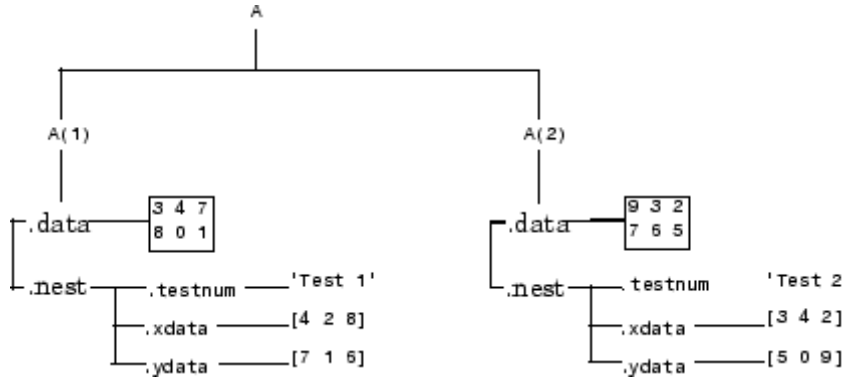
Building Nested Structures with the struct Function

To build nested structures, you can nest calls to the `struct` function. For example, create a 1-by-1 structure array:

```
A = struct('data', [3 4 7; 8 0 1], 'nest',...
    struct('testnum', 'Test 1', 'xdata', [4 2 8],...
    'ydata', [7 1 6]));
```

You can build nested structure arrays using direct assignment statements. These statements add a second element to the array:

```
A(2).data = [9 3 2; 7 6 5];
A(2).nest.testnum = 'Test 2';
A(2).nest.xdata = [3 4 2];
A(2).nest.ydata = [5 0 9];
```



Indexing Nested Structures

To index nested structures, append nested field names using dot notation. The first text string in the indexing expression identifies the structure array, and subsequent expressions access field names that contain other structures.

For example, the array `A` created earlier has three levels of nesting:

- To access the nested structure inside `A(1)`, use `A(1).nest`.
- To access the `xdata` field in the nested structure in `A(2)`, use `A(2).nest.xdata`.
- To access element 2 of the `ydata` field in `A(1)`, use `A(1).nest.ydata(2)`.

Function Summary

This table describes the MATLAB functions for working with structures.

Function	Description
`deal`	Deal inputs to outputs.
`fieldnames`	Get structure field names.
`isfield`	Return `true` if the field is in a structure array.
`isstruct`	Return `true` for structures.
`rmfield`	Remove a structure field.
`struct`	Create or convert to a structure array.
`struct2cell`	Convert a structure array into a cell array.

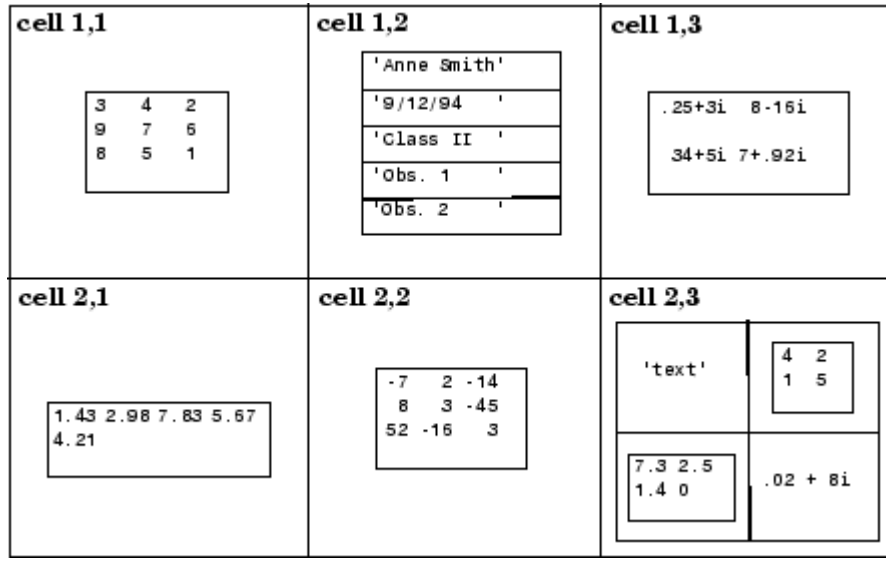
Cell Arrays

In this section...

Overview

A cell array provides a storage mechanism for dissimilar kinds of data. You can store arrays of different types and/or sizes within the cells of a cell array. For example, you can store a 1-by-50 char array, a 7-by-13 double array, and a 1-by-1 `uint32` in cells of the same cell array.

This illustration shows a cell array `A` that contains arrays of unsigned integers in `A{1,1}`, strings in `A{1,2}`, complex numbers in `A{1,3}`, floating-point numbers in `A{2,1}`, signed integers in `A{2,2}`, and another cell array in `A{2,3}`.



To access data in a cell array, you use the same type of matrix indexing as with other MATLAB matrices and arrays. However, with cell array indexing, you use curly braces, `{}`, instead of square brackets or parentheses around the array indices. For example, `A{2,3}` accesses the cell in row 2 and column 3 of cell array `A`.

Note The examples in this section focus on two-dimensional cell arrays. For examples of higher-dimension cell arrays, see “Multidimensional Arrays” on page 1-56.

Cell Array Operators

This table shows the operators used in constructing, concatenating, and indexing into the cells of a cell array.

Operation	Syntax	Description
Constructing	`C = {A B D E}`	Builds a cell array `C` that can contain data of unlike types in `A`, `B`, `D`, and `E`
Concatenating	`C3 = {C1 C2}`	Concatenates cell arrays `C1` and `C2` into a 2–element cell array `C3` such that `C3{1} = C1` and `C3{2} = C2`
	`C3 = [C1 C2]`	Concatenates the *contents* of cell arrays `C1` and `C2`
Indexing	`X = C(s)`	Returns the *cells* of array `C` that are specified by subscripts `s`
	`X = C{s}`	Returns the *contents* of the cells of `C` that are specified by subscripts `s`
	`X = C{s}(t)`	References one or more elements of an array that resides within a cell. Subscript `s` selects the cell, and subscript `t` selects the array element(s).

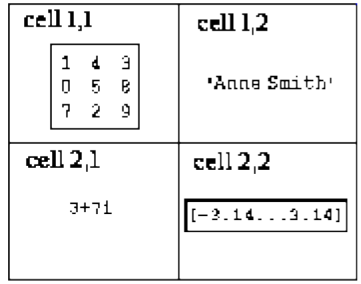
Creating a Cell Array

Creating cell arrays in MATLAB is similar to creating arrays of other MATLAB data types like double, character, etc. The main difference is that, when constructing a cell array, you enclose the array contents or indices with curly braces `{ }` instead of square brackets `[ ]`. The curly braces are cell array constructors, just as square brackets are numeric array constructors. Use commas or spaces to separate elements and semicolons to terminate each row.

For example, to create a 2-by-2 cell array `A`, type

```
A = {[1 4 3; 0 5 8; 7 2 9], 'Anne Smith'; 3+7i, -pi:pi/4:pi};
```

This results in the array shown below:



Note The notation `{}` denotes the empty cell array, just as `[]` denotes the empty matrix for numeric arrays. You can use the empty cell array in any cell array assignments.

For more information on cell arrays, refer to these topics:

- "Creating Cell Arrays Using Multiple Assignment Statements" on page 2-96
- "Building Cell Arrays with Concatenation" on page 2-98
- "Preallocating Cell Arrays with the cell Function" on page 2-99
- "Memory Requirements for Cell Arrays" on page 2-99

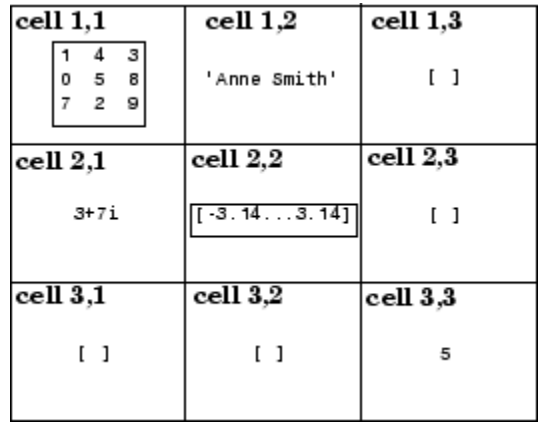
Creating Cell Arrays Using Multiple Assignment Statements

You also can create a cell array one cell at a time. MATLAB expands the size of the cell array with each assignment statement:

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
A(2,1) = {3+7i};
A(2,2) = {-pi:pi/4:pi};
```

If you assign data to a cell that is outside the dimensions of the current array, MATLAB automatically expands the array to include the subscripts you specify. It fills any intervening cells with empty matrices. For example, the assignment below turns the 2-by-2 cell array `A` into a 3-by-3 cell array.

```
A(3,3) = {5};
```



3–by-3 Cell Array

Note If you already have a numeric array of a given name, don't try to create a cell array of the same name by assignment without first clearing the numeric array. If you do not clear the numeric array, MATLAB assumes that you are trying to "mix" cell and numeric syntaxes, and generates an error. Similarly, MATLAB does not clear a cell array when you make a single assignment to it. If any of the examples in this section give unexpected results, clear the cell array from the workspace and try again.

Alternative Assignment Syntax. When assigning values to a cell array, either of the syntaxes shown below is valid. You can use the braces on the right side of the equation, enclosing the value being assigned as shown here:

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
```

Or use them on the left side, enclosing the array subscripts:

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
```

Building Cell Arrays with Concatenation

There are two ways that you can construct a new cell array from existing cell arrays:

- Concatenate entire cell arrays to individual cells of the new array. For example, join three cell arrays together to build a new cell array having three elements, each containing a cell array. This method uses the curly brace `{ }` operator.
- Concatenate the *contents* of the cells into a new array. For example, join cell arrays of size `m-by-n1`, `m-by-n2`, and `m-by-n3` together to yield a new cell array that is `m-by-(n1+n2+n3)` in size. This method uses the square bracket `[ ]` operator.

Here is an example. First, create three 3–row cell arrays of different widths.

```
C1 = {'Jan' 'Feb';  '10' '17';  uint16(2004) uint16(2001)};
C2 = {'Mar' 'Apr' 'May';  '31' '2' '10';  ...
      uint16(2006) uint16(2005) uint16(1994)};
C3 = {'Jun';  '23';  uint16(2002)};
```

This creates arrays `C1`, `C2`, and `C3`:

```
      C1                     C2                   C3
'Jan'   'Feb'       'Mar'   'Apr'   'May'       'Jun'
'10'    '17'        '31'    '2'     '10'        '23'
[2004]  [2001]      [2006]  [2005]  [1994]      [2002]
```

Use the curly brace operator to concatenate entire cell arrays, thus building a 1-by-3 cell array from the three initial arrays. Each cell of this new array holds its own cell array:

```
C4 = {C1 C2 C3}
C4 =
    {3x2 cell}    {3x3 cell}    {3x1 cell}
```

Now use the square bracket operator on the same combination of cell arrays. This time MATLAB concatenates the contents of the cells together and produces a 3-by-6 cell array:

```
C5 = [C1 C2 C3]
```

```
C5 =
    'Jan'     'Feb'     'Mar'     'Apr'     'May'     'Jun'
    '10'      '17'      '31'      '2'       '10'      '23'
    [2004]    [2001]    [2006]    [2005]    [1994]    [2002]
```

Preallocating Cell Arrays with the cell Function

The `cell` function enables you to preallocate empty cell arrays of the specified size. For example, this statement creates an empty 20-by-30 cell array:

```
B = cell(20, 30);
```

Use assignment statements to fill the cells of `B`.

It is more efficient to preallocate a cell array of a required size using the `cell` function and then assign data into it, than to grow a cell array as you go along using individual data assignments. The `cell` function, therefore, offers the most memory-efficient way of preallocating a cell array.

Memory Requirements for Cell Arrays

You do not necessarily need a contiguous block of memory to store a cell array. The memory for each cell needs to be contiguous, but not the entire array of cells.

Referencing Cells of a Cell Array

Because a cell array can contain different types of data stored in various array sizes, cell array indexing is a little more complex than indexing into a numeric or character array.

This section covers the following topics on constructing a cell array:

The examples in this section illustrate how to access the different components of a cell array. All of the examples use the following six-cell array which consists of different data types.

First, build the individual components of the example array:

```
rand('state', 0);    numArray = rand(3,5)*20;
chArray = ['Ann Lane'; 'John Doe'; 'Al Smith'];
cellArray = {1 4 3 9; 0 5 8 2; 7 2 9 2; 3 3 1 4};
logArray = numArray > 10;

stArray(1).name = chArray(1,:);
stArray(2).name = chArray(2,:);
stArray(1).billing = 28.50;
stArray(2).billing = 139.72;
stArray(1).test = numArray(1,:);
stArray(2).test = numArray(2,:);
```

and then construct the cell array from these components using the `{ }` operator:

```
A = {numArray, pi, stArray; chArray, cellArray, logArray};
```

To see what size and type of array occupies each cell in `A`, type the variable name alone:

```
A
A =
    [3x5 double]    [  3.1416]    [1x2 struct ]
    [3x8 char  ]    {4x4 cell}    [3x5 logical]
```

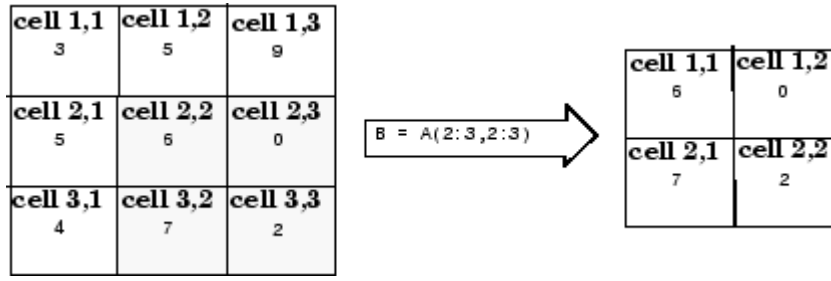
Manipulating Cells and the Contents of Cells

When working with cell arrays, you have a choice of selecting entire cells of an array to work with, or the contents of those cells. The first method is called *cell indexing*; the second is *content indexing*:

- Cell indexing enables you to work with whole cells of an array. You can access single or multiple cells within the array, but you cannot select anything less than the complete cell. If you want to manipulate the cells

of an array without regard to the contents of those cells, use cell indexing. This type of indexing is denoted by the parentheses operator `( )`.

Use cell indexing to assign any set of cells to another variable, creating a new cell array.



Creating a New Cell Array from an Existing One

- Content indexing gives you access to the contents of a cell. You can work with individual elements of an array within a cell, but you can only do so for one cell at a time. This indexing uses the curly brace operator `{ }`.

Displaying Parts of the Cell Array. Using the example cell array `A`, you can display information on the first row of cells using cell indexing. (The MATLAB colon operator functions the same when used with cell arrays as it does with numeric arrays):

```
A(1,:)
ans =
    [3x5 double]    [3.1416]    [1x2 struct]
```

To display the contents of these cells, use content indexing:

```
A{1,:}
ans =
   19.0026    9.7196    9.1294    8.8941   18.4363
    4.6228   17.8260    0.3701   12.3086   14.7641
   12.1369   15.2419   16.4281   15.8387    3.5253
ans =
    3.1416
ans =
1x2 struct array with fields:
    name
```

```
    billing
    test
```

In assignments, you can use content indexing to access only a single cell, not a subset of cells. For example, the statements `A{1,:} = value` and `B = A{1,:}` are both invalid. However, you can use a subset of cells any place you would normally use a comma-separated list of variables (for example, as function inputs or when building an array). See "Replacing Lists of Variables with Cell Arrays" on page 2-107 for details.

Assigning Cells. For cell indexing, assign the `double` array cell to `X`:

```
X = A(1,1)
X =
    [3x5 double]
```

`X` is a 1-by-1 cell array:

```
whos X
  Name      Size                    Bytes  Class

  X         1x1                       180  cell
```

For content indexing, assign the contents of the first cell of row 1 to `Y`:

```
Y = A{1,1}
Y =
   19.0026    9.7196    9.1294    8.8941   18.4363
    4.6228   17.8260    0.3701   12.3086   14.7641
   12.1369   15.2419   16.4281   15.8387    3.5253
```

`Y` is a 3-by-5 `double` array

```
whos Y
  Name      Size                    Bytes  Class

  Y         3x5                       120  double
```

Assigning Multiple Cells. Assigning multiple cells with cell indexing is similar to assigning a single cell. MATLAB creates a new cell array, each cell of which contains a cell array.

Create a 1-by-2 array with cells from `A(1,2)` and `A(1,3)`:

```
X = A(1,2:3)
X =
    [3.1416]    [1x2 struct]

whos X
  Name      Size                    Bytes  Class

  X         1x2                       808  cell
```

But assigning the contents of multiple cells returns a comma-separated list. In this case, you need one output variable on the left side of the assignment statement for each cell on the right side:

```
[Y1 Y2] = A{1,2:3}
Y1 =
    3.1416
Y2 =
1x2 struct array with fields:
    name
    billing
    test
```

Working With Arrays Within Cells

Append the parentheses operator to the cell designator `A{1,1}` to select specific elements of a cell. This example displays specific row and columns of the numeric array stored in cell `{1,1}` of `A`:

```
A{1,1}(2,3:end)
ans =
    0.3701   12.3086   14.7641
```

Working With Structures Within Cells

Use a combination of indexing operators to access the components of a structure array that resides in a cell of a cell array. The syntax for indexing into field `F` of a structure array that resides in a cell of array `C` is

```
X = C{CellArrayIndex}(StructArrayIndex).F(FieldArrayIndex);
```

For example, row 1, column 3 of cell array `A` contains a structure array. Use `A{1,3}` to select this cell, and `.name` to display the field name for all elements of the structure array:

```
A{1,3}.name
ans =
    Ann Lane
ans =
    John Doe
```

To display all fields of a particular element of the structure array, type

```
A{1,3}(2)
ans =
       name: 'John Doe'
    billing: 139.7200
       test: [4.6228 17.8260 0.3701 12.3086 14.7641]
```

The `test` field of this structure array contains a 1-by-5 numeric array. Access the odd numbered elements of this field in the second element of the structure array:

```
A{1,3}(2).test(1:2:end)
ans =
    4.6228    0.3701   14.7641
```

Working With Cell Arrays Within Cells

The syntax for indexing into a cell array that resides in a cell of array `C` using content indexing is shown below. To use cell indexing on the inner cell array, replace the curly brace operator enclosing the `InnerCellArrayIndes` with parentheses.

The syntax for content indexing is

```
X = C{OuterCellArrayIndex}{InnerCellArrayIndex}
```

In the example cell array created at the start of this section, `A{2,2}` is a cell array that resides in a cell of the outer array `A`. To get the third row of the inner cell array, type

```
A{2,2}{3,:}
```

```
ans =
     7
ans =
     2
ans =
     9
ans =
     2
```

Note that MATLAB returns a comma-separated list. To have MATLAB return the list of elements as a vector instead, surround the previous expression with square brackets:

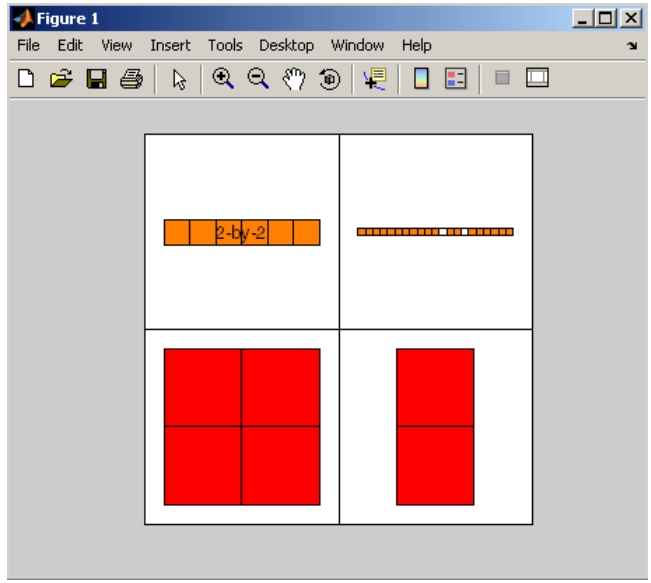
```
[A{2,2}{3,:}]
ans =
     7     2     9     2
```

Plotting the Cell Array

For a high-level graphical display of cell architecture, use the `cellplot` function. Consider a 2-by-2 cell array containing two text strings, a matrix, and a vector:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces this figure:



Deleting Cells

You can delete an entire dimension of cells using a single statement. Like standard array deletion, use vector subscripting when deleting a row or column of cells and assign the empty matrix to the dimension:

```
A(cell_subscripts) = []
```

When deleting cells, curly braces do not appear in the assignment statement at all.

Reshaping Cell Arrays

Like other arrays, you can reshape cell arrays using the `reshape` function. The number of cells must remain the same after reshaping; you cannot use `reshape` to add or remove cells:

```
A = cell(3, 4);

size(A)
ans =
```

```
    3     4

B = reshape(A, 6, 2);

size(B)
ans =
    6     2
```

Replacing Lists of Variables with Cell Arrays

Cell arrays can replace comma-separated lists of MATLAB variables in

- Function input lists
- Function output lists
- Display operations
- Array constructions (square brackets and curly braces)

If you use the colon to index multiple cells in conjunction with the curly brace notation, MATLAB treats the contents of each cell as a separate variable. For example, assume you have a cell array `T` where each cell contains a separate vector. The expression `T{1:5}` is equivalent to a comma-separated list of the vectors in the first five cells of `T`.

Consider the cell array `C`:

```
C(1) = {[1 2 3]};
C(2) = {[1 0 1]};
C(3) = {1:10};
C(4) = {[9 8 7]};
C(5) = {3};
```

To convolve the vectors in `C(1)` and `C(2)` using `conv`,

```
d = conv(C{1:2})
d =
     1     2     4     2     3
```

Display vectors two, three, and four with

```
C{2:4}
ans =
    1   0   1

ans =
    1   2   3   4   5   6   7   8   9   10

ans =
    9   8   7
```

Similarly, you can create a new numeric array using the statement

```
B = [C{1}; C{2}; C{4}]
B =
    1   2   3
    1   0   1
    9   8   7
```

You can also use content indexing on the left side of an assignment to create a new cell array where each cell represents a separate output argument:

```
[D{1:2}] = eig(B)
D =
    [3x3 double]    [3x3 double]
```

You can display the actual eigenvectors and eigenvalues using `D{1}` and `D{2}`.

Note The `varargin` and `varargout` arguments allow you to specify variable numbers of input and output arguments for MATLAB functions that you create. Both `varargin` and `varargout` are cell arrays, allowing them to hold various sizes and kinds of MATLAB data. See "Passing Variable Numbers of Arguments" on page 4-34 in the MATLAB Programming documentation for details.

Applying Functions and Operators

Use indexing to apply functions and operators to the contents of cells. For example, use content indexing to call a function with the contents of a single cell as an argument:

```
A{1,1} = [1 2; 3 4];
A{1,2} = randn(3, 3);
A{1,3} = 1:5;

B = sum(A{1,1})
B =
     4     6
```

To apply a function to several cells of an unnested cell array, use a loop:

```
for k = 1:length(A)
    M{k} = sum(A{1,k});
end
```

Organizing Data in Cell Arrays

Cell arrays are useful for organizing data that consists of different sizes or kinds of data. Cell arrays are better than structures for applications where

- You need to access multiple fields of data with one statement.
- You want to access subsets of the data as comma-separated variable lists.
- You don't have a fixed set of field names.
- You routinely remove fields from the structure.

As an example of accessing multiple fields with one statement, assume that your data consists of

- A 3-by-4 array consisting of measurements taken for an experiment.
- A 15-character string containing a technician's name.
- A 3-by-4-by-5 array containing a record of measurements taken for the past five experiments.

For many applications, the best data construct for this data is a structure. However, if you routinely access only the first two fields of information, then a cell array might be more convenient for indexing purposes.

This example shows how to access the first and second elements of the cell array `TEST`:

```
[newdata,name] = deal(TEST{1:2})
```

This example shows how to access the first and second elements of the structure `TEST`:

```
newdata = TEST.measure
name = TEST.name
```

The `varargin` and `varargout` arguments are examples of the utility of cell arrays as substitutes for comma-separated lists. Create a 3-by-3 numeric array `A`:

```
A = [0 1 2; 4 0 7; 3 1 2];
```

Now apply the `normest` (2-norm estimate) function to `A`, and assign the function output to individual cells of `B`:

```
[B{1:2}] = normest(A)
B =
    [8.8826]    [4]
```

All of the output values from the function are stored in separate cells of `B`. `B(1)` contains the norm estimate; `B(2)` contains the iteration count.

Nesting Cell Arrays

A cell can contain another cell array, or even an array of cell arrays. (Cells that contain noncell data are called *leaf cells*.) You can use nested curly braces, the `cell` function, or direct assignment statements to create nested cell arrays. You can then access and manipulate individual cells, subarrays of cells, or cell elements.

Building Nested Arrays with Nested Curly Braces

You can nest pairs of curly braces to create a nested cell array. For example,

```
clear A
A(1,1) = {magic(5)};

A(1,2) = {{[5 2 8; 7 3 0; 6 7 3] 'Test 1'; [2-4i 5+7i] {17 []}}}
A =
```

```
    [5x5 double]    {2x2 cell}
```

Note that the right side of the assignment is enclosed in two sets of curly braces. The first set represents cell `(1,2)` of cell array `A`. The second "packages" the 2-by-2 cell array inside the outer cell.

Building Nested Arrays with the cell Function

To nest cell arrays with the `cell` function, assign the output of `cell` to an existing cell:

1 Create an empty 1-by-2 cell array.

```
A = cell(1,2);
```

2 Create a 2-by-2 cell array inside `A(1,2)`.

```
A(1,2) = {cell(2,2)};
```

3 Fill `A`, including the nested array, using assignments.

```
A(1,1) = {magic(5)};
A{1,2}(1,1) = {[5 2 8; 7 3 0; 6 7 3]};
A{1,2}(1,2) = {'Test 1'};
A{1,2}(2,1) = {[2-4i 5+7i]};
A{1,2}(2,2) = {cell(1, 2)}
A{1,2}{2,2}(1) = {17};
```

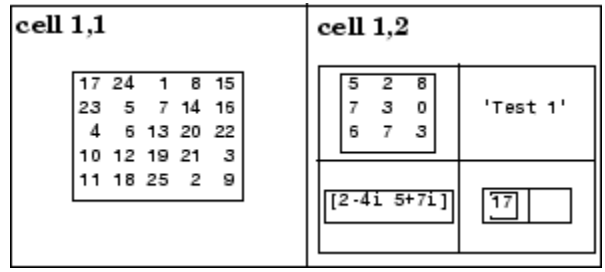
Note the use of curly braces until the final level of nested subscripts. This is required because you need to access cell contents to access cells within cells.

You can also build nested cell arrays with direct assignments using the statements shown in step 3 above.

Indexing Nested Cell Arrays

To index nested cells, concatenate indexing expressions. The first set of subscripts accesses the top layer of cells, and subsequent sets of parentheses access successively deeper layers.

For example, array A has three levels of nesting:



- To access the 5-by-5 array in cell `(1,1)`, use `A{1,1}`.
- To access the 3-by-3 array in position `(1,1)` of cell `(1,2)`, use `A{1,2}{1,1}`.
- To access the 2-by-2 cell array in cell `(1,2)`, use `A{1,2}`.
- To access the empty cell in position `(2,2)` of cell `(1,2)`, use `A{1,2}{2,2}{1,2}`.

Converting Between Cell and Numeric Arrays

Use `for` loops to convert between cell and numeric formats. For example, create a cell array `F`:

```
F{1,1} = [1 2; 3 4];
F{1,2} = [-1 0; 0 1];
F{2,1} = [7 8; 4 1];
F{2,2} = [4i 3+2i; 1-8i 5];
```

Now use three `for` loops to copy the contents of `F` into a numeric array `NUM`:

```
for k = 1:4
   for m = 1:2
      for n = 1:2
         NUM(m,n,k) = F{k}(m,n);
      end
   end
end
```

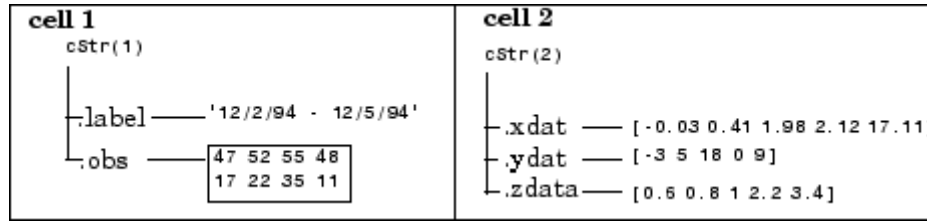
Similarly, you must use `for` loops to assign each value of a numeric array to a single cell of a cell array:

```
G = cell(1,16);
for m = 1:16
   G{m} = NUM(m);
end
```

Cell Arrays of Structures

Use cell arrays to store groups of structures with different field architectures:

```
cStr = cell(1,2);
cStr{1}.label = '12/2/94 - 12/5/94';
cStr{1}.obs = [47 52 55 48; 17 22 35 11];
cStr{2}.xdata = [-0.03 0.41 1.98 2.12 17.11];
cStr{2}.ydata = [-3 5 18 0 9];
cStr{2}.zdata = [0.6 0.8 1 2.2 3.4];
```



Cell 1 of the `cStr` array contains a structure with two fields, one a string and the other a vector. Cell 2 contains a structure with three vector fields.

When building cell arrays of structures, you must use content indexing. Similarly, you must use content indexing to obtain the contents of structures within cells. The syntax for content indexing is

```
cellArray{index}.field
```

For example, to access the `label` field of the structure in cell 1, use `cStr{1}.label`.

Function Summary

This table describes the MATLAB functions for working with cell arrays.

Function	Description
`cell`	Create a cell array.
`cell2struct`	Convert a cell array into a structure array.
`celldisp`	Display cell array contents.
`cellfun`	Apply a cell function to a cell array.
`cellplot`	Display a graphical depiction of a cell array.
`deal`	Copy input to separate outputs.
`iscell`	Return `true` for a cell array.
`num2cell`	Convert a numeric array into a cell array.

Function Handles

In this section...

Overview

A *function handle* is a MATLAB value and data type that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics® callbacks).

Read more about function handles in the section, "Function Handles" on page 4-22.

Constructing and Invoking a Function Handle

You construct a handle for a specific function by preceding the function name with an `@` sign. Use only the function *name* (with no path information) after the `@` sign:

```
fhandle = @functionname
```

Handles to Anonymous Functions

Another way to construct a function handle is to create an anonymous function. For example,

```
sqr = @(x) x.^2;
```

creates an anonymous function that computes the square of its input argument `x`. The variable `sqr` contains a handle to the anonymous function. See "Anonymous Functions" on page 5-3 for more information.

Calling a Function Using Its Handle

To execute a function associated with a function handle, use the syntax shown here, treating the function handle `fhandle` as if it were a function name:

```
fhandle(arg1, arg2, ..., argn)
```

If the function being called takes no input arguments, then use empty parentheses after the function handle name:

```
fhandle()
```

Simple Function Handle Example

The following example calls a function `plotFHandle`, passing it a handle for the MATLAB `sin` function. `plotFHandle` then calls the `plot` function, passing it some data and the function handle to `sin`. The plot function calls the function associated with the handle to compute its y-axis values:

```
function x = plotFHandle(fhandle, data)
plot(data, fhandle(data))
```

Call `plotFhandle` with a handle to the `sin` function and the value shown below:

```
plotFHandle(@sin, -pi:0.01:pi)
```

MATLAB Classes

All MATLAB data types are implemented as object-oriented classes. You can add data types of your own to your MATLAB environment by creating additional classes. These user-defined classes define the structure of your new data type, and the M-file functions, or *methods*, that you write for each class define the behavior for that data type.

These methods can also define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the new data types. For example, a class called `polynomial` might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials.

With MATLAB classes you can

- Create methods that override existing MATLAB functionality
- Restrict the operations that are allowed on an object of a class
- Enforce common behavior among related classes by inheriting from the same parent class
- Significantly increase the reuse of your code

Read more about MATLAB classes in Chapter 9, “Classes and Objects”.

Java Classes

MATLAB provides an interface to the Java programming language that enables you to create objects from Java classes and call Java methods on these objects. A Java class is a MATLAB data type. Native and third-party classes are already available through the MATLAB interface. You can also create your own Java class definitions and bring them into MATLAB.

The MATLAB Java interface enables you to

- Access Java API (application programming interface) class packages that support essential activities such as I/O and networking
- Access third-party Java classes
- Easily construct Java objects in MATLAB
- Call Java object methods, using either Java or MATLAB syntax
- Pass data between MATLAB variables and Java objects

Read more about Java classes in MATLAB in “Calling Java from MATLAB” in the MATLAB External Interfaces documentation.

3

Basic Program Components

Variables

In this section...

Types of Variables

A MATLAB variable is essentially a tag that you assign to a value while that value remains in memory. The tag gives you a way to reference the value in memory so that your programs can read it, operate on it with other data, and save it back to memory.

MATLAB provides three basic types of variables:

- "Local Variables" on page 3-2
- "Global Variables" on page 3-3
- "Persistent Variables" on page 3-5

Local Variables

Each MATLAB function has its own local variables. These are separate from those of other functions (except for nested functions), and from those of the base workspace. Variables defined in a function do not remain in memory from one function call to the next, unless they are defined as `global` or `persistent`.

Scripts, on the other hand, do not have a separate workspace. They store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace.

Note If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

Global Variables

If several functions, and possibly the base workspace, all declare a particular name as `global`, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it global.

Suppose, for example, you want to study the effect of the interaction coefficients, α and β, in the Lotka-Volterra predator-prey model.

$$\dot{y}_1 = y_1 - \alpha y_1 y_2$$

$$\dot{y}_2 = -y_2 + \beta y_1 y_2$$

Create an M-file, `lotka.m`.

```
function yp = lotka(t,y)
%LOTKA   Lotka-Volterra predator-prey model.
global ALPHA BETA
yp = [y(1) - ALPHA*y(1)*y(2); -y(2) + BETA*y(1)*y(2)];
```

Then interactively enter the statements

```
global ALPHA BETA
ALPHA = 0.01
BETA = 0.02
[t,y] = ode23(@lotka,[0,10],[1; 1]);
plot(t,y)
```

The two `global` statements make the values assigned to `ALPHA` and `BETA` at the command prompt available inside the function defined by `lotka.m`. They can be modified interactively and new solutions obtained without editing any files.

Creating Global Variables. Each function that uses a global variable must first declare the variable as `global`. It is usually best to put global declarations toward the beginning of the function. You would declare global variable `MAXLEN` as follows:

```
global MAXLEN
```

If the M-file contains subfunctions as well, then each subfunction requiring access to the global variable must declare it as `global`. To access the variable from the MATLAB command line, you must declare it as `global` at the command line.

MATLAB global variable names are typically longer and more descriptive than local variable names, and often consist of all uppercase characters. These are not requirements, but guidelines to increase the readability of MATLAB code, and to reduce the chance of accidentally redefining a global variable.

Displaying Global Variables. To see only those variables you have declared as global, use the `who` or `whos` functions with the literal, `global`.

```
global MAXLEN MAXWID
MAXLEN = 36;  MAXWID = 78;
len = 5;  wid = 21;

whos global
  Name         Size                Bytes  Class

  MAXLEN       1x1                     8  double array (global)
  MAXWID       1x1                     8  double array (global)

Grand total is 2 elements using 16 bytes
```

Suggestions for Using Global Variables. A certain amount of risk is associated with using global variables and, because of this, it is recommended that you use them sparingly. You might, for example, unintentionally give a global variable in one function a name that is already used for a global variable in another function. When you run your application, one function may overwrite the variable used by the other. This error can be difficult to track down.

Another problem comes when you want to change the variable name. To make a change without introducing an error into the application, you must find every occurrence of that name in your code (and other people's code, if you share functions).

Alternatives to Using Global Variables. Instead of using a global variable, you may be able to

- Pass the variable to other functions as an additional argument. In this way, you make sure that any shared access to the variable is intentional.

  If this means that you have to pass a number of additional variables, you can put them into a structure or cell array and just pass it as one additional argument.
- Use a persistent variable (described in the next section), if you only need to make the variable persist in memory from one function call to the next.

Persistent Variables

Characteristics of persistent variables are

- You can declare and use them within M-file functions only.
- Only the function in which the variables are declared is allowed access to it.
- MATLAB does not clear them from memory when the function exits, so their value is retained from one function call to the next.

You must declare persistent variables before you can use them in a function. It is usually best to put your persistent declarations toward the beginning of the function. You would declare persistent variable `SUM_X` as follows:

```
persistent SUM_X
```

If you clear a function that defines a persistent variable (i.e., using `clear` *`functionname`* or `clear all`), or if you edit the M-file for that function, MATLAB clears all persistent variables used in that function.

You can use the `mlock` function to keep an M-file from being cleared from memory, thus keeping persistent variables in the M-file from being cleared as well.

Initializing Persistent Variables. When you declare a persistent variable, MATLAB initializes its value to an empty matrix, `[]`. After the declaration statement, you can assign your own value to it. This is often done using an `isempty` statement, as shown here:

```
function findSum(inputvalue)
persistent SUM_X

if isempty(SUM_X)
   SUM_X = 0;
end
SUM_X = SUM_X + inputvalue
```

This initializes the variable to 0 the first time you execute the function, and then it accumulates the value on each iteration.

Naming Variables

MATLAB variable names must begin with a letter, which may be followed by any combination of letters, digits, and underscores. MATLAB distinguishes between uppercase and lowercase characters, so `A` and `a` are not the same variable.

Although variable names can be of any length, MATLAB uses only the first `N` characters of the name, (where `N` is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first `N` characters to enable MATLAB to distinguish variables.

```
N = namelengthmax
N =
    63
```

The `genvarname` function can be useful in creating variable names that are both valid and unique. See the `genvarname` reference page to find out how to do this.

Verifying a Variable Name

You can use the `isvarname` function to make sure a name is valid before you use it. `isvarname` returns 1 if the name is valid, and 0 otherwise.

```
isvarname 8th_column
ans =
     0                    % Not valid - begins with a number
```

Avoid Using Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name, either one of your own M-file functions or one of the functions in the MATLAB language. If you define a variable with a function name, you will not be able to call that function until you either remove the variable from memory with the `clear` function, or invoke the function using `builtin`.

For example, if you enter the following command, you will not be able to use the MATLAB `disp` function until you clear the variable with `clear disp`.

```
disp = 50;
```

To test whether a proposed variable name is already used as a function name, use

```
which -all variable_name
```

Potential Conflict with Function Names

There are some MATLAB functions that have names that are commonly used as variable names in programming code. A few examples of such functions are `i`, `j`, `mode`, `char`, `size`, and `path`.

If you need to use a variable that is also the name of a MATLAB function, and have determined that you have no need to call the function, you should be aware that there is still a possibility for conflict. See the following two examples:

- "Variables Loaded From a MAT-File" on page 3-8
- "Variables In Evaluation Statements" on page 3-9

Variables Loaded From a MAT-File. The function shown below loads previously saved data from MAT-file `settings.mat`. It is supposed to display the value of one of the loaded variables, `mode`. However, `mode` is also the name of a MATLAB function and, in this case, MATLAB interprets it as the function and not the variable loaded from the MAT-file:

```
function show_mode
load settings;
whos mode
fprintf('Mode is set to %s\n', mode)
```

Assume that `mode` already exists in the MAT-file. Execution of the function shows that, even though `mode` is successfully loaded into the function workspace as a variable, when MATLAB attempts to operate on it in the last line, it interprets `mode` as a function. This results in an error:

```
show_mode
  Name       Size                     Bytes  Class

  mode       1x6                         12  char array

Grand total is 6 elements using 12 bytes

??? Error using ==> mode
Not enough input arguments.

Error in ==> show_mode at 4
fprintf('Mode is set to %s\n', mode)
```

Because MATLAB parses function M-files before they are run, it needs to determine before runtime which identifiers in the code are variables and which are functions. The function in this example does not establish `mode` as a variable name and, as a result, MATLAB interprets it as a function name instead.

There are several ways to make this function work as intended without having to change the variable name. Both indicate to MATLAB that the name represents a variable, and not a function:

- Name the variable explicitly in the `load` statement:

```
function show_mode
load settings mode;
whos mode
fprintf('Mode is set to %s\n', mode)
```

- Initialize the variable (e.g., set it to an empty matrix or empty string) at the start of the function:

```
function show_mode
mode = '';
load settings;
whos mode
fprintf('Mode is set to %s\n', mode)
```

Variables In Evaluation Statements. Variables used in evaluation statements such as `eval`, `evalc`, and `evalin` can also be mistaken for function names. The following M-file defines a variable named `length` that conflicts with MATLAB `length` function:

```
function find_area
eval('length = 12;  width = 36;');
fprintf('The area is %d\n', length .* width)
```

The second line of this code would seem to establish `length` as a variable name that would be valid when used in the statement on the third line. However, when MATLAB parses this line, it does not consider the *contents* of the string that is to be evaluated. As a result, MATLAB has no way of knowing that `length` was meant to be used as a variable name in this program, and the name defaults to a function name instead, yielding the following error:

```
find_area
??? Error using ==> length
Not enough input arguments.
```

To force MATLAB to interpret `length` as a variable name, use it in an explicit assignment statement first:

```
function find_area
length = [];
eval('length = 12;  width = 36;');
fprintf('The area is %d\n', length .* width)
```

Guidelines to Using Variables

The same guidelines that apply to MATLAB variables at the command line also apply to variables in M-files:

- You do not need to type or declare variables used in M-files (with the possible exception of designating them as `global` or `persistent`).
- Before assigning one variable to another, you must be sure that the variable on the right-hand side of the assignment has a value.
- Any operation that assigns a value to a variable creates the variable, if needed, or overwrites its current value, if it already exists.

Scope of a Variable

MATLAB stores variables in a part of memory called a workspace. The *base workspace* holds variables created during your interactive MATLAB session and also any variables created by running M-file scripts. Variables created at the MATLAB command prompt can also be used by scripts without having to declare them as global.

Functions do not use the base workspace. Every function has its own *function workspace*. Each function workspace is kept separate from the base workspace and all other workspaces to protect the integrity of the data used by that function. Even subfunctions that are defined in the same M-file have a separate function workspace.

Extending Variable Scope

In most cases, variables created within a function are known only within that function. These variables are not available at the MATLAB command prompt or to any other function or subfunction.

Passing Variables from Another Workspace. The most secure way to extend the scope of a function variable is to pass it to other functions as an argument in the function call. Since MATLAB passes data only by value, you also need to add the variable to the return values of any function that modifies its value.

Evaluating in Another Workspace Using evalin. Functions can also obtain variables from either the base or the caller's workspace using the `evalin` function. The example below, `compareAB_1`, evaluates a command in the context of the MATLAB command line, taking the values of variables `A` and `B` from the base workspace.

Define `A` and `B` in the base workspace:

```
A = [13 25 82 68 9 15 77];  B = [63 21 71 42 30 15 22];
```

Use `evalin` to evaluate the command `A(find(A<=B))` in the context of the MATLAB base workspace:

```
function C = compareAB_1
C = evalin('base', 'A(find(A<=B))');
```

Call the function. You do not have to pass the variables because they are made available to the function via the `evalin` function:

```
C = compareAB_1
C =
    13     9    15
```

You can also evaluate in the context of the caller's workspace by specifying `'caller'` (instead of `'base'`) as the first input argument to `evalin`.

Using Global Variables. A third way to extend variable scope is to declare the variable as `global` within every function that needs access to it. If you do this, you need make sure that no functions with access to the variable overwrite its value unintentionally. For this reason, it is recommended that you limit the use of global variables.

Create global vectors `A` and `B` in the base workspace:

```
global A
global B
A = [13 25 82 68 9 15 77];  B = [63 21 71 42 30 15 22];
```

Also declare them in the function to be called:

```
function C = compareAB_2
global A
```

```
global B

C = A(find(A<=B));
```

Call the function. Again, you do not have to pass `A` and `B` as arguments to the called function:

```
C = compareAB_2
C =
    13     9    15
```

Scope in Nested Functions

Variables within nested functions are accessible to more than just their immediate function. As a general rule, the scope of a local variable is the largest containing function body in which the variable appears, and all functions nested within that function. For more information on nested functions, see “Variable Scope in Nested Functions” on page 5-19.

Lifetime of a Variable

Variables created at the MATLAB command prompt or in an M-file script exist until you clear them or end your MATLAB session. Variables in functions exist only until the function completes unless they have been declared as `global` or `persistent`.

Keywords

MATLAB reserves certain words for its own use as keywords of the language. To list the keywords, type

```
iskeyword
ans =
    'break'
    'case'
    'catch'
    'continue'
    'else'
    'elseif'
    'end'
    'for'
    'function'
    'global'
    'if'
    'otherwise'
    'persistent'
    'return'
    'switch'
    'try'
    'while'
```

See the online function reference pages to learn how to use these keywords.

You should not use MATLAB keywords other than for their intended purpose. For example, a keyword should not be used as follows:

```
while = 5;
??? while = 5;
          |
Error: Expected a variable, function, or constant, found "=".
```

Special Values

Several functions return important special values that you can use in your M-files.

Function	Return Value
`ans`	Most recent answer (variable). If you do not assign an output variable to an expression, MATLAB automatically stores the result in `ans`.
`eps`	Floating-point relative accuracy. This is the tolerance MATLAB uses in its calculations.
`intmax`	Largest 8-, 16-, 32-, or 64-bit integer your computer can represent.
`intmin`	Smallest 8-, 16-, 32-, or 64-bit integer your computer can represent.
`realmax`	Largest floating-point number your computer can represent.
`realmin`	Smallest positive floating-point number your computer can represent.
`pi`	`3.1415926535897...`
`i`, `j`	Imaginary unit.
`inf`	Infinity. Calculations like `n/0`, where `n` is any nonzero real value, result in `inf`.
`NaN`	Not a Number, an invalid numeric value. Expressions like `0/0` and `inf/inf` result in a `NaN`, as do arithmetic operations involving a `NaN`. Also, if `n` is complex with a zero real part, then `n/0` returns a value with a `NaN` real part.
`computer`	Computer type.
`version`	MATLAB version string.

Here are some examples that use these values in MATLAB expressions.

```
x = 2 * pi
x =
   6.2832

A = [3+2i 7-8i]
A =
   3.0000 + 2.0000i   7.0000 - 8.0000i

tol = 3 * eps
tol =
   6.6613e-016

intmax('uint64')
ans =
   18446744073709551615
```

Operators

In this section...

Arithmetic Operators

Arithmetic operators perform numeric computations, for example, adding two numbers or raising the elements of an array to a given power. The following table provides a summary. For more information, see the arithmetic operators reference page.

Operator	Description
+	Addition
-	Subtraction
.*	Multiplication
./	Right division
.\	Left division
+	Unary plus
-	Unary minus
:	Colon operator
.^	Power
.'	Transpose
'	Complex conjugate transpose
*	Matrix multiplication
/	Matrix right division

Operator	Description
\	Matrix left division
^	Matrix power

Arithmetic Operators and Arrays

Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand—this property is known as *scalar expansion*.

This example uses scalar expansion to compute the product of a scalar operand and a matrix.

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

3 * A
ans =
    24     3    18
     9    15    21
    12    27     6
```

Relational Operators

Relational operators compare operands quantitatively, using operators like “less than” and “not equal to.” The following table provides a summary. For more information, see the relational operators reference page.

Operator	Description
<	Less than
<=	Less than or equal to

Operator	Description
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Relational Operators and Arrays

The MATLAB relational operators compare corresponding elements of arrays with equal dimensions. Relational operators always operate element-by-element. In this example, the resulting matrix shows where an element of `A` is equal to the corresponding element of `B`.

```
A = [2 7 6;9 0 5;3 0.5 6];
B = [8 7 0;3 2 5;4 -1 7];

A == B
ans =
     0     1     0
     0     0     1
     0     0     0
```

For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand. Locations where the specified relation is true receive logical `1`. Locations where the relation is false receive logical `0`.

Relational Operators and Empty Arrays

The relational operators work with arrays for which any dimension has size zero, as long as both arrays are the same size or one is a scalar. However, expressions such as

```
A == []
```

return an error if A is not 0-by-0 or 1-by-1. This behavior is consistent with that of all other binary operators, such as +, -, >, <, &, |, etc.

To test for empty arrays, use the function

```
isempty(A)
```

Logical Operators

MATLAB offers three types of logical operators and functions:

- Element-wise — operate on corresponding elements of logical arrays.
- Bit-wise — operate on corresponding bits of integer values or arrays.
- Short-circuit — operate on scalar, logical expressions.

The values returned by MATLAB logical operators and functions, with the exception of bit-wise functions, are of type `logical` and are suitable for use with logical indexing.

Element-Wise Operators and Functions

The following logical operators and functions perform elementwise logical operations on their inputs to produce a like-sized output array.

The examples shown in the following table use vector inputs `A` and `B`, where

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

Operator	Description	Example
`&`	Returns `1` for every element location that is `true` (nonzero) in both arrays, and `0` for all other elements.	`A & B = 01001`
`|`	Returns `1` for every element location that is `true` (nonzero) in either one or the other, or both arrays, and `0` for all other elements.	`A | B = 11101`
`~`	Complements each element of the input array, `A`.	`~A = 10010`
`xor`	Returns `1` for every element location that is `true` (nonzero) in only one array, and `0` for all other elements.	`xor(A,B) = 10100`

For operators and functions that take two array operands, (`&`, `|`, and `xor`), both arrays must have equal dimensions, with each dimension being the same size. The one exception to this is where one operand is a scalar and the other is not. In this case, MATLAB tests the scalar against every element of the other operand.

Note MATLAB converts any finite nonzero, numeric values used as inputs to logical expressions to logical `1`, or `true`.

Operator Overloading. You can overload the `&`, `|`, and `~` operators to make their behavior dependent upon the data type on which they are being used. Each of these operators has a representative function that is called whenever that operator is used. These are shown in the table below.

Logical Operation	Equivalent Function
`A & B`	`and(A, B)`
`A | B`	`or(A, B)`
`~A`	`not(A)`

Other Array Functions. Two other MATLAB functions that operate logically on arrays, but not in an elementwise fashion, are `any` and `all`. These functions show whether *any* or *all* elements of a vector, or a vector within a matrix or an array, are nonzero.

When used on a matrix, `any` and `all` operate on the columns of the matrix. When used on an N-dimensional array, they operate on the first nonsingleton dimension of the array. Or, you can specify an additional `dimension` input to operate on a specific dimension of the array.

The examples shown in the following table use array input `A`, where

```
A = [0   1   2;
     0  -3   8;
     0   5   0];
```

Function	Description	Example
`any(A)`	Returns `1` for a vector where *any* element of the vector is `true` (nonzero), and `0` if no elements are `true`.	`any(A) ans = 0 1 1`
`all(A)`	Returns `1` for a vector where *all* elements of the vector are `true` (nonzero), and `0` if all elements are not `true`.	`all(A) ans = 0 1 0`

Note The `all` and `any` functions ignore any `NaN` values in the input arrays.

Short-Circuiting in Elementwise Operators. When used in the context of an `if` or `while` expression, and only in this context, the elementwise `|` and `&` operators use short-circuiting in evaluating their expressions. That is, `A|B` and `A&B` ignore the second operand, `B`, if the first operand, `A`, is sufficient to determine the result.

So, although the statement `1|[]` evaluates to `false`, the same statement evaluates to `true` when used in either an `if` or `while` expression:

```
A = 1;   B = [];
if(A|B) disp 'The statement is true',  end;
   The statement is true
```

while the reverse logical expression, which does not short-circuit, evaluates to `false`

```
if(B|A) disp 'The statement is true',  end;
```

Another example of short-circuiting with elementwise operators shows that a logical expression such as the following, which under most circumstances is invalid due to a size mismatch between `A` and `B`,

```
A = [1 1];   B = [2 0 1];
A|B        % This generates an error.
```

works within the context of an `if` or `while` expression:

```
if (A|B) disp 'The statement is true',  end;
```

```
The statement is true
```

Logical Expressions Using the find Function. The `find` function determines the indices of array elements that meet a given logical condition. The function is useful for creating masks and index matrices. In its most general form, `find` returns a single vector of indices. This vector can be used to index into arrays of any size or shape.

For example,

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

i = find(A > 8);
A(i) = 100
A =
   100    2    3  100
     5  100  100    8
   100    7    6  100
     4  100  100    1
```

Note An alternative to using `find` in this context is to index into the matrix using the logical expression itself. See the example below.

The last two statements of the previous example can be replaced with this one statement:

```
A(A > 8) = 100;
```

You can also use `find` to obtain both the row and column indices of a rectangular matrix for the array values that meet the logical condition:

```
A = magic(4)
A =
    16    2    3   13
```

```
     5    11    10     8
     9     7     6    12
     4    14    15     1

[row, col] = find(A > 12)
row =
     1
     4
     4
     1
col =
     1
     2
     3
     4
```

Bit-Wise Functions

The following functions perform bit-wise logical operations on nonnegative integer inputs. Inputs may be scalar or in arrays. If in arrays, these functions produce a like-sized output array.

The examples shown in the following table use scalar inputs `A` and `B`, where

```
A = 28;              % binary 11100
B = 21;              % binary 10101
```

Function	Description	Example
`bitand`	Returns the bit-wise AND of two nonnegative integer arguments.	`bitand(A,B) = 20 (binary 10100)`
`bitor`	Returns the bit-wise OR of two nonnegative integer arguments.	`bitor(A,B) = 29 (binary 11101)`

Function	Description	Example
`bitcmp`	Returns the bit-wise complement as an `n`-bit number, where `n` is the second input argument to `bitcmp`.	`bitcmp(A,5) = 3 (binary 00011)`
`bitxor`	Returns the bit-wise exclusive OR of two nonnegative integer arguments.	`bitxor(A,B) = 9 (binary 01001)`

Short-Circuit Operators

The following operators perform AND and OR operations on logical expressions containing scalar values. They are *short-circuit* operators in that they evaluate their second operand only when the result is not fully determined by the first operand.

Operator	Description
`&&`	Returns logical `1` (`true`) if both inputs evaluate to `true`, and logical `0` (`false`) if they do not.
`||`	Returns logical `1` (`true`) if either input, or both, evaluate to `true`, and logical `0` (`false`) if they do not.

The statement shown here performs an AND of two logical terms, `A` and `B`:

```
A && B
```

If `A` equals zero, then the entire expression will evaluate to logical `0` (`false`), regardless of the value of `B`. Under these circumstances, there is no need to evaluate `B` because the result is already known. In this case, MATLAB short-circuits the statement by evaluating only the first term.

A similar case is when you OR two terms and the first term is `true`. Again, regardless of the value of `B`, the statement will evaluate to `true`. There is no need to evaluate the second term, and MATLAB does not do so.

Advantage of Short-Circuiting. You can use the short-circuit operators to evaluate an expression only when certain conditions are satisfied. For example, you want to execute an M-file function only if the M-file resides on the current MATLAB path.

Short-circuiting keeps the following code from generating an error when the file, `myfun.m`, cannot be found:

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

Similarly, this statement avoids divide-by-zero errors when `b` equals zero:

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

You can also use the `&&` and `||` operators in `if` and `while` statements to take advantage of their short-circuiting behavior:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

Operator Precedence

You can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence levels determine the order in which MATLAB evaluates an expression. Within each precedence level, operators have equal precedence and are evaluated from left to right. The precedence rules for MATLAB operators are shown in this list, ordered from highest precedence level to lowest precedence level:

1. Parentheses ()
2. Transpose (`.'`), power (`.^`), complex conjugate transpose (’), matrix power (`^`)
3. Unary plus (`+`), unary minus (`-`), logical negation (`~`)
4. Multiplication (`.*`), right division (`./`), left division (`.\`), matrix multiplication (`*`), matrix right division (`/`), matrix left division (`\`)
5. Addition (`+`), subtraction (`-`)
6. Colon operator (`:`)

7. Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)
8. Element-wise AND (&)
9. Element-wise OR (|)
10. Short-circuit AND (&&)
11. Short-circuit OR (||)

Precedence of AND and OR Operators

MATLAB always gives the & operator precedence over the | operator. Although MATLAB typically evaluates expressions from left to right, the expression `a|b&c` is evaluated as `a|(b&c)`. It is a good idea to use parentheses to explicitly specify the intended precedence of statements containing combinations of & and |.

The same precedence rule holds true for the && and || operators.

Overriding Default Precedence

The default precedence can be overridden using parentheses, as shown in this example:

```
A = [3 9 5];
B = [2 1 5];
C = A./B.^2
C =
    0.7500    9.0000    0.2000

C = (A./B).^2
C =
    2.2500   81.0000    1.0000
```

MATLAB Expressions

In this section...

String Evaluation

String evaluation adds power and flexibility to the MATLAB language, letting you perform operations like executing user-supplied strings and constructing executable strings through concatenation of strings stored in variables.

eval

The `eval` function evaluates a string that contains a MATLAB expression, statement, or function call. In its simplest form, the `eval` syntax is

```
eval('string')
```

For example, this code uses `eval` on an expression to generate a Hilbert matrix of order `n`.

```
t = '1/(m + n - 1)';
for m = 1:k
    for n = 1:k
        a(m,n) = eval(t);
    end
end
```

Here is an example that uses `eval` on a statement.

```
eval('t = clock');
```

Constructing Strings for Evaluation. You can concatenate strings to create a complete expression for input to `eval`. This code shows how `eval` can create 10 variables named `P1`, `P2`, `...`, `P10`, and set each of them to a different value.

```
for n = 1:10
    eval(['P', int2str(n), '= n .^ 2'])
```

```
end
```

feval

The `feval` function differs from `eval` in that it executes a function rather than a MATLAB expression. The function to be executed is specified in the first argument by either a function handle or a string containing the function name.

You can use `feval` and the `input` function to choose one of several tasks defined by M-files. This example uses function handles for the `sin`, `cos`, and `log` functions.

```
fun = {@sin; @cos; @log};
k = input('Choose function number: ');
x = input('Enter value: ');
feval(fun{k}, x)
```

Shell Escape Functions

It is sometimes useful to access your own C or Fortran programs using *shell escape functions*. Shell escape functions use the shell escape command `!` to make external stand-alone programs act like new MATLAB functions. A shell escape M-function is an M-file that

1. Saves the appropriate variables on disk.
2. Runs an external program (which reads the data file, processes the data, and writes the results back out to disk).
3. Loads the processed file back into the workspace.

For example, look at the code for `garfield.m`, below. This function uses an external function, `gareqn`, to find the solution to Garfield's equation.

```
function y = garfield(a,b,q,r)
save gardata a b q r
!gareqn
load gardata
```

This M-file

1. Saves the input arguments `a`, `b`, `q`, and `r` to a MAT-file in the workspace using the `save` command.
2. Uses the shell escape operator to access a C or Fortran program called `gareqn` that uses the workspace variables to perform its computation. `gareqn` writes its results to the `gardata` MAT-file.
3. Loads the `gardata` MAT-file described in “Using MAT-Files” to obtain the results.

Regular Expressions

In this section...

Overview

A regular expression is a string of characters that defines a certain pattern. You would normally use a regular expression in searching through text for a group of words that matches this pattern, perhaps while parsing program input, or while processing a block of text.

The string `'Joh?n\w*'` is an example of a regular expression. It defines a pattern that starts with the letters `Jo`, is optionally followed by the letter `h` (indicated by `'h?'`), is then followed by the letter `n`, and ends with any

number of non-whitespace characters (indicated by `'\w*'`). This pattern matches any of the following:

```
Jon, John, Jonathan, Johnny
```

MATLAB supports most of the special characters, or *metacharacters*, commonly used with regular expressions and provides several functions to use in searching and replacing text with these expressions.

MATLAB Regular Expression Functions

Several MATLAB functions support searching and replacing characters using regular expressions:

Function	Description
`regexp`	Match regular expression.
`regexpi`	Match regular expression, ignoring case.
`regexprep`	Replace string using regular expression.
`regexptranslate`	Translate string into regular expression.

See the function reference pages to obtain more information on these functions. For more information on how to use regular expressions in general, consult a reference on that subject.

The `regexp` and `regexpi` functions return up to six outputs in the order shown in the reference page for `regexp`. You can select specific outputs to be returned by using one or more of the following qualifiers with these commands:

Qualifier	Value Returned
`'start'`	Starting index of each substring matching the expression
`'end'`	Ending index of each substring matching the expression
`'tokenExtents'`	Starting and ending indices of each substring matching a token in the expression
`'match'`	Text of each substring matching the expression

Qualifier	Value Returned
`'tokens'`	Text of each token captured
`'names'`	Name and text of each named token captured
`'split'`	Treating each match as a delimiter, the text of each substring between such delimiters.

There is an additional qualifier named `'once'` that you can use to return only the first match found.

Elements of an Expression

Tables and examples in the sections that follow show the metacharacters and syntax supported by the `regexp`, `regexpi`, and `regexprep` functions in MATLAB. Expressions shown in the left column have special meaning and match one or more characters according to the usage described in the right column. Any character not having a special meaning, for example, any alphabetic character, matches that same character literally. To force one of the regular expression functions to interpret a sequence of characters literally (rather than as an operator) use the `regexptranslate` function.

These elements are presented under these categories:

Each table is followed by a set of examples that show how to use the syntax presented in that table.

Character Classes

Character classes represent either a specific set of characters (e.g., uppercase) or a certain type of character (e.g., non-whitespace).

Operator	Usage
`.`	Any single character, including white space
`[c1c2c3]`	Any character contained within the brackets: c_1 or c_2 or c_3
`[^c1c2c3]`	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
`[c1-c2]`	Any character in the range of c_1 through c_2
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`
`\S`	Any non-whitespace character; equivalent to `[^ \f\n\r\t\v]`
`\w`	Any alphabetic, numeric, or underscore character; equivalent to `[a-zA-Z_0-9]`. (This does not apply to non-English character sets).
`\W`	Any character that is not alphabetic, numeric, or underscore; equivalent to `[^a-zA-Z_0-9]`. (True only for English character sets).
`\d`	Any numeric digit; equivalent to `[0-9]`
`\D`	Any nondigit character; equivalent to `[^0-9]`

The following examples demonstrate how to use the character classes listed above. See the `regexp` reference page for help with syntax. Most of these examples use the following string:

```
str = 'The rain in Spain falls mainly on the plain.';
```

Any Character — .

Use `'..ain'` in an expression to match a sequence of five characters ending in `'ain'`. Note that `.` matches white-space characters as well:

```
regexp(str, '..ain')
ans =
     4    13    24    39
```

Matches `' rain'`, `'Spain'`, `' main'`, and `'plain'`.

Returning Strings Rather than Indices. Here is the same example, this time specifying the command qualifier `'match'`. In this case, `regexp` returns the *text* of the matching strings rather than the starting index:

```
regexp(str, '..ain', 'match')
ans =
    ' rain'    'Spain'    ' main'    'plain'
```

Selected Characters — [c1c2c3]

Use `[`$c_1c_2c_3$`]` in an expression to match selected characters `r`, `p`, or `m` followed by `'ain'`. Specify two qualifiers this time, `'match'` and `'start'`, along with an output argument for each, `mat` and `idx`. This returns the matching strings and the starting indices of those strings:

```
[mat idx] = regexp(str, '[rpm]ain', 'match', 'start')
mat =
    'rain'    'pain'    'main'
idx =
     5    14    25
```

Range of Characters — [c1 - c2]

Use `[c1-c2]` in an expression to find words that begin with a letter in the range of `A` through `Z`:

```
[mat idx] = regexp(str, '[A-Z]\w*', 'match', 'start')
mat =
    'The'    'Spain'
idx =
     1    13
```

Word and White-Space Characters — \w, \s

Use `\w` and `\s` in an expression to find words that end with the letter `n` followed by a white-space character. Add a new qualifier, `'end'`, to return the `str` index that marks the end of each match:

```
[mat ix1 ix2] = regexp(str, '\w*n\s', 'match', 'start', 'end')
mat =
    'rain '    'in '    'Spain '    'on '
ix1 =
     5    10    13    32
ix2 =
     9    12    18    34
```

Numeric Digits — \d

Use `\d` to find numeric digits in the following string:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\d', 'match', 'start')
mat =
    '1'    '2'    '3'
idx =
     9    12    15
```

Character Representation

The following character combinations represent specific character and numeric values.

Operator	Usage
`\a`	Alarm (beep)
`\\`	Backslash
`\$`	Dollar sign
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\oN` or `\o{N}`	Character of octal value `N`
`\xN` or `\x{N}`	Character of hexadecimal value `N`
`\char`	If a character has special meaning in a regular expression, precede it with backslash (`\`) to match it literally.

Octal and Hexadecimal — \o, \x

Use `\x` and `\o` in an expression to find a comma (hex `2C`) followed by a space (octal `40`) followed by the character `2`:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\x2C\o{40}2', 'match', 'start')
mat =
    ', 2'
idx =
    10
```

Grouping Operators

When you need to use one of the regular expression operators on a number of consecutive elements in an expression, group these elements together with one of the grouping operators and apply the operation to the entire group. For example, this command matches a capital letter followed by a numeral and then an optional space character. These elements have to occur at least two times in succession for there to be a match. To apply the `{2,}` multiplier to all three consecutive characters, you can first make a group of the characters and then apply the `(?:)` quantifier to this group:

```
regexp('B5 A2 6F 63 R6 P4 B2 BC', '(?:[A-Z]\d\s?){2,}', 'match')
ans =
    'B5 A2 '    'R6 P4 B2 '
```

There are three types of explicit grouping operators that you can use when you need to apply an operation to more than just one element in an expression. Also in the grouping category is the alternative match (logical OR) operator, `|`. This creates two or more groups of elements in the expression and applies an operation to one of the groups.

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.
`(?>expr)`	Group atomically.
$\texttt{expr}_1\texttt{|expr}_2$	Match expression $\texttt{expr}_1$ or expression $\texttt{expr}_2$.

Grouping and Capture — (expr)

When you enclose an expression in parentheses, MATLAB not only treats all of the enclosed elements as a group, but also captures a token from these elements whenever a match with the input string is found. For an example of how to use this, see “Using Tokens — Example 1” on page 3-50.

Grouping Only — (?:expr)

Use `(?:expr)` to group a nonvowel (consonant, numeric, whitespace, punctuation, etc.) followed by a vowel in the palindrome `pstr`. Specify at least

two consecutive occurrences (`{2,}`) of this group. Return the starting and ending indices of the matched substrings:

```
pstr = 'Marge lets Norah see Sharon''s telegram';
expr = '(?:[^aeiou][aeiou]){2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'Nora'    'haro'    'tele'
ix1 =
    12    23    31
ix2 =
    15    26    34
```

Remove the grouping, and the `{2,}` now applies only to `[aeiou]`. The command is entirely different now as it looks for a nonvowel followed by at least two consecutive vowels:

```
expr = '[^aeiou][aeiou]{2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'see'
ix1 =
    18
ix2 =
    20
```

Alternative Match — expr1|expr2

Use $p_1|p_2$ to pick out words in the string that start with `let` or `tel`:

```
regexpi(pstr, '(let|tel)\w+', 'match')
ans =
    'lets'    'telegram'
```

Nonmatching Operators

The `comment` operator enables you to insert comments into your code to make it more maintainable. The text of the comment is ignored by MATLAB when matching against the input string.

Operator	Usage
`(?#comment)`	Insert a comment into the expression. Comments are ignored in matching.

Including Comments — (?#expr)

Use `(?#expr)` to add a comment to this expression that matches capitalized words in `pstr`. Comments are ignored in the process of finding a match:

```
regexp(pstr, '(?# Match words in caps)[A-Z]\w+', 'match')
ans =
    'Marge'    'Norah'    'Sharon'
```

Positional Operators

Positional operators in an expression match parts of the input string not by content, but by where they occur in the string (e.g., the first N characters in the string).

Operator	Usage
`^expr`	Match `expr` if it occurs at the beginning of the input string.
`expr$`	Match `expr` if it occurs at the end of the input string.
`\<expr`	Match `expr` when it occurs at the beginning of a word.
`expr\>`	Match `expr` when it occurs at the end of a word.
`\<expr\>`	Match `expr` when it represents the entire word.

Start and End of String Match — ^expr, expr$

Use `^expr` to match words starting with the letter `m` or `M` only when it begins the string, and `expr$` to match words ending with `m` or `M` only when it ends the string:

```
regexpi(pstr, '^m\w*|\w*m$', 'match')
ans =
    'Marge'    'telegram'
```

Start and End of Word Match — \<expr, expr\>

Use `\<expr` to match any words starting with `n` or `N`, or ending with `e` or `E`:

```
regexpi(pstr, '\<n\w*|\w*e\>', 'match')
ans =
    'Marge'    'Norah'    'see'
```

Exact Word Match — \<expr\>

Use `\<expr\>` to match a word starting with an n or N and ending with an `h` or `H`:

```
regexpi(pstr, '\<n\w*h\>', 'match')
ans =
    'Norah'
```

Lookaround Operators

Lookaround operators tell MATLAB to look either ahead or behind the current location in the string for a specified expression. If the expression is found, MATLAB attempts to match a given pattern.

This table shows the four lookaround expressions: lookahead, negative lookahead, lookbehind, and negative lookbehind.

Operator	Usage
`(?=expr)`	Look ahead from current position and test if `expr` is found.
`(?!expr)`	Look ahead from current position and test if `expr` is not found

Operator	Usage
`(?<=expr)`	Look behind from current position and test if `expr` is found.
`(?<!expr)`	Look behind from current position and test if `expr` is not found.

Lookaround operators do not change the current parsing location in the input string. They are more of a condition that must be satisfied for a match to occur.

For example, the following command uses an expression that matches alphabetic, numeric, or underscore characters (`\w*`) that meet the condition that they *look ahead to* (i.e., are immediately followed by) the letters `vision`. The resulting match includes only that part of the string that matches the `\w*` operator; it does not include those characters that match the lookahead expression `(?=vision)`:

```
[s e] = regexp('telegraph television telephone', ...
               '\w*(?=vision)', 'start', 'end')
s =
    11
e =
    14
```

If you repeat this command and match one character beyond the lookahead expression, you can see that parsing of the input string resumes at the letter `v`, thus demonstrating that matching the lookahead operator has not consumed any characters in the string:

```
regexp('telegraph television telephone', ...
       '\w*(?=vision).', 'match')
ans =
    'telev'
```

Note You can also use lookaround operators to perform a logical AND of two elements. See “Using Lookaround as a Logical Operator” on page 3-44.

Lookahead — expr(?=test)

Look ahead to the location of each of these national parks to identify those situated in Tanzania:

```
AfricanParks = {'Arusha, Tanzania', 'Limpopo, Mozambique', ...
   'Chobe, Botswana', 'Amboseli, Kenya', 'Mikumi, Tanzania', ...
   'Kabelaga, Uganda', 'Gonarezhou, Zimbabwe', ...
   'Uangudi, Ethiopia', 'Akagera, Rwanda', ...
   'Etosha, Namibia', 'Kilimanjaro, Tanzania', ...
   'Kasanga, Zambia', 'Udzungwa, Tanzania', 'Omo, Ethiopia'};

T = regexp(AfricanParks, '.*(?=, Tanzania)', 'match');
```

The result `T` is a cell array of empty and full character strings. Display the results:

```
for k=1:numel(AfricanParks)
   if k==1, disp 'Parks in Tanzania:', end
   if ~isempty(T{k})
      fprintf('   %s\n', char(T{k}))
   end
end
```

```
Parks in Tanzania:
   Arusha
   Mikumi
   Kilimanjaro
   Udzungwa
```

Negative Lookahead — expr(?!test)

Generate a series of sequential numbers:

```
n = num2str(5:15)
n =
   5   6   7   8   9  10  11  12  13  14  15
```

Use both the negative lookbehind and negative lookahead operators together to precede only the single-digit numbers with zero:

```
regexprep(n, '(?<!\d)(\d)(?!\d)', '0$1')
```

```
ans =
   05   06   07   08   09   10   11   12   13   14   15
```

Lookbehind — (?<=test)expr

This example uses the lookbehind operator to extract different types of currency from a string. Start by identifying the euro, British pound, Japanese yen, and American dollar symbols using their hexadecimal Unicode values:

```
euro = char(hex2dec('20AC'))
euro =
   €

pound = char(hex2dec('00A3'))
pound =
   £

yen = char(hex2dec('00A5'))
yen =
   ¥

dollar = char(hex2dec('0024'))
dollar =
   $
```

Compose a string of monetary values:

```
str = [euro '10.50 ' pound '6.94 ' yen '1649.40 ' dollar ...
      '13.67']
str =
   €10.50 £6.94 ¥1649.40 $13.67
```

Using `regexp`, match numeric and decimal point characters, but only if you can look behind and find the desired currency symbol immediately preceding those characters:

```
regexp(str, '(?<=\x{20AC})[\d\.]+', 'match')
ans =
    '10.50'
```

```
regexp(str, '(?<=\x{00A3})[\d\.]+', 'match')
ans =
    '6.94'

regexp(str, '(?<=\x{00A5})[\d\.]+', 'match')
ans =
    '1649.40'

regexp(str, '(?<=\x{0024})[\d\.]+', 'match')
ans =
    '13.67'
```

Negative Lookbehind — (?<!test)expr

Use `(?<!test)expr` to find all words that do not follow a comma and zero or more spaces:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

[mat idx] = regexp(poestr, '(?<!,\s*\w*)\w*', 'match', 'start')
mat =
   'While' 'I' 'nodded' 'napping' 'there' 'came' 'a' 'tapping'
idx =
     1     7     9    24    42    48    53    55
```

Using Lookaround as a Logical Operator

You can use lookaround operators to perform a logical AND, as shown in this example. The expression used here finds all words that contain a sequence of two letters under the condition that the two letters are identical *and* are in the range a through m. (The expression `'(?=[a-m])'` is a lookahead test for the range a through m, and the expression `'(.)\1'` tests for identical characters using a token):

```
[mat idx] = regexp(poestr, '\<\w*(?=[a-m])(.)\1\w*\>', ...
   'match', 'start')
mat =
    'nodded'    'suddenly'
idx =
```

```
     9    33
```

Note that when using a lookahead operator to perform an AND, you need to place the match expression `expr` *after* the test expression `test`:

```
(?=test)expr or (?!test)expr
```

Quantifiers

With the quantifiers shown below, you can specify how many instances of an element are to be matched. The basic quantifying operators are listed in the first six rows of the table.

By default, MATLAB matches as much of an expression as possible. Using the operators shown in the last two rows of the table, you can override this default behavior. Specify these options by appending a `+` or `?` immediately following one of the six basic quantifying operators.

Operator	Usage
`expr{m,n}`	Must occur at least `m` times but no more than `n` times.
`expr{m,}`	Must occur at least `m` times.
`expr{n}`	Must match exactly `n` times. Equivalent to `{n,n}`.
`expr?`	Match the preceding element 0 times or 1 time. Equivalent to `{0,1}`.
`expr*`	Match the preceding element 0 or more times. Equivalent to `{0,}`.
`expr+`	Match the preceding element 1 or more times. Equivalent to `{1,}`.
`q_expr+`	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails. The term `q_expr` represents any of the expressions shown in the top six rows of this table.
`q_expr?`	Match only as much of the quantified expression as necessary. The term `q_expr` represents any of the expressions shown in the top six rows of this table. For an example, see “Lazy Quantifiers — expr*?” on page 3-47, below.

Zero or One — expr?

Use `?` to make the HTML `<code>` and `</code>` tags optional in the string. The first string, `hstr1`, contains one occurrence of each tag. Since the expression uses `()?` around the tags, one occurrence is a match:

```
hstr1 = '<td><a name="18854"></a><code>%%</code><br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr1, expr, 'match')
ans =
    '</a><code>%%</code><br>'
```

The second string, `hstr2`, does not contain the code tags at all. Just the same, the expression matches because `()?` allows for zero occurrences of the tags:

```
hstr2 = '<td><a name="18854"></a>%%<br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr2, expr, 'match')
ans =
    '</a>%%<br>'
```

Zero or More — expr*

The first `regexp` command looks for at least one occurrence of `<br>` and finds it. The second command parses a different string for at least one `<br>` and fails. The third command uses `*` to parse the same line for zero or more line breaks and this time succeeds.

```
hstr1 = '<p>This string has <br><br>line breaks</p>';
regexp(hstr1, '<p>.*(<br>).*</p>', 'match')
ans =
    '<p>This string has <br><br>line breaks</p>';

hstr2 = '<p>This string has no line breaks</p>';
regexp(hstr2, '<p>.*(<br>).*</p>', 'match')
ans =
     {}

regexp(hstr2, '<p>.*(<br>)*.*</p>', 'match')
```

```
ans =
    '<p>This string has no line breaks</p>';
```

One or More — expr+

Use + to verify that the HTML image source is not empty. This looks for one or more characters in the `gif` filename:

```
hstr = '<a href="s12.html"><img src="b_prev.gif" border=0>';
expr = '<img src="\w+.gif';

regexp(hstr, expr, 'match')
ans =
    '<img src="b_prev.gif'
```

Exact, Minimum, and Maximum Quantities — {min,max}

Use `{m}`, `{m,}`, and `{m,n}` to verify the `href` syntax used in HTML. This statement requires the `href` to have at least one non-whitespace character, followed by exactly one occurrence of `.html`, optionally followed by # and five to eight digits:

```
hstr = '<a name="18749"></a><a href="s13.html#18760">';
expr = '<a href="\w{1,}(\.html){1}(\#\d{5,8}){0,1}"';

regexp(hstr, expr, 'match')
ans =
    '<a href="s13.html#18760"'
```

Lazy Quantifiers — expr*?

This example shows the difference between the default (*greedy*) quantifier and the *lazy* quantifier (`?`). The first part of the example uses the default quantifier to match all characters from the opening `<tr` to the ending `</td`:

```
hstr = '<tr valign=top><td><a name="19184"></a><br></td>';
regexp(hstr, '</?t.*>', 'match')
ans =
```

```
    '<tr valign=top><td><a name="19184"></a><br></td>'
```

The second part uses the lazy quantifier to match the minimum number of characters between `<tr`, `<td`, or `</td` tags:

```
regexp(hstr, '</?t.*?>', 'match')
ans =
    '<tr valign=top>'    '<td>'    '</td>'
```

Tokens

Parentheses used in a regular expression not only group elements of that expression together, but also designate any matches found for that group as *tokens*. You can use tokens to match other parts of the same string. One advantage of using tokens is that they remember what they matched, so you can recall and reuse matched text in the process of searching or replacing.

This section covers

Operators Used with Tokens

Here are the operators you can use with tokens in MATLAB.

Operator	Usage
`(expr)`	Capture in a token all characters matched by the expression within the parentheses.
`\N`	Match the N^{th} token generated by this command. That is, use `\1` to match the first token, `\2` to match the second, and so on.

Operator	Usage
`$N`	Insert the match for the N^{th} token in the replacement string. Used only by the `regexprep` function. If `N` is equal to zero, then insert the entire match in the replacement string.
`(?(N)s1|s2)`	If N^{th} token is found, then match `s1`, else match `s2`

Introduction to Using Tokens

You can turn any pattern being matched into a token by enclosing the pattern in parentheses within the expression. For example, to create a token for a dollar amount, you could use '`(\$\d+)`'. Each token in the expression is assigned a number, starting from 1, going from left to right. To make a reference to a token later in the expression, refer to it using a backslash followed by the token number. For example, when referencing a token generated by the third set of parentheses in the expression, use `\3`.

As a simple example, if you wanted to search for identical sequential letters in a string, you could capture the first letter as a token and then search for a matching character immediately afterwards. In the expression shown below, the `(\S)` phrase creates a token whenever `regexp` matches any non-whitespace character in the string. The second part of the expression, `'\1'`, looks for a second instance of the same character immediately following the first:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

[mat tok ext] = regexp(poestr, '(\S)\1', 'match', ...
   'tokens', 'tokenExtents');
mat
mat =
    'dd'    'pp'    'dd'    'pp'
```

The tokens returned in cell array `tok` are:

```
'd', 'p', 'd', 'p'
```

Starting and ending indices for each token in the input string `poestr` are:

```
11 11,  26 26,  35 35,  57 57
```

Using Tokens — Example 1

Here is an example of how tokens are assigned values. Suppose that you are going to search the following text:

```
andy ted bob jim andrew andy ted mark
```

You choose to search the above text with the following search pattern:

```
and(y|rew)|(t)e(d)
```

This pattern has three parenthetical expressions that generate tokens. When you finally perform the search, the following tokens are generated for each match.

Match	Token 1	Token 2
andy	y	
ted	t	d
andrew	rew	
andy	y	
ted	t	d

Only the highest level parentheses are used. For example, if the search pattern `and(y|rew)` finds the text `andrew`, token 1 is assigned the value `rew`. However, if the search pattern `(and(y|rew))` is used, token 1 is assigned the value `andrew`.

Using Tokens — Example 2

Use `(expr)` and `\N` to capture pairs of matching HTML tags (e.g., `<a>` and `<\a>`) and the text between them. The expression used for this example is

```
expr = '<(\w+).*?>.*?</\1>';
```

The first part of the expression, '<(\w+)', matches an opening bracket (<) followed by one or more alphabetic, numeric, or underscore characters. The enclosing parentheses capture token characters following the opening bracket.

The second part of the expression, '.*?>.*?', matches the remainder of this HTML tag (characters up to the >), and any characters that may precede the next opening bracket.

The last part, `'</\1>'`, matches all characters in the ending HTML tag. This tag is composed of the sequence `</tag>`, where `tag` is whatever characters were captured as a token.

```
hstr = '<!comment><a name="752507"></a><b>Default</b><br>';
expr = '<(\w+).*?>.*?</\1>';

[mat tok] = regexp(hstr, expr, 'match', 'tokens');
mat{:}
ans =
    <a name="752507"></a>
ans =
    <b>Default</b>

tok{:}
ans =
    'a'
ans =
    'b'
```

Tokens That Are Not Matched

For those tokens specified in the regular expression that have no match in the string being evaluated, `regexp` and `regexpi` return an empty string (`''`) as the token output, and an extent that marks the position in the string where the token was expected.

The example shown here executes `regexp` on the path string `str` returned from the MATLAB `tempdir` function. The regular expression `expr` includes six token specifiers, one for each piece of the path string. The third specifier

`[a-z]+` has no match in the string because this part of the path, `Profiles`, begins with an uppercase letter:

```
str = tempdir
str =
   C:\WINNT\Profiles\bpascal\LOCALS~1\Temp\

expr = ['([A-Z]:)\\(WINNT)\\([a-z]+)?.*\\' ...
        '([a-z]+)\\([A-Z]+~\d)\\(Temp)\\'];

[tok ext] = regexp(str, expr, 'tokens', 'tokenExtents');
```

When a token is not found in a string, MATLAB still returns a token string and token extent. The returned token string is an empty character string (`''`). The first number of the extent is the string index that marks where the token was expected, and the second number of the extent is equal to one less than the first.

In the case of this example, the empty token is the third specified in the expression, so the third token string returned is empty:

```
tok{:}
ans =
    'C:'    'WINNT'     ''    'bpascal'    'LOCALS~1'    'Temp'
```

The third token extent returned in the variable `ext` has the starting index set to 10, which is where the nonmatching substring, `Profiles`, begins in the string. The ending extent index is set to one less than the starting index, or 9:

```
ext{:}
ans =
     1     2
     4     8
    10     9
    19    25
    27    34
    36    39
```

Using Tokens in a Replacement String

When using tokens in a replacement string, reference them using `$1`, `$2`, etc. instead of `\1`, `\2`, etc. This example captures two tokens and reverses their order. The first, `$1`, is `'Norma Jean'` and the second, `$2`, is `'Baker'`. Note that `regexprep` returns the modified string, not a vector of starting indices.

```
regexprep('Norma Jean Baker', '(\w+\s\w+)\s(\w+)', '$2, $1')
ans =
    Baker, Norma Jean
```

Named Capture

If you use a lot of tokens in your expressions, it may be helpful to assign them names rather than having to keep track of which token number is assigned to which token. Use the following operator to assign a name to a token that finds a match.

Operator	Usage
`(?<name>expr)`	Capture in a token all characters matched by the expression within the parentheses. Assign a `name` to the token.
`\k<name>`	Match the token referred to by `name`.
`$<name>`	Insert the match for named token in a replacement string. Used only with the `regexprep` function.
`(?(name)s1|s2)`	If named token is found, then match `s1`; otherwise, match `s2`

When referencing a named token within the expression, use the syntax `\k<name>` instead of the numeric `\1`, `\2`, etc.:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

regexp(poestr, '(?<anychar>.)\k<anychar>', 'match')
ans =
    'dd'    'pp'    'dd'    'pp'
```

Labeling Your Output

Named tokens can also be useful in labeling the output from the MATLAB regular expression functions. This is especially true when you are processing numerous strings.

This example parses different pieces of street addresses from several strings. A short name is assigned to each token in the expression string:

```
str1 = '134 Main Street, Boulder, CO, 14923';
str2 = '26 Walnut Road, Topeka, KA, 25384';
str3 = '847 Industrial Drive, Elizabeth, NJ, 73548';

p1 = '(?<adrs>\d+\s\S+\s(Road|Street|Avenue|Drive))';
p2 = '(?<city>[A-Z][a-z]+)';
p3 = '(?<state>[A-Z]{2})';
p4 = '(?<zip>\d{5})';

expr = [p1 ', ' p2 ', ' p3 ', ' p4];
```

As the following results demonstrate, you can make your output easier to work with by using named tokens:

```
loc1 = regexp(str1, expr, 'names')
loc1 =
     adrs: '134 Main Street'
     city: 'Boulder'
    state: 'CO'
      zip: '14923'

loc2 = regexp(str2, expr, 'names')
loc2 =
     adrs: '26 Walnut Road'
     city: 'Topeka'
    state: 'KA'
      zip: '25384'

loc3 = regexp(str3, expr, 'names')
loc3 =
     adrs: '847 Industrial Drive'
     city: 'Elizabeth'
```

```
    state: 'NJ'
      zip: '73548'
```

Conditional Expressions

With conditional expressions, you can tell MATLAB to match an expression only if a certain condition is true. A conditional expression is similar to an `if-then` or an `if-then-else` clause in programming. MATLAB first tests the state of a given condition, and the outcome of this tests determines what, if anything, is to be matched next. The following table shows the two conditional syntaxes you can use with MATLAB.

Operator	Usage
`(?(cond)expr)`	If condition `cond` is true, then match expression `expr`
`(?(cond)`$\texttt{expr}_1$`|`$\texttt{expr}_2$`)`	If condition `cond` is true, then match expression $\texttt{expr}_1$. Otherwise match expression $\texttt{expr}_2$

The first entry in this table is the same as an `if-then` statement. MATLAB tests the state of condition `cond` and then matches expression `expr` only if the condition was found to be true. In the form of an `if-then` statement, it would look like this:

```
if cond then expr
```

The second entry in the table is the same as an `if-then-else` statement. If the condition is true, MATLAB matches $\texttt{expr}_1$; if false, it matches $\texttt{expr}_2$ instead. This syntax is equivalent to the following programming statement:

```
if cond then expr1 else expr2
```

The condition `cond` in either of these syntaxes can be any one of the following:

- A specific token, identified by either number or name, is located in the input string. See “Conditions Based on Tokens” on page 3-56, below.
- A lookaround operation results in a match. See “Conditions Based on a Lookaround Match” on page 3-57, below.
- A dynamic expression of the form `(?@cmd)` returns a nonzero numeric value. See “Conditions Based on Return Values” on page 3-57, below.

Conditions Based on Tokens

In a conditional expression, MATLAB matches the expression only if the condition associated with it is met. If the condition is based on a token, then the condition is met if MATLAB matches more than one character for the token in the input string.

To specify a token in a condition, use either the token number or, for tokens that you have assigned a name to, its name. Token numbers are determined by the order in which they appear in an expression. For example, if you specify three tokens in an expression (that is, if you enclose three parts of the expression in parentheses), then you would refer to these tokens in a condition statement as `1`, `2`, and `3`.

The following example uses the conditional statement `(?(1)her|his)` to match the string regardless of the gender used. You could translate this into the phrase, "**if** token `1` is found (i.e., `Mr` is followed by the letter `s`), **then** match `her`, **else** match `his`."

```
expr = 'Mr(s?)\..*?(?(1)her|his) son';

[mat tok] = regexp('Mr. Clark went to see his son', ...
   expr, 'match', 'tokens')
mat =
    'Mr. Clark went to see his son'
tok =
    {1x2 cell}

tok{:}
ans =
     ''    'his'
```

In the second part of the example, the token `s` is found and MATLAB matches the word `her`:

```
[mat tok] = regexp('Mrs. Clark went to see her son', ...
expr, 'match', 'tokens')
mat =
    'Mrs. Clark went to see her son'
tok =
    {1x2 cell}
```

```
tok{:}
ans =
    's'    'her'
```

Note When referring to a token within a condition, use just the number of the token. For example, refer to token `2` by using the number `2` alone, and not `\2` or `$2`.

Conditions Based on a Lookaround Match

Lookaround statements look for text that either precedes or follows an expression. If this lookaround text is located, then MATLAB proceeds to match the expression. You can also use lookarounds in conditional statements. In this case, if the lookaround text is located, then MATLAB considers the condition to be met and matches the associated expression. If the condition is not met, then MATLAB matches the `else` part of the expression.

Conditions Based on Return Values

MATLAB supports different types of dynamic expressions. One type of dynamic expression, having the form `(?@cmd)`, enables you to execute a MATLAB command (shown here as `cmd`) while matching an expression. You can use this type of dynamic expression in a conditional statement if the command in the expression returns a numeric value. The condition is considered to be met if the return value is nonzero.

Dynamic Regular Expressions

In a dynamic expression, you can make the pattern that you want `regexp` to match dependent on the content of the input string. In this way, you can more closely match varying input patterns in the string being parsed. You can also use dynamic expressions in replacement strings for use with the `regexprep` function. This gives you the ability to adapt the replacement text to the parsed input.

You can include any number of dynamic expressions in the `match_expr` or `replace_expr` arguments of these commands:

```
regexp(string, match_expr)
regexpi(string, match_expr)
regexprep(string, match_expr, replace_expr)
```

MATLAB supports three types of dynamic operators for use in a match expression. See “Dynamic Operators for the Match Expression” on page 3-59 for more information.

Operator	Usage
`(??expr)`	Parse `expr` as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called `regexprep` inside of a `regexp` match expression.
`(?@cmd)`	Execute the MATLAB command `cmd`, discarding any output that may be returned. This is often used for diagnosing a regular expression.
`(??@cmd)`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the match expression. This is a combination of the two dynamic syntaxes shown above: `(??expr)` and `(?@cmd)`.

MATLAB supports one type of dynamic expression for use in the replacement expression of a `regexprep` command. See “Dynamic Operators for the Replacement Expression” on page 3-64 for more information.

Operator	Usage
`${cmd}`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the replacement expression.

Example of a Dynamic Expression

As an example of a dynamic expression, the following `regexprep` command correctly replaces the term `internationalization` with its abbreviated form, `i18n`. However, to use it on a different term such as `globalization`, you have to use a different replacement expression:

```
match_expr = '(^\w)(\w*)(\w$)';
```

```
replace_expr1 = '$118$3';
regexprep('internationalization', match_expr, replace_expr1)
ans =
    i18n

replace_expr2 = '$111$3';
regexprep('globalization', match_expr, replace_expr2)
ans =
    g11n
```

Using a dynamic expression `${num2str(length($2))}` enables you to base the replacement expression on the input string so that you do not have to change the expression each time. This example uses the dynamic syntax `${cmd}` from the second table shown above:

```
match_expr = '(^\w)(\w*)(\w$)';
replace_expr = '$1${num2str(length($2))}$3';

regexprep('internationalization', match_expr, replace_expr)
ans =
    i18n

regexprep('globalization', match_expr, replace_expr)
ans =
    g11n
```

Dynamic Operators for the Match Expression

There are three types of dynamic expressions you can use when composing a match expression:

- "Dynamic Expressions that Modify the Match Expression — (??expr)" on page 3-60

  "Dynamic Commands that Modify the Match Expression — (??@cmd)" on page 3-61

  "Dynamic Commands that Serve a Functional Purpose — (?@cmd)" on page 3-62

The first two of these actually modify the match expression itself so that it can be made specific to changes in the contents of the input string. When MATLAB

evaluates one of these dynamic statements, the results of that evaluation are included in the same location within the overall match expression.

The third operator listed here does not modify the overall expression, but instead enables you to run MATLAB commands during the parsing of a regular expression. This functionality can be useful in diagnosing your regular expressions.

Dynamic Expressions that Modify the Match Expression — (??expr). The `(??expr)` operator parses expression `expr`, and inserts the results back into the match expression. MATLAB then evaluates the modified match expression.

Here is an example of the type of expression that you can use with this operator:

```
str = {'5XXXXX', '8XXXXXXXX', '1X'};
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
```

The purpose of this particular command is to locate a series of `X` characters in each of the strings stored in the input cell array. Note however that the number of `X`s varies in each string. If the count did not vary, you could use the expression `X{n}` to indicate that you want to match `n` of these characters. But, a constant value of `n` does not work in this case.

The solution used here is to capture the leading count number (e.g., the `5` in the first string of the cell array) in a token, and then to use that count in a dynamic expression. The dynamic expression in this example is `(??X{$1})`, where `$1` is the value captured by the token `\d+`. The metacharacter `{$1}` makes a quantifier of that token value. Because the expression is dynamic, the same pattern works on all three of the input strings in the cell array. With the first input string, `regexp` looks for five `X` characters; with the second, it looks for eight, and with the third, it looks for just one:

```
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
ans =
    '5XXXXX'    '8XXXXXXXX'    '1X'
```

Dynamic Commands that Modify the Match Expression — (??@cmd). MATLAB uses the `(??@function)` operator to include the results of a MATLAB command in the match expression. This command must return a string that can be used within the match expression.

The `regexp` command below uses the dynamic expression `(??@flilplr($1))` to locate a palindrome string, "Never Odd or Even", that has been embedded into a larger string:

```
regexp(pstr, '(.{3,}).?(??@fliplr($1))', 'match')
```

The dynamic expression reverses the order of the letters that make up the string, and then attempts to match as much of the reversed-order string as possible. This requires a dynamic expression because the value for `$1` relies on the value of the token `(.{3,})`:

```
% Put the string in lowercase.
str = lower(...
  'Find the palindrome Never Odd or Even in this string');

% Remove all nonword characters.
str = regexprep(str, '\W*', '')
str =
   findthepalindromeneveroddoreveninthisstring

% Now locate the palindrome within the string.
palstr = regexp(str, '(.{3,}).?(??@fliplr($1))', 'match')
str =
   'neveroddoreven'
```

Dynamic expressions in MATLAB have access to the currently active workspace. This means that you can change any of the functions or variables used in a dynamic expression just by changing variables in the workspace. Repeat the last command of the example above, but this time define the function to be called within the expression using a function handle stored in the base workspace:

```
fun = @fliplr;

palstr = regexp(str, '(.{3,}).?(??@fun($1))', 'match')
palstr =
```

```
    'neveroddoreven'
```

Dynamic Commands that Serve a Functional Purpose — (?@cmd). The `(?@cmd)` operator specifies a MATLAB command that `regexp` or `regexprep` is to run while parsing the overall match expression. Unlike the other dynamic expressions in MATLAB, this operator does not alter the contents of the expression it is used in. Instead, you can use this functionality to get MATLAB to report just what steps it's taking as it parses the contents of one of your regular expressions.

The following example parses a word for zero or more characters followed by two identical characters followed again by zero or more characters:

```
regexp('mississippi', '\w*(\w)\1\w*', 'match')
ans =
    'mississippi'
```

To track the exact steps that MATLAB takes in determining the match, the example inserts a short script `(?@disp($1))` in the expression to display the characters that finally constitute the match. Because the example uses greedy quantifiers, MATLAB attempts to match as much of the string as possible. So, even though MATLAB finds a match toward the beginning of the string, it continues to look for more matches until it arrives at the very end of the string. From there, it backs up through the letters `i` then `p` and the next `p`, stopping at that point because the match is finally satisfied:

```
regexp('mississippi', '\w*(\w)(?@disp($1))\1\w*');
i
p
p
```

Now try the same example again, this time making the first quantifier lazy (`*?`). Again, MATLAB makes the same match:

```
regexp('mississippi', '\w*?(\w)\1\w*', 'match')
ans =
    'mississippi'
```

But by inserting a dynamic script, you can see that this time, MATLAB has matched the string quite differently. In this case, MATLAB uses the very first match it can find, and does not even consider the rest of the string:

```
regexp('mississippi', '\w*?(\w)(?@disp($1))\1\w*';)
m
i
s
```

To demonstrate how versatile this type of dynamic expression can be, consider the next example that progressively assembles a cell array as MATLAB iteratively parses the input string. The `(?!)` metacharacter found at the end of the expression is actually an empty lookahead operator, and forces a failure at each iteration. This forced failure is necessary if you want to trace the steps that MATLAB is taking to resolve the expression.

MATLAB makes a number of passes through the input string, each time trying another combination of letters to see if a fit better than last match can be found. On any passes in which no matches are found, the test results in an empty string. The dynamic script `(?@if(~isempty($&)))` serves to omit these strings from the `matches` cell array:

```
matches = {};
expr = ['(Euler\s)?(Cauchy\s)?(Boole)?(?@if(~isempty($&)),' ...
   'matches{end+1}=$&;end)(?!)'];

regexp('Euler Cauchy Boole', expr);

matches
matches =
    'Euler Cauchy Boole'    'Euler Cauchy '    'Euler '
'Cauchy Boole'    'Cauchy '    'Boole'
```

The metacharacters `$&` (or the equivalent `$0`), `$`` , and `$'` refer to that part of the input string that is currently a match, all characters that precede the current match, and all characters to follow the current match, respectively. These metacharacters are sometimes useful when working with dynamic expressions, particularly those that employ the `(?@cmd)` operator.

This example parses the input string looking for the letter `g`. At each iteration through the string, `regexp` compares the current character with `g`, and not finding it, advances to the next character. The example tracks the progress of scan through the string by marking the current location being parsed with a `^` character.

(The $` and $· metacharacters capture that part of the string that precedes and follows the current parsing location. You need two single-quotation marks ($'') to express the sequence $· when it appears within a string.)

```
str = 'abcdefghij';
expr = '(?@disp(sprintf(''starting match: [%s^%s]'',$`,$'')))g';

regexp(str, expr, 'once');
starting match: [^abcdefghij]
starting match: [a^bcdefghij]
starting match: [ab^cdefghij]
starting match: [abc^defghij]
starting match: [abcd^efghij]
starting match: [abcde^fghij]
starting match: [abcdef^ghij]
```

Dynamic Operators for the Replacement Expression

The three types of dynamic expressions discussed above can be used only in the match expression (second input) argument of the regular expression functions. MATLAB provides one more type of dynamic expression; this one is for use in a replacement string (third input) argument of the `regexprep` function.

Dynamic Commands that Modify the Replacement Expression — ${cmd}. The `${cmd}` operator modifies the contents of a regular expression replacement string, making this string adaptable to parameters in the input string that might vary from one use to the next. As with the other dynamic expressions used in MATLAB, you can include any number of these expressions within the overall replacement expression.

In the `regexprep` call shown here, the replacement string is `'${convert($1,$2)}'`. In this case, the entire replacement string is a dynamic expression:

```
regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

The dynamic expression tells MATLAB to execute an M-file function named `convert` using the two tokens `(\d+\.?\d*)` and `(\w+)`, derived from the

string being matched, as input arguments in the call to `convert`. The replacement string requires a dynamic expression because the values of `$1` and `$2` are generated at runtime.

The following example defines the M-file named `convert` that converts measurements from imperial units to metric. To convert values from the string being parsed, `regexprep` calls the `convert` function, passing in values for the quantity to be converted and name of the imperial unit:

```
function valout  = convert(valin, units)
switch(units)
    case 'inches'
        fun = @(in)in .* 2.54;    uout = 'centimeters';
    case 'miles'
        fun = @(mi)mi .* 1.6093;  uout = 'kilometers';
    case 'pounds'
        fun = @(lb)lb .* 0.4536;  uout = 'kilograms';
    case 'pints'
        fun = @(pt)pt .* 0.4731;  uout = 'litres';
    case 'ounces'
        fun = @(oz)oz .* 28.35;   uout = 'grams';
end
val = fun(str2num(valin));
valout = [num2str(val) ' ' uout];


regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This highway is 201.1625 kilometers long


regexprep('This pitcher holds 2.5 pints of water', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
   This pitcher holds 1.1828 litres of water


regexprep('This stone weighs about 10 pounds', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

```
ans =
   This stone weighs about 4.536 kilograms
```

As with the `(??@ )` operator discussed in an earlier section, the `${ }` operator has access to variables in the currently active workspace. The following `regexprep` command uses the array `A` defined in the base workspace:

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

regexprep('The columns of matrix _nam are _val', ...
          {'_nam', '_val'}, ...
          {'A', '${sprintf(''%d%d%d '', A)}'})
ans =
The columns of matrix A are 834 159 672
```

String Replacement

The `regexprep` function enables you to replace a string that is identified by a regular expression with another string. The following syntax replaces all occurrences of the regular expression `expr` in string `str` with the string `repstr`. The new string is returned in `s`. If no matches are found, return string `s` is the same as input string `str`.

```
s = regexprep('str', 'expr', 'repstr')
```

The replacement string can include any ordinary characters and also any of the metacharacters shown in the following table:

Operator	Usage
Operators from Character Representation on page 3-73 table	The character represented by the metacharacter sequence
$`	That part of the input string that precedes the current match

Operator	Usage
$& or $0	That part of the input string that is currently a match
$`	That part of the input string that follows the current match. In MATLAB, use $'' to represent the character sequence $`.
$N	The string represented by the token identified by name
$<name>	The string represented by the token identified by name
${cmd}	The string returned when MATLAB executes the command cmd

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the token capture operator (...). Specify the tokens to use in the replacement string using the operators $1, $2, $N to reference the first, second, and Nth tokens captured. (See the section on "Tokens" on page 3-48 and the example "Using Tokens in a Replacement String" on page 3-53 in this documentation for information on using tokens.)

Note When referring to a token within a replacement string, use the number of the token preceded by a dollar sign. For example, refer to token 2 by using $2, and not 2 or \2.

The following example uses both the ${cmd} and $N operators in the replacement strings of nested regexprep commands to capitalize the first letter of each sentence. The inner regexprep looks for the start of the entire string and capitalizes the single instance; the outer regexprep looks for the first letter following a period and capitalizes the two instances:

```
s1 = 'here are a few sentences.';
s2 = 'none are capitalized.';
s3 = 'let''s change that.';
```

```
str = [s1 ' ' s2 ' ' s3]

regexprep(regexprep(str, '(^.)', '${upper($1)}'), ...
   '(?<=\.\s*)([a-z])','${upper($1)}')

ans =
Here are a few sentences. None are capitalized. Let's change that.
```

Make `regexprep` more specific to your needs by specifying any of a number of options with the command. See the `regexprep` reference page for more information on these options.

Handling Multiple Strings

You can use any of the MATLAB regular expression functions with cell arrays of strings as well as with single strings. Any or all of the input parameters (the string, expression, or replacement string) can be a cell array of strings. The `regexp` function requires that the string and expression arrays have the same number of elements. The `regexprep` function requires that the expression and replacement arrays have the same number of elements. (The cell arrays do not have to have the same shape.)

Whenever either input argument in a call to `regexp`, or the first input argument in a call to `regexprep` function is a cell array, all output values are cell arrays of the same size.

This section covers the following topics:

Finding a Single Pattern in Multiple Strings

The example shown here uses the `regexp` function on a cell array of strings `cstr`. It searches each string of the cell array for consecutive matching letters (e.g., `'oo'`). The function returns a cell array of the same size as the input array. Each row of the return array contains the indices for which there was a match against the input cell array.

Here is the input cell array:

```
cstr = {                                    ...
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};
```

Find consecutive matching letters by capturing a letter as a token `(.)` and then repeating that letter as a token reference, `\1`:

```
idx = regexp(cstr, '(.)\1');

whos idx
  Name      Size                    Bytes  Class

  idx       4x1                       296  cell array

idx{:}
ans =                 % 'Whose woods these are I think I know.'
    8                 %         |8

ans =                 % 'His house is in the village though;'
   23                 %                         |23

ans =                 % 'He will not see me stopping here'
    6    14    23     %       |6      |14      |23

ans =                 % 'To watch his woods fill up with snow.'
   15    22           %                |15    |22
```

To return substrings instead of indices, use the `'match'` parameter:

```
mat = regexp(cstr, '(.)\1', 'match');
mat{3}
ans =
   'll'    'ee'    'pp'
```

Finding Multiple Patterns in Multiple Strings

This example uses a cell array of strings in both the input string and the expression. The two cell arrays are of different shapes: `cstr` is 4-by-1 while `expr` is 1-by-4. The command is valid as long as they both have the same number of cells.

Find uppercase or lowercase `'i'` followed by a white-space character in `str{1}`, the sequence `'hou'` in `str{2}`, two consecutive matching letters in `str{3}`, and words beginning with `'w'` followed by a vowel in `str{4}`.

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
idx = regexpi(cstr, expr);

idx{:}
ans =                % 'Whose woods these are I think I know.'
   23    31          %                        |23     |31

ans =                % 'His house is in the village though;'
    5    30          %      |5                       |30

ans =                % 'He will not see me stopping here'
    6    14    23    %       |6      |14      |23

ans =                % 'To watch his woods fill up with snow.'
    4    14    28    %     |4        |14           |28
```

Note that the returned cell array has the dimensions of the input string, `cstr`. The dimensions of the return value are always derived from the input string, whenever the input string is a cell array. If the input string is not a cell array, then it is the dimensions of the expression that determine the shape of the return array.

Replacing Multiple Strings

When replacing multiple strings with `regexprep`, use a single replacement string if the expression consists of a single string. This example uses a common replacement value (`'--'`) for all matches found in the multiple string input `cstr`. The function returns a cell array of strings having the same dimensions as the input cell array:

```
s = regexprep(cstr, '(.)\1', '--', 'ignorecase')
```

```
s =
    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

You can use multiple replacement strings if the expression consists of multiple strings. In this example, the input string and replacement string are both 4-by-1 cell arrays, and the expression is a 1-by-4 cell array. As long as the expression and replacement arrays contain the same number of elements, the statement is valid. The dimensions of the return value match the dimensions of the input string:

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
repl = {'-1-'; '-2-'; '-3-'; '-4-'};

s = regexprep(cstr, expr, repl, 'ignorecase')
s =
    'Whose w-3-ds these are -1-think -1-know.'
    'His -2-se is in the vi-3-age t-2-gh;'
    'He -4--3- not s-3- me sto-3-ing here'
    'To -4-tch his w-3-ds fi-3- up -4-th snow.'
```

Operator Summary

MATLAB provides these operators for working with regular expressions:

- Conditional Expression Operators on page 3-77
- Dynamic Expression Operators on page 3-77
- Replacement String Operators on page 3-78

Character Classes

Operator	Usage
`.`	Any single character, including white space
`[c1c2c3]`	Any character contained within the brackets: c_1 or c_2 or c_3
`[^c1c2c3]`	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
`[c1-c2]`	Any character in the range of c_1 through c_2
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`
`\S`	Any non-whitespace character; equivalent to `[^ \f\n\r\t\v]`
`\w`	Any alphabetic, numeric, or underscore character; equivalent to `[a-zA-Z_0-9]`. (True only for English character sets).
`\W`	Any character that is not alphabetic, numeric, or underscore; equivalent to `[^a-zA-Z_0-9]`. (True only for English character sets).
`\d`	Any numeric digit; equivalent to `[0-9]`
`\D`	Any nondigit character; equivalent to `[^0-9]`

Character Classes (Continued)

Operator	Usage
`\oN` or `\o{N}`	Character of octal value `N`
`\xN` or `\x{N}`	Character of hexadecimal value `N`

Character Representation

Operator	Usage
`\\`	Backslash
`\$`	Dollar sign
`\a`	Alarm (beep)
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\char`	If a character has special meaning in a regular expression, precede it with backslash (`\`) to match it literally.

Grouping Operators

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.

Grouping Operators (Continued)

Operator	Usage
`(?>expr)`	Group atomically.
$\texttt{expr}_1\texttt{|expr}_2$	Match expression $\texttt{expr}_1$ or expression $\texttt{expr}_2$.

Nonmatching Operators

Operator	Usage
`(?#comment)`	Insert a comment into the expression. Comments are ignored in matching.

Positional Operators

Operator	Usage
`^expr`	Match `expr` if it occurs at the beginning of the input string.
`expr$`	Match `expr` if it occurs at the end of the input string.
`\<expr`	Match `expr` when it occurs at the beginning of a word.
`expr\>`	Match `expr` when it occurs at the end of a word.
`\<expr\>`	Match `expr` when it represents the entire word.

Lookaround Operators

Operator	Usage
`(?=expr)`	Look ahead from current position and test if `expr` is found.
`(?!expr)`	Look ahead from current position and test if `expr` is not found

Lookaround Operators (Continued)

Operator	Usage
`(?<=expr)`	Look behind from current position and test if `expr` is found.
`(?<!expr)`	Look behind from current position and test if `expr` is not found.

Quantifiers

Operator	Usage
`expr{m,n}`	Match `expr` when it occurs at least `m` times but no more than `n` times consecutively.
`expr{m,}`	Match `expr` when it occurs at least `m` times consecutively.
`expr{n}`	Match `expr` when it occurs exactly `n` times consecutively. Equivalent to `{n,n}`.
`expr?`	Match `expr` when it occurs 0 times or 1 time. Equivalent to `{0,1}`.
`expr*`	Match `expr` when it occurs 0 or more times consecutively. Equivalent to `{0,}`.
`expr+`	Match `expr` when it occurs 1 or more times consecutively. Equivalent to `{1,}`.
`q_expr*`	Match as much of the quantified expression as possible, where `q_expr` represents any of the expressions shown in the first six rows of this table.

Quantifiers (Continued)

Operator	Usage
`q_expr+`	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails.
`q_expr?`	Match only as much of the quantified expression as necessary.

Ordinal Token Operators

Operator	Usage
`(expr)`	Capture in a token all characters matched by the expression within the parentheses.
`\N`	Match the N^{th} token generated by this command. That is, use `\1` to match the first token, `\2` to match the second, and so on.
`$N`	Insert the match for the N^{th} token in the replacement string. Used only by the `regexprep` function. If `N` is equal to zero, then insert the entire match in the replacement string.
`(?(N)s1|s2)`	If N^{th} token is found, then match `s1`, else match `s2`

Named Token Operators

Operator	Usage
`(?<name>expr)`	Capture in a token all characters matched by the expression within the parentheses. Assign a `name` to the token.
`\k<name>`	Match the token referred to by `name`.

Named Token Operators (Continued)

Operator	Usage
`$<name>`	Insert the match for named token in a replacement string. Used only with the `regexprep` function.
`(?(name)s1|s2)`	If named token is found, then match `s1`; otherwise, match `s2`

Conditional Expression Operators

Operator	Usage
`(?(cond)expr)`	If condition `cond` is true, then match expression `expr`
`(?(cond)`$\texttt{expr}_1$`|`$\texttt{expr}_2$`)`	If condition `cond` is true, then match expression $\texttt{expr}_1$. Otherwise match expression $\texttt{expr}_2$

Dynamic Expression Operators

Operator	Usage
`(??expr)`	Parse `expr` as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called `regexprep` inside of a `regexp` match expression.
`(??@cmd)`	Execute the MATLAB command `cmd`, discarding any output that may be returned. This is often used for diagnosing a regular expression.

Dynamic Expression Operators (Continued)

Operator	Usage
`(?@cmd)`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the match expression. This is a combination of the two dynamic syntaxes shown above: `(??expr)` and `(?@cmd)`.
`${cmd}`	Execute the MATLAB command `cmd`, and include the string returned by `cmd` in the replacement expression.

Replacement String Operators

Operator	Usage
Operators from Character Representation on page 3-73 table	The character represented by the metacharacter sequence
`$``	That part of the input string that precedes the current match
`$&` or `$0`	That part of the input string that is currently a match
`$·`	That part of the input string that follows the current match. In MATLAB, use `$''` to represent the character sequence `$·`.
`$N`	The string represented by the token identified by `name`
`$<name>`	The string represented by the token identified by `name`
`${cmd}`	The string returned when MATLAB executes the command `cmd`

Comma-Separated Lists

In this section...

What Is a Comma-Separated List?

Typing in a series of numbers separated by commas gives you what is called a *comma-separated list*. MATLAB returns each value individually:

```
1, 2, 3
ans =
     1
ans =
     2
ans =
     3
```

Such a list, by itself, is not very useful. But when used with large and more complex data structures like MATLAB structures and cell arrays, the comma-separated list can enable you to simplify your MATLAB code.

Generating a Comma-Separated List

This section describes how to generate a comma-separated list from either a cell array or a MATLAB structure.

Generating a List from a Cell Array

Extracting multiple elements from a cell array yields a comma-separated list. Given a 4-by-6 cell array as shown here

```
C = cell(4, 6);
for k = 1:24,   C{k} = k * 2;   end

C
C =
    [2]    [10]    [18]    [26]    [34]    [42]
    [4]    [12]    [20]    [28]    [36]    [44]
    [6]    [14]    [22]    [30]    [38]    [46]
    [8]    [16]    [24]    [32]    [40]    [48]
```

extracting the fifth column generates the following comma-separated list:

```
C{:, 5}
ans =
    34
ans =
    36
ans =
    38
ans =
    40
```

This is the same as explicitly typing

```
C{1, 5}, C{2, 5}, C{3, 5}, C{4, 5}
```

Generating a List from a Structure

For structures, extracting a field of the structure that exists across one of its dimensions yields a comma-separated list.

Start by converting the cell array used above into a 4-by-1 MATLAB structure with six fields: `f1` through `f6`. Read field `f5` for all rows and MATLAB returns a comma-separated list:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

S.f5
ans =
    34
ans =
```

```
    36
ans =
    38
ans =
    40
```

This is the same as explicitly typing

```
S(1).f5, S(2).f5, S(3).f5, S(4).f5
```

Assigning Output from a Comma-Separated List

You can assign any or all consecutive elements of a comma-separated list to variables with a simple assignment statement. Using the cell array `C` from the previous section, assign the first row to variables `c1` through `c6`:

```
C = cell(4, 6);
for k = 1:24,   C{k} = k * 2;   end

[c1 c2 c3 c4 c5 c6] = C{1,1:6};

c5
c5 =
    34
```

If you specify fewer output variables than the number of outputs returned by the expression, MATLAB assigns the first N outputs to those N variables, and then discards any remaining outputs. In this next example, MATLAB assigns `C{1,1:3}` to the variables `c1`, `c2`, and `c3`, and then discards `C{1,4:6}`:

```
[c1 c2 c3] = C{1,1:6};
```

You can assign structure outputs in the same manner:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

[sf1 sf2 sf3] = S.f5;

sf3
sf3 =
    38
```

You also can use the `deal` function for this purpose.

Assigning to a Comma-Separated List

The simplest way to assign multiple values to a comma-separated list is to use the `deal` function. This function distributes all of its input arguments to the elements of a comma-separated list.

This example initializes a comma-separated list to a set of vectors in a cell array, and then uses `deal` to overwrite each element in the list:

```
c{1} = [31 07];    c{2} = [03 78];

c{:}
ans =
    31     7
ans =
     3    78


[c{:}] = deal([10 20],[14 12]);

c{:}
ans =
    10    20
ans =
    14    12
```

This example does the same as the one above, but with a comma-separated list of vectors in a structure field:

```
s(1).field1 = [31 07];    s(2).field1 = [03 78];

s.field1
ans =
    31     7
ans =
     3    78
```

```
[s.field1] = deal([10 20],[14 12]);

s.field1
ans =
    10    20
ans =
    14    12
```

How to Use the Comma-Separated Lists

Common uses for comma-separated lists are

The following sections provide examples of using comma-separated lists with cell arrays. Each of these examples applies to MATLAB structures as well.

Constructing Arrays

You can use a comma-separated list to enter a series of elements when constructing a matrix or array. Note what happens when you insert a *list* of elements as opposed to adding the cell itself.

When you specify a list of elements with `C{:, 5}`, MATLAB inserts the four individual elements:

```
A = {'Hello', C{:, 5}, magic(4)}
A =
    'Hello'    [34]    [36]    [38]    [40]    [4x4 double]
```

When you specify the `C` cell itself, MATLAB inserts the entire cell array:

```
A = {'Hello', C, magic(4)}
A =
    'Hello'    {4x6 cell}    [4x4 double]
```

Displaying Arrays

Use a list to display all or part of a structure or cell array:

```
A{:}
ans =
    Hello
ans =
    34
ans =
    36
ans =
    38
  .
  .
  .
```

Concatenation

Putting a comma-separated list inside square brackets extracts the specified elements from the list and concatenates them:

```
A = [C{:, 5:6}]
A =
    34    36    38    40    42    44    46    48

whos A
  Name      Size                    Bytes  Class

  A         1x8                        64  double array
```

Function Call Arguments

When writing the code for a function call, you enter the input arguments as a list with each argument separated by a comma. If you have these arguments stored in a structure or cell array, then you can generate all or part of the argument list from the structure or cell array instead. This can be especially useful when passing in variable numbers of arguments.

This example passes several attribute-value arguments to the `plot` function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));

C{1,1} = 'LineWidth';         C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';   C{2,2} = 'k';
C{1,3} = 'MarkerFaceColor';   C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Function Return Values

MATLAB functions can also return more than one value to the caller. These values are returned in a list with each value separated by a comma. Instead of listing each return value, you can use a comma-separated list with a structure or cell array. This becomes more useful for those functions that have variable numbers of return values.

This example returns four values to a cell array:

```
C = cell(1, 4);
[C{:}] = fileparts('work/mytests/strArrays.mat')
C =
    'work/mytests'    'strArrays'    '.mat'     ''
```

Fast Fourier Transform Example

The `fftshift` function swaps the left and right halves of each dimension of an array. For a simple vector such as `[0 2 4 6 8 10]` the output would be `[6 8 10 0 2 4]`. For a multidimensional array, `fftshift` performs this swap along each dimension.

`fftshift` uses vectors of indices to perform the swap. For the vector shown above, the index `[1 2 3 4 5 6]` is rearranged to form a new index `[4 5 6 1 2 3]`. The function then uses this index vector to reposition the elements. For a multidimensional array, `fftshift` must construct an index vector for each dimension. A comma-separated list makes this task much simpler.

Here is the `fftshift` function:

```
function y = fftshift(x)
```

```
numDims = ndims(x);
idx = cell(1, numDims);

for k = 1:numDims
   m = size(x, k);
   p = ceil(m/2);
   idx{k} = [p+1:m 1:p];
   end

y = x(idx{:});
```

The function stores the index vectors in cell array `idx`. Building this cell array is relatively simple. For each of the `N` dimensions, determine the size of that dimension and find the integer index nearest the midpoint. Then, construct a vector that swaps the two halves of that dimension.

By using a cell array to store the index vectors and a comma-separated list for the indexing operation, `fftshift` shifts arrays of any dimension using just a single operation: `y = x(idx{:})`. If you were to use explicit indexing, you would need to write one `if` statement for each dimension you want the function to handle:

```
if ndims(x) == 1
   y = x(index1);
else if ndims(x) == 2
   y = x(index1, index2);
end
```

Another way to handle this without a comma-separated list would be to loop over each dimension, converting one dimension at a time and moving data each time. With a comma-separated list, you move the data just once. A comma-separated list makes it very easy to generalize the swapping operation to an arbitrary number of dimensions.

Program Control Statements

In this section...

Conditional Control — if, switch

This group of control statements enables you to select at run-time which block of code is executed. To make this selection based on whether a condition is true or false, use the `if` statement (which may include `else` or `elseif`). To select from a number of possible options depending on the value of an expression, use the `switch` and `case` statements (which may include `otherwise`).

if, else, and elseif

`if` evaluates a logical expression and executes a group of statements based on the value of the expression. In its simplest form, its syntax is

```
if logical_expression
    statements
end
```

If the logical expression is `true` (that is, if it evaluates to logical `1`), MATLAB executes all the statements between the `if` and `end` lines. It resumes execution at the line following the `end` statement. If the condition is `false` (evaluates to logical `0`), MATLAB skips all the statements between the `if` and `end` lines, and resumes execution at the line following the `end` statement.

For example,

```
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end
```

You can nest any number of `if` statements.

If the logical expression evaluates to a nonscalar value, all the elements of the argument must be nonzero. For example, assume `X` is a matrix. Then the statement

```
if X
    statements
end
```

is equivalent to

```
if all(X(:))
    statements
end
```

The `else` and `elseif` statements further conditionalize the `if` statement:

- The `else` statement has no logical condition. The statements associated with it execute if the preceding `if` (and possibly `elseif` condition) evaluates to logical `0` (`false`).
- The `elseif` statement has a logical condition that it evaluates if the preceding `if` (and possibly `elseif` condition) is `false`. The statements associated with it execute if its logical condition evaluates to logical `1` (`true`). You can have multiple `elseif` statements within an `if` block.

```
if n < 0             % If n negative, display error message.
    disp('Input must be positive');
elseif rem(n,2) == 0 % If n positive and even, divide by 2.
    A = n/2;
else
    A = (n+1)/2;     % If n positive and odd, increment and divide.
end
```

if Statements and Empty Arrays. An `if` condition that reduces to an empty array represents a `false` condition. That is,

```
if A
   S1
else
   S0
```

```
end
```

executes statement `S0` when `A` is an empty array.

switch, case, and otherwise

`switch` executes certain statements based on the value of a variable or expression. Its basic form is

```
switch expression (scalar or string)
   case value1
      statements       % Executes if expression is value1
   case value2
      statements       % Executes if expression is value2
    .
    .
    .
   otherwise
      statements       % Executes if expression does not
                        % match any case
end
```

This block consists of

- The word `switch` followed by an expression to evaluate.
- Any number of `case` groups. These groups consist of the word `case` followed by a possible value for the expression, all on a single line. Subsequent lines contain the statements to execute for the given value of the expression. These can be any valid MATLAB statement including another `switch` block. Execution of a `case` group ends when MATLAB encounters the next `case` statement or the `otherwise` statement. Only the first matching `case` is executed.
- An optional `otherwise` group. This consists of the word `otherwise`, followed by the statements to execute if the expression's value is not handled by any of the preceding `case` groups. Execution of the `otherwise` group ends at the `end` statement.
- An `end` statement.

`switch` works by comparing the input expression to each case value. For numeric expressions, a case statement is true if `(value==expression)`. For string expressions, a case statement is true if `strcmp(value,expression)`.

The code below shows a simple example of the `switch` statement. It checks the variable `input_num` for certain values. If `input_num` is `-1`, `0`, or `1`, the `case` statements display the value as text. If `input_num` is none of these values, execution drops to the `otherwise` statement and the code displays the text `'other value'`.

```
switch input_num
    case -1
        disp('negative one');
    case 0
        disp('zero');
    case 1
        disp('positive one');
    otherwise
         disp('other value');
end
```

Note For C programmers, unlike the C language `switch` construct, the MATLAB `switch` does not "fall through." That is, if the first `case` statement is true, other `case` statements do not execute. Therefore, `break` statements are not used.

`switch` can handle multiple conditions in a single `case` statement by enclosing the case expression in a cell array.

```
switch var
    case 1
        disp('1')
    case {2,3,4}
        disp('2 or 3 or 4')
    case 5
        disp('5')
    otherwise
        disp('something else')
end
```

Loop Control — for, while, continue, break

With loop control statements, you can repeatedly execute a block of code, looping back through the block while keeping track of each iteration with an incrementing index variable. Use the `for` statement to loop a specific number of times. The `while` statement is more suitable for basing the loop execution on how long a condition continues to be true or false. The `continue` and `break` statements give you more control on exiting the loop.

Note You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See "Techniques for Improving Performance" on page 11-4 for more information on this.

---

for

The `for` loop executes a statement or group of statements a predetermined number of times. Its syntax is

```
for index = start:increment:end
    statements
end
```

The default increment is `1`. You can specify any increment, including a negative one. For positive indices, execution terminates when the value of the index exceeds the *`end`* value; for negative increments, it terminates when the index is less than the end value.

For example, this loop executes five times.

```
for n = 2:6
    x(n) = 2 * x(n - 1);
end
```

You can nest multiple `for` loops.

```
for m = 1:5
    for n = 1:100
        A(m, n) = 1/(m + n - 1);
    end
```

```
end
```

Note You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See "Vectorizing Loops" on page 11-4 for details.

Using Arrays as Indices. The index of a `for` loop can be an array. For example, consider an m-by-n array `A`. The statement

```
for k = A
    statements
end
```

sets `k` equal to the vector `A(:,i)`, where `i` is the iteration number of the loop. For the first loop iteration, `k` is equal to `A(:,1)`; for the second, `k` is equal to `A(:,2)`; and so on until `k` equals `A(:,n)`. That is, the loop iterates for a number of times equal to the number of columns in `A`. For each iteration, `k` is a vector containing one of the columns of `A`.

while

The `while` loop executes a statement or group of statements repeatedly as long as the controlling expression is true (`1`). Its syntax is

```
while expression
    statements
end
```

If the expression evaluates to a matrix, all its elements must be `1` for execution to continue. To reduce a matrix to a scalar value, use the `all` and `any` functions.

For example, this `while` loop finds the first integer `n` for which `n!` (`n` factorial) is a 100-digit number.

```
n = 1;
while prod(1:n) < 1e100
    n = n + 1;
end
```

Exit a `while` loop at any time using the `break` statement.

while Statements and Empty Arrays. A `while` condition that reduces to an empty array represents a `false` condition. That is,

```
while A, S1, end
```

never executes statement `S1` when `A` is an empty array.

continue

The `continue` statement passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop. The same holds true for `continue` statements in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

The example below shows a `continue` loop that counts the lines of code in the file, `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered.

```
fid = fopen('magic.m', 'r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) | strncmp(line, '%', 1)
        continue
    end
    count = count + 1;
end
disp(sprintf('%d lines', count));
```

break

The `break` statement terminates the execution of a `for` loop or `while` loop. When a `break` statement is encountered, execution continues with the next statement outside of the loop. In nested loops, `break` exits from the innermost loop only.

The example below shows a `while` loop that reads the contents of the file `fft.m` into a MATLAB character array. A `break` statement is used to exit the `while` loop when the first empty line is encountered. The resulting character array contains the M-file help for the `fft` program.

```
fid = fopen('fft.m', 'r');
s = '';
while ~feof(fid)
   line = fgetl(fid);
   if isempty(line)
      break
   end
   s = strvcat(s, line);
end
disp(s)
```

Error Control — try, catch

Error control statements provide a way for you to take certain actions in the event of an error. Use the `try` statement to test whether a certain command in your code generates an error. If an error does occur within the `try` block, MATLAB immediately jumps to the corresponding `catch` block. The `catch` part of the statement needs to respond in some way to the error.

try and catch

The general form of a try-catch statement sequence is

```
try
   statement
   ...
   statement
catch
   statement
   ...
   statement
end
```

In this sequence, the statements between `try` and `catch` are executed until an error occurs. The statements between `catch` and `end` are then executed. Use `lasterr` to see the cause of the error. If an error occurs between `catch`

and `end`, MATLAB terminates execution unless another `try-catch` sequence has been established.

Program Termination — return

Program termination control enables you to exit from your program at some point prior to its normal termination point.

return

After a MATLAB function runs to completion, it terminates and returns control either to the function that called it, or to the keyboard. If you need to exit a function prior to the point of normal completion, you can force an early termination using the `return` function. `return` immediately terminates the current sequence of commands and exits the currently running function.

`return` is also used to terminate `keyboard` mode.

Symbol Reference

In this section...

This section does not include symbols used in arithmetic, relational, and logical operations. For a description of these symbols, see the top of the list. "Functions — Alphabetical List" in the MATLAB Help browser.

Asterisk — *

An asterisk in a filename specification is used as a wildcard specifier, as described below.

Filename Wildcard

Wildcards are generally used in file operations that act on multiple files or directories. They usually appear in the string containing the file or directory specification. MATLAB matches all characters in the name exactly except for the wildcard character `*`, which can match any one or more characters.

To locate all files with names that start with `'january_'` and have a `mat` file extension, use

```
dir('january_*.mat')
```

You can also use wildcards with the `who` and `whos` functions. To get information on all variables with names starting with `'image'` and ending with `'Offset'`, use

```
whos image*Offset
```

At — @

The `@` sign signifies either a function handle constructor or a directory that supports a MATLAB class.

Function Handle Constructor

The `@` operator forms a handle to either the named function that follows the `@` sign, or to the anonymous function that follows the `@` sign.

Function Handles in General. Function handles are commonly used in passing functions as arguments to other functions. Construct a function handle by preceding the function name with an `@` sign:

```
fhandle = @myfun
```

You can read more about function handles in “Function Handles” on page 4-22.

Handles to Anonymous Functions. Anonymous functions give you a quick means of creating simple functions without having to create M-files each time. You can construct an anonymous function and a handle to that function using the syntax

```
fhandle = @(arglist) body
```

where `body` defines the body of the function and `arglist` is the list of arguments you can pass to the function.

See “Anonymous Functions” on page 5-3 for more information.

Class Directory Designator

A MATLAB class directory contains source files that define the methods and properties of a class. All MATLAB class directory names must begin with an `@` sign:

```
\@myclass\get.m
```

See “MATLAB Classes” on page 2-117 for more information.

Colon — :

The colon operator generates a sequence of numbers that you can use in creating or indexing into arrays. See “Generating a Numeric Sequence” on page 1-11 for more information on using the colon operator.

Numeric Sequence Range

Generate a sequential series of regularly spaced numbers from `first` to `last` using the syntax `first:last`. For an incremental sequence from 6 to 17, use

```
N = 6:17
```

Numeric Sequence Step

Generate a sequential series of numbers, each number separated by a `step` value, using the syntax `first:step:last`. For a sequence from 2 through 38, stepping by 4 between each entry, use

```
N = 2:4:38
```

Indexing Range Specifier

Index into multiple rows or columns of a matrix using the colon operator to specify a range of indices:

```
B = A(7, 1:5);          % Read columns 1-5 of row 7.
B = A(4:2:8, 1:5);      % Read columns 1-5 of rows 4, 6, and 8.
B = A(:, 1:5);          % Read columns 1-5 of all rows.
```

Conversion to Column Vector

Convert a matrix or array to a column vector using the colon operator as a single index:

```
A = rand(3,4);
B = A(:);
```

Preserving Array Shape on Assignment

Using the colon operator on the left side of an assignment statement, you can assign new values to array elements without changing the shape of the array:

```
A = rand(3,4);
A(:) = 1:12;
```

Comma — ,

A comma is used to separate the following types of elements.

Row Element Separator

When constructing an array, use a comma to separate elements that belong in the same row:

```
A = [5.92, 8.13, 3.53]
```

Array Index Separator

When indexing into an array, use a comma to separate the indices into each dimension:

```
X = A(2, 7, 4)
```

Function Input and Output Separator

When calling a function, use a comma to separate output and input arguments:

```
function [data, text] = xlsread(file, sheet, range, mode)
```

Command or Statement Separator

To enter more than one MATLAB command or statement on the same line, separate each command or statement with a comma:

```
for k = 1:10,   sum(A(k)),   end
```

Curly Braces — { }

Use curly braces to construct or get the contents of cell arrays.

Cell Array Constructor

To construct a cell array, enclose all elements of the array in curly braces:

```
C = {[2.6 4.7 3.9], rand(8)*6, 'C. Coolidge'}
```

Cell Array Indexing

Index to a specific cell array element by enclosing all indices in curly braces:

```
A = C{4,7,2}
```

See “Cell Arrays” on page 2-93 for more information.

Dot — .

The single dot operator has the following different uses in MATLAB.

Structure Field Definition

Add fields to a MATLAB structure by following the structure name with a dot and then a field name:

```
funds(5,2).bondtype = 'Corporate';
```

See “Structures” on page 2-74 for more information.

Object Method Specifier

Specify the properties of an instance of a MATLAB class using the object name followed by a dot, and then the property name:

```
val = asset.current_value
```

See “MATLAB Classes” on page 2-117 for more information.

Dot-Dot — ..

Two dots in sequence refer to the parent of the current directory.

Parent Directory

Specify the directory immediately above your current directory using two dots. For example, to go up two levels in the directory tree and down into the `testdir` directory, use

```
cd ..\..\testdir
```

Dot-Dot-Dot (Ellipsis) — ...

A series of three consecutive periods (`...`) is the line continuation operator in MATLAB. This is often referred to as an *ellipsis*, but it should be noted that the line continuation operator is a three-character operator and is different from the single-character ellipsis represented in ASCII by the hexadecimal number 2026.

Line Continuation

Continue any MATLAB command or expression by placing an ellipsis at the end of the line to be continued:

```
sprintf('The current value of %s is %d', ...
         vname, value)
```

Entering Long Strings. You cannot use an ellipsis within single quotes to continue a string to the next line:

```
string = 'This is not allowed and will generate an ...
   error in MATLAB.'
```

To enter a string that extends beyond a single line, piece together shorter strings using either the concatenation operator (`[]`) or the `sprintf` function.

Here are two examples:

```
quote1 = [
 'Tiger, tiger, burning bright in the forests of the night,' ...
 'what immortal hand or eye could frame thy fearful symmetry?'];
quote2 = sprintf('%s%s%s', ...
 'In Xanadu did Kubla Khan a stately pleasure-dome decree,', ...
 'where Alph, the sacred river, ran ', ...
 'through caverns measureless to man down to a sunless sea.');
```

Dot-Parentheses — .( )

Use dot-parentheses to specify the name of a dynamic structure field.

Dynamic Structure Fields

Sometimes it is useful to reference structures with field names that can vary. For example, the referenced field might be passed as an argument to a function. Dynamic field names specify a variable name for a structure field.

The variable `fundtype` shown here is a dynamic field name:

```
type = funds(5,2).(fundtype);
```

See “Using Dynamic Field Names” on page 2-80 for more information.

Exclamation Point — !

The exclamation point precedes operating system commands that you want to execute from within MATLAB.

Shell Escape

The exclamation point initiates a shell escape function. Such a function is to be performed directly by the operating system:

```
!rmdir oldtests
```

See “Shell Escape Functions” on page 3-28 for more information.

Parentheses — ( )

Parentheses are used mostly for indexing into elements of an array or for specifying arguments passed to a called function.

Array Indexing

When parentheses appear to the right of a variable name, they are indices into the array stored in that variable:

```
A(2, 7, 4)
```

Function Input Arguments

When parentheses follow a function name in a function declaration or call, the enclosed list contains input arguments used by the function:

```
function sendmail(to, subject, message, attachments)
```

Percent — %

The percent sign is most commonly used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in your code. Some functions also interpret the percent sign as a conversion specifier.

See “Help Text” on page 4-11 for more information.

Single Line Comments

Precede any one-line comments in your code with a percent sign. MATLAB does not execute anything that follows a percent sign (that is, unless the sign is quoted, `'%'`):

```
% The purpose of this routine is to compute
% the value of ...
```

Conversion Specifiers

Some functions, like `sscanf` and `sprintf`, precede conversion specifiers with the percent sign:

```
sprintf('%s = %d', name, value)
```

Percent-Brace — %{ %}

The `%{` and `%}` symbols enclose a block of comments that extend beyond one line.

Block Comments

Enclose any multiline comments with percent followed by an opening or closing brace.

```
%{
The purpose of this routine is to compute
the value of ...
%}
```

Note With the exception of whitespace characters, the `%{` and `%}` operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Semicolon — ;

The semicolon can be used to construct arrays, suppress output from a MATLAB command, or to separate commands entered on the same line.

Array Row Separator

When used within square brackets to create a new array or concatenate existing arrays, the semicolon creates a new row in the array:

```
A = [5, 8; 3, 4]
A =
     5     8
     3     4
```

Output Suppression

When placed at the end of a command, the semicolon tells MATLAB not to display any output from that command. In this example, MATLAB does not display the resulting 100-by-100 matrix:

```
A = ones(100, 100);
```

Command or Statement Separator

Like the comma operator, you can enter more than one MATLAB command on a line by separating each command with a semicolon. MATLAB suppresses output for those commands terminated with a semicolon, and displays the output for commands terminated with a comma.

In this example, assignments to variables `A` and `C` are terminated with a semicolon, and thus do not display. Because the assignment to `B` is comma-terminated, the output of this one command is displayed:

```
A = 12.5;  B = 42.7,  C = 1.25;
B =
   42.7000
```

Single Quotes — ' '

Single quotes are the constructor symbol for MATLAB character arrays.

Character and String Constructor

MATLAB constructs a character array from all characters enclosed in single quotes. If only one character is in quotes, then MATLAB constructs a 1-by-1 array:

```
S = 'Hello World'
```

See “Characters and Strings” on page 2-37 for more information.

Space Character

The space character serves a purpose similar to the comma in that it can be used to separate row elements in an array constructor, or the values returned by a function.

Row Element Separator

You have the option of using either commas or spaces to delimit the row elements of an array when constructing the array. To create a 1-by-3 array, use

```
A = [5.92 8.13 3.53]
A =
    5.9200    8.1300    3.5300
```

When indexing into an array, you must always use commas to reference each dimension of the array.

Function Output Separator

Spaces are allowed when specifying a list of values to be returned by a function. You can use spaces to separate return values in both function declarations and function calls:

```
function [data text] = xlsread(file, sheet, range, mode)
```

Slash and Backslash — / \

The slash (/) and backslash (\) characters separate the elements of a path or directory string. On Windows-based systems, both slash and backslash have the same effect. On UNIX-based systems, you must use slash only.

On a Windows system, you can use either backslash or slash:

```
dir([matlabroot '\toolbox\matlab\elmat\shiftdim.m'])
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

On a UNIX system, use only the forward slash:

```
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

Square Brackets — [ ]

Square brackets are used in array construction and concatenation, and also in declaring and capturing values returned by a function.

Array Constructor

To construct a matrix or array, enclose all elements of the array in square brackets:

```
A = [5.7, 9.8, 7.3; 9.2, 4.5, 6.4]
```

Concatenation

To combine two or more arrays into a new array through concatenation, enclose all array elements in square brackets:

```
A = [B, eye(6), diag([0:2:10])]
```

Function Declarations and Calls

When declaring or calling a function that returns more than one output, enclose each return value that you need in square brackets:

```
[data, text] = xlsread(file, sheet, range, mode)
```

Internal MATLAB Functions

In this section...

Overview

Many of the functions provided with MATLAB are implemented as M-files just like the M-files that you will create with MATLAB. Other MATLAB functions are precompiled executable programs called built-ins that run much more efficiently. Many of the MATLAB functions are also overloaded so that they handle different data types appropriately.

M-File Functions

If you look in the subdirectories of the `toolbox\matlab` directory, you can find the M-file sources to many of the functions supplied with MATLAB. You can locate your `toolbox\matlab` directory by typing

```
dir([matlabroot '\toolbox\matlab\'])
```

MATLAB functions with an M-file source are just like any other functions coded with MATLAB. When one of these M-file functions is called, MATLAB parses and executes each line of code in the M-file. It saves the parsed version of the function in memory, eliminating parsing time on any further calls to this function.

Identifying M-File Functions

To find out if a function is implemented with an M-file, use the `exist` function. The `exist` function searches for the name you enter on the MATLAB path and returns a number identifying the source. If the source is an M-file, then `exist` returns the number `2`. This example identifies the source for the `repmat` function as an M-file:

```
exist repmat
ans =
     2
```

The `exist` function also returns 2 for files that have a file type unknown to MATLAB. However, if you invoke `exist` on a MATLAB function name, the file type will be known to MATLAB and will return 2 only on M-files.

Viewing the Source Code

One advantage of functions implemented as M-files is that you can look at the source code. This may help when you need to understand why the function returns a value you did not expect, if you need to figure out how to code something in MATLAB that is already coded in a function, or perhaps to help you create a function that overloads one of the MATLAB functions.

To find the source code for any MATLAB M-file function, use `which`:

```
which repmat
   D:\matlabR14\toolbox\matlab\elmat\repmat.m
```

Built-In Functions

Functions that are frequently used or that can take more time to execute are often implemented as executable files. These functions are called *built-ins*.

Unlike M-file functions, you cannot see the source code for built-ins. Although most built-in functions do have an M-file associated with them, this file is there mainly to supply the help documentation for the function. Take the `reshape` function, for example, and find it on the MATLAB path:

```
which reshape
   D:\matlabR14\toolbox\matlab\elmat\reshape.m
```

If you type this M-file out, you will see that it consists almost entirely of help text. At the bottom is a call to the built-in executable image.

Identifying Built-In Functions

As with M-file functions, you can identify which functions are built-ins using the `exist` function. This function identifies built-ins by returning the number `5`:

```
exist reshape
ans =
     5
```

Forcing a Built-In Call

If you overload any of the MATLAB built-in functions to handle a specific data type, then MATLAB will always call the overloaded function on that type. If, for some reason, you need to call the built-in version, you can override the usual calling mechanism using a function called `builtin`. The expression

```
builtin('reshape', arg1, arg2, ..., argN);
```

forces a call to MATLAB built-in `reshape`, passing the arguments shown even though an overload exists for the data types in this argument list.

Overloaded MATLAB Functions

An overloaded function is an additional implementation of an existing function that has been designed specifically to handle a certain data type. When you pass an argument of this type in a call to the function, MATLAB looks for the function implementation that handles that type and executes that function code.

Each overloaded MATLAB function has an M-file on the MATLAB path. The M-files for a certain data type (or class) are placed in a directory named with an `@` sign followed by the class name. For example, to overload the MATLAB `plot` function to plot expressions of a class named `polynom` differently than other data types, you would create a directory called `@polynom` and store your own version of `plot.m` in that directory.

You can add your own overloads to any function by creating a class directory for the data type you wish to support for that function, and creating an M-file that handles that type in a manner different from the default. See “Setting

Up Class Directories” on page 9-6 and “Designing User Classes in MATLAB” on page 9-9.

When you use the `which` command with the `-all` option, MATLAB returns all occurrences of the file you are looking for. This is an easy way to find functions that are overloaded:

```
which -all set          % Show all implementations for 'set'
```

4

M-File Programming

Program Development

In this section...

Overview

When you write a program in MATLAB, you save it to a file called an M-file (named after its `.m` file extension). There are two types of M-files that you can write: scripts and functions. This section covers basic program development, describes how to write and call scripts and functions, and shows how to pass different types of data in a function call. Associated with each step of this process are certain MATLAB tools and utilities that are fully documented in the Desktop Tools and Development Environment documentation.

For more ideas on good programming style, see “Program Development” on page 12-20 in the MATLAB Programming Tips documentation. The Programming Tips section is a compilation of useful pieces of information that can show you alternate and often more efficient ways to accomplish common programming tasks while also expanding your knowledge of MATLAB.

Creating a Program

You can type in your program code using any text editor. This section focuses on using the MATLAB Editor/Debugger for this purpose. The Editor/Debugger is fully documented in Ways to Edit and Debug Files in the Desktop Tools and Development Environment documentation.

The first step in creating a program is to open an editing window. To create a new M-file, type the word `edit` at the MATLAB command prompt. To edit an existing M-file, type `edit` followed by the filename:

```
edit drawPlot.m
```

MATLAB opens a new window for entering your program code. As you type in your program, MATLAB keeps track of the line numbers in the left column.

Saving the Program

It is usually a good idea to save your program periodically while you are in the development process. To do this, click **File > Save** in the Editor/Debugger. Enter a filename with a `.m` extension in the **Save file as** dialog box that appears and click **OK**. It is customary and less confusing if you give the M-file the same name as the first function in the M-file.

Running the Program

Before trying to run your program, make sure that its M-file is on the MATLAB path. The MATLAB path defines those directories that you want MATLAB to know about when executing M-files. The path includes all the directories that contain functions provided with MATLAB. It should also include any directories that you use for your own functions.

Use the `which` function to see if your program is on the path:

```
which drawPlot
   D:\matlabR14\work\drawPlot.m
```

If not, add its directory to the path using the `addpath` function:

```
addpath('D:\matlabR14\work')
```

Now you can run the program just by typing the name of the M-file at the MATLAB command prompt:

```
drawPlot(xdata, ydata)
```

Getting the Bugs Out

In all but the simplest programs, you are likely to encounter some type of unexpected behavior when you run the program for the first time. Program defects can show up in the form of warning or error messages displayed in the command window, programs that hang (never terminate), inaccurate results,

or some number of other symptoms. This is where the second functionality of the MATLAB Editor/Debugger becomes useful.

The MATLAB Debugger enables you to examine the inner workings of your program while you run it. You can stop the execution of your program at any point and then continue from that point, stepping through the code line by line and examining the results of each operation performed. You have the choice of operating the debugger from the Editor window that displays your program, from the MATLAB command line, or both.

The Debugging Process

You can step through the program right from the start if you want. For longer programs, you will probably save time by stopping the program somewhere in the middle and stepping through from there. You can do this by approximating where the program code breaks and setting a stopping point (or *breakpoint*) at that line. Once a breakpoint has been set, start your program from the MATLAB command prompt. MATLAB opens an Editor/Debugger window (if it is not already open) showing a green arrow pointing to the next line to execute.

From this point, you can examine any values passed into the program, or the results of each operation performed. You can step through the program line by line to see which path is taken and why. You can step into any functions that your program calls, or choose to step over them and just see the end results. You can also modify the values assigned to a variable and see how that affects the outcome.

To learn about using the MATLAB Debugger, see Debugging and Improving M-Files in the Desktop Tools and Development Environment documentation. Type `help debug` for a listing of all MATLAB debug functions.

For programming tips on how to debug, see “Debugging” on page 12-23.

Cleaning Up the Program

Even after your program is bug-free, there are still some steps you can take to improve its performance and readability. The MATLAB M-Lint utility generates a report that can highlight potential problems in your code. For example, you may be using the elementwise AND operator (`&`) where the

short-circuit AND (&&) is more appropriate. You may be using the `find` function in a context where logical subscripting would be faster.

MATLAB offers M-Lint and several other reporting utilities to help you make the finishing touches to your program code. These tools are described under Tuning and Refining M-Files in the Desktop Tools and Development Environment documentation.

Improving Performance

The MATLAB Profiler generates a report that shows how your program spends its processing time. For details about using the MATLAB Profiler, see Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation. For tips on other ways to improve the performance of your programs, see Chapter 11, "Improving Performance and Memory Usage".

Three types of reports are available:

- "Summary Report" on page 4-5
- "Detail Report" on page 4-5
- "File Listing" on page 4-6

Summary Report

The summary report provides performance information on your main program and on every function it calls. This includes how many times each function is called, the total time spent in that function, along with a bar graph showing the relative time spent by each function.

Detail Report

When you click a function name in the summary report, MATLAB displays a detailed report on that function. This report shows the lines of that function that take up the most time, the time spent executing that line, the percentage of total time for that function that is spent on that line, and a bar graph showing the relative time spent on the line.

File Listing

The detail report for a function also displays the entire M-file code for that function. This listing enables you to view the time-consuming code in the context of the entire function body. For every line of code that takes any significant time, additional performance information is provided by the statistics and by the color and degree of highlighting of the program code.

Checking It In

Source control systems offer a way to manage large numbers of files while they are under development. They keep track of the work done on these files as your project progresses, and also ensure that changes are made in a secure and orderly fashion.

If you have a source control system available to you, you will probably want to check your M-files into the system once they are complete. If further work is required on one of those files, you just check it back out, make the necessary modifications, and then check it back in again.

MATLAB provides an interface to external source control systems so that you can check files in and out directly from your MATLAB session. See Revision Control in the Desktop Tools and Development Environment documentation for instructions on how to use this interface.

Working with M-Files

In this section...

Overview

MATLAB provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file, giving the file a name of `filename.m`. The term you use for `filename` becomes the new command that MATLAB associates with the program. The file extension of `.m` makes this a MATLAB M-file.

Types of M-Files

M-files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that also accept input arguments and produce output.

MATLAB scripts:

- Are useful for automating a series of steps you need to perform many times.
- Do not accept input arguments or return output arguments.
- Store variables in a workspace that is shared with other scripts and with the MATLAB command line interface.

MATLAB functions:

- Are useful for extending the MATLAB language for your application.
- Can accept input arguments and return output arguments.

- Store variables in a workspace internal to the function.

Basic Parts of an M-File

This simple function shows the basic parts of an M-file. Note that any line that begins with `%` is not executable:

```
function f = fact(n)                    Function
definition line
% Compute a factorial value.            H1 line
% FACT(N) returns the factorial of N,   Help text
% usually denoted by N!

% Put simply, FACT(N) is PROD(1:N).     Comment
f = prod(1:n);                          Function body
```

The table below briefly describes each of these M-file parts. Both functions and scripts can have all of these parts, except for the function definition line which applies to functions only. These parts are described in greater detail following the table.

M-File Element	Description
Function definition line (functions only)	Defines the function name, and the number and order of input and output arguments
H1 line	A one line summary description of the program, displayed when you request `help` on an entire directory, or when you use `lookfor`
Help text	A more detailed description of the program, displayed together with the H1 line when you request `help` on a specific function
Function or script body	Program code that performs the actual computations and assigns values to any output arguments
Comments	Text in the body of the program that explains the internal workings of the program

Function Definition Line

The function definition line informs MATLAB that the M-file contains a function, and specifies the argument calling sequence of the function. The function definition line for the `fact` function is

All MATLAB functions have a function definition line that follows this pattern.

Function Name. Function names must begin with a letter, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed length (returned by the function `namelengthmax`). Because variables must obey similar rules, you can use the `isvarname` function to check whether a function name is valid:

```
isvarname myfun
```

Although function names can be of any length, MATLAB uses only the first `N` characters of the name (where `N` is the number returned by the function `namelengthmax`) and ignores the rest. Hence, it is important to make each function name unique in the first `N` characters:

```
N = namelengthmax
N =
    63
```

Note Some operating systems may restrict file names to shorter lengths.

The name of the text file that contains a MATLAB function consists of the function name with the extension `.m` appended. For example,

```
average.m
```

If the filename and the function definition line name are different, the internal (function) name is ignored. Thus, if `average.m` is the file that defines a function named `computeAverage`, you would invoke the function by typing

```
average
```

Note While the function name specified on the function definition line does not have to be the same as the filename, it is best to use the same name for both to avoid confusion.

Function Arguments. If the function has multiple output values, enclose the output argument list in square brackets. Input arguments, if present, are enclosed in parentheses following the function name. Use commas to separate multiple input or output arguments. Here is the declaration for a function named `sphere` that has three inputs and three outputs:

```
function [x, y, z] = sphere(theta, phi, rho)
```

If there is no output, leave the output blank

```
function printresults(x)
```

or use empty square brackets:

```
function [] = printresults(x)
```

The variables that you pass to the function do not need to have the same name as those in the function definition line.

The H1 Line

The H1 line, so named because it is the first help text line, is a comment line immediately following the function definition line. Because it consists of comment text, the H1 line begins with a percent sign, `%`. For the `average` function, the H1 line is

```
% AVERAGE Mean of vector elements.
```

This is the first line of text that appears when a user types `help` *`functionname`* at the MATLAB prompt. Further, the `lookfor` function searches on and displays only the H1 line. Because this line provides important summary information about the M-file, it is important to make it as descriptive as possible.

Help Text

You can create online help for your M-files by entering help text on one or more consecutive comment lines at the start of your M-file program. MATLAB considers the first group of consecutive lines immediately following the H1 line that begin with `%` to be the online help text for the function. The first line without `%` as the left-most character ends the help.

The help text for the `average` function is

```
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
```

When you type `help` *`functionname`* at the command prompt, MATLAB displays the H1 line followed by the online help text for that function. The help system ignores any comment lines that appear after this help block.

Note Help text in an M-file can be viewed at the MATLAB command prompt only (using `help functionname`). You cannot display this text using the MATLAB Help browser. You can, however, use the Help browser to get help on MATLAB functions and also to read the documentation on any MathWorks products.

The Function or Script Body

The function body contains all the MATLAB code that performs computations and assigns values to output arguments. The statements in the function body can consist of function calls, programming constructs like flow control and interactive input/output, calculations, assignments, comments, and blank lines.

For example, the body of the `average` function contains a number of simple programming statements:

```
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1)) % Flow control
    error('Input must be a vector') % Error message display
end
y = sum(x)/length(x);             % Computation and assignment
```

Comments

As mentioned earlier, comment lines begin with a percent sign (%). Comment lines can appear anywhere in an M-file, and you can append comments to the end of a line of code. For example,

```
% Add up all the vector elements.
y = sum(x)           % Use the sum function.
```

In addition to comment lines, you can insert blank lines anywhere in an M-file. Blank lines are ignored. However, a blank line can indicate the end of the help text entry for an M-file.

Block Comments. To write comments that require more than one line, use the block comment operators, %{ and %}:

```
%{
This next block of code checks the number of inputs
passed in, makes sure that each input is a valid data
type, and then branches to start processing the data.
%}
```

Note The %{ and %} operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Creating a Simple M-File

You create M-files using a text editor. MATLAB provides a built-in editor, but you can use any text editor you like. Once you have written and saved the M-file, you can run the program as you would any other MATLAB function or command.

The process looks like this:

1 Create an M-file using a text editor.

```
function c = myfile(a,b)
c = sqrt((a.^2)+(b.^2))
```

2 Call the M-file from the command line, or from within another M-file.

```
a = 7.5
b = 3.342
c = myfile(a,b)

c =

    8.2109
```

Using Text Editors

M-files are ordinary text files that you create using a text editor. If you use the MATLAB Editor/Debugger, open a new file by selecting **New > M-File** from the **File** menu at the top of the MATLAB Command Window.

Another way to edit an M-file is from the MATLAB command line using the `edit` function. For example,

```
edit foo
```

opens the editor on the file `foo.m`. Omitting a filename opens the editor on an untitled file.

You can create the `fact` function shown in “Basic Parts of an M-File” on page 4-8 by opening your text editor, entering the lines shown, and saving the text in a file called `fact.m` in your current directory.

Once you have created this file, here are some things you can:

- List the names of the files in your current directory:

```
what
```

- List the contents of M-file `fact.m`:

```
type fact
```

- Call the `fact` function:

```
fact(5)
ans =
     120
```

A Word of Caution on Saving M-Files

Save any M-files you create and any MathWorks supplied M-files that you edit in directories outside of the directory tree in which the MATLAB software is installed. If you keep your files in any of the installed directories, your files may be overwritten when you install a new version of MATLAB.

MATLAB installs its software into directories under *`matlabroot`*`/toolbox`. To see what *`matlabroot`* is on your system, type `matlabroot` at the MATLAB command prompt.

Also note that locations of files in the *`matlabroot`*`/toolbox` directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to *`matlabroot`*`/toolbox` directories using an external editor, or if you add or remove files from these directories using file system operations, enter the commands `clear functionname` and `rehash toolbox` before you use the files in the current session.

For more information, see the `rehash` function reference page or the section Toolbox Path Caching in the Desktop Tools and Development Environment documentation.

Providing Help for Your Program

You can provide user information for the programs you write by including a help text section at the beginning of your M-file. (See “Help Text” on page 4-11).

You can also make help entries for an entire directory by creating a file with the special name `Contents.m` that resides in the directory. This file must contain only comment lines; that is, every line must begin with a percent sign. MATLAB displays the lines in a `Contents.m` file whenever you type

```
help directoryname
```

`Contents.m` files are optional. You might have directories of your own with M-files that you don’t necessarily want public. For this or other reasons, you might choose not to provide this type of help listing for these directories. If you have a directory that is on the path that does not have a `Contents.m` file, MATLAB displays `(No table of contents file)` for that directory in response to the `help` command. If you do not want to see this displayed, creating an empty `Contents.m` file will disable this message for that directory.

Also, if a directory does not contain a `Contents.m` file, typing

```
help directoryname
```

displays the first help line (the H1 line) for each M-file in the directory.

There is a tool in the Current Directory browser, called the Contents Report, that you can use to help create and validate your `Contents.m` files. See Contents File Report in the MATLAB Desktop Tools and Development Environment documentation for more information.

Creating P-Code Files

You can save a preparsed version of a function or script, called P-code files, for later MATLAB sessions using the `pcode` function. For example,

```
pcode average
```

parses `average.m` and saves the resulting pseudocode to the file named `average.p`. This saves MATLAB from reparsing `average.m` the first time you call it in each session.

MATLAB is very fast at parsing so the `pcode` function rarely makes much of a speed difference.

One situation where `pcode` does provide a speed benefit is for large GUI applications. In this case, many M-files must be parsed before the application becomes visible.

You can also use `pcode` to hide algorithms you have created in your M-file, if you need to do this for proprietary reasons.

M-File Scripts and Functions

In this section...

M-File Scripts

Scripts are the simplest kind of M-file because they have no input or output arguments. They are useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line.

The Base Workspace

Scripts share the base workspace with your interactive MATLAB session and with other scripts. They operate on existing data in the workspace, or they can create new data on which to operate. Any variables that scripts create remain in the workspace after the script finishes so you can use them for further computations. You should be aware, though, that running a script can unintentionally overwrite data stored in the base workspace by commands entered at the MATLAB command prompt.

Simple Script Example

These statements calculate `rho` for several trigonometric functions of `theta`, then create a series of polar plots:

```
% An M-file script to produce       % Comment lines
% "flower petal" plots
theta = -pi:0.01:pi;                % Computations
rho(1,:) = 2 * sin(5 * theta) .^ 2;
rho(2,:) = cos(10 * theta) .^ 3;
rho(3,:) = sin(theta) .^ 2;
rho(4,:) = 5 * cos(3.5 * theta) .^ 3;
for k = 1:4
```

```
    polar(theta, rho(k,:))          % Graphics output
    pause
end
```

Try entering these commands in an M-file called `petals.m`. This file is now a MATLAB script. Typing `petals` at the MATLAB command line executes the statements in the script.

After the script displays a plot, press **Enter** or **Return** to move to the next plot. There are no input or output arguments; `petals` creates the variables it needs in the MATLAB workspace. When execution completes, the variables (`i`, `theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.

M-File Functions

Functions are program routines, usually implemented in M-files, that accept input arguments and return output arguments. They operate on variables within their own workspace. This workspace is separate from the workspace you access at the MATLAB command prompt.

The Function Workspace

Each M-file function has an area of memory, separate from the MATLAB base workspace, in which it operates. This area, called the function workspace, gives each function its own workspace context.

While using MATLAB, the only variables you can access are those in the calling context, be it the base workspace or that of another function. The variables that you pass to a function must be in the calling context, and the function returns its output arguments to the calling workspace context. You can, however, define variables as global variables explicitly, allowing more than one workspace context to access them. You can also evaluate any MATLAB statement using variables from either the base workspace or the workspace of the calling function using the `evalin` function. See “Extending Variable Scope” on page 3-10 for more information.

Simple Function Example

The `average` function is a simple M-file that calculates the average of the elements in a vector:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1))
    error('Input must be a vector')
end
y = sum(x)/length(x);      % Actual computation
```

Try entering these commands in an M-file called `average.m`. The `average` function accepts a single input argument and returns a single output argument. To call the `average` function, enter

```
z = 1:99;

average(z)
ans =
    50
```

Types of Functions

MATLAB provides the following types of functions. Each function type is described in more detail in a later section of this documentation:

- The “Primary M-File Functions” on page 5-15 is the first function in an M-file and typically contains the main program.
- “Subfunctions” on page 5-33 act as subroutines to the main function. You can also use them to define multiple functions within a single M-file.
- “Nested Functions” on page 5-16 are functions defined within another function. They can help to improve the readability of your program and also give you more flexible access to variables in the M-file.
- “Anonymous Functions” on page 5-3 provide a quick way of making a function from any MATLAB expression. You can compose anonymous

functions either from within another function or at the MATLAB command prompt.

- "Overloaded Functions" on page 5-37 are useful when you need to create a function that responds to different types of inputs accordingly. They are similar to overloaded functions in any object-oriented language.
- "Private Functions" on page 5-35 give you a way to restrict access to a function. You can call them only from an M-file function in the parent directory.

You might also see the term function functions in the documentation. This is not really a separate function type. The term function functions refers to any functions that accept another function as an input argument. You can pass a function to another function using a function handle.

Identifying Dependencies

Most any program you write will make calls to other functions and scripts. If you need to know what other functions and scripts your program is dependent upon, use one of the techniques described below.

Simple Display of M-File Dependencies

For a simple display of all M-files referenced by a particular function, follow these steps:

1. Type `clear functions` to clear all functions from memory (see Note below).

   Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions (which you can check using `inmem`) unlock them with `munlock`, and then repeat step 1.

2. Execute the function you want to check. Note that the function arguments you choose to use in this step are important, because you can get different results when calling the same function with different arguments.

3 Type `inmem` to display all M-files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output:

```
[mfiles, mexfiles] = inmem
```

Detailed Display of M-File Dependencies

For a much more detailed display of dependent function information, use the `depfun` function. In addition to M-files, `depfun` shows which built-ins and classes a particular function depends on:

```
[list, builtins, classes] = depfun('strtok.m');

list
list =
   'D:\matlabR14\toolbox\matlab\strfun\strtok.m'
   'D:\matlabR14\toolbox\distcomp\toChar.m'
   'D:\matlabR14\toolbox\matlab\datafun\prod.m'
   'D:\matlabR14\toolbox\matlab\datatypes\@opaque\char.m'
           .
           .
           .
```

Function Handles

In this section...

Constructing a Function Handle

A *function handle* is a MATLAB value that provides a means of calling a function indirectly. You can pass function handles in calls to other functions (often called *function functions*). You can also store function handles in data structures for later use (for example, as Handle Graphics callbacks).

Use the following syntax to construct a function handle, preceding the name of the function with an `@` sign. Use only the function *name*, with no path information, after the `@` sign:

```
fhandle = @functionname
```

MATLAB maps the handle to the function you specify and saves this mapping information in the handle. If there is more than one function with this name, MATLAB maps to the one function source it would dispatch to if you were actually calling the function.

A function handle retains that same mapping even if its corresponding function goes out of scope. For example, if, after creating the handle, you change the MATLAB path so that a different function of the same name now takes precedence, invoking the function handle still executes the code to which the handle was originally mapped.

Handles to Anonymous Functions

Another way to construct a function handle is to create an anonymous function. For example,

```
sqr = @(x) x.^2;
```

creates an anonymous function that computes the square of its input argument `x`. The variable `sqr` contains a handle to the anonymous function. See “Anonymous Functions” on page 5-3 for more information.

Arrays of Function Handles

To store function handles in an array, use a cell array:

```
trigFun = {@sin, @cos, @tan};

plot(trigFun{2}(-pi:0.01:pi))
```

Invalid or Obsolete Function Handles

If you create a handle to a function that is not on the MATLAB path, or if you `load` a handle to a function that is no longer on the path, MATLAB catches the error only when the handle is invoked. You can assign an invalid handle and use it in such operations as `func2str`. MATLAB catches and reports an error only when you attempt to use it in a runtime operation.

Calling a Function Using Its Handle

To execute a function associated with a function handle, use the syntax shown here, treating the function handle `fhandle` as if it were a function name:

```
fhandle(arg1, arg2, ..., argn)
```

If the function being called takes no input arguments, then use empty parentheses after the function handle name:

```
fhandle()
```

Handling Return Values

When you invoke a function by means of its handle, you can capture any or all values returned from the call in the same way you would if you were calling the function directly. Just list the output variable to the left of the

equals sign. When assigning to multiple outputs, enclose the output variables within square brackets:

```
[out1 out2 ...] = fhandle(arg1, arg2, arg3, ...)
```

This example returns multiple values from a call to an anonymous function. Create anonymous function `f` that locates the nonzero elements of an array, and returns the row, column, and value of each element in variables `row`, `col`, and `val`:

```
f = @(X)find(X);
```

Call the function on matrix `m` using the function handle `f`. Because the function uses the MATLAB `find` function which returns up to 3 outputs, you can specify from 0 to 3 outputs in the call:

```
m = [3 2 0; -5 0 7; 0 0 1]
m =
     3     2     0
    -5     0     7
     0     0     1

[row col val] = f(m);

val
val =
     3
    -5
     2
     7
     1
```

Simple Function Handle Example

The following example calls a function `plotFHandle`, passing it a handle for the MATLAB `sin` function. `plotFHandle` then calls the `plot` function, passing it some data and the function handle to `sin`. The plot function calls the function associated with the handle to compute its *y*-axis values:

```
function x = plotFHandle(fhandle, data)
plot(data, fhandle(data))
```

Call `plotFhandle` with a handle to the `sin` function and the value shown below:

```
plotFHandle(@sin, -pi:0.01:pi)
```

Functions That Operate on Function Handles

MATLAB provides the following functions for working with function handles. See the reference pages for these functions for more information.

Function	Description
`functions`	Return information describing a function handle.
`func2str`	Construct a function name string from a function handle.
`str2func`	Construct a function handle from a function name string.
`save`	Save a function handle from the current workspace to a MAT-file.
`load`	Load a function handle from a MAT-file into the current workspace.
`isa`	Determine if a variable contains a function handle.
`isequal`	Determine if two function handles are handles to the same function.

Comparing Function Handles

This section describes how MATLAB determines whether or not separate function handles are equal to each other:

Handles Constructed from a Named Function

Function handles that you construct from the same named function, e.g., `handle = @sin`, are considered by MATLAB to be equal. The `isequal` function returns a value of `true` when comparing these types of handles:

```
func1 = @sin;
func2 = @sin;
isequal(func1, func2)
 ans =
     1
```

If you save these handles to a MAT-file and then load them back into the workspace later on, they are still equal:

```
save temp1 func1
save temp2 func2
clear

load temp1
load temp2
isequal(func1, func2)
ans =
     1
```

Handles to Anonymous Functions

Unlike handles to named functions, any two function handles that represent the same anonymous function (i.e., handles to anonymous functions that contain the same text) are not equal. This is because MATLAB cannot guarantee that the frozen values of non-argument variables are the same.

```
q = 1;
a1 = @(x)q * x.^2;

q = 2;
a2 = @(x)q * x.^2;

isequal(a1, a2)
ans =
     0
```

This `false` result is accurate because `a1` and `a2` do indeed behave differently.

Note In general, MATLAB may underestimate the equality of function handles. That is, a test for equality may return `false` even when the functions happen to behave the same. But in cases where MATLAB does indicate equality, the functions are guaranteed to behave in an identical manner.

On the other hand, if you make a copy of an anonymous function handle, the copy and the original are equal:

```
h1 = @(x)sin(x);
h2 = h1;

isequal(h1, h2)
ans =
     1
```

In this case, function handles `h1` and `h2` are guaranteed to behave identically.

Handles to Nested Functions

Function handles to the same nested function are considered equal only if your code constructs these handles on the same call to the function containing the nested functions. Given this function that constructs two handles to the same nested function,

```
function [h1, h2] = test_eq(a, b, c)
h1 = @findZ;
h2 = @findZ;

   function z = findZ
   z = a.^3 + b.^2 + c';
   end
end
```

any two function handles constructed from the same nested function and on the same call to the parent function are equal:

```
[h1 h2] = test_eq(4, 19, -7);
```

```
isequal(h1, h2)
ans =
     1

[q1 q2] = test_eq(3, -1, 2);
isequal(q1, q2)
ans =
     1
```

The answer makes sense because `h1` and `h2` will always produce the same answer:

```
x = h1(),   y = h2()
x =
   418
y =
   418
```

However, handles constructed on different calls to the parent function are not equal:

```
isequal(h1, q1)
ans =
     0
```

In this case, `h1` and `q1` behave differently:

```
x = h1(),   y = q1()
x =
   418
y =
    30
```

Handles Saved to a MAT-File

If you save equivalent anonymous or nested function handles to separate MAT-files and then load them back into the MATLAB workspace, they are no longer equal. This is because saving the function handle in effect loses track of the original circumstances under which the function handle was created, and reloading it results in a function handle that compares as being unequal to the original function handle.

Create two equivalent anonymous function handles:

```
h1 = @(x) sin(x);
h2 = h1;

isequal(h1, h2)
ans =
     1
```

Save each to a different MAT-file:

```
save fname1 h1;
save fname2 h2;
```

Clear the MATLAB workspace and then load the function handles back into the workspace:

```
clear all
load fname1
load fname2
```

The function handles are no longer equal:

```
isequal(h1, h2)
ans =
     0
```

Note however that equal anonymous and nested function handles that you save to the same MAT-file are equal when loaded back into MATLAB:

```
h1 = @(x) sin(x);
h2 = h1;

isequal(h1, h2)
ans =
     1

save fname h1 h2;

clear all
load fname
```

```
isequal(h1, h2)
ans =
     1
```

Additional Information on Function Handles

This section covers the following topics:

Maximum Length of a Function Name

Function names used in handles are unique up to `N` characters, where `N` is the number returned by the function `namelengthmax`. If the function name exceeds that length, MATLAB truncates the latter part of the name.

For function handles created for Java constructors, the length of any segment of the package name or class name must not exceed `namelengthmax` characters. (The term *segment* refers to any portion of the name that lies before, between, or after a dot. For example, `java.lang.String` has three segments). The overall length of the string specifying the package and class has no limit.

How MATLAB Constructs a Function Handle

At the time you create a function handle, MATLAB maps the handle to one or more implementations of the function specified in the constructor statement:

```
fhandle = @functionname
```

In selecting which function(s) to map to, MATLAB considers

- Scope — The function named must be on the MATLAB path at the time the handle is constructed.
- Precedence — MATLAB selects which function(s) to map to according to the function precedence rules described under "How MATLAB Determines Which Method to Call" on page 9-72.

- Overloading — If additional M-files on the path overload the function for any of the standard MATLAB data types, such as `double` or `char`, then MATLAB maps the handle to these M-files as well.

M-files that overload a function for classes outside of the standard MATLAB data types are not mapped to the function handle at the time it is constructed. Function handles do operate on these types of overloaded functions, but MATLAB determines which implementation to call at the time of evaluation in this case.

Saving and Loading Function Handles

You can save and load function handles in a MAT-file using the MATLAB `save` and `load` functions. If you load a function handle that you saved in an earlier MATLAB session, the following conditions could cause unexpected behavior:

- Any of the M-files that define the function have been moved, and thus no longer exist on the path stored in the handle.
- You load the function handle into an environment different from that in which it was saved. For example, the source for the function either doesn't exist or is located in a different directory than on the system on which the handle was saved.

In both of these cases, the function handle is now invalid because it no longer maps to any existing function code. Although the handle is invalid, MATLAB still performs the load successfully and without displaying a warning. Attempting to invoke the handle, however, results in an error.

Function Arguments

In this section...

Overview

When calling a function, the caller provides the function with any data it needs by passing the data in an argument list. Data that needs to be returned to the caller is passed back in a list of return values.

Semantically speaking, MATLAB always passes argument data by value. (Internally, MATLAB optimizes away any unnecessary copy operations.)

If you pass data to a function that then modifies this data, you will need to update your own copy of the data. You can do this by having the function return the updated value as an output argument.

Checking the Number of Input Arguments

The `nargin` and `nargout` functions enable you to determine how many input and output arguments a function is called with. You can then use conditional statements to perform different tasks depending on the number of arguments. For example,

```
function c = testarg1(a, b)
if (nargin == 1)
    c = a .^ 2;
elseif (nargin == 2)
    c = a + b;
end
```

Given a single input argument, this function squares the input value. Given two inputs, it adds them together.

Here is a more advanced example that finds the first token in a character string. A *token* is a set of characters delimited by white space or some other character. Given one input, the function assumes a default delimiter of white space; given two, it lets you specify another delimiter if desired. It also allows for two possible output argument lists:

```
function [token, remainder] = strtok(string, delimiters)
% Function requires at least one input argument
if nargin < 1
   error('Not enough input arguments.');
end
token = []; remainder = [];
len = length(string);
if len == 0
   return
end

% If one input, use white space delimiter
if (nargin == 1)
   delimiters = [9:13 32]; % White space characters
end
i = 1;

% Determine where nondelimiter characters begin
while (any(string(i) == delimiters))
   i = i + 1;
   if (i > len), return, end
end

% Find where token ends
start = i;
while (~any(string(i) == delimiters))
   i = i + 1;
   if (i > len), break, end
end
finish = i - 1;
token = string(start:finish);
```

```
% For two output arguments, count characters after
% first delimiter (remainder)
if (nargout == 2)
   remainder = string(finish+1:end);
end
```

The `strtok` function is a MATLAB M-file in the `strfun` directory.

Note The order in which output arguments appear in the function declaration line is important. The argument that the function returns in most cases appears first in the list. Additional, optional arguments are appended to the list.

Passing Variable Numbers of Arguments

The `varargin` and `varargout` functions let you pass any number of inputs or return any number of outputs to a function. This section describes how to use these functions and also covers

- "Unpacking varargin Contents" on page 4-35
- "Packing varargout Contents" on page 4-35
- "varargin and varargout in Argument Lists" on page 4-36

MATLAB packs all specified input arguments into a *cell array*, a special kind of MATLAB array that consists of cells instead of array elements. Each cell can hold any size or kind of data — one might hold a vector of numeric data, another in the same array might hold an array of string data, and so on. For output arguments, your function code must pack them into a cell array so that MATLAB can return the arguments to the caller.

Here is an example function that accepts any number of two-element vectors and draws a line to connect them:

```
function testvar(varargin)
for k = 1:length(varargin)
   x(k) = varargin{k}(1); % Cell array indexing
   y(k) = varargin{k}(2);
```

```
end
xmin = min(0,min(x));
ymin = min(0,min(y));
axis([xmin fix(max(x))+3 ymin fix(max(y))+3])
plot(x,y)
```

Coded this way, the `testvar` function works with various input lists; for example,

```
testvar([2 3],[1 5],[4 8],[6 5],[4 2],[2 3])
testvar([-1 0],[3 -5],[4 2],[1 1])
```

Unpacking varargin Contents

Because `varargin` contains all the input arguments in a cell array, it's necessary to use cell array indexing to extract the data. For example,

```
y(n) = varargin{n}(2);
```

Cell array indexing has two subscript components:

- The indices within curly braces `{}` specify which cell to get the contents of.
- The indices within parentheses `()` specify a particular element of that cell.

In the preceding code, the indexing expression `{i}` accesses the nth cell of `varargin`. The expression `(2)` represents the second element of the cell contents.

Packing varargout Contents

When allowing a variable number of output arguments, you must pack all of the output into the `varargout` cell array. Use `nargout` to determine how many output arguments the function is called with. For example, this code accepts a two-column input array, where the first column represents a set of `x` coordinates and the second represents `y` coordinates. It breaks the array into separate `[xi yi]` vectors that you can pass into the `testvar` function shown at the beginning of the section on "Passing Variable Numbers of Arguments" on page 4-34:

```
function [varargout] = testvar2(arrayin)
```

```
for k = 1:nargout
   varargout{k} = arrayin(k,:);   % Cell array assignment
end
```

The assignment statement inside the `for` loop uses cell array assignment syntax. The left side of the statement, the cell array, is indexed using curly braces to indicate that the data goes inside a cell. For complete information on cell array assignment, see "Cell Arrays" on page 2-93.

To call `testvar2`, type

```
a = [1 2; 3 4; 5 6; 7 8; 9 0];

[p1, p2, p3, p4, p5] = testvar2(a)
p1 =
     1     2
p2 =
     3     4
p3 =
     5     6
p4 =
     7     8
p5 =
     9     0
```

varargin and varargout in Argument Lists

`varargin` or `varargout` must appear last in the argument list, following any required input or output variables. That is, the function call must specify the required arguments first. For example, these function declaration lines show the correct placement of `varargin` and `varargout`:

```
function [out1,out2] = example1(a,b,varargin)
function [i,j,varargout] = example2(x1,y1,x2,y2,flag)
```

Parsing Inputs with inputParser

MATLAB provides a class called `inputParser` to handle the different types of arguments passed into an M-file function. Using `inputParser`, you create a schema that both represents and verifies the content of the entire list of

input arguments passed on a call to the function. When used in all of your code development, this schema offers a consistent and thorough means of managing and validating the input information.

This section covers the following topics

To illustrate how to use the inputParser class, the documentation in this section develops a new M-file program called `publish_ip`, (based on the MATLAB `publish` function). There are three calling syntaxes for this function:

```
publish_ip('script')
publish_ip('script', 'format')
publish_ip('script', options)
```

There is one required argument (`script`), one optional argument (`format`), and a number of optional arguments that are specified as parameter-value pairs (`options`).

Defining a Specification for Each Input Parameter

Most programs have a block of code toward the beginning that parses the values in the input argument list and checks these values against what is expected. The `inputParser` class provides the following methods with which you can specify what the inputs are and whether they are required, optional, or to be specified using the parameter-value syntax:

- `addRequired` — Add a required parameter to the schema
- `addOptional` — Add an optional parameter to the schema
- `addParamValue` — Add an optional parameter-value pair to the schema

Creating the inputParser Object. Call the class constructor for `inputParser` to create an instance of the class. This class instance, or object, gives you access to all of the methods and properties of the class.

Begin writing the example `publish_ip` M-file by entering the following two statements:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.
```

After calling the constructor, use the `addRequired`, `addOptional`, and `addParamValue` methods to add arguments to the schema.

Note The constructor and all methods and properties of the `inputParser` class are case sensitive.

Adding Arguments to the Schema. Add any required arguments to the schema using the `addRequired` method. This method takes two inputs: the name of the required parameter, and an optional handle to a function that validates the parameter:

```
addRequired(name, validator);
```

Put an `addRequired` statement at the end of your `publish_ip` code. The two arguments for `addRequired` in this case are the filename input, `script`, and a handle to a function that will validate the filename, `ischar`. After adding the `addRequired` statement, your `publish_ip` function should now look like this:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.

p.addRequired('script', @ischar);
```

Use the `addOptional` method to add any arguments that are not required. The syntax for `addOptional` is similar to that of `addRequired` except that you also need to specify a default value to be used whenever the optional argument is not passed:

```
addOptional(name, default, validator);
```

In this case, the `validator` input is a handle to an anonymous function:

```
p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));
```

Use `addParamValue` to specify any arguments that use a parameter-value format. The syntax is

```
addParamValue(name, default, validator);
```

For example,

```
p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

Listing the Arguments. At this point, the schema is complete. Here is the file `publish_ip.m`:

```
function x = publish_ip(script, varargin)
p = inputParser;   % Create an instance of the class.

p.addRequired('script', @ischar);

p.addOptional('format', 'html', ...
    @(x)any(strcmpi(x,{'html','ppt','xml','latex'})));p.

p.addParamValue('outputDir', pwd, @ischar);
p.addParamValue('maxHeight', [], @(x)x>0 && mod(x,1)==0);
p.addParamValue('maxWidth', [], @(x)x>0 && mod(x,1)==0);
```

When you call the program, MATLAB stores the name of each argument in the `Parameters` property of object `p`. Add the following two lines to your `publish_ip` M-file to display `p.Parameters`:

```
sprintf('%s\n  %s\n  %s\n  %s\n  %s\n  %s', ...
   'The input parameters for this program are:', ...
   p.Parameters{:})
```

Save the M-file, and then run it as shown here:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'maxHeight', 300);
```

The output is

```
The input parameters for this program are:
  format
  maxHeight
  maxWidth
  outputDir
  script
```

Parsing Parameter Values on the Function Call

Once you have constructed a schema that represents all possible inputs to the function, the next task is to write the code that parses and verifies these arguments whenever the function is called. The `parse` method of the `inputParser` class reads and validates the required `script` argument and any optional arguments that follow it in the argument list:

```
p.parse(script, varargin{:});
```

Execution of the parse method validates each argument and also builds a structure from the input arguments. The name of the structure is `Results`, which is accessible as a property of the object. To get the value of all arguments, type

```
p.Results
```

To get the value of any single input argument, type

```
p.Results.argname
```

where `argname` is the name of the argument. Continue with the `publish_ip` exercise started earlier in this section by removing the `sprintf` statement that was inserted in the last section, and then adding the following lines:

```
% Parse and validate all input arguments.
p.parse(script, varargin{:});

% Display the value of a specific argument.
disp(' ')
disp(sprintf('\nThe maximum height is %d.', ...
     p.Results.maxHeight))

% Display all arguments.
disp(' ')
disp 'List of all arguments:'
disp(p.Results)
```

Now save and execute the M-file, passing the required script argument, the optional format argument, as well as several parameter-value arguments. MATLAB assigns those values you pass in the argument list to the appropriate fields of the `Results` structure:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'maxHeight', 300);

The maximum height is 300.

List of all arguments:
              format: 'ppt'
           maxHeight: 300
            maxWidth: 500
           outputDir: 'C:/matlab/test'
              script: 'ipscript.m'
```

Packaging Arguments in a Structure

By setting the `StructExpand` property of the `inputParser` object to `true`, you can pass arguments to your function in the form of a structure instead of individually in the argument list. This property must be set prior to calling the `parse` method.

`StructExpand` defaults to the `true` state, so you don't have to make any changes to your test program to run this example.

Put all of the input arguments into a structure:

```
s.format = 'xml';
s.outputDir = 'C:/matlab/test';
s.maxWidth = 200;
s.maxHeight = 150;
```

Now call the function, passing the filename and input structure:

```
publish_ip('ipscript.m', s);

The maximum height is 150.

List of all arguments:
              format: 'xml'
           maxHeight: 150
            maxWidth: 200
           outputDir: 'C:/matlab/test'
              script: 'ipscript.m'
```

To disable struct expansion, include the following statement somewhere in your program code before the `p.parse` statement:

```
p.StructExpand = false;
```

Overriding the Input Structure. If you want to pass your argument list in a structure, as described in the previous section, but you also want to alter the value of one or more of these arguments without having to modify the structure, you can do so by passing both the structure and the modified argument:

```
publish_ip('ipscript.m', s, ...
           'outputDir', 'C:/matlab/R2007a/temp');

List of all arguments:
              format: 'xml'
           maxHeight: 150
            maxWidth: 200
           outputDir: 'C:/matlab/R2007a/temp'
              script: 'ipscript.m'
```

Arguments That Default

Any arguments that you do not include in a call to your function are given their default values by MATLAB. You defined these default values when you created your schema using the `addOptional` and `addParamValue` methods. The `UsingDefaults` property is actually a structure that contains the names of any arguments that were not passed in the function call, and thus were assigned default values.

Add the following to your M-file:

```
% Show which arguments were not specified in the call.
disp(' ')
disp 'List of arguments given default values:'
for k=1:numel(p.UsingDefaults)
   field = char(p.UsingDefaults(k));
   value = p.Results.(field);
   if isempty(value),   value = '[]';   end
   disp(sprintf('   ''%s''    defaults to %s', field, value))
end
```

Save the M-file and run it without specifying the `format`, `outputDir`, or `maxHeight` arguments:

```
publish_ip('ipscript.m', 'maxWidth', 500);

List of arguments given default values:
   'format'    defaults to html
   'outputDir'    defaults to D:\work_r14
   'maxHeight'    defaults to []
```

Validating the Input Arguments

When you call your function, MATLAB checks any arguments for which you have specified a validator function. If the validator finds an error, MATLAB displays an error message and aborts the function. In the `publish` function example, the `outputDir` argument validates the value passed in using `@ischar`.

Pass a number instead of a string for the `outputDir` argument:

```
publish_ip('ipscript.m', 'outputDir', 5);
??? Argument 'outputDir' failed validation ischar.

Error in ==> publish_ip at 14
p.parse(varargin{:});
```

Handling Unmatched Arguments. MATLAB throws an error if you call your function with any arguments that are not part of the `inputParser` schema. You can disable this error by setting the `KeepUnmatched` property to `true`. When `KeepUnmatched` is in the `true` state, MATLAB does not throw an error, but instead stores any arguments that are not in the schema in a cell array of strings accessible through the `Unmatched` property of the object. `KeepUnmatched` defaults to `false`.

At some point in your `publish_ip` M-file before executing the `parse` method, set the `KeepUnmatched` property to `true`, and following the `parse` statement, examine the `Unmatched` property:

```
p.KeepUnmatched = true;

% Parse and validate all input arguments.
p.parse(script, varargin{:});

disp(' ')
disp 'List of unmatched arguments:'
p.Unmatched
```

Save and run the function, passing two arguments that are not defined in the schema:

```
publish_ip('ipscript.m', s, ...
           'outputDir', 'C:/matlab/R2007a/temp', ...
           'colorSpace', 'CMYK', 'density', 200);

List of unmatched arguments:
    colorSpace: 'CMYK'
       density: 200
```

Enabling Case-Sensitive Matching. When you pass optional arguments in the function call, MATLAB compares these arguments with the names of parameter-value argument names in the schema. By default, MATLAB does not use case sensitivity in this comparison. So, an argument name entered into the schema (using `addParamValue`) as `maxHeight` will match an argument passed as `MAXHeight` in the function call. You can override the default and make these comparisons case sensitive by setting the `CaseSensitive` property of the object to `true`. MATLAB does not error on a case mismatch, however, unless the `KeepUnmatched` property is set to `false`: its default state.

At some point in your `publish_ip` M-file before executing the `parse` method, set `KeepUnmatched` to `false` and `CaseSensitive` to `true`, and then execute the `publish_ip` function using `MAXHeight` as the name of the argument for specifying maximum height:

```
p.KeepUnmatched = false;
p.CaseSensitive = true;

% Parse and validate all input arguments.
p.parse(script, varargin{:});
```

Save and run the function, using `MAXHeight` as the name of the argument for specifying maximum height:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', ...
    'C:/matlab/test', 'maxWidth', 500, 'MAXHeight', 300);
??? Argument 'MAXHeight' did not match any valid parameter of
the parser.

Error in ==> publish_ip at 17
```

Adding the Function Name to Error Messages. Use the `FunctionName` property to include the name of your function in error messages thrown by the validating function:

At some point in your `publish_ip` M-file before executing the `parse` method, set the `FunctionName` property to `PUBLISH_IP`, and then run the function:

```
p.FunctionName = 'PUBLISH_IP';

% Parse and validate all input arguments.
```

```
p.parse(script, varargin{:});
```

Save and run the function and observe text of the error message:

```
publish_ip('ipscript.m', 'ppt', 'outputDir', 5, ...
    'maxWidth', 500, 'maxHeight', 300);
??? Argument 'outputDir' failed validation ischar in PUBLISH_IP.
```

Making a Copy of the Schema

The `createCopy` method enables you to make a copy of an existing schema. Because the `inputParser` class uses handle semantics, you cannot make a copy of the object using an assignment statement.

The following statement creates an `inputParser` object `s` that is a copy of `p`:

```
s = p.createCopy
```

Summary of inputParser Methods

Method	Description
`addOptional`	Add an optional argument to the schema
`addParamValue`	Add a parameter-value pair argument to the schema
`addRequired`	Add a required argument to the schema
`createCopy`	Create a copy of the `inputParser` object
`parse`	Parse and validate the named inputs

Summary of inputParser Properties that Control Parsing

Property	Description
`CaseSensitivity`	Enable or disable case-sensitive matching of argument names. Defaults to `false`.

Property	Description
`FunctionName`	Function name to be included in error messages. Defaults to an empty string.
`KeepUnmatched`	Enable or disable errors on unmatched arguments. Defaults to `false`.
`StructExpand`	Enable or disable passing arguments in a structure. Defaults to `true`.

Summary of inputParser Properties that Provide Information

Property	Description
`Parameters`	Names of arguments defined in `inputParser` schema.
`Results`	Names and values of arguments passed in function call that are in the schema for this function.
`Unmatched`	Names and values of arguments passed in function call that are not in the schema for this function.
`UsingDefaults`	Names of arguments not passed in function call that are given default values.

Passing Optional Arguments to Nested Functions

You can use optional input and output arguments with nested functions, but you should be aware of how MATLAB interprets `varargin`, `varargout`, `nargin`, and `nargout` under those circumstances.

`varargin` and `varargout` are variables and, as such, they follow exactly the same scoping rules as any other MATLAB variable. Because nested functions share the workspaces of all outer functions, `varargin` and `varargout` used in a nested function can refer to optional arguments passed to or from the nested function, or passed to or from one of its outer functions.

nargin and nargout, on the other hand, are functions and when called within a nested function, always return the number of arguments passed to or from the nested function itself.

Using varargin and varargout

varargin or varargout used in a nested function can refer to optional arguments passed to or from that function, or to optional arguments passed to or from an outer function.

- If a nested function includes varargin or varargout in its function declaration line, then the use of varargin or varargout within that function returns optional arguments passed to or from that function.
- If varargin or varargout are not in the nested function declaration but are in the declaration of an outer function, then the use of varargin or varargout within the nested function returns optional arguments passed to the outer function.

In the example below, function C is nested within function B, and function B is nested within function A. The term varargin{1} in function B refers to the second input passed to the primary function A, while varargin{1} in function C refers to the first argument, z, passed from function B:

```
function x = A(y, varargin)   % Primary function A
B(nargin, y * rand(4))

   function B(argsIn, z)      % Nested function B
   if argsIn >= 2
      C(z, varargin{1}, 4.512, 1.729)
   end

      function C(varargin)    % Nested function C
      if nargin >= 2
         x = varargin{1}
      end
      end   % End nested function C
   end   % End nested function B
end   % End primary function A
```

Using nargin and nargout

When `nargin` or `nargout` appears in a nested function, it refers to the number of inputs or outputs passed to that particular function, regardless of whether or not it is nested.

In the example shown above, `nargin` in function `A` is the number of inputs passed to `A`, and `nargin` in function `C` is the number of inputs passed to `C`. If a nested function needs the value of `nargin` or `nargout` from an outer function, you can pass this value in as a separate argument, as done in function `B`.

Example of Passing Optional Arguments to Nested Functions

This example references the primary function's `varargin` cell array from each of two nested functions. (Because the workspace of an outer function is shared with all functions nested within it, there is no need to pass `varargin` to the nested functions.)

Both nested functions make use of the `nargin` value that applies to the primary function. Calling `nargin` from the nested function would return the number of inputs passed to that nested function, and not those that had been passed to the primary. For this reason, the primary function must pass its `nargin` value to the nested functions.

```
function meters = convert2meters(miles, varargin)
% Converts MILES (plus optional FEET and INCHES input)
% values to METERS.

if nargin < 1 || nargin > 3
   error('1 to 3 input arguments are required');
end

   function feet = convert2Feet(argsIn)
   % Nested function that converts miles to feet and adds in
   % optional FEET argument.

   feet = miles .* 5280;

   if argsIn >= 2
      feet = feet + varargin{1};
   end
```

```
    end  % End nested function convert2Feet

    function inches = convert2Inches(argsIn)
    % Nested function that converts feet to inches and adds in
    % optional INCHES argument.

    inches = feet .* 12;

    if argsIn == 3
       inches = inches + varargin{2};
    end
    end  % End nested function convert2Inches

feet = convert2Feet(nargin);
inches = convert2Inches(nargin);

meters = inches .* 2.54 ./ 100;
end  % End primary function convert2meters

convert2meters(5)
ans =
  8.0467e+003

convert2meters(5, 2000, 4.7)
ans =
  8.6564e+003
```

Returning Modified Input Arguments

If you pass any input variables that the function can modify, you will need to include the same variables as output arguments so that the caller receives the updated value.

For example, if the function `readText`, shown below, reads one line of a file each time is it called, then it must keep track of the offset into the file. But when `readText` terminates, its copy of the `offset` variable is cleared from memory. To keep the offset value from being lost, `readText` must return this value to the caller:

```
function [text, offset] = readText(filestart, offset)
```

Calling Functions

In this section...

What Happens When You Call a Function

When you call a function M-file from either the command line or from within another M-file, MATLAB parses the function into pseudocode and stores it in memory. This prevents MATLAB from having to reparse a function each time you call it during a session. The pseudocode remains in memory until you clear it using the `clear` function, or until you quit MATLAB.

Clearing Functions from Memory

You can use `clear` in any of the following ways to remove functions from the MATLAB workspace.

Syntax	Description
`clear functionname`	Remove specified function from workspace.
`clear functions`	Remove all compiled M-functions.
`clear all`	Remove all variables and functions.

Determining Which Function Is Called

When more than one function has the same name, which one does MATLAB call? This section explains the process that MATLAB uses to make this decision. It covers the following topics:

Also keep in mind that there are certain situations in which function names can conflict with *variables* of the same name. See "Potential Conflict with Function Names" on page 3-7for more information.

Function Scope

Any functions you call must first be within the scope of (i.e., visible to) the calling function or your MATLAB session. MATLAB determines if a function is in scope by searching for the function's executable file according to a certain order (see "Precedence Order" on page 4-53).

One key part of this search order is the MATLAB path. The path is an ordered list of directories that MATLAB defines on startup. You can add or remove any directories you want from the path. MATLAB searches the path for the given function name, starting at the first directory in the path string and continuing until either the function file is found or the list of directories is exhausted. If no function of that name is found, then the function is considered to be out of scope and MATLAB issues an error.

Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. MATLAB selects the correct function for a given context by applying the following function precedence rules in the order given here.

For items 3 through 7 in this list, the file MATLAB searches for can be any of four types: an M- or built-in file, preparsed M-file (P-Code), compiled

C or Fortran file (MEX-file), or Simulink® model (MDL-file). See "Multiple Implementation Types" on page 4-55 for more on this.

1. Variable

   Before assuming that a name should match a function, MATLAB checks the current workspace to see if it matches a variable name. If MATLAB finds a match, it stops the search.

2. Subfunction

   Subfunctions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the subfunction and ignores the overloaded method.

3. Private function

   Private functions are called if there is no subfunction of the same name within the current scope. As with subfunctions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method.

4. Class constructor

   Constructor functions (functions having names that are the same as the @ directory, for example `@polynom/polynom.m`) take precedence over other MATLAB functions. Therefore, if you create an M-file called `polynom.m` and put it on your path before the constructor `@polynom/polynom.m` version, MATLAB will always call the constructor version.

5. Overloaded method

   MATLAB calls an overloaded method if it is not superseded by a subfunction or private function. Which overloaded method gets called depends on the classes of the objects passed in the argument list.

6. Function in the current directory

   A function in the current working directory is selected before one elsewhere on the path.

7 Function elsewhere on the path

   Finally, a function elsewhere on the path is selected. A function in a directory that is toward the beginning of the path string is given higher precedence.

Note Because variables have the highest precedence, if you have created a variable of the same name as a function, MATLAB will not be able to run that function until you clear the variable from memory.

Multiple Implementation Types

There are five file precedence types. MATLAB uses file precedence to select between identically named functions in the same directory. The order of precedence for file types is

1 Built-in file

2 MEX-files

3 MDL (Simulink® model) file

4 P-code file

5 M-file

For example, if MATLAB finds a P-code and an M-file version of a method in a class directory, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the M-file.

Querying Which Function MATLAB Will Call

You can determine which function MATLAB will call using the `which` command. For example,

```
which pie3
matlabroot/toolbox/matlab/specgraph/pie3.m
```

However, if `p` is a portfolio object,

```
which pie3(p)
dir_on_your_path/@portfolio/pie3.m      % portfolio method
```

The `which` command determines which version of `pie3` MATLAB will call if you passed a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use the `-all` option. See the `which` reference page for more information on this command.

MATLAB Calling Syntax

This section explains how to use the MATLAB command and function syntax:

You can call function M-files from either the MATLAB command line or from within other M-files. Be sure to include all necessary arguments, enclosing input arguments in parentheses and output arguments in square brackets.

Note Function names are sensitive to case. When you call a function, use the correct combination of upper and lowercase letters so that the name is an exact match. Otherwise, you risk calling a different function that does match but is elsewhere on the path.

You often have the choice of using one of two syntaxes for a function call. You can use either a command or a function type of syntax. This is referred to in MATLAB as *command/function duality*.

MATLAB Command Syntax

A function call made in command syntax consists of the function name followed by one or more arguments separated by spaces:

```
functionname in1 in2 ... inN
```

While the command syntax is simpler to write, it has the restriction that you may not assign any return values the function might generate. Attempting to do so generates an error.

Two examples of command syntax are

```
save mydata.mat x y z
clear length width depth
```

In the command syntax, MATLAB treats all arguments as string literals.

MATLAB Function Syntax

Function calls written in the function syntax look essentially the same as those in many other programming languages. One difference is that, in MATLAB, functions can return more than one output value.

A function call with a single return value looks like this:

```
out = functionname(in1, in2, ..., inN)
```

If the function returns more than one value, separate the output variables with commas or spaces, and enclose them all in square brackets (`[]`):

```
[out1, out2, ..., outN] = functionname(in1, in2, ..., inN)
```

Here are two examples:

```
copyfile(srcfile, '..\mytests', 'writable')
[x1, x2, x3, x4] = deal(A{:})
```

In the function syntax, MATLAB passes arguments to the function by value. See the examples under “Passing Arguments with Command and Function Syntax” on page 4-57.

Passing Arguments with Command and Function Syntax

When you call a function using function syntax, MATLAB passes the *values* assigned to each variable in the argument list. For example, this expression passes the values assigned to `A0`, `A1`, and `A2` to the `polyeig` function:

```
e = polyeig(A0, A1, A2)
```

Function calls written in command syntax pass all arguments as string literals. This expression passes the strings `'mydata.mat'`, `'x'`, `'y'`, and `'z'` to the `save` function:

```
save mydata.mat x y z
```

The following examples show the difference between passing arguments in the two syntaxes.

Passing Arguments — Example 1. Calling `disp` with the function syntax, `disp(A)`, passes the value of variable `A` to the `disp` function:

```
A = pi;

disp(A)                     % Function syntax
   3.1416
```

Calling it with the command syntax, `disp A`, passes the string `'A'`:

```
A = pi;

disp A                      % Command syntax
   A
```

Passing Arguments — Example 2. Passing two variables representing equal strings to the `strcmp` function using function and command syntaxes gives different results. The function syntax passes the values of the arguments. `strcmp` returns a `1`, which means they are equal:

```
str1 = 'one';       str2 = 'one';

strcmp(str1, str2)                  % Function syntax
ans =
   1        (equal)
```

The command syntax passes the names of the variables, `'str1'` and `'str2'`, which are unequal:

```
str1 = 'one';       str2 = 'one';
```

```
strcmp str1 str2                    % Command syntax
ans =
   0        (unequal)
```

How MATLAB Recognizes Function Calls That Use Command Syntax

It can be difficult to tell whether a MATLAB expression is a function call using command syntax or another kind of expression, such as an operation on one or more variables. Consider the following example:

```
ls ./d
```

Is this a call to the `ls` function with the directory `./d` as its argument? Or is it a request to perform elementwise division on the array that is the value of the `ls` variable, using the value of the `d` variable as the divisor?

This example might appear unambiguous because MATLAB can determine whether `ls` and `d` are functions or variables. But that is not always true. Some MATLAB components, such as M-Lint and the Editor/Debugger, must operate without reference to the MATLAB path or workspace. MATLAB therefore uses syntactic rules to determine when an expression is a function call using command syntax.

The rules are complicated and have exceptions. In general, when MATLAB recognizes an identifier (which might name a function or a variable), it analyzes the characters that follow the identifier to determine what kind of expression exists. The expression is usually a function call using command syntax when all of the following are true:

1. The identifier is followed immediately by white space.
2. The characters following the white space are not parentheses or an assignment operator.
3. The characters following the white space are not an operator that is itself followed by additional white space and then by characters that can legitimately follow an operator.

The example above meets all three criteria and is therefore a function call using command syntax:

```
ls ./d
```

The following examples are not function calls using command syntax:

```
% No white space following the ls identifier
% Interpretation: elementwise division
ls./d

% Parenthesis following white space
% Interpretation: function call using function syntax
ls ('./d')

% Assignment operator following white space
% Interpretation: assignment to a variable
ls =d

% Operator following white space, followed in turn by
% more white space and a variable
% Interpretation: elementwise division
ls ./ d
```

Passing Certain Argument Types

This section explains how to pass the following types of data in a function call:

- "Passing Strings" on page 4-60
- "Passing Filenames" on page 4-61
- "Passing Function Handles" on page 4-62

Passing Strings

When using the function syntax to pass a string literal to a function, you must enclose the string in single quotes, (`'string'`). For example, to create a new directory called `myapptests`, use

```
mkdir('myapptests')
```

On the other hand, variables that contain strings do not need to be enclosed in quotes:

```
dirname = 'myapptests';
mkdir(dirname)
```

Passing Filenames

You can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`):

```
load mydata.mat           % Command syntax
load('mydata.mat')        % Function syntax
```

If you assign the output to a variable, you must use the function syntax:

```
savedData = load('mydata.mat')
```

Specify ASCII files as shown here. In this case, the file extension is required:

```
load mydata.dat -ascii            % Command syntax
load('mydata.dat','-ascii')       % Function syntax
```

Determining Filenames at Run-Time. There are several ways that your function code can work on specific files without your having to hardcode their filenames into the program. You can

- Pass the filename as an argument:

  ```
  function myfun(datafile)
  ```

- Prompt for the filename using the `input` function:

  ```
  filename = input('Enter name of file:  ', 's');
  ```

- Browse for the file using the `uigetfile` function:

  ```
  [filename, pathname] = uigetfile('*.mat', 'Select MAT-file');
  ```

Passing Function Handles

The MATLAB function handle has several uses, the most common being a means of immediate access to the function it represents. You can pass function handles in argument lists to other functions, enabling the receiving function to make calls by means of the handle.

To pass a function handle, include its variable name in the argument list of the call:

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

The receiving function invokes the function being passed using the usual MATLAB calling syntax:

```
function [xf, fval, exitflag, output] = ...
         fminbnd(fhandle, ax, bx, options, varargin)
            .
            .
            .
113  fx = fhandle(x, varargin{:});
```

Passing Arguments in Structures or Cell Arrays

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure or cell array.

Passing Arguments in a Structure

Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields. Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

This example updates weather statistics from information in the following chart.

City	Temp.	Heat Index	Wind Speed	Wind Chill
Boston	43	32	8	37
Chicago	34	27	3	30
Lincoln	25	17	11	16
Denver	15	-5	9	0
Las Vegas	31	22	4	35
San Francisco	52	47	18	42

The information is stored in structure `W`. The structure has one field for each column of data:

```
W = struct('city', {'Bos','Chi','Lin','Dnv','Vgs','SFr'}, ...
           'temp', {43, 34, 25, 15, 31, 52}, ...
           'heatix', {32, 27, 17, -5, 22, 47}, ...
           'wspeed', {8, 3, 11, 9, 4, 18}, ...
           'wchill', {37, 30, 16, 0, 35, 42});
```

To update the data base, you can pass the entire structure, or just one field with its associated data. In the call shown here, `W.wchill` is a comma-separated list:

```
updateStats(W.wchill);
```

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The advantage over structures is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function. The disadvantage is that you don't have field names to describe each variable.

This example passes several attribute-value arguments to the `plot` function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));

C{1,1} = 'LineWidth';         C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';   C{2,2} = 'k';
```

```
C{1,3} = 'MarkerFaceColor';   C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Assigning Output Arguments

Use the syntax shown here to store any values that are returned by the function you are calling. To store one output, put the variable that is to hold that output to the left of the equal sign:

```
vout = myfun(vin1, vin2, ...);
```

To store more than one output, list the output variables inside square brackets and separate them with commas or spaces:

```
[vout1 vout2 ...] = myfun(vin1, vin2, ...);
```

The number of output variables in your function call statement does not have to match the number of return values declared in the function being called. For a function that declares N return values, you can specify anywhere from zero to N output variables in the call statement. Any return values that you do not have an output variable for are discarded.

Functions return output values in the order in which the corresponding output variables appear in the function definition line within the M-file. This function returns 100 first, then `x * y`, and lastly `x.^2`:

```
function [a b c] = myfun(x, y)
b = x * y;    a = 100;    c = x.^2;
```

If called with only one output variable in the call statement, the function returns only 100 and discards the values of `b` and `c`. If called with no outputs, the functions returns 100 in the MATLAB default variable `ans`.

Assigning Optional Return Values

The section "Passing Variable Numbers of Arguments" on page 4-34 describes the method of returning optional outputs in a cell array called `varargout`. A function that uses `varargout` to return optional values has a function definition line that looks like one of the following:

```
function varargout = myfun(vin1, vin2, ...)
```

```
function [vout1 vout2 ... varargout] = myfun(vin1, vin2, ...)
```

The code within the function builds the `varargout` cell array. The content and order of elements in the cell array determines how MATLAB assigns optional return values to output variables in the function call.

In the case where `varargout` is the only variable shown to the left of the equal sign in the function definition line, MATLAB assigns `varargout{1}` to the first output variable, `varargout{2}` to the second, and so on. If there are other outputs declared in the function definition line, then MATLAB assigns those outputs to the leftmost output variables in the call statement, and then assigns outputs taken from the `varargout` array to the remaining output variables in the order just described.

This function builds the `varargout` array using descending rows of a 5-by-5 matrix. The function is capable of returning up to six outputs:

```
function varargout = byRow(a)
varargout{1} = '    With VARARGOUT constructed by row ...';
for k = 1:5
    row = 5 - (k-1);              % Reverse row order
    varargout{k+1} = a(row,:);
end
```

Call the function, assigning outputs to four variables. MATLAB returns `varargout{1:4}`, with rows of the matrix in `varargout{2:4}` and in the order in which they were stored by the function:

```
[text r1 r2 r3] = byRow(magic(5))
text =
    With VARARGOUT constructed by row ...
r1 =
    11    18    25     2     9
r2 =
    10    12    19    21     3
r3 =
     4     6    13    20    22
```

A similar function builds the `varargout` array using diagonals of a 5-by-5 matrix:

```
function varargout = byDiag(a)
varargout{1} = '    With VARARGOUT constructed by diagonal ...';
for k = -4:4
    varargout{k + 6} = diag(a, k);
end
```

Call the function with five output variables. Again, MATLAB assigns elements of `varargout` according to the manner in which it was constructed within the function:

```
[text d1 d2 d3 d4] = byDiag(magic(5))
text =
    With VARARGOUT constructed by diagonal ...
d1 =
    11
d2 =
    10
    18
d3 =
     4
    12
    25
d4 =
    23
     6
    19
     2
```

Calling External Functions

The MATLAB external interface offers a number of ways to run external functions from MATLAB. This includes programs written in C or Fortran, methods invoked on Java or COM (Component Object Model) objects, functions that interface with serial port hardware, and functions stored in shared libraries. The *MATLAB External Interfaces* documentation describes these various interfaces and how to call these external functions.

Running External Programs

For information on how to invoke operating systems commands or execute programs that are external to MATLAB, see Running External Programs in the MATLAB Desktop Tools and Development documentation.

5

Types of Functions

Overview of MATLAB Function Types

There are essentially two ways to create a new function in MATLAB: in a command entered at run-time, or in a file saved to permanent storage.

The command-oriented function, called an *anonymous function*, is relatively brief in its content. It consists of a single MATLAB statement that can interact with multiple input and output arguments. The benefit of using anonymous functions is that you do not have to edit and maintain a file for functions that require only a brief definition.

There are several types of functions that are stored in files (called M-files). The most basic of these are *primary functions* and *subfunctions*. Primary functions are visible to other functions outside of their M-file, while subfunctions, generally speaking, are not. That is, you can call a primary function from an anonymous function or from a function defined in a separate M-file, whereas you can call a subfunction only from functions within the same M-file. (See the Description section of the `function_handle` reference page for information on making a subfunction externally visible.)

Two specific types of primary M-file functions are the *private* and *overloaded function*. Private functions are visible only to a limited group of other functions. This type of function can be useful if you want to limit access to a function, or when you choose not to expose the implementation of a function. Overloaded functions act the same way as overloaded functions in most computer languages. You can create multiple implementations of a function so that each responds accordingly to different types of inputs.

The last type of MATLAB function is the *nested function*. Nested functions are not an independent function type; they exist within the body of one of the other types of functions discussed here (with the exception of anonymous functions), and also within other nested functions.

One type of function that is not discussed in this chapter is the MATLAB built-in function. Built-ins are defined only as executables internal to MATLAB. See “Built-In Functions” on page 3-109 for more information.

Anonymous Functions

In this section...

Constructing an Anonymous Function

Anonymous functions give you a quick means of creating simple functions without having to create M-files each time. You can construct an anonymous function either at the MATLAB command line or in any M-file function or script.

The syntax for creating an anonymous function from an expression is

```
fhandle = @(arglist) expr
```

Starting from the right of this syntax statement, the term `expr` represents the body of the function: the code that performs the main task your function is to accomplish. This consists of any single, valid MATLAB expression. Next is `arglist`, which is a comma-separated list of input arguments to be passed to the function. These two components are similar to the body and argument list components of any function.

Leading off the entire right side of this statement is an `@` sign. The `@` sign is the MATLAB operator that constructs a function handle. Creating a function handle for an anonymous function gives you a means of invoking the function. It is also useful when you want to pass your anonymous function in a call to some other function. The `@` sign is a required part of an anonymous function definition.

Note Function handles not only provide access to anonymous functions. You can create a function handle to any MATLAB function. The constructor uses a different syntax: `fhandle = @functionname` (e.g., `fhandle = @sin`). To find out more about function handles, see "Function Handles" on page 4-22.

The syntax statement shown above constructs the anonymous function, returns a handle to this function, and stores the value of the handle in variable `fhandle`. You can use this function handle in the same way as any other MATLAB function handle.

Simple Example

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable `x`, and then uses `x` in the equation `x.^2`:

```
sqr = @(x) x.^2;
```

The `@` operator constructs a function handle for this function, and assigns the handle to the output variable `sqr`. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the `sqr` function defined above, type

```
a = sqr(5)
a =
   25
```

Because `sqr` is a function handle, you can pass it in an argument list to other functions. The code shown here passes the `sqr` anonymous function to the MATLAB `quad` function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
   0.3333
```

A Two-Input Example

As another example, you could create the following anonymous function that uses two input arguments, `x` and `y`. (The example assumes that variables `A` and `B` are already defined):

```
sumAxBy = @(x, y) (A*x + B*y);

whos sumAxBy
Name        Size                    Bytes  Class

sumAxBy     1x1                        16  function_handle
```

To call this function, assigning `5` to `x` and `7` to `y`, type

```
sumAxBy(5, 7)
```

Evaluating With No Input Arguments

For anonymous functions that do not take any input arguments, construct the function using empty parentheses for the input argument list:

```
t = @() datestr(now);
```

Also use empty parentheses when invoking the function:

```
t()

ans =
04-Sep-2003 10:17:59
```

You must include the parentheses. If you type the function handle name with no parentheses, MATLAB just identifies the handle; it does not execute the related function:

```
t

t =
    @() datestr(now)
```

Arrays of Anonymous Functions

To store multiple anonymous functions in an array, use a cell array. The example shown here stores three simple anonymous functions in cell array A:

```
A = {@(x)x.^2, @(y)y+10, @(x,y)x.^2+y+10}
A =
    [@(x)x.^2]    [@(y)y+10]    [@(x,y)x.^2+y+10]
```

Execute the first two functions in the cell array by referring to them with the usual cell array syntax, A{1} and A{2}:

```
A{1}(4) + A{2}(7)
ans =
    33
```

Do the same with the third anonymous function that takes two input arguments:

```
A{3}(4, 7)
ans =
    33
```

Space Characters in Anonymous Function Elements

Note that while using space characters in the definition of any function can make your code easier to read, spaces in the body of an anonymous function that is defined in a cell array can sometimes be ambiguous to MATLAB. To ensure accurate interpretation of anonymous functions in cell arrays, you can do any of the following:

- Remove all spaces from at least the body (not necessarily the argument list) of each anonymous function:

  ```
  A = {@(x)x.^2, @(y)y+10, @(x, y)x.^2+y+10};
  ```

- Enclose in parentheses any anonymous functions that include spaces:

  ```
  A = {(@(x)x .^ 2), (@(y) y +10), (@(x, y) x.^2 + y+10)};
  ```

- Assign each anonymous function to a variable, and use these variable names in creating the cell array:

```
A1 = @(x)x .^ 2;  A2 = @(y) y +10;  A3 = @(x, y)x.^2 + y+10;
A = {A1, A2, A3};
```

Outputs from Anonymous Functions

As with other MATLAB functions, the number of outputs returned by an anonymous function depends mainly on how many variables you specify to the left of the equals (=) sign when you call the function.

For example, consider an anonymous function `getPersInfo` that returns a person's address, home phone, business phone, and date of birth, in that order. To get someone's address, you can call the function specifying just one output:

```
address = getPersInfo(name);
```

To get more information, specify more outputs:

```
[address, homePhone, busPhone] = getPersInfo(name);
```

Of course, you cannot specify more outputs than the maximum number generated by the function, which is four in this case.

Example

The anonymous `getXLSData` function shown here calls the MATLAB `xlsread` function with a preset spreadsheet filename (`records.xls`) and a variable worksheet name (`worksheet`):

```
getXLSData = @(worksheet) xlsread('records.xls', worksheet);
```

The `records.xls` worksheet used in this example contains both numeric and text data. The numeric data is taken from instrument readings, and the text data describes the category that each numeric reading belongs to.

Because the MATLAB `xlsread` function is defined to return up to three values (numeric, text, and raw data), `getXLSData` can also return this same number of values, depending on how many output variables you specify to the left of the equals sign in the call. Call `getXLSData` a first time, specifying only a single (numeric) output, `dNum`:

```
dNum = getXLSData('Week 12');
```

Display the data that is returned using a `for` loop. You have to use generic names (`v1`, `v2`, `v3`) for the categories, due to the fact that the text of the real category names was not returned in the call:

```
for k = 1:length(dNum)
   disp(sprintf('%s    v1: %2.2f    v2: %d    v3: %d', ...
      datestr(clock, 'HH:MM'), dNum(k,1), dNum(k,2), ...
      dNum(k,3)));
   end
```

Here is the output from the first call:

```
12:55   v1: 78.42    v2: 32    v3: 37
13:41   v1: 69.73    v2: 27    v3: 30
14:26   v1: 77.65    v2: 17    v3: 16
15:10   v1: 68.19    v2: 22    v3: 35
```

Now try this again, but this time specifying two outputs, numeric (`dNum`) and text (`dTxt`):

```
[dNum, dTxt] = getXLSData('Week 12');

for k = 1:length(dNum)
   disp(sprintf('%s    %s: %2.2f    %s: %d    %s: %d', ...
      datestr(clock, 'HH:MM'), dTxt{1}, dNum(k,1), ...
      dTxt{2}, dNum(k,2), dTxt{3}, dNum(k,3)));
   end
```

This time, you can display the category names returned from the spreadsheet:

```
12:55    Temp: 78.42    HeatIndex: 32    WindChill: 37
13:41    Temp: 69.73    HeatIndex: 27    WindChill: 30
14:26    Temp: 77.65    HeatIndex: 17    WindChill: 16
15:10    Temp: 68.19    HeatIndex: 22    WindChill: 35
```

Variables Used in the Expression

Anonymous functions commonly include two types of variables:

- Variables specified in the argument list. These often vary with each function call.

- Variables specified in the body of the expression. MATLAB captures these variables and holds them constant throughout the lifetime of the function handle.

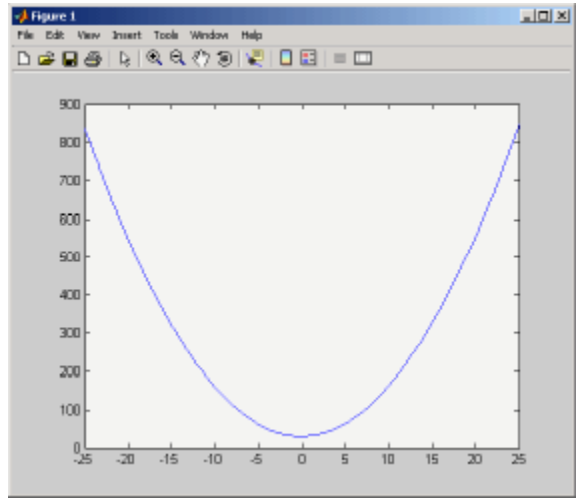
The latter variables must have a value assigned to them at the time you construct an anonymous function that uses them. Upon construction, MATLAB captures the current value for each variable specified in the body of that function. The function will continue to associate this value with the variable even if the value should change in the workspace or go out of scope.

The fact that MATLAB captures the values of these variables when the handle to the anonymous function is constructed enables you to execute an anonymous function from anywhere in the MATLAB environment, even outside the scope in which its variables were originally defined. But it also means that to supply new values for any variables specified within the expression, you must reconstruct the function handle.

Changing Variables Used in an Anonymous Function

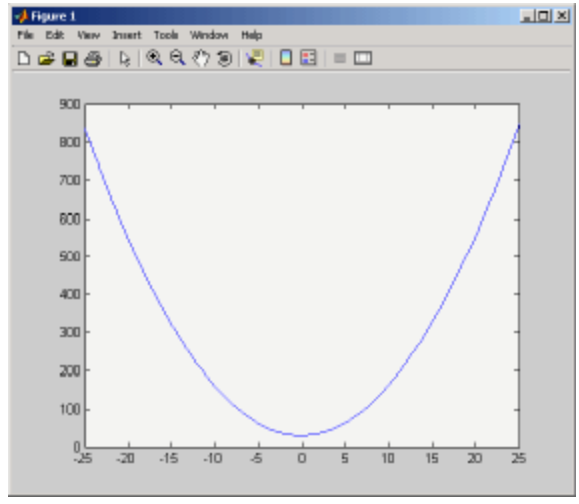
The second statement shown below constructs a function handle for an anonymous function called `parabola` that uses variables `a`, `b`, and `c` in the expression. Passing the function handle to the MATLAB `fplot` function plots it out using the initial values for these variables:

```
a = 1.3;   b = .2;   c = 30;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```



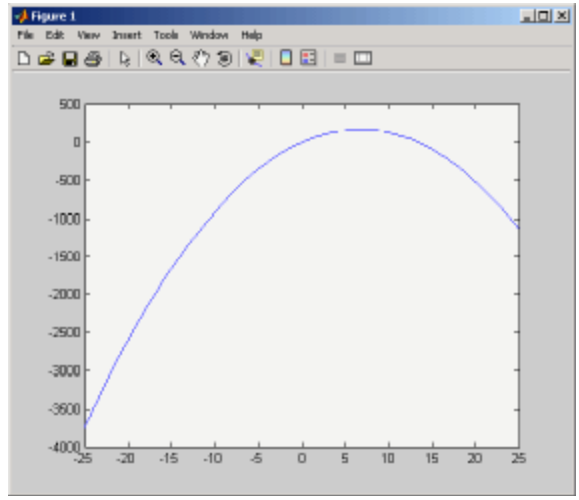
If you change the three variables in the workspace and replot the figure, the parabola remains unchanged because the `parabola` function is still using the initial values of `a`, `b`, and `c`:

```
a = -3.9;   b = 52;   c = 0;
fplot(parabola, [-25 25])
```



To get the function to use the new values, you need to reconstruct the function handle, causing MATLAB to capture the updated variables. Replot using the new construct, and this time the parabola takes on the new values:

```
a = -3.9;   b = 52;   c = 0;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```



For the purposes of this example, there is no need to store the handle to the anonymous function in a variable (`parabola`, in this case). You can just construct and pass the handle right within the call to `fplot`. In this way, you update the values of `a`, `b`, and `c` on each call:

```
fplot(@(x) a*x.^2 + b*x + c, [-25 25])
```

Examples of Anonymous Functions

This section shows a few examples of how you can use anonymous functions. These examples are intended to show you how to program with this type of function. For more mathematically oriented examples, see the MATLAB Mathematics documentation.

The examples in this section include

Example 1 — Passing a Function to quad

The equation shown here has one variable `t` that can vary each time you call the function, and two additional variables, `g` and `omega`. Leaving these two variables flexible allows you to avoid having to hardcode values for them in the function definition:

```
x = g * cos(omega * t)
```

One way to program this equation is to write an M-file function, and then create a function handle for it so that you can pass the function to other functions, such as the MATLAB `quad` function as shown here. However, this requires creating and maintaining a new M-file for a purpose that is likely to be temporary, using a more complex calling syntax when calling `quad`, and passing the `g` and `omega` parameters on every call. Here is the function M-file:

```
function f = vOut(t, g, omega)
f = g * cos(omega * t);
```

This code has to specify `g` and `omega` on each call:

```
g = 2.5;  omega = 10;

quad(@vOut, 0, 7, [], [], g, omega)
ans =
   0.1935

quad(@vOut, -5, 5, [], [], g, omega)
ans =
   -0.1312
```

You can simplify this procedure by setting the values for `g` and `omega` just once at the start, constructing a function handle to an anonymous function that only lasts the duration of your MATLAB session, and using a simpler syntax when calling `quad`:

```
g = 2.5;  omega = 10;
```

```
quad(@(t) (g * cos(omega * t)), 0, 7)
ans =
   0.1935

quad(@(t) (g * cos(omega * t)), -5, 5)
ans =
   -0.1312
```

To preserve an anonymous function from one MATLAB session to the next, `save` the function handle to a MAT-file

```
save anon.mat f
```

and then `load` it into the MATLAB workspace in a later session:

```
load anon.mat f
```

Example 2 — Multiple Anonymous Functions

This example solves the following equation by combining two anonymous functions:

$$g(c) = \int_0^1 (x^2 + cx + 1)dx$$

The equivalent anonymous function for this expression is

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));
```

This was derived as follows. Take the parenthesized part of the equation (the integrand) and write it as an anonymous function. You don't need to assign the output to a variable as it will only be passed as input to the `quad` function:

```
@(x) (x.^2 + c*x + 1)
```

Next, evaluate this function from zero to one by passing the function handle, shown here as the entire anonymous function, to `quad`:

```
quad(@(x) (x.^2 + c*x + 1), 0, 1)
```

Supply the value for c by constructing an anonymous function for the entire equation and you are done:

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));

g(2)
ans =
   2.3333
```

Primary M-File Functions

The first function in any M-file is called the *primary function*. Following the primary function can be any number of subfunctions, which can serve as subroutines to the primary function.

Under most circumstances, the primary function is the only function in an M-file that you can call from the MATLAB command line or from another M-file function. You invoke this function using the name of the M-file in which it is defined.

For example, the `average` function shown here resides in the file `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.

y = sum(x)/length(x);      % Actual computation
```

You can invoke this function from the MATLAB command line with this command to find the average of three numbers:

```
average([12 60 42])
```

Note that it is customary to give the primary function the same name as the M-file in which it resides. If the function name differs from the filename, then you must use the filename to invoke the function.

Nested Functions

In this section...

Writing Nested Functions

You can define one or more functions within another function in MATLAB. These inner functions are said to be *nested* within the function that contains them. You can also nest functions within other nested functions.

To write a nested function, simply define one function within the body of another function in an M-file. Like any M-file function, a nested function contains any or all of the components described in “Basic Parts of an M-File” on page 4-8. In addition, you must always terminate a nested function with an `end` statement:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
   end
...
end
```

Note M-file functions don’t normally require a terminating `end` statement. This rule does not hold, however, when you nest functions. If an M-file contains one or more nested functions, you must terminate *all* functions (including subfunctions) in the M-file with `end`, whether or not they contain nested functions.

Example — More Than One Nested Function

This example shows function `A` and two additional functions nested inside `A` at the same level:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
   end

   function z = C(p4)
   ...
   end
...
end
```

Example — Multiply Nested Functions

This example shows multiply nested functions, `C` nested inside `B`, and `B` in `A`:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
      function z = C(p4)
      ...
      end
   ...
   end
...
end
```

Calling Nested Functions

You can call a nested function

- From the level immediately above it. (In the following code, function A can call B or D, but not C or E.)
- From a function nested at the same level within the same parent function. (Function B can call D, and D can call B.)

- From a function at any lower level. (Function C can call B or D, but not E.)

```
function A(x, y)              % Primary function
B(x, y);
D(y);

   function B(x, y)           % Nested in A
   C(x);
   D(y);

      function C(x)           % Nested in B
      D(x);
      end
   end

   function D(x)              % Nested in A
   E(x);

      function E(x)           % Nested in D
      ...
      end
   end
end
```

You can also call a subfunction from any nested function in the same M-file.

You can pass variable numbers of arguments to and from nested functions, but you should be aware of how MATLAB interprets `varargin`, `varargout`, `nargin`, and `nargout` under those circumstances. See "Passing Optional Arguments to Nested Functions" in the MATLAB Programming documentation for more information on this.

Note If you construct a function handle for a nested function, you can call the nested function from any MATLAB function that has access to the handle. See “Using Function Handles with Nested Functions” on page 5-21.

Variable Scope in Nested Functions

The scope of a variable is the range of functions that have direct access to the variable to set, modify, or acquire its value. When you define a local (i.e., nonglobal) variable within a function, its scope is normally restricted to that function alone. For example, subfunctions do not share variables with the primary function or with other subfunctions. This is because each function and subfunction stores its variables in its own separate workspace.

Like other functions, a nested function has its own workspace. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

In the following two examples, variable `x` is stored in the workspace of the outer `varScope` function and can be read or written to by all functions nested within it.

```
function varScope1
x = 5;
nestfun1
   function nestfun1
      nestfun2

      function nestfun2
         x = x + 1
      end
   end
end
```

```
function varScope2
nestfun1
   function nestfun1
      nestfun2

      function nestfun2
         x = 5;
      end
   end
x = x + 1
end
```

As a rule, a variable used or defined within a nested function resides in the workspace of the outermost function that both contains the nested function and accesses that variable. The scope of this variable is then the function to which this workspace belongs, and all functions nested to any level within that function.

In the next example, the outer function, `varScope3`, does not access variable `x`. Following the rule just stated, `x` is unknown to the outer function and thus is

not shared between the two nested functions. In fact, there are two separate x variables in this example: one in the function workspace of nestfun1 and one in the function workspace of nestfun2. When nestfun2 attempts to update x, it fails because x does not yet exist in this workspace:

```
function varScope3
nestfun1
nestfun2

   function nestfun1
      x = 5;
   end

   function nestfun2
      x = x + 1
   end
end
```

The Scope of Output Variables

Variables containing values returned by a nested function are not in the scope of outer functions. In the two examples shown here, the one on the left fails in the second to last line because, although the *value* of y is returned by the nested function, the *variable* y is local to the nested function, and unknown to the outer function. The example on the right assigns the return value to a variable, z, and then displays the value of z correctly.

Incorrect	Correct

`function varScope4`
`x = 5; nestfun;`
`  function y = nestfun`
`  y = x + 1;`
`  end`
 	
`y`
`end`
	`function varScope5`
	`x = 5;`
	`z = nestfun;`
	`  function y = nestfun`
	`  y = x + 1;`
	`  end`
	
	`z`
	`end`

Using Function Handles with Nested Functions

Every function has a certain *scope*, that is, a certain range of other functions to which it is visible. A function's scope determines which other functions can call it. You can call a function that is out of scope by providing an alternative means of access to it in the form of a function handle. (The function handle, however, must be within the scope of its related function when you construct the handle.) Any function that has access to a function handle can call the function with which the handle is associated.

Note Although you can call an out of scope function by means of a function handle, the handle itself must be within the scope of its related function at the time it is constructed.

The section on "Calling Nested Functions" on page 5-17 defines the scope of a nested function. As with other types of functions, you can make a nested function visible beyond its normal scope with a function handle. The following function `getCubeHandle` constructs a handle for nested function `findCube` and returns its handle, `h`, to the caller. The `@` sign placed before a function name (e.g., `@findCube`) is the MATLAB operator that constructs a handle for that function:

```
function h = getCubeHandle
h = @findCube;                    % Function handle constructor

   function cube = findCube(X)    % Nested function
      cube = X .^ 3;
   end
end
```

Call `getCubeHandle` to obtain the function handle to the nested function `findCube`. Assign the function handle value returned by `getCubeHandle` to an output variable, `cubeIt` in this case:

```
cubeIt = getCubeHandle;
```

You can now use this variable as a means of calling `findCube` from outside of its M-file:

```
cubeIt(8)
ans =
    512
```

Note When calling a function by means of its handle, use the same syntax as if you were calling a function directly. But instead of calling the function by its name (e.g., `strcmp(S1, S2)`), use the variable that holds the function handle (e.g., `fhandle(S1, S2)`).

Function Handles and Nested Function Variables

One characteristic of nested functions that makes them different from other MATLAB functions is that they can share nonglobal variables with certain other functions within the same M-file. A nested function `nFun` can share variables with any outer function that contains `nFun`, and with any function nested within `nFun`. This characteristic has an impact on how certain variables are stored when you construct a handle for a nested function.

Defining Variables When Calling Via Function Handle. The example below shows a primary function `getHandle` that returns a function handle for the nested function `nestFun`. The `nestFun` function uses three different types of variables. The `VLoc` variable is local to the nested function, `VInp` is passed in when the nested function is called, and `VExt` is defined by the outer function:

```
function h = getHandle(X)
h = @nestFun;
VExt = someFun(X);

   function nestFun(VInp)
   VLoc = 173.5;
   doSomeTask(VInp, VLoc, VExt);
   end
end
```

As with any function, when you call `nestFun`, you must ensure that you supply the values for any variables it uses. This is a straightforward matter

when calling the nested function directly (that is, calling it from `getHandle`). `VLoc` has a value assigned to it within `nestFun`, `VInp` has its value passed in, and `VExt` acquires its value from the workspace it shares with `getHandle`.

However, when you call `nestFun` using a function handle, only the nested function executes; the outer function, `getHandle`, does not. It might seem at first that the variable `VExt`, otherwise given a value by `getHandle`, has no value assigned to it in the case. What in fact happens though is that MATLAB stores variables such as `VExt` inside the function handle itself when it is being constructed. These variables are available for as long as the handle exists.

The `VExt` variable in this example is considered to be *externally scoped* with respect to the nested function. Externally scoped variables that are used in nested functions for which a function handle exists are stored within the function handle. So, function handles not only contain information about accessing a function. For nested functions, a function handle also stores the values of any externally scoped variables required to execute the function.

Example Using Externally Scoped Variables

The `sCountFun` and `nCountFun` functions shown below return function handles for subfunction `subCount` and nested function `nestCount`, respectively.

These two inner functions store a persistent value in memory (the value is retained in memory between function calls), and then increment this value on every subsequent call. `subCount` makes its `count` value persistent with an explicit `persistent` declaration. In `nestCount`, the count variable is externally scoped and thus is maintained in the function handle:

Using a Subfunction	Using a Nested Function

`function h = sCountFun(X)`
`h = @subCount;`
`count = X`
`subCount(0, count);`
`function subCount(incr, ini)`
`persistent count;`
`initializing = nargin > 1;`
`if initializing`
`  count = ini; else`
`  count = count + incr`
`end`
	`function h = nCountFun(X)`
	`h = @nestCount;`
	`count = X`
	`  function nestCount(incr)`
	`  count = count + incr`
	`  end`
	`end`

When `sCountFun` executes, it passes the initial value for `count` to the `subCount` subfunction. Keep in mind that the `count` variable in `sCountFun` is not the same as the count variable in `subCount`; they are entirely independent of each other. Whenever `subCount` is called via its function handle, the value for `count` comes from its persistent place in memory.

In `nestCount`, the `count` variable again gets its value from the primary function when called from within the M-file. However, in this case the `count` variable in the primary and nested functions are one and the same. When `nestCount` is called by means of its function handle, the value for `count` is assigned from its storage within the function handle.

Running the Example. The `subCount` and `nestCount` functions increment a value in memory by another value that you pass as an input argument. Both of these functions give the same results.

Get the function handle to `nestCount`, and initialize the `count` value to a four-element vector:

```
h = nCountFun([100 200 300 400])
count =
   100   200   300   400
```

Increment the persistent vector by 25, and then by 42:

```
h(25)
```

```
count =
   125   225   325   425

h(42)
count =
   167   267   367   467
```

Now do the same using `sCountFun` and `subCount`, and verify that the results are the same.

Note If you construct a new function handle to `subCount` or `nestCount`, the former value for `count` is no longer retained in memory. It is replaced by the new value.

Separate Instances of Externally Scoped Variables

The code shown below constructs two separate function handles to the same nested function, `nestCount`, that was used in the last example. It assigns the handles to fields `counter1` and `counter2` of structure `s`. These handles reference different instances of the `nestCount` function. Each handle also maintains its own separate value for the externally scoped `count` variable.

Call `nCountFun` twice to get two separate function handles to `nestCount`. Initialize the two instances of `count` to two different vectors:

```
s.counter1 = nCountFun([100 200 300 400]);
count =
   100   200   300   400

s.counter2 = nCountFun([-100 -200 -300 -400]);
count =
  -100  -200  -300  -400
```

Now call `nestCount` by means of each function handle to demonstrate that MATLAB increments the two `count` variables individually.

Increment the first counter:

```
s.counter1(25)
```

```
count =
   125   225   325   425
s.counter1(25)
count =
   150   250   350   450
```

Now increment the second counter:

```
s.counter2(25)
count =
   -75  -175  -275  -375
s.counter2(25)
count =
   -50  -150  -250  -350
```

Go back to the first counter and you can see that it keeps its own value for `count`:

```
s.counter1(25)
count =
   175   275   375   475
```

Restrictions on Assigning to Variables

The scoping rules for nested, and in some cases anonymous, functions require that all variables used within the function be present in the text of the M-file code. Adding variables to the workspace of this type of function at run time is not allowed.

MATLAB issues an error if you attempt to dynamically add a variable to the workspace of an anonymous function, a nested function, or a function that contains a nested function. Examples of operations that might use dynamic assignment in this way are shown in the table below.

Type of Operation	How to Avoid Using Dynamic Assignment
Evaluating an expression using `eval` or `evalin`, or assigning a variable with `assignin`	As a general suggestion, it is best to avoid using the `eval`, `evalin`, and `assignin` functions altogether.
Loading variables from a MAT-file with the `load` function	Use the form of `load` that returns a MATLAB structure.
Assigning to a variable in a MATLAB script	Convert the script to a function, where argument- and result-passing can often clarify the code as well.
Assigning to a variable in the MATLAB debugger	You can declare the variable to be `global`. For example, to create a variable X for temporary use in debugging, use `K>> global X;  X = value`

One way to avoid this error in the other cases is to pre-declare the variable in the desired function.

Examples of Nested Functions

This section shows a few examples of how you can use nested functions. These examples are intended to show you how to program with this type of function. For more mathematically oriented examples, see the MATLAB Mathematics documentation.

The examples in this section include

Example 1 — Creating a Function Handle for a Nested Function

The following example constructs a function handle for a nested function and then passes the handle to the MATLAB `fplot` function to plot the parabola

shape. The `makeParabola` function shown here constructs and returns a function handle `fhandle` for the nested `parabola` function. This handle gets passed to `fplot`:

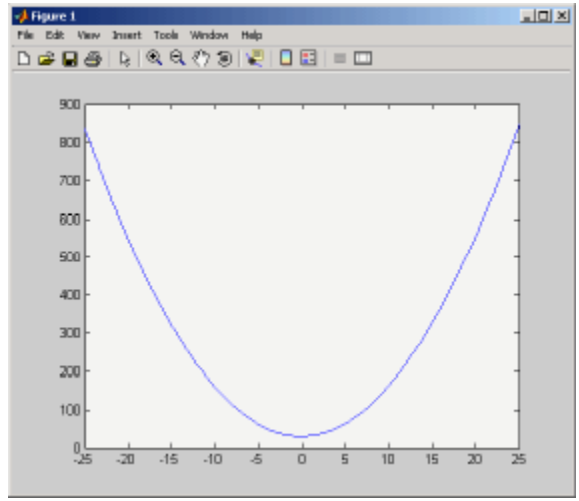
```
function fhandle = makeParabola(a, b, c)
% MAKEPARABOLA returns a function handle with parabola
% coefficients.

fhandle = @parabola;     % @ is the function handle constructor

   function y = parabola(x)
   y = a*x.^2 + b*x + c;
   end
end
```

Assign the function handle returned from the call to a variable (`h`) and evaluate the function at points 0 and 25:

```
h = makeParabola(1.3, .2, 30)
h =
    @makeParabola/parabola

h(0)
ans =
    30

h(25)
ans =
  847.5000
```

Now pass the function handle `h` to the `fplot` function, evaluating the parabolic equation from `x = -25` to `x = +25`:

```
fplot(h, [-25 25])
```



Example 2 — Function-Generating Functions

The fact that a function handle separately maintains a unique instance of the function from which it is constructed means that you can generate multiple handles for a function, each operating independently from the others. The function in this example makes IIR filtering functions by constructing function handles from nested functions. Each of these handles maintains its own internal state independent of the others.

The function `makeFilter` takes IIR filter coefficient vectors `a` and `b` and returns a filtering function in the form of a function handle. Each time a new input value x_n is available, you can call the filtering function to get the new output value y_n. Each filtering function created by `makeFilter` keeps its own private `a` and `b` vectors, in addition to its own private state vector, in the form of a transposed direct form II delay line:

```
function [filtfcn, statefcn] = makeFilter(b, a)
%   FILTFCN = MAKEFILTER(B, A) creates an IIR filtering
%   function and returns it in the form of a function handle,
```

```
%   FILTFCN. Each time you call FILTFCN with a new filter
%   input value, it computes the corresponding new filter
%   output value, updating its internal state vector at the
%   same time.
%
%   [FILTFCN, STATEFCN] = MAKEFILTER(B, A) also returns a
%   function (in the form of a function handle, STATEFCN)
%   that can return the filter's internal state.  The internal
%   state vector is in the form of a transposed direct form
%   II delay line.

%   Initialize state vector. To keep this example a bit
%   simpler, assume that a and b have the same length.
%   Also assume that a(1) is 1.

v = zeros(size(a));

filtfcn =  @iirFilter;
statefcn = @getState;

   function yn = iirFilter(xn)
      % Update the state vector
      v(1) = v(2) + b(1) * xn;
      v(2:end-1) = v(3:end) + b(2:end-1) * xn - ...
         a(2:end-1) * v(1);
      v(end) = b(end) * xn - a(end) * v(1);

      % Output is the first element of the state vector.
      yn = v(1);
   end

   function vOut = getState
      vOut = v;
   end
end
```

This sample session shows how `makeFilter` works. Make a filter that has a decaying exponential impulse response and then call it a few times in succession to see the output values change:

```
[filt1, state1] = makeFilter([1 0], [1 -.5]);

% First input to the filter is 1.
filt1(1)
ans =
   1

% Second input to the filter is 0.
filt1(0)
ans =
   0.5000

filt1(0)
ans =
   0.2500

% Show the filter's internal state.
state1()
ans =
   0.2500    0.1250

% Hit the filter with another impulse.
filt1(1)
ans =
   1.1250

% How did the state change?
state1()
ans =
   1.1250    0.5625

% Make an averaging filter.
filt2 = makeFilter([1 1 1]/3, [1 0 0]);

% Put a step input into filt2.
filt2(1)
ans =
   0.3333

filt2(1)
```

```
ans =
   0.6667

filt2(1)
ans =
   1

% The two filter functions can be used independently.
filt1(0)
ans =
   0.5625
```

As an extension of this example, suppose you were looking for a way to develop simulations of different filtering structures and compare them. This might be useful if you were interested in obtaining the range of values taken on by elements of the state vector, and how those values compare with a different filter structure. Here is one way you could capture the filter state at each step and save it for later analysis:

Call `makeFilter` with inputs `v1` and `v2` to construct function handles to the `iirFilter` and `getState` subfunctions:

```
[filtfcn, statefcn] = makeFilter(v1, v2);
```

Call the `iirFilter` and `getState` functions by means of their handles, passing in random values:

```
x = rand(1, 20);
for k = 1:20
    y(k) = filtfcn(x(k));
    states{k} = statefcn();  % Save the state at each step.
end
```

Subfunctions

In this section...

Overview

M-files can contain code for more than one function. Additional functions within the file are called *subfunctions*, and these are only visible to the primary function or to other subfunctions in the same file.

Each subfunction begins with its own function definition line. The functions immediately follow each other. The various subfunctions can occur in any order, as long as the primary function appears first:

```
function [avg, med] = newstats(u) % Primary function
% NEWSTATS Find mean and median with internal functions.
n = length(u);
avg = mean(u, n);
med = median(u, n);

function a = mean(v, n)            % Subfunction
% Calculate average.
a = sum(v)/n;

function m = median(v, n)          % Subfunction
% Calculate median.
w = sort(v);
if rem(n, 2) == 1
   m = w((n+1) / 2);
else
   m = (w(n/2) + w(n/2+1)) / 2;
end
```

The subfunctions `mean` and `median` calculate the average and median of the input list. The primary function `newstats` determines the length of the list and calls the subfunctions, passing to them the list length `n`.

Subfunctions cannot access variables used by other subfunctions, even within the same M-file, or variables used by the primary function of that M-file, unless you declare them as global within the pertinent functions, or pass them as arguments.

Calling Subfunctions

When you call a function from within an M-file, MATLAB first checks the file to see if the function is a subfunction. It then checks for a private function (described in the following section) with that name, and then for a standard M-file or built-in function on your search path. Because it checks for a subfunction first, you can override existing M-files using subfunctions with the same name.

Accessing Help for a Subfunction

You can write help for subfunctions using the same rules that apply to primary functions. To display the help for a subfunction, precede the subfunction name with the name of the M-file that contains the subfunction (minus file extension) and a > character.

For example, to get help on subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

Private Functions

In this section...

Overview

Private functions are functions that reside in subdirectories with the special name `private`. These functions are called *private* because they are visible only to M-file functions and M-file scripts that meet these conditions:

- A function that calls a private function must be defined in an M-file that resides in the directory immediately above that `private` subdirectory.
- A script that calls a private function must itself be called from an M-file function that has access to the private function according to the above rule.

For example, assume the directory `newmath` is on the MATLAB search path. A subdirectory of `newmath` called `private` can contain functions that only the functions in `newmath` can call.

Because private functions are invisible outside the parent directory, they can use the same names as functions in other directories. This is useful if you want to create your own version of a particular function while retaining the original in another directory. Because MATLAB looks for private functions before standard M-file functions, it will find a private function named `test.m` before a nonprivate M-file named `test.m`.

Primary functions and subfunctions can also be implemented as private functions.

Private Directories

You can create your own private directories simply by creating subdirectories called `private` using the standard procedures for creating directories or folders on your computer. Do not place these private directories on your path.

Accessing Help for a Private Function

You can write help for private functions using the same rules that apply to primary functions. To display the help for a private function, precede the private function name with `private/`.

For example, to get help on private function `myprivfun`, type

```
help private/myprivfun
```

Overloaded Functions

Overloaded functions are useful when you need to create a function that responds to different types of inputs accordingly. For instance, you might want one of your functions to accept both double-precision and integer input, but to handle each type somewhat differently. You can make this difference invisible to the user by creating two separate functions having the same name, and designating one to handle `double` types and one to handle integers.

MATLAB overloaded functions reside in subdirectories having a name starting with the symbol `@` and followed by the name of a recognized MATLAB data type. For example, functions in the `\@double` directory execute when invoked with arguments of MATLAB type `double`. Those in an `\@int32` directory execute when invoked with arguments of MATLAB type `int32`.

See “Classes and Objects: An Overview” on page 9-2 for more information on overloading functions in MATLAB.

6

Data Import and Export

Working with Audio and Video Data (p. 6-93)	Import and export audio and video data.
Working with Spreadsheets (p. 6-98)	Interact with Microsoft Excel and Lotus 123 spreadsheets.
Using Low-Level File I/O Functions (p. 6-104)	Use the MATLAB low-level file I/O functions, such as fopen, fread, and fwrite.
Exchanging Files over the Internet (p. 6-117)	Exchange files over the Internet with MATLAB URL, zip, and e-mail functions.

Overview

In this section...

For more information and examples on importing and exporting data, see Technical Note 1602:

`http://www.mathworks.com/support/tech-notes/1600/1602.html`

File Types Supported by MATLAB

MATLAB provides many ways to load data from disk files or the clipboard into the workspace, a process called *importing* data. Also there are many ways to save workspace variables to a disk file, a process called *exporting* data. Your choice of which import or export mechanism to use depends mainly on the format of the data being transferred: text, binary, or a standard format such as JPEG.

Note For unsupported high-level function data formats, you can use the MATLAB low-level file I/O functions if you know how the binary data is formatted in the file. See "Using Low-Level File I/O Functions" on page 6-104 for more information.

MATLAB has built-in capabilities to import and export the following types of data files:

- "Binary Data from a MATLAB Session" on page 6-4
- "Text Data" on page 6-4
- "Graphics Files" on page 6-4
- "Audio and Audio/Video Data" on page 6-4
- "Spreadsheets" on page 6-5

Binary Data from a MATLAB Session

Using the MATLAB `save` and `load` functions, you can store all or part of the data in your MATLAB workspaces to disk, and then read that data back into MATLAB at a later time.

Text Data

In text format, the data values are American Standard Code for Information Interchange (ASCII) codes that represent alphabetic and numeric characters. ASCII text data can be viewed in a text editor. For more information about working with text data in MATLAB, see

These sections also describe how to import and export to XML documents.

Graphics Files

MATLAB imports and exports images from many standard graphics file formats, including the Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats.

Audio and Audio/Video Data

MATLAB provides functions to enable you to interact with the following types of audio and audio/video files:

- NeXT/SUN SPARCstation sound
- Microsoft WAVE sound
- Audio/Video Interleaved (AVI)
- Windows-compatible sound devices
- Audio player and recorder objects

- Linear audio signals

Spreadsheets

You can use MATLAB to import and export data to the following types of spreadsheets:

- Microsoft Excel spreadsheets
- Lotus 123 spreadsheets

Data from the System Clipboard

Using the Import Wizard or the `clipboard` function, you can temporarily hold string data on your system's clipboard, and then paste it back into MATLAB.

Information from the Internet

From your MATLAB session, you can

- Send e-mail
- Download from the Internet
- Compress (zip) and decompress (unzip) files
- Connect to an FTP server to perform remote file operations

Other MATLAB I/O Capabilities

Using the Import Wizard

The Import Wizard is a graphical user interface that simplified the process of locating and loading various types of data files into MATLAB. You do not need to know the format of the data to use this tool. You simply specify the

file that contains the data and the Import Wizard processes the file contents automatically. See the section on “Using the Import Wizard” on page 6-11.

Mapping Files to Memory

Memory—mapping enables you to read and write data in a file as if were stored in the computer’s dynamic memory. The contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read or write the desired data from the file. See the section on “Accessing Files with Memory-Mapping” on page 6-23.

Reading Files with Large Data Sets

An efficient way to read files with large data sets is to read the file in segments and process the data as you go. This method requires significantly less memory than if you were to try reading in the entire file at once. Using the `textscan` function, you can read a specified amount of data from a file, and maintain a pointer to the location in the file where your last read operation ended and your next read is to begin.

This example opens a large data file and reads the file a segment at a time in a `for` loop. The code calls `textscan` to read a particular pattern of data (as specified by `format`) 10,000 times for each segment. Following each read, the subfunction `process_data` processes the data collected in cell array `segarray`:

```
format = '%s %n %s %8.2f %8.2f %8.2f %8.2f %u8';
file_id = fopen('largefile.dat', 'r');

for k = 1:segcount
   segarray = textscan(file_id, format, 10000);
   process_data(segarray);
end

fclose(file_id);
```

Low-Level File I/O

MATLAB also supports C-style, low-level I/O functions that you can use with any data format. For more information, see “Using Low-Level File I/O Functions” on page 6-104.

Importing Data with Toolboxes

In addition to MATLAB import functions, you can perform specialized import features using toolboxes. For example, use Database Toolbox for importing data from relational databases. Refer to the documentation on the specific toolbox to see what import features are offered.

Functions Used in File Management

The following functions are available in MATLAB to help you to create, manage, and locate the files and directories you work with. For more information on these and other file management functions, see "File Management Operations" in the Desktop Tools and Development Environment documentation:

Function	Description
`cd`	Switch your current working directory to another directory
`clipboard`	Copy and paste strings to and from the system clipboard
`copyfile`	Copy a file or directory to another location
`delete`	Delete the specified files
`dir`	List the files that reside in the specified directory
`edit`	Create a new M-file or edit an existing one
`exist`	Check the existence of a file or directory
`fileattrib`	Set or get attributes of a file or directory
`filebrowser`	Start the Current Directory Browser
`fileparts`	Show the components of a file name and its place on the path
`fullfile`	Build a full file name from its components
`ls`	List the contents of a specific directory
`mkdir`	Create a new directory
`movefile`	Move a file or directory to a new location
`open`	Open files based on extension
`pwd`	Identify the directory you are currently working in

Function	Description
`recycle`	Set an option to move deleted files to recycle folder
`rmdir`	Delete a specific directory
`what`	List the MATLAB files in a specific directory
`which`	Locate functions and files

Supported File Formats

The table below shows the file formats that you can read or write from MATLAB along with the functions that support each format.

File Format	File Content	Extension	Functions
MATLAB formatted	Saved MATLAB workspace	`.mat`	`load`, `save`
Text	Text	any	`textscan`
	Text	any	`textread`
	Delimited text	any	`dlmread`, `dlmwrite`
	Comma-separated numbers	`.csv`	`csvread`, `csvwrite`
Extended Markup Language	XML-formatted text	`.xml`	`xmlread`, `xmlwrite`
Audio	NeXT/SUN sound	`.au`	`auread`, `auwrite`
	Microsoft WAVE sound	`.wav`	`wavread`, `wavwrite`
Movie	Audio/video	`.avi`	`aviread`
Scientific data	Data in Common Data Format	`.cdf`	`cdfread`, `cdfwrite`
	Flexible Image Transport System data	`.fits`	`fitsread`
	Data in Hierarchical Data Format	`.hdf`	`hdfread`
Spreadsheet	Excel worksheet	`.xls`	`xlsread`, `xlswrite`
	Lotus 123 worksheet	`.wk1`	`wk1read`, `wk1write`

File Format	File Content	Extension	Functions
Graphics	TIFF image	`.tiff`	`imread`, `imwrite`
	PNG image	`.png`	same
	HDF image	`.hdf`	same
	BMP image	`.bmp`	same
	JPEG image	`.jpeg`	same
	GIF image	`.gif`	same
	PCX image	`.pcx`	same
	XWD image	`.xwd`	same
	Cursor image	`.cur`	same
	Icon image	`.ico`	same

Using the Import Wizard

In this section...

Overview

The easiest way to import data into MATLAB is to use the Import Wizard. You do not need to know the format of the data to use this tool. You simply specify the file that contains the data and the Import Wizard processes the file contents automatically. You can also use the Import Wizard to import HDF data. See "Using the HDF Import Tool" on page 7-36"Using the HDF Import Tool" on page 7-36 for more information.

The sections on Previewing Contents of the File or Clipboard and Specifying Delimiters and Header Format apply only to text files and the clipboard.

Starting the Import Wizard

To start the Import Wizard and select the source to import, see

- "Importing from a File" on page 6-12
- "Importing from the Clipboard" on page 6-12

If you use the `uiimport` function to start the Wizard, you can choose to have the imported data written to a MATLAB structure. See "Importing to a Structure" on page 6-12.

Importing from a File

To start the Wizard and use a file browser to locate the file to import, use one of the menu options or MATLAB commands shown here:

- Select **Import Data** from the **File** menu
- Type `uiimport -file`
- Type `uiimport`, and then click **Browse**

If you already know the name of the file to import, use one of the following means to initiate the operation:

- In the Current Directory browser, right-click the filename and select **Import Data**
- Type `uiimport filename`, where `filename` is an unquoted string containing the name of the file to import.

Importing from the Clipboard

To import from the system clipboard, use one of the menu options or MATLAB commands shown here:

- Select **Paste to Workspace** from the **Edit** menu
- Type `uiimport -pastespecial`
- Type `uiimport`, and then click **Clipboard**

Importing to a Structure

Specifying an output argument with the `uiimport` command tells MATLAB to return the imported data in the fields of a single structure rather than as separate variables.

The command

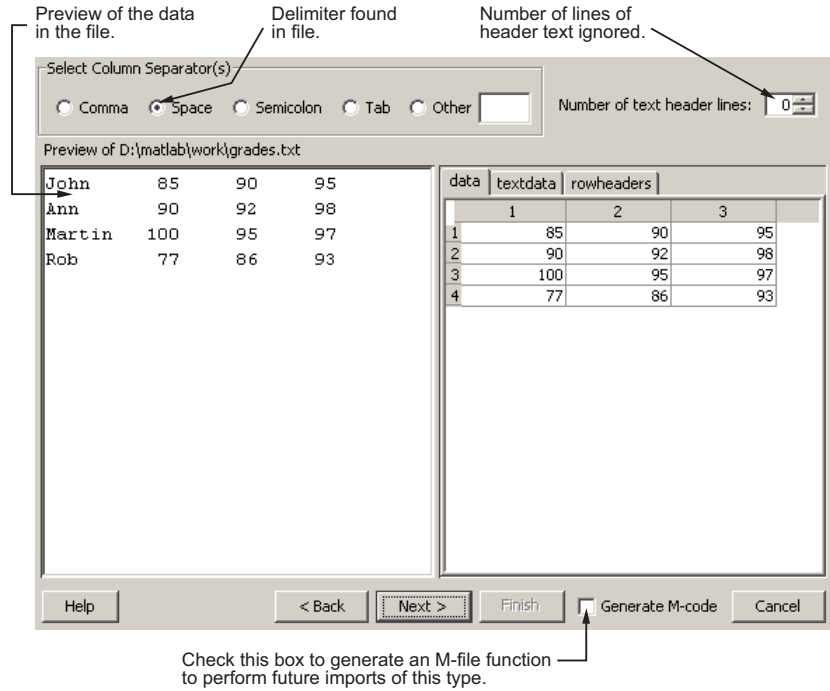
```
S = uiimport('filename')
```

imports the file `filename` to the fields of structure `S`. The `filename` argument is a single-quoted string containing the name of the file to import.

If you are importing from a binary file, skip ahead to step 4: Determine Assignment to Variables.

Previewing Contents of the File or Clipboard [Text only]

When the Import Wizard imports text data from a file or the clipboard, it opens the dialog box shown here and displays a portion of the raw data in the preview pane on the left. You can use this display to verify that the file contains the data you expect.



The pane on the right side of the dialog box shows how MATLAB has assigned the imported data to a default set of variables. The variable names appear in the tabs above the display pane. Click any of these tabs to see the values assigned to that variable. The variable names are derived from categories into which the Import Wizard has sorted the data. These are

- `rowheaders`—Column vector containing the names of all row headers.
- `colheaders`—Row vector containing the names of all column headers.
- `textdata`—Matrix containing all imported text data. Empty elements are set to `''`.
- `data`—Matrix containing all imported numeric data. Empty elements are set to `NaN`.

If the imported file or clipboard contains *only* numeric or *only* text data, then the Import Wizard does not use the variable names shown above. Instead, it assigns all of the data to just one variable:

- For data imported from a text file, the name of the variable is the same as the filename, minus the file extension.
- For data imported from the clipboard, the name of the variable is `A_pastespecial`.

Specifying Delimiters and Header Format [Text only]

Using the options shown at the top of the Import Wizard dialog box, you can specify a delimiter character for separating data items, and also the number of lines you want to use for column headers.

Delimiters

Most text files use a unique character called a *delimiter* or column separator to mark the separation between items of data. For example, data in a comma-separated value (CSV) file is, of course, separated by commas. Data in some other file might be separated by tab or space characters.

When the Import Wizard imports from a text file or the clipboard, it makes its best guess as to which character was intended as the delimiter and displays the data accordingly. If this is not correct, you will need to set the correct delimiter from the choices shown under **Select Column Separator(s)** in the upper left of the dialog box. When you do this, the Import Wizard immediately reformats the data, displaying new values for the data shown in the preview pane.

Header Format

When reading in data from a text file or the clipboard, the Wizard looks for any lines at the top that have no numeric characters, and assigns these lines to the variable `textdata`. MATLAB counts these lines and displays the count in the value field of **Number of text header lines** in the upper right of the Import Wizard window. You can adjust this count if it does not accurately represent the header format within the file.

Note The **Number of text header lines** selector applies only to column headers. It has no effect on row headers.

MATLAB creates a row vector from the bottommost of these lines and assigns it to the variable `colheaders`.

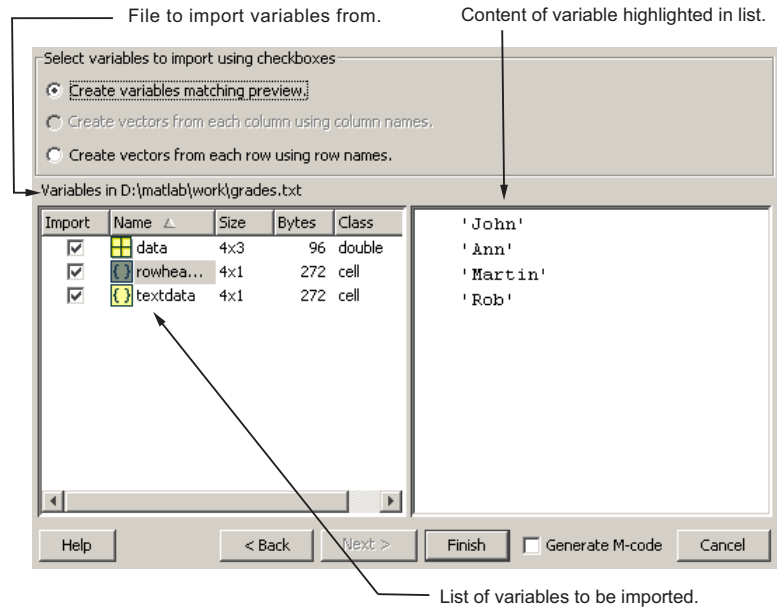
Generate M-Code Checkbox

The **Generate M-code** checkbox at the bottom of the Import Wizard dialog box applies to both text and binary data, and thus is described in "Automated M-Code Generation" on page 6-18.

To continue, click **Next** at the bottom of the dialog box.

Determining Assignment to Variables

At this point, the Import Wizard displays the dialog box shown below. This dialog displays data for both text and binary files.



The left pane of the dialog box displays a list of the variables MATLAB created for your data. For text files, MATLAB derives the variable names as described in step 2: Preview Contents of the File. For binary files, the variable names are taken directly from the file.

Click any variable name and MATLAB displays the contents of that variable in the pane to the right. MATLAB highlights the name of the variable that is currently displayed in the right pane.

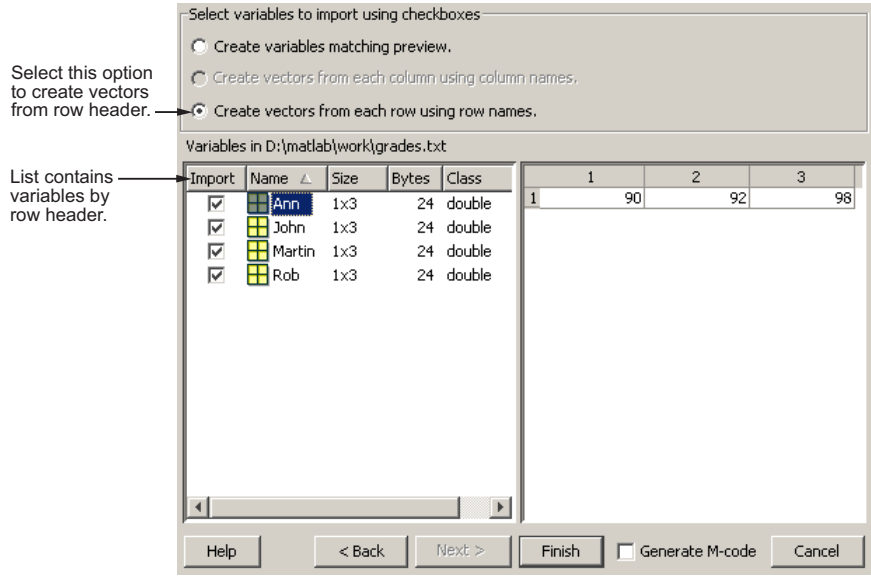
Structuring the Output Data

The top portion of this dialog box offers three options for organizing the file's data:

- **Create variables matching preview**
- **Create vectors from each column using column names**
- **Create vectors from each row using row names**

Note For data imported from a binary file, only the top option is active. Variable names and assignment are taken directly from the imported file. For text data, you can use any of the three options, however, the bottom two are active only if the file or clipboard contains row or column headers.

While importing from the example text file `grades.txt`, select the third option to create vectors from row names. Observe that the display replaces the default variable assignments with new variables derived from the row headers. Click any of these variable names, and the Wizard displays the contents of the corresponding row vector.



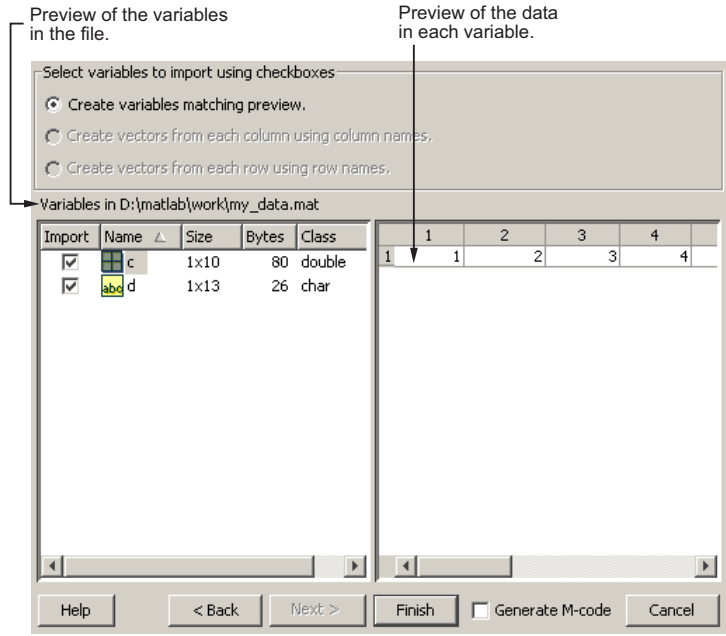
Selecting Which Variables to Write to the Workspace

The checkboxes to the left of each variable name enable you to include or exclude individual variables from those that will be written to the workspace. By default, all variables are selected. Select the checkbox of any variable you do not want written to the workspace. The check mark is removed from any variables that you deselect.

Example of Selecting Variables to Load. Use the Import Wizard to import this sample binary MAT-file, `my_data.mat`,

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

D =
    a test string
```

The Import Wizard displays two variables, as listed in the preview pane. To select a variable to import, select the check box next to its name. All variables are preselected by default.



Automated M-Code Generation

To perform additional imports from this or a similar type of file, you can automate this process by creating a MATLAB function that performs all of the steps you just went through. To have the Import Wizard write this function

for you, select the **Generate M-code** checkbox in the lower right corner of the Wizard dialog.

Once you click **Finish** to complete the import, MATLAB opens an Editor window displaying the generated M-file function. The function is called `importfile.m`. If this name is already taken, then MATLAB names the file `importfileN.m`, where `N` is a number that is one greater than the highest existing `importfile.m` file.

The generated function has the following input and output arguments:

- Input: `fileToRead1`—Name of the file to import from. This argument exists only when importing from a file.
- Output: `newData1`—Structure to assign all imported data to. This argument exists only if you have specified an output argument with the call to `uiimport` when starting the Import Wizard. Otherwise, variables retain the same naming as assigned within the Wizard.

The `newData1` output is a structure that has one field for each output of the import operation.

The workspace variables created by this generated M-code are the same as those created by running the Import Wizard. For example, if you elect to format the output in column vectors when running the Import Wizard, the generated M-file does the same. However, unlike the Import Wizard, you cannot mark any variables to be excluded from the output.

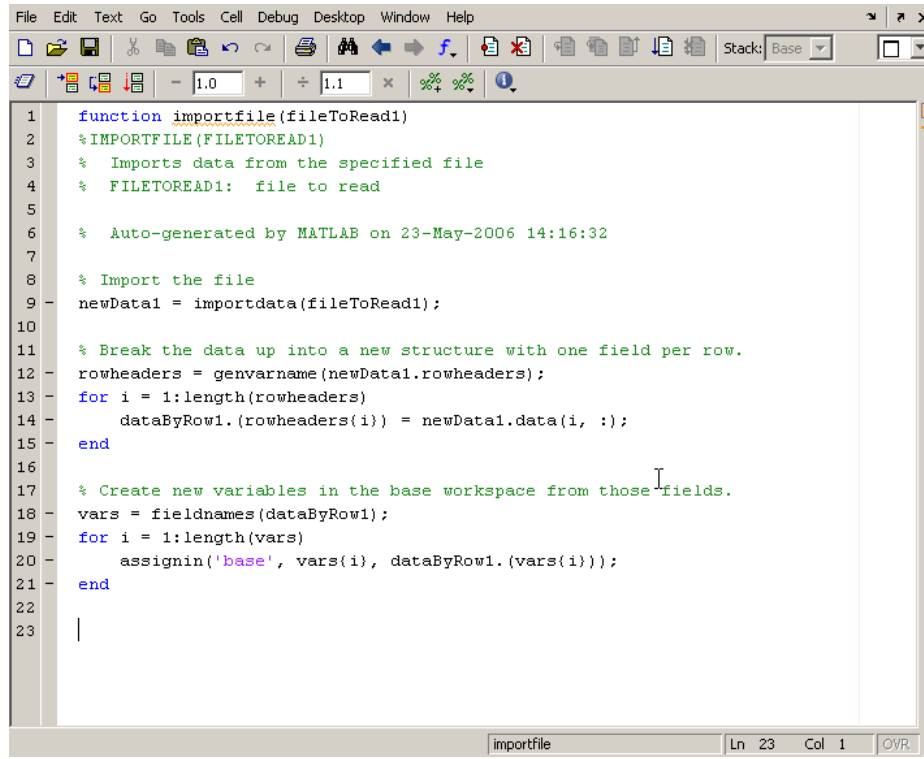
Make any necessary modifications to the generated M-file function in the Editor window. To save the M-file, select **Save** from the **File** menu at the top.

Caution You must save the file yourself; MATLAB does not automatically save it for you.

Example of M-Code Generation

The M-file shown below was generated by MATLAB during an import of the file `grades.txt`, shown earlier in this section. During the import that created this file, the option to **Create vectors from each row using row names**

was selected, thus generating four row vectors for output: `Ann`, `John`, `Martin`, and `Rob`. Also, the row vector for `John` was deselected by clearing the checkbox next to that name in the Wizard.



If you run the function, you find that the workspace now holds the four row vectors `Ann`, `John`, `Martin`, and `Rob`, instead of the default variables created by the Import Wizard (`data`, `textdata`, and `rowheaders`). Also, note that the vector for `John` is written to the workspace along with the others, even though this one variable had been deselected from the Import Wizard interface.

```
importfile grades.txt

whos
  Name        Size              Bytes  Class     Attributes

  Ann         1x3                  24  double
```

```
John        1x3                24  double
Martin      1x3                24  double
Rob         1x3                24  double
```

Writing Data to the Workspace

To complete the import operation, click **Finish** to bring the data into the MATLAB workspace. This button also dismisses the Import Wizard.

Variables written to the workspace are in one of the following formats. The first three apply only to data read from text files or the clipboard, the fourth applies only to binary files, and the last applies to both:

Variable Name	Output
`data`, `textdata`, `rowheaders`, `colheaders`	Separate matrices for numeric, text, and header data.
Variables named after row or column headers	One vector for each row or column.
Single variable named after the filename, or `A_pastespecial`	One matrix for all data named after the filename
Variable names taken from binary file	Data assigned to each variable stored in a binary file.
Output variable assigned during call to `uiimport`	A single structure having fields that match one of the formats described above.

Examples

Here are a few examples of how to use the Import Wizard.

Example 1—Text Data. Start by creating the text file `grades.txt` using the MATLAB editor. The file contains the following:

```
John      85     90     95
Ann       90     92     98
Martin   100     95     97
Rob       77     86     93
```

Import from text file `grades.txt`, using default variables to store the data:

```
uiimport grades.txt
whos
  Name            Size            Bytes  Class     Attributes

  data            4x3                96  double
  rowheaders      4x1               272  cell
  textdata        4x1               272  cell
```

Example 2—Partial Text File with Row Vectors. Import from the same file as in the above example, but this time select **Create vectors from each row using row names**. Also, clear the checkbox next to variable `John` so that this one vector does not get written to the workspace:

```
whos
  Name        Size            Bytes  Class     Attributes

  Ann         1x3                24  double
  Martin      1x3                24  double
  Rob         1x3                24  double
```

Example 3—Binary Data Assigned to a Structure. Save numeric and text data in binary format in file `importtest.mat` and use the Import Wizard to import the binary file into the workspace.

```
C = [1 2 3 4 5;6 7 8 9 10];
D = 'a test string';
save importtest C D

clear
s = uiimport('importtest.mat')
s =
    C: [2x5 double]
    D: 'a test string'
```

Accessing Files with Memory-Mapping

In this section...

Overview of Memory-Mapping in MATLAB

Memory-mapping is a mechanism that maps a portion of a file, or an entire file, on disk to a range of addresses within an application's address space. The application can then access files on disk in the same way it accesses dynamic memory. This makes file reads and writes faster in comparison with using functions such as `fread` and `fwrite`.

Another advantage of using memory-mapping in MATLAB is that it enables you to access file data using standard MATLAB indexing operations. Once you have mapped a file to memory, you can read the contents of that file using the same type of MATLAB statements used to read variables from the MATLAB workspace. The contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read or write the desired data from the file.

This section describes the benefits and limitations of memory-mapping in MATLAB. The last part of this section gives details on which types of applications derive the greatest advantage from using memory-mapping:

- "Benefits of Memory-Mapping" on page 6-24
- "Limitations of Memory-Mapping in MATLAB" on page 6-25
- "When to Use Memory-Mapping" on page 6-26

Benefits of Memory-Mapping

The principal benefits of memory-mapping are efficiency, faster file access, the ability to share memory between applications, and more efficient coding.

Faster File Access. Accessing files via memory map is faster than using I/O functions such as `fread` and `fwrite`. Data is read and written using the virtual memory capabilities that are built in to the operating system rather than having to allocate, copy into, and then deallocate data buffers owned by the process.

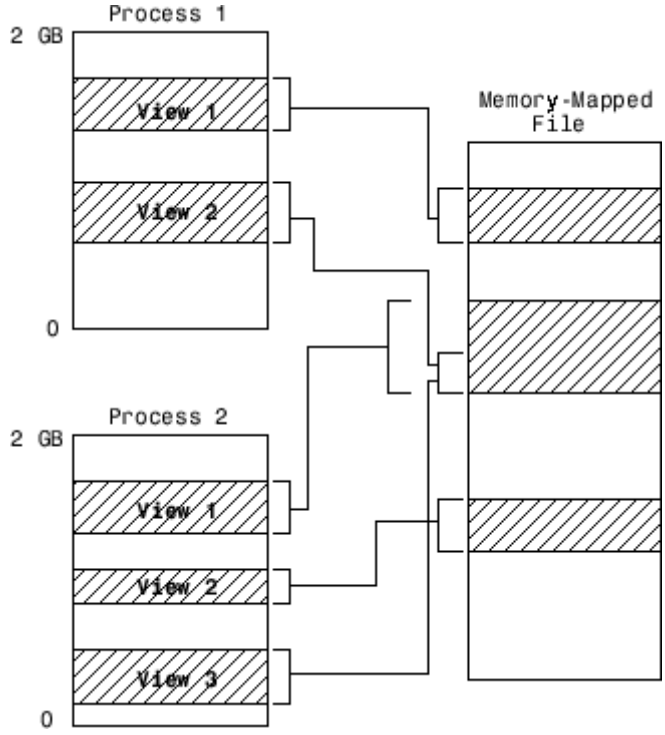
MATLAB does not access data from the disk when the map is first constructed. It only reads or writes the file on disk when a specified part of the memory map is accessed, and then it only reads that specific part. This provides faster random access to the mapped data.

Efficiency. Mapping a file into memory allows access to data in the file as if that data had been read into an array in the application's address space. Initially, MATLAB only allocates address space for the array; it does not actually read data from the file until you access the mapped region. As a result, memory-mapped files provide a mechanism by which applications can access data segments in an extremely large file without having to read the entire file into memory first.

Efficient Coding Style. Memory-mapping eliminates the need for explicit calls to the `fread` and `fwrite` functions. In MATLAB, if `x` is a memory-mapped variable, and `y` is the data to be written to a file, then writing to the file is as simple as

```
x.Data = y;
```

Sharing Memory Between Applications. Memory-mapped files also provide a mechanism for sharing data between applications, as shown in the figure below. This is achieved by having each application map sections of the same file. This feature can be used to transfer large data sets between MATLAB and other applications.



Also, within a single application, you can map the same segment of a file more than once.

Limitations of Memory-Mapping in MATLAB

MATLAB restricts the size of a memory map to 2 gigabytes, and on some platforms, requires that you set up your memory-mapping so that all data access is aligned properly. See the following section, “Maximum Size of a Memory Map”, for more information.

Maximum Size of a Memory Map. Due to limits set by the operating system, the maximum amount of data you can map with a single instance of a memory map is 2^{31} - 1 (or 2 GB). If you need to map more than 2 GB, you can either create separate maps for different regions of the file, or you can move the 2 GB window of one map to different locations in the file.

The 2 GB limit also applies to 64-bit platforms. However, because 64-bit platforms have a much larger address space, they can support having many more map instances in memory at any given time.

Aligned Access on Sol64. The Sol64 platform only supports aligned data access. This means that numeric values of type `double` that are to be read from a memory-mapped file must start at some multiple of 8 bytes from the start of the file. (Note that this is from the start of the *file*, and not the start of the mapped region.) Furthermore, numeric values of type `single` and also 32-bit integers must start at multiples of 4 bytes, and 16-bit integers at 2-byte multiples.

If you attempt to map a file on Sol64 that does not take into account these alignment considerations, MATLAB generates an error.

Byte Ordering

Memory-mapping works only with data that has the same byte ordering scheme as the native byte ordering of your operating system. For example, because both Linux and Windows use little-endian byte ordering, data created on a Linux system can be read on Windows. You can use the `computer` function to determine the native byte ordering of your current system.

When to Use Memory-Mapping

Just how much advantage you get from mapping a file to memory depends mostly on the size and format of the file, the way in which data in the file is used, and the computer platform you are using.

When Memory-Mapping Is Most Useful. Memory-mapping works best with binary files, and in the following scenarios:

- For large files that you want to access randomly one or more times

- For small files that you want to read into memory once and access frequently
- For data that you want to share between applications
- When you want to work with data in a file as if it were a MATLAB array

When the Advantage Is Less Significant. The following types of files do not fully utilize the benefits of memory-mapping:

- Formatted binary files like HDF or TIFF that require customized readers are not good for memory-mapping. For one thing, describing the data contained in these files can be a very complex task. Also, you cannot access data directly from the mapped segment, but must instead create arrays to hold the data.
- Text or ASCII files require that you convert the text in the mapped region to an appropriate type for the data to be meaningful. This takes up additional address space.
- Files that are larger than several hundred megabytes in size consume a significant amount of the virtual address space needed by MATLAB to process your program. Mapping files of this size may result in MATLAB reporting out-of-memory errors more often. This is more likely if MATLAB has been running for some time, or if the memory used by MATLAB becomes fragmented.

The memmapfile Class

MATLAB implements memory-mapping using an object-oriented class called `memmapfile`. The `memmapfile` class has the properties and methods you need to map to a file, read and write the file via the map, and remove the map from memory when you are done.

Properties of the memmapfile Class

There are six properties defined for the `memmapfile` class. These are shown in the table below. These properties control which file is being mapped, where in the file the mapping is to begin and end, how the contents of the file are to be formatted, and whether or not the file is writable. One property of the file contains the file data itself.

Property	Description	Data Type	Default
`Data`	Contains the data read from the file or to be written to the file. (See "Reading a Mapped File" on page 6-43 and "Writing to a Mapped File" on page 6-48)	Any of the numeric types	None
`Filename`	Path and name of the file to map into memory. (See "Selecting the File to Map" on page 6-32)	`char` array	None
`Format`	Format of the contents of the mapped region, including data type, array shape, and variable or field name by which to access the data. (See "Identifying the Contents of the Mapped Region" on page 6-34)	`char` array or N-by-3 cell array	`uint8`
`Offset`	Number of bytes from the start of the file to the start of the mapped region. This number is zero-based. That is, offset 0 represents the start of the file. Must be a nonnegative integer value. (See "Setting the Start of the Mapped Region" on page 6-34)	`double`	`0`
`Repeat`	Number of times to apply the specified format to the mapped region of the file. Must be a positive integer value or `Inf`. (See "Repeating a Format Scheme" on page 6-41)	`double`	`Inf`
`Writable`	Type of access allowed to the mapped region. Must be logical 1 or logical 0. (See "Setting the Type of Access" on page 6-42)	`logical`	`false`

You can set the values for any property except for `Data` at the time you call the `memmapfile` constructor, or at any time after that while the map is still valid. Any properties that are not explicitly set when you construct the object are given their default values as shown in the table above. For information on calling the constructor, see "Constructing a memmapfile Object" on page 6-29.

Once a `memmapfile` object has been constructed, you can change the value of any of its properties. Use the `objname.property` syntax in assigning the new value. For example, to set a new `Offset` value for memory map object `m`, type

```
m.Offset = 2048;
```

Note Property names are not case sensitive. For example, MATLAB considers `m.offset` to be the same as `m.Offset`.

To display the value of all properties of a `memmapfile` object, simply type the object name. For a `memmapfile` object `m`, typing the variable name `m` displays the following. Note that this example requires the file `records.dat` which you will create at the beginning of the next section.

```
m =
    Filename: 'records.dat'
    Writable: true
      Offset: 1024
      Format: 'uint32'
      Repeat: Inf
        Data: 4778x1 uint32 array
```

To display the value of any individual property, for example the `Writable` property of object `m`, type

```
m.Writable
ans =
   true
```

Constructing a memmapfile Object

The first step in mapping to any file is to construct an instance of the `memmapfile` class using the class constructor function. You can have MATLAB assign default values to each of the new object's properties, or you can specify property values yourself in the call to the `memmapfile` constructor.

For information on how to set these values, see the sections that cover

- "Constructing the Object with Default Property Values" on page 6-30
- "Changing Property Values" on page 6-31
- "Selecting the File to Map" on page 6-32
- "Setting the Start of the Mapped Region" on page 6-34
- "Identifying the Contents of the Mapped Region" on page 6-34

All the examples in this section use a file named `records.dat` that contains a 5000-by-1 matrix of double-precision floating point numbers. Use the following code to generate this file before going on to the next sections of this documentation.

First, save this function in your current working directory:

```
function gendatafile(filename, count)
dmax32 = double(intmax('uint32'));
rand('state', 0)

fid = fopen(filename,'w');
fwrite(fid, rand(count,1)*dmax32, 'double');
fclose(fid);
```

Now execute the `gendatafile` function to generate the `records.dat` file that is referenced in this section. You can use this function at any time to regenerate the file:

```
gendatafile('records.dat', 5000);
```

Constructing the Object with Default Property Values

The simplest and most general way to call the constructor is with one input argument that specifies the name of the file you want to map. All other properties are optional and are given their default values. Use the syntax shown here:

```
objname = memmapfile(filename)
```

To construct a map for the file `records.dat` that resides in your current working directory, type the following:

```
m = memmapfile('records.dat')
m =
    Filename: 'd:\matlab\mfiles\records.dat'
```

```
    Writable: false
      Offset: 0
      Format: 'uint8'
      Repeat: Inf
        Data: 40000x1 uint8 array
```

MATLAB constructs an instance of the `memmapfile` class, assigns it to the variable `m`, and maps the entire `records.dat` file to memory, setting all properties of the object to their default values. In this example, the command maps the entire file as a sequence of unsigned 8-bit integers, and gives the caller read-only access to its contents.

Changing Property Values

You can make the memory map more specific to your needs by including more information when calling the constructor. In addition to the `filename` argument, there are four other parameters that you can pass to the constructor. Each of these parameters represents a property of the object, and each requires an accompanying value to be passed as well:

```
objname = memmapfile(filename, prop1, value1, prop2, value2, ...)
```

For example, to construct a map using nondefault values for the `Offset`, `Format`, and `Writable` properties, type the following, enclosing all property names and string parameter values in quotes:

```
m = memmapfile('records.dat', ...
      'Offset', 1024,         ...
      'Format', 'double',     ...
      'Writable', true);
```

Type the object name to see the current settings for all properties:

```
m

m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: true
      Offset: 1024
      Format: 'double'
      Repeat: Inf
```

```
          Data: 4872x1 double array
```

You can also change the value of any property after the object has been constructed. Use the syntax

```
objname.property = newvalue;
```

For example, to set the format to `uint16`, type the following. (Property names, like `Format`, are not case sensitive.)

```
m.format = 'uint16'
m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: true
      Offset: 1024
      Format: 'uint16'
      Repeat: Inf
        Data: 19488x1 uint16 array
```

Further read and write operations to the region mapped by `m` will now treat the data in the file as a sequence of unsigned 16-bit integers. Whenever you change the value of a `memmapfile` property, MATLAB remaps the file to memory.

Selecting the File to Map

`filename` is the only required argument when you call the `memmapfile` constructor. When you call the `memmapfile` constructor, MATLAB assigns the filename that you specify to the `Filename` property of the new object instance.

Specify the filename as a quoted string, (e.g., `'records.dat'`). It must be first in the argument list and not specified as a parameter-value pair. `filename` must include a filename extension if the name of the file being mapped has an extension. The `filename` argument cannot include any wildcard characters (e.g., `*` or `?`), and is not case sensitive.

Note Unlike the other `memmapfile` constructor arguments, you must specify `filename` as a single string, and not as a parameter-value pair.

If the file to be mapped resides somewhere on the MATLAB path, you can use a partial pathname. If the path to the file is not fully specified, MATLAB searches for the file in your current working directory first, and then on the MATLAB path.

Once `memmapfile` locates the file, MATLAB stores the absolute pathname for the file internally, and then uses this stored path to locate the file from that point on. This enables you to work in other directories outside your current work directory and retain access to the mapped file.

You can change the value of the `Filename` property at any time after constructing the `memmapfile` object. You might want to do this if

- You want to use the same `memmapfile` object on more than one file.
- You save your `memmapfile` object to a MAT-file, and then later load it back into MATLAB in an environment where the mapped file has been moved to a different location. This requires that you modify the path segment of the `Filename` string to represent the new location.

For example, save `memmapfile` object `m` to file `mymap.mat`:

```
disp(m.Filename)
    d:\matlab\mfiles\records.dat

save mymat m
```

Now move the file to another location, load the object back into MATLAB, and update the path in the `Filename` property:

```
load mymat m
m.Filename = 'f:\testfiles\oct1\records.dat'
```

Note You can only map an existing file. You cannot create a new file and map that file to memory in one operation. Use the MATLAB file I/O functions to create the file before attempting to map it to memory.

Setting the Start of the Mapped Region

By default, MATLAB begins a memory map at the start of the file. To begin the mapped region at some point beyond the start of the file, specify an `Offset` parameter in the call to the `memmapfile` constructor:

```
objname = memmapfile(filename, 'Offset', bytecount)
```

The `bytecount` value is the number of bytes from the beginning of the file to the point in the file where you want the memory map to start (a zero-based offset). To map the file `records.dat` from a point 1024 bytes from the start and extending to the end of the file, type

```
m = memmapfile('records.dat', 'Offset', 1024);
```

You can change the starting position of an existing memory map by setting the `Offset` property for the associated object to a new value. The following command sets the offset of `memmapfile` object `m` to be 2,048 bytes from the start of the mapped file:

```
m.Offset = 2048;
```

Note The Sol64 platform supports aligned data access only. If you attempt to use a `memmapfile` offset on Sol64 that does not take the necessary alignment considerations into account, MATLAB generates an error. (See “Aligned Access on Sol64” on page 6-26).

Identifying the Contents of the Mapped Region

By default, MATLAB considers all the data in a mapped file to be a sequence of unsigned 8-bit integers. To have the data interpreted otherwise as it is read or written to in the mapped file, specify a `Format` parameter and value in your call to the constructor:

```
objname = memmapfile(filename, 'Format', formatspec)
```

The `formatspec` argument can either be a character string that identifies a single data type used throughout the mapped region, or a cell array that specifies more than one data type.

For example, say that you map a file that is 12K bytes in length. Data read from this file could be treated as a sequence of 6,000 16-bit (2-byte) integers, or as 1,500 8-byte double-precision floating-point numbers, to name just a couple of possibilities. Or you could read this data in as a combination of different types: for example, as 4,000 8-bit (1-byte) integers followed by 1,000 64-bit (8-byte) integers. You determine how MATLAB will interpret the mapped data by setting the `Format` property of the `memmapfile` object when you call its constructor.

Note MATLAB arrays are stored on disk in column-major order. (The sequence of array elements is column 1, row 1; column 1, row 2; column 1, last row; column 2, row 1, and so on.) You might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Note The Sol64 platform supports aligned data access only. If you attempt to use a `memmapfile` format on Sol64 that does not take the necessary alignment considerations into account, MATLAB generates an error. (See "Aligned Access on Sol64" on page 6-26).

Data types supported for the `Format` property are shown at the end of this section. See "Supported Data Types for the Format Property" on page 6-40.

For more information on format options see

- "Mapping a Single Data Type" on page 6-35
- "Formatting the Mapped Data to an Array" on page 6-36
- "Mapping Multiple Data Types and Arrays" on page 6-38

Mapping a Single Data Type. If the file region being mapped contains data of only one type, specify the `Format` value as a character string identifying that type:

```
objname = memmapfile(filename, 'Format', datatype)
```

The following command constructs a `memmapfile` object for the entire file `records.dat`, and sets the `Format` property for that object to `uint64`. Any read or write operations made via the memory map will read and write the file contents as a sequence of unsigned 64-bit integers:

```
m = memmapfile('records.dat', 'Format', 'uint64')
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 0
      Format: 'uint64'
      Repeat: Inf
        Data: 5000x1 uint64 array
```

You can change the value of the `Format` property at any time after the `memmapfile` object is constructed. Use the `object.property` syntax shown here in assigning the new value:

```
m.Format = 'int32';
```

Further read and write operations to the region mapped by `m` will now treat the data in the file as a sequence of signed 32-bit integers.

Property names, like `Format`, are not case sensitive.

Formatting the Mapped Data to an Array. You can also specify an array shape to be applied to the data read or written to the mapped file, and a field name to be used in referencing this array. Use a cell array to hold these values either when calling the `memmapfile` constructor or when modifying `m.Format` after the object has been constructed. The cell array contains three elements: the data type to be applied to the mapped region, the dimensions of the array shape that is applied to the region, and a field name to use in referencing the data:

```
objname = memmapfile(filename, ...
            'Format', {datatype, dimensions, varname})
```

The command below constructs a `memmapfile` object for a region of `records.dat` such that the contents of the region are handled by MATLAB as a 4-by-10-by-18 array of unsigned 32-bit integers, and can be referenced in the structure of the returned object using the field name `x`:

```
m = memmapfile('records.dat',   ...
      'Offset', 1024,           ...
      'Format', {'uint32' [4 10 18] 'x'})
m =
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 1024
      Format: {'uint32' [4 10 18] 'x'}
      Repeat: Inf
        Data: 13x1 struct array with fields:
                x

A = m.Data(1).x;

whos A
  Name      Size                            Bytes  Class

  A        4x10x18                          2880   uint32 array

Grand total is 720 elements using 2880 bytes
```

You can change the data type, array shape, or field name that MATLAB applies to the mapped region at any time by setting a new value for the `Format` property of the object:

```
m.Format = {'uint64' [30 4 10] 'x'};
A = m.Data(1).x;

whos A
  Name      Size                            Bytes  Class

  A        30x4x10                           9600  uint64 array

Grand total is 1200 elements using 9600 bytes
```

Mapping Multiple Data Types and Arrays. If the region being mapped is composed of segments of varying data types or array shapes, you can specify an individual format for each segment using an N-by-3 cell array, where N is the number of segments. The cells of each cell array row identify the data type for that segment, the array dimensions to map the data to, and a field name by which to reference that segment:

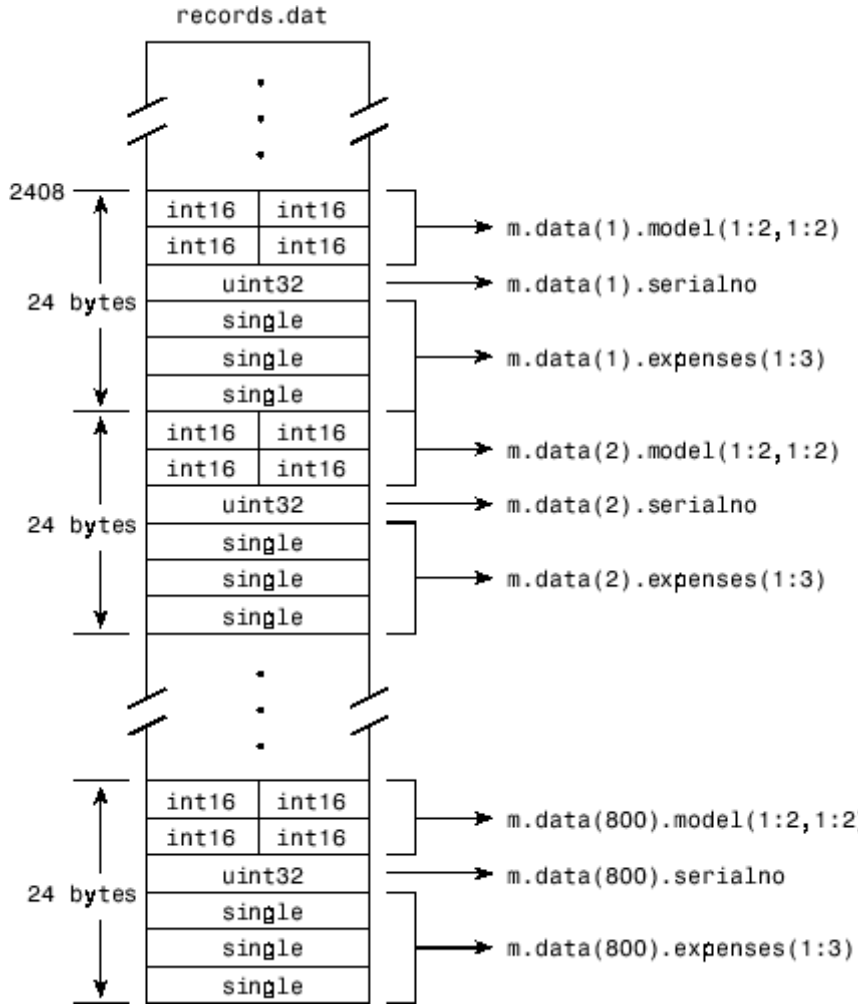
```
objname = memmapfile(filename,                          ...
            'Format', {                                 ...
               datatype1, dimensions1, fieldname1;      ...
               datatype2, dimensions2, fieldname2;      ...
                   :          :            :           ...
               datatypeN, dimensionsN, fieldnameN})
```

The following command maps a 24 kilobyte file containing data of three different data types: `int16`, `uint32`, and `single`. The `int16` data is mapped as a 2-by-2 matrix that can be accessed using the field name `model`. The `uint32` data is a scalar value accessed as field `serialno`. The `single` data is a 1-by-3 matrix named `expenses`.

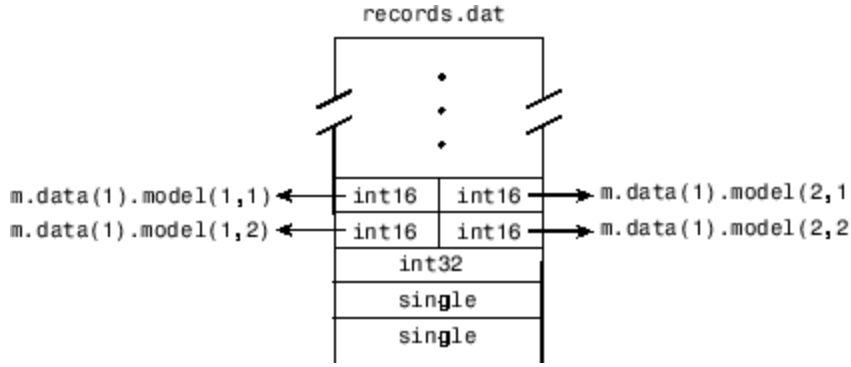
Each of these fields belongs to the 800-by-1 structure array `m.Data`:

```
m = memmapfile('records.dat',        ...
      'Offset', 2048,                ...
      'Format', {                    ...
         'int16'  [2 2] 'model';     ...
         'uint32' [1 1] 'serialno';  ...
         'single' [1 3] 'expenses'});
```

Mapping of the Example File



The figure below shows the ordering of the array elements more closely. In particular, it illustrates that MATLAB arrays are stored on the disk in column-major order. The sequence of array elements in the mapped file is row 1, column 1; row 2, column 1; row 1, column 2; and row 2, column 2.



If the data in your file is not stored in this order, you might need to transpose or rearrange the order of array elements when reading or writing via a memory map.

Supported Data Types for the Format Property. Any of the following data types can be used when you specify a `Format` value. The default type is `uint8`.

Format String	Data Type Description
`'int8'`	Signed 8-bit integers
`'int16'`	Signed 16-bit integers
`'int32'`	Signed 32-bit integers
`'int64'`	Signed 64-bit integers
`'uint8'`	Unsigned 8-bit integers
`'uint16'`	Unsigned 16-bit integers
`'uint32'`	Unsigned 32-bit integers
`'uint64'`	Unsigned 64-bit integers
`'single'`	32-bit floating-point
`'double'`	64-bit floating-point

Repeating a Format Scheme

Once you have set a `Format` value for the `memmapfile` object, you can have MATLAB apply that format to the file data multiple times by specifying a `Repeat` value when you call the `memmapfile` constructor:

```
objname = memmapfile(filename,    ...
            'Format', formatspec, ...
            'Repeat', count)
```

The `Repeat` value applies to the whole format specifier, whether that specifier describes just a single data type that repeats, or a more complex format that includes various data types and array shapes. The default `Repeat` value is infinity (`inf`), which means that the full extent of the `Format` specifier repeats as many times as possible within the mapped region.

The next example maps a file region identical to that of the previous example, except the pattern of `int16`, `uint32`, and `single` data types is repeated only three times within the mapped region of the file:

```
m = memmapfile('records.dat',         ...
      'Offset', 2048,                 ...
      'Format', {                     ...
         'int16'  [2 2] 'model';      ...
         'uint32' [1 1] 'serialno';   ...
         'single' [1 3] 'expenses'},  ...
      'Repeat', 3);
```

You can change the value of the `Repeat` property at any time. To change the repeat value to 5, type

```
m.Repeat = 5;
```

Property names, like `Repeat`, are not case sensitive.

Keeping the Repeated Format Within the Mapped Region. MATLAB maps only the *full* pattern specified by the `Format` property. If you repeat a format such that it would cause the map to extend beyond the end of the file, then either of two things can happen:

- If you specify a repeat value of `Inf`, then only those repeated segments that fit within the file in their entirety are applied to the map.

- If you specify a repeat value other than `Inf`, and that value would cause the map to extend beyond the end of the file, then MATLAB generates an error.

Considering the last example, if the part of the file from `m.Offset` to the end were 70 bytes (instead of the 72 bytes required to repeat `m.Format` three times) and you used a `Repeat` value of `Inf`, then only two full repetitions of the specified format would have been mapped. The end result would be as if you had constructed the map with this command:

```
m = memmapfile('records.dat',         ...
      'Offset', 2048,                 ...
      'Format', {                     ...
         'int16'  [2 2] 'model';      ...
         'uint32' [1 1] 'serialno';   ...
         'single' [1 3] 'expenses'},  ...
      'Repeat', 2);
```

If `Repeat` were set to 3 and you had only 70 bytes to the end of the file, you would get an error.

Note `memmapfile` does not expand or append to a mapped file. Use standard file I/O functions like `fopen` and `fwrite` to do this.

Setting the Type of Access

You can map a file region to allow either read-only or read and write access to its contents. Pass a `Writable` parameter and value in the `memmapfile` constructor, or set `m.Writable` on an existing object to set the type of access allowed:

```
objname = memmapfile(filename, 'Writable', trueorfalse)
```

The value passed can be either `true` (equal to `logical(1)`) or `false` (equal to `logical(0)`). By default, it is `false`, meaning that the mapped region is read only.

To map a read and write region of the file `records.dat` in memory, type

```
m = memmapfile('records.dat', 'Writable', true);
```

Note To successfully modify the file you are mapping to, you must have write permission for that file. If you do not have write permission, you can still set the `Writable` property to `true`, but attempting to write to the file generates an error.

You can change the value of the `Writable` property at any time. To make the memory map to `records.dat` read only, type

```
m.Writable = false;
```

Property names, like `Writable`, are not case sensitive.

Reading a Mapped File

The most commonly used property of the `memmapfile` class is the `Data` property. It is through this property of the memory-map object that MATLAB provides all read and write access to the contents of the mapped file.

The actual mapping of a file to the MATLAB address space does not take place when you construct a `memmapfile` object. A memory map, based on the information currently stored in the mapped object, is generated the first time you reference or modify the `Data` property for that object.

Once you have mapped a file to memory, you can read the contents of that file using the same MATLAB statements used to read variables from the MATLAB workspace. By accessing the `Data` property of the memory map object, the contents of the mapped file appear as if they were an array in the currently active workspace. You simply index into this array to read the desired data from the file.

This section covers the following topics:

Improving Performance

MATLAB accesses data in structures more efficiently than it does data contained in objects. The main reason is that structures do not require the extra overhead of a `subsref` routine. Instead of reading directly from the `memmapfile` object, as shown here

```
for k = 1 : N
   y(k) = m.Data(k);
end
```

you will get better performance when you assign the `Data` field to a variable and then read or write the mapped file through this variable, as shown in this second example:

```
dataRef = m.Data;
for k = 1 : N
   y(k) = dataRef(k);
end
```

Example 1 — Reading a Single Data Type

This example maps a file of 100 double-precision floating-point numbers to memory. The map begins 1024 bytes from the start of the file, and ends 800 bytes (8 bytes per `double` times a `Repeat` value of 100) from that point.

If you haven't done so already, generate a test data file for use in the following examples by executing the `gendatafile` function defined under "Constructing a memmapfile Object" on page 6-29:

```
gendatafile('records.dat', 5000);
```

Now, construct the `memmapfile` object `m`, and show the format of its `Data` property:

```
m = memmapfile('records.dat', 'Format', 'double', ...
      'Offset', 1024, 'Repeat', 100);
```

```
d = m.Data;

whos d
  Name      Size                    Bytes  Class

  d       100x1                       800  double array

Grand total is 100 elements using 800 bytes
```

Read a selected set of numbers from the file by indexing into the single-precision array `m.Data`:

```
d(15:20)
ans =
  1.0e+009 *
    3.6045
    2.7006
    0.5745
    0.8896
    2.6079
    2.7053
```

Example 2 — Formatting File Data as a Matrix

This example is similar to the last, except that the constructor of the `memmapfile` object now specifies an array shape of 4-by-6 to be applied to the data as it is read from the mapped file. MATLAB maps the file contents into a structure array rather than a numeric array, as in the previous example:

```
m = memmapfile('records.dat',            ...
      'Format', {'double', [4 6], 'x'},  ...
      'Offset', 1024, 'Repeat', 100);

d = m.Data;

whos d
  Name      Size                    Bytes  Class

  d       100x1                     25264  struct array

Grand total is 2500 elements using 25264 bytes
```

When you read an element of the structure array, MATLAB presents the data in the form of a 4-by-6 array:

```
d(5).x
ans =
  1.0e+009 *
    3.1564    0.6684    2.1056    1.9357    1.2773    4.2219
    2.9520    0.8208    3.5044    1.7705    0.2112    2.3737
    1.4865    1.8144    1.9790    3.8724    2.9772    1.7183
    0.7131    3.6764    1.9643    0.0240    2.7922    0.8538
```

To index into the structure array field, use

```
d(5).x(3,2:6)
ans =
  1.0e+009 *
    1.8144    1.9790    3.8724    2.9772    1.7183
```

Example 3 — Reading Multiple Data Types

This example maps a file containing more than one data type. The different data types contained in the file are mapped as fields of the returned structure array `m.Data`.

The `Format` parameter passed in the constructor specifies that the first 80 bytes of the file are to be treated as a 5-by-8 matrix of `uint16`, and the 160 bytes after that as a 4-by-5 matrix of `double`. This pattern repeats until the end of the file is reached. The example shows different ways of reading the `Data` property of the object.

Start by calling the `memmapfile` constructor to create a memory map object, `m`:

```
m = memmapfile('records.dat',  ...
      'Format', {              ...
         'uint16' [5 8] 'x';   ...
         'double' [4 5] 'y' });
```

If you examine the `Data` property, MATLAB shows a 166-element structure array with two fields, one for each format specifier in the constructor:

```
d = m.Data
```

```
ans =
166x1 struct array with fields:
    x
    y
```

Examine one structure in the array to show the format of each field:

```
d(3)
ans =
    x: [5x8 uint16]
    y: [4x5 double]
```

Now read the `x` and `y` fields of that structure from the file. MATLAB formats the first block of data as a 5-by-8 matrix of `uint16`, as specified in the `Format` property, and the second block as a 4-by-5 matrix of `double`:

```
d(3).x
ans =
  34432  47500  19145  16868  38165  47956  35550  16853
  60654  51944  16874  47166  35397  58072  16850  56576
  51075  16876  12471  34369   8341  16853  44509  57652
  16863  16453   6666  11480  16869  58695  36217   5932
  57883  15551  41755  16874  37774  31693  54813  16865

d(3).y
ans =
  1.0e+009 *
    3.1229    1.5909    2.9831    2.2445    1.1659
    1.3284    3.0182    2.6685    3.7802    1.0837
    3.6013    2.3475    3.4137    0.7428    3.7613
    2.4399    1.9107    4.1096    4.2080    3.1667
```

Example 4 — Modifying Map Parameters

This example plots the Fourier transform output of data read from a file via a memory map. It then modifies several parameters of the existing map, reads from a different part of the data file, and plots a histogram from that data.

Create a memory-mapped object, mapping 1,000 elements of type `double` starting at the 1025th byte:

```
m = memmapfile('mybinary.bin', 'Offset', 1024,  ...
      'Format', 'double', 'Repeat', 1000);
```

Get data associated with the map and plot the FFT of the first 1000 values of the map. This is when the map is actually created, because no data has been referenced until this point:

```
plot(abs(fft(m.Data(1:1000))));
```

Get information about the memory map:

```
mapStruct = get(m)

mapStruct =
    Filename: 'd:\matlab\mfiles\mybinary.bin'
    Writable: 0
      Offset: 1024
      Format: 'double'
      Repeat: 1000
        Data: [1000x1 double]
```

Change the map, but continue using the same file:

```
m.Offset = 4096;
m.Format = 'single';
m.Repeat = 800;
```

Read from a different area of the file, and plot a histogram of the data. This maps a new region and unmaps the previous region:

```
hist(m.Data)
```

Writing to a Mapped File

Writing to a mapped file is done with standard MATLAB subscripted assignment commands. To write to a particular location in the file mapped to `memmapfile` object `m`, assign the value to the `m.Data` structure array index and field that map to that location.

If you haven't done so already, generate a test data file for use in the following examples by executing the `gendatafile` function defined under "Constructing a memmapfile Object" on page 6-29:

```
gendatafile('records.dat', 5000);
```

Now call the `memmapfile` constructor to create the object:

```
m = memmapfile('records.dat',  ...
      'Format', {              ...
         'uint16' [5 8] 'x';   ...
         'double' [4 5] 'y' });
```

If you are going to modify the mapped file, be sure that you have write permission, and that you set the `Writable` property of the `memmapfile` object to `true` (logical 1):

```
m.Writable = true;
```

(You do not have to set `Writable` as a separate command, as done here. You can include a `Writable` parameter-value argument in the call to the `memmapfile` constructor.)

Read from the 5-by-8 matrix `x` at `m.Data(2)`:

```
d = m.Data;

d(2).x
ans =
  35330   4902  31861  16877  23791  61500  52748  16841
  51314  58795  16860  43523   8957   5182  16864  60110
  18415  16871  59373  61001  52007  16875  26374  28570
  16783   4356  52847  53977  16858  38427  16067  33318
  65372  48883  53612  16861  18882  39824  61529  16869
```

Update all values in that matrix using a standard MATLAB assignment statement:

```
d(2).x = d(2).x * 1.5;
```

Verify the results:

```
d(2).x
ans =
  52995   7353  47792  25316  35687  65535  65535  25262
  65535  65535  25290  65285  13436   7773  25296  65535
  27623  25307  65535  65535  65535  25313  39561  42855
  25175   6534  65535  65535  25287  57641  24101  49977
  65535  65535  65535  25292  28323  59736  65535  25304
```

This section covers the following topics:

Dimensions of the Data Field

The dimensions of a `memmapfile` object's `Data` field are set at the time you construct the object and cannot be changed. This differs from other MATLAB arrays that have dimensions you can modify using subscripted assignment.

For example, you can add a new column to the field of a MATLAB structure:

```
A.s = ones(4,5);

A.s(:,6) = [1 2 3 4];         % Add new column to A.s
size(A.s)
ans =
     4     6
```

But not to a similar field of a structure that represents data mapped from a file. The following assignment to `m.Data(60).y` does not expand the size of `y`, but instead generates an error:

```
m.Data(60)
ans =
    x: [5x8 uint16]
    y: [4x5 double]
```

```
m.Data(60).y(:,6) = [1 2 3 4];        % Generates an error.
```

Thus, if you map an entire file and then append to that file after constructing the map, the appended data is not included in the mapped region. If you need to modify the dimensions of data that you have mapped to a `memmapfile` object, you must either modify the `Format` or `Repeat` properties for the object, or reconstruct the object.

Examples. Two examples of statements that attempt to modify the dimensions of a mapped `Data` field are shown here. These statements result in an error.

The first example attempts to diminish the size of the array by removing a row from the mapped array `m.Data`.

```
m.Data(5) = [];
```

The second example attempts to expand the size of a 50-row mapped array `x` by adding another row to it:

```
m.Data(2).x(1:51,31) = 1:51;
```

Writing Matrices to a Mapped File

The syntax to use when writing to mapped memory can depend on what format was used when you mapped memory to the file.

When Memory Is Mapped in Nonstructure Format. When you map a file as a sequence of a single data type (e.g., a sequence of `uint16`), you can use the following syntax to write matrix `X` to the file:

```
m.Data = X;
```

This statement is valid only if all of the following conditions are true:

- The file is mapped as a sequence of elements of the same data type, making `m.Data` an array of a nonstructure type.
- The class of `X` is the same as the class of `m.Data`.
- The number of elements in `X` equals the number of elements in `m.Data`.

This example maps a file as a sequence of 16-bit unsigned integers, and then uses the syntax shown above to write a matrix to the file. Map only a small part of the file, using a `uint16` format for this segment:

```
m = memmapfile('records.dat', 'Writable', true', ...
    'Offset', 2000, 'Format', 'uint16', 'Repeat', 15);
```

Create a matrix `X` of the same size and write it to the mapped part of the file:

```
X = uint16(5:5:75);    % Sequence of 5 to 75, counting by fives.
m.data = X;
```

Verify that new values were written to the file:

```
m.offset = 1980;   m.repeat = 35;
reshape(m.data,5,7)'
ans =
  29158  16841  32915  37696    421      % <== At offset 1980
  16868  51434  17455  30645  16871
      5     10     15     20     25      % <== At offset 2000
     30     35     40     45     50
     55     60     65     70     75
  16872  50155  51100  26469  16873
  56776   6257  28746  16877  34374
```

When Memory Is Mapped in Scalar Structure Format. When you map a file as a sequence of a single data type (e.g., a sequence of `uint16`), you can use the following syntax to write matrix `X` to the file:

```
m.Data.f = X;
```

This statement is valid only if all of the following conditions are true:

- The file is mapped as containing multiple data types that do not repeat, making `m.Data` a scalar structure.
- The class of `X` is the same as the class of `m.Data.f`.
- The number of elements in `X` equals that of `m.Data.f`.

This example maps a file as a 300-by-8 matrix of type `uint16` followed by a 200-by-5 matrix of type `double`, and then uses the syntax shown above to write a matrix to the file.

```
m = memmapfile('records.dat',     ...
      'Format', {                 ...
         'uint16' [300 8] 'x';    ...
         'double' [200 5] 'y' },  ...
      'Repeat', 1, 'Writable', true);

m.Data.x = ones(300, 8, 'uint16');
```

When Memory Is Mapped in Nonscalar Structure Format. When you map a file as a repeating sequence of multiple data types, you can use the following syntax to write matrix `X` to the file, providing that `k` is a scalar index:

```
m.Data(k).field = X;
```

To do this, the following conditions must be true:

- The file is mapped as containing multiple data types that can repeat, making `m.Data` a nonscalar structure.
- `k` is a scalar index.
- The class of `X` is the same as the class of `m.Data(k).field`.
- The number of elements in `X` equals that of `m.Data(k).field`.

This example maps a file as a matrix of type `uint16` followed by a matrix of type `double` that repeat 20 times, and then uses the syntax shown above to write a matrix to the file.

```
m = memmapfile('records.dat',    ...
      'Format', {                ...
         'uint16' [25 8] 'x';    ...
         'double' [15 5] 'y' },  ...
      'Repeat', 20, 'Writable', true);

d = m.Data;

d(12).x = ones(25,8,'uint16');
```

You can write to specific elements of field `x` as shown here:

```
d(12).x(3:5,1:end) = uint16(500);
d(12).x(3:5,1:end)
```

```
ans =
    500    500    500    500    500    500    500    500
    500    500    500    500    500    500    500    500
    500    500    500    500    500    500    500    500
```

Selecting Appropriate Data Types

All of the usual MATLAB indexing and data type rules apply when assigning values to data via a memory map. The data type that you assign to must be big enough to hold the value being assigned. For example,

```
m = memmapfile('records.dat', 'Format', 'uint8', ...
      'Writable', true);

d = m.Data;
d(5) = 300;
```

saturates the `x` variable because `x` is defined as an 8-bit integer:

```
d(5)
ans =
   255
```

Working with Copies of the Mapped Data

In the following code, the data in variable `block2` is a *copy* of the file data mapped by `m.Data(2)`. Because it is a copy, modifying array data in `block2` does not modify the data contained in the file:

First, destroy the `memmapfile` object and restore the test file `records.dat`, since it has been modified by running the previous examples:

```
clear m
gendatafile('records.dat',50000);
```

Map the file as a series of `uint16` and `double` matrices and make a copy of `m.Data(2)` in `block2`:

```
m = memmapfile('records.dat',   ...
      'Format', {               ...
         'uint16' [5 8] 'x';    ...
         'double' [4 5] 'y' });
```

```
d = m.Data;
```

Write all zeros to the copy:

```
d(2).x(1:5,1:8) = 0;

d(2).x
ans =
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

Verify that the data in the mapped file has not changed even though the copy of `m.Data(2).x` has been written with zeros:

```
m.Data(2).x
ans =
  35330   4902  31861  16877  23791  61500  52748  16841
  51314  58795  16860  43523   8957   5182  16864  60110
  18415  16871  59373  61001  52007  16875  26374  28570
  16783   4356  52847  53977  16858  38427  16067  33318
  65372  48883  53612  16861  18882  39824  61529  16869
```

Invalid Syntax for Writing to Mapped Memory

Although you can expand the dimensions of a typical MATLAB array by assigning outside its current dimensions, this does not apply to the `Data` property of a `memmapfile` object. The following operation is invalid if `m.Data` has only 100 elements:

```
m.Data(120) = x;
```

If you need to expand the size of the mapped data region, first extend the map by updating the `Format` or `Repeat` property of the `memmapfile` object to reflect the new structure of the data in the mapped file.

Methods of the memmapfile Class

You can use the following methods on objects constructed from the `memmapfile` class.

Syntax	Description
`disp`	Displays properties of the object. The display does not include the object's name.
`get(obj)`	Returns the values of all properties of the `memmapfile` object in a structure array.
`get(obj, property)`	Returns the value of the specified property. `property` can be a string or cell array of strings, each containing a property name.

Using the disp Method

Use the `disp` method to display all properties of a `memmapfile` object. The text displayed includes only the property value, and not the object name or the MATLAB response string, `ans =`.

Construct object `m`:

```
m = memmapfile('records.dat',        ...
      'Offset', 2048,                ...
      'Format', {                    ...
         'int16'  [2 2] 'model';     ...
         'uint32' [1 1] 'serialno';  ...
         'single' [1 3] 'expenses'});
```

and display all of its properties:

```
disp(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: false
      Offset: 2048
      Format: {'int16' [2 2] 'model'
               'uint32' [1 1] 'serialno'
               'single' [1 3] 'expenses'}
      Repeat: Inf
```

```
    Data: 16581x1 struct array with fields:
        model
     serialno
     expenses
```

Using the get Method

You can use the `get` method of the `memmapfile` class to return information on any or all of the object's properties. Specify one or more property names to get the values of specific properties.

This example returns the values of the `Offset`, `Repeat`, and `Format` properties for a `memmapfile` object. Use the `get` method to return the specified property values in a 1-by-3 cell array, `m_props`:

```
m_props = get(m, {'Offset', 'Repeat', 'Format'})
m_props =
    [2048]    [Inf]    {3x3 cell}

m_props{3}
ans =
    'int16'     [1x2 double]    'model'
    'uint32'    [1x2 double]    'serialno'
    'single'    [1x2 double]    'expenses'
```

You can also choose to use the `objname.property` syntax:

```
m_props = {m.Offset, m.Repeat, m.Format}
m_props =
    [2048]    [Inf]    {3x3 cell}
```

To return the values for all properties with `get`, pass just the object name:

```
get(m)
    Filename: 'd:\matlab\mfiles\records.dat'
    Writable: 0
      Offset: 2048
      Format: {3x3 cell}
      Repeat: Inf
        Data: [16581x1 struct]
```

Deleting a Memory Map

It is not necessary to explicitly call a destructor method to clear a `memmapfile` object from memory when you no longer need it. MATLAB calls the destructor for you whenever you do any of the following:

- Reassign another value to the `memmapfile` object's variable
- Clear the object's variable from memory
- Exit the function scope in which the object was created

The Effect of Shared Data Copies On Performance

When you assign the `Data` field of the `memmapfile` object to a variable, MATLAB makes a shared data copy of the mapped data. This is very efficient as no memory actually gets copied. In the following statement, `memdat` is a shared data copy of the data mapped from the file:

```
memdat = m.Data;
```

When you finish using the mapped data, make sure to clear any variables that shared data with the mapped file before clearing the object itself. If you clear the object first, then the sharing of data between the file and dependent variables is broken, and the data assigned to such variables must be copied into memory before the object is destroyed. If access to the mapped file was over a network, then copying this data to local memory can take considerable time. So, if the statement shown above assigns data to the variable `memdat`, you should be sure to clear `memdat` before clearing `m` when you are finished with the object.

Note Keep in mind that the `memmapfile` object can be cleared in any of the three ways described under "Deleting a Memory Map" on page 6-58.

Memory-Mapping Demo

In this demonstration, two separate MATLAB processes communicate with each other by writing and reading from a shared file. They share the file by mapping part of their memory space to a common location in the file. A write operation to the memory map belonging to the first process can be read from the map belonging to the second, and vice versa.

One MATLAB process (running `send.m`) writes a message to the file via its memory map. It also writes the length of the message to byte 1 in the file, which serves as a means of notifying the other process that a message is available. The second process (running `answer.m`) monitors byte 1 and, upon seeing it set, displays the received message, puts it into uppercase, and echoes the message back to the sender.

The send Function

This function prompts you to enter a string and then, using memory-mapping, passes the string to another instance of MATLAB that is running the `answer` function.

Copy the `send` and `answer` functions to files `send.m` and `answer.m` in your current working directory. Begin the demonstration by calling `send` with no inputs. Next, start a second MATLAB session on the same machine, and call the `answer` function in this session. To exit, press **Enter**.

```
function send
% Interactively send a message to ANSWER using memmapfile class.

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:send:cannotOpenFile', ...
              'Cannot open file "%s": %s.', filename, msg);
    end
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Set first byte to zero, indicating a message is not
```

```
        % yet ready.
        m.Data(1) = 0;

        str = input('Enter send string (or RETURN to end): ', 's');

        len = length(str);
        if (len == 0)
            disp('Terminating SEND function.')
            break;
        end

        str = str(1:min(len, 255));  % Message limited to 255 chars.

        % Update the file via the memory map.
        m.Data(2:len+1) = str;
        m.Data(1)=len;

        % Wait until the first byte is set back to zero,
        % indicating that a response is available.
        while (m.Data(1) ~= 0)
            pause(.25);
        end

        % Display the response.
        disp('response from ANSWER is:')
        disp(char(m.Data(2:len+1))')
    end
```

The answer Function

The `answer` function starts a server that, using memory-mapping, watches for a message from `send`. When the message is received, `answer` replaces the message with an uppercase version of it, and sends this new message back to `send`.

To use `answer`, call it with no inputs.

```
function answer
% Respond to SEND using memmapfile class.
```

```
disp('ANSWER server is awaiting message');

filename = fullfile(tempdir, 'talk_answer.dat');

% Create the communications file if it is not already there.
if ~exist(filename, 'file')
    [f, msg] = fopen(filename, 'wb');
    if f ~= -1
        fwrite(f, zeros(1,256), 'uint8');
        fclose(f);
    else
        error('MATLAB:demo:answer:cannotOpenFile', ...
              'Cannot open file "%s": %s.', filename, msg);
    end
end

% Memory map the file.
m = memmapfile(filename, 'Writable', true, 'Format', 'uint8');

while true
    % Wait till first byte is not zero.
    while m.Data(1) == 0
        pause(.25);
    end

    % The first byte now contains the length of the message.
    % Get it from m.
    msg = char(m.Data(2:1+m.Data(1)))';

    % Display the message.
    disp('Received message from SEND:')
    disp(msg)

    % Transform the message to all uppercase.
    m.Data(2:1+m.Data(1)) = upper(msg);

    % Signal to SEND that the response is ready.
    m.Data(1) = 0;
end
```

Running the Demo

Here is what the demonstration looks like when it is run. First, start two separate MATLAB sessions on the same computer system. Call the `send` function in one and the `answer` function in the other to create a map in each of the processes' memory to the common file:

```
% Run SEND in the first MATLAB session.
send
Enter send string (or RETURN to end):


% Run ANSWER in the second MATLAB session.
answer
ANSWER server is awaiting message
```

Next, enter a message at the prompt displayed by the `send` function. MATLAB writes the message to the shared file. The second MATLAB session, running the `answer` function, loops on byte 1 of the shared file and, when the byte is written by `send`, `answer` reads the message from the file via its memory map. The `answer` function then puts the message into uppercase and writes it back to the file, and `send` (waiting for a reply) reads the message and displays it:

```
% SEND writes a message and reads the uppercase reply.
Hello. Is there anybody out there?
response from ANSWER is:
HELLO.  IS THERE ANYBODY OUT THERE?
Enter send string (or RETURN to end):


% ANSWER reads the message from SEND.
Received message from SEND:
Hello.  Is there anybody out there?
```

`send` writes a second message to the file. `answer` reads it, put it into uppercase, and then writes the message to the file:

```
% SEND writes a second message to the shared file.
I received your reply.
response from ANSWER is:
I RECEIVED YOUR REPLY.
Enter send string (or RETURN to end): <Enter>
```

```
Terminating SEND function.


% ANSWER reads the second message.
Received message from SEND:
I received your reply.
```

Exporting Data to MAT-Files

In this section...

MAT-Files

MAT-files are double-precision, binary, MATLAB format files. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. They can also be manipulated by other programs external to MATLAB.

This section explains how to save the variables in your MATLAB session to a binary file called a MAT-file. The next section explains how to load them back into your MATLAB workspace.

Using the save Function

To export workspace variables to a binary or ASCII file, use the `save` function. You can save all variables from the workspace in a single operation (if you omit the filename, MATLAB uses the name `matlab.mat`):

```
save filename
```

or save just those variables that you specify:

```
save filename var1 var2 ... varN
```

Use the wildcard character (*) in the variable name to save those variables that match a specific pattern. For example, the following command saves all variables that start with `str`.

```
save strinfo str*
```

Use `whos -file` to examine what has been written to the MAT-file:

```
whos -file strinfo
  Name           Size                    Bytes  Class

  str2           1x15                       30  char array
  strarray       2x5                       678  cell array
  strlen         1x1                         8  double array
```

Saving Structures

When saving a MATLAB structure, you have the option of saving the entire structure, saving each structure field as an individual variable in the MAT-file, or saving specific fields as individual variables.

For structure `S`,

```
S.a = 12.7;  S.b = {'abc', [4 5; 6 7]};  S.c = 'Hello!';
```

Save the entire structure to `newstruct.mat` with the usual syntax:

```
save newstruct.mat S;

whos -file newstruct
  Name      Size                   Bytes  Class

  S         1x1                      550  struct array
```

Save the fields individually with the `-struct` option:

```
save newstruct.mat -struct S;

whos -file newstruct
```

```
  Name      Size                    Bytes  Class

  a         1x1                         8  double array
  b         1x2                       158  cell array
  c         1x6                        12  char array
```

Or save only selected fields using `-struct` and specifying each field name:

```
save newstruct.mat -struct S a c;

whos -file newstruct
  Name      Size                    Bytes  Class

  a         1x1                         8  double array
  c         1x6                        12  char array
```

Appending to an Existing File

You can add new variables to those already stored in an existing MAT-file by using `save -append`. When you append to a MAT-file, MATLAB first looks in the designated file for each variable name specified in the argument list, or for all variables if no specific variable names are specified. Based on that information, MATLAB does both of the following:

- For each variable that already exists in the MAT-file, MATLAB overwrites its saved value with the new value taken from the workspace.
- For each variable not found in the MAT-file, MATLAB adds that variable to the file and stores its value from the workspace.

Note Saving with the `-append` option does not append additional elements to any arrays that are already saved in the MAT-file.

Data Compression

MATLAB compresses the data that you save to a MAT-file. Data compression can save you a significant amount of storage space when you are working with large files or working over a network.

Data compression is optional, however, and you can disable it either for an individual `save` operation, or for all of your MATLAB sessions. Use the `-v6` option with the `save` function to turn off compression on a per-command basis:

```
save filename -v6
```

To disable data compression for all of your MATLAB sessions, open the **Preferences** dialog, select **General** and then **MAT-Files**, and click the option that is equivalent to the command `save -v6`. See General Preferences for MATLAB in the Desktop Tools and Development Environment documentation for more information.

Note You cannot read a compressed MAT-file with MATLAB versions earlier than Version 7. To write a MAT-file that you will be able to read with one of these versions, save to the file with data compression disabled.

Information returned by the command `whos -file` is independent of whether the variables in that file are compressed or not. The byte counts returned by this command represent the number of bytes data occupies in the MATLAB workspace, and not in the file the data was saved to.

Evaluating When to Compress

You should consider both data set size and the type of data being saved when deciding whether or not to compress the data you save to a file. The benefits of data compression are greater when saving large data sets (over 3 MB), and are usually negligible with smaller data sets. Data that has repeating patterns or more consistent values compresses better than random data. Compressing data that has a random pattern is not recommended as it slows down the performance of `save` and `load` significantly, and offers little benefit in return.

In general, data compression and decompression slows down all `save` and some `load` operations to some extent. In most cases, however, the resulting reduction in file size is worth the additional time spent compressing or decompressing. Because loading is typically done more frequently than saving, `load` is considered to be the most critical of the two operations. Up to a certain threshold (relative to the size of the uncompressed MAT-file), loading a compressed MAT-File is slightly slower than loading an uncompressed

file containing the same data. Beyond that threshold, however, loading the compressed file is *faster*.

For example, say that you have a block of data that takes up 100 MB in memory, and this data has been saved to both a 10 MB compressed file and a 100 MB uncompressed file. When you load each of these files back into the MATLAB workspace, the first 10 MB of data takes the same amount of time to load for each file. Loading the remaining 90 MB from the uncompressed file will take 9 times as long as the first 10 MB, while all that remains to be done with the compressed file is to decompress the data, and this takes a relatively short amount of time.

The loading size threshold is lower for network files, and also varies depending on the type of computer being used. Network users loading compressed MAT-files generally see faster load times than when loading uncompressed files, and at smaller data sizes than users loading the same files locally.

Note Compression and decompression during `save` and `load` is done transparently without the use of temporary files on disk. This is of significance to large dataset users in particular.

Unicode Character Encoding

MATLAB saves character data to a MAT-file using Unicode character encoding. As with data compression, Unicode character encoding is optional. If you disable it, MATLAB writes the MAT-file using the default encoding for your system. To disable Unicode character encoding on a per-command basis, use the `-v6` option with the `save` function:

```
save filename -v6
```

To disable Unicode character encoding for all of your MATLAB sessions, open the **Preferences** dialog, select **General** and then **MAT-Files**, and click the option that is equivalent to the command `save -v6`. See General Preferences for MATLAB in the Desktop Tools and Development Environment documentation for more information.

When writing character data to a non-HDF5-based MAT-file using Unicode encoding (the default), MATLAB checks if the data is 7-bit ASCII. If it is,

MATLAB writes the 7-bit ASCII character data to the MAT-file using 8 bits per character (UTF-8 format), thus minimizing the size of the resulting file. Any character data that is not 7-bit ASCII is written in 16-bit Unicode form (UTF-16). This algorithm operates on a per-string basis.

Note You cannot read a Unicode encoded MAT-file with MATLAB versions earlier than Version 7. To write a MAT-file that you will be able to read with one of these versions, save to the file with Unicode character encoding disabled.

For more information on how MATLAB saves specific ASCII data formats, and on preventing loss or corruption of character data, see "Writing Character Data" in the MATLAB External Interfaces documentation.

Optional Output Formats

You can choose from any of the following formats for your output file. If you do not specify a format, MATLAB uses the binary MAT-file format.

Output File Format	Command
Binary MAT-file (default)	`save` *`filename`*
8-digit ASCII	`save` *`filename`* `-ascii`
8-digit ASCII, tab delimited	`save` *`filename`* `-ascii -tabs`
16-digit ASCII	`save` *`filename`* `-ascii -double`
16-digit ASCII, tab delimited	`save` *`filename`* `-ascii -double -tabs`
MATLAB Version 4 compatible	`save` *`filename`* `-v4`

Saving in ASCII Format

When saving in any of the ASCII formats, consider the following:

- Each variable to be saved must be either a two-dimensional double array or a two-dimensional character array. Saving a complex double array causes the imaginary part of the data to be lost, as MATLAB cannot load nonnumeric data (`'i'`).

- To read the file with the MATLAB `load` function, make sure all the variables have the same number of columns. If you are using a program other than MATLAB to read the saved data, this restriction can be relaxed.
- Each MATLAB character in a character array is converted to a floating-point number equal to its internal ASCII code and written out as a floating-point number string. There is no information in the saved file that indicates whether the value was originally a number or a character.
- The values of all variables saved merge into a single variable that takes the name of the ASCII file (minus any extension). Therefore, it is advisable to save only one variable at a time.

Saving in Version 4 Format

With the `-v4` option, you can save only those data constructs that are compatible with MATLAB Version 4. Therefore, you cannot save structures, cell arrays, multidimensional arrays, or objects. Variable names cannot exceed 19 characters in length. In addition, you must use filenames that are supported by MATLAB Version 4.

Storage Requirements

The binary formats used by `save` depend on the size and type of each array. Arrays with any noninteger entries and arrays with 10,000 or fewer elements are saved in floating-point formats requiring 8 bytes per real element. Arrays with all integer entries and more than 10,000 elements are saved in the formats shown, requiring fewer bytes per element.

Element Range	Bytes per Element
0 to 255	1
0 to 65535	2
-32767 to 32767	2
-2^{31} to $2^{31}-1$	4
Other	8

Saving From External Programs

The *MATLAB External Interfaces* documentation provides details on reading and writing MAT-files from external C or Fortran programs. It is important to use recommended access methods, rather than rely upon the specific MAT-file format, which is likely to change in the future.

Importing Data From MAT-Files

In this section...

Using the load Function

To import variables from a binary or ASCII file on your disk to your workspace, use the `load` function. You can load all variables from the workspace in a single operation (if you omit the filename, MATLAB loads from file `matlab.mat`):

```
load filename
```

or load just those variables that you specify:

```
load filename var1 var2 ... varN
```

Use the wildcard character (*) in the variable name to load those variables that match a specific pattern. (This works for MAT-files only.) For example, the following command loads all variables that start with `str` from file `strinfo.mat`:

```
load strinfo str*
```

Caution When you import data into the MATLAB workspace, it overwrites any existing variable in the workspace with the same name.

Previewing MAT-File Contents

To see what variables are stored in a MAT-file before actually loading the file into your workspace, use `whos -file filename`. This command returns the name, dimensions, size, and data type of all variables in the specified MAT-file.

You can use `whos -file` on binary MAT-files only:

```
whos -file mydata.mat
  Name           Size             Bytes  Class

  javArray      10x1                     java.lang.Double[][]
  spArray        5x5                84  double array (sparse)
  strArray       2x5               678  cell array
  x              3x2x2              96  double array
  y              4x5              1230  cell array
```

Loading Into a Structure

To load MAT-file data into a MATLAB structure, specify an output variable in your `load` command. This example reads the data in `mydata.mat` into the fields of structure `S`:

```
S = load('mydata.mat')
S =
           x: [3x2x2 double]
           y: {4x5 cell}
     spArray: [5x5 double]
    strArray: {2x5 cell}
    javArray: [10x1 java.lang.Double[][]]

whos S
  Name      Size                    Bytes  Class

  S         1x1                      2840  struct array
```

Loading Binary Data

MAT-files are double-precision binary MATLAB format files created by the `save` function and readable by the `load` function. They can be created on one machine and later read by MATLAB on another machine with a different floating-point format, retaining as much accuracy and range as the different formats allow. They can also be manipulated by other programs, external to MATLAB.

MAT-files can contain data in an uncompressed or a compressed form, or both. MATLAB knows which variables in the file have been compressed by looking at a tag that it attaches to each variable during the `save` operation. When

loading data from a MAT-file into the workspace, MATLAB automatically handles the decompression of the appropriate data.

The External Interface libraries contain C- and Fortran-callable routines to read and write MAT-files from external programs.

Loading ASCII Data

ASCII files must be organized as a rectangular table of numbers, with each number in a row separated by a blank, comma, or tab character, and with an equal number of elements in each row. MATLAB generates an error if the number of values differs between any two rows. ASCII files can contain MATLAB comments (lines that begin with `%`).

MATLAB returns all the data in the file as a single two-dimensional array of type `double`. The number of rows in the array is equal to the number of lines in the file, and the number of columns is equal to the number of values on a line.

In the workspace, MATLAB assigns the array to a variable named after the file being loaded (minus any file extension). For example, the command

```
load mydata.dat
```

reads all of the data from `mydata.dat` into the MATLAB workspace as a single array, and assigns it to a variable called `mydata`. In naming the variable, `load` precedes any leading underscores or digits in `filename` with an `X` and replaces any other nonalphabetic characters with underscores.

For example, the command

```
load 10-May-data.dat
```

assigns the data in file `10-May-data.dat` to a new workspace variable called `X10_May_data`.

Importing Text Data

In this section...

Caution When you import data into the MATLAB workspace, you overwrite any existing variable in the workspace with the same name.

The MATLAB Import Wizard

The easiest way to import data into MATLAB is to use the Import Wizard. You do not need to know the format of the data to use this tool. You simply specify the file that contains the data and the Import Wizard processes the file contents automatically.

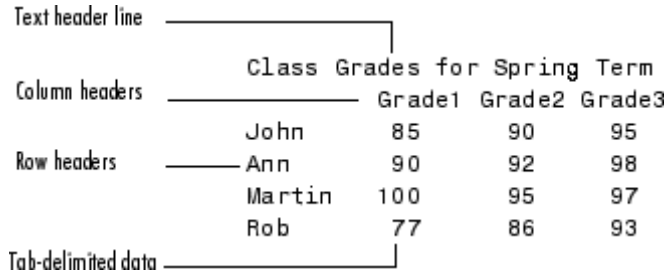
For more information, see "Using the Import Wizard" on page 6-11.

Using Import Functions with Text Data

To import text data from the command line or in an M-file, you must use one of the MATLAB import functions. Your choice of function depends on how the data in the text file is formatted.

The text data must be formatted in a uniform pattern of rows and columns, using a text character, called a *delimiter* or *column separator*, to separate each data item. The delimiter can be a space, comma, semicolon, tab, or any other character. The individual data items can be alphabetic or numeric characters or a mix of both.

The text file can also contain one or more lines of text, called *header lines*, or can use text headers to label each column or row. The following example illustrates a tab-delimited text file with header text and row and column headers.



To find out how your data is formatted, view it in a text editor. After you determine the format, find the sample in the table below that most closely resembles the format of your data. Then read the topic referred to in the table for information on how to import that format.

Table 6-1 ASCII Data File Formats

Data Format Sample	File Extension	Description

`1 2 3 4 5`
`6 7 8 9 10`
	`.txt`
	`.dat`
	or other
		See "Importing Numeric Text Data" on page 6-78 or "Using the Import Wizard" on page 6-11 for information.

`1; 2; 3; 4; 5`
`6; 7; 8; 9; 10`
`or`
`1, 2, 3, 4, 5`
`6, 7, 8, 9, 10`
	`.txt`
	`.dat`
	`.csv`
	or other
		See "Importing Delimited ASCII Data Files" on page 6-79 or "Using the Import Wizard" on page 6-11 for information.

Table 6-1 ASCII Data File Formats (Continued)

Data Format Sample	File Extension	Description

`Ann Type1 12.34 45 Yes`
`Joe Type2 45.67 67 No`
	`.txt`
	`.dat`
	or other
		See "Importing Mixed Alphabetic and Numeric Data" on page 6-81 for information.

`Grade1 Grade2 Grade3`
`91.5  89.2  77.3`
`88.0  67.8  91.0`
`67.3  78.1  92.5`
	`.txt`
	`.dat`
	or other
		See "Importing Numeric Data with Text Headers" on page 6-80 or "Using the Import Wizard" on page 6-11 for information.

If you are familiar with MATLAB import functions but are not sure when to use them, see the following table, which compares the features of each function.

Table 6-2 ASCII Data Import Function Features

Function	Data Type	Delimiters	Number of Return Values	Notes
`csvread`	Numeric data	Commas only	One	Primarily used with spreadsheet data. See "Working with Spreadsheets" on page 6-98.
`dlmread`	Numeric data	Any character	One	Flexible and easy to use.

Table 6-2 ASCII Data Import Function Features (Continued)

Function	Data Type	Delimiters	Number of Return Values	Notes
`fscanf`	Alphabetic and numeric; however, both types returned in a single return variable	Any character	One	Part of low-level file I/O routines. Requires use of `fopen` to obtain file identifier and `fclose` after read.
`load`	Numeric data	Spaces only	One	Easy to use. Use the functional form of `load` to specify the name of the output variable.
`textread`	Alphabetic and numeric	Any character	Multiple values in cell arrays	Flexible, powerful, and easy to use. Use format string to specify conversions.
`textscan`	Alphabetic and numeric	Any character	Multiple values returned to one cell array	More flexible than `textread`. Also more format options.

Importing Numeric Text Data

If your data file contains only numeric data, you can use many of the MATLAB import functions (listed in ASCII Data Import Function Features on page 6-77), depending on how the data is delimited. If the data is rectangular, that is, each row has the same number of elements, the simplest command to use is the `load` command. (The `load` function can also be used to import MAT-files, the MATLAB binary format for saving the workspace.)

For example, the file named `my_data.txt` contains two rows of numbers delimited by space characters:

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

When you use `load` as a command, it imports the data and creates a variable in the workspace with the same name as the filename, minus the file extension:

```
load my_data.txt;
whos
   Name          Size         Bytes   Class

   my_data       2x5           80    double array

my_data

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

If you want to name the workspace variable something other than the filename, use the functional form of `load`. In the following example, the data from `my_data.txt` is loaded into the workspace variable `A`:

```
A = load('my_data.txt');
```

Importing Delimited ASCII Data Files

If your data file uses a character other than a space as a delimiter, you have a choice of several import functions you can use. (See ASCII Data Import Function Features on page 6-77 for a complete list.) The simplest to use is the `dlmread` function.

For example, consider a file named `ph.dat` whose contents are separated by semicolons:

```
7.2;8.5;6.2;6.6
5.4;9.2;8.1;7.2
```

To read the entire contents of this file into an array named `A`, enter

```
A = dlmread('ph.dat', ';');
```

You specify the delimiter used in the data file as the second argument to `dlmread`. Note that, even though the last items in each row are not followed by a delimiter, `dlmread` can still process the file correctly. `dlmread` ignores space characters between data elements. So, the preceding `dlmread` command works even if the contents of `ph.dat` are

```
7.2;   8.5;       6.2;6.6
5.4;   9.2   ;8.1;7.2
```

Importing Numeric Data with Text Headers

To import an ASCII data file that contains text headers, use the `textscan` function, specifying the `headerlines` parameter. `textscan` accepts a set of predefined parameters that control various aspects of the conversion. (For a complete list of these parameters, see the `textscan` reference page.) Using the `headerlines` parameter, you can specify the number of lines at the head of the file that `textscan` should ignore.

For example, the file `grades.dat` contains formatted numeric data with a one-line text header:

```
Grade1 Grade2 Grade3
  78.8   55.9   45.9
  99.5   66.8   78.0
  89.5   77.0   56.7
```

To import this data, first open the file and then use this `textscan` command to read the contents:

```
fid = fopen('grades.dat', 'r');
grades = textscan(fid, '%f %f %f', 3, 'headerlines', 1);

grades{:}
ans =
   78.8000
   99.5000
   89.5000

ans =
   55.9000
```

```
    66.8000
    77.0000

ans =
    45.9000
    78.0000
    56.7000

fclose(fid);
```

Importing Mixed Alphabetic and Numeric Data

If your data file contains a mix of alphabetic and numeric ASCII data, use the `textscan` or `textread` function to import the data. `textscan` returns its output in a single cell array, while `textread` returns its output in separate variables and you can specify the data type of each variable. The `textscan` function offers better performance than `textread`, making it a better choice when reading large files.

This example uses `textread` to import the file `mydata.dat` that contains a mix of alphabetic and numeric data:

```
Sally    Type1 12.34 45 Yes
Larry    Type2 34.56 54 Yes
Tommy    Type1 67.89 23 No
```

Note To read an ASCII data file that contains numeric data with text column headers, see "Importing Numeric Data with Text Headers" on page 6-80.

To read the entire contents of the file `mydata.dat` into the workspace, specify the name of the data file and the format string as arguments to `textread`. In the format string, you include conversion specifiers that define how you want each data item to be interpreted. For example, specify `%s` for string data, `%f` for floating-point data, and so on. (For a complete list of format specifiers, see the `textread` reference page.)

For each conversion specifier in your format string, you must specify a separate output variable. `textread` processes each data item in the file as specified in the format string and puts the value in the output variable. The

number of output variables must match the number of conversion specifiers in the format string.

In this example, `textread` reads the file `mydata.dat`, applying the format string to each line in the file until the end of the file:

```
[names, types, x, y, answer] = ...
   textread('mydata.dat', '%s %s %f %d %s', 3)
names =
    'Sally'
    'Larry'
    'Tommy'

types =
    'Type1'
    'Type2'
    'Type1'

x =
   12.3400
   34.5600
   67.8900

y =
    45
    54
    23

answer =
    'Yes'
    'Yes'
    'No'
```

If your data uses a character other than a space as a delimiter, you must use the `textread` parameter `'delimiter'` to specify the delimiter. For example, if the file `mydata.dat` used a semicolon as a delimiter, you would use this command:

```
[names, types, x, y, answer]= ...
   textread('mydata.dat', '%s %s %f %d %s', 'delimiter', ';')
```

See the `textread` reference page for more information about these optional parameters.

Importing from XML Documents

With the `xmlread` function, you can read from a given URL or file, generating a Document Object Model (DOM) node to represent the parsed document.

MATLAB also provides these other XML functions:

- `xmlwrite` — Serializes a Document Object Model node to a file
- `xslt` — Transforms an XML document using an XSLT engine

See the reference pages for these functions for more information.

Exporting Text Data

In this section...

Overview

This section describes how to use MATLAB functions to export data in several common ASCII formats. For example, you can use these functions to export a MATLAB matrix as a text file where the rows and columns are represented as space-separated, numeric values. The function you use depends on the amount of data you want to export and its format. Topics covered include

If you are not sure which section describes your data, find the sample in the table below that most nearly matches the data format you want to create. Then read the section referred to in the table.

If you are familiar with MATLAB export functions but are not sure when to use them, see ASCII Data Export Function Features on page 6-85, which compares the features of each function.

Note If C or Fortran routines for writing data files in the form needed by other applications exist, create a MEX-file to write the data. See the *MATLAB External Interfaces* documentation for more information.

Table 6-3 ASCII Data File Formats

Data Format Sample	MATLAB Export Function
`1 2 3 4 5 6` `7 8 9 10`	See "Exporting Delimited ASCII Data Files" on page 6-86 and "Using the diary Function to Export Data" on page 6-87 for information about these options.

`1; 2; 3; 4;`
`5; 6; 7; 8;`
`9; 10;`
	See "Exporting Delimited ASCII Data Files" on page 6-86 for information. The example shows a semicolon-delimited file, but you can specify another character as the delimiter.

Table 6-4 ASCII Data Export Function Features

Function	Use With	Delimiters	Notes
`csvwrite`	Numeric data	Commas only	Primarily used with spreadsheet data. See "Working with Spreadsheets" on page 6-98.
`diary`	Numeric data or cell array	Spaces only	Can be used for small arrays. Requires editing of data file to remove extraneous text.
`dlmwrite`	Numeric data	Any character	Easy to use, flexible.

Table 6-4 ASCII Data Export Function Features (Continued)

Function	Use With	Delimiters	Notes
`fprintf`	Alphabetic and numeric data	Any character	Part of low-level file I/O routines. This function is the most flexible but also the most difficult to use. You must use `fopen` to obtain a file identifier before writing the data and `fclose` to close the file after writing the data.
`save`	Numeric data	Tabs or spaces	Easy to use; output values are high precision.

Exporting Delimited ASCII Data Files

To export an array as a delimited ASCII data file, you can use either the `save` function, specifying the `-ASCII` qualifier, or the `dlmwrite` function. The `save` function is easy to use; however, the `dlmwrite` function provides more flexibility, allowing you to specify any character as a delimiter and to export subsets of an array by specifying a range of values.

Using the save Function

To export the array `A`,

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

use the `save` function, as follows:

```
save my_data.out A -ASCII
```

If you view the created file in a text editor, it looks like this:

```
1.0000000e+000  2.0000000e+000  3.0000000e+000  4.0000000e+000
5.0000000e+000  6.0000000e+000  7.0000000e+000  8.0000000e+000
```

By default, `save` uses spaces as delimiters but you can use tabs instead of spaces by specifying the `-tabs` option.

When you use `save` to write a character array to an ASCII file, it writes the ASCII equivalent of the characters to the file. If you write the character string `'hello'` to a file, save writes the values

```
104 101 108 108 111
```

Using the dlmwrite Function

To export an array in ASCII format and specify the delimiter used in the file, use the `dlmwrite` function.

For example, to export the array `A`,

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

as an ASCII data file that uses semicolons as a delimiter, use this command:

```
dlmwrite('my_data.out',A, ';')
```

If you view the created file in a text editor, it looks like this:

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

Note that `dlmwrite` does not insert delimiters at the end of rows.

By default, if you do not specify a delimiter, `dlmwrite` uses a comma as a delimiter. You can specify a space (`' '`) as a delimiter or, if you specify empty quotes (`''`), no delimiter.

Using the diary Function to Export Data

To export small numeric arrays or cell arrays, you can use the `diary` function. `diary` creates a verbatim copy of your MATLAB session in a disk file (excluding graphics).

For example, if you have the array `A` in your workspace,

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

execute these commands at the MATLAB prompt to export this array using `diary`:

1. Turn on the `diary` function. You can optionally name the output file `diary` creates.

   ```
   diary my_data.out
   ```

2. Display the contents of the array you want to export. This example displays the array `A`. You could also display a cell array or other MATLAB data type.

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

3. Turn off the `diary` function.

   ```
   diary off
   ```

   `diary` creates the file `my_data.out` and records all the commands executed in the MATLAB session until it is turned off.

   ```
   A =

        1     2     3     4
        5     6     7     8

   diary off
   ```

4. Open the `diary` file `my_data.out` in a text editor and remove all the extraneous text.

Exporting to XML Documents

With the `xmlwrite` function, you can serialize a Document Object Model (DOM) node to an XML file.

MATLAB also provides these other XML functions:

- `xmlread` — Imports from a given URL or file to a Document Object Model node

- `xslt` — Transforms an XML document using an XSLT engine

See the reference pages for these functions for more information.

Working with Graphics Files

In this section...

Getting Information About Graphics Files

If you have a file in a standard graphics format, use the `imfinfo` function to get information about its contents. The `imfinfo` function returns a structure containing information about the file. The fields in the structure vary with the file format but `imfinfo` always returns some basic information including filename, last modification date, file size, and format.

This example returns information about a file in Joint Photographic Experts Group (JPEG) format:

```
info = imfinfo('ngc6543a.jpg')

info =

           Filename: [1x57 char]
        FileModDate: '01-Oct-1996 16:19:44'
           FileSize: 27387
             Format: 'jpg'
      FormatVersion: ''
              Width: 600
             Height: 650
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
       CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
            Comment: {[1x69 char]}
```

Importing Graphics Data

To import data into the MATLAB workspace from a graphics file, use the `imread` function. Using this function, you can import data from files in many standard file formats, including the Tagged Image File Format (TIFF), Graphics Interchange Format (GIF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG) formats. For a complete list of supported formats, see the `imread` reference page.

This example reads the image data stored in a file in JPEG format into the MATLAB workspace as the array `I`:

```
I = imread('ngc6543a.jpg');
```

`imread` represents the image in the workspace as a multidimensional array of class `uint8`. The dimensions of the array depend on the format of the data. For example, `imread` uses three dimensions to represent RGB color images:

```
whos I
  Name      Size                          Bytes  Class

  I       650x600x3                     1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
```

Exporting Graphics Data

To export data from the MATLAB workspace using one of the standard graphics file formats, use the `imwrite` function. Using this function, you can export data in formats such as the Tagged Image File Format (TIFF), Joint Photographic Experts Group (JPEG), and Portable Network Graphics (PNG). For a complete list of supported formats, see the `imwrite` reference page.

The following example writes a multidimensional array of `uint8` data `I` from the MATLAB workspace into a file in TIFF format. The class of the output image written to the file depends on the format specified. For most formats, if the input array is of class `uint8`, `imwrite` outputs the data as 8-bit values. See the `imwrite` reference page for details.

```
whos I
  Name      Size                          Bytes  Class
```

```
  I       650x600x3                        1170000  uint8 array

Grand total is 1170000 elements using 1170000 bytes
imwrite(I, 'my_graphics_file.tif','tif');
```

Note `imwrite` supports different syntaxes for several of the standard formats. For example, with TIFF file format, you can specify the type of compression used to store the image. See the `imwrite` reference page for details.

Working with Audio and Video Data

In this section...

Getting Information About Audio/Video Files

MATLAB includes several functions that you can use to get information about files that contain audio data, video data, or both. Some work only with specific file formats. One function, the `mmfileinfo` function, can retrieve information about many file formats.

Format-Specific Functions

MATLAB includes several functions that return information about files that contain audio and video data in specific formats.

- `aufinfo` — Returns a text description of the contents of a sound (AU) file
- `aviinfo` — Returns a structure containing information about the contents of an Audio/Video Interleaved (AVI) file
- `wavfinfo` — Returns a text description of the contents of a sound (WAV) file

Using the General Multimedia Information Function

MATLAB also includes a general-purpose, audio/video file information function named `mmfileinfo`. The `mmfileinfo` function returns information about both the audio data in a file as well as the video data in the file, if present.

Note `mmfileinfo` can be used only on Windows systems.

Importing Audio/Video Data

MATLAB includes several functions that you can use to bring audio or video data into the MATLAB workspace. Some of these functions read audio or video data from files. Another way to import audio data into the MATLAB workspace is to record it using an audio input device, such as a microphone. The following sections describe

Reading Audio and Video Data from a File

MATLAB includes several functions for reading audio or video data from a file. These files are format-specific.

- `auread` — Returns sound data from a sound (AU) file
- `aviread` — Returns AVI data as a MATLAB movie
- `mmreader` — Returns AVI, MPG, or WMV video data
- `wavread` — Returns sound data from a sound (WAV) file

Note `mmreader` can be used only on Windows systems.

Recording Audio Data

To bring sound data into the MATLAB workspace by recording it from an audio input device, use the audio recorder object. This object represents the connection between MATLAB and an audio input device, such as a microphone, that is connected to your system. You use the `audiorecorder` function to create this object and then use methods and properties of the object to record the audio data.

On PCs running Windows, you can also use the `wavrecord` function to bring live audio data in WAV format into the MATLAB workspace.

Once you import audio data, MATLAB supports several ways to listen to the data. You can use an audio player object to play the audio data. Use the `audioplayer` function to create an audio player object.

You can also use the `sound` or `soundsc` function.

On PCs running Windows, you can use the `wavplay` function to listen to `.wav` files.

Exporting Audio/Video Data

MATLAB includes several functions that you can use to export audio or video data from the MATLAB workspace. These functions write audio data to a file using specific file formats. The following sections describe

- "Exporting Audio Data" on page 6-95
- "Exporting Video Data in AVI Format" on page 6-95

This section also provides an example of writing video data to a file in "Example: Creating an AVI file" on page 6-96.

Exporting Audio Data

In MATLAB, audio data is simply numeric data that you can export using standard MATLAB data export functions, such as `save`.

MATLAB also includes several functions that write audio data to files in specific file formats:

- `auwrite` — Exports sound data in AU file format
- `wavwrite` — Exports sound data in WAV file format

Exporting Video Data in AVI Format

You can export MATLAB video data as an Audio/Video Interleaved (AVI) file. To do this, you use the `avifile` function to create an `avifile` object. Once you have the object, you can use AVI file object methods and properties to control various aspects of the data export process.

For example, in MATLAB, you can save a sequence of graphs as a movie that can then be played back using the `movie` function. You can export a MATLAB movie by saving it in MAT-file format, like any other MATLAB workspace variable. However, anyone who wants to view your movie must have MATLAB. (For more information about MATLAB movies, see the Animation section in the MATLAB Graphics documentation.)

To export a sequence of MATLAB graphs in a format that does not require MATLAB for viewing, save the figures in Audio/Video Interleaved (AVI) format. AVI is a file format that allows animation and video clips to be played on a PC running Windows or on UNIX systems.

Note To convert an existing MATLAB movie into an AVI file, use the `movie2avi` function.

Example: Creating an AVI file

To export a sequence of MATLAB graphs as an AVI format movie, perform these steps:

1 Create an AVI file object, using the `avifile` function.

```
aviobj = avifile('mymovie.avi','fps',5);
```

AVI file objects support properties that let you control various characteristics of the AVI movie, such as colormap, compression, and quality. (See the `avifile` reference page for a complete list.) `avifile` uses default values for all properties, unless you specify a value. The example sets the value of the frames per second (`fps`) property.

2 Capture the sequence of graphs and put them into the AVI file, using the `addframe` function.

```
for k=1:25
    h = plot(fft(eye(k+16)));
    set(h,'EraseMode','xor');
    axis equal;
    frame = getframe(gca);
    aviobj = addframe(aviobj,frame);
```

```
end
```

The example uses a `for` loop to capture the series of graphs to be included in the movie. You typically use `addframe` to capture a sequence of graphs for AVI movies. However, because this particular MATLAB animation uses `XOR` graphics, you must call `getframe` to capture the graphs and then call `addframe` to add the captured frame to the movie.

3 Close the AVI file, using the `close` function.

```
aviobj = close(aviobj);
```

Working with Spreadsheets

In this section...

Microsoft Excel Spreadsheets

This section covers

See the `xlsfinfo`, `xlswrite`, and `xlsread` reference pages for more detailed information and examples.

Getting Information About the File

Use the `xlsfinfo` function to determine if a file contains a readable Microsoft Excel spreadsheet.

Inputs to `xlsfinfo` are

- Name of the spreadsheet file

Outputs from `xlsfinfo` are

- String `'Microsoft Excel Spreadsheet'` if the file contains an Excel worksheet readable with the `xlsread` function. Otherwise, it contains an empty string (`''`).
- Cell array of strings containing the names of each worksheet in the file.

Example — Querying an XLS File. This example returns information about spreadsheet file `tempdata.xls`:

```
[type, sheets] = xlsfinfo('tempdata.xls')
```

```
type =
Microsoft Excel Spreadsheet
sheets =
    'Locations'    'Rainfall'    'Temperatures'
```

Exporting to the File

Use the `xlswrite` function to export a matrix to an Excel spreadsheet file. With `xlswrite`, you can export data from the workspace to any worksheet in the file, and to any location within that worksheet.

Inputs to `xlswrite` are

- Name of the spreadsheet file
- Matrix to be exported
- Name of the worksheet to receive the data
- Range of cells on the worksheet in which to write the data

Outputs from `xlswrite` are

- Pass or fail status
- Any warning or error message generated along with its message identifier

Example — Writing To an XLS File. This example writes a mix of text and numeric data to the file `tempdata.xls`. Call `xlswrite`, specifying a worksheet labeled `Temperatures`, and the region within the worksheet to write the data to. The 4-by-2 matrix is written to the rectangular region that starts at cell `E1` in its upper-left corner:

```
d = {'Time', 'Temp'; 12 98; 13 99; 14 97}
d =
    'Time'    'Temp'
    [  12]    [  98]
    [  13]    [  99]
    [  14]    [  97]

xlswrite('tempdata.xls', d, 'Temperatures', 'E1');
```

Adding a New Worksheet. If the worksheet being written to does not already exist in the file, MATLAB displays the following warning:

```
Warning: Added specified worksheet.
```

You can disable these warnings with the command

```
warning off MATLAB:xlswrite:AddSheet
```

Importing from the File

Use `xlsread` to import a matrix from an Excel spreadsheet file into the MATLAB workspace. You can import data from any worksheet in the file, and from any location within that worksheet. You can also optionally have `xlsread` open an Excel window showing the file and then interactively select the worksheet and range of data to be read by the function.

Inputs to `xlsread` are

- Name of the spreadsheet file
- Matrix to be imported
- Name of the worksheet from which to read the data
- Range of cells on the worksheet from which to read the data
- Keyword that opens an Excel window, enabling you to interactively select the worksheet and range of data to read
- Keyword that imports using basic import mode

Three separate outputs from `xlsread` are

- Numeric data
- String data
- Any unprocessed cell content

Example — Reading from an XLS File. Continuing with the previous example, to import only the numeric data, use `xlsread` with a single return argument. `xlsread` ignores any leading row or column of text in the numeric result:

```
ndata = xlsread('tempdata.xls', 'Temperatures')
ndata =
    12    98
    13    99
    14    97
```

To import both numeric data and text data, specify two return values for `xlsread`:

```
[ndata, headertext] = xlsread('tempdata.xls', 'Temperatures')

headertext =
    'Time'    'Temp'

ndata =
    12    98
    13    99
    14    97
```

Lotus 123 Spreadsheets

This section covers

See the `wk1finfo`, `wk1write`, and `wk1read` reference pages for more detailed information and examples.

Getting Information About the File

Use the `wk1finfo` function to determine if a file contains a Lotus WK1 spreadsheet:

Inputs to `wk1finfo` are

- Name of the spreadsheet file

Outputs from `wk1finfo` are

- String 'WK1' if the file is a Lotus spreadsheet readable with the wk1read function. Otherwise, it contains an empty string ('').
- String 'Lotus 123 Spreadsheet'

Example — Querying a WK1 File. This example returns information about spreadsheet file matA.wk1:

```
[extens, type] = wk1finfo('matA.wk1')

extens =
   WK1
type =
   Lotus 123 Spreadsheet
```

Exporting to the File

Use the wk1write function to export a matrix to a Lotus spreadsheet file. You have the choice of positioning the matrix starting at the first row and column of the spreadsheet, or at any other location in the file.

To export to a specific location in the file, use the second syntax, indicating a zero-based starting row and column.

Inputs to wk1write are

- Name of the spreadsheet file
- Matrix to be exported
- Location in the file in which to write the data

Example — Writing to a WK1 File. This example exports an 8-by-8 matrix to spreadsheet file matA.wk1:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48; 51:58; 61:68; 71:78];
A =
     1     2     3     4     5     6     7     8
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
```

```
    41    42    43    44    45    46    47    48
    51    52    53    54    55    56    57    58
    61    62    63    64    65    66    67    68
    71    72    73    74    75    76    77    78

wk1write('matA.wk1', A);
```

Importing from the File

To import data from the spreadsheet into the MATLAB workspace, use `wk1read`. There are three ways to call `wk1read`. The first two shown here are similar to `wk1write`. The third enables you to select a range of values from the spreadsheet. You can specify the `range` argument with a one-based vector, spreadsheet notation (e.g., `'A1..B7'`), or using a named range (e.g., `'Sales'`).

Inputs to `wk1read` are

- Name of the spreadsheet file
- Spreadsheet location from which to read the data
- Range of cells from which to read the data

Outputs from `wk1read` are

- Requested data from the spreadsheet

Example — Reading from a WK1 File. Read in a limited block of the spreadsheet data by specifying the upper-left row and column of the block using zero-based indexing:

```
M = wk1read('matA.wk1', 3, 2)
M =
    33    34    35    36    37    38
    43    44    45    46    47    48
    53    54    55    56    57    58
    63    64    65    66    67    68
    73    74    75    76    77    78
```

Using Low-Level File I/O Functions

In this section...

Overview

MATLAB includes a set of low-level file I/O functions that are based on the I/O functions of the ANSI Standard C Library. If you know C, you are probably familiar with these routines.

To read or write data, perform these steps:

1. Open the file, using `fopen`. `fopen` returns a file identifier that you use with all the other low-level file I/O routines.
2. Operate on the file.
   a. Read binary data, using `fread`.
   b. Write binary data, using `fwrite`.
   c. Read text strings from a file line-by-line, using `fgets` or `fgetl`.
   d. Read formatted ASCII data, using `fscanf`.
   e. Write formatted ASCII data, using `fprintf`.
3. Close the file, using `fclose`.

This section also describes how these functions affect the current position in the file where read or write operations happen and how you can change the position in the file.

Note While the MATLAB file I/O commands are modeled on the C language I/O routines, in some ways their behavior is different. For example, the `fread` function is *vectorized*; that is, it continues reading until it encounters a text string or the end of file. These sections, and the MATLAB reference pages for these functions, highlight any differences in behavior.

Opening Files

Before reading or writing a text or binary file, you must open it with the `fopen` command.

```
fid = fopen('filename','permission')
```

Specifying the Permission String

The `permission` string specifies the kind of access to the file you require. Possible `permission` strings include

- `r` for reading only
- `w` for writing only
- `a` for appending only
- `r+` for both reading and writing

Note Systems such as Microsoft Windows that distinguish between text and binary files might require additional characters in the permission string, such as `'rb'` to open a binary file for reading.

Using the Returned File Identifier (fid)

If successful, `fopen` returns a nonnegative integer, called a *file identifier* (`fid`). You pass this value as an argument to the other I/O functions to access

the open file. For example, this `fopen` statement opens the data file named `penny.dat` for reading:

```
fid = fopen('penny.dat','r')
```

If `fopen` fails, for example if you try to open a file that does not exist, `fopen`

- Assigns `-1` to the file identifier.
- Assigns an error message to an optional second output argument. Note that the error messages are system dependent and are not provided for all errors on all systems. The function `ferror` can also provide information about errors.

Test the file identifier each time you open a file in your code. For example, this code loops until a readable filename is entered:

```
fid=0;
while fid < 1
   filename=input('Open file: ', 's');
   [fid,message] = fopen(filename, 'r');
   if fid == -1
     disp(message)
   end
end
```

When you run this code, if you specify a file that doesn't exist, such as `nofile.mat`, at the `Open file:` prompt, the results are

```
Open file: nofile.mat
Sorry. No help in figuring out the problem . . .
```

If you specify a file that does exist, such as `goodfile.mat`, the code example returns the file identifier, `fid`, and exits the loop.

```
Open file: goodfile.mat
```

Opening Temporary Files and Directories

The `tempdir` and `tempname` functions assist in locating temporary data on your system.

Function	Purpose
`tempdir`	Get temporary directory name.
`tempname`	Get temporary filename.

Use these functions to create temporary files. Some systems delete temporary files every time you reboot the system. On other systems, designating a file as temporary can mean only that the file is not backed up.

The `tempdir` function returns the name of the directory or folder that has been designated to hold temporary files on your system. For example, issuing `tempdir` on a UNIX system returns the `/tmp` directory.

MATLAB also provides a `tempname` function that returns a filename in the temporary directory. The returned filename is a suitable destination for temporary data. For example, if you need to store some data in a temporary file, then you might issue the following command first:

```
fid = fopen(tempname, 'w');
```

Note The filename that `tempname` generates is not guaranteed to be unique; however, it is likely to be so.

Reading Binary Data

The `fread` function reads all or part of a binary file (as specified by a file identifier) and stores it in a matrix. In its simplest form, it reads an entire file and interprets each byte of input as the next element of the matrix. For example, the following code reads the data from a file named `nickel.dat` into matrix `A`:

```
fid = fopen('nickel.dat','r');
A = fread(fid);
```

To echo the data to the screen after reading it, use `char` to display the contents of `A` as characters, transposing the data so it is displayed horizontally:

```
disp(char(A'))
```

The `char` function causes MATLAB to interpret the contents of `A` as characters instead of as numbers. Transposing `A` displays it in its more natural horizontal format.

Controlling the Number of Values Read

`fread` accepts an optional second argument that controls the number of values read (if unspecified, the default is the entire file). For example, this statement reads the first 100 data values of the file specified by `fid` into the column vector `A`.

```
A = fread(fid,100);
```

Replacing the number `100` with the matrix dimensions `[10 10]` reads the same 100 elements into a 10-by-10 array.

Controlling the Data Type of Each Value

An optional third argument to `fread` controls the data type of the input. The data type argument controls both the number of bits read for each value and the interpretation of those bits as character, integer, or floating-point values. MATLAB supports a wide range of precisions, which you can specify with MATLAB specific strings or their C or Fortran equivalents.

Some common precisions include

- `'char'` and `'uchar'` for signed and unsigned characters (usually 8 bits)
- `'short'` and `'long'` for short and long integers (usually 16 and 32 bits, respectively)
- `'float'` and `'double'` for single- and double-precision floating-point values (usually 32 and 64 bits, respectively)

Note The meaning of a given precision can vary across different hardware platforms. For example, a `'uchar'` is not always 8 bits. `fread` also provides a number of more specific precisions, such as `'int8'` and `'float32'`. If in doubt, use precisions that are not platform dependent. See `fread` for a complete list of precisions.

For example, if `fid` refers to an open file containing single-precision floating-point values, then the following command reads the next 10 floating-point values into a column vector `A`:

```
A = fread(fid,10,'float');
```

Writing Binary Data

The `fwrite` function writes the elements of a matrix to a file in a specified numeric precision, returning the number of values written. For instance, these lines create a 100-byte binary file containing the 25 elements of the 5-by-5 magic square, each stored as 4-byte integers:

```
fwriteid = fopen('magic5.bin','w');
count = fwrite(fwriteid,magic(5),'int32');
status = fclose(fwriteid);
```

In this case, `fwrite` sets the `count` variable to `25` unless an error occurs, in which case the value is less.

Controlling Position in a File

Once you open a file with `fopen`, MATLAB maintains a file position indicator that specifies a particular location within a file. MATLAB uses the file position indicator to determine where in the file the next read or write operation will begin. The following sections describe how to

- Determine whether the file position indicator is at the end of the file
- Move to a specific location in the file
- Retrieve the current location of the file position indicator
- Reset the file position indicator to the beginning of the file

Setting and Querying the File Position

The `fseek` and `ftell` functions enable you to set and query the position in the file at which the next input or output operation takes place:

- The `fseek` function repositions the file position indicator, letting you skip over data or back up to an earlier part of the file.
- The `ftell` function gives the offset in bytes of the file position indicator for a specified file.

The syntax for `fseek` is

```
status = fseek(fid,offset,origin)
```

`fid` is the file identifier for the file. `offset` is a positive or negative offset value, specified in bytes. `origin` is one of the following strings that specify the location in the file from which to calculate the position.

`'bof'`	Beginning of file
`'cof'`	Current position in file
`'eof'`	End of file

Example of Using fseek And ftell

To see how `fseek` and `ftell` work, consider this short M-file:

```
A = 1:5;
fid = fopen('five.bin','w');
fwrite(fid, A,'short');
status = fclose(fid);
```

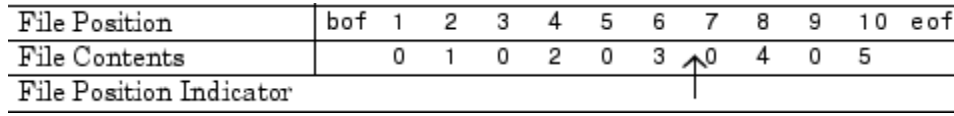
This code writes out the numbers 1 through 5 to a binary file named `five.bin`. The call to `fwrite` specifies that each numerical element be stored as a `short`. Consequently, each number uses two storage bytes.

Now reopen `five.bin` for reading:

```
fid = fopen('five.bin','r');
```

This call to `fseek` moves the file position indicator forward 6 bytes from the beginning of the file:

```
status = fseek(fid,6,'bof');
```

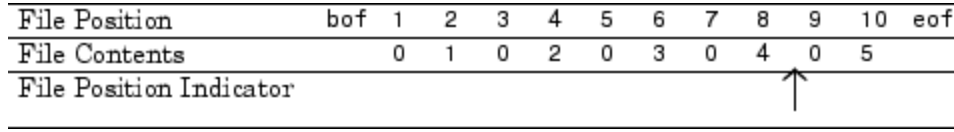


This call to `fread` reads whatever is at file positions 7 and 8 and stores it in variable `four`:

```
four = fread(fid,1,'short');
```

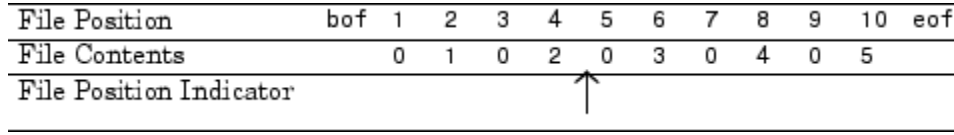
The act of reading advances the file position indicator. To determine the current file position indicator, call `ftell`:

```
position = ftell(fid)

position =

     8
```



This call to `fseek` moves the file position indicator back 4 bytes:

```
status = fseek(fid,-4,'cof');
```



Calling `fread` again reads in the next value (3):

```
three = fread(fid,1,'short');
```

Reading Strings Line by Line from Text Files

MATLAB provides two functions, `fgetl` and `fgets`, that read lines from formatted text files and store them in string vectors. The two functions are almost identical; the only difference is that `fgets` copies the newline character to the string vector but `fgetl` does not.

The following M-file function demonstrates a possible use of `fgetl`. This function uses `fgetl` to read an entire file one line at a time. For each line, the function determines whether an input literal string (`literal`) appears in the line.

If it does, the function prints the entire line preceded by the number of times the literal string appears on the line.

```
function y = litcount(filename, literal)
% Search for number of string matches per line.

fid = fopen(filename, 'rt');
y = 0;
while feof(fid) == 0
   tline = fgetl(fid);
   matches = findstr(tline, literal);
   num = length(matches);
   if num > 0
      y = y + num;
      fprintf(1,'%d:%s\n',num,tline);
   end
end
fclose(fid);
```

For example, consider the following input data file called `badpoem`:

```
Oranges and lemons,
Pineapples and tea.
Orangutans and monkeys,
Dragonflys or fleas.
```

To find out how many times the string `'an'` appears in this file, use `litcount`:

```
litcount('badpoem','an')
2: Oranges and lemons,
```

```
1: Pineapples and tea.
3: Orangutans and monkeys,
```

Reading Formatted ASCII Data

The `fscanf` function is like the `fscanf` function in standard C. Both functions operate in a similar manner, reading data from a file and assigning it to one or more variables. Both functions use the same set of conversion specifiers to control the interpretation of the input data.

The conversion specifiers for `fscanf` begin with a `%` character; common conversion specifiers include.

Conversion Specifier	Description
`%s`	Match a string.
`%d`	Match an integer in base 10 format.
`%g`	Match a double-precision floating-point value.

You can also specify that `fscanf` skip a value by specifying an asterisk in a conversion specifier. For example, `%*f` means skip the floating-point value in the input data; `%*d` means skip the integer value in the input data.

Differences Between the MATLAB fscanf and the C fscanf

Despite all the similarities between the MATLAB and C versions of `fscanf`, there are some significant differences. For example, consider a file named `moon.dat` for which the contents are as follows:

```
3.654234533
2.71343142314
5.34134135678
```

The following code reads all three elements of this file into a matrix named `MyData`:

```
fid = fopen('moon.dat','r');
MyData = fscanf(fid,'%g');
status = fclose(fid);
```

Notice that this code does not use any loops. Instead, the `fscanf` function continues to read in text as long as the input format is compatible with the format specifier.

An optional size argument controls the number of matrix elements read. For example, if `fid` refers to an open file containing strings of integers, then this line reads 100 integer values into the column vector `A`:

```
A = fscanf(fid,'%5d',100);
```

This line reads 100 integer values into the 10-by-10 matrix `A`:

```
A = fscanf(fid,'%5d',[10 10]);
```

A related function, `sscanf`, takes its input from a string instead of a file. For example, this line returns a column vector containing 2 and its square root:

```
root2 = num2str([2, sqrt(2)]);
rootvalues = sscanf(root2,'%f');
```

Writing Formatted Text Files

The `fprintf` function converts data to character strings and outputs them to the screen or a file. A format control string containing conversion specifiers and any optional text specify the output format. The conversion specifiers control the output of array elements; `fprintf` copies text directly.

Common conversion specifiers include

Conversion Specifier	Description
`%e`	Exponential notation
`%f`	Fixed-point notation
`%g`	Automatically select the shorter of `%e` and `%f`

Optional fields in the format specifier control the minimum field width and precision. For example, this code creates a text file containing a short table of the exponential function:

```
x = 0:0.1:1;
y = [x; exp(x)];
```

The code below writes x and y into a newly created file named exptable.txt:

```
fid = fopen('exptable.txt','w');
fprintf(fid,'Exponential Function\n\n');
fprintf(fid,'%6.2f  %12.8f\n',y);
status = fclose(fid);
```

The first call to fprintf outputs a title, followed by two carriage returns. The second call to fprintf outputs the table of numbers. The format control string specifies the format for each line of the table:

- A fixed-point value of six characters with two decimal places
- Two spaces
- A fixed-point value of twelve characters with eight decimal places

fprintf converts the elements of array y in column order. The function uses the format string repeatedly until it converts all the array elements.

Now use fscanf to read the exponential data file:

```
fid = fopen('exptable.txt','r');
title = fgetl(fid);
[table,count] = fscanf(fid,'%f %f',[2 11]);
table = table';
status = fclose(fid);
```

The second line reads the file title. The third line reads the table of values, two floating-point values on each line, until it reaches end of file. count returns the number of values matched.

A function related to fprintf, sprintf, outputs its results to a string instead of a file or the screen. For example,

```
root2 = sprintf('The square root of %f is %10.8e.\n',2,sqrt(2));
```

Closing a File

When you finish reading or writing, use fclose to close the file. For example, this line closes the file associated with file identifier fid:

```
status = fclose(fid);
```

This line closes all open files:

```
status = fclose('all');
```

Both forms return `0` if the file or files were successfully closed or `-1` if the attempt was unsuccessful.

MATLAB automatically closes all open files when you exit from MATLAB. It is still good practice, however, to close a file explicitly with `fclose` when you are finished using it. Not doing so can unnecessarily drain system resources.

Note Closing a file does not clear the file identifier variable `fid`. However, subsequent attempts to access a file through this file identifier variable will not work.

Exchanging Files over the Internet

In this section...

Overview

MATLAB provides functions for exchanging files over the Internet. You can exchange files using common protocols, such as File Transfer Protocol (FTP), Simple Mail Transport Protocol (SMTP), and HyperText Transfer Protocol (HTTP). In addition, you can create zip archives to minimize the transmitted file size, and also save and work with Web pages.

Downloading Web Content and Files

MATLAB provides two functions for downloading Web pages and files using HTTP: `urlread` and `urlwrite`. With the `urlread` function, you can read and save the contents of a Web page to a string variable in the MATLAB workspace. With the `urlwrite` function, you can save a Web page's content to a file.

Because it creates a string variable in the workspace, the `urlread` function is useful for working with the contents of Web pages in MATLAB. The `urlwrite` function is useful for saving Web pages to a local directory.

Note When using `urlread`, remember that only the HTML in that specific Web page is retrieved. The hyperlink targets, images, and so on will not be retrieved.

If you need to pass parameters to a Web page, the `urlread` and `urlwrite` functions let you use HTTP `post` and `get` methods. For more information, see the `urlread` and `urlwrite` reference pages.

Example — Using the urlread Function

The following procedure demonstrates how to retrieve the contents of the Web page containing the Recent File list at the MATLAB Central File Exchange, `http://www.mathworks.com/matlabcentral/fileexchange/index.jsp`. It assigns the results to a string variable, `recentFile`, and it uses the `strfind` function to search the retrieved content for a specific word:

1. Retrieve the Web page content with the `urlread` function:

   ```
   recentFile =
   urlread('http://www.mathworks.com/matlabcentral/fileexchange/
   loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');
   ```

2. After retrieving the content, run the `strfind` function on the `recentFile` variable:

   ```
   hits = strfind(recentFile,'Simulink');
   ```

   If the file contains the word `Simulink`, MATLAB will store the matches in the `hits` variable.

While you can manually pass arguments using the URL, the `urlread` function also lets you pass parameters to a Web page using standard HTTP methods, including `post` and `form`. Using the HTTP `get` method, which passes parameters in the URL, the following code queries Google for the word `Simulink`:

```
s =
urlread('http://www.google.com/search','get',{'q','Simulink'})
```

For more information, see the `urlread` reference page.

Example — Using the urlwrite Function

The following example builds on the procedure in the previous section. This example still uses `urlread` and checks for a specific word, but it also uses `urlwrite` to save the file if it contains any matches:

```
% The urlread function loads the contents of the Web page into
the % MATLAB workspace.

recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.

urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');
end;
```

MATLAB saves the Web page as `contains_simulink.html`.

Creating and Decompressing Zip Archives

Using the `zip` and `unzip` functions, you can compress and decompress files and directories. The `zip` function compresses files or directories into a zip archive. The `unzip` function decompresses zip archives.

Example — Using the zip Function

Again building on the example from previous sections, the following code creates a zip archive of the retrieved Web page:

```
% The urlread function loads the contents of the Web page into
the % MATLAB workspace.
recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
```

```
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');

% The zip function creates a zip archive of the retrieved Web
page.
zip('simulink_matches.zip','contains_simulink.html');
end;
```

Sending E-Mail

To send an e-mail from within MATLAB, use the `sendmail` function. You can also attach files to an e-mail, which lets you mail files directly from MATLAB. To use `sendmail`, you must first set up your e-mail address and your SMTP server information with the `setpref` function.

The `setpref` function defines two mail-related preferences:

- E-mail address: This preference sets your e-mail address that will appear on the message. Here is an example of the syntax:

  ```
  setpref('Internet','E_mail','youraddress@yourserver.com');
  ```

- SMTP server: This preference sets your outgoing SMTP server address, which can be almost any e-mail server that supports the Post Office Protocol (POP) or the Internet Message Access Protocol (IMAP). Here is an example of the syntax:

  ```
  setpref('Internet', 'SMTP_Server', 'mail.server.network');
  ```

You should be able to find your outgoing SMTP server address in your e-mail account settings in your e-mail client application. You can also contact your system administrator for the information.

Note The `sendmail` function does not support e-mail servers that require authentication.

Once you have properly configured MATLAB, you can use the `sendmail` function. The `sendmail` function requires at least two arguments: the recipient's e-mail address and the e-mail subject:

```
sendmail('recepient@someserver.com', 'Hello From MATLAB!');
```

You can supply multiple e-mail addresses using a cell array of strings, such as:

```
sendmail({'recepient@someserver.com', ...
'recepient2@someserver.com'}, 'Hello From MATLAB!');
```

You can also specify a message body with the `sendmail` function, such as:

```
sendmail('recepient@someserver.com', 'Hello From MATLAB!', ...
'Thanks for using sendmail.');
```

In addition, you can also attach files to an e-mail using the `sendmail` function, such as:

```
sendmail('recepient@somesever.com', 'Hello from MATLAB!', ...
'Thanks for using sendmail.', 'C:\yourFileSystem\message.txt');
```

You cannot attach a file without including a message. However, the message can be empty. You can also attach multiple files to an e-mail with the `sendmail` function, such as:

```
sendmail('recepient@somesever.com', 'Hello from MATLAB!', ...
'Thanks for using sendmail.', ...
{'C:\yourFileSystem\message.txt',...
'C:\yourFileSystem\message2.txt'});
```

Example — Using the sendmail Function

The following example sends e-mail with the retrieved Web page archive attached if it contains any matches for the specified word:

```
% The urlread function loads the contents of the Web page into
the  % MATLAB workspace.
recentFile =
urlread('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0');

% The strfind function searches for the word "Simulink".
hits = strfind(recentFile,'Simulink');

% The if statement checks for any hits.
if ~isempty(hits)

% If there are hits, the Web page will be saved locally
% using the urlwrite function.
urlwrite('http://www.mathworks.com/matlabcentral/fileexchange/
loadFileList.do?objectType=fileexchange&orderBy=date&srt3=0',
'contains_simulink.html');

% The zip function creates a zip archive of the retrieved web
page.
zip('simulink_matches.zip','contains_simulink.html');

% The setpref function supplies your e-mail address and SMTP
% server address to MATLAB.
setpref('Internet','SMTP_Server','mail.server.network');
setpref('Internet', 'E_mail', 'youraddress@yourserver.com');

% The sendmail function sends an e-mail with the zip archive of
the
% retrieved Web page attached.
sendmail('youraddress@yourserver.com', 'New Simulink Files
Found', 'New Simulink files uploaded to MATLAB Central. See
attached zip archive.', 'simulink_matches.zip');
end;
```

Performing FTP File Operations

From within MATLAB, you can connect to an FTP server to perform remote file operations. The following procedure uses a public MathWorks FTP server

(`ftp.mathworks.com`). To perform any file operation on an FTP server, follow these steps:

1. Connect to the server using the `ftp` function.

   For example, you can create an FTP object for the public MathWorks FTP server with `tmw=ftp('ftp.mathworks.com')`.

2. Perform the file operations using appropriate MATLAB FTP functions as methods acting on the server object.

   For example, you can display the file directories on the FTP server with `dir(tmw)`.

3. When you finish working on the server, close the connection object using the close function.

   For example, you can disconnect from the FTP server with `close(tmw)`.

Example — Retrieving a File from an FTP Server

In this example, you retrieve the file pub/pentium/Moler_1.txt, which is on the MathWorks FTP server. You can run this example; the FTP server and content are valid.

1. Connect to the MathWorks FTP server using `ftp`. This creates the server object `tmw`:

   ```
   tmw=ftp('ftp.mathworks.com');
   ```

2. List the contents of the server using the `dir` FTP function, which operates on the server object `tmw`:

   ```
   dir(tmw)
   ```

3. Change to the `pub` directory by using the FTP cd function. As with all FTP functions, you need to specify the server object you created using `ftp` as part of the syntax. In this case, this is `tmw`:

   ```
   cd(tmw,'pub');
   ```

The server object `tmw` represents the current directory on the FTP server, which now is pub.

4 Now when you run

```
dir(tmw)
```

you see the contents of pub, rather than the top level contents as displayed previously when you ran `dir(tmw)`.

5 Use `mget` to retrieve any of the files from the current directory on the FTP server to the MATLAB current directory:

```
mget(tmw,'filename');
```

6 Close the FTP connection using `close`.

```
close(tmw);
```

Summary of FTP Functions

The following table lists the available FTP functions. For more information, refer to the applicable reference pages.

Function	Description
`ascii`	Set FTP transfer type to ASCII (convert new lines).
`binary`	Set FTP transfer type to binary (transfer verbatim, default).
`cd (ftp)`	Change current directory on FTP server.
`delete (ftp)`	Delete file on FTP server.
`dir (ftp)`	List contents of directory on FTP server.
`close (ftp)`	Close connection with FTP server.
`ftp`	Connect to FTP server, creating an FTP object.
`mget`	Download file from FTP site.
`mkdir (ftp)`	Create new directory on FTP server.
`mput (ftp)`	Upload file or directory to FTP server.

Function	Description
`rename`	Rename file on FTP server.
`rmdir (ftp)`	Remove directory on FTP server.

7

Working with Scientific Data Formats

This section describes how to import and export data in several standard scientific data formats. Topics covered include

Common Data Format (CDF) Files (p. 7-2)	Reading and writing data and metadata using the Common Data Format (CDF) file format.
Flexible Image Transport System (FITS) Files (p. 7-8)	Reading data and metadata using the Flexible Image Transport System (FITS) file format.
Hierarchical Data Format (HDF5) Files (p. 7-11)	Reading and writing data and metadata using the Hierarchical Data Format (HDF5) file format.
Hierarchical Data Format (HDF4) Files (p. 7-36)	Reading and writing data and metadata using the Hierarchical Data Format (HDF4) file format.

Common Data Format (CDF) Files

In this section...

Getting Information About CDF Files

To get information about the contents of a Common Data Format (CDF) file, use the `cdfinfo` function. CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). The `cdfinfo` function returns a structure containing general information about the file and detailed information about the variables and attributes in the file. For more information about this format, see the CDF Web site.

The following example returns information about the sample CDF file included with MATLAB. To determine the variables contained in the file, view the `Variables` field. This field contains a cell array that lists all the variables in the file with information that describes the variable, such as name, size, and data type. For an example, see "Importing Data from a CDF File" on page 7-3.

Note Because `cdfinfo` creates temporary files, make sure that your current working directory is writable before attempting to use the function.

```
info = cdfinfo('example.cdf')

info =

             Filename: 'example.cdf'
          FileModDate: '09-Mar-2001 16:45:22'
             FileSize: 1240
               Format: 'CDF'
```

```
      FormatVersion: '2.7.0'
       FileSettings: [1x1 struct]
           Subfiles: {}
          Variables: {5x6 cell}
    GlobalAttributes: [1x1 struct]
  VariableAttributes: [1x1 struct]
```

Importing Data from a CDF File

To import data into the MATLAB workspace from a Common Data Format (CDF) file, use the `cdfread` function. CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). Using this function, you can import all the data in the file, specific variables, specific records, or subsets of the data in a specific variable. The following examples illustrate some of these capabilities.

1. To get information about the contents of a CDF file, such as the names of variables in the CDF file, use the `cdfinfo` function. In this example, the `Variables` field indicates that the file contains five variables. The first variable, `Time`, is made up of 24 records containing CDF epoch data. The next two variables, `Longitude` and `Latitude`, have only one associated record containing `int8` data. For details about how to interpret the data returned in the `Variables` field, see `cdfinfo`.

```
info = cdfinfo('example.cdf');

vars = info.Variables

vars =

  Columns 1 through 5

   'Time'               [1x2 double]   [24]   'epoch'    'T/'
   'Longitude'          [1x2 double]   [ 1]   'int8'     'F/FT'
   'Latitude'           [1x2 double]   [ 1]   'int8'     'F/TF'
   'Data'               [1x3 double]   [ 1]   'double'   'T/TTT'
   'multidimensional    [1x4 double]   [ 1]   'uint8'    'T/TTTT'

  Column 6
```

```
'Full'
'Full'
'Full'
'Full'
'Full'
```

2 To read all of the data in the CDF file, use the `cdfread` function. The function returns the data in a 24-by-5 cell array. The five columns of data correspond to the five variables; the 24 rows correspond to the 24 records associated with the Time variable and padding elements for the rows associated with the other variables. The padding value used is specified in the CDF file.

```
data = cdfread('example.cdf');

whos data
  Name       Size             Bytes  Class    Attributes

  data      24x5              14784  cell
```

3 To read the data associated with a particular variable, use the `'Variable'` parameter, specifying a cell array of variable names as the value of this parameter. Variable names are case sensitive. For example, the following code reads the Longitude and Latitude variables from the file. The return value `data` is a 24-by-2 cell array, where each cell contains `int8` data.

```
var_time = cdfread('example.cdf','Variable',{'Longitude','Latitude'});

whos var_time
  Name           Size            Bytes  Class    Attributes

  var_time      24x1              4608  cell
```

Speeding Up Read Operations

The `cdfread` function offers two ways to speed up read operations when working with large data sets:

- Reducing the number of elements in the returned cell array

- Returning CDF epoch values as MATLAB serial date numbers rather than as MATLAB `cdfepoch` objects

To reduce the number of elements in the returned cell array, specify the `'CombineRecords'` parameter. By default, `cdfread` creates a cell array with a separate element for every variable and every record in each variable, padding the records dimension to create a rectangular cell array. For example, reading all the data from the example file produces an output cell array, 24-by-5, where the columns represent variables and the rows represent the records for each variable. When you set the `'CombineRecords'` parameter to `true`, `cdfread` creates a separate element for each variable but saves time by putting all the records associated with a variable in a single cell array element. Thus, reading the data from the example file with `'CombineRecords'` set to `true` produces a 1-by-5 cell array, as shown below.

```
data_combined = cdfread('example.cdf','CombineRecords',true);

whos
  Name                Size            Bytes  Class    Attributes

  data               24x5             14784  cell
  data_combined       1x5              2364  cell
```

When combining records, note that the dimensions of the data in the cell change. For example, if a variable has 20 records, each of which is a scalar value, the data in the cell array for the combined element contains a 20-by-1 vector of values. If each record is a 3-by-4 array, the cell array element contains a 20-by-3-by-4 array. For combined data, `cdfread` adds a dimension to the data, the first dimension, that is the index into the records.

Another way to speed up read operations is to read CDF epoch values as MATLAB serial date numbers. By default, `cdfread` creates a MATLAB `cdfepoch` object for each CDF epoch value in the file. If you specify the `'ConvertEpochToDatenum'` parameter, setting it to `true`, `cdfread` returns CDF epoch values as MATLAB serial date numbers. For more information about working with MATLAB `cdfepoch` objects, see “Representing CDF Time Values” on page 7-6.

```
data_datenums = cdfread('example.cdf','ConvertEpochToDatenum',true);
```

```
whos
  Name                Size            Bytes  Class    Attributes

  data               24x5             14784  cell
  data_combined       1x5              2364  cell
  var_time           24x1              4608  cell
```

Representing CDF Time Values

CDF represents time differently than MATLAB. CDF represents date and time as the number of milliseconds since 1-Jan-0000. This is called an *epoch* in CDF terminology. MATLAB represents date and time as a serial date number, which is the number of days since 0-Jan-0000. To represent CDF dates, MATLAB uses an object called a CDF epoch object. To access the time information in a CDF object, use the object's `todatenum` method.

For example, this code extracts the date information from a CDF epoch object:

1. Extract the date information from the CDF epoch object returned in the cell array `data` (see "Importing Data from a CDF File" on page 7-3). Use the `todatenum` method of the CDF epoch object to get the date information, which is returned as a MATLAB serial date number.

   ```
   m_date = todatenum(data{1});
   ```

2. View the MATLAB serial date number as a string.

   ```
   datestr(m_date)
   ans =

   01-Jan-2001
   ```

Exporting Data to a CDF File

To export data from the MATLAB workspace to a Common Data Format (CDF) file, use the `cdfwrite` function. CDF was created by the National Space Science Data Center (NSSDC) to provide a self-describing data storage and manipulation format that matches the structure of scientific data and applications (i.e., statistical and numerical methods, visualization, and management). Using this function, you can write variables and attributes to the file, specifying their names and associated values. See the `cdfwrite` reference page for more information.

This example shows how to write date information to a CDF file. Note how the example uses the CDF epoch object constructor, `cdfepoch`, to convert a MATLAB serial date number into a CDF epoch.

```
cdfwrite('myfile',{'Time_val',cdfepoch(now)});
```

You can convert a `cdfepoch` object back into a MATLAB serial date number with the `todatenum` function.

Flexible Image Transport System (FITS) Files

In this section...

Getting Information About FITS Files

To get information about the contents of a Flexible Image Transport System (FITS) file, use the `fitsinfo` function. The FITS file format is the standard data format used in astronomy, endorsed by both NASA and the International Astronomical Union (IAU). For more information about the FITS standard, go to the official FITS Web site, `http://fits.gsfc.nasa.gov/`.

A data file in FITS format can contain multiple components, each marked by an ASCII text header followed by binary data. The first component in a FITS file is known as the *primary*, which can be followed by any number of other components, called *extensions,* in FITS terminology. The `fitsinfo` function returns a structure containing the information about the file and detailed information about the data in the file. This example returns information about a sample FITS file included with MATLAB. The structure returned contains fields for the primary component, `PrimaryData`, and all the extensions in the file, such as the `BinaryTable`, `Image`, and `AsciiTable` extensions.

```
info = fitsinfo('tst0012.fits')

info =

       Filename: 'tst0012.fits'
    FileModDate: '12-Mar-2001 18:37:46'
       FileSize: 109440
       Contents: {1x5 cell}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]
        Unknown: [1x1 struct]
          Image: [1x1 struct]
     AsciiTable: [1x1 struct]
```

Importing Data from a FITS File

To import data into the MATLAB workspace from a Flexible Image Transport System (FITS) file, use the `fitsread` function. The FITS file format is designed to store scientific data sets consisting of multidimensional arrays (1-D spectra, 2-D images, or 3-D data cubes) and two-dimensional tables containing rows and columns of data. Using this function, you can import the data in the `PrimaryData` section of the file or you can import the data in any of the extensions in the file, such as the `Image` extension. This example illustrates how to use the `fitsread` function to read data from a FITS file:

1 Determine which extensions the FITS file contains, using the `fitsinfo` function.

```
info = fitsinfo('tst0012.fits')

info =

       Filename: 'tst0012.fits'
    FileModDate: '12-Mar-2001 18:37:46'
       FileSize: 109440
       Contents: {1x5 cell}
    PrimaryData: [1x1 struct]
    BinaryTable: [1x1 struct]
        Unknown: [1x1 struct]
          Image: [1x1 struct]
     AsciiTable: [1x1 struct]
```

The `info` structure shows that the file contains several extensions including the `BinaryTable`, `AsciiTable`, and `Image` extensions.

2 Read data from the file.

To read the `PrimaryData` in the file, specify the filename as the only argument:

```
pdata = fitsread('tst0012.fits');
```

To read any of the extensions in the file, you must specify the name of the extension as an optional parameter. This example reads the `BinaryTable` extension from the FITS file:

```
bindata = fitsread('tst0012.fits','bintable');
```

Note To read the `BinaryTable` extension using `fitsread`, you must specify the parameter `'bintable'`. Similarly, to read the `AsciiTable` extension, you must specify the parameter `'table'`. See the `fitsread` reference page for more information.

Hierarchical Data Format (HDF5) Files

In this section...

Note For information about working with HDF4 data, which is a completely separate, incompatible format, see "Hierarchical Data Format (HDF4) Files" on page 7-36.

Using the MATLAB High-Level HDF5 Functions

Hierarchical Data Format, Version 5, (HDF5) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). HDF5 is used by a wide range of engineering and scientific fields that want a standard way to store data so that it can be shared. For more information about the HDF5 file format, read the HDF5 documentation available at the HDF Web site (`http://www.hdfgroup.org`).

The MATLAB high-level HDF5 functions provide an easy way to import data or metadata from an HDF5 file, or write data to an HDF5 file. The following sections provide more detail about using these functions.

Determining the Contents of an HDF5 File

HDF5 files can contain data and metadata, called *attributes*. HDF5 files organize the data and metadata in a hierarchical structure similar to the hierarchical structure of a UNIX file system.

In an HDF5 file, the directories in the hierarchy are called *groups*. A group can contain other groups, data sets, attributes, links, and data types. A data set is a collection of data, such as a multidimensional numeric array or string. An attribute is any data that is associated with another entity, such as a data set. A link is similar to a UNIX file system symbolic link. Links are a way to reference data without having to make a copy of the data.

Data types are a description of the data in the data set or attribute. Data types tell how to interpret the data in the data set. For example, a file might contain a data type called "Reading" that is comprised of three elements: a longitude value, a latitude value, and a temperature value.

To explore the hierarchical organization of an HDF5 file, use the `hdf5info` function. For example, to find out what the sample HDF5 file, `example.h5`, contains, use this syntax:

```
fileinfo = hdf5info('example.h5');
```

`hdf5info` returns a structure that contains various information about the HDF5 file, including the name of the file and the version of the HDF5 library that MATLAB is using:

```
fileinfo =

          Filename: 'example.h5'
        LibVersion: '1.6.5'
            Offset: 0
          FileSize: 8172
    GroupHierarchy: [1x1 struct]
```

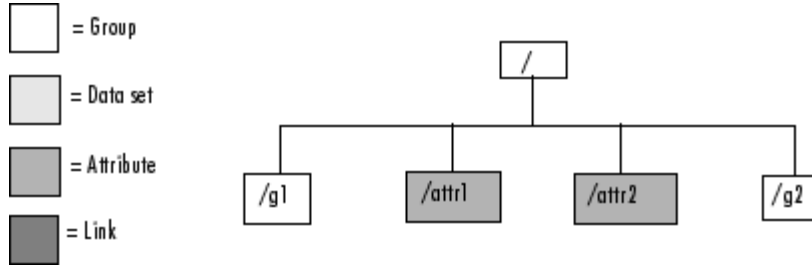
In the information returned by `hdf5info`, look at the `GroupHierarchy` field. This field is a structure that describes the top-level group in the file, called the *root* group. Using the UNIX convention, HDF5 names this top-level group `/` (forward slash), as shown in the `Name` field of the `GroupHierarchy` structure.

```
toplevel = fileinfo.GroupHierarchy

toplevel =

      Filename: 'C:\matlab\toolbox\matlab\demos\example.h5'
          Name: '/'
```

```
          Groups: [1x2 struct]
        Datasets: []
      Datatypes: []
           Links: []
      Attributes: [1x2 struct]
```

By looking at the `Groups` and `Attributes` fields, you can see that the file contains two groups and two attributes. The `Datasets`, `Datatypes`, and `Links` fields are all empty, indicating that the root group does not contain any data sets, data types, or links.

The following figure illustrates the organization of the root group in the sample HDF5 file `example.h5`.

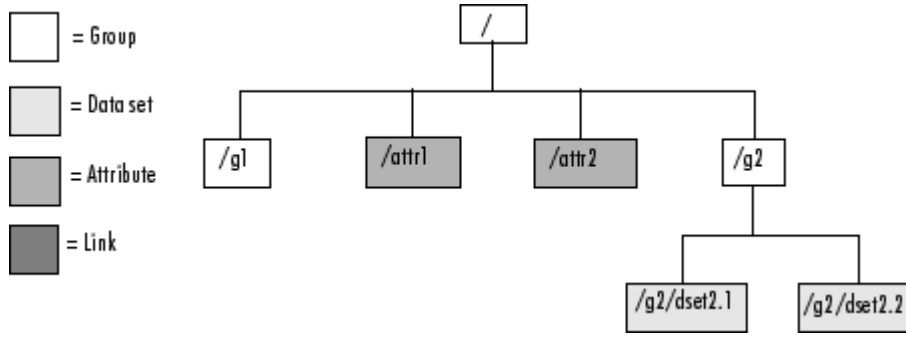


Organization of the Root Group of the Sample HDF5 File

To explore the contents of the sample HDF5 file further, examine one of the two structures in the `Groups` field of the `GroupHierarchy` structure. Each structure in this field represents a group contained in the root group. The following example shows the contents of the second structure in this field.

```
level2 = toplevel.Groups(2)

level2 =

      Filename: 'C:\matlab\toolbox\matlab\demos\example.h5'
          Name: '/g2'
        Groups: []
      Datasets: [1x2 struct]
     Datatypes: []
         Links: []
    Attributes: []
```

In the sample file, the group named `/g2` contains two data sets. The following figure illustrates this part of the sample HDF5 file organization.

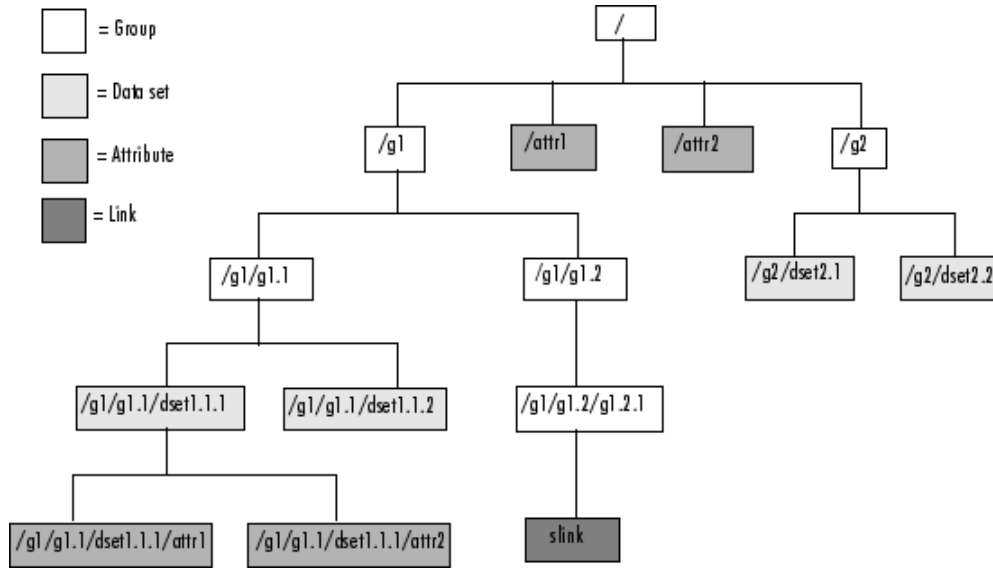


Organization of the Data Set /g2 in the Sample HDF5 File

To get information about a data set, look at either of the structures returned in the `Datasets` field. These structures provide information about the data set, such as its name, dimensions, and data type.

```
dataset1 = level2.Datasets(1)

dataset1 =
      Filename: 'L:\matlab\toolbox\matlab\demos\example.h5'
          Name: '/g2/dset2.1'
          Rank: 1
      Datatype: [1x1 struct]
          Dims: 10
       MaxDims: 10
        Layout: 'contiguous'
    Attributes: []
         Links: []
     Chunksize: []
     Fillvalue: []
```

By examining the structures at each level of the hierarchy, you can traverse the entire file. The following figure describes the complete hierarchical organization of the sample file `example.h5`.



Hierarchical Structure of example.h5 HDF5 File

Importing Data from an HDF5 File

To read data or metadata from an HDF5 file, use the `hdf5read` function. As arguments, you must specify the name of the HDF5 file and the name of the data set or attribute. Alternatively, you can specify just the field in the structure returned by `hdf5info` that contains the name of the data set or attribute; `hdf5read` can determine the file name from the `Filename` field in the structure. For more information about finding the name of a data set or attribute in an HDF5 file, see "Determining the Contents of an HDF5 File" on page 7-11.

To illustrate, this example reads the data set, `/g2/dset2.1` from the HDF5 sample file `example.h5`.

```
data = hdf5read('example.h5','/g2/dset2.1');
```

The return value contains the values in the data set, in this case a 1-by-10 vector of single-precision values:

```
data =

    1.0000
    1.1000
    1.2000
    1.3000
    1.4000
    1.5000
    1.6000
    1.7000
    1.8000
    1.9000
```

The `hdf5read` function maps HDF5 data types to appropriate MATLAB data types, whenever possible. If the HDF5 file contains data types that cannot be represented in MATLAB, `hdf5write` uses one of the predefined MATLAB HDF5 data type objects to represent the data.

For example, if an HDF5 data set contains four array elements, `hdf5read` can return the data as a 1-by-4 array of `hdf5.h5array` objects:

```
whos

Name    Size        Bytes     Class

data     1x4                hdf5.h5array

Grand total is 4 elements using 0 bytes
```

For more information about the MATLAB HDF5 data type objects, see "Mapping HDF5 Data Types to MATLAB Data Types" on page 7-18.

Exporting Data to HDF5 Files

To write data or metadata from the MATLAB workspace to an HDF5 file, use the `hdf5write` function. As arguments, specify:

- Name of an existing HDF5 file, or the name you want to assign to a new file.

- Name of an existing data set or attribute, or the name you want to assign to a new data set or attribute. To learn how to determine the name of data sets in an existing HDF5 file, see “Determining the Contents of an HDF5 File” on page 7-11.
- Data or metadata you want to write to the file. `hdf5write` converts MATLAB data types to the appropriate HDF5 data type automatically. For nonatomic data types, you can also create HDF5 objects to represent the data.

This example creates a 5-by-5 array of `uint8` values and then writes the array to an HDF5 file. By default, `hdf5write` overwrites the file, if it already exists. The example specifies an `hdf5write` mode option to append data to existing file.

1 Create a MATLAB variable in the workspace. This example creates a 5-by-5 array of `uint8` values.

```
testdata = uint8(magic(5))
```

2 Write the data to an HDF5 file. As arguments to `hdf5read`, the example specifies the name you want to assign to the HDF5 file, the name you want to assign to the data set, and the MATLAB variable.

```
hdf5write('myfile.h5', '/dataset1', testdata)
```

To add data to an existing file, you must use the `'writemode'` option, specifying the`'append'` value. The file must already exist and it cannot already contain a data set with the same name

```
hdf5write('myfile.h5', '/dataset12', testdata,'writemode','append')
```

If you are writing simple data sets, such as scalars, strings, or a simple compound data types, you can just pass the data directly to `hdf5write`; this function automatically maps the MATLAB data types to appropriate HDF5 data types. However, if your data is a complex data set, you might need to use one of the predefined MATLAB HDF5 objects to pass the data to the `hdf5write` function. The HDF5 objects are designed for situations where the mapping between MATLAB and HDF5 types is ambiguous.

For example, when passed a cell array of strings, the `hdf5write` function writes a data set made up of strings, not a data set of arrays containing

strings. If that is not the mapping you intend, use HDF5 objects to specify the correct mapping. In addition, note that HDF5 makes a distinction between the size of a data set and the size of a data type. In MATLAB, data types are always scalar. In HDF5, data types can have a size; that is, types can be either scalar (like MATLAB) or m-by-n. In HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8` values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. For more information about the MATLAB HDF5 data type objects, see "Mapping HDF5 Data Types to MATLAB Data Types" on page 7-18.

Mapping HDF5 Data Types to MATLAB Data Types

When the `hdf5read` function reads data from an HDF5 file into the MATLAB workspace, it maps HDF5 data types to MATLAB data types, depending on whether the data in the data set is in an *atomic* data type or a nonatomic *composite* data type.

Mapping Atomic Data Types. Atomic data types describe commonly used binary formats for numbers (integers and floating point) and characters (ASCII). Because different computing architectures and programming languages support different number and character representations, the HDF5 library provides the platform-independent data types, which it then maps to an appropriate data type for each platform. For example, a computer may support 8-, 16-, 32-, and 64-bit signed integers, stored in memory in little endian byte order.

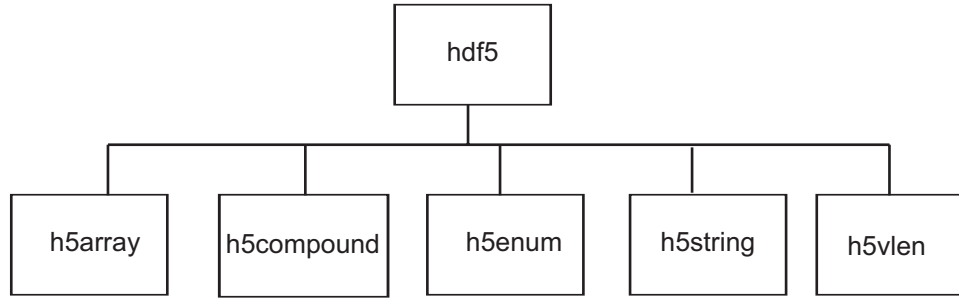
If the data in the data set is stored in one of the HDF5 atomic data types, `hdf5read` uses the equivalent MATLAB data type to represent the data. Each data set contains a `Datatype` field that names the data type. For example, the data set `/g2/dset2.2` in the sample HDF5 file includes atomic data and data type information.

```
dtype = dataset1.Datatype
dtype =

        Name: []
       Class: 'H5T_IEEE_F32BE'
    Elements: []
```

The `H5T_IEEE_F32BE` class name indicates the data is a 4-byte, big endian, IEEE floating-point data type. (See the HDF5 specification for more information about atomic data types.)

Mapping Composite Data Types. A composite data type is an aggregation of one or more atomic data types. Composite data types include structures, multidimensional arrays, and variable-length data types (one-dimensional arrays).

If the data in the data set is stored in one of the HDF5 nonatomic data types and the data cannot be represented in the workspace using a native MATLAB data type,`hdf5read` uses one of a set of classes MATLAB defines to represent HDF5 data types. The following figure illustrates the `hdf5` class and its subclasses. For more information about a specific class, see the sections that follow. To learn more about the HDF5 data types in general, see the HDF Web page at `http://www.hdfgroup.org`.



For example, if an HDF5 file contains a data set made up of an enumerated data type which cannot be represented in MATLAB, `hdf5read` uses the HDF5 `h5enum` class to represent the data. An `h5enum` object has data members that store the enumerations (text strings), their corresponding values, and the enumerated data.

You might also need to use these HDF5 data type classes when using the `hdf5write` function to write data from the MATLAB workspace to an HDF5 file. By default, `hdf5write` can convert most MATLAB data to appropriate HDF5 data types. However, if this default data type mapping is not suitable, you can create HDF5 data types directly.

To access the data in the data set in the MATLAB workspace, you must access the `Data` field in the object.

This example converts a simple MATLAB vector into an `h5array` object and then displays the fields in the object:

```
vec = [ 1 2 3];

hhh = hdf5.h5array(vec);

hhh:

    Name: ''
    Data: [1 2 3]

hhh.Data

ans =

     1     2     3
```

MATLAB HDF5 h5array Data Class. The `h5array` data class associates a name with an array. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object

Methods	Description
`arr = hdf5.h5array`	Creates an `h5array` object.
`arr = hdf5.h5array(data)`	Creates an `h5array` object, where `data` specifies the value of the `Data` member. `data` can be numeric, a cell array, or an HDF5 data type.

Methods	Description
`setData(arr, data)`	Sets the value of the `Data` member, where `arr` is an `h5array` object and `data` can be numeric, a cell array, or an HDF5 data type.
`setName(arr, name)`	Sets the value of the `Name` member, where `arr` is an `h5array` object and `name` is a string or cell array.

MATLAB HDF5 h5compound Data Class. The `h5compound` data class associates a name with a structure. You can define the field names in the structure and their values. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`MemberNames`	Text strings specifying name of the object

Methods	Description
`C = hdf5.h5compound`	Creates an `h5compound` object.
`C = hdf5.h5compound(n1,n2,...)`	Creates an `h5compound` object, where `n1`, `n2` and so on are text strings that specify field names. The constructor creates a corresponding data field for every member name.
`addMember(C, mName)`	Creates a new field in the object `C`. `mName` specifies the name of the field.
`setMember(C, mName, mData)`	Sets the value of the `Data` element associated with the field specified by `mName`, where `C` is an `h5compound` object and `mData` can be numeric or an HDF5 data type.

Methods	Description
`setMemberNames(C, n1, n2,...)`	Specifies the names of fields in the structure, where `C` is an `h5compound` object and `n1`,`n2`, and so on are text strings that specify field names. The method creates a corresponding data field for every name specified.
`setName(C, name)`	Sets the value of the `Name` member, where `C` is an `h5compound` object and `name` is a string or cell array.

MATLAB HDF5 h5enum Data Class. The `h5enum` data class defines an enumerated type. You can specify the enumerations (text strings) and the values they represent. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`EnumNames`	Text string specifying the enumerations, that is, the text strings that represent values
`EnumValues`	Values associated with enumerations

Methods	Description
`E = hdf5.h5enum`	Creates an `h5enum` object.
`E = hdf5.h5enum(eNames, eVals)`	Creates an `h5enum` object, where `eNames` is a cell array of strings, and `eVals` is vector of integers. `eNames` and `eVals` must have the same number of elements.
`defineEnum(E, eNames, eVals)`	Defines the set of enumerations with the integer values they represent where `eNames` is a cell array of strings, and `eVals` is vector of integers. `eNames` and `eVals` must have the same number of elements.

Methods	Description
`enumdata = getString(E)`	Returns a cell array containing the names of the enumerations, where `E` is an `h5enum` object.
`setData(E, eData)`	Sets the value of the object's `Data` member, where `E` is an `h5enum` object and `eData` is a vector of integers.
`setEnumNames(E, eNames)`	Specifies the enumerations, where `E` is an `h5enum` object and `eNames` is a cell array of strings.
`setEnumValues(E, eVals)`	Specifies the value associated with each enumeration, where `E` is an `h5enum` object and `eVals` is a vector of integers.
`setName(E, name)`	Sets the value of the object's `Name` member, where `E` is an `h5enum` object and `name` is a string or cell array.

This example uses an HDF5 enumeration object.

1 Create an HDF5 enumerated object.

```
enum_obj = hdf5.h5enum;
```

2 Define the enumerated values and their corresponding names.

```
enum_obj.defineEnum({'RED' 'GREEN' 'BLUE'}, uint8([1 2 3]));
```

`enum_obj` now contains the definition of the enumeration that associates the names `RED`, `GREEN`, and `BLUE` with the numbers 1, 2, and 3.

3 Add enumerated data to the object.

```
enum_obj.setData(uint8([2 1 3 3 2 3 2 1]));
```

In the HDF5 file, these numeric values map to the enumerated values `GREEN`, `RED`, `BLUE`, `BLUE`, `GREEN`, etc.

4 Write the enumerated data to a data set named `objects` in an HDF5 file.

```
hdf5write('myfile3.h5', '/g1/objects', enum_obj);
```

5 Read the enumerated data set from the file.

```
ddd = hdf5read('myfile3.h5','/g1/objects')

hdf5.h5enum:

          Name: ''
          Data: [2 1 3 3 2 3 2 1]
     EnumNames: {'RED'  'GREEN'  'BLUE'}
    EnumValues: [1 2 3]
```

MATLAB HDF5 h5string Data Class. The `h5string` data class associates a name with a text string and provides optional padding behavior. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object
`Length`	Scalar defining length of string

`Padding`	Type of padding to use:
	`'spacepad'`
	`'nullterm'`
	`'nullpad'`

Methods	Description
`str = hdf5.h5string`	Creates an `h5string` object.
`str = hdf5.h5string(data)`	Creates an `h5string` object, where `data` is a text string.
`str = hdf5.h5string(data, padtype)`	Creates an `h5string`object, where `data` is a text string and `padtype` specifies the type of padding to use.
`setData(str, data)`	Sets the value of the object's `Data` member, where `str` is an `h5string` object and`data` is a text string.

Methods	Description
`setLength(str, lenVal)`	Sets the value of the object's `Length` member, where `str` is an `h5string` object and `lenVal` is a scalar.
`setName(str, name)`	Sets the value of the object's `Name` member, where `str` is an `h5string` object and `name` is a string or cell array.
`setPadding(str, padType)`	Specifies the value of the object's `Padding` member, where `str` is an `h5string` object and `padType` is a text string specifying one of the supported pad types.

The following example creates an HDF5 string object.

```
hdf5.h5vlen({0 [0 1] [0 2] [0:10]})

hdf5.h5vlen:

    Name: ''
    Data: [0 0 1 0 2 0 1 2 3 4 5 6 7 8 9 10]
```

The following example creates an HDF5 `h5vlen` object.

```
hdf5.h5vlen({0 [0 1] [0 2] [0:10]})

hdf5.h5vlen:

    Name: ''
    Data: [0 0 1 0 2 0 1 2 3 4 5 6 7 8 9 10]
```

MATLAB HDF5 h5vlen Data Class. The `h5vlen` data class associates a name with an array. The following tables list the class data members and methods.

Data Members	Description
`Data`	Multidimensional array
`Name`	Text string specifying name of the object

Methods	Description
`V = hdf5.h5vlen`	Creates an `h5vlen` object.
`V = hdf5.h5vlen(data)`	Creates an `h5vlen` object, where `data` specifies the value of the `Data` member. `data` can be numeric, a cell array, or an HDF5 data type.
`setData(V, data)`	Sets the value of the object's `Data` member, where `V` is an `h5vlen` object and `data` can be a scalar, vector, text string, a cell array, or an HDF5 data type.
`setName(V, name)`	Sets the value of the object's`Name` member, where `V` is an `h5vlen` object and `name` is a string or cell array.

Using the MATLAB Low-Level HDF5 Functions

MATLAB provides direct access to the over 200 functions in the HDF5 library by creating MATLAB functions that correspond to the functions in the HDF5 library. In this way, you can access the features of the HDF5 library from MATLAB, such as reading and writing complex data types and using the HDF5 subsetting capabilities.

The HDF5 library organizes the library functions into groups, called *interfaces*. For example, all the routines related to working with files, such as opening and closing, are in the H5F interface, where *F* stands for file. MATLAB organizes the low-level HDF5 functions into classes that correspond to each HDF5 interface. For example, the MATLAB functions that correspond to the HDF5 file interface (H5F) are in the @H5F class directory. For a complete list of the HDF5 interfaces and the corresponding MATLAB class directories, see `hdf5`.

The following sections provide more details about how to use the MATLAB HDF5 low-level functions. Topics covered include:

Note This section does not attempt to describe all features of the HDF5 library or explain basic HDF5 programming concepts. To use the MATLAB HDF5 low-level functions effectively, you must refer to the official HDF5 documentation available at the HDF Web site (`http://www.hdfgroup.org`).

Mapping HDF5 Function Syntax to MATLAB Function Syntax

In most cases, the syntax of the MATLAB low-level HDF5 functions is identical to the syntax of the corresponding HDF5 library functions. For example, the following is the function signature of the `H5Fopen` function in the HDF5 library. In the HDF5 function signatures, `hid_t` and `herr_t` are HDF5 types that return numeric values that represent object identifiers or error status values.

```
hid_t H5Fopen(const char *name, unsigned flags, hid_t access_id ) /* C syntax */
```

In MATLAB, each function in an HDF5 interface is a method of a MATLAB class. To view the function signature for a function, specify the class directory name and then the function name, as in the following.

```
help @H5F/open
```

The following shows the signature of the corresponding MATLAB function. First note that, because it's a method of a class, you must use the dot notation to call the MATLAB function: `H5F.open`. This MATLAB function accepts the same three arguments as the HDF5 function: a text string for the name, an HDF5-defined constant for the flags argument, and an HDF5 property list ID. You use property lists to specify characteristics of many different HDF5 objects. In this case, it's a file access property list. Refer to the HDF5

documentation to see which constants can be used with a particular function and note that, in MATLAB, constants are passed as text strings.

```
file_id = H5F.open(name, flags, plist_id)
```

There are, however, some functions where the MATLAB function signature is different than the corresponding HDF5 library function. The following sections describe some general differences that you should keep in mind when using the MATLAB low-level HDF5 functions.

Output Parameters Become Return Values. Some HDF5 library functions use function parameters to return data on the right-hand side (RHS) of the function signature, i.e. as input parameters. The corresponding MATLAB function, because MATLAB allows multiple return values, moves these output parameters to the left-hand side (LHS) of the function signature, i.e. as return values. To illustrate, look at the `H5Dread` function. This function returns data in the `buf` parameter.

```
herr_t H5Dread(hid_t dataset_id, hid_t mem_type_id, hid_t mem_space_id,
        hid_t file_space_id, hid_t xfer_plist_id, void * buf ) /* C syntax */
```

The corresponding MATLAB function changes the output parameter `buf` into a return value. Note that the HDF5 error return is not used. In MATLAB, the nonzero or negative value `herr_t` return values become MATLAB errors. Use MATLAB `try-catch` statements to handle errors.

```
buf = H5D.read(dataset_id, mem_type_id, mem_space_id, file_space_id, plist_id)
```

String Length Parameters Unnecessary. The length parameter used by some HDF5 library functions to specify the length of string parameters are not necessary in the corresponding MATLAB function. For example, the `H5Aget_name` function in the HDF5 library includes a buffer as an output parameter and the size of the buffer as an input parameter.

```
ssize_t H5Aget_name(hid_t attr_id,size_t buf_size,char *buf ) /* C syntax */
```

The corresponding MATLAB function changes the output parameter `buf` into a return value and drops the `buf_size` parameter:

```
attr_name = H5A.get_name(attr_id)
```

Use Empty Array to Specify NULL. The MATLAB functions use empty arrays (`[]`) where HDF5 library functions accept the value `NULL`. For example, the `H5Dfill` function in the HDF5 library accepts the value `NULL` in place of a specified fill value.

```
herr_t H5Dfill(const void *fill, hid_t fill_type_id, void *buf,
               hid_t buf_type_id, hid_t space_id ) /* C syntax */
```

When using the corresponding MATLAB function, you can specify an empty array (`[]`) instead of `NULL`.

Specifying Multiple Constants. Some functions in the HDF5 library require you to specify an array of constants. For example, in the `H5Screate_simple` function, if you want to specify that each dimension in the data space can be unlimited, you use the constant `H5S_UNLIMITED` for each dimension in `maxdims`. In MATLAB, because you pass constants as text strings, you must use a cell array to achieve the same result. The following code fragment provides an example of using a cell array to specify this constant for each dimension of this data space.

```
ds_id = H5S.create_simple(2,[3 4],{'H5S_UNLIMITED' 'H5S_UNLIMITED'});
```

Mapping Between HDF5 Data Types and MATLAB Data Types

When the HDF5 low-level functions read data from an HDF5 file or write data to an HDF5 file, the functions map HDF5 data types to MATLAB data types automatically.

For *atomic* data types, such as commonly used binary formats for numbers (integers and floating point) and characters (ASCII), the mapping is typically straightforward because MATLAB supports similar types. See the table Mapping Between HDF5 Atomic Data Types and MATLAB Data Types on page 7-30 for a list of these mappings.

Mapping Between HDF5 Atomic Data Types and MATLAB Data Types

HDF5 Atomic Data Type	MATLAB Data Type
Bit-field	Array of packed 8-bit integers
Float	MATLAB single and double types, provided that they occupy 64 bits or fewer
Integer types, signed and unsigned	Equivalent MATLAB integer types, signed and unsigned
Opaque	Array of `uint8` values
Reference	Array of `uint8` values
String	MATLAB character arrays.

For *composite* data types, such as aggregations of one or more atomic data types into structures, multidimensional arrays, and variable-length data types (one-dimensional arrays), the mapping is sometimes ambiguous with reference to the HDF5 data type. In HDF5, a 5-by-5 data set containing a single `uint8` value in each element is distinct from a 1-by-1 data set containing a 5-by-5 array of `uint8` values. In the first case, the data set contains 25 observations of a single value; in the second case, the data set contains a single observation with 25 values. In MATLAB both of these data sets are represented by a 5-by-5 matrix.

If your data is a complex data set, you might need to create HDF5 data types directly to make sure you have the mapping you intend. See the table Mapping Between HDF5 Composite Data Types and MATLAB Data Types on page 7-31 for a list of the default mappings. You can specify the data type when you write data to the file using the `H5Dwrite` function. See the HDF5 data type interface documentation for more information.

Mapping Between HDF5 Composite Data Types and MATLAB Data Types

HDF5 Composite Data Type	MATLAB Data Type
Array	Extends the dimensionality of the data type which it contains. For example, an array of an array of integers in HDF5 would map onto a two dimensional array of integers in MATLAB.
Compound	MATLAB structure. Note: All structures representing HDF5 data in MATLAB are scalar.
Enumeration	Array of integers which each have an associated name
Variable Length	MATLAB 1-D cell arrays

Reporting Data Set Dimensions. The MATLAB low-level HDF5 functions report data set dimensions and the shape of data sets differently than the MATLAB high-level functions. For ease of use, the MATLAB high-level functions report data set dimensions consistent with MATLAB column-major indexing. To be consistent with the HDF5 library, and to support the possibility of nested data sets and complicated data types, the MATLAB low-level functions report array dimensions using the C row-major orientation.

Example: Using the MATLAB HDF5 Low-level Functions

This example shows how to use the MATLAB HDF5 low-level functions to write a data set to an HDF5 file and then read the data set from the file.

1. Create the MATLAB variable that you want to write to the HDF5 file. The examples creates a three-dimensional array of `uint8` data.

```
testdata = uint8(ones(5,10,3));
```

2. Create the HDF5 file or open an existing file. The example creates a new HDF5 file, named `my_file.h5`, in the system temp directory.

```
filename = fullfile(tempdir,'my_file.h5');

fileID = H5F.create(filename,'H5F_ACC_TRUNC','H5P_DEFAULT','H5P_DEFAULT');
```

In HDF5, you use the `H5Fcreate` function to create a file. The example uses the MATLAB equivalent, `H5F.create`. As arguments, specify the name you want to assign to the file, the type of access you want to the file (`'H5F_ACC_TRUNC'` in the example), and optional additional characteristics specified by a file creation property list and a file access property list. This example uses default values for these property lists (`'H5P_DEFAULT'`). In the example, note how the C constants are passed to the MATLAB functions as strings. The function returns an ID to the HDF5 file.

3. Create the data set in the file to hold the MATLAB variable. In the HDF5 programming model, you must define the data type and dimensionality (data space) of the data set as separate entities.

   a. Specify the data type used by the data set. In HDF5, you use the `H5Tcopy` function to create integer or floating-point data types. The example uses the corresponding MATLAB function, `H5T.copy`, to create a `uint8` data type because the MATLAB data is `uint8`. The function returns a data type ID.

   ```
   datatypeID = H5T.copy('H5T_NATIVE_UINT8');
   ```

   b. Specify the dimensions of the data set. In HDF5, you use the `H5Screate_simple` routine to create a data space. The example uses the corresponding MATLAB function, `H5S.create_simple`, to specify the dimensions. The function returns a data space ID.

   ```
   dims(1) = 5;
   dims(2) = 10;
   dims(3) = 3
   dataspaceID = H5S.create_simple(3, dims, []);
   ```

   c. Create the data set. In HDF5, you use the `H5Dcreate` routine to create a data set. The example uses the corresponding MATLAB function, `H5D.create`, specifying the file ID, the name you want to assign to the data set, data type ID, the data space ID, and a data set creation property list ID as arguments. The example uses the defaults for the property lists. The function returns a data set ID.

   ```
   dsetname = 'my_dataset';
   datasetID = H5D.create(fileID,dsetname,datatypeID,dataspaceID,'H5P_DEFAULT');
   ```

Note To write a large data set, you must use the chunking capability of the HDF5 library. To do this, create a property list and use the `H5P.set_chunk` function to set the chunk size in the property list. In the following example, the dimensions of the data set are `dims = [2^16 2^16]` and the chunk size is 1024-by-1024. You then pass the property list as the last argument to the data set creation function, `H5D.create`, instead of using the `H5P_DEFAULT` value.

```
plistID = H5P.create('H5P_DATASET_CREATE'); % create property list

chunk_size = min([1024 1024], dims); % define chunk size
H5P.set_chunk(plistID, chunk_size); % set chunk size in property list

datasetID = H5D.create(fileID, dsetname, datatypeID, dataspaceID, plistID);
```

---

4. Write the data to the data set. In HDF5, you use the `H5Dwrite` routine to write data to a data set. The example uses the corresponding MATLAB function, `H5D.write`, specifying as arguments the data set ID, the memory data type ID, the memory space ID, the data space ID, the transfer property list ID and the name of the MATLAB variable to be written to the data set.

   You can use the memory data type to specify the data type used to represent the data in the file. The example uses the constant `'H5ML_DEFAULT'` which lets the MATLAB function do an automatic mapping to HDF5 data types. The memory data space ID and the data set's data space ID specify to write subsets of the data set to the file. The example uses the constant `'H5S_ALL'` to write all the data to the file and uses the default property list.

Note Because HDF5 stores data in row-major order and MATLAB accesses data in column-major order, you should permute your data before writing it to the file.

```
data_perm = permute(testdata,[3 2 1]);
```

```
H5D.write(datasetID,'H5ML_DEFAULT','H5S_ALL','H5S_ALL', ...
          'H5P_DEFAULT',data_perm);
```

5. Close the data set, data space, data type, and file objects. If used inside a MATLAB function, these identifiers are closed automatically when they go out of scope.

   ```
   H5D.close(datasetID);
   H5S.close(dataspaceID);
   H5T.close(datatypeID);
   H5F.close(fileID);
   ```

6. To read the data set you wrote to the file, you must open the file. In HDF5, you use the `H5Fopen` routine to open an HDF5 file, specifying the name of the file, the access mode, and a property list as arguments. The example uses the corresponding MATLAB function, `H5F.open`, opening the file for read-only access.

   ```
   fileID = H5F.open(filename,'H5F_ACC_RDONLY','H5P_DEFAULT');
   ```

7. After opening the file, you must open the data set. In HDF5, you use the `H5Dopen` function to open a data set. The example uses the corresponding MATLAB function, `H5D.open`, specifying as arguments the file ID and the name of the data set, defined earlier in the example.

   ```
   datasetID = H5D.open(fileID, dsetname);
   ```

8. After opening the data set, you can read the data into the MATLAB workspace. In HDF5, you use the `H5Dread` function to read an HDF5 file. The example uses the corresponding MATLAB function, `H5D.read`, specifying as arguments the data set ID, the memory data type ID, the memory space ID, the data space ID, and the transfer property list ID.

   ```
   returned_data = H5D.read(datasetID,'H5ML_DEFAULT',...
                            'H5S_ALL','H5S_ALL','H5P_DEFAULT');
   ```

   Note that the data returned must be indexed in reverse order: HDF5 stores the data in row-major order; MATLAB accesses data in column-major order. To rearrange the data into column-major order, use the MATLAB `permute` function.

```
data = permute(returned_data,[3 2 1]);
```

You can compare the original MATLAB variable, `testdata`, with the variable just created, `data`, to see if they are the same.

Hierarchical Data Format (HDF4) Files

In this section...

Note For information about importing HDF5 data, which is a separate, incompatible format, see "Hierarchical Data Format (HDF5) Files" on page 7-11.

Using the HDF Import Tool

Hierarchical Data Format (HDF4) is a general-purpose, machine-independent standard for storing scientific data in files, developed by the National Center for Supercomputing Applications (NCSA). For more information about these file formats, read the HDF documentation at the HDF Web site (`www.hdfgroup.org`).

HDF-EOS is an extension of HDF4 that was developed by the National Aeronautics and Space Administration (NASA) for storage of data returned from the Earth Observing System (EOS). For more information about this extension to HDF4, see the HDF-EOS documentation at the NASA Web site (`www.hdfeos.org`).

The HDF Import Tool is a graphical user interface that you can use to navigate through HDF4 or HDF-EOS files and import data from them. Importing data using the HDF Import Tool involves these steps:

- "Step 1: Opening an HDF4 File in the HDF Import Tool" on page 7-37
- "Step 2: Selecting a Data Set in an HDF File" on page 7-38
- "Step 3: Specifying a Subset of the Data (Optional)" on page 7-39
- "Step 4: Importing Data and Metadata" on page 7-40

- "Step 5: Closing HDF Files and the HDF Import Tool" on page 7-41

The following sections provide more detail about each of these steps.

Step 1: Opening an HDF4 File in the HDF Import Tool

Open an HDF4 or HDF-EOS file in MATLAB using one of the following methods:

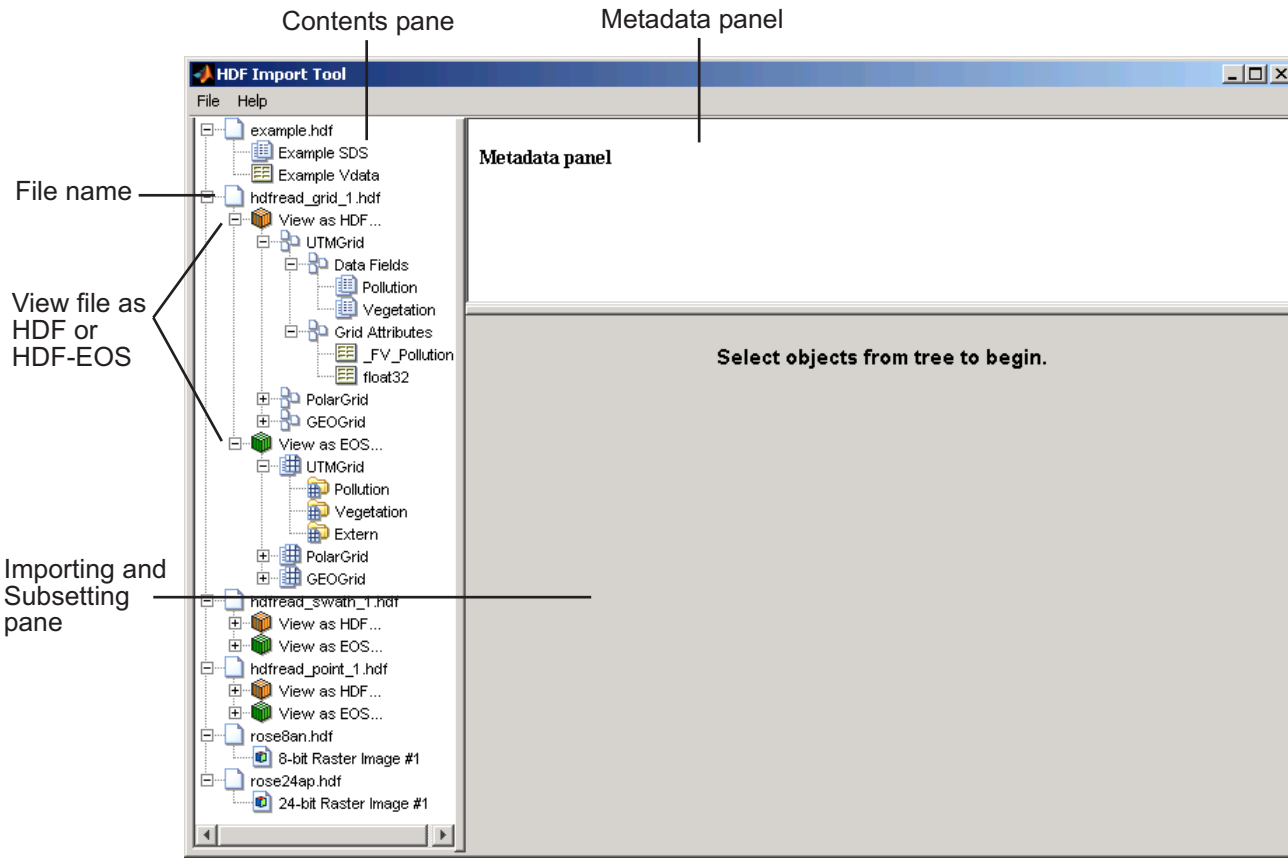
- Choose the **Import Data** option from the MATLAB **File** menu. If you select an HDF4 or HDF-EOS file, the MATLAB Import Wizard automatically starts the HDF Import Tool.
- Start the HDF Import Tool by entering the `hdftool` command at the MATLAB command line:

  ```
  hdftool
  ```

  This opens an empty HDF Import Tool. To open a file, click the **Open** option on the HDFTool **File** menu and select the file you want to open. You can open multiple files in the HDF Import Tool.
- Open an HDF or HDF-EOS file by specifying the file name with the `hdftool` command on the MATLAB command line:

  ```
  hdftool('example.hdf')
  ```

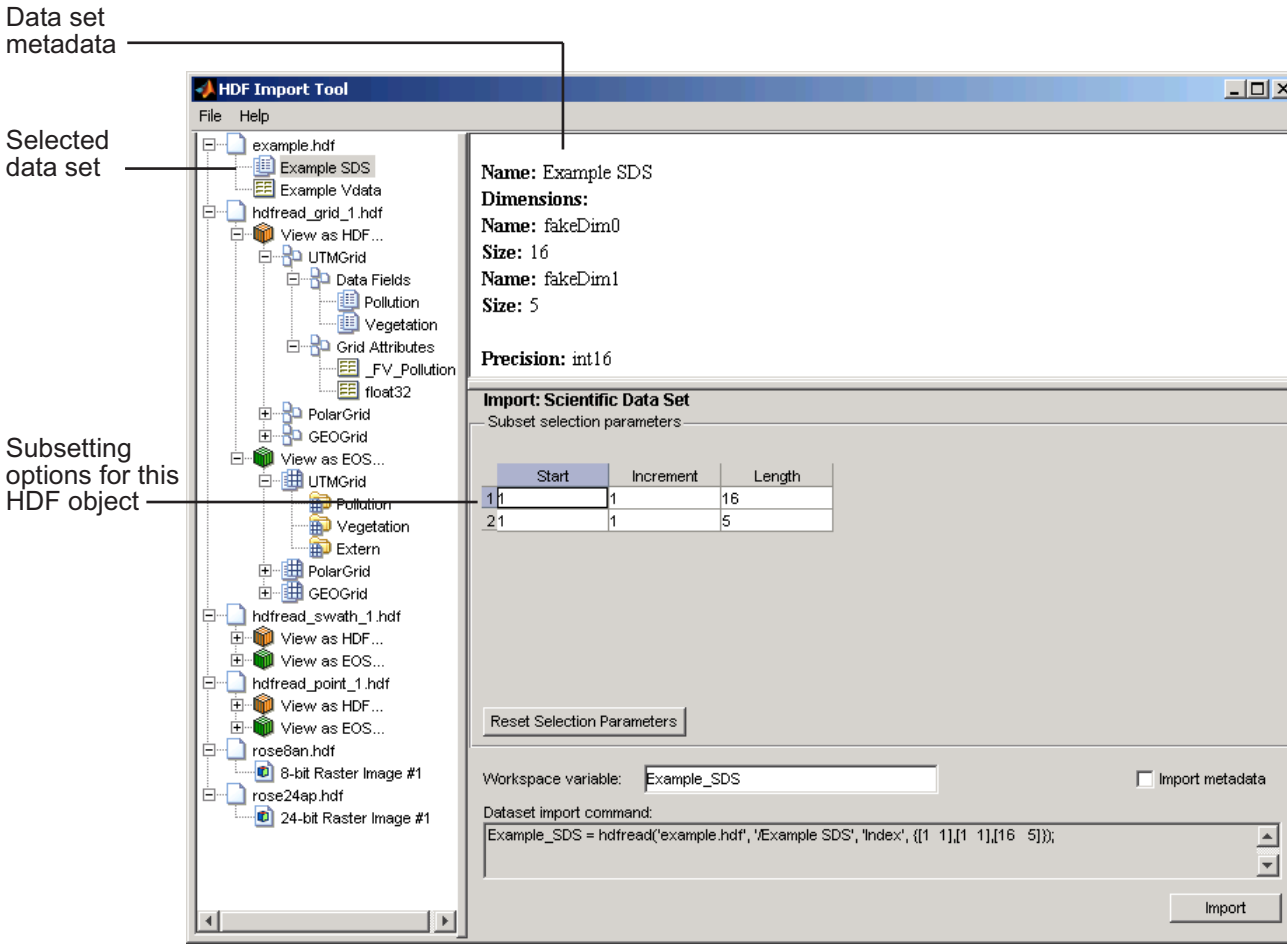
Viewing a File in the HDF Import Tool. When you open an HDF4 or HDF-EOS file in the HDF Import Tool, the tool displays the contents of the file in the Contents pane. You can use this pane to navigate within the file to see what data sets it contains. You can view the contents of HDF-EOS files as HDF data sets or as HDF-EOS files. The icon in the contents pane indicates the view, as illustrated in the following figure. Note that these are just two views of the same data.



Step 2: Selecting a Data Set in an HDF File

To import a data set, you must first select the data set in the contents pane of the HDF Import Tool. Use the Contents pane to view the contents of the file and navigate to the data set you want to import.

For example, the following figure shows the data set `Example SDS` in the HDF file selected. Once you select a data set, the Metadata panel displays information about the data set and the importing and subsetting pane displays subsetting options available for this type of HDF object.



Step 3: Specifying a Subset of the Data (Optional)

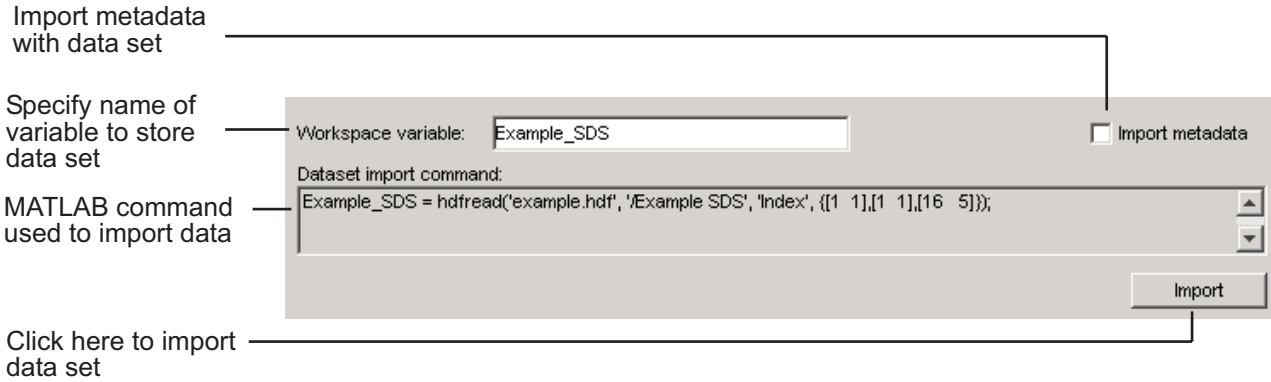
When you select a data set in the contents pane, the importing and subsetting pane displays the subsetting options available for that type of HDF object. The subsetting options displayed vary depending on the type of HDF object. For more information, see “Using the HDF Import Tool Subsetting Options” on page 7-41.

Step 4: Importing Data and Metadata

To import the data set you have selected, click the **Import** button, bottom right corner of the Importing and Subsetting pane. Using the Importing and Subsetting pane, you can

- Specify the name of the workspace variable — By default, the HDF Import Tool uses the name of the HDF4 data set as the name of the MATLAB workspace variable. In the following figure, the variable name is `Example_SDS`. To specify another name, enter text in the **Workspace Variable** text box.
- Specify whether to import metadata associated with the data set — To import any metadata that might be associated with the data set, select the **Import Metadata** check box. To store the metadata, the HDF Import Tool creates a second variable in the workspace with the same name with "`_info`" appended to it. For example, if you select this check box, the name of the metadata variable for the data set in the figure would be `Example_SDS_info`.
- Save the data set import command syntax — The **Dataset import command** text window displays the MATLAB command used to import the data set. This text is not editable, but you can copy and paste it into the MATLAB Command Window or a text editor for reuse.

The following figure shows how to specify these options in the HDF Import Tool.



Step 5: Closing HDF Files and the HDF Import Tool

To close a file, select the file in the contents pane and click **Close File** on the HDF Import Tool **File** menu.

To close all the files open in the HDF Import Tool, click **Close All Files** on the HDF Import Tool **File** menu.

To close the tool, click **Close HDFTool** in the HDF Import Tool **File** menu or click the **Close** button in the upper right corner of the tool.

If you used the `hdftool` syntax that returns a handle to the tool,

```
h = hdftool('example.hdf')
```

you can use the `close(h)` command to close the tool from the MATLAB command line.

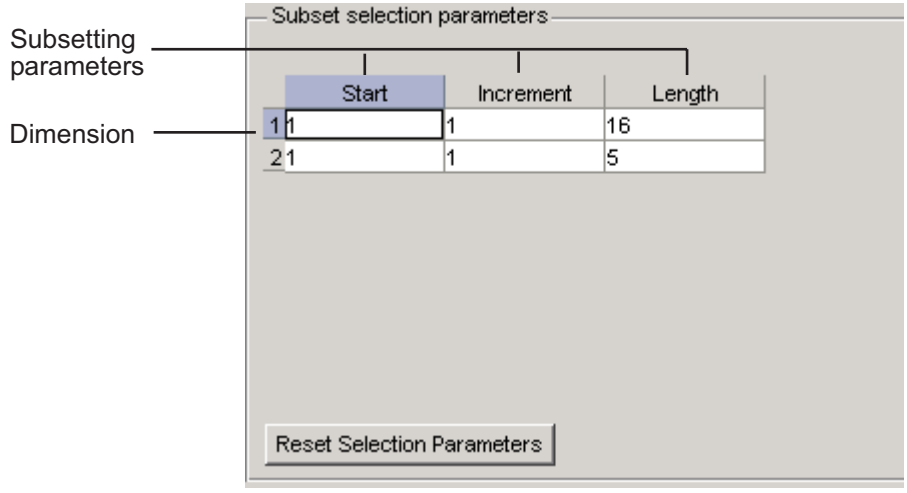
Using the HDF Import Tool Subsetting Options

When you select a data set, the importing and subsetting pane displays the subsetting options available for that type of data set. The following sections provide information about these subsetting options for all supported data set types. For general information about the HDF Import tool, see “Using the HDF Import Tool” on page 7-36.

Note To use these data subsetting options effectively, you must understand the HDF and HDF-EOS data formats. Therefore, use this documentation in conjunction with the HDF documentation (`www.hdfgroup.org`) and the HDF-EOS documentation (`www.hdfeos.org`).

HDF Scientific Data Sets (SD)

The HDF scientific data set (SD) is a group of data structures used to store and describe multidimensional arrays of scientific data. Using the HDF Import Tool subsetting parameters, you can import a subset of an HDF scientific data set by specifying the location, range, and number of values to be read along each dimension.



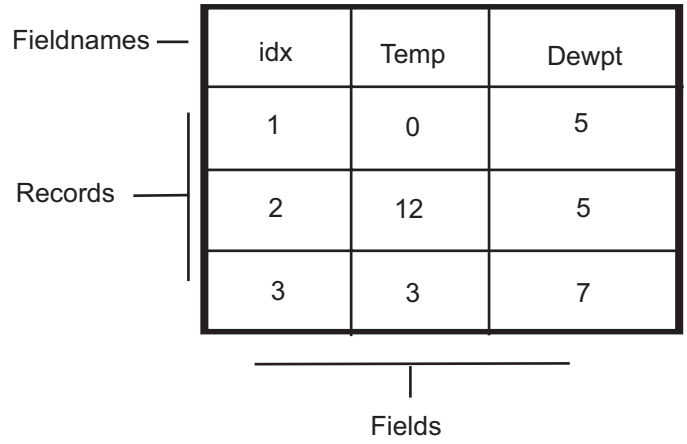
The subsetting parameters are:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

HDF Vdata

HDF Vdata data sets provide a framework for storing customized tables. A Vdata table consists of a collection of records whose values are stored in

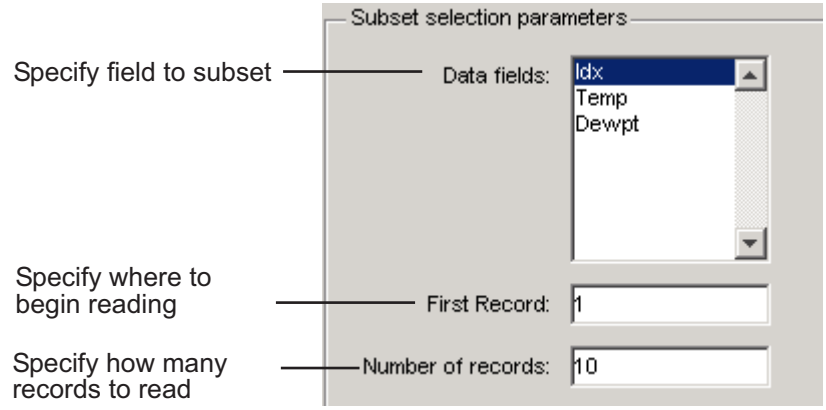
fixed-length fields. All records have the same structure and all values in each field have the same data type. Each field is identified by a name. The following figure illustrates a Vdata table.



You can import a subset of an HDF Vdata data set in the following ways:

- Specifying the name of the field that you want to import
- Specifying the range of records that you want to import

The following figure shows how you specify these subsetting parameters for Vdata.

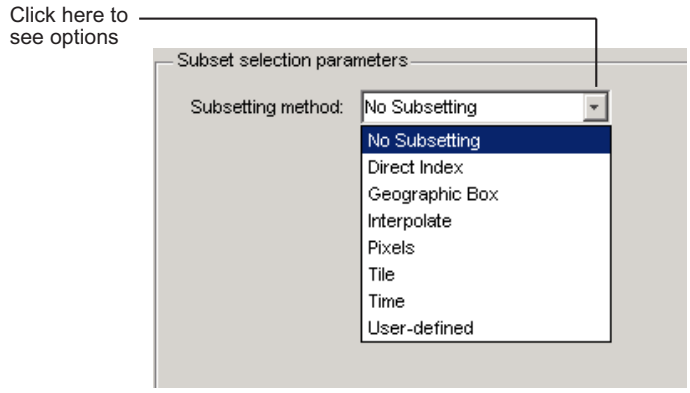


HDF-EOS Grid Data

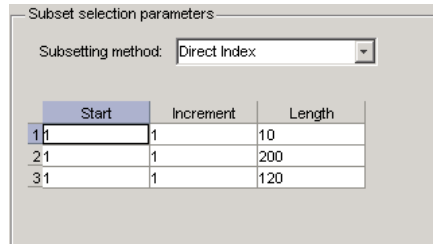
In HDF-EOS Grid data, a rectilinear grid overlays a map. The map uses a known map projection. The HDF Import Tool supports the following mutually exclusive subsetting options for Grid data:

- "Direct Index" on page 7-44
- "Geographic Box" on page 7-45
- "Interpolation" on page 7-46
- "Pixels" on page 7-47
- "Tile" on page 7-47
- "Time" on page 7-47
- "User-Defined" on page 7-48

To access these options, click the Subsetting method menu in the importing and subsetting pane.



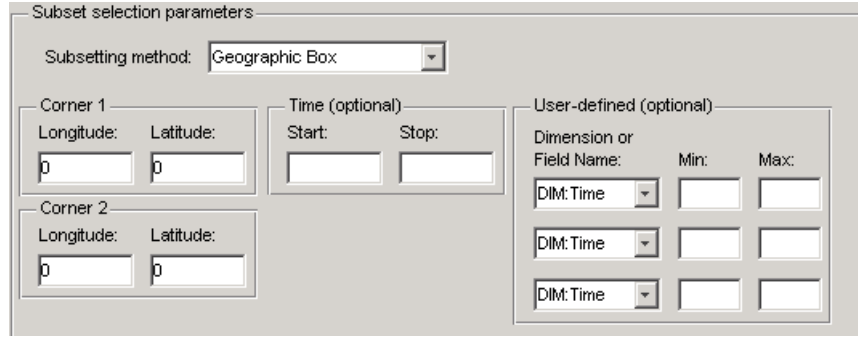
Direct Index. You can import a subset of an HDF-EOS Grid data set by specifying the location, range, and number of values to be read along each dimension.



Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

Geographic Box. You can import a subset of an HDF-EOS Grid data set by specifying the rectangular area of the grid that you are interested in. To define this rectangular area, you must specify two points, using longitude and latitude in decimal degrees. These points are two corners of the rectangular area. Typically, **Corner 1** is the upper-left corner of the box, and **Corner 2** is the lower-right corner of the box.

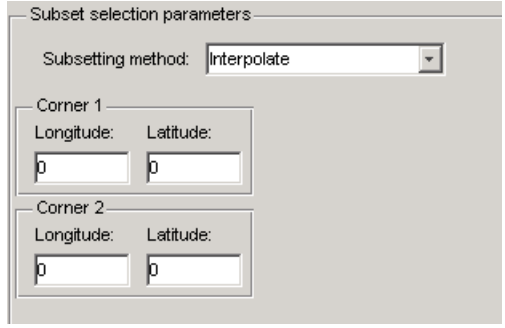


Optionally, you can further define the subset of data you are interested in by using Time parameters (see “Time” on page 7-47) or by specifying other User-Defined subsetting parameters (see “User-Defined” on page 7-48).

Interpolation. Interpolation is the process of estimating a pixel value at a location in between other pixels. In interpolation, the value of a particular pixel is determined by computing the weighted average of some set of pixels in the vicinity of the pixel.

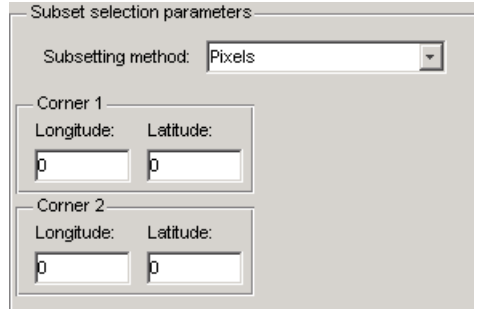
You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box

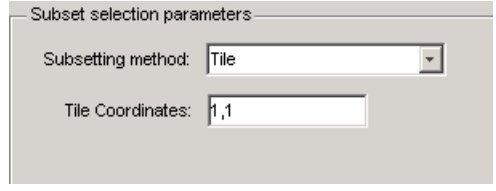


Pixels. You can import a subset of the pixels in a Grid data set by defining a rectangular area over the grid. You define the region used for bilinear interpolation by specifying two points that are corners of the interpolation area:

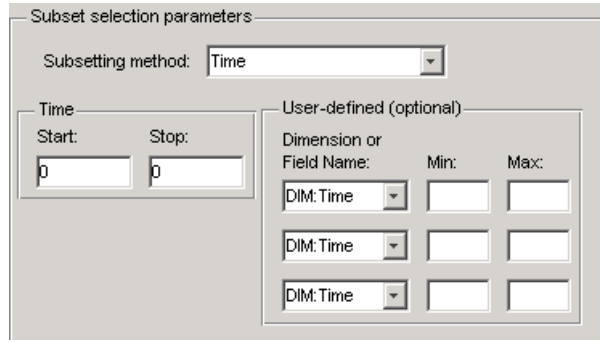
- **Corner 1** – Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box



Tile. In HDF-EOS Grid data, a rectilinear grid overlays a map. Each rectangle defined by the horizontal and vertical lines of the grid is referred to as a *tile*. If the HDF-EOS Grid data is stored as tiles, you can import a subset of the data by specifying the coordinates of the tile you are interested in. Tile coordinates are 1-based, with the upper-left corner of a two-dimensional data set identified as `1,1`. In a three-dimensional data set, this tile would be referenced as `1,1,1`.

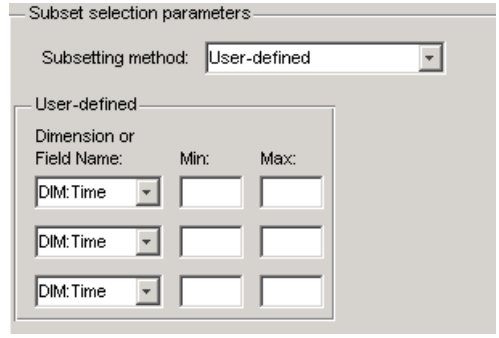


Time. You can import a subset of the Grid data set by specifying a time period. You must specify both the start time and the stop time (the endpoint of the time span). The units (hours, minutes, seconds) used to specify the time are defined by the data set.



Along with these time parameters, you can optionally further define the subset of data to import by supplying user-defined parameters.

User-Defined. You can import a subset of the Grid data set by specifying user-defined subsetting parameters.

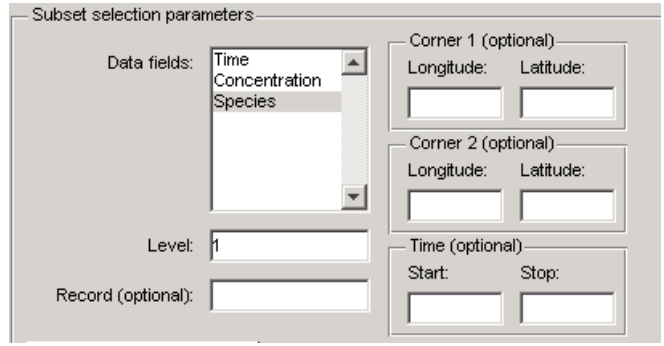


When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters `DIM:`.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF-EOS Point Data

HDF-EOS Point data sets are tables. You can import a subset of an HDF-EOS Point data set by specifying field names and level. Optionally, you can refine the subsetting by specifying the range of records you want to import, by defining a rectangular area, or by specifying a time period. For information about specifying a rectangular area, see “Geographic Box” on page 7-45. For information about subsetting by time, see “Time” on page 7-47.



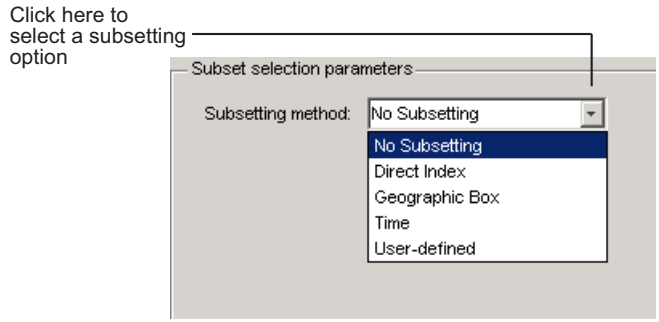
HDF-EOS Swath Data

HDF-EOS Swath data is data that is produced by a satellite as it traces a path over the earth. This path is called its ground track. The sensor aboard the satellite takes a series of scans perpendicular to the ground track. Swath data can also include a vertical measure as a third dimension. For example, this vertical dimension can represent the height above the Earth of the sensor.

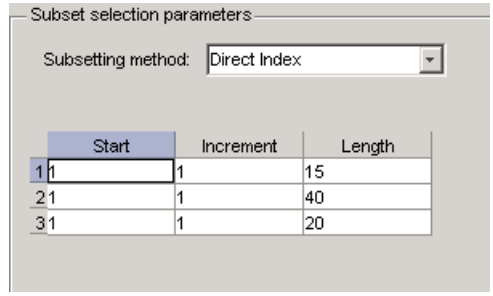
The HDF Import Tool supports the following mutually exclusive subsetting options for Swath data:

- “Direct Index” on page 7-50
- “Geographic Box” on page 7-51
- “Time” on page 7-52
- “User-Defined” on page 7-52

To access these options, click the `Subsetting method` menu in the **Importing and Subsetting** pane.



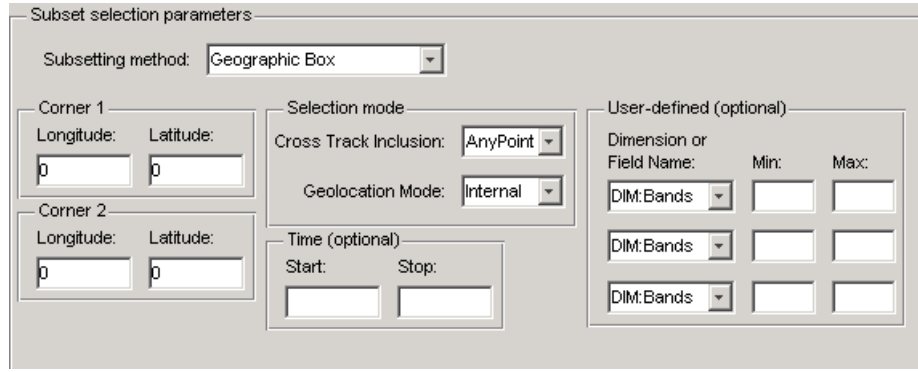
Direct Index. You can import a subset of an HDF-EOS Swath data set by specifying the location, range, and number of values to be read along each dimension.



Each row represents a dimension in the data set and each column represents these subsetting parameters:

- **Start** — Specifies the position on the dimension to begin reading. The default value is 1, which starts reading at the first element of each dimension. The values specified must not exceed the size of the relevant dimension of the data set.
- **Increment** — Specifies the interval between the values to read. The default value is 1, which reads every element of the data set.
- **Length** — Specifies how much data to read along each dimension. The default value is the length of the dimension, which causes all the data to be read.

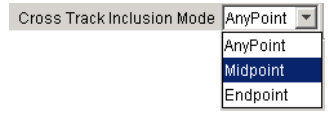
Geographic Box. You can import a subset of an HDF-EOS Swath data set by specifying the rectangular area of the grid that you are interested in and by specifying the selection Mode.



You define the rectangular area by specifying two points that specify two corners of the box:

- **Corner 1** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 1** is the upper-left corner of the box.
- **Corner 2** — Specify longitude and latitude values in decimal degrees. Typically, **Corner 2** is the lower-right corner of the box.

You specify the selection mode by choosing the type of **Cross Track Inclusion** and the **Geolocation mode**. The **Cross Track Inclusion** value determines how much of the area of the geographic box that you define must fall within the boundaries of the swath.

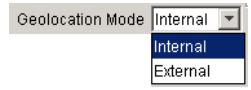


Select from these values:

- **AnyPoint** — Any part of the box overlaps with the swath.
- **Midpoint** — At least half of the box overlaps with the swath.

- **Endpoint** — All of the area defined by the box overlaps with the swath.

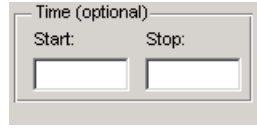
The **Geolocation Mode** value specifies whether geolocation fields and data must be in the same swath.



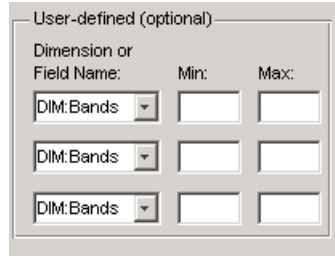
Select from these values:

- **Internal** — Geolocation fields and data fields must be in the same swath.
- **External** — Geolocation fields and data fields can be in different swaths.

Time. You can optionally also subset swath data by specifying a time period. The units used (hours, minutes, seconds) to specify the time are defined by the data set



User-Defined. You can optionally also subset a swath data set by specifying user-defined parameters.



When specifying user-defined parameters, you must first specify whether you are subsetting along a dimension or by field. Select the dimension or field by name using the **Dimension or Field Name** menu. Dimension names are prefixed with the characters `DIM:`.

Once you specify the dimension or field, you use **Min** and **Max** to specify the range of values that you want to import. For dimensions, **Min** and **Max** represent a range of *elements*. For fields, **Min** and **Max** represent a range of *values*.

HDF Raster Image Data

For 8-bit HDF raster image data, you can specify the colormap.

Using the MATLAB HDF4 High-Level Functions

To import data from an HDF or HDF-EOS file, you can use the MATLAB HDF4 high-level function `hdfread`. The `hdfread` function provides a programmatic way to import data from an HDF4 or HDF-EOS file that still hides many of the details that you need to know if you use the low-level HDF functions, described in "Using the HDF4 Low-Level Functions" on page 7-56. You can also import HDF4 data using an interactive GUI, described in "Using the HDF Import Tool" on page 7-36.

This section describes these high-level MATLAB HDF functions, including

- "Using hdfinfo to Get Information About an HDF4 File" on page 7-53
- "Using hdfread to Import Data from an HDF4 File" on page 7-54

To export data to an HDF4 file, you must use the MATLAB HDF4 low-level functions.

Using hdfinfo to Get Information About an HDF4 File

To get information about the contents of an HDF4 file, use the `hdfinfo` function. The `hdfinfo` function returns a structure that contains information about the file and the data in the file.

Note You can also use the HDF Import Tool to get information about the contents of an HDF4 file. See "Using the HDF Import Tool" on page 7-36 for more information.

This example returns information about a sample HDF4 file included with MATLAB:

```
info = hdfinfo('example.hdf')

info =

    Filename: 'example.hdf'
         SDS: [1x1 struct]
       Vdata: [1x1 struct]
```

To get information about the data sets stored in the file, look at the `SDS` field.

Using hdfread to Import Data from an HDF4 File

To use the`hdfread` function, you must specify the data set that you want to read. You can specify the filename and the data set name as arguments, or you can specify a structure returned by the `hdfinfo` function that contains this information. The following example shows both methods. For information about how to import a subset of the data in a data set, see "Reading a Subset of the Data in a Data Set" on page 7-56.

1. Determine the names of data sets in the HDF4 file, using the `hdfinfo` function.

   ```
   info = hdfinfo('example.hdf')

   info =

       Filename: 'example.hdf'
            SDS: [1x1 struct]
          Vdata: [1x1 struct]
   ```

   To determine the names and other information about the data sets in the file, look at the contents of the `SDS` field. The `Name` field in the SDS structure gives the name of the data set.

   ```
   dsets = info.SDS

   dsets =
   ```

```
      Filename: 'example.hdf'
          Type: 'Scientific Data Set'
          Name: 'Example SDS'
          Rank: 2
      DataType: 'int16'
    Attributes: []
          Dims: [2x1 struct]
         Label: {}
   Description: {}
         Index: 0
```

2 Read the data set from the HDF4 file, using the `hdfread` function. Specify the name of the data set as a parameter to the function. Note that the data set name is case sensitive. This example returns a 16-by-5 array:

```
dset = hdfread('example.hdf', 'Example SDS');

dset =

     3     4     5     6     7
     4     5     6     7     8
     5     6     7     8     9
     6     7     8     9    10
     7     8     9    10    11
     8     9    10    11    12
     9    10    11    12    13
    10    11    12    13    14
    11    12    13    14    15
    12    13    14    15    16
    13    14    15    16    17
    14    15    16    17    18
    15    16    17    18    19
    16    17    18    19    20
    17    18    19    20    21
    18    19    20    21    22
```

Alternatively, you can specify the specific field in the structure returned by `hdfinfo` that contains this information. For example, to read a scientific data set, use the `SDS` field.

```
dset = hdfread(info.SDS);
```

Reading a Subset of the Data in a Data Set. To read a subset of a data set, you can use the optional `'index'` parameter. The value of the index parameter is a cell array of three vectors that specify the location in the data set to start reading, the skip interval (e.g., read every other data item), and the amount of data to read (e.g., the length along each dimension). In HDF4 terminology, these parameters are called the *start*, *stride*, and *edge* values.

For example, this code

- Starts reading data at the third row, third column (`[3 3]`).
- Reads every element in the array (`[]`).
- Reads 10 rows and 2 columns (`[10 2]`).

```
subset = hdfread('Example.hdf','Example SDS',...
                 'Index',{[3 3],[],[10 2 ]})

subset =

      7      8
      8      9
      9     10
     10     11
     11     12
     12     13
     13     14
     14     15
     15     16
     16     17
```

Using the HDF4 Low-Level Functions

This section describes how to use MATLAB functions to access the HDF4 Application Programming Interfaces (APIs). These APIs are libraries of C routines that you can use to import data from an HDF4 file or export data from the MATLAB workspace into an HDF4 file. To import or export data, you must use the functions in the HDF4 API associated with the particular HDF4 data type you are working with. Each API has a particular programming model, that is, a prescribed way to use the routines to write data sets to the file. To illustrate this concept, this section describes the programming

model of one particular HDF4 API: the HDF4 Scientific Data (SD) API. For a complete list of the HDF4 APIs supported by MATLAB and the functions you use to access each one, see the `hdf` reference page.

Note This section does not attempt to describe all HDF4 features and routines. To use the MATLAB HDF4 functions effectively, you must refer to the official NCSA documentation at the HDF Web site (`www.hdfgroup.org`).

Topics covered include

Understanding the HDF4 to MATLAB Syntax Mapping

Each HDF4 API includes many individual routines that you use to read data from files, write data to files, and perform other related functions. For example, the HDF4 Scientific Data (SD) API includes separate C routines to open (`SDopen`), close (`SDend`), and read data (`SDreaddata`).

Instead of supporting each routine in the HDF4 APIs, MATLAB provides a single function that serves as a gateway to all the routines in a particular HDF4 API. For example, the HDF Scientific Data (SD) API includes the C routine `SDend` to close an HDF4 file:

```
status = SDend(sd_id); /* C code */
```

To call this routine from MATLAB, use the MATLAB function associated with the SD API, `hdfsd`. You must specify the name of the routine, minus the API acronym, as the first argument and pass any other required arguments to the routine in the order they are expected. For example,

```
status = hdfsd('end',sd_id); % MATLAB code
```

Handling HDF4 Routines with Output Arguments. Some HDF4 API routines use output arguments to return data. Because MATLAB does not support output arguments, you must specify these arguments as return values.

For example, the `SDfileinfo` routine returns data about an HDF4 file in two output arguments, `ndatasets` and `nglobal_atts`. Here is the C code:

```
status = SDfileinfo(sd_id, ndatasets, nglobal_atts);
```

To call this routine from MATLAB, change the output arguments into return values:

```
[ndatasets, nglobal_atts, status] = hdfsd('fileinfo',sd_id);
```

Specify the return values in the same order as they appear as output arguments. The function status return value is always specified as the last return value.

Example: Importing Data Using the HDF4 SD API Functions

To illustrate using HDF4 API routines in MATLAB, the following sections provide a step-by-step example of how to import HDF4 Scientific Data (SD) into the MATLAB workspace.

Note The following sections, when referring to specific routines in the HDF4 SD API, use the C library name rather than the MATLAB function name. The MATLAB syntax is used in all examples.

Step 1: Opening the HDF4 File. To import an HDF4 SD data set, you must first open the file using the SD API routine `SDstart`. (In HDF4 terminology, the numeric arrays stored in HDF4 files are called data sets.) In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `start` in this case.
- Name of the file you want to open.
- Mode in which you want to open it. The following table lists the file access modes supported by the `SDstart` routine. In MATLAB, you specify these modes as text strings. You can specify the full HDF4 mode name or one of the abbreviated forms listed in the table.

HDF4 File Creation Mode	HDF4 Mode Name	MATLAB String
Create a new file	`'DFACC_CREATE'`	`'create'`
Read access	`'DFACC_RDONLY'`	`'read'` or `'rdonly'`
Read and write access	`'DFACC_RDWR'`	`'rdwr'` or `'write'`

For example, this code opens the file `mydata.hdf` for read access:

```
sd_id = hdfsd('start','mydata.hdf','read');
```

If `SDstart` can find and open the file specified, it returns an HDF4 SD file identifier, named `sd_id` in the example. Otherwise, it returns `-1`.

Step 2: Retrieving Information About the HDF4 File. To get information about an HDF4 file, you must use the SD API routine `SDfileinfo`. This function returns the number of data sets in the file and the number of global attributes in the file, if any. (For more information about global attributes, see “Example: Exporting Data Using the HDF4 SD API Functions” on page 7-64.) In MATLAB, you use the `hdfsd` function, specifying the following arguments:

- Name of the SD API routine, `fileinfo` in this case
- SD file identifier, `sd_id`, returned by `SDstart`

In this example, the HDF4 file contains three data sets and one global attribute.

```
[ndatasets, nglobal_atts, stat] = hdfsd('fileinfo',sd_id)

ndatasets =
    3

nglobal_atts =
    1

status =
    0
```

Step 3: Retrieving Attributes from an HDF4 File (Optional). HDF4 files can optionally include information, called *attributes*, that describes the data the file contains. Attributes associated with an entire HDF4 file are called *global* attributes. Attributes associated with a data set are called *local* attributes. (You can also associate attributes with files or dimensions. For more information, see "Step 4: Writing Metadata to an HDF4 File" on page 7-69.)

To retrieve attributes from an HDF4 file, use the HDF4 API routine `SDreadattr`. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `readattr` in this case.
- File identifier (`sd_id`) returned by `SDstart`, for global attributes, or the data set identifier for local attributes. (See "Step 4: Selecting the Data Sets to Import" on page 7-61 to learn how to get a data set identifier.)
- Index identifying the attribute you want to view. HDF4 uses zero-based indexing. If you know the name of an attribute but not its index, use the `SDfindattr` routine to determine the index value associated with the attribute.

For example, this code returns the contents of the first global attribute, which is the character string `my global attribute`:

```
attr_idx = 0;
[attr, status] = hdfsd('readattr', sd_id, attr_idx);

attr =
    my global attribute
```

Step 4: Selecting the Data Sets to Import. To select a data set, use the SD API routine `SDselect`. In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `select` in this case
- HDF4 SD file identifier (`sd_id`) returned by `SDstart`

If `SDselect` finds the specified data set in the file, it returns an HDF4 SD data set identifier, called `sds_id` in the example. If it cannot find the data set, it returns `-1`.

Note Do not confuse HDF4 SD *file* identifiers, named `sd_id` in the examples, with HDF4 SD *data set* identifiers, named `sds_id` in the examples.

```
sds_id = hdfsd('select',sd_id,1)
```

Step 5: Getting Information About a Data Set. To read a data set, you must get information about the data set, such as its name, size, and data type. In the HDF4 SD API, you use the `SDgetinfo` routine to gather this information. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `getinfo` in this case
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`

This code retrieves information about the data set identified by `sds_id`:

```
[dsname, dsndims, dsdims, dstype, dsatts, stat] =
              hdfsd('getinfo',sds_id)
dsname =
      A
```

```
dsndims =
      2

dsdims =
      5     3

dstype =
      double

dsatts =
      0

stat =
      0
```

Step 6: Reading Data from the HDF4 File. To read data from an HDF4 file, you must use the `SDreaddata` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API function, `readdata` in this case.
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`.
- Location in the data set where you want to start reading data, specified as a vector of index values, called the *start* vector. To read from the beginning of a data set, specify zero for each element of the start vector. Use `SDgetinfo` to determine the dimensions of the data set.
- Number of elements along each dimension to skip between each read operation, specified as a vector of scalar values, called the *stride* vector. To read every element of a data set, specify `1` as the value for each element of the vector or specify an empty array (`[]`).
- Total number of elements to read along each dimension, specified as a vector of scalar values, called the *edges* vector. To read every element of a data set, set each element of the edges vector to the size of each dimension of the data set. Use `SDgetinfo` to determine these sizes.

Note `SDgetinfo` returns dimension values in row-major order, the ordering used by HDF4. Because MATLAB stores data in column-major order, you must specify the dimensions in column-major order, that is, `[columns,rows]`. In addition, you must use zero-based indexing in these arguments.

For example, to read the entire contents of a data set, use this code:

```
[ds_name, ds_ndims, ds_dims, ds_type, ds_atts, stat] =
hdfsd('getinfo',sds_id);

ds_start = zeros(1,ds_ndims); % Creates the vector [0 0]
ds_stride = [];
ds_edges = ds_dims;

[ds_data, status] =
            hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);

disp(ds_data)
    1    2    3    4    5
    6    7    8    9    10
   11   12   13   14    15
```

To read less than the entire data set, use the start, stride, and edges vectors to specify where you want to start reading data and how much data you want to read. For example, this code reads the entire second row of the sample data set:

```
ds_start = [0 1]; % Start reading at the first column, second row
ds_stride = []; % Read each element
ds_edges = [5 1]; % Read a 1-by-5 vector of data

[ds_data, status] =
           hdfsd('readdata',sds_id,ds_start,ds_stride,ds_edges);
```

Step 7: Closing the HDF4 Data Set. After writing data to a data set in an HDF4 file, you must close access to the data set. In the HDF4 SD API, you use the `SDendaccess` routine to close a data set. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `endaccess` in this case
- HDF4 SD data set identifier (`sds_id`) returned by `SDselect`

For example, this code closes the data set:

```
stat = hdfsd('endaccess',sds_id);
```

You must close access to all the data sets in an HDF4 file before closing it.

Step 8: Closing the HDF4 File. After writing data to a data set and closing the data set, you must also close the HDF4 file. In the HDF4 SD API, you use the `SDend` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `end` in this case
- HDF4 SD file identifier (`sd_id`) returned by `SDstart`

For example, this code closes the data set:

```
stat = hdfsd('end',sd_id);
```

Example: Exporting Data Using the HDF4 SD API Functions

The following sections provide a step-by-step example of how to export data from the MATLAB workspace to an HDF4 file using Scientific Data (SD) API functions.

Step 1: Creating an HDF4 File. To export MATLAB data in HDF4 format, you must first create an HDF4 file, or open an existing one. In the HDF4 SD API, you use the `SDstart` routine. In MATLAB, use the `hdfsd` function, specifying `start` as the first argument. As other arguments, specify

- A text string specifying the name you want to assign to the HDF4 file (or the name of an existing HDF4 file)
- A text string specifying the HDF4 SD interface file access mode

For example, this code creates an HDF4 file named `mydata.hdf`:

```
sd_id = hdfsd('start','mydata.hdf','DFACC_CREATE');
```

When you specify the `DFACC_CREATE` access mode, `SDstart` creates the file and initializes the HDF4 SD multifile interface, returning an HDF4 SD file identifier, named `sd_id` in the example.

If you specify `DFACC_CREATE` mode and the file already exists, `SDstart` fails, returning `-1`. To open an existing HDF4 file, you must use HDF4 read or write modes. For information about using `SDstart` in these modes, see “Step 1: Opening the HDF4 File” on page 7-59.

Step 2: Creating an HDF4 Data Set. After creating the HDF4 file, or opening an existing one, you must create a data set in the file for each MATLAB array you want to export. If you are writing data to an existing data set, you can skip ahead to the next step.

In the HDF4 SD API, you use the `SDcreate` routine to create data sets. In MATLAB, you use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `'create'` in this case
- Valid HDF4 SD file identifier, `sd_id`, returned by `SDstart`
- Name you want assigned to the data set
- Data type of the data set.
- Number of dimensions in the data set. This is called the *rank* of the data set in HDF4 terminology.
- Size of each dimension, specified as a vector

Once you create a data set, you cannot change its name, data type, or dimensions.

For example, to create a data set in which you can write the following MATLAB 3-by-5 array of `doubles`,

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

you could call `hdfsd`, specifying as arguments `'create'` and a valid HDF file identifier, `sd_id`. In addition, set the values of the other arguments as in this code fragment:

```
ds_name = 'A';
ds_type = 'double';
ds_rank = ndims(A);
ds_dims = fliplr(size(A));

sds_id = hdfsd('create',sd_id,ds_name,ds_type,ds_rank,ds_dims);
```

If `SDcreate` can successfully create the data set, it returns an HDF4 SD data set identifier, (`sds_id`). Otherwise, `SDcreate` returns `-1`.

In this example, note the following:

- The data type you specify in `ds_type` must match the data type of the MATLAB array that you want to write to the data set. In the example, the array is of class `double` so the value of `ds_type` is set to `'double'`. If you wanted to use another data type, such as `uint8`, convert the MATLAB array to use this data type,

  ```
  A = uint8([ 1 2 3 4 5 ; 6 7 8 9 10 ; 11 12 13 14 15 ]);
  ```

  and specify the name of the MATLAB data type, `uint8` in this case, in the `ds_type` argument.

  ```
  ds_type = 'uint8';
  ```

- The code fragment reverses the order of the values in the dimensions argument (`ds_dims`). This processing is necessary because the MATLAB `size` function returns the dimensions in column-major order and HDF4 expects to receive dimensions in row-major order.

Step 3: Writing MATLAB Data to an HDF4 File. After creating an HDF4 file and creating a data set in the file, you can write data to the entire data set or just a portion of the data set. In the HDF4 SD API, you use the `SDwritedata` routine. In MATLAB, use the `hdfsd` function, specifying as arguments:

- Name of the SD API routine, `'writedata'` in this case
- Valid HDF4 SD data set identifier, `sds_id`, returned by `SDcreate`
- Location in the data set where you want to start writing data, called the *start* vector in HDF4 terminology
- Number of elements along each dimension to skip between each write operation, called the *stride* vector in HDF4 terminology
- Total number of elements to write along each dimension, called the *edges* vector in HDF4 terminology
- MATLAB array to be written

Note You must specify the values of the start, stride, and edges arguments in row-major order, rather than the column-major order used in MATLAB. Note how the example uses `fliplr` to reverse the order of the dimensions in the vector returned by the `size` function before assigning it as the value of the edges argument.

The values you assign to these arguments depend on the MATLAB array you want to export. For example, the following code fragment writes this MATLAB 3-by-5 array of doubles,

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

into an HDF4 file:

```
ds_start = zeros(1:ndims(A)); % Start at the beginning
ds_stride = [];               % Write every element.
ds_edges = fliplr(size(A));   % Reverse the dimensions.

stat = hdfsd('writedata',sds_id,...
                         ds_start, ds_stride, ds_edges, A);
```

If it can write the data to the data set, `SDwritedata` returns `0`; otherwise, it returns `-1`.

Note `SDwritedata` queues write operations. To ensure that these queued write operations are executed, you must close the file, using the `SDend` routine. See "Step 6: Closing an HDF4 File" on page 7-71 for more information. As a convenience, MATLAB provides a function, `MLcloseall`, that you can use to close all open data sets and file identifiers with a single call. See "Using the MATLAB HDF4 Utility API" on page 7-71 for more information.

To write less than the entire data set, use the start, stride, and edges vectors to specify where you want to start writing data and how much data you want to write.

For example, the following code fragment uses `SDwritedata` to replace the values of the entire second row of the sample data set:

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

with the vector `B`:

```
B = [ 9 9 9 9 9];
```

In the example, the start vector specifies that you want to start the write operation in the first column of the second row. Note how HDF4 uses zero-based indexing and specifies the column dimension first. In MATLAB, you would specify this location as `(2,1)`. The edges argument specifies the dimensions of the data to be written. Note that the size of the array of data to be written must match the edge specification.

```
ds_start = [0 1]; % Start writing at the first column, second row.
ds_stride = []; % Write every element.
ds_edges = [5 1]; % Each row is a 1-by-5 vector.

stat = hdfsd('writedata',sds_id,ds_start,ds_stride,ds_edges,B);
```

Step 4: Writing Metadata to an HDF4 File. You can optionally include information in an HDF4 file, called attributes, that describes the file and its contents. Using the HDF4 SD API, you can associate attributes with three types of HDF4 objects:

- An entire HDF4 file — File attributes, also called *global* attributes, generally contain information pertinent to all the data sets in the file.
- A data set in an HDF4 file — Data set attributes, also called *local* attributes, describe individual data sets.
- A dimension of a data set — Dimension attributes provide information about one particular dimension of a data set.

To create an attribute in the HDF4 SD API, use the `SDsetattr` routine. In MATLAB, use the `hdfsd` function, specifying `'setattr'` as the first argument. As other arguments, specify

- A valid HDF4 SD identifier associated with the object. This value can be a file identifier (`sd_id`), a data set identifier (`sds_id`), or a dimension identifier (`dim_id`).
- A text string that defines the name of the attribute.
- The attribute value.

For example, this code creates a global attribute, named `my_global_attr`, and associates it with the HDF4 file identified by `sd_id`:

```
status = hdfsd('setattr',sd_id,'my_global_attr','my_attr_val');
```

Note In the NCSA documentation, the `SDsetattr` routine has two additional arguments: data type and the number of values in the attribute. When calling this routine from MATLAB, you do not have to include these arguments. The MATLAB HDF4 function can determine the data type and size of the attribute from the value you specify.

The SD interface supports predefined attributes that have reserved names and, in some cases, data types. Predefined attributes are identical to user-defined attributes except that the HDF4 SD API has already defined

their names and data types. For example, the HDF4 SD API defines an attribute, named `cordsys`, in which you can specify the coordinate system used by the data set. Possible values of this attribute include the text strings `'cartesian'`, `'polar'`, and `'spherical'`.

Predefined attributes can be useful because they establish conventions that applications can depend on. The HDF4 SD API supports predefined attributes for data sets and dimensions only; there are no predefined attributes for files. For a complete list of the predefined attributes, see the NCSA documentation.

In the HDF4 SD API, you create predefined attributes the same way you create user-defined attributes, using the `SDsetattr` routine. In MATLAB, use the `hdfsd` function, specifying `setattr` as the first argument:

```
attr_name = 'cordsys';
attr_value = 'polar';

status = hdfsd('setattr',sds_id,attr_name,attr_value);
```

The HDF4 SD API also includes specialized functions for writing and reading the predefined attributes. These specialized functions, such as `SDsetdatastrs`, are sometimes easier to use, especially when you are reading or writing multiple related predefined attributes. You must use specialized functions to read or write the predefined dimension attributes.

You can associate multiple attributes with a single HDF4 object. HDF4 maintains an attribute index for each object. The attribute index is zero-based. The first attribute has index value 0, the second has index value 1, and so on. You access an attribute by its index value.

Each attribute has the format `name=value`, where `name` (called `label` in HDF4 terminology) is a text string up to 256 characters in length and `value` contains one or more entries of the same data type. A single attribute can have multiple values.

Step 5: Closing HDF4 Data Sets. After writing data to a data set in an HDF4 file, you must close access to the data set. In the HDF4 SD API, you use the `SDendaccess` routine to close a data set. In MATLAB, use the `hdfsd` function, specifying `endaccess` as the first argument. As the only other argument, specify a valid HDF4 SD data set identifier, `sds_id` in this example:

```
stat = hdfsd('endaccess',sds_id);
```

Step 6: Closing an HDF4 File. After writing data to a data set and closing the data set, you must also close the HDF4 file. In the HDF4 SD API, you use the `SDend` routine. In MATLAB, use the `hdfsd` function, specifying `end` as the first argument. As the only other argument, specify a valid HDF4 SD file identifier, `sd_id` in this example:

```
stat = hdfsd('end',sd_id);
```

You must close access to all the data sets in an HDF4 file before closing it.

Note Closing an HDF4 file executes all the write operations that have been queued using `SDwritedata`. As a convenience, the MATLAB HDF Utility API provides a function that can close all open data set and file identifiers with a single call. See "Using the MATLAB HDF4 Utility API" on page 7-71 for more information.

Using the MATLAB HDF4 Utility API

In addition to the standard HDF4 APIs, listed in the `hdf`reference page, MATLAB supports utility functions that are designed to make it easier to use HDF4 in the MATLAB environment.

For example, using the gateway function to the MATLAB HDF4 utility API, `hdfml`, and specifying the name of the `listinfo` routine as an argument, you can view all the currently open HDF4 identifiers. MATLAB updates this list whenever HDF identifiers are created or closed. In the following example only two identifiers are open.

```
hdfml('listinfo')
No open RI identifiers
No open GR identifiers
No open grid identifiers
No open grid file identifiers
No open annotation identifiers
No open AN identifiers
Open scientific dataset identifiers:
 262144
```

```
Open scientific data file identifiers:
 393216
No open Vdata identifiers
No open Vgroup identifiers
No open Vfile identifiers
No open point identifiers
No open point file identifiers
No open swath identifiers
No open swath file identifiers
No open access identifiers
No open file identifiers
```

Closing All Open HDF4 Identifiers. To close all the currently open HDF4 identifiers in a single call, use the gateway function to the MATLAB HDF4 utility API, `hdfml`, specifying the name of the `closeall` routine as an argument. The following example closes all the currently open HDF4 identifiers.

```
hdfml('closeall')
```

8

Error Handling

Error Reporting in MATLAB

In this section...

Overview

No matter how carefully you plan and test the programs you write, they may not always run as smoothly as expected when executed under different conditions. It is always a good idea to include error checking in programs to ensure reliable operation under all conditions.

In MATLAB, you can decide how your programs respond to different types of errors. You may want to prompt the user for more input, display extended error or warning information, or perhaps repeat a calculation using default values. The error-handling capabilities in MATLAB help your programs check for particular error conditions and execute the appropriate code depending on the situation.

When MATLAB detects a severe fault in the command or program it is running, it collects information about what was happening at the time of the error, displays a message to help the user understand what went wrong, and terminates the command or program. This is called *throwing an exception*. You can get an exception while entering commands at the MATLAB command prompt or while executing your program code.

Getting an Exception at the Command Line

If you get an exception at the MATLAB prompt, you have several options on how to deal with it as described below.

Determine the Fault from the Error Message

Evaluate the error message MATLAB has displayed. Most error messages attempt to explain at least the immediate cause of the program failure. There

is often sufficient information to determine the cause and what you need to do to remedy the situation.

Review the Failing Code

If the function in which the error occurred is implemented as an M-file, the error message should include a line that looks something like this:

```
surf
```

```
??? Error using ==> surf at 50
Not enough input arguments.
```

The underlined text to the right names the function that threw the error (`surf`, in this case) and shows the failing line number within that function's M-file. Click the underlined text; MATLAB opens the M-file and positions the cursor at the location in the file where the error originated. You may be able to determine the cause of the error by examining this line and the code that precedes it.

Step Through the Code in the Debugger

You can use the MATLAB Debugger to step through the failing code. Click the underlined error text to open the M-file in the MATLAB Editor at or near the point of the error. Next, click the hyphen at the beginning of that line to set a breakpoint at that location. When you rerun your program, MATLAB pauses execution at the breakpoint and enables you to step through the program code. The command `dbstop on error` is also helpful in finding the point of error.

See the documentation on "Editing and Debugging M-Files" for more information.

Getting an Exception in Your Program Code

When you are writing your own program in an M-file, you can *catch* exceptions and attempt to handle or resolve them instead of allowing your program to terminate. When you catch an exception, you interrupt the normal termination process and enter a block of code that deals with the faulty situation. This block of code is called a *catch block*.

Some of the things you might want to do in the catch block are:

- Examine information that has been captured about the error.
- Gather further information to report to the user.
- Try to accomplish the task at hand in some other way.
- Clean up any unwanted side effects of the error.

When you reach the end of the catch block, you can either continue executing the program, if possible, or terminate it.

The documentation on “Capturing Information About the Error” on page 8-5 describes how to acquire information about what caused the error, and “Responding to an Exception” on page 8-17 presents some ideas on how to respond to it.

Generating a New Exception

When your program code detects a condition that will either make the program fail or yield unacceptable results, it should throw an exception. This procedure

- Saves information about what went wrong and what code was executing at the time of the error.
- Gathers any other pertinent information about the error.
- Instructs MATLAB to throw the exception.

The documentation on “Capturing Information About the Error” on page 8-5 describes how to use an `MException` object to capture information about the error, and “Throwing an Exception” on page 8-16 explains how to initiate the exception process.

Capturing Information About the Error

In this section...

Overview

When MATLAB throws an exception, it captures information about what caused the error in a data structure called an `MException` object. This object is an instance of the MATLAB `MException` class. You can obtain access to the `MException` object by *catching* the exception before your program aborts and accessing the object constructed for this particular error via the `catch` command. When throwing an exception in response to an error in your own M-file code, you will have to create a new `MException` object and store information about the error in that object.

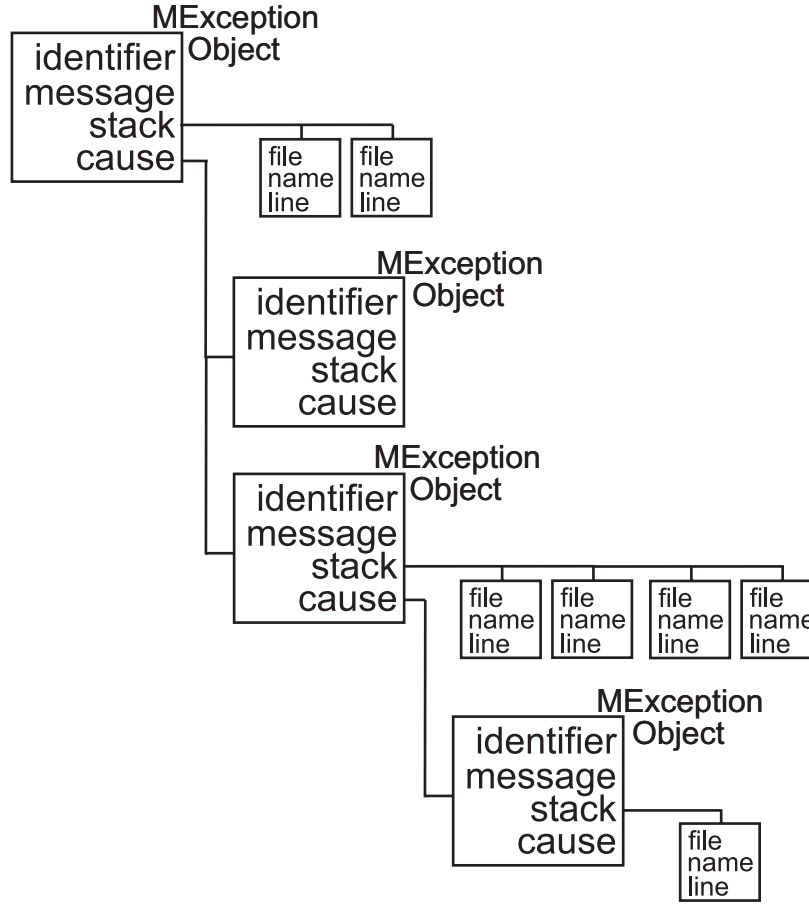
This section describes the `MException` class and objects constructed from that class:

Information on how to use this class is presented in later sections on "Responding to an Exception" on page 8-17 and "Throwing an Exception" on page 8-16.

The MException Class

The figure shown below illustrates one possible configuration of an object of the `MException` class. The object has four properties: `identifier`, `message`, `stack`, and `cause`. Each of these properties is implemented as a field of the structure that represents the `MException` object. The `stack` field is an N-by-1 array of additional structures, each one identifying an M-file, function, and line number from the call stack. The `cause` field is an M-by-1 cell array of `MException` objects, each representing an exception that is related to the current one.

See “Properties of the MException Class” on page 8-7 for a full description of these properties.



Object Constructor

Any code that detects an error and throws an exception must also construct an `MException` object in which to record and transfer information about the error. The syntax of the `MException` constructor is

```
ME = MException(identifier, message)
```

where `identifier` is a MATLAB message identifier of the form

```
component:mnemonic
```

that is enclosed in single quotes, and `message` is a text string, also enclosed in single quotes, that describes the error. The output `ME` is the resulting `MException` object.

If you are responding to an exception rather than throwing one, you do not have to construct an `MException` object. The object has already been constructed and populated by the code that originally detected the error.

Properties of the MException Class

The `MException` class has four properties. Each of these properties is implemented as a field of the structure that represents the `MException` object. Each of these properties is described in the sections below and referenced in the sections on "Responding to an Exception" on page 8-17 and "Throwing an Exception" on page 8-16:

- `identifier`
- `message`
- `stack`
- `cause`

Repeating the `surf` example shown above, but this time catching the exception, you can see the four properties of the `MException` object structure. (This example uses `try-catch` in an atypical fashion. See the section on "The try-catch Statement" on page 8-17 for more information on using `try-catch`).

```
try
    surf
catch ME
    ME
end
```

Run this at the command line and MATLAB returns the contents of the `MException` object:

```
ME =
 MException object with properties:

    identifier: 'MATLAB:nargchk:notEnoughInputs'
       message: 'Not enough input arguments.'
         stack: [1x1 struct]
         cause: {}
```

The `stack` field shows the filename, function, and line number where the exception was thrown:

```
ME.stack
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

The `cause` field is empty in this case. Each field is described in more detail in the sections that follow.

Message Identifiers

A message identifier is a tag that you attach to an error or warning statement that makes that error or warning uniquely recognizable by MATLAB. You can use message identifiers with error reporting to better identify the source of an error, or with warnings to control any selected subset of the warnings in your programs.

The message identifier is a read-only character string that specifies a *component* and a *mnemonic* label for an error or warning. The format of a simple identifier is

```
component:mnemonic
```

A colon separates the two parts of the identifier: `component` and `mnemonic`. If the identifier uses more than one `mnemonic`, then additional colons are required to separate them. A message identifier must always contain at least one colon.

Some examples of message identifiers are

```
MATLAB:divideByZero
Simulink:actionNotTaken
TechCorp:OpenFile:notFoundInPath
```

Both the `component` and `mnemonic` fields must adhere to the following syntax rules:

- No white space (space or tab characters) is allowed anywhere in the identifier.
- The first character must be alphabetic, either uppercase or lowercase.
- The remaining characters can be alphanumeric or an underscore.

There is no length limitation to either the `component` or `mnemonic`. The identifier can also be an empty string.

Component Field. The `component` field specifies a broad category under which various errors and warnings can be generated. Common components are a particular product or toolbox name, such as `MATLAB` or `Control`, or perhaps the name of your company, such as `TechCorp` in the preceding example.

You can also use this field to specify a multilevel component. The following statement has a three-level component followed by a mnemonic label:

```
TechCorp:TestEquipDiv:Waveform:obsoleteSyntax
```

The component field enables you to guarantee the uniqueness of each identifier. Thus, while MATLAB uses the identifier `MATLAB:divideByZero` for its `'Divide by zero'` warning, you can reuse the `divideByZero` mnemonic by using your own unique component. For example,

```
warning('TechCorp:divideByZero', ...
        'A sprocket value was divided by zero.')
```

Mnemonic Field. The `mnemonic` field is a string normally used as a tag relating to the particular message. For example, when reporting an error resulting from the use of ambiguous syntax, a simple component and mnemonic such as the following might be appropriate:

```
MATLAB:ambiguousSyntax
```

Message Identifiers in an MException Object. When throwing an exception, create an appropriate identifier and save it to the `MException` object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

For example,

```
ME = MException('AcctError:Incomplete', ...
      'Client name not recognized.');

ME.identifier
ans =
    AcctError:NoClient
```

When responding to an exception, you can extract the message identifier from the `MException` object as shown here:

```
try
    surf
catch ME
    id = ME.identifier
end

id =
    MATLAB:nargchk:notEnoughInputs
```

Text of the Error Message

An error message in MATLAB is a read-only character string issued by the program code and returned in the `MException` object. This message can assist the user in determining the cause, and possibly the remedy, of the failure.

When throwing an exception, compose an appropriate error message and save it to the `MException` object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

If your message string requires formatting specifications, like those available with the `sprintf` function, use this syntax for the `MException` constructor:

```
ME = MException(identifier, formatstring, arg1, arg2, ...)
```

For example,

```
S = 'Accounts';  f1 = 'ClientName';
ME = MException('AcctError:Incomplete', ...
      'Field ''%s.%s'' is not defined.', S, f1);

ME.message
ans =
    Field 'Accounts.ClientName' is not defined.
```

When responding to an exception, you can extract the error message from the `MException` object as follows:

```
try
    surf
catch ME
    msg = ME.message
end

msg =
    Not enough input arguments.
```

The Call Stack

The `stack` field of the `MException` object identifies the line number, function, and filename where the error was detected. If the error occurs in a called function, as in the following example, the `stack` field contains the line number, function name, and filename not only for the location of the immediate error, but also for each of the calling functions. In this case, `stack` is an N-by-1 array, where N represents the depth of the call stack. That is, the stack field displays the M-file function name and line number where the exception occurred, the name and line number of the M-file caller, the caller's caller, etc., until the top-most M-file function is reached.

When throwing an exception, MATLAB stores call stack information in the `stack` field. You cannot write to this field; access is read-only.

For example, suppose you have three functions that reside in two separate M-files:

```
 mfileA.m
=========================
        .
        .
42 function A1(x, y)
43 B1(x, y);



 mfileB.m
=========================
        .
        .
 8 function B1(x, y)
 9 B2(x, y)
        .
        .
26 function B2(x, y)
27      .
28      .
29      .
30      .
31 %  Throw exception here
```

Catch the exception in variable `ME` and then examine the `stack` field:

```
for k=1:length(ME.stack)
    ME.stack(k)
end

ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B2'
    line: 31
```

```
ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B1'
    line: 9
ans =
    file: 'C:\matlab\test\mfileA.m'
    name: 'A1'
    line: 43
```

The Cause Array

In some situations, it can be important to record information about not only the one command that caused execution to stop, but also other exceptions that your code caught. You can save these additional `MException` objects in the `cause` field of the primary exception.

The `cause` field of an `MException` is an optional cell array of related `MException` objects. You must use the following syntax when adding objects to the `cause` cell array:

```
primaryException = addCause(primaryException, secondaryException)
```

This example attempts to assign an array `D` to variable `X`. If the `D` array does not exist, the code attempts to load it from a MAT-file and then retries assigning it to `X`. If the load fails, a new `MException` object (`ME3`) is constructed to store the cause of the first two errors (`ME1` and `ME2`):

```
try
    X = D(1:25)
catch ME1
    try
        filename = 'test200';
        load(filename);
        X = D(1:25)
    catch ME2
        ME3 = MException('MATLAB:LoadErr', ...
               'Unable to load from file %s', filename);
        ME3 = addCause(ME3, ME1);
        ME3 = addCause(ME3, ME2);
    end
end
```

There are two exceptions in the cause field of ME3:

```
ME3.cause
ans =
    [1x1 MException]
    [1x1 MException]
```

Examine the cause field of ME3 to see the related errors:

```
ME3.cause{:}
ans =

 MException object with properties:

    identifier: 'MATLAB:UndefinedFunction'
       message: 'Undefined function or method 'D' for input
arguments of type 'double'.'
         stack: [0x1 struct]
         cause: {}
ans =

 MException object with properties:

    identifier: 'MATLAB:load:couldNotReadFile'
       message: 'Unable to read file test204: No such file or
directory.'
         stack: [0x1 struct]
         cause: {}
```

Methods of the MException Class

There are ten methods that you can use with the MException class. The names of these methods are case-sensitive. See the MATLAB function reference pages for more information.

Method Name	Description
addCause	Append an MException to the cause field of another MException.
disp	Display an MException object.

Method Name	Description
`eq`	Compare `MException` objects for equality.
`getReport`	Return a formatted message based on the current exception.
`isequal`	Compare `MException` objects for equality.
`last`	Return the last uncaught exception. This is a static method.
`ne`	Compare `MException` objects for inequality.
`rethrow`	Reissue an exception that has previously been caught.
`throw`	Issue an exception.
`throwAsCaller`	Issue an exception, but omit the current stack frame from the `stack` field.

Throwing an Exception

When your program detects a fault that will keep it from completing as expected or will generate erroneous results, you should halt further execution and report the error by throwing an exception. The basic steps to take are

- Detect the error. This is often done with some type of conditional statement, such as an `if` statement that checks the output of the current operation.
- Construct an `MException` object to represent the error. Add a message identifier string and error message string to the object when calling the constructor.
- If there are other exceptions that may have contributed to the current error, you can store the `MException` object for each in the `cause` field of a single `MException` that you intend to throw. Use the `addCause` method for this.
- Use the `throw` or `throwAsCaller` function to have MATLAB issue the exception. At this point, MATLAB stores call stack information in the `stack` field of the `MException`, exits the currently running function, and returns control to either the keyboard or an enclosing catch block in a calling function.

Responding to an Exception

In this section...

Overview

As stated earlier, MATLAB by default, terminates the currently running program when an exception is thrown. If you catch the exception in your program, however, you can capture information about what went wrong, and deal with the situation in a way that is appropriate for the particular condition. This requires a `try-catch` statement.

This section covers the following topics:

The try-catch Statement

When you have statements in your code that could generate undesirable results, put those statements into a `try-catch` block that catches any errors and handles them appropriately.

A `try-catch` statement looks something like the following pseudocode. It consists of two parts:

- A `try` block that includes all lines between the `try` and `catch` statements.
- A `catch` block that includes all lines of code between the `catch` and `end` statements.

```
    try
       Perform one ...
          or more operations
A   catch ME
       Examine error info in exception object ME
       Attempt to figure out what went wrong
       Either attempt to recover, or clean up and abort
```

```
    end

B   Program continues
```

The program executes the statements in the `try` block. If it encounters an error, it skips any remaining statements in the `try` block and jumps to the start of the `catch` block (shown here as point `A`). If all operations in the `try` block succeed, then execution skips the `catch` block entirely and goes to the first line following the `end` statement (point `B`).

Specifying the `try`, `catch`, and `end` commands and also the code of the `try` and `catch` blocks on separate lines is recommended. If you combine any of these components on the same line, separate them with commas:

```
try, surf, catch ME, ME.stack, end
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

The Try Block

On execution, your code enters the `try` block and executes each statement as if it were part of the regular program. If no errors are encountered, MATLAB skips the `catch` block entirely and continues execution following the `end` statement. If any of the `try` statements fail, MATLAB immediately exits the `try` block, leaving any remaining statements in that block unexecuted, and enters the `catch` block.

The Catch Block

The `catch` command marks the start of a `catch` block and provides access to a data structure that contains information about what caused the exception. This is shown as the variable `ME` in the preceding pseudocode. This data structure is an object of the MATLAB `MException` class. When an exception occurs, MATLAB constructs an instance of this class and returns it in the `catch` statement that handles that error.

You are not required to specify any argument with the `catch` statement. If you do not need any of the information or methods provided by the `MException` object, just specify the `catch` keyword alone.

The `MException` object is constructed by internal code in the program that fails. The object has properties that contain information about the error that can be useful in determining what happened and how to proceed. The `MException` object also provides access to methods that enable you to respond to the exception. See the section on"The MException Class" on page 8-5 to find out more about the `MException` class.

Having entered the `catch` block, MATLAB executes the statements in sequence. These statements can attempt to

- Attempt to resolve the error.
- Capture more information about the error.
- Switch on information found in the `MException` object and respond appropriately.
- Clean up the environment that was left by the failing code.

The `catch` block often ends with a `rethrow` command. The `rethrow` causes MATLAB to exit the current function, keeping the call stack information as it was when the exception was first thrown. If this function is at the highest level, that is, it was not called by another function, the program terminates. If the failing function was called by another function, it returns to that function. Program execution continues to return to higher level functions, unless any of these calls were made within a higher-level `try` block, in which case the program executes the respective catch block.

More information about the `MException` class is provided in the section "Capturing Information About the Error" on page 8-5.

Suggestions on How to Handle an Exception

The following example reads the contents of an image file. The `try` block attempts to open and read the file. If either the open or read fails, the program catches the resulting exception and saves the `MException` object in the variable `ME1`.

The `catch` block in the example checks to see if the specified file could not be found. If so, the program allows for the possibility that a common variation of the filename extension (e.g., `jpeg` instead of `jpg`) was used by retrying

the operation with a modified extension. This is done using a `try-catch` statement nested within the original `try-catch`.

```
function d_in = read_image(filename)
file_format = regexp(filename, '(?<=\.)\w+$', 'match');

try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch ME1
   % Get last segment of the error message identifier.
   idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', 'match');

   % Did the read fail because the file could not be found?
   if strcmp(idSegLast, 'InvalidFid') && ...
      ~exist(filename, 'file')

      % Yes. Try modifying the filename extension.
      switch file_format
      case 'jpg'    % Change jpg to jpeg
         filename = regexprep(filename, '(?<=\.)\w+$', 'jpeg');
      case 'jpeg'   % Change jpeg to jpg
         filename = regexprep(filename, '(?<=\.)\w+$', 'jpg');
      case 'tif'    % Change tif to tiff
         filename = regexprep(filename, '(?<=\.)\w+$', 'tiff');
      case 'tiff'   % Change tiff to tif
         filename = regexprep(filename, '(?<=\.)\w+$', 'tif');
      otherwise
         rethrow(ME1);
      end

      % Try again, with modifed filenames.
      try
         fid = fopen(filename, 'r');
         d_in = fread(fid);
      catch ME2
         ME2 = addCause(ME2, ME1);
         rethrow(ME2)
      end
   end
```

```
end
```

This example illustrates some of the actions that you can take in response to an exception:

- Compare the `identifier` field of the `MException` object against possible causes of the error.
- Use a nested `try-catch` statement to retry the open and read operations using a known variation of the filename extension.
- Display an appropriate message in the case that the file truly does not exist and then rethrow the exception.
- Add the first `MException` object to the `cause` field of the second.
- Rethrow the exception. This stops program execution and displays the error message.

Cleaning up any unwanted results of the error is also advisable. For example, your program may have allocated a significant amount of memory that it no longer needs.

Warnings

In this section...

Reporting a Warning

Like `error`, the `warning` function alerts the user of unexpected conditions detected when running a program. However, `warning` does not halt the execution of the program. It displays the specified warning message and then continues.

Use `warning` in your code to generate a warning message during execution. Specify the message string as the input argument to `warning`. For example,

```
warning('Input must be a string')
```

Warnings also differ from errors in that you can disable any warnings that you do not want to see. You do this by invoking `warning` with certain control parameters. See “Warning Control” on page 8-24 for more information.

Formatted Message Strings

The warning message string you specify can contain formatting conversion characters, such as those used with the MATLAB `sprintf` function. Make the warning string the first argument, and add any variables used by the conversion as subsequent arguments.

```
warning('formatted_warningmsg', arg1, arg2, ...)
```

For example, if your program cannot process a given parameter, you might report a warning with

```
warning('Ambiguous parameter name, "%s".', param)
```

MATLAB converts special characters like `%d` and `%s` in the warning message string only when you specify more than one input argument with `warning`. See “Formatted Message Strings” on page 8-22 for information.

Message Identifiers

Use a message identifier argument with `warning` to attach a unique tag to a warning message. MATLAB uses this tag to better identify the source of a warning. The first argument in this example is the message identifier.

```
warning('MATLAB:paramAmbiguous', ...
        'Ambiguous parameter name, "%s".', param)
```

See “Warning Control Statements” on page 8-26 for more information on how to use identifiers with warnings.

Identifying the Cause

The `lastwarn` function returns a string containing the last warning message issued by MATLAB. Use this to enable your program to identify the cause of a warning that has just been issued. To return the most recent warning message to the variable `warnmsg`, type

```
warnmsg = lastwarn;
```

You can also change the text of the last warning message with a new message or with an empty string as shown here:

```
lastwarn('newwarnmsg');   % Replace last warning with new string
lastwarn('');           % Replace last warning with empty string
```

Warning Control

In this section...

Overview

MATLAB gives you the ability to control what happens when a warning is encountered during M-file program execution. Options that are available include

- Display selected warnings.
- Ignore selected warnings.
- Stop in the debugger when a warning is invoked.
- Display an M-stack trace after a warning is invoked.

Depending on how you set your warning controls, you can have these actions affect all warnings in your code, specific warnings that you select, or just the most recently invoked warning.

Setting up this system of warning control involves several steps.

1. Start by determining the scope of the control you need for the warnings generated by your code. Do you want the control operations to affect all the warnings in your code at once, or do you want to be able to control certain warnings separately?

2. If the latter is true, you will need to identify those warnings you want to selectively control. This requires going through your code and attaching unique *message identifiers* to each of those warnings. If, on the other

hand, you do not require that fine a granularity of control, the warning statements in your code need no message identifiers.

3 When you are ready to run your programs, use the MATLAB warning control statements to exercise the desired controls on all or selected warnings. Include message identifiers in these control statements when selecting specific warnings to act upon.

Warning Statements

The warning statements you put into your M-file code must contain the string to be displayed when the warning is incurred, and may also contain a message identifier. If you are not planning to use warning control or if you do not need to single out certain warnings for control, you need to specify only the message string. Use the syntax shown in "Warnings" on page 8-22. Valid formats are

```
warning('warnmsg')
warning('formatted_warnmsg', arg1, arg2, ...)
```

Attaching an Identifier to the Warning Statement

If you want to be able to apply control statements to specific warnings, you need to include a message identifier in the warning statements you wish to control. The message identifier must be the first argument in the statement. Valid formats are

```
warning('msg_id', 'warnmsg')
warning('msg_id', 'formatted_warnmsg', arg1, arg2, ...)
```

See "Message Identifiers" on page 8-8 for information on how to specify the `msg_id` argument.

Note When you specify more than one input argument with `warning`, MATLAB treats the `warnmsg` string as if it were a `formatted_warnmsg`. This is explained in "Formatted Message Strings" on page 8-22.

Warning Control Statements

Once you have the warning statements in your M-file and are ready to execute it, you tell MATLAB how to act on these warnings by issuing control statements. These statements place the specified warning(s) into a desired state and have the format

```
warning state msg_id
```

Control statements can return information on the state of selected warnings if you assign the output to a variable, as shown below. See “Output from Control Statements” on page 8-28.

```
s = warning('state', 'msg_id');
```

Warning States

There are three possible values for the `state` argument of a warning control statement.

State	Description
`on`	Enable the display of selected warning message.
`off`	Disable the display of selected warning message.
`query`	Display the current state of selected warning.

Message Identifiers

In addition to the message identifiers already discussed, there are three other identifiers that you can use in control statements only.

Identifier	Description
`msg_id string`	Set selected warning to the specified state.
`all`	Set all warnings to the specified state.
`last`	Set only the last displayed warning to the specified state.

Note MATLAB starts up with all warnings enabled, except for those displayed in response to the command, `warning('query', 'all')`.

Example 1 — Enabling a Selected Warning

Enable just the `actionNotTaken` warning from Simulink by first turning off all warnings and then setting just that warning to `on`.

```
warning off all
warning on Simulink:actionNotTaken
```

Next, use `query` to determine the current state of all warnings. It reports that you have set all warnings to `off`, with the exception of `Simulink:actionNotTaken`.

```
warning query all
The default warning state is 'off'. Warnings not set to the
default are

  State  Warning Identifier

     on  Simulink:actionNotTaken
```

Example 2 — Disabling the Most Recent Warning

Evaluating `inv` on zero displays a warning message. Turn off the most recently invoked warning with `warning off last`.

```
inv(0)
Warning: Matrix is singular to working precision.
ans =
   Inf

warning off last

inv(0)              % No warning is displayed this time
ans =
   Inf
```

Output from Control Statements

The `warning` function, when used in a control statement, returns a MATLAB structure array containing the previous state of the selected warning(s). Use the following syntax to return this information in structure array `s`:

```
s = warning('state', 'msg_id');
```

You must type the command using the MATLAB function format; parentheses and quotation marks are required.

Note MATLAB does not display `warning` output if you do not assign the output to a variable.

The next example turns off `divideByZero` warnings for the MATLAB component, and returns the `identifier` and previous `state` in a 1-by-1 structure array.

```
s = warning('off','MATLAB:divideByZero')
s =
    identifier: 'MATLAB:divideByZero'
         state: 'on'
```

You can use output variables with any type of warning control statement. If you just want to collect the information but do not want to change state, simply perform a `query` on the warning(s). MATLAB returns the current state of those warnings selected by the message identifier.

```
s = warning('query', 'msg_id');
```

If you want to change state, but save the former state so you can restore it later, use the return structure array to save that state. The following example does an implicit `query`, returning state information in `s`, and then turns on all warnings.

```
s = warning('on', 'all');
```

See “Saving and Restoring State” on page 8-30, for more information on restoring the former state of warnings.

Output Structure Array

Each element of the structure array returned by `warning` contains two fields.

Field Name	Description
`identifier`	Message identifier string, `'all'`, or `'last'`
`state`	State of warning(s) prior to invoking this control statement

If you query for the state of just one warning, using a message identifier or `'last'` in the command, MATLAB returns a one-element structure array. The `identifier` field contains the selected message identifier, and the `state` field holds the current state of that warning:

```
s = warning('query','last')
s =
   identifier: 'MATLAB:divideByZero'
        state: 'on'
```

If you query for the state of all warnings, using `'all'` in the command, MATLAB returns a structure array having one or more elements:

- The first element of the array always represents the default state. (This is the state set by the last `warning on|off all` command.)
- Each other element of the array represents a warning that is in a state different from the default.

```
warning off all
warning on MATLAB:divideByZero
warning on MATLAB:fileNotFound

s = warning('query', 'all')
s =
   3x1 struct array with fields:
       identifier
       state

s(1)
ans =
```

```
   identifier: 'all'
        state: 'off'

s(2)
ans =
   identifier: 'MATLAB:divideByZero'
        state: 'on'

s(3)
ans =
   identifier: 'MATLAB:fileNotFound'
        state: 'on'
```

Saving and Restoring State

To temporarily change the state of some warnings and then later return to your original settings, save the original state in a structure array and then restore it from that array. You can save and restore the state of all of your warnings or just one that you select with a message identifier.

To save the current warning state, assign the output of a warning control statement, as discussed in “Output from Control Statements” on page 8-28. The following statement saves the current state of all warnings in structure array `s`:

```
s = warning('query', 'all');
```

To restore state from `s`, use the syntax shown below. Note that the MATLAB function format (enclosing arguments in parentheses) is required.

```
warning(s)
```

Example 1 — Performing an Explicit Query

Perform a query of all warnings to save the current state in structure array `s`:

```
s = warning('query', 'all');
```

Then, after doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:

```
warning(s)
```

Example 2 — Performing an Implicit Query

Turn on one particular warning, saving the previous state of this warning in `s`. Remember that this nonquery syntax (where `state` equals `on` or `off`) performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

Restore the state of that one warning when you are ready, with

```
warning(s)
```

Backtrace and Verbose Modes

In addition to warning messages, there are two *modes* that can be enabled or disabled with a warning control statement. These modes are shown here.

Mode	Description	Default
`backtrace`	Display an M-stack trace after a warning is invoked.	`on (enabled)`
`verbose`	Display a message on how to suppress the warning.	`off (terse)`

The syntax for this type of control statement is as follows, where `state`, in this case, can be only `on`, `off`, or `query`:

```
warning state mode
```

Note that there is no need to include a message identifier with this type of control statement. All enabled warnings are affected by the this type of control statement.

Note You cannot save and restore the current state of the `backtrace` or `verbose` modes as you can with other states.

Example 1 — Displaying a Stack Trace on a Specific Warning

It can be difficult to locate the source of a warning when it is generated from code buried in several levels of function calls. This example generates

a warning within a function that is nested several levels deep within the primary function in file `f1.m`:

```
function f1(a, b)
   for k = a:-1:b
       f2(k)
   end
   function f2(x)
       f3(x-1)
       function f3(y)
           x = log(y);
       end
   end
end
```

After enabling all warnings, run the M-file. The code generates a `Log of zero` warning. In an M-file of this size, it is not difficult to find the cause of the warning, but in an M-file of several hundred lines, this could take some time:

```
warning on all

f1(50,1)
Warning: Log of zero.
```

To simplify the debug process, enable backtrace mode. In this mode, MATLAB reports which function generated the warning (`f3`), the line number of the attempted operation (line `8`), the sequence of function calls that led up to the execution of the function (`f1>f2/f3`), and the line at which each of these function call was made (`3` and `6`):

```
warning on backtrace
f1(50,1)
Warning: Log of zero.
> In f1>f2/f3 at 8
  In f1>f2 at 6
  In f1 at 3
```

Example 2 — Enabling Verbose Warnings

When you enable verbose warnings, MATLAB displays an extra line of information with each warning that tells you how to suppress it:

Turn on all warnings, disable backtrace (if you have just run the previous example), and enable verbose warnings:

```
warning on all
warning off backtrace
warning on verbose
```

Call the function described in Example 1 to find out how to suppress any warnings generated by that function:

```
f1(50,1)
Warning: Log of zero.
(Type "warning off MATLAB:log:logOfZero" to suppress this warning.)
```

Use the message identifier `MATLAB:log:logOfZero` to disable only this warning, and run the function again. This time the warning message is not displayed:

```
warning off MATLAB:log:logOfZero

f1(50,1)
```

Debugging Errors and Warnings

You can direct MATLAB to temporarily stop the execution of an M-file in the event of a run-time error or warning, at the same time opening a debug window paused at the M-file line that generated the error or warning. This enables you to examine values internal to the program and determine the cause of the error.

Use the `dbstop` function to have MATLAB stop execution and enter debug mode when any M-file you subsequently run produces a run-time error or warning. There are three types of such breakpoints that you can set.

Command	Description
`dbstop if all error`	Stop on any error.
`dbstop if error`	Stop on any error not detected within a `try-catch` block.
`dbstop if warning`	Stop on any warning.

In all three cases, the M-file you are trying to debug must be in a directory that is on the search path or in the current directory.

You cannot resume execution after an error; use `dbquit` to exit from the Debugger. To resume execution after a warning, use `dbcont` or `dbstep`.

9

Classes and Objects

Classes and Objects: An Overview

In this section...

Overview

You can view classes as new data types having specific behaviors defined for the class. For example, a polynomial class might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials. Operations defined to work with objects of a particular class are known as *methods* of that class.

You can also view classes as new items that you can treat as single entities. An example is an arrow object that MATLAB can display on graphs (perhaps composed of MATLAB line and patch objects) and that has properties like a Handle Graphics object. You can create an arrow simply by instantiating the arrow class.

You can add classes to your MATLAB environment by specifying a MATLAB structure that provides data storage for the object and creating a class directory containing M-files that operate on the object. These M-files contain the methods for the class. The class directory can also include functions that define the way various MATLAB operators, including arithmetic operations,

subscript referencing, and concatenation, apply to the objects. Redefining how a built-in operator works for your class is known as *overloading* the operator.

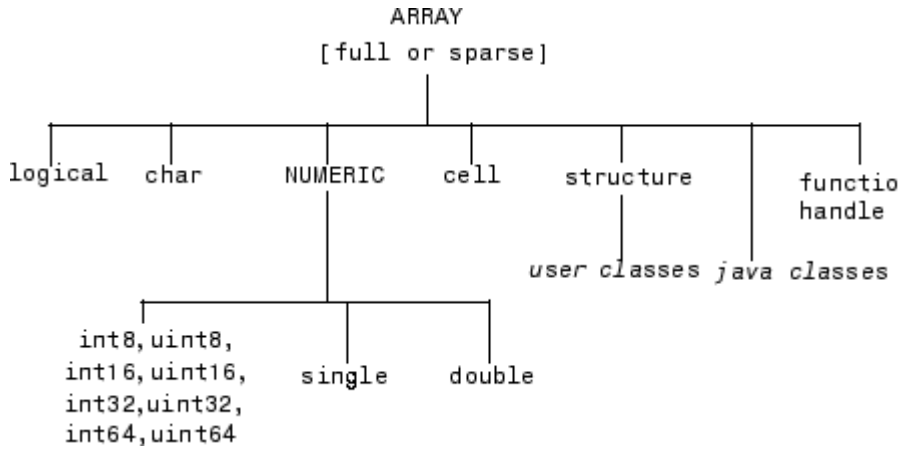
Features of Object-Oriented Programming

When using well-designed classes, object-oriented programming can significantly increase code reuse and make your programs easier to maintain and extend. Programming with classes and objects differs from ordinary structured programming in these important ways:

- **Function and operator overloading**. You can create methods that override existing MATLAB functions. When you call a function with a user-defined object as an argument, MATLAB first checks to see if there is a method defined for the object's class. If there is, MATLAB calls it, rather than the normal MATLAB function.
- **Encapsulation of data and methods**. Object properties are not visible from the command line; you can access them only with class methods. This protects the object properties from operations that are not intended for the object's class.
- **Inheritance**. You can create class hierarchies of parent and child classes in which the child class inherits data fields and methods from the parent. A child class can inherit from one parent (*single inheritance*) or many parents (*multiple inheritance*). Inheritance can span one or more generations. Inheritance enables sharing common parent functions and enforcing common behavior amongst all child classes.
- **Aggregation**. You can create classes using *aggregation*, in which an object contains other objects. This is appropriate when an object type is part of another object type. For example, a savings account object might be a part of a financial portfolio object.

MATLAB Data Class Hierarchy

All MATLAB data types are designed to function as classes in object-oriented programming. The diagram below shows the fifteen fundamental data types (or classes) defined in MATLAB. You can add new data types to MATLAB by extending the class hierarchy.



The diagram shows a *user class* that inherits from the structure class. All classes that you create are structure based since this is the point in the class hierarchy where you can insert your own classes. (For more information about MATLAB data types, see Chapter 2, “Data Types”)

Creating Objects

You create an object by calling the class constructor and passing it the appropriate input arguments. In MATLAB, constructors have the same name as the class name. For example, the statement,

```
p = polynom([1 0 -2 -5]);
```

creates an object named `p` belonging to the class `polynom`. Once you have created a polynom object, you can operate on the object using methods that are defined for the `polynom` class. See “Example — A Polynomial Class” on page 9-26 for a description of the `polynom` class.

Invoking Methods on Objects

Class methods are M-file functions that take an object as one of the input arguments. The methods for a specific class must be placed in the class directory for that class (the `@classname` directory). This is the first place that MATLAB looks to find a class method.

The syntax for invoking a method on an object is similar to a function call. Generally, it looks like

```
[out1,out2,...] = methodName(object,arg1,arg2, ...);
```

For example, suppose a user-defined class called `polynom` has a `char` method defined for the class. This method converts a polynom object to a character string and returns the string. This statement calls the `char` method on the polynom object `p`.

```
s = char(p);
```

Using the `class` function, you can confirm that the returned value `s` is a character string.

```
class(s)
ans =
    char

s
s =
    x^3-2*x-5
```

You can use the `methods` command to produce a list of all of the methods that are defined for a class.

Private Methods

Private methods can be called only by other methods of their class. You define private methods by placing the associated M-files in a `private` subdirectory of the `@`*`classname`* directory. In the example,

```
@classname/private/updateObj.m
```

the method `updateObj` has scope only within the *`classname`* class. This means that `updateObj` can be called by any method that is defined in the `@`*`classname`* directory, but it cannot be called from the MATLAB command line or by methods outside of the class directory, including parent methods.

Private methods and private functions differ in that private methods (in fact all methods) have an object as one of their input arguments and private

functions do not. You can use private functions as helper functions, such as described in the next section.

Helper Functions

In designing a class, you may discover the need for functions that perform support tasks for the class, but do not directly operate on an object. These functions are called *helper functions*. A helper function can be a subfunction in a class method file or a private function. When determining which version of a particular function to call, MATLAB looks for these functions in the order listed above. For more information about the order in which MATLAB calls functions and methods, see "How MATLAB Determines Which Method to Call" on page 9-72.

Debugging Class Methods

You can use the MATLAB debugging commands with object methods in the same way that you use them with other M-files. The only difference is that you need to include the class directory name before the method name in the command call, as shown in this example using `dbstop`.

```
dbstop @polynom/char
```

While debugging a class method, you have access to all methods defined for the class, including inherited methods, private methods, and private functions.

Changing Class Definition

If you change the class definition, such as the number or names of fields in a class, you must issue a

```
clear classes
```

command to propagate the changes to your MATLAB session. This command also clears all objects from the workspace. See the `clear` command help entry for more information.

Setting Up Class Directories

The M-files defining the methods for a class are collected together in a directory referred to as the class directory. The directory name is formed with

the class name preceded by the character @. For example, one of the examples used in this chapter is a class involving polynomials in a single variable. The name of the class, and the name of the class constructor, is `polynom`. The M-files defining a polynomial class would be located in directory with the name `@polynom`.

The class directories are subdirectories of directories on the MATLAB search path, but are not themselves on the path. For instance, the new `@polynom` directory could be a subdirectory of the MATLAB working directory or your own personal directory that has been added to the search path.

Adding the Class Directory to the MATLAB Path

After creating the class directory, you need to update the MATLAB path so that MATLAB can locate the class source files. The class directory should not be directly on the MATLAB path. Instead, you should add the parent directory to the MATLAB path. For example, if the `@polynom` class directory is located at

```
c:\myClasses\@polynom
```

you add the class directory to the MATLAB path with the `addpath` command

```
addpath c:\myClasses;
```

Using Multiple Class Directories

A MATLAB class can access methods in multiple *@classname* directories if all such directories are visible to MATLAB (i.e., the parent directories are on the MATLAB path or in the current directory). When you attempt to use a method of the class, MATLAB searches all the visible directories named *@classname* for the appropriate method.

For more information, see “How MATLAB Determines Which Method to Call” on page 9-72.

Data Structure

One of the first steps in the design of a new class is the choice of the data structure to be used by the class. Objects are stored in MATLAB structures. The fields of the structure, and the details of operations on the fields, are

visible only within the methods for the class. The design of the appropriate data structure can affect the performance of the code.

Tips for C++ and Java Programmers

If you are accustomed to programming in other object-oriented languages, such as C++ or Java, you will find that the MATLAB programming language differs from these languages in some important ways:

- In MATLAB, method dispatching is not syntax based, as it is in C++ and Java. When the argument list contains objects of equal precedence, MATLAB uses the left-most object to select the method to call.
- In MATLAB, there is no equivalent to a destructor method. To remove an object from the workspace, use the `clear` function.
- Construction of MATLAB data types occurs at runtime rather than compile time. You register an object as belonging to a class by calling the `class` function.
- When using inheritance in MATLAB, the inheritance relationship is established in the child class by creating the parent object, and then calling the `class` function. For more information on writing constructors for inheritance relationships, see “Building on Other Classes” on page 9-38.
- When using inheritance in MATLAB, the child object contains a parent object in a property with the name of the parent class.
- In MATLAB, there is no passing of variables by reference. When writing methods that update an object, you must pass back the updated object and use an assignment statement. For instance, this call to the `set` method updates the `name` field of the object `A` and returns the updated object.

  ```
  A = set(A,'name','John Smith');
  ```

- In MATLAB, there is no equivalent to an abstract class.
- In MATLAB, there is no equivalent to the C++ scoping operator.
- In MATLAB, there is no virtual inheritance or virtual base classes.
- In MATLAB, there is no equivalent to C++ templates.

Designing User Classes in MATLAB

In this section...

The MATLAB Canonical Class

When you design a MATLAB class, you should include a standard set of methods that enable the class to behave in a consistent and logical way within the MATLAB environment. Depending on the nature of the class you are defining, you may not need to include all of these methods and you may include a number of other methods to realize the class's design goals.

This table lists the basic methods included in MATLAB classes.

Class Method	Description
class constructor	Creates an object of the class.

Class Method	Description
`display`	Called whenever MATLAB displays the contents of an object (e.g., when an expression is entered without terminating with a semicolon).
`set` and `get`	Accesses class properties.
`subsref` and `subsasgn`	Enables indexed reference and assignment for user objects.
`end`	Supports `end` syntax in indexing expressions using an object; e.g., `A(1:end)`.
`subsindex`	Supports using an object in indexing expressions.
converters like `double` and `char`	Methods that convert an object to a MATLAB data type.

The following sections discuss the implementation of each type of method, as well as providing references to examples used in this chapter.

The Class Constructor Method

The @ directory for a particular class must contain an M-file known as the *constructor* for that class. The name of the constructor is the same as the name of the directory (excluding the @ prefix and `.m` extension) that defines the name of the class. The constructor creates the object by initializing the data structure and instantiating an object of the class.

Guidelines for Writing a Constructor

Class constructors must perform certain functions so that objects behave correctly in the MATLAB environment. In general, a class constructor must handle three possible combinations of input arguments:

- No input arguments
- An object of the same class as an input argument
- The input arguments used to create an object of the class (typically data of some kind)

No Input Arguments. If there are no input arguments, the constructor should create a default object. Since there are no inputs, you have no data from which to create the object, so you simply initialize the object's data structures with empty or default values, call the `class` function to instantiate the object, and return the object as the output argument. Support for this syntax is required for two reasons:

- When loading objects into the workspace, the `load` function calls the class constructor with no arguments.
- When creating arrays of objects, MATLAB calls the class constructor to add objects to the array.

Object Input Argument. If the first input argument in the argument list is an object of the same class, the constructor should simply return the object. Use the `isa` function to determine if an argument is a member of a class. See "Overloading the + Operator" on page 9-32 for an example of a method that uses this constructor syntax.

Data Input Arguments. If the input arguments exist and are not objects of the same class, then the constructor creates the object using the input data. Of course, as in any function, you should perform proper argument checking in your constructor function. A typical approach is to use a `varargin` input argument and a `switch` statement to control program flow. This provides an easy way to accommodate the three cases: no inputs, object input, or the data inputs used to create an object.

It is in this part of the constructor that you assign values to the object's data structure, call the `class` function to instantiate the object, and return the object as the output argument. If necessary, place the object in an object hierarchy using the `superiorto` and `inferiorto` functions.

Using the class Function in Constructors

Within a constructor method, you use the `class` function to associate an object structure with a particular class. This is done using an internal class tag that is only accessible using the `class` and `isa` functions. For example, this call to the `class` function identifies the object `p` to be of type `polynom`.

```
p = class(p,'polynom');
```

Examples of Constructor Methods

See the following sections for examples of constructor methods:

- "The Polynom Constructor Method" on page 9-27
- "The Asset Constructor Method" on page 9-43
- "The Stock Constructor Method" on page 9-50
- "The Portfolio Constructor Method" on page 9-59

Identifying Objects Outside the Class Directory

The `class` and `isa` functions used in constructor methods can also be used outside of the class directory. The expression

```
isa(a,'classname');
```

checks whether `a` is an object of the specified class. For example, if `p` is a polynom object, each of the following expressions is true.

```
isa(pi,'double');
isa('hello','char');
isa(p,'polynom');
```

Outside of the class directory, the `class` function takes only one argument (it is only within the constructor that `class` can have more than one argument).

The expression

```
class(a)
```

returns a string containing the class name of `a`. For example,

```
class(pi),
class('hello'),
class(p)
```

return

```
'double',
'char',
'polynom'
```

Use the whos function to see what objects are in the MATLAB workspace.

```
whos

  Name      Size          Bytes  Class
  p         1x1           156 polynom object
```

The display Method

MATLAB calls a method named display whenever an object is the result of a statement that is not terminated by a semicolon. For example, creating the variable a, which is a double, calls the MATLAB display method for doubles.

```
a = 5
a =
     5
```

You should define a display method so MATLAB can display values on the command line when referencing objects from your class. In many classes, display can simply print the variable name, and then use the char converter method to print the contents or value of the variable, since MATLAB displays output as strings. You must define the char method to convert the object's data to a character string.

Examples of display Methods

See the following sections for examples of display methods:

- "The Polynom display Method" on page 9-30
- "The Asset display Method" on page 9-48
- "The Stock display Method" on page 9-57
- "The Portfolio display Method" on page 9-61

Accessing Object Data

You need to write methods for your class that provide access to an object's data. Accessor methods can use a variety of approaches, but all methods that change object data always accept an object as an input argument and return a new object with the data changed. This is necessary because MATLAB does

not support passing arguments by reference (i.e., pointers). Functions can change only their private, temporary copy of an object. Therefore, to change an existing object, you must create a new one, and then replace the old one.

The following sections provide more detail about implementation techniques for the `set`, `get`, `subsasgn`, and `subsref` methods.

The set and get Methods

The `set` and `get` methods provide a convenient way to access object data in certain cases. For example, suppose you have created a class that defines an arrow object that MATLAB can display on graphs (perhaps composed of existing MATLAB line and patch objects).

To produce a consistent interface, you could define `set` and `get` methods that operate on arrow objects the way the MATLAB `set` and `get` functions operate on built-in graphics objects. The `set` and `get` verbs convey what operations they perform, but insulate the user from the internals of the object.

Examples of set and get Methods

See the following sections for examples of `set` and `get` methods:

- "The Asset get Method" on page 9-44 and "The Asset set Method" on page 9-45
- "The Stock get Method" on page 9-52 and "The Stock set Method" on page 9-53

Property Name Methods

As an alternative to a general `set` method, you can write a method to handle the assignment of an individual property. The method should have the same name as the property name.

For example, if you defined a class that creates objects representing employee data, you might have a field in an employee object called `salary`. You could then define a method called `salary.m` that takes an employee object and a value as input arguments and returns the object with the specified value set.

Indexed Reference Using subsref and subsasgn

User classes implement new data types in MATLAB. It is useful to be able to access object data via an indexed reference, as is possible with the MATLAB built-in data types. For example, if `A` is an array of class `double`, `A(i)` returns the i^{th} element of `A`.

As the class designer, you can decide what an index reference to an object means. For example, suppose you define a class that creates polynomial objects and these objects contain the coefficients of the polynomial.

An indexed reference to a polynomial object,

```
p(3)
```

could return the value of the coefficient of x^3, the value of the polynomial at $x = 3$, or something different depending on the intended design.

You define the behavior of indexing for a particular class by creating two class methods - `subsref` and `subsasgn`. MATLAB calls these methods whenever a subscripted reference or assignment is made on an object from the class. If you do not define these methods for a class, indexing is undefined for objects of this class.

In general, the rules for indexing objects are the same as the rules for indexing structure arrays. For details, see “Structures” on page 2-74.

Behavior Within Class Methods

If `A` is an array of one of the fundamental MATLAB data types, then referencing a value of `A` using an indexed reference calls the built-in MATLAB `subsref` method. It does not call any `subsref` method that you might have overloaded for that data type.

For example, if `A` is an array of type double, and there is an `@double/subsref` method on your MATLAB path, the statement `B = A(I)` does not call this method, but calls the MATLAB built-in `subsref` method instead.

The same is true for user-defined classes. Whenever a class method requires the functionality of the overloaded `subsref` or `subsassign`, it must call the

overloaded methods with function calls rather than using operators like '()', '{}', or '.'.

For example, suppose you define a polynomial class that defines a subsref method that causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. Therefore,

```
p = polynom([1 0 -2 -5]);
```

The following subscripted expression returns the value of the polynomial at

```
x = 3 and x = 4.
p([3 4])
ans =
    16   51
```

Now suppose that you want to use this feature in one of the class methods. To do so, you must call the subsref function directly:

```
y = polyval(p,x);
 subs.type = '()';
 subs.subs = {x};
 y = subsref(p, subs);  % Need to call subsref here
```

Handling Subscripted Reference

The use of a subscript or field designator with an object on the right-hand side of an assignment statement is known as a *subscripted reference*. MATLAB calls a method named subsref in these situations.

Object subscripted references can be of three forms — an array index, a cell array index, and a structure field name:

```
A(I)
A{I}
A.field
```

Each of these results in a call by MATLAB to the subsref method in the class directory. MATLAB passes two arguments to subsref.

```
B = subsref(A,S)
```

The first argument is the object being referenced. The second argument, `S`, is a structure array with two fields:

- `S.type` is a string containing `'()'`, `'{}'`, or `'.'` specifying the subscript type. The parentheses represent a numeric array; the curly braces, a cell array; and the dot, a structure array.
- `S.subs` is a cell array or string containing the actual subscripts. A colon used as a subscript is passed as a cell array containing the string `':'`.

For instance, the expression

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

causes MATLAB to call `subsref(A,S)`, where `S` is a 1-by-1 structure with

```
S.type = '()'
S.subs = {1:2,':'}
```

Similarly, the expression

```
A{1:2}
```

uses

```
S.type ='{}'
S.subs = {1:2}
```

The expression

```
A.field
```

calls `subsref(A,S)` where

```
S.type = '.'
S.subs = 'field'
```

These simple calls are combined for more complicated subscripting expressions. In such cases, `length(S)` is the number of subscripting levels. For example,

```
A(1,2).name(3:4)
```

calls `subsref(A,S)`, where `S` is a 3-by-1 structure array with the values:

```
S(1).type = '()'    S(2).type = '.'      S(3).type = '()'
S(1).subs = {1,2}   S(2).subs = 'name'   S(3).subs = {3:4}
```

How to Write subsref

The `subsref` method must interpret the subscripting expressions passed in by MATLAB. A typical approach is to use the `switch` statement to determine the type of indexing used and to obtain the actual indices. The following three code fragments illustrate how to interpret the input arguments. In each case, the function must return the value `B`.

For an array index:

```
switch S.type
case '()'
   B = A(S.subs{:});
end
```

For a cell array:

```
switch S.type
case '{}'
   B = A(S.subs{:});          % A is a cell array
end
```

For a structure array:

```
switch S.type
case '.'
   switch S.subs
   case 'field1'
      B = A.field1;
   case 'field2'
      B = A.field2;
   end
end
```

Examples of the subsref Method

See the following sections for examples of the `subsref` method:

- "The Polynom subsref Method" on page 9-31
- "The Asset subsref Method" on page 9-46
- "The Stock subsref Method" on page 9-54
- "The Portfolio subsref Method" on page 9-68

Handling Subscripted Assignment

The use of a subscript or field designator with an object on the left-hand side of an assignment statement is known as a *subscripted assignment*. MATLAB calls a method named `subsasgn` in these situations. Object subscripted assignment can be of three forms - an array index, a cell array index, and a structure field name.

```
A(I) = B
A{I} = B
A.field = B
```

Each of these results in a call to `subsasgn` of the form

```
A = subsasgn(A,S,B)
```

The first argument, `A`, is the object being referenced. The second argument, `S`, has the same fields as those used with `subsref`. The third argument, `B`, is the new value.

Examples of the subsasgn Method

See the following sections for examples of the `subsasgn` method:

- "The Asset subsasgn Method" on page 9-47
- "The Stock subsasgn Method" on page 9-55

Object Indexing Within Methods

If a subscripted reference is made within a class method, MATLAB uses its built-in `subsref` function to access data within the method's own class. If the method accesses data from another class, MATLAB calls the overloaded `subsref` function in that class. The same holds true for subscripted assignment and `subsasgn`.

The following example shows a method, `testref`, that is defined in the class, `employee`. This method makes a reference to a field, `address`, in an object of its own class. For this, MATLAB uses the built-in `subsref` function. It also references the same field in another class, this time using the overloaded `subsref` of that class.

```
% ---- EMPLOYEE class method: testref.m ----
function testref(myclass,otherclass)

myclass.address                % use built-in subsref
otherclass.address             % use overloaded subsref
```

The example creates an `employee` object and a `company` object.

```
empl = employee('Johnson','Chicago');
comp = company('The MathWorks','Natick');
```

The `employee` class method, `testref`, is called. MATLAB uses an overloaded `subsref` only to access data outside of the method's own class.

```
testref(empl,comp)
ans =                           % built-in subsref was called
   Chicago

ans =                           % @company\subsref was called
Executing @company\subsref ...
   Natick
```

Defining end Indexing for an Object

When you use `end` in an object indexing expression, MATLAB calls the object's `end` class method. If you want to be able to use `end` in indexing expressions involving objects of your class, you must define an `end` method for your class.

The `end` method has the calling sequence

```
end(a,k,n)
```

where `a` is the user object, `k` is the index in the expression where the `end` syntax is used, and `n` is the total number of indices in the expression.

For example, consider the expression

```
A(end-1,:)
```

MATLAB calls the `end` method defined for the object `A` using the arguments

```
end(A,1,2)
```

That is, the `end` statement occurs in the first index element and there are two index elements. The class method for `end` must then return the index value for the last element of the first dimension. When you implement the `end` method for your class, you must ensure it returns a value appropriate for the object.

Indexing an Object with Another Object

When MATLAB encounters an object as an index, it calls the `subsindex` method defined for the object. For example, suppose you have an object `a` and you want to use this object to index into another object `b`.

```
c = b(a);
```

A `subsindex` method might do something as simple as convert the object to double format to be used as an index, as shown in this sample code.

```
function d = subsindex(a)
%SUBSINDEX
% convert the object a to double format to be used
% as an index in an indexing expression
d = double(a);
```

`subsindex` values are 0-based, not 1-based.

Converter Methods

A converter method is a class method that has the same name as another class, such as `char` or `double`. Converter methods accept an object of one class as input and return an object of another class. Converters enable you to

- Use methods defined for another class
- Ensure that expressions involving objects of mixed class types execute properly

A converter function call is of the form

```
b = classname(a)
```

where `a` is an object of a class other than *`classname`*. In this case, MATLAB looks for a method called *`classname`* in the class directory for object `a`. If the input object is already of type *`classname`*, then MATLAB calls the constructor, which just returns the input argument.

Examples of Converter Methods

See the following sections for examples of converter methods:

- "The Polynom to Double Converter" on page 9-28
- "The Polynom to Char Converter" on page 9-29

Overloading Operators and Functions

In this section...

Overloading Operators

In many cases, you may want to change the behavior of the MATLAB operators and functions for cases when the arguments are objects. You can accomplish this by overloading the relevant functions. Overloading enables a function to handle different types and numbers of input arguments and perform whatever operation is appropriate for the highest-precedence object. See "Object Precedence" on page 9-70 for more information on object precedence.

Each built-in MATLAB operator has an associated function name (e.g., the + operator has an associated `plus.m` function). You can overload any operator by creating an M-file with the appropriate name in the class directory. For example, if either `p` or `q` is an object of type `classname`, the expression

```
p + q
```

generates a call to a function `@classname/plus.m`, if it exists. If `p` and `q` are both objects of different classes, then MATLAB applies the rules of precedence to determine which method to use.

Examples of Overloaded Operators

See the following sections for examples of overloaded operators:

- "Overloading the + Operator" on page 9-32
- "Overloading the - Operator" on page 9-33
- "Overloading the * Operator" on page 9-33

The following table lists the function names for most of the MATLAB operators.

Operation	M-File	Description
`a + b`	`plus(a,b)`	Binary addition
`a - b`	`minus(a,b)`	Binary subtraction
`-a`	`uminus(a)`	Unary minus
`+a`	`uplus(a)`	Unary plus
`a.*b`	`times(a,b)`	Element-wise multiplication
`a*b`	`mtimes(a,b)`	Matrix multiplication
`a./b`	`rdivide(a,b)`	Right elementwise division
`a.\b`	`ldivide(a,b)`	Left elementwise division
`a/b`	`mrdivide(a,b)`	Matrix right division
`a\b`	`mldivide(a,b)`	Matrix left division
`a.^b`	`power(a,b)`	Element-wise power
`a^b`	`mpower(a,b)`	Matrix power
`a < b`	`lt(a,b)`	Less than
`a > b`	`gt(a,b)`	Greater than
`a <= b`	`le(a,b)`	Less than or equal to
`a >= b`	`ge(a,b)`	Greater than or equal to
`a ~= b`	`ne(a,b)`	Not equal to
`a == b`	`eq(a,b)`	Equality
`a & b`	`and(a,b)`	Logical AND
`a | b`	`or(a,b)`	Logical OR
`~a`	`not(a)`	Logical NOT
`a:d:b` `a:b`	`colon(a,d,b)` `colon(a,b)`	Colon operator
`a'`	`ctranspose(a)`	Complex conjugate transpose
`a.'`	`transpose(a)`	Matrix transpose
`command window output`	`display(a)`	Display method

Operation	M-File	Description
`[a b]`	`horzcat(a,b,...)`	Horizontal concatenation
`[a; b]`	`vertcat(a,b,...)`	Vertical concatenation
`a(s1,s2,...sn)`	`subsref(a,s)`	Subscripted reference
`a(s1,...,sn) = b`	`subsasgn(a,s,b)`	Subscripted assignment
`b(a)`	`subsindex(a)`	Subscript index

Overloading Functions

You can overload any function by creating a function of the same name in the class directory. When a function is invoked on an object, MATLAB always looks in the class directory before any other location on the search path. To overload the `plot` function for a class of objects, for example, simply place your version of `plot.m` in the appropriate class directory.

Examples of Overloaded Functions

See the following sections for examples of overloaded functions:

- "Overloading Functions for the Polynom Class" on page 9-34
- "The Portfolio pie3 Method" on page 9-61

Example — A Polynomial Class

In this section...

Polynom Data Structure

This example implements a MATLAB data type for polynomials by defining a new class called `polynom`. The polynom class represents a polynomial with a row vector containing the coefficients of powers of the variable, in decreasing order. Therefore, a polynom object `p` is a structure with a single field, `p.c`, containing the coefficients. This field is accessible only within the methods in the `@polynom` directory.

Polynom Methods

To create a class that is well behaved within the MATLAB environment and provides useful functionality for a polynomial data type, the polynom class implements the following methods:

- A constructor method `polynom.m`
- A polynom to double converter
- A polynom to char converter
- A `display` method
- A `subsref` method

- Overloaded +, -, and * operators
- Overloaded `roots`, `polyval`, `plot`, and `diff` functions

The Polynom Constructor Method

Here is the polynom class constructor, `@polynom/polynom.m`.

```
function p = polynom(a)
%POLYNOM Polynomial class constructor.
%   p = POLYNOM(v) creates a polynomial object from vector v,
%   containing the coefficients of descending powers of x.
if nargin == 0
   p.c = [];
   p = class(p,'polynom');
elseif isa(a,'polynom')
   p = a;
else
   p.c = a(:).';
   p = class(p,'polynom');
end
```

Constructor Calling Syntax

You can call the polynom constructor method with one of three different arguments:

- No input argument — If you call the constructor function with no arguments, it returns a polynom object with empty fields.
- Input argument is an object — If you call the constructor function with an input argument that is already a polynom object, MATLAB returns the input argument. The `isa` function (pronounced “is a”) checks for this situation.
- Input argument is a coefficient vector — If the input argument is a variable that is not a polynom object, reshape it to be a row vector and assign it to the `.c` field of the object’s structure. The `class` function creates the `polynom` object, which is then returned by the constructor.

An example use of the `polynom` constructor is the statement

```
p = polynom([1 0 -2 -5])
```

This creates a polynomial with the specified coefficients.

Converter Methods for the Polynom Class

A converter method converts an object of one class to an object of another class. Two of the most important converter methods contained in MATLAB classes are `double` and `char`. Conversion to `double` produces the MATLAB traditional matrix, although this may not be appropriate for some classes. Conversion to `char` is useful for producing printed output.

The Polynom to Double Converter

The double converter method for the polynom class is a very simple M-file, `@polynom/double.m`, which merely retrieves the coefficient vector.

```
function c = double(p)
% POLYNOM/DOUBLE  Convert polynom object to coefficient vector.
%   c = DOUBLE(p) converts a polynomial object to the vector c
%   containing the coefficients of descending powers of x.
c = p.c;
```

On the object `p`,

```
p = polynom([1 0 -2 -5])
```

the statement

```
double(p)
```

returns

```
 ans =
 1 0 -2 -5
```

Having implemented the `double` method, you can use it to call MATLAB functions on polynom objects that require double values as inputs. For example,

```
size(double(p))
 ans =
 1  4
```

The Polynom to Char Converter

The converter to char is a key method because it produces a character string involving the powers of an independent variable, `x`. Therefore, once you have specified `x`, the string returned is a syntactically correct MATLAB expression, which you can then evaluate.

Here is `@polynom/char.m`.

```
function s = char(p)
% POLYNOM/CHAR
% CHAR(p) is the string representation of p.c
if all(p.c == 0)
   s = '0';
else
   d = length(p.c) - 1;
   s = [];
   for a = p.c;
      if a ~= 0;
         if ~isempty(s)
            if a > 0
               s = [s ' + '];
            else
               s = [s ' - '];
               a = -a;
            end
         end
         if a ~= 1 | d == 0
            s = [s num2str(a)];
            if d > 0
               s = [s '*'];
            end
         end
         if d >= 2
            s = [s 'x^' int2str(d)];
         elseif d == 1
            s = [s 'x'];
         end
      end
      d = d - 1;
   end
```

```
end
```

Evaluating the Output

If you create the polynom object `p`

```
p = polynom([1 0 -2 -5]);
```

and then call the `char` method on `p`

```
char(p)
```

MATLAB produces the result

```
ans =
    x^3 - 2*x - 5
```

The value returned by `char` is a string that you can pass to `eval` once you have defined a scalar value for `x`. For example,

```
x = 3;

eval(char(p))
ans =
    16
```

See “The Polynom subsref Method” on page 9-31 for a better method to evaluate the polynomial.

The Polynom display Method

Here is `@polynom/display.m`. This method relies on the `char` method to produce a string representation of the polynomial, which is then displayed on the screen. This method produces output that is the same as standard MATLAB output. That is, the variable name is displayed followed by an equal sign, then a blank line, then a new line with the value.

```
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom
disp(' ');
disp([inputname(1),' = '])
disp(' ');
```

```
disp(['   ' char(p)])
disp(' ');
```

The statement

```
p = polynom([1 0 -2 -5])
```

creates a polynom object. Since the statement is not terminated with a semicolon, the resulting output is

```
p =
    x^3 - 2*x - 5
```

The Polynom subsref Method

Suppose the design of the polynom class specifies that a subscripted reference to a `polynom` object causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. That is, for a polynom object `p`,

```
p = polynom([1 0 -2 -5]);
```

the following subscripted expression returns the value of the polynomial at `x = 3` and `x = 4`.

```
p([3 4])
ans =
    16   51
```

subsref Implementation Details

This implementation takes advantage of the `char` method already defined in the polynom class to produce an expression that can then be evaluated.

```
function b = subsref(a,s)
% SUBSREF
switch s.type
case '()'
   ind = s.subs{:};
   for k = 1:length(ind)
      b(k) = eval(strrep(char(a), 'x', ...
             ['(' num2str(ind(k)) ')']));
   end
otherwise
```

```
    error('Specify value for x as p(x)')
end
```

Once the polynomial expression has been generated by the `char` method, the `strrep` function is used to swap the passed in value for the character `x`. The `eval` function then evaluates the expression and returns the value in the output argument.

Note that if you perform an indexed reference from within other class methods, MATLAB calls the built-in `subsref` or `subsassign`. See "Behavior Within Class Methods" on page 9-15 for more information.

Overloading Arithmetic Operators for polynom

Several arithmetic operations are meaningful on polynomials and should be implemented for the polynom class. When overloading arithmetic operators, keep in mind what data types you want to operate on. In this section, the `plus`, `minus`, and `mtimes` methods are defined for the polynom class to handle addition, subtraction, and multiplication on polynom/polynom and polynom/double combinations of operands.

Overloading the + Operator

If either `p` or `q` is a polynom, the expression

```
p + q
```

generates a call to a function `@polynom/plus.m`, if it exists (unless `p` or `q` is an object of a higher precedence, as described in "Object Precedence" on page 9-70).

The following M-file redefines the + operator for the polynom class.

```
function r = plus(p,q)
% POLYNOM/PLUS  Implement p + q for polynoms.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] + [zeros(1,-k) q.c]);
```

The function first makes sure that both input arguments are polynomials. This ensures that expressions such as

```
p + 1
```

that involve both a polynom and a double, work correctly. The function then accesses the two coefficient vectors and, if necessary, pads one of them with zeros to make them the same length. The actual addition is simply the vector sum of the two coefficient vectors. Finally, the function calls the `polynom` constructor a third time to create the properly typed result.

Overloading the - Operator

You can implement the overloaded minus operator (`-`) using the same approach as the plus (`+`) operator. MATLAB calls `@polynom/minus.m` to compute `p-q`.

```
function r = minus(p,q)
% POLYNOM/MINUS Implement p - q for polynoms.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] - [zeros(1,-k) q.c]);
```

Overloading the * Operator

MATLAB calls the method `@polynom/mtimes.m` to compute the product `p*q`. The letter `m` at the beginning of the function name comes from the fact that it is overloading the MATLAB *matrix* multiplication. Multiplication of two polynomials is simply the convolution of their coefficient vectors.

```
function r = mtimes(p,q)
% POLYNOM/MTIMES   Implement p * q for polynoms.
p = polynom(p);
q = polynom(q);
r = polynom(conv(p.c,q.c));
```

Using the Overloaded Operators

Given the polynom object

```
p = polynom([1 0 -2 -5])
```

MATLAB calls these two functions `@polynom/plus.m` and `@polynom/mtimes.m` when you issue the statements

```
q = p+1
r = p*q
```

to produce

```
q =
     x^3 - 2*x - 4

r =
     x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

Overloading Functions for the Polynom Class

MATLAB already has several functions for working with polynomials represented by coefficient vectors. They should be overloaded to also work with the new polynom object. In many cases, the overloading methods can simply apply the original function to the coefficient field.

Overloading roots for the Polynom Class

The method `@polynom/roots.m` finds the roots of polynom objects.

```
function r = roots(p)
% POLYNOM/ROOTS.  ROOTS(p) is a vector containing the roots of p.
r = roots(p.c);
```

The statement

```
roots(p)
```

results in

```
ans =
     2.0946
    -1.0473 + 1.1359i
    -1.0473 - 1.1359i
```

Overloading polyval for the Polynom Class

The function `polyval` evaluates a polynomial at a given set of points. `@polynom/polyval.m` uses nested multiplication, or Horner's method to reduce the number of multiplication operations used to compute the various powers of x.

```
function y = polyval(p,x)
% POLYNOM/POLYVAL  POLYVAL(p,x) evaluates p at the points x.
y = 0;
for a = p.c
   y = y.*x + a;
end
```

Overloading plot for the Polynom Class

The overloaded `plot` function uses both `root` and `polyval`. The function selects the domain of the independent variable to be slightly larger than an interval containing the roots of the polynomial. Then `polyval` is used to evaluate the polynomial at a few hundred points in the domain.

```
function plot(p)
% POLYNOM/PLOT  PLOT(p) plots the polynom p.
r = max(abs(roots(p)));
x = (-1.1:0.01:1.1)*r;
y = polyval(p,x);
plot(x,y);
title(char(p))
grid on
```

Overloading diff for the Polynom Class

`The method @polynom/diff.m` differentiates a polynomial by reducing the degree by 1 and multiplying each coefficient by its original degree.

```
function q = diff(p)
% POLYNOM/DIFF  DIFF(p) is the derivative of the polynom p.
c = p.c;
d = length(c) - 1;  % degree
q = polynom(p.c(1:d).*(d:-1:1));
```

Listing Class Methods

The function call

```
methods('classname')
```

or its command form

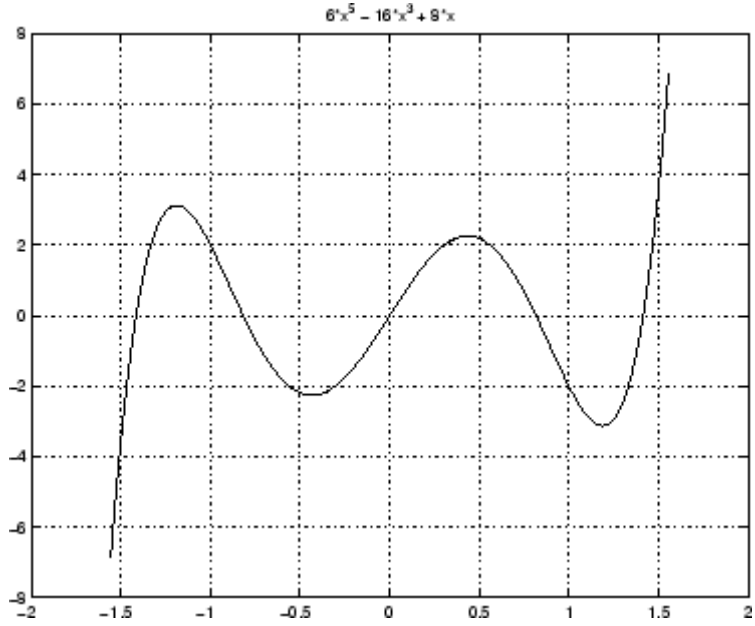
```
methods classname
```

shows all the methods available for a particular class. For the `polynom` example, the output is

```
methods polynom
Methods for class polynom:

char       display    minus      plot       polynom    roots
diff       double     mtimes     plus       polyval    subsref
```

Plotting the two polynom objects x and p calls most of these methods.

```
x = polynom([1 0]);
p = polynom([1 0 -2 -5]);
plot(diff(p*p + 10*p + 20*x) - 20)
```



Building on Other Classes

In this section...

Overview

A MATLAB classes can *inherit* properties and behavior from another MATLAB class. When one object (of the derived class) inherits from another (of the base class), the derived class object includes all the fields of the base class object and can call the base class methods. The base class methods can access those fields that the derived class object inherited from the base class, but not fields new to the derived class.

Inheritance is a key feature of object-oriented programming. It makes it easy to reuse code by allowing derived class objects to take advantage of code that exists for base class objects. Inheritance enables a derived class object to behave exactly like a base class object, which facilitates the development of related classes that behave similarly, but are implemented differently.

There are two kinds of inheritance:

- Simple inheritance, in which a derived class object inherits characteristics from one base class.
- Multiple inheritance, in which a derived class object inherits characteristics from more than one parent class.

This section also discusses a related topic, aggregation. Aggregation allows one object to contain another object as one of its fields.

Simple Inheritance

A class that inherits attributes from a single base class, and adds new attributes of its own, uses simple inheritance. Inheritance implies that objects

belonging to the derived class have the same fields as the base class, as well as any fields added by the derived class.

Base class methods can operate on objects belonging to the derived class. However, derived class methods cannot operate on objects belonging to the base class. You cannot access the base class object's fields directly from the derived class; you must use access methods defined for the base class.

Derived Class Constructor

The constructor function for a derived class has two special characteristics:

- It calls the constructor function for the base class to create the inherited fields.
- It requires a special calling syntax for the `class` function, specifying both the derived class and the base class.

The syntax for establishing simple inheritance using the `class` function is:

```
derivedObj = class(derivedObj, 'derivedClass', baseObj);
```

Simple inheritance can span more than one generation. If a base class is itself an inherited class, the derived class object automatically inherits from the original base class.

Visibility of Class Properties and Methods

The base class does not have access to the derived class properties. The derived class cannot access the base class properties directly, but must use base class access methods (e.g., `get` or `subsref` method) to access the base class properties. From the derived class methods, this access is accomplished via the base class field in the derived class structure. For example, when a constructor creates a derived class object `c`,

```
c = class(c, 'derivedClassname', baseObject);
```

MATLAB automatically creates a field, `c.`*`baseClassname`*, in the object's structure that contains the base object. You could then have a statement in the derived class display method that calls the base class display method.

```
display(c.baseClassname)
```

See “Designing the Stock Class” on page 9-49 for examples that use simple inheritance.

Multiple Inheritance

In the multiple inheritance case, a class of objects inherits attributes from more than one base class. The derived class object gets fields from all the base classes, as well as fields of its own.

Multiple inheritance can encompass more than one generation. For example, each of the base class objects could have inherited fields from multiple base class objects, and so on. Multiple inheritance is implemented in the constructors by calling `class` with more than three arguments.

```
obj = class(structure,'classname',baseclass1,baseclass2,...)
```

You can append as many base class arguments as desired to the class input list.

Nonunique Method Names in Base Classes

Multiple base classes can have associated methods of the same name. In this case, MATLAB calls the method associated with the base class that appears first in the `class` function call in the constructor function. There is no way to access subsequent base class functions of this name.

Aggregation

In addition to standard inheritance, MATLAB objects support containment or *aggregation*. That is, one object can contain (embed) another object as one of its fields. For example, a rational object might use two polynom objects, one for the numerator and one for the denominator.

You can call a method for the contained object only from within a method for the outer object. When determining which version of a function to call, MATLAB considers only the outermost containing class of the objects passed as arguments; the classes of any contained objects are ignored.

See “Example — The Portfolio Container” on page 9-58 for an example of aggregation.

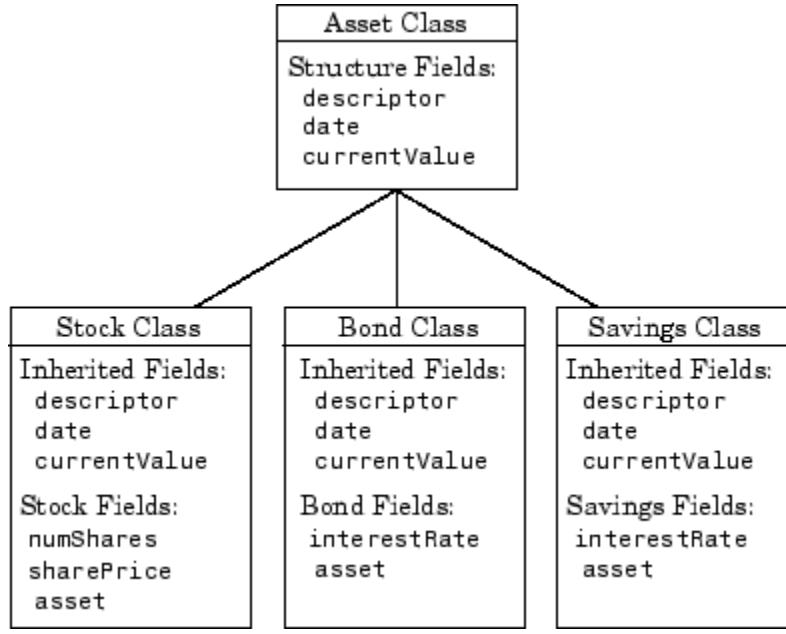
Example — Assets and Asset Subclasses

In this section...

Inheritance Model for the Asset Class

As an example of simple inheritance, consider a general `asset` class that can be used to represent any item that has monetary value. Some examples of an asset are: stocks, bonds, savings accounts, and any other piece of property. In designing this collection of classes, the `asset` class holds the data that is common to all of the specialized asset subclasses. The individual asset subclasses, such as the `stock` class, inherit the `asset` properties and contribute additional properties. The subclasses are "kinds of" assets.

An example of a simple inheritance relationship using an asset base class is shown in this diagram.



As shown in the diagram, the `stock`, `bond`, and `savings` classes inherit structure fields from the `asset` class. In this example, the `asset` class is used to provide storage for data common to all subclasses and to share asset methods with these subclasses. This example shows how to implement the asset and stock classes. The `bond` and `savings` classes can be implemented in a way that is very similar to the `stock` class, as would other types of asset subclasses.

Asset Class Design

The asset class provides storage and access for information common to all asset children. It is not intended to be instantiated directly, so it does not require an extensive set of methods. To serve its purpose, the class needs to contain the following methods:

- Constructor

- `get` and `set`
- `subsref` and `subsasgn`
- `display`

Other Asset Methods

The asset class provides inherited data storage for its child classes, but is not instanced directly. The `set`, `get`, and `display` methods provide access to the stored data. It is not necessary to implement the full complement of methods for asset objects (such as converters, `end`, and `subsindex`) since only the child classes access the data.

The Asset Constructor Method

The asset class is based on a structure array with four fields:

- `descriptor` — Identifier of the particular asset (e.g., stock name, savings account number, etc.)
- `date` — The date the object was created (calculated by the `date` command)
- `type` — The type of asset (e.g., savings, bond, stock)
- `currentValue` — The current value of the asset (calculated from subclass data)

This information is common to asset child objects (stock, bond, and savings), so it is handled from the parent object to avoid having to define the same fields in each child class. This is particularly helpful as the number of child classes increases.

```
function a = asset(varargin)
% ASSET Constructor function for asset object
% a = asset(descriptor, type, currentValue)
switch nargin
case 0
% if no input arguments, create a default object
   a.descriptor = 'none';
   a.date = date;
   a.type = 'none';
   a.currentValue = 0;
```

```
   a = class(a,'asset');
case 1
% if single argument of class asset, return it
   if (isa(varargin{1},'asset'))
      a = varargin{1};
   else
      error('Wrong argument type')
   end
case 3
% create object using specified values
   a.descriptor = varargin{1};
   a.date = date;
   a.type = varargin{2};
   a.currentValue = varargin{3};
   a = class(a,'asset');
otherwise
   error('Wrong number of input arguments')
end
```

The function uses a `switch` statement to accommodate three possible scenarios:

- Called with no arguments, the constructor returns a default asset object.
- Called with one argument that is an asset object, the object is simply returned.
- Called with three arguments (subclass descriptor, type, and current value), the constructor returns a new asset object.

The asset constructor method is not intended to be called directly; it is called from the child constructors since its purpose is to provide storage for common data.

The Asset get Method

The asset class needs methods to access the data contained in asset objects. The following function implements a `get` method for the class. It uses capitalized property names rather than literal field names to provide an interface similar to other MATLAB objects.

```
function val = get(a, propName)
% GET Get asset properties from the specified object
% and return the value
switch propName
case 'Descriptor'
   val = a.descriptor;
case 'Date'
   val = a.date;
case 'CurrentValue'
   val = a.currentValue;
otherwise
   error([propName,' Is not a valid asset property'])
end
```

This function accepts an object and a property name and uses a `switch` statement to determine which field to access. This method is called by the subclass `get` methods when accessing the data in the inherited properties. See "The Stock get Method" on page 9-52 for an example.

The Asset set Method

The asset class `set` method is called by subclass `set` methods. This method accepts an asset object and variable length argument list of property name/property value pairs and returns the modified object.

```
function a = set(a,varargin)
% SET Set asset properties and return the updated object
propertyArgIn = varargin;
while length(propertyArgIn) >= 2,
   prop = propertyArgIn{1};
   val = propertyArgIn{2};
   propertyArgIn = propertyArgIn(3:end);
   switch prop
   case 'Descriptor'
      a.descriptor = val;
   case 'Date'
      a.date = val;
   case 'CurrentValue'
      a.currentValue = val;
   otherwise
```

```
        error('Asset properties: Descriptor, Date, CurrentValue')
    end
end
```

Subclass `set` methods call the asset `set` method and require the capability to return the modified object since MATLAB does not support passing arguments by reference. See "The Stock set Method" on page 9-53 for an example.

The Asset subsref Method

The `subsref` method provides access to the data contained in an asset object using one-based numeric indexing and structure field name indexing. The outer `switch` statement determines if the index is a numeric or field name syntax. The inner `switch` statements map the index to the appropriate value.

MATLAB calls `subsref` whenever you make a subscripted reference to an object (e.g., `A(i)`, `A{i}`, or `A.fieldname`).

```
function b = subsref(a,index)
%SUBSREF Define field name indexing for asset objects
switch index.type
case '()'
   switch index.subs{:}
   case 1
      b = a.descriptor;
   case 2
      b = a.date;
   case 3
      b = a.currentValue;
   otherwise
      error('Index out of range')
   end
case '.'
   switch index.subs
   case 'descriptor'
      b = a.descriptor;
   case 'date'
      b = a.date;
   case 'currentValue'
      b = a.currentValue;
```

```
    otherwise
       error('Invalid field name')
    end
 case '{}'
    error('Cell array indexing not supported by asset objects')
 end
```

See the "The Stock subsref Method" on page 9-54 for an example of how the child `subsref` method calls the parent `subsref` method.

The Asset subsasgn Method

The `subsasgn` method is the assignment equivalent of the `subsref` method. This version enables you to change the data contained in an object using one-based numeric indexing and structure field name indexing. The outer `switch` statement determines if the index is a numeric or field name syntax. The inner `switch` statements map the index value to the appropriate value in the stock structure.

MATLAB calls `subsasgn` whenever you execute an assignment statement (e.g., `A(i) = val`, `A{i} = val`, or `A.fieldname = val`).

```
function a = subsasgn(a,index,val)
% SUBSASGN Define index assignment for asset objects
switch index.type
case '()'
   switch index.subs{:}
   case 1
      a.descriptor = val;
   case 2
      a.date = val;
   case 3
      a.currentValue = val;
   otherwise
      error('Index out of
   end
case '.'
   switch index.subs
   case 'descriptor'
      a.descriptor = val;
```

```
      case 'date'
         a.date = val;
      case 'currentValue'
         a.currentValue = val;
      otherwise
         error('Invalid field name')
      end
   end
```

The `subsasgn` method enables you to assign values to the asset object data structure using two techniques. For example, suppose you have a child stock object `s`. (If you want to run this statement, you first need to create a stock constructor method.)

```
s = stock('XYZ',100,25);
```

Within stock class methods, you could change the `descriptor` field with either of the following statements

```
s.asset(1) = 'ABC';
```

or

```
s.asset.descriptor = 'ABC';
```

See the "The Stock subsasgn Method" on page 9-55 for an example of how the child `subsasgn` method calls the parent `subsasgn` method.

The Asset display Method

The asset `display` method is designed to be called from child-class `display` methods. Its purpose is to display the data it stores for the child object. The method simply formats the data for display in a way that is consistent with the formatting of the child's display method.

```
function display(a)
% DISPLAY(a) Display an asset object
stg = sprintf(...
   'Descriptor: %s\nDate: %s\nType: %s\nCurrent Value:%9.2f',...
   a.descriptor,a.date,a.type,a.currentValue);
disp(stg)
```

The stock class display method can now call this method to display the data stored in the parent class. This approach isolates the stock `display` method from changes to the asset class. See "The Stock display Method" on page 9-57 for an example of how this method is called.

The Asset fieldcount Method

The asset `fieldcount` method returns the number of fields in the asset object data structure. `fieldcount` enables asset child methods to determine the number of fields in the asset object during execution, rather than requiring the child methods to have knowledge of the asset class. This allows you to make changes to the number of fields in the asset class data structure without having to change child-class methods.

```
function numFields = fieldcount(assetObj)
% Determines the number of fields in an asset object
% Used by asset child class methods
numFields = length(fieldnames(assetObj));
```

The `struct` function converts an object to its equivalent data structure, enabling access to the structure's contents.

Designing the Stock Class

A stock object is designed to represent one particular asset in a person's investment portfolio. This object contains two properties of its own and inherits three properties from its parent asset object.

Stock properties:

- `NumberShares` — The number of shares for the particular stock object.
- `SharePrice` — The value of each share.

Asset properties:

- `Descriptor` — The identifier of the particular asset (e.g., stock name, savings account number, etc.).
- `Date` — The date the object was created (calculated by the `date` command).
- `CurrentValue` — The current value of the asset.

Note that the property names are not actually the same as the field names of the structure array used internally by stock and asset objects. The property name interface is controlled by the stock and asset `set` and `get` methods and is designed to resemble the interface of other MATLAB object properties.

The `asset` field in the stock object structure contains the parent asset object and is used to access the inherited fields in the parent structure.

Stock Class Methods

The stock class implements the following methods:

- Constructor
- `get` and `set`
- `subsref` and `subsasgn`
- `display`

The Stock Constructor Method

The stock constructor creates a stock object from three input arguments:

- The stock name
- The number of shares
- The share price

The constructor must create an asset object from within the stock constructor to be able to specify it as a parent to the stock object. The stock constructor must, therefore, call the asset constructor. The `class` function, which is called to create the stock object, defines the asset object as the parent.

Keep in mind that the asset object is created in the temporary workspace of the stock constructor function and is stored as a field (`.asset`) in the stock structure. The stock object inherits the asset fields, but the asset object is not returned to the base workspace.

```
function s = stock(varargin)
% STOCK Stock class constructor.
% s = stock(descriptor, numShares, sharePrice)
```

```
switch nargin
case 0
   % if no input arguments, create a default object
   s.numShares = 0;
   s.sharePrice = 0;
   a = asset('none',0);
   s = class(s,'stock',a);
case 1
   % if single argument of class stock, return it
   if (isa(varargin{1},'stock'))
      s = varargin{1};
   else
      error('Input argument is not a stock object')
   end
case 3
   % create object using specified values
   s.numShares = varargin{2};
   s.sharePrice = varargin{3};
   a = asset(varargin{1},'stock',varargin{2} * varargin{3});
   s = class(s,'stock',a);
otherwise
   error('Wrong number of input arguments')
end
```

Constructor Calling Syntax

The stock constructor method can be called in one of three ways:

- No input argument — If called with no arguments, the constructor returns a default object with empty fields.
- Input argument is a stock object — If called with a single input argument that is a stock object, the constructor returns the input argument. A single argument that is not a stock object generates an error.
- Three input arguments — If there are three input arguments, the constructor uses them to define the stock object.

Otherwise, if none of the above three conditions are met, return an error.

For example, this statement creates a stock object to record the ownership of 100 shares of XYZ corporation stocks with a price per share of 25 dollars.

```
XYZStock = stock('XYZ',100,25);
```

The Stock get Method

The `get` method provides a way to access the data in the stock object using a "property name" style interface, similar to Handle Graphics. While in this example the property names are similar to the structure field name, they can be quite different. You could also choose to exclude certain fields from access via the `get` method or return the data from the same field for a variety of property names, if such behavior suits your design.

```
function val = get(s,propName)
% GET Get stock property from the specified object
% and return the value. Property names are: NumberShares
% SharePrice, Descriptor, Date, CurrentValue
switch propName
case 'NumberShares'
   val = s.numShares;
case 'SharePrice'
   val = s.sharePrice;
case 'Descriptor'
   val = get(s.asset,'Descriptor'); % call asset get method
case 'Date'
   val = get(s.asset,'Date');
case 'CurrentValue'
   val = get(s.asset,'CurrentValue');
otherwise
   error([propName ,'Is not a valid stock property'])
end
```

Note that the asset object is accessed via the stock object's asset field (`s.asset`). MATLAB automatically creates this field when the `class` function is called with the parent argument.

The Stock set Method

The `set` method provides a "property name" interface like the `get` method. It is designed to update the number of shares, the share value, and the descriptor. The current value and the date are automatically updated.

```
function s = set(s,varargin)
% SET Set stock properties to the specified values
% and return the updated object
propertyArgIn = varargin;
while length(propertyArgIn) >= 2,
   prop = propertyArgIn{1};
   val = propertyArgIn{2};
   propertyArgIn = propertyArgIn(3:end);
   switch prop
   case 'NumberShares'
      s.numShares = val;
   case 'SharePrice'
      s.sharePrice = val;
   case 'Descriptor'
      s.asset = set(s.asset,'Descriptor',val);
   otherwise
      error('Invalid property')
   end
end
s.asset = set(s.asset,'CurrentValue',...
   s.numShares * s.sharePrice,'Date',date);
```

Note that this function creates and returns a new stock object with the new values, which you then copy over the old value. For example, given the stock object,

```
s = stock('XYZ',100,25);
```

the following `set` command updates the share price.

```
s = set(s,'SharePrice',36);
```

It is necessary to copy over the original stock object (i.e., assign the output to `s`) because MATLAB does not support passing arguments by reference. Hence the `set` method actually operates on a copy of the object.

The Stock subsref Method

The `subsref` method defines subscripted indexing for the stock class. In this example, `subsref` is implemented to enable numeric and structure field name indexing of stock objects.

```
function b = subsref(s,index)
% SUBSREF Define field name indexing for stock objects
fc = fieldcount(s.asset);
switch index.type
case '()'
   if (index.subs{:} <= fc)
      b = subsref(s.asset,index);
   else
      switch index.subs{:} - fc
      case 1
         b = s.numShares;
      case 2
         b = s.sharePrice;
      otherwise
         error(['Index must be in the range 1 to ', ...
            num2str(fc + 2)])
      end
   end
case '.'
   switch index.subs
   case 'numShares'
      b = s.numShares;
   case 'sharePrice'
      b = s.sharePrice;
   otherwise
      b = subsref(s.asset,index);
   end
end
```

The outer `switch` statement determines if the index is a numeric or field name syntax.

The `fieldcount` asset method determines how many fields there are in the asset structure, and the `if` statement calls the asset `subsref` method for

indices 1 to `fieldcount`. See "The Asset fieldcount Method" on page 9-49 and "The Asset subsref Method" on page 9-46 for a description of these methods.

Numeric indices greater than the number returned by `fieldcount` are handled by the inner `switch` statement, which maps the index value to the appropriate field in the stock structure.

Field name indexing assumes field names other than `numShares` and `sharePrice` are asset fields, which eliminates the need for knowledge of asset fields by child methods. The asset `subsref` method performs field-name error checking.

See the `subsref` help entry for general information on implementing this method.

The Stock subsasgn Method

The `subsasgn` method enables you to change the data contained in a stock object using numeric indexing and structure field name indexing. MATLAB calls `subsasgn` whenever you execute an assignment statement (e.g., `A(i) = val`, `A{i} = val`, or `A.fieldname = val`).

```
function s = subsasgn(s,index,val)
% SUBSASGN Define index assignment for stock objects
fc = fieldcount(s.asset);
switch index.type
case '()'
   if (index.subs{:} <= fc)
      s.asset = subsasgn(s.asset,index,val);
   else
      switch index.subs{:}-fc
      case 1
         s.numShares = val;
      case 2
         s.sharePrice = val;
      otherwise
         error(['Index must be in the range 1 to ', ...
            num2str(fc + 2)])
      end
   end
```

```
case '.'
   switch index.subs
   case 'numShares'
      s.numShares = val;
   case 'sharePrice'
      s.sharePrice = val;
   otherwise
      s.asset = subsasgn(s.asset,index,val);
   end
end
```

The outer `switch` statement determines if the index is a numeric or field name syntax.

The `fieldcount` asset method determines how many fields there are in the asset structure and the `if` statement calls the asset `subsasgn` method for indices `1` to `fieldcount`. See "The Asset fieldcount Method" on page 9-49 and "The Asset subsasgn Method" on page 9-47 for a description of these methods.

Numeric indices greater than the number returned by `fieldcount` are handled by the inner `switch` statement, which maps the index value to the appropriate field in the stock structure.

Field name indexing assumes field names other than `numShares` and `sharePrice` are asset fields, which eliminates the need for knowledge of asset fields by child methods. The asset `subsasgn` method performs field-name error checking.

The `subsasgn` method enables you to assign values to stock object data structure using two techniques. For example, suppose you have a stock object

```
s = stock('XYZ',100,25)
```

You could change the `descriptor` field with either of the following statements

```
s(1) = 'ABC';
```

or

```
s.descriptor = 'ABC';
```

See the subsasgn help entry for general information on assignment statements in MATLAB.

The Stock display Method

When you issue the statement (without terminating with a semicolon)

```
XYZStock = stock('XYZ',100,25)
```

MATLAB looks for a method in the @stock directory called display. The display method for the stock class produces this output.

```
Descriptor: XYZ
Date: 17-Nov-1998
Type: stock
Current Value:  2500.00
Number of shares: 100
Share price: 25.00
```

Here is the stock display method.

```
function display(s)
% DISPLAY(s) Display a stock object
display(s.asset)
stg = sprintf('Number of shares: %g\nShare price: %3.2f\n',...
   s.numShares,s.sharePrice);
disp(stg)
```

First, the parent asset object is passed to the asset display method to display its fields (MATLAB calls the asset display method because the input argument is an asset object). The stock object's fields are displayed in a similar way using a formatted text string.

Note that if you did not implement a stock class display method, MATLAB would call the asset display method. This would work, but would display only the descriptor, date, type, and current value.

Example — The Portfolio Container

In this section...

Overview

Aggregation is the containment of one class by another class. The basic relationship is: each contained class "is a part of" the container class.

For example, consider a financial portfolio class as a container for a set of assets (stocks, bonds, savings, etc.). Once the individual assets are grouped, they can be analyzed, and useful information can be returned. The contained objects are not accessible directly, but only via the portfolio class methods.

See "Example — Assets and Asset Subclasses" on page 9-41 for information about the assets collected by this portfolio class.

Designing the Portfolio Class

The portfolio class is designed to contain the various assets owned by a given individual and provide information about the status of his or her investment portfolio. This example implements a somewhat over-simplified portfolio class that

- Contains an individual's assets
- Displays information about the portfolio contents
- Displays a 3-D pie chart showing the relative mix of asset types in the portfolio

Required Portfolio Methods

The portfolio class implements only three methods:

- `portfolio` — The portfolio constructor.
- `display` — Displays information about the portfolio contents.
- `pie3` — Overloaded version of `pie3` function designed to take a single portfolio object as an argument.

Since a portfolio object contains other objects, the portfolio class methods can use the methods of the contained objects. For example, the portfolio `display` method calls the stock class `display` method, and so on.

The Portfolio Constructor Method

The portfolio constructor method takes as input arguments a client's name and a variable length list of asset subclass objects (stock, bond, and savings objects in this example). The portfolio object uses a structure array with the following fields:

- `name` — The client's name.
- `indAssets` — The array of asset subclass objects (stock, bond, savings).
- `totalValue` — The total value of all assets. The constructor calculates this value from the objects passed in as arguments.
- `accountNumber` — The account number. This field is assigned a value only when you save a portfolio object (see "Saving and Loading Objects" on page 9-64).

```
function p = portfolio(name,varargin)
% PORTFOLIO Create a portfolio object containing the
% client's name and a list of assets
switch nargin
case 0
   % if no input arguments, create a default object
   p.name = 'none';
   p.totalValue = 0;
   p.indAssets = {};
```

```
        p.accountNumber = '';
        p = class(p,'portfolio');
    case 1
        % if single argument of class portfolio, return it
        if isa(name,'portfolio')
            p = name;
        else
            disp([inputname(1) ' is not a portfolio object'])
            return
        end
    otherwise
        % create object using specified arguments
        p.name = name;
        p.totalValue = 0;
        for k = 1:length(varargin)
            p.indAssets(k) = {varargin{k}};
            assetValue = get(p.indAssets{k},'CurrentValue');
            p.totalValue = p.totalValue + assetValue;
        end
        p.accountNumber = '';
        p = class(p,'portfolio');
    end
```

Constructor Calling Syntax

The portfolio constructor method can be called in one of three different ways:

- No input arguments — If called with no arguments, it returns an object with empty fields.
- Input argument is an object — If the input argument is already a portfolio object, MATLAB returns the input argument. The `isa` function checks for this case.
- More than two input arguments — If there are more than two input arguments, the constructor assumes the first is the client's name and the rest are asset subclass objects. A more thorough implementation would perform more careful input argument checking, for example, using the `isa` function to determine if the arguments are the correct class of objects.

The Portfolio display Method

The portfolio `display` method lists the contents of each contained object by calling the object's `display` method. It then lists the client name and total asset value.

```
function display(p)
% DISPLAY Display a portfolio object
for k = 1:length(p.indAssets)
   display(p.indAssets{k})
end
stg = sprintf('\nAssets for Client: %s\nTotal Value: %9.2f\n',...
p.name,p.totalValue);
disp(stg)
```

The Portfolio pie3 Method

The portfolio class overloads the MATLAB `pie3` function to accept a portfolio object and display a 3-D pie chart illustrating the relative asset mix of the client's portfolio. MATLAB calls the `@portfolio/pie3.m` version of `pie3` whenever the input argument is a single portfolio object.

```
function pie3(p)
% PIE3 Create a 3-D pie chart of a portfolio
stockAmt = 0; bondAmt = 0; savingsAmt = 0;
for k = 1:length(p.indAssets)
   if isa(p.indAssets{k}, 'stock')
      stockAmt = stockAmt + ...
         get(p.indAssets{k}, 'CurrentValue');
   elseif isa(p.indAssets{k}, 'bond')
      bondAmt = bondAmt + ...
         get(p.indAssets{k}, 'CurrentValue');
   elseif isa(p.indAssets{k}, 'savings')
      savingsAmt = savingsAmt + ...
         get(p.indAssets{k}, 'CurrentValue');
   end
end
i = 1;
if stockAmt ~= 0
   label(i) = {'Stocks'};
   pieVector(i) = stockAmt;
   i = i + 1;
```

```
end
if bondAmt ~= 0
   label(i) = {'Bonds'};
   pieVector(i) = bondAmt;
   i = i + 1;
end
if savingsAmt ~= 0
   label(i) = {'Savings'};
   pieVector(i) = savingsAmt;
end
pie3(pieVector, label)
set(gcf, 'Renderer', 'zbuffer')
set(findobj(gca, 'Type', 'Text'), 'FontSize', 14)
cm = gray(64);
colormap(cm(48:end, :))
stg(1) = {['Portfolio Composition for ', p.name]};
stg(2) = {['Total Value of Assets: $', num2str(p.totalValue)]};
title(stg, 'FontSize', 12)
```

There are three parts in the overloaded `pie3` method.

- The first uses the asset subclass `get` methods to access the `CurrentValue` property of each contained object. The total value of each class is summed.
- The second part creates the pie chart labels and builds a vector of graph data, depending on which objects are present.
- The third part calls the MATLAB `pie3` function, makes some font and colormap adjustments, and adds a title.

Creating a Portfolio

Suppose you have implemented a collection of asset subclasses in a manner similar to the stock class. You can then use a portfolio object to present the individual's financial portfolio. For example, given the following assets

```
XYZStock = stock('XYZ', 200, 12);
SaveAccount = savings('Acc # 1234', 2000, 3.2);
Bonds = bond('U.S. Treasury', 1600, 12);
```

create a portfolio object:

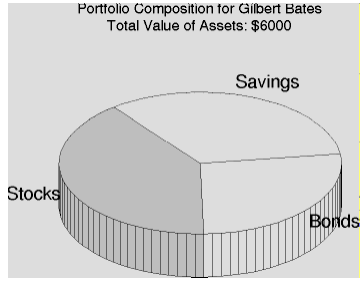
```
p = portfolio('Gilbert Bates',XYZStock,SaveAccount,Bonds)
```

The portfolio `display` method summarizes the portfolio contents (because this statement is not terminated by a semicolon).

```
Descriptor: XYZ
Date: 24-Nov-1998
Current Value:   2400.00
Type: stock
Number of shares: 200
Share price: 12.00
Descriptor: Acc # 1234
Date: 24-Nov-1998
Current Value:   2000.00
Type: savings
Interest Rate: 3.2%
Descriptor: U.S. Treasury
Date: 24-Nov-1998
Current Value:   1600.00
Type: bond
Interest Rate: 12%
Assets for Client: Gilbert Bates
Total Value:   6000.00
```

The portfolio `pie3` method displays the relative mix of assets using a pie chart.

```
pie3(p)
```



Saving and Loading Objects

You can use the MATLAB `save` and `load` commands to save and retrieve user-defined objects to and from `.mat` files, just like any other variables.

When you load objects, MATLAB calls the object's class constructor to register the object in the workspace. The constructor function for the object class you are loading must be able to be called with no input arguments and return a default object. See "Guidelines for Writing a Constructor" on page 9-10 for more information.

When you issue a `save` or `load` command on objects, MATLAB looks for class methods called `saveobj` and `loadobj` in the class directory. You can overload these methods to modify the object before the save or load operation. For example, you could define a `saveobj` method that saves related data along with the object or you could write a `loadobj` method that updates objects to a newer version when this type of object is loaded into the MATLAB workspace.

Example — Defining saveobj and loadobj for Portfolio

In this section...

Methods Executed by Save and Load

In the section "Example — The Portfolio Container" on page 9-58, portfolio objects are used to collect information about a client's investment portfolio. Suppose you decide to add an account number to each portfolio object that is saved. You can define a portfolio `saveobj` method to carry out this task automatically during the save operation.

Suppose further that you have already saved a number of portfolio objects without the account number. You want to update these objects during the load operation so that they are still valid portfolio objects. You can do this by defining a `loadobj` method for the portfolio class.

Summary of Code Changes

To implement the account number scenario, you need to add or change the following functions:

- `portfolio` — The portfolio constructor method needs to be modified to create a new field, `accountNumber`, which is initialized to the empty string when an object is created.
- `saveobj` — A new portfolio method designed to add an account number to a portfolio object during the save operation, only if the object does not already have one.

- `loadobj` — A new portfolio method designed to update older versions of portfolio objects that were saved before the account number structure field was added.
- `subsref` — A new portfolio method that enables subscripted reference to portfolio objects outside of a portfolio method.
- `getAccountNumber` — a MATLAB function that returns an account number that consists of the first three letters of the client's name.

New Portfolio Class Behavior

With the additions and changes made in this example, the portfolio class now

- Includes a field for an account number
- Adds the account number when a portfolio object is saved for the first time
- Automatically updates the older version of portfolio objects when you load them into the MATLAB workspace

The saveobj Method

MATLAB looks for the portfolio `saveobj` method whenever the `save` command is passed a portfolio object. If `@portfolio/saveobj` exists, MATLAB passes the portfolio object to `saveobj`, which must then return the modified object as an output argument. The following implementation of `saveobj` determines if the object has already been assigned an account number from a previous save operation. If not, `saveobj` calls `getAccountNumber` to obtain the number and assigns it to the `accountNumber` field.

```
function b = saveobj(a)
if isempty(a.accountNumber)
   a.accountNumber = getAccountNumber(a);
end
b = a;
```

The loadobj Method

MATLAB looks for the portfolio `loadobj` method whenever the `load` command detects portfolio objects in the `.mat` file being loaded. If `loadobj` exists, MATLAB passes the portfolio object to `loadobj`, which must then return the modified object as an output argument. The output argument is then loaded into the workspace.

If the input object does not match the current definition as specified by the constructor function, then MATLAB converts it to a structure containing the same fields and the object's structure with all the values intact (that is, you now have a structure, not an object).

The following implementation of `loadobj` first uses `isa` to determine whether the input argument is a portfolio object or a structure. If the input is an object, it is simply returned since no modifications are necessary. If the input argument has been converted to a structure by MATLAB, then the new `accountNumber` field is added to the structure and is used to create an updated portfolio object.

```
function b = loadobj(a)
% loadobj for portfolio class
if isa(a,'portfolio')
   b = a;
else % a is an old version
   a.accountNumber = getAccountNumber(a);
   b = class(a,'portfolio');
end
```

Changing the Portfolio Constructor

The portfolio structure array needs an additional field to accommodate the account number. To create this field, add the line

```
p.accountNumber = '';
```

to `@portfolio/portfolio.m` in both the zero argument and variable argument sections.

The getAccountNumber Function

In this example, `getAccountNumber` is a MATLAB function that returns an account number composed of the first three letters of the client name prepended to a series of digits. To illustrate implementation techniques, `getAccountNumber` is not a portfolio method so it cannot access the portfolio object data directly. Therefore, it is necessary to define a portfolio `subsref` method that enables access to the name field in a portfolio object's structure.

For this example, `getAccountNumber` simply generates a random number, which is formatted and concatenated with elements `1` to `3` from the portfolio `name` field.

```
function n = getAccountNumber(p)
% provides a account number for object p
n = [upper(p.name(1:3)) strcat(num2str(round(rand(1,7)*10))')'];
```

Note that the portfolio object is indexed by field name, and then by numerical subscript to extract the first three letters. The `subsref` method must be written to support this form of subscripted reference.

The Portfolio subsref Method

When MATLAB encounters a subscripted reference, such as that made in the `getAccountNumber` function

```
p.name(1:3)
```

MATLAB calls the portfolio `subsref` method to interpret the reference. If you do not define a `subsref` method, the above statement is undefined for portfolio objects (recall that here `p` is an object, not just a structure).

The portfolio `subsref` method must support field-name and numeric indexing for the `getAccountNumber` function to access the portfolio `name` field.

```
function b = subsref(p,index)
% SUBSREF Define field name indexing for portfolio objects
switch index(1).type
case '.'
   switch index(1).subs
   case 'name'
      if length(index)== 1
         b = p.name;
      else
         switch index(2).type
         case '()'
            b = p.name(index(2).subs{:});
         end
      end
   end
```

```
end
```

Note that the portfolio implementation of `subsref` is designed to provide access to specific elements of the `name` field; it is not a general implementation that provides access to all structure data, such as the stock class implementation of `subsref`.

See the `subsref` help entry for more information about indexing and objects.

Object Precedence

In this section...

How MATLAB Determines Precedence

Object precedence is a means to resolve the question of which of possibly many versions of an operator or function to call in a given situation. Object precedence enables you to control the behavior of expressions containing different classes of objects. For example, consider the expression

```
objectA + objectB
```

Ordinarily, MATLAB assumes that the objects have equal precedence and calls the method associated with the leftmost object. However, there are two exceptions:

- User-defined classes have precedence over MATLAB built-in classes.
- User-defined classes can specify their relative precedence with respect to other user-defined classes using the `inferiorto` and `superiorto` functions.

For example, in the section "Example — A Polynomial Class" on page 9-26 the polynom class defines a `plus` method that enables addition of polynom objects. Given the polynom object `p`

```
p = polynom([1 0 -2 -5])
p =
    x^3-2*x-5
```

The expression,

```
1 + p
ans =
    x^3-2*x-4
```

calls the polynom `plus` method (which converts the double, 1, to a polynom object, and then adds it to `p`). The user-defined polynom class has precedence over the MATLAB double class.

Specifying Precedence of User-Defined Classes

You can specify the relative precedence of user-defined classes by calling the `inferiorto` or `superiorto` function in the class constructor.

The `inferiorto` function places a class below other classes in the precedence hierarchy. The calling syntax for the `inferiorto` function is

```
inferiorto('class1','class2',...)
```

You can specify multiple classes in the argument list, placing the class below many other classes in the hierarchy.

Similarly, the `superiorto` function places a class above other classes in the precedence hierarchy. The calling syntax for the `superiorto` function is

```
superiorto('class1','class2',...)
```

Location in the Hierarchy

If *`objectA`* is above *`objectB`* in the precedence hierarchy, then the expression

```
objectA + objectB
```

calls `@`*`classA`*`/plus.m`. Conversely, if *`objectB`* is above *`objectA`* in the precedence hierarchy, then MATLAB calls `@`*`classB`*`/plus.m`.

See “How MATLAB Determines Which Method to Call” on page 9-72 for related information.

How MATLAB Determines Which Method to Call

In this section...

Overview

In MATLAB, functions exist in directories in the computer's file system. A directory may contain many functions (M-files). Function names are unique only within a single directory (e.g., more than one directory may contain a function called `pie3`). When you type a function name on the command line, MATLAB must search all the directories it is aware of to determine which function to call. This list of directories is called the *MATLAB path*.

When looking for a function, MATLAB searches the directories in the order they are listed in the path, and calls the first function whose name matches the name of the specified function.

If you write an M-file called `pie3.m` and put it in a directory that is searched before the `specgraph` directory that contains the MATLAB `pie3` function, then MATLAB uses your `pie3` function instead.

Object-oriented programming allows you to have many methods (MATLAB functions located in class directories) with the same name and enables MATLAB to determine which method to use based on the type or class of the variables passed to the function. For example, if `p` is a portfolio object, then

```
pie3(p)
```

calls `@portfolio/pie3.m` because the argument is a portfolio object.

Selecting a Method

When you call a method for which there are multiple versions with the same name, MATLAB determines the method to call by:

- Looking at the classes of the objects in the argument list to determine which argument has the highest object precedence; the class of this object controls the method selection and is called the *dispatch type*.
- Applying the *function precedence order* to determine which of possibly several implementations of a method to call. This order is determined by the location and type of function.

Determining the Dispatch Type

MATLAB first determines which argument controls the method selection. The class type of this argument then determines the class in which MATLAB searches for the method. The controlling argument is either

- The argument with the highest precedence, or
- The leftmost of arguments having equal precedence

User-defined objects take precedence over the MATLAB built-in classes such as `double` or `char`. You can set the relative precedence of user-defined objects with the `inferiorto` and `superiorto` functions, as described in "Object Precedence" on page 9-70.

MATLAB searches for functions by name. When you call a function, MATLAB knows the name, number of arguments, and the type of each argument. MATLAB uses the dispatch type to choose among multiple functions of the same name, but does not consider the number of arguments.

Function Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. MATLAB selects the correct function for a given context by applying the following function precedence rules, in the order given:

1 Subfunctions

Subfunctions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the subfunction and ignores the overloaded method.

2 Private functions

   Private functions are called if there is no subfunction of the same name within the current scope. As with subfunctions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method.

3 Class constructor functions

   Constructor functions (functions having names that are the same as the @ directory, for example `@polynom/polynom.m`) take precedence over other MATLAB functions. Therefore, if you create an M-file called `polynom.m` and put it on your path before the constructor `@polynom/polynom.m` version, MATLAB will always call the constructor version.

4 Overloaded methods

   MATLAB calls an overloaded method if it is not masked by a subfunction or private function.

5 Current directory

   A function in the current working directory is selected before one elsewhere on the path.

6 Elsewhere on path

   Finally, a function anywhere else on the path is selected.

Selecting Methods from Multiple Directories

There may be a number of directories on the path that contain methods with the same name. MATLAB stops searching when it finds the first implementation of the method on the path, regardless of the implementation type (MEX-file, P-code, M-file).

Selecting Methods from Multiple Implementation Types

There are five file precedence types. MATLAB uses file precedence to select between identically named functions in the same directory. The order of precedence for file types is

1. Built-in file
2. MEX-files
3. MDL (Simulink model) file
4. P-code file
5. M-file

For example, if MATLAB finds a P-code and an M-file version of a method in a class directory, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the M-file.

Querying Which Method MATLAB Will Call

You can determine which method MATLAB will call using the `which` command. For example,

```
which pie3
your_matlab_path/toolbox/matlab/specgraph/pie3.m
```

However, if `p` is a portfolio object,

```
which pie3(p)
dir_on_your_path/@portfolio/pie3.m      % portfolio method
```

The `which` command determines which version of `pie3` MATLAB will call if you passed a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use the `-all` option. See the `which` reference page for more information on this command.

10

Scheduling Program Execution with Timers

Using a MATLAB Timer Object

In this section...

Overview

MATLAB includes a timer object that you can use to schedule the execution of MATLAB commands. This section describes how you can create timer objects, start a timer running, and specify the processing that you want performed when a timer fires. A timer is said to *fire* when the amount of time specified by the timer object elapses and the timer object executes the commands you specify.

To use a timer, perform these steps:

1. Create a timer object.

   You use the `timer` function to create a timer object. See "Creating Timer Objects" on page 10-5 for more information.

2. Specify which MATLAB commands you want executed when the timer fires and control other aspects of timer object behavior.

   You use timer object properties to specify this information. To learn about all the properties supported by the timer object, see "Working with Timer Object Properties" on page 10-7. (You can also set timer object properties when you create them, in step 1.)

3. Start the timer object.

   After you create the timer object, you must start it, using either the `start` or `startat` function. See "Starting and Stopping Timers" on page 10-10 for more information.

4. Delete the timer object when you are done with it.

After you are finished using a timer object, you should delete it from memory. See “Deleting Timer Objects from Memory” on page 10-23 for more information.

Note The specified execution time and the actual execution of a timer can vary because timer objects work in the MATLAB single-threaded execution environment. The length of this time lag is dependent on what other processing MATLAB is performing. To force the execution of the callback functions in the event queue, include a call to the `drawnow` function in your code. The `drawnow` function flushes the event queue.

Example: Displaying a Message

The following example sets up a timer object that executes a MATLAB command string after 10 seconds elapse. The example creates a timer object, specifying the values of two timer object properties, `TimerFcn` and `StartDelay`. `TimerFcn` specifies the timer callback function. This is the MATLAB command string or M-file that you want to execute when the timer fires. In the example, the timer callback function sets the value of the MATLAB workspace variable `stat` and executes the MATLAB `disp` command. The `StartDelay` property specifies how much time elapses before the timer fires.

After creating the timer object, the example uses the `start` function to start the timer object. (The additional commands in this example are included to illustrate the timer but are not required for timer operation.)

```
t = timer('TimerFcn', 'stat=false; disp(''Timer!'')',...
                  'StartDelay',10);
start(t)

stat=true;
while(stat==true)
  disp('.')
  pause(1)
end
```

When you execute this code, it produces this output:

```
.
.
.
.
.
.
.
.
.
Timer!

delete(t) % Always delete timer objects after using them.
```

Creating Timer Objects

In this section...

Creating the Object

To use a timer in MATLAB, you must create a timer object. The timer object represents the timer in MATLAB, supporting various properties and functions that control its behavior.

To create a timer object, use the `timer` function. This creates a valid timer object with default values for most properties. The following shows an example of the default timer object and its summary display:

```
t = timer
Timer Object: timer-1

   Timer Settings
      ExecutionMode: singleShot
              Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: ''
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''
```

MATLAB names the timer object `timer-1`. (See “Naming the Object” on page 10-6 for more information.)

To specify the value of timer object properties after you create it, you can use the `set` function. This example sets the value of the `TimerFcn` property and the `StartDelay` property. For more information about timer object properties, see “Working with Timer Object Properties” on page 10-7.

```
set(t,'TimerFcn','disp(''Hello World!'')','StartDelay',5)
```

You can also set timer object properties when you create the timer object by specifying property name and value pairs as arguments to the `timer` function. The following example sets the same properties at object creation time:

```
t = timer('TimerFcn', 'disp(''Hello World!'')','StartDelay',5);
```

Always delete timer objects when you are done using them. See "Deleting Timer Objects from Memory" on page 10-23 for more information.

Naming the Object

MATLAB assigns a name to each timer object you create. This name has the form `'timer-`*`i`*`'`, where *`i`* is a number representing the total number of timer objects created this session.

For example, the first time you call the timer function to create a timer object, MATLAB names the object `timer-1`. If you call the timer function again to create another timer object, MATLAB names the object `timer-2`.

MATLAB keeps incrementing the number associated with each timer object it creates, even if you delete the timer objects you already created. For example, if you delete the first two timer objects and create a new object, MATLAB names it `timer-3`, even though the other two timer objects no longer exist in memory. To reset the numeric part of timer object names to 1, execute the `clear classes` command.

Working with Timer Object Properties

In this section...

To get information about timer object properties, see the `timer` function reference page.

Retrieving the Value of Timer Object Properties

The timer object supports many properties that provide information about the current state of the timer object and control aspects of its functioning. To retrieve the value of a timer object property, you can use the `get` function or use subscripts (dot notation) to access the field.

The following example uses the `set` function to retrieve the value of the `ExecutionMode` property:

```
t = timer;

tmode = get(t,'ExecutionMode')

tmode =

singleShot
```

The following example uses dot notation to retrieve the value of the `ExecutionMode` property:

```
tmode = t.ExecutionMode

tmode =

singleShot
```

To view a list of all the properties of a timer object, use the `get` function, specifying the timer object as the only argument:

```
get(t)
       AveragePeriod: NaN
            BusyMode: 'drop'
            ErrorFcn: ''
       ExecutionMode: 'singleShot'
       InstantPeriod: NaN
                Name: 'timer-4'
    ObjectVisibility: 'on'
              Period: 1
             Running: 'off'
          StartDelay: 0
            StartFcn: ''
             StopFcn: ''
                 Tag: ''
       TasksExecuted: 0
      TasksToExecute: Inf
            TimerFcn: ''
                Type: 'timer'
            UserData: []
```

Setting the Value of Timer Object Properties

To set the value of a timer object property, use the `set` function or subscripted assignment (dot notation). You can also set timer object properties when you create the timer object. For more information, see "Creating Timer Objects" on page 10-5.

The following example uses both methods to assign values to timer object properties. The example creates a timer that, once started, displays a message every second until you stop it with the `stop` command.

1 Create a timer object.

```
t = timer;
```

2 Assign values to timer object properties using the `set` function.

```
set(t,'ExecutionMode','fixedRate','BusyMode','drop','Period',1);
```

3 Assign a value to the timer object `TimerFcn` property using dot notation.

```
t.TimerFcn = 'disp(''Processing...'')'
```

4 Start the timer object. It displays a message at 1-second intervals.

```
start(t)
```

5 Stop the timer object.

```
stop(t)
```

6 Delete timer objects after you are done using them.

```
delete(t)
```

Viewing a List of All Settable Properties

To view a list of all timer object properties that can have values assigned to them (in contrast to the read-only properties), use the `set` function, specifying the timer object as the only argument.

The display includes the values you can use to set the property if, like the `BusyMode` property, the property accepts an enumerated list of values.

```
t = timer;

set(t)
   BusyMode: [ {drop} | queue | error ]
   ErrorFcn: string -or- function handle -or- cell array
   ExecutionMode: [{singleShot} | fixedSpacing | fixedDelay | fixedRate]
   Name
   ObjectVisibility: [ {on} | off ]
   Period
   StartDelay
   StartFcn: string -or- function handle -or- cell array
   StopFcn: string -or- function handle -or- cell array
   Tag
   TasksToExecute
   TimerFcn: string -or- function handle -or- cell array
   UserData
```

Starting and Stopping Timers

In this section...

Note Because the timer works within the MATLAB single-threaded environment, it cannot guarantee execution times or execution rates.

Starting a Timer

To start a timer object, call the `start` function, specifying the timer object as the only argument. The `start` function starts a timer object running; the amount of time the timer runs is specified in seconds in the `StartDelay` property.

This example creates a timer object that displays a greeting after 5 seconds elapse.

1 Create a timer object, specifying values for timer object properties.

```
t = timer('TimerFcn','disp(''Hello World!'')','StartDelay', 5);
```

2 Start the timer object.

```
start(t)
```

3 Delete the timer object after you are finished using it.

```
delete(t);
```

Starting a Timer at a Specified Time

To start a timer object and specify a date and time for the timer to fire, (rather than specifying the number of seconds to elapse), use the `startat` function. This function starts a timer object and allows you to specify the date, hour,

minute, and second when you want to the timer to execute. You specify the time as a MATLAB serial date number or as a specially formatted date text string.

This example creates a timer object that displays a message after an hour has elapsed. The `startat` function starts the timer object running and calculates the value of the `StartDelay` property based on the time you specify.

```
t2=timer('TimerFcn','disp(''It has been an hour now.'')');

startat(t2,now+1/24);
```

Stopping Timer Objects

Once started, the timer object stops running if one of the following conditions apply:

- The timer function callback (`TimerFcn`) has been executed the number of times specified in the `TasksToExecute` property.
- An error occurred while executing a timer function callback (`TimerFcn`).

You can also stop a timer object by using the `stop` function, specifying the timer object as the only argument. The following example illustrates stopping a timer object:

1 Create a timer object.

```
t = timer('TimerFcn','disp(''Hello World!'')', ...
          'StartDelay', 100);
```

2 Start it running.

```
start(t)
```

3 Check the state of the timer object after starting it.

```
get(t,'Running')

ans =

on
```

4 Stop the timer using the `stop` command and check the state again. When a timer stops, the value of the `Running` property of the timer object is set to `'off'`.

```
stop(t)

get(t,'Running')

ans =

off
```

5 Delete the timer object when you are finished using it.

```
delete(t)
```

Note The timer object can execute a callback function that you specify when it starts or stops. See "Creating and Executing Callback Functions" on page 10-14.

Blocking the MATLAB Command Line

By default, when you use the `start` or `startat` function to start a timer object, the function returns control to the command line immediately. For some applications, you might prefer to block the command line until the timer fires. To do this, call the `wait` function right after calling the `start` or `startat` function.

1 Create a timer object.

```
t = timer('StartDelay', 5,'TimerFcn', ...
          'disp(''Hello World!'')');
```

2 Start the timer object running.

```
start(t)
```

3 After the `start` function returns, call the `wait` function immediately. The `wait` function blocks the command line until the timer object fires.

```
wait(t)
```

4 Delete the timer object after you are finished using it.

```
delete(t)
```

Creating and Executing Callback Functions

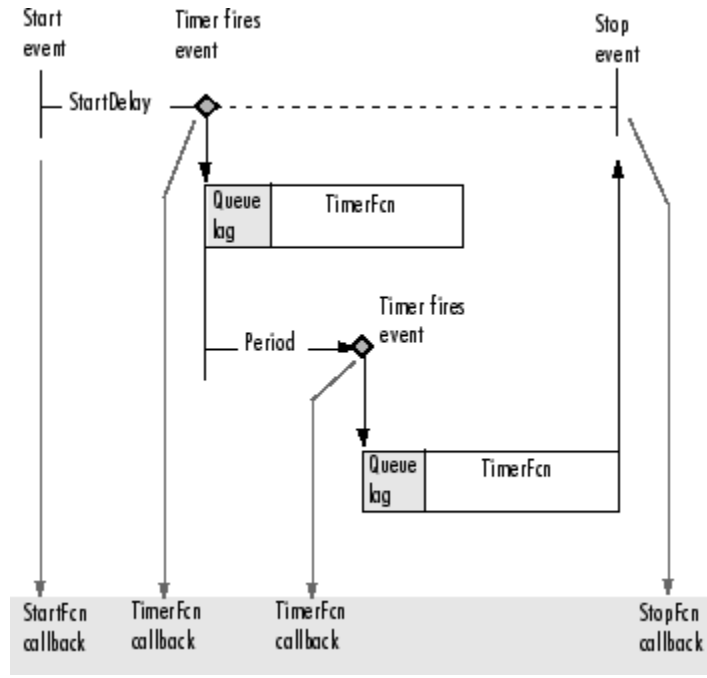
In this section...

Note Callback function execution might be delayed if the callback involves a CPU-intensive task such as updating a figure.

Associating Commands with Timer Object Events

The timer object supports properties that let you specify the MATLAB commands that execute when a timer fires, and for other timer object events, such as starting, stopping, or when an error occurs. These are called *callbacks*. To associate MATLAB commands with a timer object event, set the value of the associated timer object callback property.

The following diagram shows when the events occur during execution of a timer object and give the names of the timer object properties associated with each event. For example, to associate MATLAB commands with a start event, assign a value to the `StartFcn` callback property. Error callbacks can occur at any time.



Timer Object Events and Related Callback Function

Creating Callback Functions

When the time period specified by a timer object elapses, the timer object executes one or more MATLAB functions of your choosing. You can specify the functions directly as the value of the callback property. You can also put the commands in an M-file function and specify the M-file function as the value of the callback property.

Specifying Callback Functions Directly

This example creates a timer object that displays a greeting after 5 seconds. The example specifies the value of the `TimerFcn` callback property directly, putting the commands in a text string.

```
t = timer('TimerFcn','disp(''Hello World!'')','StartDelay',5);
```

Note When you specify the callback commands directly as the value of the callback function property, the commands are evaluated in the MATLAB workspace.

Putting Commands in a Callback Function

Instead of specifying MATLAB commands directly as the value of a callback property, you can put the commands in an M-file and specify the M-file as the value of the callback property.

When you create a callback function, the first two arguments must be a handle to the timer object and an event structure. An event structure contains two fields: `Type` and `Data`. The `Type` field contains a text string that identifies the type of event that caused the callback. The value of this field can be any of the following strings: `'StartFcn'`, `'StopFcn'`, `'TimerFcn'`, or `'ErrorFcn'`. The `Data` field contains the time the event occurred.

In addition to these two required input arguments, your callback function can accept application-specific arguments. To receive these input arguments, you must use a cell array when specifying the name of the function as the value of a callback property. For more information, see “Specifying the Value of Callback Function Properties” on page 10-17.

Example: Writing a Callback Function

This example implements a simple callback function that displays the type of event that triggered the callback and the time the callback occurred. To illustrate passing application-specific arguments, the example callback function accepts as an additional argument a text string and includes this text string in the display output. To see this function used with a callback property, see “Specifying the Value of Callback Function Properties” on page 10-17.

```
function my_callback_fcn(obj, event, string_arg)

txt1 = ' event occurred at ';
txt2 = string_arg;

event_type = event.Type;
event_time = datestr(event.Data.time);
```

```
msg = [event_type txt1 event_time];
disp(msg)
disp(txt2)
```

Specifying the Value of Callback Function Properties

You associate a callback function with a specific event by setting the value of the appropriate callback property. You can specify the callback function as a text string, cell array, or function handle. To access the object and event arguments, you must specify the function as a cell array or as a function handle. If your callback function accepts additional arguments, you must use a cell array.

The following table shows the syntax for several sample callback functions and describes how you call them.

Callback Function Syntax	How to Specify as a Property Value
`function myfile`	`set(h, 'StartFcn', 'myfile')`
`function myfile(obj, event)`	`set(h, 'StartFcn', @myfile)`
`function myfile(obj, event, arg1, arg2)`	`set(h, 'StartFcn', {'myfile', 5, 6})`
`function myfile(obj, event, arg1, arg2)`	`set(h, 'StartFcn', {@myfile, 5, 6})`

This example illustrates several ways you can specify the value of timer object callback function properties, some with arguments and some without. To see the code of the callback function, `my_callback_fcn`, see “Example: Writing a Callback Function” on page 10-16.

1 Create a timer object.

```
t = timer('StartDelay', 4,'Period', 4,'TasksToExecute', 2,...
          'ExecutionMode','fixedRate');
```

2. Specify the value of the `StartFcn` callback. Note that the example specifies the value in a cell array because the callback function needs to access arguments passed to it.

```
t.StartFcn = {'my_callback_fcn', 'My start message'};
```

3. Specify the value of the `StopFcn` callback. The example specifies the callback function by its handle, rather than as a text string. Again, the value is specified in a cell array because the callback function needs to access the arguments passed to it.

```
t.StopFcn = { @my_callback_fcn, 'My stop message'};
```

4. Specify the value of the `TimerFcn` callback. The example specifies the MATLAB commands in a text string.

```
t.TimerFcn = 'disp(''Hello World!'')';
```

5. Start the timer object.

```
start(t)
```

   The example outputs the following.

```
StartFcn event occurred at 10-Mar-2004 17:16:59
Start message
Hello World!
Hello World!
StopFcn event occurred at 10-Mar-2004 17:16:59
Stop message
```

6. Delete the timer object after you are finished with it.

```
delete(t)
```

Timer Object Execution Modes

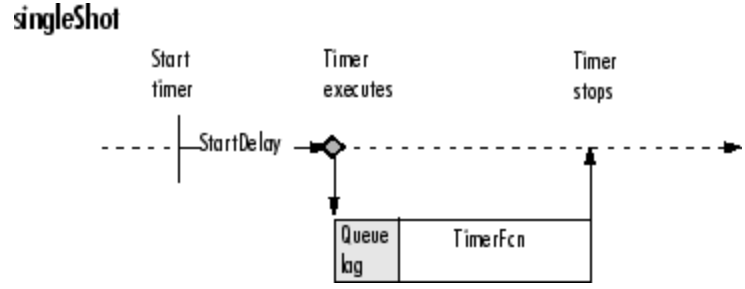
In this section...

Executing a Timer Callback Function Once

The timer object supports several execution modes that determine how it schedules the timer callback function (`TimerFcn`) for execution. You specify the execution mode by setting the value of the `ExecutionMode` property.

To execute a timer callback function once, set the `ExecutionMode` property to `'singleShot'`. This is the default execution mode. In this mode, the timer object starts the timer and, after the time period specified in the `StartDelay` property elapses, adds the timer callback function (`TimerFcn`) to the MATLAB execution queue. When the timer callback function finishes, the timer stops.

The following figure graphically illustrates the parts of timer callback execution for a `singleShot` execution mode. The shaded area in the figure, labelled queue lag, represents the indeterminate amount of time between when the timer adds a timer callback function to the MATLAB execution queue and when the function starts executing. The duration of this lag is dependent on what other processing MATLAB happens to be doing at the time.



Timer Callback Execution (singleShot Execution Mode)

Executing a Timer Callback Function Multiple Times

The timer object supports three multiple-execution modes:

- `'fixedRate'`
- `'fixedDelay'`
- `'fixedSpacing'`

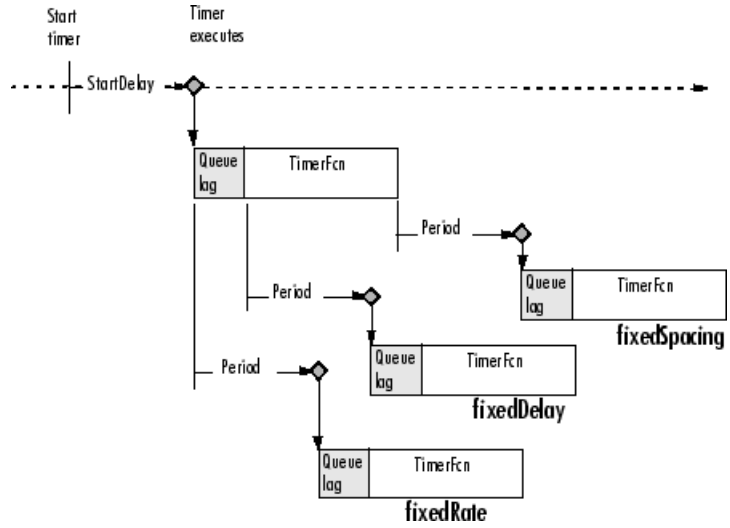
In many ways, these execution modes operate the same:

- The `TasksToExecute` property specifies the number of times you want the timer to execute the timer callback function (`TimerFcn`).
- The `Period` property specifies the amount of time between executions of the timer callback function.
- The `BusyMode` property specifies how the timer object handles queuing of the timer callback function when the previous execution of the callback function has not completed. See "Handling Callback Function Queuing Conflicts" on page 10-21 for more information.

The execution modes differ only in where they start measuring the time period between executions. The following table describes these differences.

Execution Mode	Description
`'fixedRate'`	Time period between executions begins immediately after the timer callback function is added to the MATLAB execution queue.
`'fixedDelay'`	Time period between executions begins when the timer function callback actually starts executing, after any time lag due to delays in the MATLAB execution queue.
`'fixedSpacing'`	Time period between executions begins when the timer callback function finishes executing.

The following figure illustrates the difference between these modes. Note that the amount of time between executions (specified by the `Period` property) remains the same. Only the point at which execution begins is different.



Differences Between Execution Modes

Handling Callback Function Queuing Conflicts

At busy times, in multiple-execution scenarios, the timer may need to add the timer callback function (`TimerFcn`) to the MATLAB execution queue before the previously queued execution of the callback function has completed. You can determine how the timer object handles this scenario by using the `BusyMode` property.

If you specify `'drop'` as the value of the `BusyMode` property, the timer object skips the execution of the timer function callback if the previously scheduled callback function has not already completed.

If you specify `'queue'`, the timer object waits until the currently executing callback function finishes before queuing the next execution of the timer callback function.

Note In `'queue'` mode, the timer object tries to make the average time between executions equal the amount of time specified in the `Period` property. If the timer object has to wait longer than the time specified in the `Period` property between executions of the timer function callback, it shortens the time period for subsequent executions to make up the time.

If the `BusyMode` property is set to `'error'`, the timer object stops and executes the timer object error callback function (`ErrorFcn`), if one is specified.

Deleting Timer Objects from Memory

In this section...

Deleting One or More Timer Objects

When you are finished with a timer object, delete it from memory using the `delete` function:

```
delete(t)
```

When you delete a timer object, workspace variables that referenced the object remain. Deleted timer objects are invalid and cannot be reused. Use the `clear` command to remove workspace variables that reference deleted timer objects.

To remove all timer objects from memory, enter

```
delete(timerfind)
```

For information about the `timerfind` function, see "Finding Timer Objects in Memory" on page 10-24.

Testing the Validity of a Timer Object

To test if a timer object has been deleted, use the `isvalid` function. The `isvalid` function returns logical `0` (`false`) for deleted timer objects:

```
isvalid(t)
ans =

     0
```

Finding Timer Objects in Memory

In this section...

Finding All Timer Objects

To find all the timer objects that exist in memory, use the `timerfind` function. This function returns an array of timer objects. If you leave off the semicolon, and there are multiple timer objects in the array, `timerfind` displays summary information in a table:

```
t1 = timer;
t2 = timer;
t3 = timer;
t_array = timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:    Name:
   1       singleShot      1        ''           timer-3
   2       singleShot      1        ''           timer-4
   3       singleShot      1        ''           timer-5
```

Using `timerfind` to determine all the timer objects that exist in memory can be helpful when deleting timer objects.

Finding Invisible Timer Objects

If you set the value of a timer object's `ObjectVisibility` property to `'off'`, the timer object does not appear in listings of existing timer objects returned by `timerfind`. The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the timer objects created by their application.

Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that created it), you can set its properties. To

retrieve a list of all the timer objects in memory, including invisible ones, use the `timerfindall` function.

11

Improving Performance and Memory Usage

Analyzing Your Program's Performance

In this section...

Overview

The M-file Profiler graphical user interface and the stopwatch timer functions enable you to get back information on how your program is performing and help you identify areas that need improvement. The Profiler can be more useful in measuring relative execution time and in identifying specific performance bottlenecks in your code, while the stopwatch functions tend to be more useful for providing absolute time measurements.

The M-File Profiler Utility

A good first step to speeding up your programs is to find out where the bottlenecks are. This is where you need to concentrate your attention to optimize your code.

MATLAB provides the M-file Profiler, a graphical user interface that shows you where your program is spending its time during execution. Use the Profiler to help you determine where you can modify your code to make performance improvements.

To start the Profiler, type `profile viewer` or select **Desktop > Profiler** in the MATLAB Command Window. See Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation, and the `profile` function reference page.

Stopwatch Timer Functions

If you just need to get an idea of how long your program (or a portion of it) takes to run, or to compare the speed of different implementations of a program, you can use the stopwatch timer functions, `tic` and `toc`. Invoking

`tic` starts the timer, and the first subsequent `toc` stops it and reports the time elapsed between the two.

Use `tic` and `toc` as shown here:

```
tic
   -- run the program section to be timed --
toc
```

Keep in mind that `tic` and `toc` measure overall elapsed time. Make sure that no other applications are running in the background on your system that could affect the timing of your MATLAB programs.

Measuring Smaller Programs

Shorter programs sometimes run too fast to get useful data from `tic` and `toc`. When this is the case, try measuring the program running repeatedly in a loop, and then average to find the time for a single run:

```
tic
   for k = 1:100
      -- run the program --
   end
toc
```

Using tic and toc Versus the cputime Function

Although it is possible to measure performance using the `cputime` function, it is recommended that you use the `tic` and `toc` functions for this purpose exclusively. It has been the general rule for CPU-intensive calculations run on Windows machines that the elapsed time using `cputime` and the elapsed time using `tic` and `toc` are close in value, ignoring any first time costs. There are cases however that show a significant difference between these two methods. For example, in the case of a Pentium 4 with hyperthreading running Windows, there can be a significant difference between the values returned by `cputime` versus `tic` and `toc`.

Techniques for Improving Performance

In this section...

Vectorizing Loops

MATLAB is a matrix language, which means it is designed for vector and matrix operations. You can often speed up your M-file code by using vectorizing algorithms that take advantage of this design. *Vectorization* means converting `for` and `while` loops to equivalent vector or matrix operations.

Simple Example of Vectorizing

Here is one way to compute the sine of 1001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
```

```
end
```

A vectorized version of the same code is

```
t = 0:.01:10;
y = sin(t);
```

The second example executes much faster than the first and is the way MATLAB is meant to be used. Test this on your system by creating M-file scripts that contain the code shown, and then using the `tic` and `toc` functions to time the M-files.

Advanced Example of Vectorizing

`repmat` is an example of a function that takes advantage of vectorization. It accepts three input arguments: an array `A`, a row dimension `M`, and a column dimension `N`.

`repmat` creates an output array that contains the elements of array `A`, replicated and "tiled" in an `M`-by-`N` arrangement:

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

B = repmat(A,2,3);
B =
    1    2    3    1    2    3    1    2    3
    4    5    6    4    5    6    4    5    6
    1    2    3    1    2    3    1    2    3
    4    5    6    4    5    6    4    5    6
```

`repmat` uses vectorization to create the indices that place elements in the output array:

```
function B = repmat(A, M, N)

% Step 1 Get row and column sizes
[m,n] = size(A);

% Step 2 Generate vectors of indices from 1 to row/column size
mind = (1:m)';
nind = (1:n)';
```

```
% Step 3 Create index matrices from vectors above
mind = mind(:,ones(1, M));
nind = nind(:,ones(1, N));

% Step 4 Create output array
B = A(mind,nind);
```

Step 1, above, obtains the row and column sizes of the input array.

Step 2 creates two column vectors. `mind` contains the integers from 1 through the row size of `A`. The `nind` variable contains the integers from 1 through the column size of `A`.

Step 3 uses a MATLAB vectorization trick to replicate a single column of data through any number of columns. The code is

```
B = A(:,ones(1,nCols))
```

where `nCols` is the desired number of columns in the resulting matrix.

Step 4 uses array indexing to create the output array. Each element of the row index array, `mind`, is paired with each element of the column index array, `nind`, using the following procedure:

1. The first element of `mind`, the row index, is paired with each element of `nind`. MATLAB moves through the `nind` matrix in a columnwise fashion, so `mind(1,1)` goes with `nind(1,1)`, and then `nind(2,1)`, and so on. The result fills the first row of the output array.
2. Moving columnwise through `mind`, each element is paired with the elements of `nind` as above. Each complete pass through the `nind` matrix fills one row of the output array.

Caution While `repmat` can take advantage of vectorization, it can do so at the expense of memory usage. When this is the case, you might find the `bsxfun` function be more appropriate in this respect.

Functions Used in Vectorizing

Some of the most commonly used functions for vectorizing are as follows

Function	Description
`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`cumsum`	Find cumulative sum
`diff`	Find differences and approximate derivatives
`find`	Find indices and values of nonzero elements
`ind2sub`	Convert from linear index to subscripts
`ipermute`	Inverse permute dimensions of a multidimensional array
`logical`	Convert numeric values to logical
`ndgrid`	Generate arrays for multidimensional functions and interpolation
`permute`	Rearrange dimensions of a multidimensional array
`prod`	Find product of array elements
`repmat`	Replicate and tile an array
`reshape`	Change the shape of an array
`shiftdim`	Shift array dimensions
`sort`	Sort array elements in ascending or descending order
`squeeze`	Remove singleton dimensions from an array
`sub2ind`	Convert from subscripts to linear index
`sum`	Find the sum of array elements

Preallocating Arrays

`for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can adversely affect performance and memory use. Repeatedly resizing arrays often requires that MATLAB spend extra time looking for larger contiguous blocks of memory and then moving the array into those blocks. You can often improve on code execution time by

preallocating the maximum amount of space that would be required for the array ahead of time.

The following code creates a scalar variable `x`, and then gradually increases the size of `x` in a `for` loop instead of preallocating the required amount of memory at the start:

```
x = 0;
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

Change the first line to preallocate a 1-by-1000 block of memory for `x` initialized to zero. This time there is no need to repeatedly reallocate memory and move data as more values are assigned to `x` in the loop:

```
x = zeros(1, 1000);
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

Preallocation Functions

Preallocation makes it unnecessary for MATLAB to resize an array each time you enlarge it. Use the appropriate preallocation function for the kind of array you are working with.

Array Type	Function	Examples
Numeric	`zeros`	`y = zeros(1, 100);`

Cell	`cell`	`B = cell(2, 3);`
		`B{1,3} = 1:3;`
		`B{2,2} = 'string';`

Preallocating a Nondouble Matrix

When you preallocate a block of memory to hold a matrix of some type other than `double`, avoid using the method

```
A = int8(zeros(100));
```

This statement preallocates a 100-by-100 matrix of `int8` first by creating a full matrix of `doubles`, and then converting each element to `int8`. This costs time and uses memory unnecessarily.

The next statement shows how to do this more efficiently:

```
A = zeros(100, 'int8');
```

Use Distributed Arrays for Large Datasets

This topic is described in the "Parallel Math" section of the Distributed Computing Toolbox documentation.

When Possible, Replace for with parfor (Parallel for)

This topic is described in the "Parallel for-Loops" section of the Distributed Computing Toolbox documentation.

Multithreading Capabilities in MATLAB

See "Implicit Multiprocessing" on page 11-14 to learn more about making use of multithreaded computation.

Limiting M-File Size and Complexity

Running programs that are unusually large or complex can put a strain on your system's resources. For example, a program that nearly exceeds memory capacity may work some of the time and sometimes not, depending on the commands it uses and on what other applications are running at the time. An example of unnecessary complexity might be having a large number of `if` and `else` statements where `switch` and `case` might be more suitable. This can also lead to performance and space problems.

If you have an M-file that includes thousands of variables or functions, tens of thousands of statements, or hundreds of language keyword pairs (e.g., `if-else`, or `try-catch`), then making some of the changes suggested here is likely to not only boost its performance and reliability, but should make your program code easier to understand and maintain as well.

Here are a few suggestions on how to make your programs less resource-intensive:

- Split large script files into smaller ones, having the first file call the second if necessary.
- Take your larger chunks of program code and make separate functions (or subfunctions and nested functions) of them.
- If you have functions or expressions by that seem overly complicated, make smaller and simpler functions or expressions of them. Simpler functions are also more likely to be made into utility functions that you can share with others.

Coding Loops in a MEX-File

If there are instances where you cannot vectorize and must use a `for` or `while` loop, consider coding the loop in a MEX-file. In this way, the loop executes much more quickly since the instructions in the loop do not have to be interpreted each time they execute.

See “Introducing MEX-Files” in the External Interfaces documentation.

Assigning to Variables

For best performance, keep the following suggestions in mind when assigning values to variables.

Changing a Variable’s Data Type or Dimension

Changing the data type or array shape of an existing variable slows MATLAB down as it must take extra time to process this. When you need to store data of a different type, it is advisable to create a new variable.

This code changes the type for `X` from `double` to `char`, which has a negative impact on performance:

```
X = 23;
     .
-- other code  --
     .
X = 'A';                  % X changed from type double to char
     .
-- other code  --
```

Assigning Real and Complex Numbers

Assigning a complex number to a variable that already holds a real number impacts the performance of your program. Similarly, you should not assign a real value to a variable that already holds a complex value.

Operating on Real Data

When operating on real (i.e., noncomplex) numbers, it is more efficient to use MATLAB functions that have been designed specifically for real numbers. The following functions return numeric values that are real.

Function	Description
`reallog`	Find natural logarithm for nonnegative real arrays
`realpow`	Find array power for real-only output
`realsqrt`	Find square root for nonnegative real arrays

Using Appropriate Logical Operators

When performing a logical AND or OR operation, you have a choice of two operators of each type.

Operator	Description
`&, |`	Perform logical AND and OR on arrays element by element
`&&, ||`	Perform logical AND and OR on scalar values with short-circuiting

In `if` and `while` statements, it is more efficient to use the short-circuiting operators, `&&` for logical AND and `||` for logical OR. This is because these operators often don't have to evaluate the entire logical expression. For example, MATLAB evaluates only the first part of this expression whenever the number of input arguments is less than three:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

See Short-Circuit Operators in the MATLAB documentation for a discussion on short-circuiting with `&&` and `||`.

Overloading Built-In Functions

Overloading MATLAB built-in functions on any of the standard MATLAB data types can negatively affect performance. For example, if you overload the `plus` function to handle any of the integer data types differently, you may hinder certain optimizations in the MATLAB built-in function code for `plus`, and thus may slow down any programs that make use of this overload.

Functions Are Generally Faster Than Scripts

Your code executes more quickly if it is implemented in a function rather than a script.

Load and Save Are Faster Than File I/O Functions

If you have a choice of whether to use `load` and `save` instead of the low-level MATLAB file I/O routines such as `fread` and `fwrite`, choose the former. `load` and `save` have been optimized to run faster and reduce memory fragmentation.

Avoid Large Background Processes

Avoid running large processes in the background at the same time you are executing your program in MATLAB. This frees more CPU time for your MATLAB session.

Multiprocessing in MATLAB

In this section...

Overview

MATLAB supports two types of multiprocessing: *implicit* and *explicit*.

Implicit Multiprocessing

Characteristics of implicit multiprocessing:

- Runs multiple threads on a single machine, most often using one thread per processing unit.
- Requires a multiple CPU (multiprocessor or multicore) system.
- Speeds up elementwise computations such as those done by the `sin` and `log` functions, and computations that use the Basic Linear Algebra Subroutines (BLAS) library, such as matrix multiply.
- Does *not* require any changes to your MATLAB code.
- Works behind the scenes to take advantage of the processing units available to you. It does this by multithreading the computationally-intensive math library functions that you use in the course of your MATLAB session.

Enable implicit multiprocessing with the MATLAB Preferences Panel to enable or disable, or to set the number of threads to be used. You can change the maximum number of threads programmatically using the `maxNumCompThreads` function.

Explicit Multiprocessing

Characteristics of explicit multiprocessing:

- Runs separate processes on one or many machines.

- Requires installation of Distributed Computing Toolbox (DCT).
- Speeds up execution of large MATLAB jobs. Enables you to run jobs simultaneously on a cluster of computers, or as several processes on a single machine.
- Requires that you modify your MATLAB code.
- DCT supports programming constructs for distributed arrays and parallel `for` (`parfor`) loops. It also supports both interactive and batch execution.

Enable explicit multiprocessing by installing Distributed Computing Toolbox.

Implicit Multiprocessing

Multithreaded computation runs in a single instance of MATLAB and generates simultaneous instruction streams on a multiple CPU (multiprocessor or multicore) system. The multiple processors share the memory of a single computer. The work to be processed is implicitly partitioned for execution on multiple threads. In particular, multithreaded computation in MATLAB speeds up elementwise computations such as those done by the `sin` and `log` functions, and computations that use the Basic Linear Algebra Subroutines (BLAS) library, such as matrix multiply.

If you are using a multiple-CPU system, you can run a demo to see the performance impact—see Multithreaded Computation in the Help browser **Demos** pane, under MATLAB Mathematics.

For information regarding specific functions, search for “What MATLAB Functions Support Multithreaded Computation” on The MathWorks online Support page.

Platform Differences and Multithreaded Computation

The BLAS library used for multithreaded computation differs according to which platform you are using:

Platform	BLAS Used
Windows with Intel processors	Intel MKL BLAS
Windows with AMD processors	AMD ACML BLAS
Linux with Intel processors	Intel MKL BLAS
Linux with AMD processors	AMD ACML BLAS
Macintosh Intel-based	Intel MKL BLAS
MacIntosh PowerPC	Mac Accelerate BLAS
Solaris	Sun Performance Library BLAS

Note On Macintosh PowerPC platforms, multithreaded computation is always enabled for the Accelerate BLAS. To enable multithreaded computation for elementwise operations, use MATLAB preferences.

Enabling Multithreaded Computation

The preference automatically detects the number of CPUs on your system and recommends the number of threads based on that.

Multithreaded computation in MATLAB is disabled by default. To enable it and set the maximum number of threads to use, follow these steps:

1. Select **File > Preferences > General > Multithreading**.

   The **General Multithreading Preferences** panel opens.

2. On the **General Multithreading Preferences** panel, select **Enable multithreaded computation**.

3. Specify the **Maximum number of computational threads**. Accepting the **Automatic** option is recommended—MATLAB automatically sets the value to the actual number of computational cores on your system. Note

that if your system uses hyperthreading (where one processor is logically configured as two), MATLAB sets the value to 1.

If you choose **Manual**, enter the maximum number of threads you want to set; use a positive integer not greater than 16. (Selecting a number other than the recommended value might increase performance for some computations, but might decrease performance for others.)

Note You may find that, at certain times, a library function uses a number of threads smaller than what you have specified. This can happen if the function finds the specified number of threads to be inappropriate.

In the event of an abnormal termination with multithreaded computation enabled, MATLAB behaves differently than when multithreaded computation is not enabled. For details, see .

Making this setting in the **Preferences** panel not only affects your current MATLAB session, but future sessions as well. To disable multithreaded computation, clear the **Enable multithreaded computation** selection and click **OK**.

Note For Macintosh PowerPC platforms, BLAS multithreaded computation cannot be disabled.

Setting the Number of Threads Programmatically

To set or retrieve the maximum number of computational threads from within an M-file program, use the `maxNumCompThreads` function. You can either set the maximum number of computational threads to a specific number, or indicate that you want the setting to be done automatically by MATLAB.

To set the maximum number of computational threads to a specific number `N`, use

```
maxNumCompThreads(N)
```

To have MATLAB set the maximum number of threads, use:

```
maxNumCompThreads('automatic')
```

`maxNumCompThreads` also returns the current maximum number of threads if you call it with an output value:

```
old_N = MaxNumCompThreads(new_N)
```

MATLAB keeps the settings you make using `maxNumCompThreads` synchronous with your **Preferences** settings. If you change the maximum number of computational threads by means of the `maxNumCompThreads` function, MATLAB updates the **Preferences** panel to agree with the new setting.

Note Setting the maximum number of computational threads using `maxNumCompThreads` does not propagate to your next MATLAB session. To make this setting carry over to future sessions, use the **Preferences** panel instead.

Crash Recovery and Multithreading

If MATLAB experiences a segmentation violation or other serious problem when multithreaded computation is enabled, it cannot try to return control to the Command Window. You do not have an opportunity to view a segmentation violation message in the Command Window as you might when multithreaded computation is *not* enabled. Instead, your platform's vendor, for example, Microsoft or Apple, provides an error dialog box. MATLAB then terminates.

Upon the next MATLAB startup after a fatal problem, the "Error Log Reporter" prompts you to e-mail the log to The MathWorks.

Explicit Multiprocessing

See the Distributed Computing Toolbox documentation for information regarding explicit multiprocessing in MATLAB.

Memory Allocation in MATLAB

In this section...

For more information on memory management, see Technical Note 1106: "Memory Management Guide" at the following URL:

```
http://www.mathworks.com/support/tech-notes/1100/1106.html
```

Memory Allocation for Arrays

The topics below provide information on how MATLAB allocates memory when working with arrays and variables. The purpose is to help you use memory more efficiently when writing code. Most of the time, however, you should not need to be concerned with these internal operations as MATLAB handles data storage for you automatically.

Note Any information on how data is handled internally by MATLAB is subject to change in future releases.

Creating and Modifying Arrays

When you assign any type of data (a numeric, string, or structure array, for example) to a variable, MATLAB allocates a contiguous block of memory and stores the array data in that block. It also stores information about the array data, such as its data type and dimensions, in a separate, small block of memory called a header. The variable that you assign this data to is actually a *pointer* to the data; it does not *contain* the data.

If you add new elements to an existing array, MATLAB expands the existing array in memory in a way that keeps its storage contiguous. This might require finding a new block of memory large enough to hold the expanded array, and then copying the contents of the array from its original location to the new block in memory, adding the new elements to the array in this block, and freeing up the original array location in memory.

If you remove elements from an existing array, MATLAB keeps the memory storage contiguous by removing the deleted elements, and then compacting its storage in the original memory location.

Working with Large Data Sets. If you are working with large data sets, you need to be careful when increasing the size of an array to avoid getting errors caused by insufficient memory. If you expand the array beyond the available contiguous memory of its original location, MATLAB has to make a copy of the array in a new location in memory, as explained above, and then set this array to its new value. During this operation, there are two copies of the original array in memory, thus temporarily doubling the amount of memory required for the array and increasing the risk of your program running out of memory during execution. It is better to preallocate sufficient memory for the array at the start. See “Preallocating Arrays” on page 11-7.

Copying Arrays

Internally, multiple variables can point to the same block of data, thus sharing that array’s value. When you copy a variable to another variable (e.g., `B = A`), MATLAB makes a copy of the pointer, not the array. For example, the following code creates a single 500-by-500 matrix and two pointers to it, `A` and `B`:

```
A = magic(500);
B = A;
```

As long as the contents of the array are not modified, there is no need to store two copies of it. If you modify the array, then MATLAB does create a separate array to hold the new values.

If you modify the array shown above by referencing it with variable `A` (e.g., `A(400,:) = 0`), then MATLAB creates a copy of the array, modifies it accordingly, and stores a pointer to the new array in `A`. Variable `B` continues

to point to the original array. If you modify the array by referencing it with variable `B` (e.g., `B(400,:) = 0`), the same thing happens except that it is `B` that points to the new array.

Array Headers

When you assign an array to a variable, MATLAB also stores information about the array (such as data type and dimensions) in a separate piece of memory called a header. For most arrays, the memory required to store the header is insignificant. There is a small advantage though to storing large data sets in a small number of large arrays as opposed to a large number of small arrays, as the former configuration requires fewer array headers.

Structure and Cell Arrays. For structures and cell arrays, MATLAB creates a header not only for each array, but also for each field of the structure and for each cell of a cell array. Because of this, the amount of memory required to store a structure or cell array depends not only on how much data it holds, but also how it is constructed.

For example, a scalar structure array `S1` having fields `R`, `G`, and `B`, each field of size 100-by-50, requires one array header to describe the overall structure, and one header to describe each of the three field arrays, making a total of 4 array headers for the entire data structure:

```
S1.R(1:100,1:50)
S1.G(1:100,1:50)
S1.B(1:100,1:50)
```

On the other hand, a 100-by-50 structure array `S2` in which each element has scalar fields `R`, `G`, and `B` requires one array header to describe the overall structure, and one array header per field for each of the 5,000 elements of the structure, making a total of 15,001 array headers for the entire data structure:

```
S2(1:100,1:50).R
S2(1:100,1:50).G
S2(1:100,1:50).B
```

Thus, even though `S1` and `S2` contain the same amount of data, S1 uses significantly less space in memory. Not only is less memory required, but there is a corresponding speed benefit to using the `S1` format as well.

Memory Usage Reported By the whos Function. The `whos` function displays the amount of memory consumed by any variable. For reasons of simplicity, `whos` reports only the memory used to store the actual data. It does not report storage for the variable itself or the array header.

Function Arguments

MATLAB handles arguments passed in function calls in a similar way. When you pass a variable to a function, you are actually passing a pointer to the data that the variable represents. As long as the input data is not modified by the function being called, the variable in the calling function and the variable in the called function point to the same location in memory. If the called function modifies the value of the input data, then MATLAB makes a copy of the original array in a new location in memory, updates that copy with the modified value, and points the input variable in the called function to this new array.

In the example below, function `myfun` modifies the value of the array passed into it. MATLAB makes a copy in memory of the array pointed to by `A`, sets variable `X` as a pointer to this new array, and then sets one row of `X` to zero. The array referenced by `A` remains unchanged:

```
A = magic(500);
myfun(A);

function myfun(X)
X(400,:) = 0;
```

If the calling function needs the modified value of the array it passed to `myfun`, you will need to return the updated array as an output of the called function, as shown here for variable `A`:

```
A = magic(500);
A = myfun(A);
sprintf('The new value of A is %d', A)

function Y = myfun(X)
X(400,:) = 0;
Y = X;
```

Working with Large Data Sets. Again, when working with large data sets, you should be aware that MATLAB makes a temporary copy of `A` if the called function modifies its value. This temporarily doubles the memory required to store the array, which causes MATLAB to generate an error if sufficient memory is not available.

One way to avoid running out of memory in this situation is to use nested functions. A nested function shares the workspace of all outer functions, giving the nested function access to data outside of its usual scope. In the example shown here, nested function `setrowval` has direct access to the workspace of the outer function `myfun`, making it unnecessary to pass a copy of the variable in the function call. When `setrowval` modifies the value of `A`, it modifies it in the workspace of the calling function. There is no need to use additional memory to hold a separate array for the function being called, and there also is no need to return the modified value of `A`:

```
function myfun
A = magic(500);

   function setrowval(row, value)
   A(row,:) = value;
   end

setrowval(400, 0);
disp('The new value of A(399:401,1:10) is')
A(399:401,1:10)
end
```

Data Structures and Memory

Memory requirements differ for the various types of MATLAB data structures. You may be able to reduce the amount of memory used for these structures by considering how MATLAB stores them.

Numeric Arrays

MATLAB requires 1, 2, 4, or 8 bytes to store 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers respectively. For floating-point numbers, MATLAB uses 4 or 8 bytes for `single` and `double` types. To conserve memory, The MathWorks recommends that you use the smallest integer or floating-point type that will contain your data without overflowing. For more

information, see "Numeric Types" in the MATLAB Programming section on Data Types.

Complex Arrays

MATLAB stores complex data as separate real and imaginary parts. If you make a copy of a complex array variable, and then modify only the real or imaginary part of the array, MATLAB creates a new array containing both real and imaginary parts.

Sparse Matrices

It is best to store matrices with values that are mostly zero in sparse format. Sparse matrices can use less memory and may also be faster to manipulate than full matrices. You can convert a full matrix to sparse format using the `sparse` function.

Compare two 1000-by-1000 matrices: `X`, a matrix of doubles with 2/3 of its elements equal to zero; and `Y`, a sparse copy of `X`. As shown below, approximately half as much memory is required for the sparse matrix:

```
whos
  Name      Size                    Bytes  Class

  X      1000x1000                8000000  double array
  Y      1000x1000                4004000  double array (sparse)
```

Memory Management Functions

The following functions can help you to manage memory use in MATLAB:

- `whos` shows how much memory has been allocated for variables in the workspace.
- `pack` saves existing variables to disk, and then reloads them contiguously. This reduces the chances of running into problems due to memory fragmentation.

  See "Compressing Data in Memory" on page 11-28.
- `clear` removes variables from memory. One way to increase the amount of available memory is to periodically clear variables from memory that you no longer need.

  If you use `pack` and there is still not enough free memory to proceed, you probably need to remove some of the variables you are no longer using from memory. Use `clear` to do this.
- `save` selectively stores variables to the disk. This is a useful technique when you are working with large amounts of data. Save data to the disk periodically, and then use the `clear` function to remove the saved data from memory.
- `load` reloads a data file saved with the `save` function.
- `quit` exits MATLAB and returns all allocated memory to the system. This can be useful on UNIX systems as UNIX does not free up memory allocated to an application (e.g., MATLAB) until the application exits.

You can use the `save` and `load` functions in conjunction with the `quit` command to free memory by:

1. Saving any needed variables with the `save` function.
2. Quitting MATLAB to free all memory allocated to MATLAB.
3. Starting a new MATLAB session and loading the saved variables back into the clean MATLAB workspace.

Strategies for Efficient Use of Memory

In this section...

To conserve memory when creating variables,

- Avoid creating large temporary variables, and clear temporary variables when they are no longer needed.
- When working with arrays of fixed size, preallocate them rather than having MATLAB resize the array each time you enlarge it.
- Allocate your larger matrices first, as explained in .
- Set variables equal to the empty matrix `[]` to free memory, or clear the variables using the `clear` function.
- Reuse variables as much as possible, but keeping in mind the guidelines stated in “Assigning to Variables” on page 11-10.

Preallocating Arrays to Reduce Fragmentation

In the course of a MATLAB session, memory can become fragmented due to dynamic memory allocation and deallocation. `for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can add to this fragmentation as they have to repeatedly find and allocate larger blocks of memory to store the data.

To make more efficient use of your memory, preallocate a block of memory large enough to hold the matrix at its final size before entering the loop. When you preallocate memory for a potentially large array, MATLAB “grabs” sufficient contiguous space for the data at the beginning of the computation. Once you have this space, you can add elements to the array without having to continually allocate new space for it in memory.

For more information on preallocation, see “Preallocating Arrays” on page 11-7.

Allocating Large Matrices Earlier

MATLAB uses a heap method of memory management. It requests memory from the operating system when there is not enough memory available in the MATLAB heap to store the current variables. It reuses memory as long as the size of the memory segment required is available in the MATLAB heap.

For example, on one machine these statements use approximately 15.4 MB of RAM:

```
a = rand(1e6,1);
b = rand(1e6,1);
```

This statement uses approximately 16.4 MB of RAM:

```
c = rand(2.1e6,1);
```

The following statements can use approximately 32.4 MB of RAM. This is because MATLAB may not be able to reuse the space previously occupied by two 1MB arrays when allocating space for a 2.1 MB array:

```
a = rand(1e6,1);
b = rand(1e6,1);
clear
c = rand(2.1e6,1);
```

The simplest way to prevent overallocation of memory is to allocate the largest vectors first. These statements use only about 16.4 MB of RAM:

```
c = rand(2.1e6,1);
clear
a = rand(1e6,1);
b = rand(1e6,1);
```

Working with Large Amounts of Data

If your program generates very large amounts of data, consider writing the data to disk periodically. After saving that portion of the data, use the `clear` function to remove the variable from memory and continue with the data generation.

Resolving "Out of Memory" Errors

In this section...

For more information on this topic, search for "Avoiding Out of Memory Errors" on The MathWorks online "Support" page.

General Suggestions for Reclaiming Memory

MATLAB generates an `Out of Memory` message whenever it requests a segment of memory from the operating system that is larger than what is currently available. When you see the `Out of Memory` message, use any of the techniques discussed under "Memory Allocation in MATLAB" on page 11-18 to help optimize the available memory. If the `Out of Memory` message still appears, you can try any of the following:

- Compress data to reduce memory fragmentation
- If possible, break large matrices into several smaller matrices so that less memory is used at any one time.
- If possible, reduce the size of your data.
- Make sure that there are no external constraints on the memory accessible to MATLAB. (On UNIX systems, use the `limit` command to check).
- Increase the size of the swap file. We recommend that your machine be configured with twice as much swap space as you have RAM. See "Increasing System Swap Space" on page 11-28, below.
- Add more memory to the system.

On machines running Windows 2000 Advanced Server or Windows 2000 Datacenter Server, the amount of virtual memory space reserved by the

operating system can be reduced by using the /3GB switch in the `boot.ini` file. More documentation on this option can be found at the following URL:

`http://support.microsoft.com/support/kb/articles/Q291/9/88.ASP`

Similarly, on machines running Windows Vista, you can achieve the same effect by using the command:

```
BCDEdit /set increaseuserva 3072
```

More documentation on this option can be found at the following URL:

`http://msdn2.microsoft.com/en-us/library/aa906211.aspx`

Compressing Data in Memory

Since MATLAB uses a heap method of memory management, extended MATLAB sessions may cause memory to become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable. If you get the `Out of Memory` message from MATLAB, the `pack` function may be able to compress some of your data in memory, thus freeing up larger contiguous blocks.

Note Because of time considerations, you should not use `pack` within loops or M-file functions.

Increasing System Swap Space

How you set the swap space for your computer depends on what operating system you are running on.

UNIX

Information about swap space can be procured by typing `pstat -s` at the UNIX command prompt. For detailed information on changing swap space, ask your system administrator.

Linux

Swap space can be changed by using the `mkswap` and `swapon` commands. For more information on the above commands, type `man` followed by the command name at the Linux prompt.

Windows 2000

Follow the steps shown here:

1. Right-click the **My Computer** icon, and select **Properties**.
2. Select the **Advanced** tab and click the **Performance Options** button.
3. Click the **Change** button to change the amount of virtual memory.

Windows XP

Follow the steps shown here:

1. Right-click the **My Computer** icon, and select **Properties**.
2. In the **System Properties** GUI, select the **Advanced** tab. In the section labeled **Performance**, click the **Settings** button.
3. In the **Performance Options** GUI, click **Advanced**. In the section labeled **Virtual Memory**, click the **Change** button
4. In the **Virtual Memory** GUI, under **Paging file size for selected drive**, you can change the amount of virtual memory.

Freeing Up System Resources on Windows Systems

There are no functions implemented to manipulate the way MATLAB handles Microsoft Windows system resources. Windows uses system resources to track fonts, windows, and screen objects. Resources can be depleted by using multiple figure windows, multiple fonts, or several UI controls. One way to free up system resources is to close all inactive windows. Windows icons still use resources.

Reloading Variables on UNIX Systems

On UNIX systems, MATLAB does not return memory to the operating system even after variables have been cleared. This is due to the manner in which UNIX manages memory. UNIX does not accept memory back from a program until the program has terminated. So, the amount of memory used in a MATLAB session is not returned to the operating system until you exit MATLAB.

To free up the memory used in your MATLAB session, save your workspace variables, exit MATLAB, and then load your variables back in.

12

Programming Tips

Introduction

This section is a categorized compilation of tips for the MATLAB® programmer. Each item is relatively brief to help you browse through them and find information that is useful. Many of the tips include a reference to specific MATLAB documentation that gives you more complete coverage of the topic. You can find information on the following topics:

For suggestions on how to improve the performance of your MATLAB programs, and how to write programs that use memory more efficiently, see Chapter 11, “Improving Performance and Memory Usage”

Command and Function Syntax

In this section...

Syntax Help

For help about the general syntax of MATLAB functions and commands, type

```
help syntax
```

Command and Function Syntaxes

You can enter MATLAB commands using either a *command* or *function* syntax. It is important to learn the restrictions and interpretation rules for both.

```
functionname arg1 arg2 arg3               % Command syntax
functionname('arg1','arg2','arg3')        % Function syntax
```

For more information: See “Calling Functions” on page 4-52in the MATLAB Programming documentation.

Command Line Continuation

You can continue most statements to one or more additional lines by terminating each incomplete line with an ellipsis (`...`). Breaking down a statement into a number of lines can sometimes result in a clearer programming style.

```
sprintf ('Example %d shows a command coded on %d lines.\n', ...
          exampleNumber, ...
```

```
            numberOfLines)
```

Note that you cannot continue an incomplete string to another line.

```
disp 'This statement attempts to continue a string ...
      to another line, resulting in an error.'
```

For more information: See Entering Long Lines in the MATLAB Desktop Tools and Development Environment documentation.

Completing Commands Using the Tab Key

You can save some typing when entering commands by entering only the first few letters of the command, variable, property, etc. followed by the **Tab** key. Typing the second line below (with **T** representing **Tab**) yields the expanded, full command shown in the third line:

```
f = figure;
set(f, 'papTuT,'cT)                    % Type this line.
set(f, 'paperunits','centimeters')     % This is what you get.
```

If there are too many matches for the string you are trying to complete, you will get no response from the first **Tab**. Press **Tab** again to see all possible choices:

```
set(f, 'paTT
PaperOrientation   PaperPositionMode  PaperType     Parent
PaperPosition      PaperSize          PaperUnits
```

For more information: See Tab Completion in the MATLAB Desktop Tools and Development Environment documentation

Recalling Commands

Use any of the following methods to simplify recalling previous commands to the screen:

- To recall an earlier command to the screen, press the up arrow key one or more times, until you see the command you want. If you want to modify the recalled command, you can edit its text before pressing **Enter** or **Return** to execute it.

- To recall a specific command by name without having to scroll through your earlier commands one by one, type the starting letters of the command, followed by the up arrow key.
- Open the Command History window (**View > Command History**) to see all previous commands. Double-click the command you want to execute.

For more information: See Recalling Previous Lines and Command History in the MATLAB Desktop Tools and Development Environment documentation.

Clearing Commands

If you have typed a command that you then decide not to execute, you can clear it from the Command Window by pressing the Escape (**Esc**) key.

Suppressing Output to the Screen

To suppress output to the screen, end statements with a semicolon. This can be particularly useful when generating large matrices.

```
A = magic(100);     % Create matrix A, but do not display it.
```

Help

In this section...

Using the Help Browser

Open the Help browser from the MATLAB Command Window using one of the following:

- Click the question mark symbol in the toolbar.
- Select **Help > Product Help** from the menu.
- Type the word `doc` at the command prompt.

Some of the features of the Help browser are listed below.

Feature	Description
Product Filter	Establish which products to find help on.
Contents	Look up topics in the Table of Contents.
Index	Look up help using the documentation Index.
Search	Search the documentation for one or more words.
Demos	See what demos are available; run selected demos.
Favorites	Save bookmarks for frequently used Help pages.

For more information: See Finding Information with the Help Browser in the MATLAB Desktop Tools and Development Environment documentation.

Help on Functions from the Help Browser

To find help on any function from the Help browser, do either of the following:

- Select the **Contents** tab of the Help browser, open the **Contents** entry labeled `MATLAB`, and find the two subentries shown below. Use one of these to look up the function you want help on.
  - Functions — Categorical List
  - Functions — Alphabetical List
- Type `doc functionname` at the command line.

Help on Functions from the Command Window

Several types of help on functions are available from the Command Window:

- To list all categories that you can request help on from the Command Window, just type

  ```
  help
  ```

- To see a list of functions for one of these categories, along with a brief description of each function, type `help` *`category`*. For example,

  ```
  help datafun
  ```

- To get help on a particular function, type `help functionname`. For example,

  ```
  help sortrows
  ```

Topical Help

In addition to the help on individual functions, you can get help on any of the following topics by typing `help topicname` at the command line.

Topic Name	Description
`arith`	Arithmetic operators
`relop`	Relational and logical operators
`punct`	Special character operators
`slash`	Arithmetic division operators
`paren`	Parentheses, braces, and bracket operators
`precedence`	Operator precedence
`datatypes`	MATLAB data types, their associated functions, and operators that you can overload
`lists`	Comma separated lists
`strings`	Character strings
`function_handle`	Function handles and the `@` operator
`debug`	Debugging functions
`java`	Using Java from within MATLAB
`fileformats`	A list of readable file formats
`changeNotification`	Windows directory change notification

Paged Output

Before displaying a lengthy section of help text or code, put MATLAB into its paged output mode by typing `more on`. This breaks up any ensuing display into pages for easier viewing. Turn off paged output with `more off`.

Page through the displayed text using the space bar key. Or step through line by line using **Enter** or **Return**. Discontinue the display by pressing the **Q** key or **Ctrl+C**.

Writing Your Own Help

Start each program you write with a section of text providing help on how and when to use the function. If formatted properly, the MATLAB `help` function displays this text when you enter

```
help functionname
```

MATLAB considers the first group of consecutive lines immediately following the function definition line that begin with `%` to be the help section for the function. The first line without `%` as the left-most character ends the help.

For more information: See Help Text in the MATLAB Desktop Tools and Development Environment documentation.

Help for Subfunctions and Private Functions

You can write help for subfunctions using the same rules that apply to main functions. To display the help for the subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

To display the help for a private function, precede the function name with `private/`. To get help on private function `myprivfun`, type

```
help private/myprivfun
```

Help for Methods and Overloaded Functions

You can write help text for object-oriented class methods implemented with M-files. Display help for the method by typing

```
help classname/methodname
```

where the file `methodname.m` resides in subdirectory `@classname`.

For example, if you write a `plot` method for a class named `polynom`, (where the `plot` method is defined in the file `@polynom/plot.m`), you can display this help by typing

```
help polynom/plot
```

You can get help on overloaded MATLAB functions in the same way. To display the help text for the `eq` function as implemented in `matlab/iofun/@serial`, type

```
help serial/eq
```

Development Environment

In this section...

Workspace Browser

The Workspace browser is a graphical interface to the variables stored in the MATLAB base and function workspaces. You can view, modify, save, load, and create graphics from workspace data using the browser. Select **View > Workspace** to open the browser.

To view function workspaces, you need to be in debug mode.

For more information: See MATLAB Workspace in the MATLAB Desktop Tools and Development Environment documentation.

Using the Find and Replace Utility

Find any word or phrase in a group of files using the Find and Replace utility. Click **View > Current Directory**, and then click the binoculars icon at the top of the **Current Directory** window.

When entering search text, you do not need to put quotes around a phrase. In fact, parts of words, like `win` for `windows`, will not be found if enclosed in quotes.

For more information: See Finding and Replacing a String in the MATLAB Desktop Tools and Development Environment documentation.

Commenting Out a Block of Code

To comment out a block of text or code within the MATLAB editor,

1. Highlight the block of text you would like to comment out.
2. Holding the mouse over the highlighted text, select **Text > Comment** (or **Uncomment**, to do the reverse) from the toolbar. (You can also get these options by right-clicking the mouse.)

For more information: See Commenting in the MATLAB Desktop Tools and Development Environment documentation.

Creating M-Files from Command History

If there is part of your current MATLAB session that you would like to put into an M-file, this is easily done using the Command History window:

1. Open this window by selecting **View > Command History**.
2. Use **Shift+Click** or **Ctrl+Click** to select the lines you want to use. MATLAB highlights the selected lines.
3. Right-click once, and select **Create M-File** from the menu that appears. MATLAB creates a new Editor window displaying the selected code.

Editing M-Files in EMACS

If you use Emacs, you can download editing modes for editing M-files with GNU-Emacs or with early versions of Emacs from the MATLAB Central Web site:

```
http://www.mathworks.com/matlabcentral/
```

At this Web site, select **File Exchange**, and then **Utilities > Emacs**.

For more information: See General Preferences for the Editor/Debugger in the MATLAB Desktop Tools and Development Environment documentation.

M-File Functions

In this section...

M-File Structure

An M-File consists of the components shown here:

```
function [x, y] = myfun(a, b, c)   % Function definition line
% H1 line -- A one-line summary of the function's purpose.
% Help text -- One or more lines of help text that explain
%   how to use the function. This text is displayed when
%   the user types "help functionname".

% The Function body normally starts after the first blank line.
% Comments -- Description (for internal use) of what the
%   function does, what inputs are expected, what outputs
%   are generated. Typing "help functionname" does not display
%   this text.

x = prod(a, b);                    % Start of Function code
```

For more information: See “Basic Parts of an M-File” on page 4-8 in the MATLAB Programming documentation.

Using Lowercase for Function Names

Function names appear in uppercase in MATLAB help text only to make the help easier to read. In practice, however, it is usually best to use lowercase when calling functions.

For M-file functions, case requirements depend on the case sensitivity of the operating system you are using. As a rule, naming and calling functions using lowercase generally makes your M-files more portable from one operating system to another.

Getting a Function's Name and Path

To obtain the name of an M-file that is currently being executed, use the following function in your M-file code.

```
mfilename
```

To include the path along with the M-file name, use

```
mfilename('fullpath')
```

For more information: See the `mfilename` function reference page.

What M-Files Does a Function Use?

For a simple display of all M-files referenced by a particular function, follow the steps below:

1. Type `clear functions` to clear all functions from memory (see Note below).
2. Execute the function you want to check. Note that the function arguments you choose to use in this step are important, since you can get different results when calling the same function with different arguments.
3. Type `inmem` to display all M-Files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output, as shown here:

   ```
   [mfiles, mexfiles] = inmem
   ```

Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions, (which you can check using `inmem`), unlock them with `munlock`, and then repeat step 1.

Dependent Functions, Built-Ins, Classes

For a much more detailed display of dependent function information, use the `depfun` function. In addition to M-files, `depfun` shows which built-ins and classes a particular function depends on.

Function Arguments

In this section...

Getting the Input and Output Arguments

Use `nargin` and `nargout` to determine the number of input and output arguments in a particular function call. Use `nargchk` and `nargoutchk` to verify that your function is called with the required number of input and output arguments.

```
function [x, y] = myplot(a, b, c, d)
disp(nargchk(2, 4, nargin))           % Allow 2 to 4 inputs
disp(nargoutchk(0, 2, nargout))       % Allow 0 to 2 outputs

x = plot(a, b);
if nargin == 4
   y = myfun(c, d);
end
```

Variable Numbers of Arguments

You can call functions with fewer input and output arguments than you have specified in the function definition, but not more. If you want to call a function with a variable number of arguments, use the `varargin` and `varargout` function parameters in the function definition.

This function returns the size vector and, optionally, individual dimensions:

```
function [s, varargout] = mysize(x)
nout = max(nargout, 1) - 1;
s = size(x);
for k = 1:nout
```

```
    varargout(k) = {s(k)};
end
```

Try calling it with

```
[s, rows, cols] = mysize(rand(4, 5))
```

String or Numeric Arguments

If you are passing only string arguments into a function, you can use MATLAB command syntax. All arguments entered in command syntax are interpreted as strings.

```
strcmp string1 string1
ans =
     1
```

When passing numeric arguments, it is best to use function syntax unless you want the number passed as a string. The right-hand example below passes the number `75` as the string, `'75'`.

```
isnumeric(75)
ans =
    1
```

```
isnumeric 75
ans =
    0
```

For more information: See “Passing Arguments with Command and Function Syntax” on page 4-57 in the MATLAB Programming documentation.

Passing Arguments in a Structure

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure and pass the structure. Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields.

Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The disadvantage over structures is that you do not have field names to describe each variable. The advantage is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function.

Program Development

In this section...

Planning the Program

When planning how to write a program, take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

Using Pseudo-Code

You may find it helpful to write the initial draft of your program in a structured format using your own natural language. This *pseudo-code* is often easier to think through, review, and modify than using a formal programming language, yet it is easily translated into a programming language in the next stage of development.

Selecting the Right Data Structures

Look at what data types and data structures are available to you in MATLAB and determine which of those best fit your needs in storing and passing your data.

For more information: See Data Types in the MATLAB Programming documentation.

General Coding Practices

A few suggested programming practices:

- Use descriptive function and variable names to make your code easier to understand.
- Order subfunctions alphabetically in an M-file to make them easier to find.
- Precede each subfunction with a block of help text describing what that subfunction does. This not only explains the subfunctions, but also helps to visually separate them.
- Do not extend lines of code beyond the 80th column. Otherwise, it will be hard to read when you print it out.
- Use full Handle Graphics® property and value names. Abbreviated names are often allowed, but can make your code unreadable. They also could be incompatible in future releases of MATLAB.

Naming a Function Uniquely

To avoid choosing a name for a new function that might conflict with a name already in use, check for any occurrences of the name using this command:

```
which -all functionname
```

For more information: See the `which` function reference page.

The Importance of Comments

Be sure to document your programs well to make it easier for you or someone else to maintain them. Add comments generously, explaining each major section and any smaller segments of code that are not obvious. You can add a block of comments as shown here.

```
%-----------------------------------------
% This function computes the ... <and so on>
%-----------------------------------------
```

For more information: See Comments in the MATLAB Programming documentation.

Coding in Steps

Do not try to write the entire program all at once. Write a portion of it, and then test that piece out. When you have that part working the way you want, then write the next piece, and so on. It's much easier to find programming errors in a small piece of code than in a large program.

Making Modifications in Steps

When making modifications to a working program, do not make widespread changes all at one time. It's better to make a few small changes, test and debug, make a few more changes, and so on. Tracking down a difficult bug in the small section that you've changed is much easier than trying to find it in a huge block of new code.

Functions with One Calling Function

If you have a function that is called by only one other function, put it in the same M-file as the calling function, making it a subfunction.

For more information: See Subfunctions in the MATLAB Programming documentation.

Testing the Final Program

One suggested practice for testing a new program is to step through the program in the MATLAB debugger while keeping a record of each line that gets executed on a printed copy of the program. Use different combinations of inputs until you have observed that every line of code is executed at least once.

Debugging

In this section...

The MATLAB Debug Functions

For a brief description of the main debug functions in MATLAB, type

```
help debug
```

For more information: See Debugging M-Files in the MATLAB Desktop Tools and Development Environment documentation.

More Debug Functions

Other functions you may find useful in debugging are listed below.

Function	Description
`echo`	Display function or script code as it executes.
`disp`	Display specified values or messages.
`sprintf`, `fprintf`	Display formatted data of different types.

Function	Description
`whos`	List variables in the workspace.
`size`	Show array dimensions.
`keyboard`	Interrupt program execution and allow input from keyboard.
`return`	Resume execution following a keyboard interruption.
`warning`	Display specified warning message.
`error`	Display specified error message.
`lasterr`	Return error message that was last issued.
`lasterror`	Return last error message and related information.
`lastwarn`	Return warning message that was last issued.

The MATLAB Graphical Debugger

Learn to use the MATLAB graphical debugger. You can view the function and its calling functions as you debug, set and clear breakpoints, single-step through the program, step into or over called functions, control visibility into all workspaces, and find and replace strings in your files.

Start out by opening the file you want to debug using **File > Open** or the `open` function. Use the debugging functions available on the toolbar and pull-down menus to set breakpoints, run or step through the program, and examine variables.

For more information: See Debugging M-Files and Using Debugging Features in the MATLAB Desktop Tools and Development Environment documentation.

A Quick Way to Examine Variables

To see the value of a variable from the Editor/Debugger window, hold the mouse cursor over the variable name for a second or two. You will see the value of the selected variable displayed.

Setting Breakpoints from the Command Line

You can set breakpoints with `dbstop` in any of the following ways:

- Break at a specific M-file line number.
- Break at the beginning of a specific subfunction.
- Break at the first executable line in an M-file.
- Break when a warning, or error, is generated.
- Break if any infinite or `NaN` values are encountered.

For more information: See Setting Breakpoints in the MATLAB Desktop Tools and Development Environment documentation.

Finding Line Numbers to Set Breakpoints

When debugging from the command line, a quick way to find line numbers for setting breakpoints is to use `dbtype`. The `dbtype` function displays all or part of an M-file, also numbering each line. To display delaunay`.m`, use

```
dbtype delaunay
```

To display only lines 35 through 41, use

```
dbtype delaunay 35:41
```

Stopping Execution on an Error or Warning

Use `dbstop if error` to stop program execution on any error and enter debug mode. Use `warning debug` to stop execution on any warning and enter debug mode.

For more information: See Debug, Backtrace, and Verbose Modes in the MATLAB Programming documentation.

Locating an Error from the Error Message

Click on the underlined text in an error message, and MATLAB opens the M-file being executed in its editor and places the cursor at the point of error.

For more information: See Types of Errors in the MATLAB Desktop Tools and Development Environment documentation.

Using Warnings to Help Debug

You can detect erroneous or unexpected behavior in your programs by inserting warning messages that MATLAB will display under the conditions you specify. See the section on "Warning Control" on page 8-24 in the MATLAB Programming documentation to find out how to selectively enable warnings.

For more information: See the `warning` function reference page.

Making Code Execution Visible

An easy way to see the end result of a particular line of code is to edit the program and temporarily remove the terminating semicolon from that line. Then, run your program and the evaluation of that statement is displayed on the screen.

For more information: See Finding Errors in the MATLAB Desktop Tools and Development Environment documentation.

Debugging Scripts

Scripts store their variables in a workspace that is shared with the caller of the script. So, when you debug a script from the command line, the script uses variables from the base workspace. To avoid errors caused by workspace sharing, type `clear all` before starting to debug your script to clear the base workspace.

Variables

In this section...

Rules for Variable Names

Although variable names can be of any length, MATLAB uses only the first N characters of the name, (where N is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first N characters to enable MATLAB to distinguish variables. Also note that variable names are case sensitive.

```
N = namelengthmax
N =
     63
```

For more information: See "Naming Variables" on page 3-6 in the MATLAB Programming documentation.

Making Sure Variable Names Are Valid

Before using a new variable name, you can check to see if it is valid with the `isvarname` function. Note that `isvarname` does not consider names longer than `namelengthmax` characters to be valid.

For example, the following name cannot be used for a variable since it begins with a number.

```
isvarname 8thColumn
ans =
     0
```

For more information: See “Naming Variables” on page 3-6 in the MATLAB Programming documentation.

Do Not Use Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name. If you do define a variable with a function name, you will not be able to call that function until you `clear` the variable from memory. (If it’s a MATLAB built-in function, then you will still be able to call that function but you must do so using `builtin`.)

To test whether a proposed variable name is already used as a function name, use

```
which -all name
```

For more information: See “Potential Conflict with Function Names” on page 3-7 in the MATLAB Programming documentation.

Checking for Reserved Keywords

MATLAB reserves certain keywords for its own use and does not allow you to override them. Attempts to use these words may result in any one of a number of error messages, some of which are shown here:

```
Error: Expected a variable, function, or constant, found "=".
Error: "End of Input" expected, "case" found.
Error: Missing operator, comma, or semicolon.
Error: "identifier" expected, "=" found.
```

Use the `iskeyword` function with no input arguments to list all reserved words.

Avoid Using i and j for Variables

MATLAB uses the characters `i` and `j` to represent imaginary units. Avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.

If you want to create a complex number without using `i` and `j`, you can use the `complex` function.

Avoid Overwriting Variables in Scripts

MATLAB scripts store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace. If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

For more information: See "M-File Scripts" on page 4-17 in the MATLAB Programming documentation.

Persistent Variables

To get the equivalent of a static variable in MATLAB, use `persistent`. When you declare a variable to be `persistent` within a function, its value is retained in memory between calls to that function. Unlike `global` variables, `persistent` variables are known only to the function in which they are declared.

For more information: See "Persistent Variables" on page 3-5 in the MATLAB Programming documentation.

Protecting Persistent Variables

You can inadvertently clear `persistent` variables from memory by either modifying the function in which the variables are defined, or by clearing the function with one of the following commands:

```
clear all
clear functions
```

Locking the M-file in memory with `mlock` prevents any `persistent` variables defined in the file from being reinitialized.

Global Variables

Use global variables sparingly. The global workspace is shared by all of your functions and also by your interactive MATLAB session. The more global variables you use, the greater the chances of unintentionally reusing a variable name, thus leaving yourself open to having those variables change in value unexpectedly. This can be a difficult bug to track down.

For more information: See "Global Variables" on page 3-3 in the MATLAB Programming documentation.

Strings

In this section...

Creating Strings with Concatenation

Strings are often created by concatenating smaller elements together (e.g., strings, values, etc.). Two common methods of concatenating are to use the MATLAB concatenation operator (`[]`) or the `sprintf` function. The second and third line below illustrate both of these methods. Both lines give the same result:

```
numChars = 28;
s = ['There are ' int2str(numChars) ' characters here']
s = sprintf('There are %d characters here\n', numChars)
```

For more information: See “Creating Character Arrays” on page 2-37 and “Converting from Numeric to String” on page 2-59 in the MATLAB Programming documentation.

Comparing Methods of Concatenation

When building strings with concatenation, `sprintf` is often preferable to `[]` because

- It is easier to read, especially when forming complicated expressions
- It gives you more control over the output format
- It often executes more quickly

You can also concatenate using the `strcat` function, However, for simple concatenations, `sprintf` and `[]` are faster.

Store Arrays of Strings in a Cell Array

It is usually best to store an array of strings in a cell array instead of a character array, especially if the strings are of different lengths. Strings in a character array must be of equal length, which often requires padding the strings with blanks. This is not necessary when using a cell array of strings that has no such requirement.

The `cellRecord` below does not require padding the strings with spaces:

```
cellRecord = {'Allison Jones'; 'Development'; 'Phoenix'};
```

For more information: See "Cell Arrays of Strings" on page 2-39 in the MATLAB Programming documentation.

Converting Between Strings and Cell Arrays

You can convert between standard character arrays and cell arrays of strings using the `cellstr` and `char` functions:

```
charRecord = ['Allison Jones'; 'Development  '; ...
              'Phoenix      '];
cellRecord = cellstr(charRecord);
```

Also, a number of the MATLAB string operations can be used with either character arrays, or cell arrays, or both:

```
cellRecord2 = {'Brian Lewis'; 'Development'; 'Albuquerque'};
strcmp(charRecord, cellRecord2)
ans =
     0
     1
     0
```

For more information: See "Converting to a Cell Array of Strings" on page 2-40 and "String Comparisons" on page 2-55 in the MATLAB Programming documentation.

Search and Replace Using Regular Expressions

Using regular expressions in MATLAB offers a very versatile way of searching for and replacing characters or phrases within a string. See the help on these functions for more information.

Function	Description
`regexp`	Match regular expression.
`regexpi`	Match regular expression, ignoring case.
`regexprep`	Replace string using regular expression.

For more information: See “Regular Expressions” on page 3-30 in the MATLAB Programming documentation.

Evaluating Expressions

In this section...

Find Alternatives to Using eval

While the `eval` function can provide a convenient solution to certain programming challenges, it is best to limit its use. The main reason is that code that uses `eval` is often difficult to read and hard to debug. A second reason is that `eval` statements cannot always be translated into C or C++ code by the MATLAB Compiler.

If you are evaluating a function, it is more efficient to use `feval` than `eval`. The `feval` function is made specifically for this purpose and is optimized to provide better performance.

For more information: See MATLAB Technical Note 1103, "What Is the EVAL Function, When Should I Use It, and How Can I Avoid It?" at URL `http://www.mathworks.com/support/tech-notes/1100/1103.html`.

Assigning to a Series of Variables

One common pattern for creating variables is to use a variable name suffixed with a number (e.g., `phase1`, `phase2`, `phase3`, etc.). We recommend using a cell array to build this type of variable name series, as it makes code more readable and executes more quickly than some other methods. For example:

```
for k = 1:800
   phase{k} = expression;
end
```

Short-Circuit Logical Operators

MATLAB has logical `AND` and `OR` operators (`&&` and `||`) that enable you to partially evaluate, or *short-circuit*, logical expressions. Short-circuit operators are useful when you want to evaluate a statement only when certain conditions are satisfied.

In this example, MATLAB does not execute the function `myfun` unless its M-file exists on the current path.

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

For more information: See "Short-Circuit Operators" on page 3-24 in the MATLAB Programming documentation.

Changing the Counter Variable within a for Loop

You cannot change the value of the loop counter variable (e.g., the variable `k` in the example below) in the body of a `for` loop. For example, this loop executes just `10` times, even though `k` is set back to `1` on each iteration.

```
for k = 1:10
   fprintf('Pass %d\n', k)
   k = 1;
end
```

Although MATLAB does allow you to use a variable of the same name as the loop counter within a loop, this is not a recommended practice.

MATLAB Path

In this section...

Precedence Rules

When MATLAB is given a name to interpret, it determines its usage by checking the name against each of the entities listed below, and in the order shown:

1. Variable
2. Subfunction
3. Private function
4. Class constructor
5. Overloaded method
6. M-file in the current directory
7. M-file on the path, or MATLAB built-in function

If you have two or more M-files on the path that have the same name, MATLAB selects the function that has its M-file in the directory closest to the beginning of the path string.

For more information: See “Function Precedence Order” on page 9-73 in the MATLAB Programming documentation.

File Precedence

If you refer to a file by its filename only (leaving out the file extension), and there is more than one file of this name in the directory, MATLAB selects the file to use according to the following precedence:

1. MEX-file
2. MDL-file (Simulink® model)
3. P-Code file
4. M-file

For more information: See “Multiple Implementation Types” on page 4-55 in the MATLAB Programming documentation.

Adding a Directory to the Search Path

To add a directory to the search path, use either of the following:

- At the toolbar, select **File > Set Path**.
- At the command line, use the `addpath` function.

You can also add a directory and all of its subdirectories in one operation by either of these means. To do this from the command line, use `genpath` together with `addpath`. The online help for the `genpath` function shows how to do this.

This example adds `/control` and all of its subdirectories to the MATLAB path:

```
addpath(genpath('K:/toolbox/control'))
```

For more information: See Search Path in the MATLAB Desktop Tools and Development Environment documentation.

Handles to Functions Not on the Path

You cannot create function handles to functions that are not on the MATLAB path. But you can achieve essentially the same thing by creating the handles through a script file placed in the same off-path directory as the functions.

If you then run the script, using `run path/script`, you will have created the handles that you need.

For example,

1. Create a script in this off-path directory that constructs function handles and assigns them to variables. That script might look something like this:

```
File E:/testdir/createFhandles.m
   fhset = @setItems
   fhsort = @sortItems
   fhdel = @deleteItem
```

2. Run the script from your current directory to create the function handles:

```
run E:/testdir/createFhandles
```

3. You can now execute one of the functions by means of its handle.

```
fhset(item, value)
```

Making Toolbox File Changes Visible to MATLAB

Unlike functions in user-supplied directories, M-files (and MEX-files) in the *matlabroot*/toolbox directories are not time-stamp checked, so MATLAB does not automatically see changes to them. If you modify one of these files, and then rerun it, you may find that the behavior does not reflect the changes that you made. This is most likely because MATLAB is still using the previously loaded version of the file.

To force MATLAB to reload a function from disk, you need to explicitly clear the function from memory using `clear functionname`. Note that there are rare cases where `clear` will not have the desired effect, (for example, if the file is locked, or if it is a class constructor and objects of the given class exist in memory).

Similarly, MATLAB does not automatically detect the presence of new files in *matlabroot*/toolbox directories. If you add (or remove) files from these directories, use `rehash toolbox` to force MATLAB to see your changes. Note that if you use the MATLAB Editor to create files, these steps are unnecessary, as the Editor automatically informs MATLAB of such changes.

Making Nontoolbox File Changes Visible to MATLAB

For M-files outside of the toolbox directories, MATLAB sees the changes made to these files by comparing timestamps and reloads any file that has changed the next time you execute the corresponding function.

If MATLAB does not see the changes you make to one of these files, try clearing the old copy of the function from memory using `clear` *`functionname`*. You can verify that MATLAB has cleared the function using `inmem` to list all functions currently loaded into memory.

Change Notification on Windows

If MATLAB, running on Windows, is unable to see new files or changes you have made to an existing file, the problem may be related to operating system change notification handles.

Type the following for more information:

```
help changeNotification
help changeNotificationAdvanced
```

Program Control

In this section...

Using break, continue, and return

It's easy to confuse the `break`, `continue`, and `return` functions as they are similar in some ways. Make sure you use these functions appropriately.

Function	Where to Use It	Description
`break`	`for` or `while` loops	Exits the loop in which it appears. In nested loops, control passes to the next outer loop.
`continue`	`for` or `while` loops	Skips any remaining statements in the current loop. Control passes to next iteration of the same loop.
`return`	Anywhere	Immediately exits the function in which it appears. Control passes to the caller of the function.

Using switch Versus if

It is possible, but usually not advantageous, to implement `switch-case` statements using `if-elseif` instead. See pros and cons in the table.

switch-case Statements	if-elseif Statements
Easier to read.	Can be difficult to read.
Can compare strings of different lengths.	You need `strcmp` to compare strings of different lengths.
Test for equality only.	Test for equality or inequality.

MATLAB case Evaluates Strings

A useful difference between `switch-case` statements in MATLAB and C is that you can specify string values in MATLAB `case` statements, which you cannot do in C.

```
switch(method)
   case 'linear'
      disp('Method is linear')
   case 'cubic'
      disp('Method is cubic')
end
```

Multiple Conditions in a case Statement

You can test against more than one condition with `switch`. The first case below tests for either a `linear` or `bilinear` method by using a cell array in the case statement.

```
switch(method)
   case {'linear', 'bilinear'}
      disp('Method is linear or bilinear')
   case (<and so on>)
end
```

Implicit Break in switch-case

In C, if you do not end each `case` with a `break` statement, code execution falls through to the following `case`. In MATLAB, `case` statements do not fall

through; only one `case` may execute. Using `break` within a `case` statement is not only unnecessary, it is also invalid and generates a warning.

In this example, if `result` is `52`, only the first `disp` statement executes, even though the second is also a valid match:

```
switch(result)
   case 52
      disp('result is 52')
   case {52, 78}
      disp('result is 52 or 78')
end
```

Variable Scope in a switch

Since MATLAB executes only one `case` of any `switch` statement, variables defined within one `case` are not known in the other `cases` of that `switch` statement. The same holds true for `if-elseif` statements.

In these examples, you get an error when `choice` equals `2`, because `x` is undefined.

```
  -- SWITCH-CASE --
switch choice
   case 1
      x = -pi:0.01:pi;
   case 2
      plot(x, sin(x));
end
```

```
 -- IF-ELSEIF --

if choice == 1
   x = -pi:0.01:pi;
elseif choice == 2
   plot(x, sin(x));
end
```

Catching Errors with try-catch

When you have statements in your code that could possibly generate unwanted results, put those statements into a `try-catch` block that will catch any errors and handle them appropriately.

The example below shows a `try-catch` block within a function that multiplies two matrices. If a statement in the `try` segment of the block fails, control passes to the `catch` segment. In this case, the `catch` statements check the error message that was issued (returned by `lasterr`) and respond appropriately.

```
try
   X = A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
end
```

For more information: See “The try-catch Statement” on page 8-17 in the MATLAB Programming documentation.

Nested try-catch Blocks

You can also nest `try-catch` blocks, as shown here. You can use this to attempt to recover from an error caught in the first `try` section:

```
try
   statement1                    % Try to execute statement1
catch
   try
      statement2                 % Attempt to recover from error
   catch
      disp 'Operation failed'    % Handle the error
   end
end
```

Forcing an Early Return from a Function

To force an early return from a function, place a `return` statement in the function at the point where you want to exit. For example,

```
if <done>
   return
end
```

Save and Load

In this section...

Saving Data from the Workspace

To save data from your workspace, you can do any of the following:

- Copy from the MATLAB Command Window and paste into a text file.
- Record part of your session in a `diary` file, and then edit the file in a text editor.
- Save to a binary or ASCII file using the `save` function.
- Save spreadsheet, scientific, image, or audio data with appropriate function.
- Save to a file using low-level file I/O functions (`fwrite`, `fprintf`, `...`).

For more information: See Saving the Current Workspace in the MATLAB Desktop Tools and Development Environment documentation, "Using the diary Function to Export Data" on page 6-87, and "Using Low-Level File I/O Functions" on page 6-104.

Loading Data into the Workspace

Similarly, to load new or saved data into the workspace, you can do any of the following:

- Enter or paste data at the command line.
- Create a script file to initialize large matrices or data structures.

- Read a binary or ASCII file using `load`.
- Load spreadsheet, scientific, image, or audio data with appropriate function.
- Load from a file using low-level file I/O functions (`fread`, `fscanf`, `...`).

For more information: See Loading a Saved Workspace and Importing Data in the MATLAB Development Environment documentation, and "Using Low-Level File I/O Functions" on page 6-104.

Viewing Variables in a MAT-File

To see what variables are saved in a MAT-file, use `who` or `whos` as shown here (the `.mat` extension is not required). `who` returns a cell array and `whos` returns a structure array.

```
mydataVariables = who('-file', 'mydata.mat');
```

Appending to a MAT-File

To save additional variables to an existing MAT-file, use

```
save matfilename -append
```

Any variables you save that do not yet exist in the MAT-file are added to the file. Any variables you save that already exist in the MAT-file overwrite the old values.

Note Saving with the `-append` switch does not append additional elements to an array that is already saved in a MAT-file. See the example below.

In this example, the second `save` operation does not concatenate new elements to vector `A`, (making `A` equal to `[1 2 3 4 5 6 7 8]`) in the MAT-file. Instead, it replaces the 5 element vector, `A`, with a 3 element vector, also retaining all other variables that were stored on the first `save` operation.

```
A = [1 2 3 4 5];   B = 12.5;   C = rand(4);
save savefile;
A = [6 7 8];
```

```
save savefile A -append;
```

Save and Load on Startup or Quit

You can automatically save your variables at the end of each MATLAB session by creating a `finish.m` file to `save` the contents of your base workspace every time you quit MATLAB. Load these variables back into your workspace at the beginning of each session by creating a `startup.m` file that uses the `load` function to load variables from your MAT-file.

For more information: See the `startup` and `finish` function reference pages.

Saving to an ASCII File

When you save matrix data to an ASCII file using `save -ascii`, MATLAB combines the individual matrices into one collection of numbers. Variable names are not saved. If this is not acceptable for your application, use `fprintf` to store your data instead.

For more information: See “Exporting Delimited ASCII Data Files” on page 6-86.

Files and Filenames

In this section...

Naming M-files

M-file names must start with an alphabetic character, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed M-file name length (returned by the function `namelengthmax`).

```
N = namelengthmax
N =
    63
```

Since variables must obey similar rules, you can use the `isvarname` function to check whether a filename (minus its `.m` file extension) is valid for an M-file.

```
isvarname mfilename
```

Naming Other Files

The names of other files that MATLAB interacts with (e.g., MAT, MEX, and MDL-files) follow the same rules as M-files, but may be of any length.

Depending on your operating system, you may be able to include certain nonalphanumeric characters in your filenames. Check your operating system manual for information on valid filename restrictions.

Passing Filenames as Arguments

In MATLAB commands, you can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`).

```
load mydata.mat           % Command syntax
load('mydata.mat')        % Function syntax
```

If you assign the output to a variable, you must use the function syntax.

```
savedData = load('mydata.mat')
```

Passing Filenames to ASCII Files

ASCII files are specified as follows. Here, the file extension is required.

```
load mydata.dat -ascii            % Command syntax
load('mydata.dat','-ascii')       % Function syntax
```

Determining Filenames at Run-Time

There are several ways that your function code can work on specific files without you having to hardcode their filenames into the program. You can

- Pass the filename in as an argument

  ```
  function myfun(datafile)
  ```

- Prompt for the filename using the `input` function

  ```
  filename = input('Enter name of file:  ', 's');
  ```

- Browse for the file using the `uigetfile` function

  ```
  [filename, pathname] =
     uigetfile('*.mat', 'Select MAT-file');
  ```

For more information: See the `input` and `uigetfile` function reference pages.

Returning the Size of a File

Two ways to have your program determine the size of a file are shown here.

```
  -- METHOD #1 --
s = dir('myfile.dat');
filesize = s.bytes
```

```
   -- METHOD #2 --
fid = fopen('myfile.dat');
fseek(fid, 0, 'eof');
filesize = ftell(fid)
fclose(fid);
```

The `dir` function also returns the filename (`s.name`), last modification date (`s.date`), and whether or not it's a directory (`s.isdir`).

(The second method requires read access to the file.)

For more information: See the `fopen`, `fseek`, `ftell`, and `fclose` function reference pages.

Input/Output

In this section...

For more information and examples on importing and exporting data, see Technical Note 1602:

http://www.mathworks.com/support/tech-notes/1600/1602.html

File I/O Function Overview

For a good overview of MATLAB file I/O functions, use the online "Functions — Categorical List" reference. In the Help browser **Contents**, select **MATLAB > Functions — Categorical List**, and then click **File I/O**.

Common I/O Functions

The most commonly used, high-level, file I/O functions in MATLAB are `save` and `load`. For help on these, type `doc save` or `doc load`.

Functions for I/O to text files with delimited values are `textread`, `dlmread`, `dlmwrite`. Functions for I/O to text files with comma-separated values are `csvread`, `csvwrite`.

For more information: See Text Files in the MATLAB "Functions — Categorical List" reference documentation.

Readable File Formats

Type `doc fileformats` to see a list of file formats that MATLAB can read, along with the associated MATLAB functions.

Using the Import Wizard

A quick method of importing text or binary data from a file (e.g., Excel files) is to use the MATLAB Import Wizard. Open the Import Wizard with the command, `uiimport filename` or by selecting **File > Import Data** at the Command Window.

Specify or browse for the file containing the data you want to import and you will see a preview of what the file contains. Select the data you want and click **Finish**.

For more information: See "Using the Import Wizard" on page 6-11 in the MATLAB Programming documentation.

Loading Mixed Format Data

To load data that is in mixed formats, use `textread` instead of `load`. The `textread` function lets you specify the format of each piece of data.

If the first line of file `mydata.dat` is

```
Sally    12.34 45
```

Read the first line of the file as a free format file using the % format:

```
[names, x, y] = textread('mydata.dat', '%s %f %d', 1)
```

returns

```
names =
    'Sally'
x =
  12.34000000000000
y =
    45
```

Reading Files with Different Formats

Attempting to read data from a file that was generated on a different platform may result in an error because the binary formats of the platforms may differ. Using the `fopen` function, you can specify a machine format when you open the file to avoid these errors.

Reading ASCII Data into a Cell Array

A common technique used to read an ASCII data file into a cell array is

```
[a,b,c,d] = textread('data.txt', '%s %s %s %s');
mydata = cellstr([a b c d]);
```

For more information: See the `textread` and `cellstr` function reference pages.

Interactive Input into Your Program

Your program can accept interactive input from users during execution. Use the `input` function to prompt the user for input, and then read in a response. When executed, `input` causes the program to display your prompt, pause while a response is entered, and then resume when the **Enter** key is pressed.

Starting MATLAB

Getting MATLAB to Start Up Faster

Here are some things that you can do to make MATLAB start up faster.

- Make sure toolbox path caching is enabled.
- Make sure that the system on which MATLAB is running has enough RAM.
- Choose only the windows you need in the MATLAB desktop.
- Close the Help browser before exiting MATLAB. When you start your next session, MATLAB will not open the Help browser, and thus will start faster.
- If disconnected from the network, check the `LM_LICENSE_FILE` variable. See http://www.mathworks.com/support/solutions/data/1-17VEB.html for a more detailed explanation.

For more information: See Reduced Startup Time with Toolbox Path Caching in the MATLAB Desktop Tools and Development Environment documentation.

Operating System Compatibility

In this section...

Executing O/S Commands from MATLAB

To execute a command from your operating system prompt without having to exit MATLAB, precede the command with the MATLAB `!` operator.

On Windows, you can add an ampersand (`&`) to the end of the line to make the output appear in a separate window.

For more information: See Running External Programs in the MATLAB Desktop Tools and Development Environment documentation, and the `system` and `dos` function reference pages.

Searching Text with grep

`grep` is a powerful tool for performing text searches in files on UNIX systems. To `grep` from within MATLAB, precede the command with an exclamation point (`!grep`).

For example, to search for the word `warning`, ignoring case, in all M-files of the current directory, you would use

```
!grep -i 'warning' *.m
```

Constructing Paths and Filenames

Use the `fullfile` function to construct path names and filenames rather than entering them as strings into your programs. In this way, you always get the correct path specification, regardless of which operating system you are using at the time.

Finding the MATLAB Root Directory

The `matlabroot` function returns the location of the MATLAB installation on your system. Use `matlabroot` to create a path to MATLAB and toolbox directories that does not depend on a specific platform or MATLAB version.

The following example uses `matlabroot` with `fullfile` to return a platform-independent path to the `general` toolbox directory:

```
fullfile(matlabroot,'toolbox','matlab','general')
```

Temporary Directories and Filenames

If you need to locate the directory on your system that has been designated to hold temporary files, use the `tempdir` function. `tempdir` returns a string that specifies the path to this directory.

To create a new file in this directory, use the `tempname` function. `tempname` returns a string that specifies the path to the temporary file directory, plus a unique filename.

For example, to store some data in a temporary file, you might issue the following command first.

```
fid = fopen(tempname, 'w');
```

Demos

Demos Available with MATLAB

MATLAB comes with a wide array of visual demonstrations to help you see the extent of what you can do with the product. To start running any of the demos, simply type `demo` at the MATLAB command prompt. Demos cover the following major areas:

- MATLAB
- Toolboxes
- Simulink
- Blocksets
- Real-Time Workshop®
- Stateflow®

For more information: See Demos in the Help Browser in the MATLAB Desktop Tools and Development Environment documentation, and the `demo` function reference page.

For More Information

In this section...

Current CSSM

```
http://newsreader.mathworks.com/WebX?14@@/comp.soft-sys.matlab
```

Archived CSSM

```
http://mathforum.org/kb/forum.jspa?forumID=80
```

MATLAB Technical Support

```
http://www.mathworks.com/support/
```

Tech Notes

```
http://www.mathworks.com/support/tech-notes/list_all.html
```

MATLAB Central

```
http://www.mathworks.com/matlabcentral/
```

MATLAB Newsletters (Digest, News & Notes)

```
http://www.mathworks.com/company/newsletters/index.html
```

MATLAB Documentation

```
http://www.mathworks.com/access/helpdesk/help/helpdesk.html
```

MATLAB Index of Examples

```
http://www.mathworks.com/access/helpdesk/help/techdoc/demo_example.shtml
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

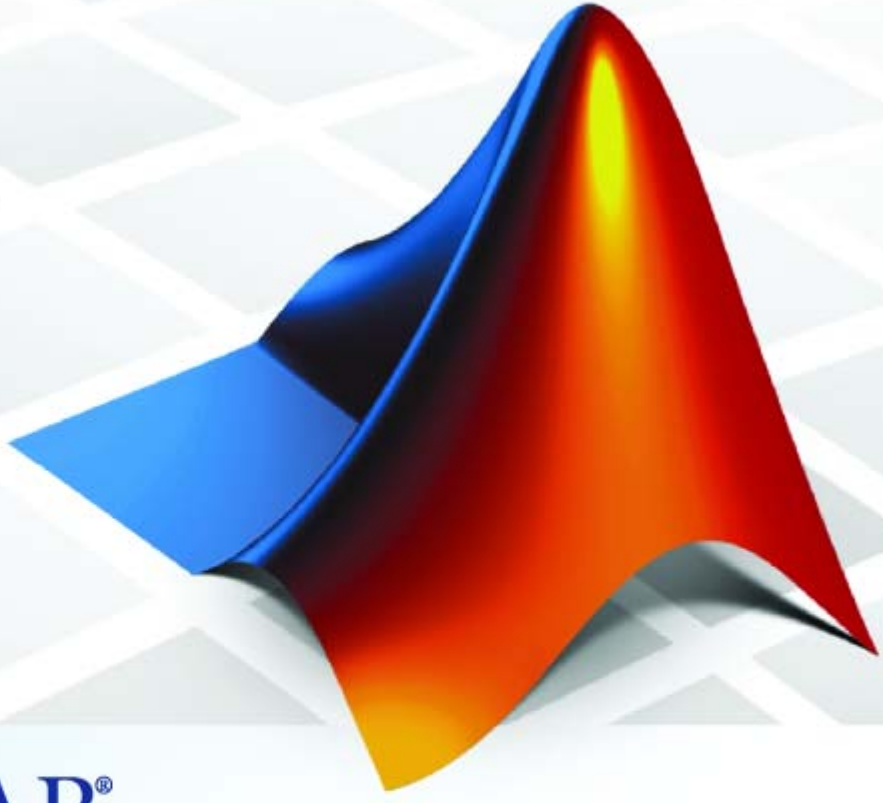
U

V

W

Z

MATLAB® 7
Programming Tips



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Programming Tips

Revision History

July 2002	Online only	New for MATLAB 6.5 (Release 13)
June 2004	Online only	Revised for MATLAB 7.0 (Release 14)
March 2005	Online only	Minor revision for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Minor revision for MATLAB 7.1 (Release 14SP3)
September 2007	Online only	Minor revision for MATLAB 7.5 (Release 2007b)

Contents

1 Programming Tips

1

Programming Tips

Introduction

This section is a categorized compilation of tips for the MATLAB® programmer. Each item is relatively brief to help you browse through them and find information that is useful. Many of the tips include a reference to specific MATLAB documentation that gives you more complete coverage of the topic. You can find information on the following topics:

For suggestions on how to improve the performance of your MATLAB programs, and how to write programs that use memory more efficiently, see Improving Performance and Memory Usage

Command and Function Syntax

In this section...

Syntax Help

For help about the general syntax of MATLAB functions and commands, type

```
help syntax
```

Command and Function Syntaxes

You can enter MATLAB commands using either a *command* or *function* syntax. It is important to learn the restrictions and interpretation rules for both.

```
functionname arg1 arg2 arg3               % Command syntax
functionname('arg1','arg2','arg3')        % Function syntax
```

For more information: See Calling Functions in the MATLAB Programming documentation.

Command Line Continuation

You can continue most statements to one or more additional lines by terminating each incomplete line with an ellipsis (`...`). Breaking down a statement into a number of lines can sometimes result in a clearer programming style.

```
sprintf ('Example %d shows a command coded on %d lines.\n', ...
          exampleNumber, ...
```

```
            numberOfLines)
```

Note that you cannot continue an incomplete string to another line.

```
disp 'This statement attempts to continue a string ...
      to another line, resulting in an error.'
```

For more information: See Entering Long Statements in the MATLAB Desktop Tools and Development Environment documentation.

Completing Commands Using the Tab Key

You can save some typing when entering commands by entering only the first few letters of the command, variable, property, etc. followed by the **Tab** key. Typing the second line below (with **T** representing **Tab**) yields the expanded, full command shown in the third line:

```
f = figure;
set(f, 'papTuT,'cT)                    % Type this line.
set(f, 'paperunits','centimeters')     % This is what you get.
```

If there are too many matches for the string you are trying to complete, you will get no response from the first **Tab**. Press **Tab** again to see all possible choices:

```
set(f, 'paTT
PaperOrientation   PaperPositionMode  PaperType     Parent
PaperPosition      PaperSize          PaperUnits
```

For more information: See Tab Completion in the Command Window in the MATLAB Desktop Tools and Development Environment documentation

Recalling Commands

Use any of the following methods to simplify recalling previous commands to the screen:

- To recall an earlier command to the screen, press the up arrow key one or more times, until you see the command you want. If you want to modify the recalled command, you can edit its text before pressing **Enter** or **Return** to execute it.

- To recall a specific command by name without having to scroll through your earlier commands one by one, type the starting letters of the command, followed by the up arrow key.
- Open the Command History window (**View > Command History**) to see all previous commands. Double-click the command you want to execute.

For more information: See Recalling Previous Lines and Command History Window in the MATLAB Desktop Tools and Development Environment documentation.

Clearing Commands

If you have typed a command that you then decide not to execute, you can clear it from the Command Window by pressing the Escape (**Esc**) key.

Suppressing Output to the Screen

To suppress output to the screen, end statements with a semicolon. This can be particularly useful when generating large matrices.

```
A = magic(100);     % Create matrix A, but do not display it.
```

Help

In this section...

Using the Help Browser

Open the Help browser from the MATLAB Command Window using one of the following:

- Click the question mark symbol in the toolbar.
- Select **Help > Product Help** from the menu.
- Type the word `doc` at the command prompt.

Some of the features of the Help browser are listed below.

Feature	Description
Product Filter	Establish which products to find help on.
Contents	Look up topics in the Table of Contents.
Index	Look up help using the documentation Index.
Search	Search the documentation for one or more words.
Demos	See what demos are available; run selected demos.
Favorites	Save bookmarks for frequently used Help pages.

For more information: See Finding Information with the Help Browser in the MATLAB Desktop Tools and Development Environment documentation.

Help on Functions from the Help Browser

To find help on any function from the Help browser, do either of the following:

- Select the **Contents** tab of the Help browser, open the **Contents** entry labeled `MATLAB`, and find the two subentries shown below. Use one of these to look up the function you want help on.
  - Functions — Categorical List
  - Functions — Alphabetical List
- Type `doc functionname` at the command line.

Help on Functions from the Command Window

Several types of help on functions are available from the Command Window:

- To list all categories that you can request help on from the Command Window, just type

  ```
  help
  ```

- To see a list of functions for one of these categories, along with a brief description of each function, type `help` *`category`*. For example,

  ```
  help datafun
  ```

- To get help on a particular function, type `help functionname`. For example,

  ```
  help sortrows
  ```

Topical Help

In addition to the help on individual functions, you can get help on any of the following topics by typing `help topicname` at the command line.

Topic Name	Description
`arith`	Arithmetic operators
`relop`	Relational and logical operators
`punct`	Special character operators
`slash`	Arithmetic division operators
`paren`	Parentheses, braces, and bracket operators
`precedence`	Operator precedence
`datatypes`	MATLAB data types, their associated functions, and operators that you can overload
`lists`	Comma separated lists
`strings`	Character strings
`function_handle`	Function handles and the `@` operator
`debug`	Debugging functions
`java`	Using Java from within MATLAB
`fileformats`	A list of readable file formats
`changeNotification`	Windows directory change notification

Paged Output

Before displaying a lengthy section of help text or code, put MATLAB into its paged output mode by typing `more on`. This breaks up any ensuing display into pages for easier viewing. Turn off paged output with `more off`.

Page through the displayed text using the space bar key. Or step through line by line using **Enter** or **Return**. Discontinue the display by pressing the **Q** key or **Ctrl+C**.

Writing Your Own Help

Start each program you write with a section of text providing help on how and when to use the function. If formatted properly, the MATLAB `help` function displays this text when you enter

```
help functionname
```

MATLAB considers the first group of consecutive lines immediately following the function definition line that begin with `%` to be the help section for the function. The first line without `%` as the left-most character ends the help.

For more information: See Help Text in the MATLAB Desktop Tools and Development Environment documentation.

Help for Subfunctions and Private Functions

You can write help for subfunctions using the same rules that apply to main functions. To display the help for the subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

To display the help for a private function, precede the function name with `private/`. To get help on private function `myprivfun`, type

```
help private/myprivfun
```

Help for Methods and Overloaded Functions

You can write help text for object-oriented class methods implemented with M-files. Display help for the method by typing

```
help classname/methodname
```

where the file `methodname.m` resides in subdirectory `@classname`.

For example, if you write a `plot` method for a class named `polynom`, (where the `plot` method is defined in the file `@polynom/plot.m`), you can display this help by typing

```
help polynom/plot
```

You can get help on overloaded MATLAB functions in the same way. To display the help text for the `eq` function as implemented in `matlab/iofun/@serial`, type

```
help serial/eq
```

Development Environment

In this section...

Workspace Browser

The Workspace browser is a graphical interface to the variables stored in the MATLAB base and function workspaces. You can view, modify, save, load, and create graphics from workspace data using the browser. Select **View > Workspace** to open the browser.

To view function workspaces, you need to be in debug mode.

For more information: See MATLAB Workspace in the MATLAB Desktop Tools and Development Environment documentation.

Using the Find and Replace Utility

Find any word or phrase in a group of files using the Find and Replace utility. Click **View > Current Directory**, and then click the binoculars icon at the top of the **Current Directory** window.

When entering search text, you do not need to put quotes around a phrase. In fact, parts of words, like `win` for `windows`, will not be found if enclosed in quotes.

For more information: See Finding and Replacing Text in the Current File in the MATLAB Desktop Tools and Development Environment documentation.

Commenting Out a Block of Code

To comment out a block of text or code within the MATLAB editor,

1. Highlight the block of text you would like to comment out.
2. Holding the mouse over the highlighted text, select **Text > Comment** (or **Uncomment**, to do the reverse) from the toolbar. (You can also get these options by right-clicking the mouse.)

For more information: See Adding Comments in the MATLAB Desktop Tools and Development Environment documentation.

Creating M-Files from Command History

If there is part of your current MATLAB session that you would like to put into an M-file, this is easily done using the Command History window:

1. Open this window by selecting **View > Command History**.
2. Use **Shift+Click** or **Ctrl+Click** to select the lines you want to use. MATLAB highlights the selected lines.
3. Right-click once, and select **Create M-File** from the menu that appears. MATLAB creates a new Editor window displaying the selected code.

Editing M-Files in EMACS

If you use Emacs, you can download editing modes for editing M-files with GNU-Emacs or with early versions of Emacs from the MATLAB Central Web site:

```
http://www.mathworks.com/matlabcentral/
```

At this Web site, select **File Exchange**, and then **Utilities > Emacs**.

For more information: See General Preferences for the Editor/Debugger in the MATLAB Desktop Tools and Development Environment documentation.

M-File Functions

In this section...

M-File Structure

An M-File consists of the components shown here:

```
function [x, y] = myfun(a, b, c)   % Function definition line
% H1 line -- A one-line summary of the function's purpose.
% Help text -- One or more lines of help text that explain
%   how to use the function. This text is displayed when
%   the user types "help functionname".

% The Function body normally starts after the first blank line.
% Comments -- Description (for internal use) of what the
%   function does, what inputs are expected, what outputs
%   are generated. Typing "help functionname" does not display
%   this text.

x = prod(a, b);                    % Start of Function code
```

For more information: See Basic Parts of an M-File in the MATLAB Programming documentation.

Using Lowercase for Function Names

Function names appear in uppercase in MATLAB help text only to make the help easier to read. In practice, however, it is usually best to use lowercase when calling functions.

For M-file functions, case requirements depend on the case sensitivity of the operating system you are using. As a rule, naming and calling functions using lowercase generally makes your M-files more portable from one operating system to another.

Getting a Function's Name and Path

To obtain the name of an M-file that is currently being executed, use the following function in your M-file code.

```
mfilename
```

To include the path along with the M-file name, use

```
mfilename('fullpath')
```

For more information: See the `mfilename` function reference page.

What M-Files Does a Function Use?

For a simple display of all M-files referenced by a particular function, follow the steps below:

1. Type `clear functions` to clear all functions from memory (see Note below).
2. Execute the function you want to check. Note that the function arguments you choose to use in this step are important, since you can get different results when calling the same function with different arguments.
3. Type `inmem` to display all M-Files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output, as shown here:

   ```
   [mfiles, mexfiles] = inmem
   ```

Note `clear functions` does not clear functions locked by `mlock`. If you have locked functions, (which you can check using `inmem`), unlock them with `munlock`, and then repeat step 1.

Dependent Functions, Built-Ins, Classes

For a much more detailed display of dependent function information, use the `depfun` function. In addition to M-files, `depfun` shows which built-ins and classes a particular function depends on.

Function Arguments

In this section...

Getting the Input and Output Arguments

Use `nargin` and `nargout` to determine the number of input and output arguments in a particular function call. Use `nargchk` and `nargoutchk` to verify that your function is called with the required number of input and output arguments.

```
function [x, y] = myplot(a, b, c, d)
disp(nargchk(2, 4, nargin))           % Allow 2 to 4 inputs
disp(nargoutchk(0, 2, nargout))       % Allow 0 to 2 outputs

x = plot(a, b);
if nargin == 4
   y = myfun(c, d);
end
```

Variable Numbers of Arguments

You can call functions with fewer input and output arguments than you have specified in the function definition, but not more. If you want to call a function with a variable number of arguments, use the `varargin` and `varargout` function parameters in the function definition.

This function returns the size vector and, optionally, individual dimensions:

```
function [s, varargout] = mysize(x)
nout = max(nargout, 1) - 1;
s = size(x);
for k = 1:nout
```

```
    varargout(k) = {s(k)};
end
```

Try calling it with

```
[s, rows, cols] = mysize(rand(4, 5))
```

String or Numeric Arguments

If you are passing only string arguments into a function, you can use MATLAB command syntax. All arguments entered in command syntax are interpreted as strings.

```
strcmp string1 string1
ans =
     1
```

When passing numeric arguments, it is best to use function syntax unless you want the number passed as a string. The right-hand example below passes the number `75` as the string, `'75'`.

```
isnumeric(75)
ans =
    1
```

```
isnumeric 75
ans =
    0
```

For more information: See Passing Arguments with Command and Function Syntax in the MATLAB Programming documentation.

Passing Arguments in a Structure

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure and pass the structure. Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields.

Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The disadvantage over structures is that you do not have field names to describe each variable. The advantage is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function.

Program Development

In this section...

Planning the Program

When planning how to write a program, take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

Using Pseudo-Code

You may find it helpful to write the initial draft of your program in a structured format using your own natural language. This *pseudo-code* is often easier to think through, review, and modify than using a formal programming language, yet it is easily translated into a programming language in the next stage of development.

Selecting the Right Data Structures

Look at what data types and data structures are available to you in MATLAB and determine which of those best fit your needs in storing and passing your data.

For more information: See Data Types in the MATLAB Programming documentation.

General Coding Practices

A few suggested programming practices:

- Use descriptive function and variable names to make your code easier to understand.
- Order subfunctions alphabetically in an M-file to make them easier to find.
- Precede each subfunction with a block of help text describing what that subfunction does. This not only explains the subfunctions, but also helps to visually separate them.
- Do not extend lines of code beyond the 80th column. Otherwise, it will be hard to read when you print it out.
- Use full Handle Graphics® property and value names. Abbreviated names are often allowed, but can make your code unreadable. They also could be incompatible in future releases of MATLAB.

Naming a Function Uniquely

To avoid choosing a name for a new function that might conflict with a name already in use, check for any occurrences of the name using this command:

```
which -all functionname
```

For more information: See the `which` function reference page.

The Importance of Comments

Be sure to document your programs well to make it easier for you or someone else to maintain them. Add comments generously, explaining each major section and any smaller segments of code that are not obvious. You can add a block of comments as shown here.

```
%-------------------------------------------
% This function computes the ... <and so on>
%-------------------------------------------
```

For more information: See Comments in the MATLAB Programming documentation.

Coding in Steps

Do not try to write the entire program all at once. Write a portion of it, and then test that piece out. When you have that part working the way you want, then write the next piece, and so on. It's much easier to find programming errors in a small piece of code than in a large program.

Making Modifications in Steps

When making modifications to a working program, do not make widespread changes all at one time. It's better to make a few small changes, test and debug, make a few more changes, and so on. Tracking down a difficult bug in the small section that you've changed is much easier than trying to find it in a huge block of new code.

Functions with One Calling Function

If you have a function that is called by only one other function, put it in the same M-file as the calling function, making it a subfunction.

For more information: See Subfunctions in the MATLAB Programming documentation.

Testing the Final Program

One suggested practice for testing a new program is to step through the program in the MATLAB debugger while keeping a record of each line that gets executed on a printed copy of the program. Use different combinations of inputs until you have observed that every line of code is executed at least once.

Debugging

In this section...

The MATLAB Debug Functions

For a brief description of the main debug functions in MATLAB, type

```
help debug
```

For more information: See Debugging Process and Features in the MATLAB Desktop Tools and Development Environment documentation.

More Debug Functions

Other functions you may find useful in debugging are listed below.

Function	Description
`echo`	Display function or script code as it executes.
`disp`	Display specified values or messages.
`sprintf`, `fprintf`	Display formatted data of different types.

Function	Description
`whos`	List variables in the workspace.
`size`	Show array dimensions.
`keyboard`	Interrupt program execution and allow input from keyboard.
`return`	Resume execution following a keyboard interruption.
`warning`	Display specified warning message.
`error`	Display specified error message.
`lasterr`	Return error message that was last issued.
`lasterror`	Return last error message and related information.
`lastwarn`	Return warning message that was last issued.

The MATLAB Graphical Debugger

Learn to use the MATLAB graphical debugger. You can view the function and its calling functions as you debug, set and clear breakpoints, single-step through the program, step into or over called functions, control visibility into all workspaces, and find and replace strings in your files.

Start out by opening the file you want to debug using **File > Open** or the `open` function. Use the debugging functions available on the toolbar and pull-down menus to set breakpoints, run or step through the program, and examine variables.

For more information: See Debugging Process and Features in the MATLAB Desktop Tools and Development Environment documentation.

A Quick Way to Examine Variables

To see the value of a variable from the Editor/Debugger window, hold the mouse cursor over the variable name for a second or two. You will see the value of the selected variable displayed.

Setting Breakpoints from the Command Line

You can set breakpoints with `dbstop` in any of the following ways:

- Break at a specific M-file line number.
- Break at the beginning of a specific subfunction.
- Break at the first executable line in an M-file.
- Break when a warning, or error, is generated.
- Break if any infinite or `NaN` values are encountered.

For more information: See Setting Breakpoints in the MATLAB Desktop Tools and Development Environment documentation.

Finding Line Numbers to Set Breakpoints

When debugging from the command line, a quick way to find line numbers for setting breakpoints is to use `dbtype`. The `dbtype` function displays all or part of an M-file, also numbering each line. To display delaunay`.m`, use

```
dbtype delaunay
```

To display only lines 35 through 41, use

```
dbtype delaunay 35:41
```

Stopping Execution on an Error or Warning

Use `dbstop if error` to stop program execution on any error and enter debug mode. Use `warning debug` to stop execution on any warning and enter debug mode.

For more information: See Backtrace and Verbose Modes in the MATLAB Programming documentation.

Locating an Error from the Error Message

Click on the underlined text in an error message, and MATLAB opens the M-file being executed in its editor and places the cursor at the point of error.

For more information: See Finding Errors, Debugging, and Correcting M-Files in the MATLAB Desktop Tools and Development Environment documentation.

Using Warnings to Help Debug

You can detect erroneous or unexpected behavior in your programs by inserting warning messages that MATLAB will display under the conditions you specify. See the section on Warning Control in the MATLAB Programming documentation to find out how to selectively enable warnings.

For more information: See the `warning` function reference page.

Making Code Execution Visible

An easy way to see the end result of a particular line of code is to edit the program and temporarily remove the terminating semicolon from that line. Then, run your program and the evaluation of that statement is displayed on the screen.

For more information: See Finding Errors, Debugging, and Correcting M-Files in the MATLAB Desktop Tools and Development Environment documentation.

Debugging Scripts

Scripts store their variables in a workspace that is shared with the caller of the script. So, when you debug a script from the command line, the script uses variables from the base workspace. To avoid errors caused by workspace sharing, type `clear all` before starting to debug your script to clear the base workspace.

Variables

In this section...

Rules for Variable Names

Although variable names can be of any length, MATLAB uses only the first N characters of the name, (where N is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first N characters to enable MATLAB to distinguish variables. Also note that variable names are case sensitive.

```
N = namelengthmax
N =
     63
```

For more information: See Naming Variables in the MATLAB Programming documentation.

Making Sure Variable Names Are Valid

Before using a new variable name, you can check to see if it is valid with the `isvarname` function. Note that `isvarname` does not consider names longer than `namelengthmax` characters to be valid.

For example, the following name cannot be used for a variable since it begins with a number.

```
isvarname 8thColumn
ans =
     0
```

For more information: See Naming Variables in the MATLAB Programming documentation.

Do Not Use Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name. If you do define a variable with a function name, you will not be able to call that function until you `clear` the variable from memory. (If it's a MATLAB built-in function, then you will still be able to call that function but you must do so using `builtin`.)

To test whether a proposed variable name is already used as a function name, use

```
which -all name
```

For more information: See Potential Conflict with Function Names in the MATLAB Programming documentation.

Checking for Reserved Keywords

MATLAB reserves certain keywords for its own use and does not allow you to override them. Attempts to use these words may result in any one of a number of error messages, some of which are shown here:

```
Error: Expected a variable, function, or constant, found "=".
Error: "End of Input" expected, "case" found.
Error: Missing operator, comma, or semicolon.
Error: "identifier" expected, "=" found.
```

Use the `iskeyword` function with no input arguments to list all reserved words.

Avoid Using i and j for Variables

MATLAB uses the characters `i` and `j` to represent imaginary units. Avoid using `i` and `j` for variable names if you intend to use them in complex arithmetic.

If you want to create a complex number without using `i` and `j`, you can use the `complex` function.

Avoid Overwriting Variables in Scripts

MATLAB scripts store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace. If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

For more information: See M-File Scripts in the MATLAB Programming documentation.

Persistent Variables

To get the equivalent of a static variable in MATLAB, use `persistent`. When you declare a variable to be `persistent` within a function, its value is retained in memory between calls to that function. Unlike `global` variables, `persistent` variables are known only to the function in which they are declared.

For more information: See Persistent Variables in the MATLAB Programming documentation.

Protecting Persistent Variables

You can inadvertently clear `persistent` variables from memory by either modifying the function in which the variables are defined, or by clearing the function with one of the following commands:

```
clear all
clear functions
```

Locking the M-file in memory with `mlock` prevents any `persistent` variables defined in the file from being reinitialized.

Global Variables

Use global variables sparingly. The global workspace is shared by all of your functions and also by your interactive MATLAB session. The more global variables you use, the greater the chances of unintentionally reusing a variable name, thus leaving yourself open to having those variables change in value unexpectedly. This can be a difficult bug to track down.

For more information: See Global Variables in the MATLAB Programming documentation.

Strings

In this section...

Creating Strings with Concatenation

Strings are often created by concatenating smaller elements together (e.g., strings, values, etc.). Two common methods of concatenating are to use the MATLAB concatenation operator (`[]`) or the `sprintf` function. The second and third line below illustrate both of these methods. Both lines give the same result:

```
numChars = 28;
s = ['There are ' int2str(numChars) ' characters here']
s = sprintf('There are %d characters here\n', numChars)
```

For more information: See Creating Character Arrays and Converting from Numeric to String in the MATLAB Programming documentation.

Comparing Methods of Concatenation

When building strings with concatenation, `sprintf` is often preferable to `[]` because

- It is easier to read, especially when forming complicated expressions
- It gives you more control over the output format
- It often executes more quickly

You can also concatenate using the `strcat` function, However, for simple concatenations, `sprintf` and `[]` are faster.

Store Arrays of Strings in a Cell Array

It is usually best to store an array of strings in a cell array instead of a character array, especially if the strings are of different lengths. Strings in a character array must be of equal length, which often requires padding the strings with blanks. This is not necessary when using a cell array of strings that has no such requirement.

The `cellRecord` below does not require padding the strings with spaces:

```
cellRecord = {'Allison Jones'; 'Development'; 'Phoenix'};
```

For more information: See Cell Arrays of Strings in the MATLAB Programming documentation.

Converting Between Strings and Cell Arrays

You can convert between standard character arrays and cell arrays of strings using the `cellstr` and `char` functions:

```
charRecord = ['Allison Jones'; 'Development  '; ...
              'Phoenix      '];
cellRecord = cellstr(charRecord);
```

Also, a number of the MATLAB string operations can be used with either character arrays, or cell arrays, or both:

```
cellRecord2 = {'Brian Lewis'; 'Development'; 'Albuquerque'};
strcmp(charRecord, cellRecord2)
ans =
     0
     1
     0
```

For more information: See Converting to a Cell Array of Strings and String Comparisons in the MATLAB Programming documentation.

Search and Replace Using Regular Expressions

Using regular expressions in MATLAB offers a very versatile way of searching for and replacing characters or phrases within a string. See the help on these functions for more information.

Function	Description
`regexp`	Match regular expression.
`regexpi`	Match regular expression, ignoring case.
`regexprep`	Replace string using regular expression.

For more information: See “Regular Expressions” in the MATLAB Programming documentation.

Evaluating Expressions

In this section...

Find Alternatives to Using eval

While the `eval` function can provide a convenient solution to certain programming challenges, it is best to limit its use. The main reason is that code that uses `eval` is often difficult to read and hard to debug. A second reason is that `eval` statements cannot always be translated into C or C++ code by the MATLAB Compiler.

If you are evaluating a function, it is more efficient to use `feval` than `eval`. The `feval` function is made specifically for this purpose and is optimized to provide better performance.

For more information: See MATLAB Technical Note 1103, "What Is the EVAL Function, When Should I Use It, and How Can I Avoid It?" at URL `http://www.mathworks.com/support/tech-notes/1100/1103.html`.

Assigning to a Series of Variables

One common pattern for creating variables is to use a variable name suffixed with a number (e.g., `phase1`, `phase2`, `phase3`, etc.). We recommend using a cell array to build this type of variable name series, as it makes code more readable and executes more quickly than some other methods. For example:

```
for k = 1:800
   phase{k} = expression;
end
```

Short-Circuit Logical Operators

MATLAB has logical `AND` and `OR` operators (`&&` and `||`) that enable you to partially evaluate, or *short-circuit*, logical expressions. Short-circuit operators are useful when you want to evaluate a statement only when certain conditions are satisfied.

In this example, MATLAB does not execute the function `myfun` unless its M-file exists on the current path.

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

For more information: See “Short-Circuit Operators” in the MATLAB Programming documentation.

Changing the Counter Variable within a for Loop

You cannot change the value of the loop counter variable (e.g., the variable `k` in the example below) in the body of a `for` loop. For example, this loop executes just `10` times, even though `k` is set back to `1` on each iteration.

```
for k = 1:10
   fprintf('Pass %d\n', k)
   k = 1;
end
```

Although MATLAB does allow you to use a variable of the same name as the loop counter within a loop, this is not a recommended practice.

MATLAB Path

In this section...

Precedence Rules

When MATLAB is given a name to interpret, it determines its usage by checking the name against each of the entities listed below, and in the order shown:

1. Variable
2. Subfunction
3. Private function
4. Class constructor
5. Overloaded method
6. M-file in the current directory
7. M-file on the path, or MATLAB built-in function

If you have two or more M-files on the path that have the same name, MATLAB selects the function that has its M-file in the directory closest to the beginning of the path string.

For more information: See Function Precedence Order in the MATLAB Programming documentation.

File Precedence

If you refer to a file by its filename only (leaving out the file extension), and there is more than one file of this name in the directory, MATLAB selects the file to use according to the following precedence:

1. MEX-file
2. MDL-file (Simulink® model)
3. P-Code file
4. M-file

For more information: See Multiple Implementation Types in the MATLAB Programming documentation.

Adding a Directory to the Search Path

To add a directory to the search path, use either of the following:

- At the toolbar, select **File > Set Path**.
- At the command line, use the `addpath` function.

You can also add a directory and all of its subdirectories in one operation by either of these means. To do this from the command line, use `genpath` together with `addpath`. The online help for the `genpath` function shows how to do this.

This example adds `/control` and all of its subdirectories to the MATLAB path:

```
addpath(genpath('K:/toolbox/control'))
```

For more information: See Search Path in the MATLAB Desktop Tools and Development Environment documentation.

Handles to Functions Not on the Path

You cannot create function handles to functions that are not on the MATLAB path. But you can achieve essentially the same thing by creating the handles through a script file placed in the same off-path directory as the functions.

If you then run the script, using `run path/script`, you will have created the handles that you need.

For example,

1. Create a script in this off-path directory that constructs function handles and assigns them to variables. That script might look something like this:

```
File E:/testdir/createFhandles.m
   fhset = @setItems
   fhsort = @sortItems
   fhdel = @deleteItem
```

2. Run the script from your current directory to create the function handles:

```
run E:/testdir/createFhandles
```

3. You can now execute one of the functions by means of its handle.

```
fhset(item, value)
```

Making Toolbox File Changes Visible to MATLAB

Unlike functions in user-supplied directories, M-files (and MEX-files) in the *matlabroot*/toolbox directories are not time-stamp checked, so MATLAB does not automatically see changes to them. If you modify one of these files, and then rerun it, you may find that the behavior does not reflect the changes that you made. This is most likely because MATLAB is still using the previously loaded version of the file.

To force MATLAB to reload a function from disk, you need to explicitly clear the function from memory using `clear functionname`. Note that there are rare cases where `clear` will not have the desired effect, (for example, if the file is locked, or if it is a class constructor and objects of the given class exist in memory).

Similarly, MATLAB does not automatically detect the presence of new files in *matlabroot*/toolbox directories. If you add (or remove) files from these directories, use `rehash toolbox` to force MATLAB to see your changes. Note that if you use the MATLAB Editor to create files, these steps are unnecessary, as the Editor automatically informs MATLAB of such changes.

Making Nontoolbox File Changes Visible to MATLAB

For M-files outside of the toolbox directories, MATLAB sees the changes made to these files by comparing timestamps and reloads any file that has changed the next time you execute the corresponding function.

If MATLAB does not see the changes you make to one of these files, try clearing the old copy of the function from memory using `clear` *`functionname`*. You can verify that MATLAB has cleared the function using `inmem` to list all functions currently loaded into memory.

Change Notification on Windows

If MATLAB, running on Windows, is unable to see new files or changes you have made to an existing file, the problem may be related to operating system change notification handles.

Type the following for more information:

```
help changeNotification
help changeNotificationAdvanced
```

Program Control

In this section...

Using break, continue, and return

It's easy to confuse the `break`, `continue`, and `return` functions as they are similar in some ways. Make sure you use these functions appropriately.

Function	Where to Use It	Description
`break`	`for` or `while` loops	Exits the loop in which it appears. In nested loops, control passes to the next outer loop.
`continue`	`for` or `while` loops	Skips any remaining statements in the current loop. Control passes to next iteration of the same loop.
`return`	Anywhere	Immediately exits the function in which it appears. Control passes to the caller of the function.

Using switch Versus if

It is possible, but usually not advantageous, to implement `switch-case` statements using `if-elseif` instead. See pros and cons in the table.

switch-case Statements	if-elseif Statements
Easier to read.	Can be difficult to read.
Can compare strings of different lengths.	You need `strcmp` to compare strings of different lengths.
Test for equality only.	Test for equality or inequality.

MATLAB case Evaluates Strings

A useful difference between `switch-case` statements in MATLAB and C is that you can specify string values in MATLAB `case` statements, which you cannot do in C.

```
switch(method)
   case 'linear'
      disp('Method is linear')
   case 'cubic'
      disp('Method is cubic')
end
```

Multiple Conditions in a case Statement

You can test against more than one condition with `switch`. The first case below tests for either a `linear` or `bilinear` method by using a cell array in the case statement.

```
switch(method)
   case {'linear', 'bilinear'}
      disp('Method is linear or bilinear')
   case (<and so on>)
end
```

Implicit Break in switch-case

In C, if you do not end each `case` with a `break` statement, code execution falls through to the following `case`. In MATLAB, `case` statements do not fall

through; only one `case` may execute. Using `break` within a `case` statement is not only unnecessary, it is also invalid and generates a warning.

In this example, if `result` is `52`, only the first `disp` statement executes, even though the second is also a valid match:

```
switch(result)
   case 52
      disp('result is 52')
   case {52, 78}
      disp('result is 52 or 78')
end
```

Variable Scope in a switch

Since MATLAB executes only one `case` of any `switch` statement, variables defined within one `case` are not known in the other `cases` of that `switch` statement. The same holds true for `if-elseif` statements.

In these examples, you get an error when `choice` equals `2`, because `x` is undefined.

```
  -- SWITCH-CASE --
switch choice
   case 1
      x = -pi:0.01:pi;
   case 2
      plot(x, sin(x));
end
```

```
 -- IF-ELSEIF --

if choice == 1
   x = -pi:0.01:pi;
elseif choice == 2
   plot(x, sin(x));
end
```

Catching Errors with try-catch

When you have statements in your code that could possibly generate unwanted results, put those statements into a `try-catch` block that will catch any errors and handle them appropriately.

The example below shows a `try-catch` block within a function that multiplies two matrices. If a statement in the `try` segment of the block fails, control passes to the `catch` segment. In this case, the `catch` statements check the error message that was issued (returned by `lasterr`) and respond appropriately.

```
try
   X = A * B
catch
   errmsg = lasterr;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
end
```

For more information: See “The try-catch Statement” in the MATLAB Programming documentation.

Nested try-catch Blocks

You can also nest `try-catch` blocks, as shown here. You can use this to attempt to recover from an error caught in the first `try` section:

```
try
   statement1                    % Try to execute statement1
catch
   try
      statement2                 % Attempt to recover from error
   catch
      disp 'Operation failed'    % Handle the error
   end
end
```

Forcing an Early Return from a Function

To force an early return from a function, place a `return` statement in the function at the point where you want to exit. For example,

```
if <done>
   return
end
```

Save and Load

In this section...

Saving Data from the Workspace

To save data from your workspace, you can do any of the following:

- Copy from the MATLAB Command Window and paste into a text file.
- Record part of your session in a `diary` file, and then edit the file in a text editor.
- Save to a binary or ASCII file using the `save` function.
- Save spreadsheet, scientific, image, or audio data with appropriate function.
- Save to a file using low-level file I/O functions (`fwrite`, `fprintf`, `...`).

For more information: See Saving the Current Workspace in the MATLAB Desktop Tools and Development Environment documentation, and "Using the diary Function to Export Data" and "Using Low-Level File I/O Functions" in the MATLAB Programming documentation.

Loading Data into the Workspace

Similarly, to load new or saved data into the workspace, you can do any of the following:

- Enter or paste data at the command line.
- Create a script file to initialize large matrices or data structures.

- Read a binary or ASCII file using `load`.
- Load spreadsheet, scientific, image, or audio data with appropriate function.
- Load from a file using low-level file I/O functions (`fread`, `fscanf`, `...`).

For more information: See Loading a Saved Workspace and Importing Data in the MATLAB Development Environment documentation, and "Using Low-Level File I/O Functions" in the MATLAB Programming documentation.

Viewing Variables in a MAT-File

To see what variables are saved in a MAT-file, use `who` or `whos` as shown here (the `.mat` extension is not required). `who` returns a cell array and `whos` returns a structure array.

```
mydataVariables = who('-file', 'mydata.mat');
```

Appending to a MAT-File

To save additional variables to an existing MAT-file, use

```
save matfilename -append
```

Any variables you save that do not yet exist in the MAT-file are added to the file. Any variables you save that already exist in the MAT-file overwrite the old values.

Note Saving with the `-append` switch does not append additional elements to an array that is already saved in a MAT-file. See the example below.

In this example, the second `save` operation does not concatenate new elements to vector `A`, (making `A` equal to `[1 2 3 4 5 6 7 8]`) in the MAT-file. Instead, it replaces the 5 element vector, `A`, with a 3 element vector, also retaining all other variables that were stored on the first `save` operation.

```
A = [1 2 3 4 5];   B = 12.5;   C = rand(4);
save savefile;
A = [6 7 8];
```

```
save savefile A -append;
```

Save and Load on Startup or Quit

You can automatically save your variables at the end of each MATLAB session by creating a `finish.m` file to `save` the contents of your base workspace every time you quit MATLAB. Load these variables back into your workspace at the beginning of each session by creating a `startup.m` file that uses the `load` function to load variables from your MAT-file.

For more information: See the `startup` and `finish` function reference pages.

Saving to an ASCII File

When you save matrix data to an ASCII file using `save -ascii`, MATLAB combines the individual matrices into one collection of numbers. Variable names are not saved. If this is not acceptable for your application, use `fprintf` to store your data instead.

For more information: See “Exporting Delimited ASCII Data Files”.

Files and Filenames

In this section...

Naming M-files

M-file names must start with an alphabetic character, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed M-file name length (returned by the function `namelengthmax`).

```
N = namelengthmax
N =
     63
```

Since variables must obey similar rules, you can use the `isvarname` function to check whether a filename (minus its `.m` file extension) is valid for an M-file.

```
isvarname mfilename
```

Naming Other Files

The names of other files that MATLAB interacts with (e.g., MAT, MEX, and MDL-files) follow the same rules as M-files, but may be of any length.

Depending on your operating system, you may be able to include certain nonalphanumeric characters in your filenames. Check your operating system manual for information on valid filename restrictions.

Passing Filenames as Arguments

In MATLAB commands, you can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The `.mat` file extension is optional for `save` and `load`).

```
load mydata.mat           % Command syntax
load('mydata.mat')        % Function syntax
```

If you assign the output to a variable, you must use the function syntax.

```
savedData = load('mydata.mat')
```

Passing Filenames to ASCII Files

ASCII files are specified as follows. Here, the file extension is required.

```
load mydata.dat -ascii            % Command syntax
load('mydata.dat','-ascii')       % Function syntax
```

Determining Filenames at Run-Time

There are several ways that your function code can work on specific files without you having to hardcode their filenames into the program. You can

- Pass the filename in as an argument

  ```
  function myfun(datafile)
  ```

- Prompt for the filename using the `input` function

  ```
  filename = input('Enter name of file:  ', 's');
  ```

- Browse for the file using the `uigetfile` function

  ```
  [filename, pathname] =
     uigetfile('*.mat', 'Select MAT-file');
  ```

For more information: See the `input` and `uigetfile` function reference pages.

Returning the Size of a File

Two ways to have your program determine the size of a file are shown here.

```
  -- METHOD #1 --
s = dir('myfile.dat');
filesize = s.bytes
```

```
   -- METHOD #2 --
fid = fopen('myfile.dat');
fseek(fid, 0, 'eof');
filesize = ftell(fid)
fclose(fid);
```

The `dir` function also returns the filename (`s.name`), last modification date (`s.date`), and whether or not it's a directory (`s.isdir`).

(The second method requires read access to the file.)

For more information: See the `fopen`, `fseek`, `ftell`, and `fclose` function reference pages.

Input/Output

In this section...

For more information and examples on importing and exporting data, see Technical Note 1602:

http://www.mathworks.com/support/tech-notes/1600/1602.html

File I/O Function Overview

For a good overview of MATLAB file I/O functions, use the online "Functions — Categorical List" reference. In the Help browser **Contents**, select **MATLAB > Functions — Categorical List**, and then click **File I/O**.

Common I/O Functions

The most commonly used, high-level, file I/O functions in MATLAB are `save` and `load`. For help on these, type `doc save` or `doc load`.

Functions for I/O to text files with delimited values are `textread`, `dlmread`, `dlmwrite`. Functions for I/O to text files with comma-separated values are `csvread`, `csvwrite`.

For more information: See Text Files in the MATLAB "Functions — Categorical List" reference documentation.

Readable File Formats

Type `doc fileformats` to see a list of file formats that MATLAB can read, along with the associated MATLAB functions.

Using the Import Wizard

A quick method of importing text or binary data from a file (e.g., Excel files) is to use the MATLAB Import Wizard. Open the Import Wizard with the command, `uiimport filename` or by selecting **File > Import Data** at the Command Window.

Specify or browse for the file containing the data you want to import and you will see a preview of what the file contains. Select the data you want and click **Finish**.

For more information: See "Using the Import Wizard" in the MATLAB Programming documentation.

Loading Mixed Format Data

To load data that is in mixed formats, use `textread` instead of `load`. The `textread` function lets you specify the format of each piece of data.

If the first line of file `mydata.dat` is

```
Sally    12.34 45
```

Read the first line of the file as a free format file using the % format:

```
[names, x, y] = textread('mydata.dat', '%s %f %d', 1)
```

returns

```
names =
    'Sally'
x =
  12.34000000000000
y =
    45
```

Reading Files with Different Formats

Attempting to read data from a file that was generated on a different platform may result in an error because the binary formats of the platforms may differ. Using the `fopen` function, you can specify a machine format when you open the file to avoid these errors.

Reading ASCII Data into a Cell Array

A common technique used to read an ASCII data file into a cell array is

```
[a,b,c,d] = textread('data.txt', '%s %s %s %s');
mydata = cellstr([a b c d]);
```

For more information: See the `textread` and `cellstr` function reference pages.

Interactive Input into Your Program

Your program can accept interactive input from users during execution. Use the `input` function to prompt the user for input, and then read in a response. When executed, `input` causes the program to display your prompt, pause while a response is entered, and then resume when the **Enter** key is pressed.

Starting MATLAB

Getting MATLAB to Start Up Faster

Here are some things that you can do to make MATLAB start up faster.

- Make sure toolbox path caching is enabled.
- Make sure that the system on which MATLAB is running has enough RAM.
- Choose only the windows you need in the MATLAB desktop.
- Close the Help browser before exiting MATLAB. When you start your next session, MATLAB will not open the Help browser, and thus will start faster.
- If disconnected from the network, check the `LM_LICENSE_FILE` variable. See http://www.mathworks.com/support/solutions/data/1-17VEB.html for a more detailed explanation.

For more information: See Toolbox Path Caching in MATLAB in the MATLAB Desktop Tools and Development Environment documentation.

Operating System Compatibility

In this section...

Executing O/S Commands from MATLAB

To execute a command from your operating system prompt without having to exit MATLAB, precede the command with the MATLAB `!` operator.

On Windows, you can add an ampersand (`&`) to the end of the line to make the output appear in a separate window.

For more information: See Running External Programs in the MATLAB Desktop Tools and Development Environment documentation, and the `system` and `dos` function reference pages.

Searching Text with grep

`grep` is a powerful tool for performing text searches in files on UNIX systems. To `grep` from within MATLAB, precede the command with an exclamation point (`!grep`).

For example, to search for the word `warning`, ignoring case, in all M-files of the current directory, you would use

```
!grep -i 'warning' *.m
```

Constructing Paths and Filenames

Use the `fullfile` function to construct path names and filenames rather than entering them as strings into your programs. In this way, you always get the correct path specification, regardless of which operating system you are using at the time.

Finding the MATLAB Root Directory

The `matlabroot` function returns the location of the MATLAB installation on your system. Use `matlabroot` to create a path to MATLAB and toolbox directories that does not depend on a specific platform or MATLAB version.

The following example uses `matlabroot` with `fullfile` to return a platform-independent path to the `general` toolbox directory:

```
fullfile(matlabroot,'toolbox','matlab','general')
```

Temporary Directories and Filenames

If you need to locate the directory on your system that has been designated to hold temporary files, use the `tempdir` function. `tempdir` returns a string that specifies the path to this directory.

To create a new file in this directory, use the `tempname` function. `tempname` returns a string that specifies the path to the temporary file directory, plus a unique filename.

For example, to store some data in a temporary file, you might issue the following command first.

```
fid = fopen(tempname, 'w');
```

Demos

Demos Available with MATLAB

MATLAB comes with a wide array of visual demonstrations to help you see the extent of what you can do with the product. To start running any of the demos, simply type `demo` at the MATLAB command prompt. Demos cover the following major areas:

- MATLAB
- Toolboxes
- Simulink
- Blocksets
- Real-Time Workshop®
- Stateflow®

For more information: See Demos in the Help Browser in the MATLAB Desktop Tools and Development Environment documentation, and the `demo` function reference page.

For More Information

In this section...

Current CSSM

```
http://newsreader.mathworks.com/WebX?14@@/comp.soft-sys.matlab
```

Archived CSSM

```
http://mathforum.org/kb/forum.jspa?forumID=80
```

MATLAB Technical Support

```
http://www.mathworks.com/support/
```

Tech Notes

```
http://www.mathworks.com/support/tech-notes/list_all.html
```

MATLAB Central

```
http://www.mathworks.com/matlabcentral/
```

MATLAB Newsletters (Digest, News & Notes)

```
http://www.mathworks.com/company/newsletters/index.html
```

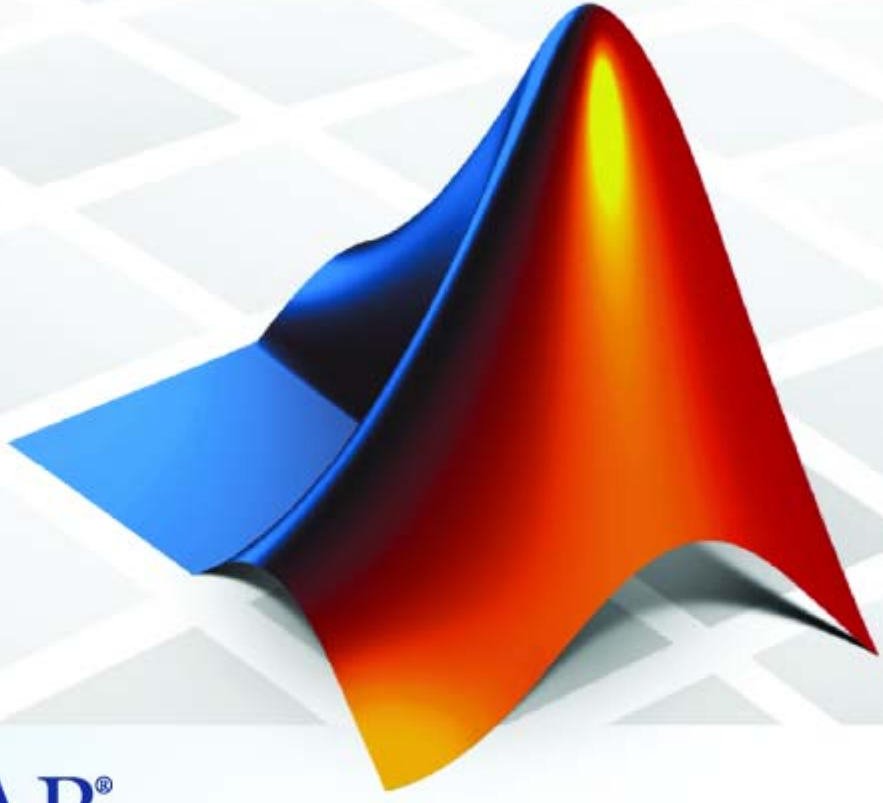
MATLAB Documentation

http://www.mathworks.com/access/helpdesk/help/helpdesk.html

MATLAB Index of Examples

http://www.mathworks.com/access/helpdesk/help/techdoc/demo_example.shtml

MATLAB® 7
3-D Visualization



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

3-D Visualization

Revision History

March 2006	Online only	New for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b) This publication was previously part of the Using MATLAB Graphics User Guide.

Contents

1 Creating 3-D Graphs

2 Defining the View

3 Lighting as a Visualization Tool

4 Transparency

5 Creating 3-D Models with Patches

6 Volume Visualization Techniques

Index

1

Creating 3-D Graphs

A Typical 3-D Graph

This table illustrates typical steps involved in producing 3-D scenes containing either data graphs or models of 3-D objects. Example applications include pseudocolor surfaces illustrating the values of functions over specific regions and objects drawn with polygons and colored with light sources to produce realism. Usually, you follow either step 4 or step 5.

Step	Typical Code
1 Prepare your data.	`Z = peaks(20);`
2 Select window and position plot region within window.	`figure(1) subplot(2,1,2)`
3 Call 3-D graphing function.	`h = surf(Z);`

4 Set colormap and shading algorithm.	`colormap hot`
	`shading interp`
	`set(h,'EdgeColor','k')`

Step	Typical Code

5 Add lighting.	`light('Position',[-2,2,20])`
	`lighting phong`
	`material([0.4,0.6,0.5,30])`
	`set(h,'FaceColor',[0.7 0.7 0],...`
	`  'BackFaceLighting','lit')`

6 Set viewpoint.	`view([30,25])` `set(gca,'CameraViewAngleMode','Manual')`
7 Set axis limits and tick marks.	`axis([5 15 5 15 -8 8])` `set(gca,'ZTickLabel','Negative||Positive')`
8 Set aspect ratio.	`set(gca,'PlotBoxAspectRatio',[2.5 2.5 1])`

9 Annotate the graph with axis labels, legend, and text.	`xlabel('X Axis')`
	`ylabel('Y Axis')`
	`zlabel('Function Value')`
	`title('Peaks')`

10 Print graph.	`set(gcf,'PaperPositionMode','auto')` `print -dps2`

Line Plots of 3-D Data

In this section...

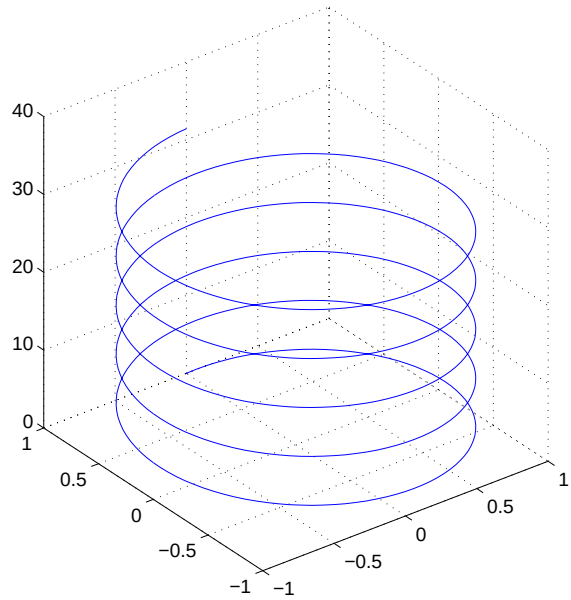
Basic 3-D Plotting: The plot3 function

The 3-D analog of the `plot` function is `plot3`. If `x`, `y`, and `z` are three vectors of the same length,

```
plot3(x,y,z)
```

generates a line in 3-D through the points whose coordinates are the elements of `x`, `y`, and `z` and then produces a 2-D projection of that line on the screen. For example, these statements produce a helix.

```
t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
axis square; grid on
```

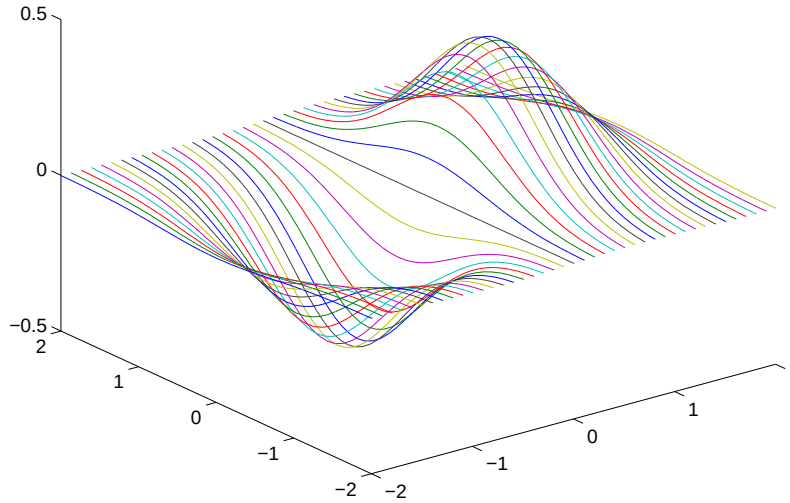


Plotting Matrix Data

If the arguments to `plot3` are matrices of the same size, MATLAB© plots lines obtained from the columns of `X`, `Y`, and `Z`. For example,

```
[X,Y] = meshgrid([-2:0.1:2]);
Z = X.*exp(-X.^2-Y.^2);
plot3(X,Y,Z)
grid on
```

Notice how MATLAB cycles through line colors.


0.5
0
−0.5
2
1
0
−1
−2
−2
−1
0
1
2

Representing a Matrix as a Surface

In this section...

Functions for Plotting Data Grids

MATLAB defines a surface by the *z*-coordinates of points above a rectangular grid in the *x*-*y* plane. The plot is formed by joining adjacent points with straight lines. Surface plots are useful for visualizing matrices that are too large to display in numerical form and for graphing functions of two variables.

MATLAB can create different forms of surface plots. Mesh plots are wire-frame surfaces that color only the lines connecting the defining points. Surface plots display both the connecting lines and the faces of the surface in color. This table lists the various forms.

Function	Used to Create
`mesh, surf`	Surface plot
`meshc, surfc`	Surface plot with contour plot beneath it
`meshz`	Surface plot with curtain plot (reference plane)
`pcolor`	Flat surface plot (value is proportional only to color)
`surfl`	Surface plot illuminated from specified direction
`surface`	Low-level function (on which high-level functions are based) for creating surface graphics objects

Mesh and Surface Plots

The `mesh` and `surf` commands create 3-D surface plots of matrix data. If `Z` is a matrix for which the elements `Z(i,j)` define the height of a surface over an underlying `(i,j)` grid, then

```
mesh(Z)
```

generates a colored, wire-frame view of the surface and displays it in a 3-D view. Similarly,

```
surf(Z)
```

generates a colored, faceted view of the surface and displays it in a 3-D view. Ordinarily, the facets are quadrilaterals, each of which is a constant color, outlined with black mesh lines, but the `shading` command allows you to eliminate the mesh lines (`shading flat`) or to select interpolated shading across the facet (`shading interp`).

Surface object properties provide additional control over the visual appearance of the surface. You can specify edge line styles, vertex markers, face coloring, lighting characteristics, and so on.

Visualizing Functions of Two Variables

The first step in displaying a function of two variables, $z = f(x,y)$, is to generate `X` and `Y` matrices consisting of repeated rows and columns, respectively, over the domain of the function. Then use these matrices to evaluate and graph the function.

The `meshgrid` function transforms the domain specified by two vectors, `x` and `y`, into matrices `X` and `Y`. You then use these matrices to evaluate functions of two variables. The rows of `X` are copies of the vector `x` and the columns of `Y` are copies of the vector `y`.

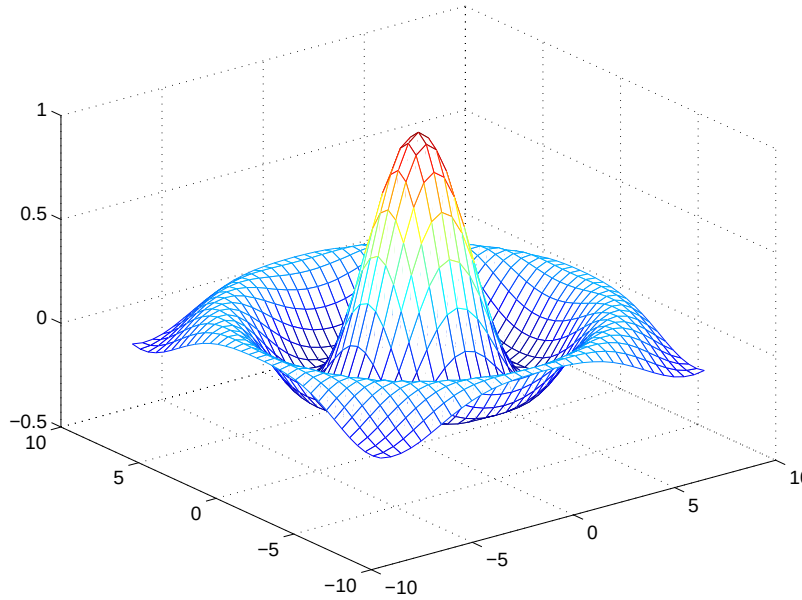
To illustrate the use of `meshgrid`, consider the `sin(r)/r` or `sinc` function. To evaluate this function between -8 and 8 in both x and y, you need pass only one vector argument to `meshgrid`, which is then used in both directions.

```
[X,Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
```

The matrix `R` contains the distance from the center of the matrix, which is the origin. Adding `eps` prevents the divide by zero (in the next step) that produces `Inf` values in the data.

Forming the `sinc` function and plotting `Z` with `mesh` results in the 3-D surface.

```
Z = sin(R)./R;
mesh(X,Y,Z)
```

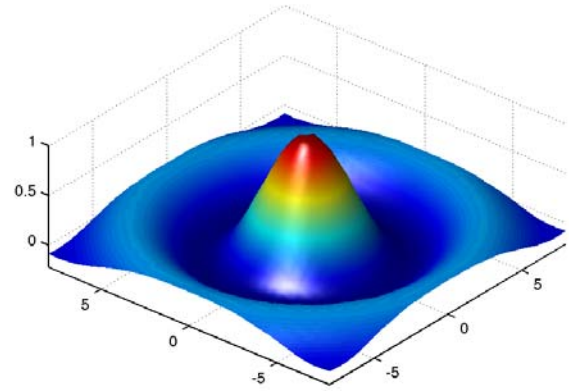


Emphasizing Surface Shape

MATLAB provides a number of techniques that can enhance the information content of your graphs. For example, this graph of the `sinc` function uses the same data as the previous graph, but employs lighting and view adjustment to emphasize the shape of the graphed function (`daspect`, `axis`, `view`, `camlight`).

```
surf(X,Y,Z,'FaceColor','interp',...
 'EdgeColor','none',...
 'FaceLighting','phong')
```

```
daspect([5 5 1])
axis tight
view(-50,30)
camlight left
```



See the `surf` function for more information on surface plots.

Surface Plots of Nonuniformly Sampled Data

You can use `meshgrid` to create a grid of uniformly sampled data points at which to evaluate and graph the `sinc` function. MATLAB then constructs the surface plot by connecting neighboring matrix elements to form a mesh of quadrilaterals.

To produce a surface plot from nonuniformly sampled data, first use `griddata` to interpolate the values at uniformly spaced points, and then use `mesh` and `surf` in the usual way.

Example — Displaying Nonuniform Data on a Surface

This example evaluates the `sinc` function at random points within a specific range and then generates uniformly sampled data for display as a surface plot. The process involves these tasks:

- Use `linspace` to generate evenly spaced values over the range of your unevenly sampled data.
- Use `meshgrid` to generate the plotting grid with the output of `linspace`.
- Use `griddata` to interpolate the irregularly sampled data to the regularly spaced grid returned by `meshgrid`.
- Use a plotting function to display the data.

1 First, generate unevenly sampled data within the range [-8, 8] and use it to evaluate the function.

```
x = rand(100,1)*16 - 8;
y = rand(100,1)*16 - 8;
r = sqrt(x.^2 + y.^2) + eps;
z = sin(r)./r;
```

2 The `linspace` function provides a convenient way to create uniformly spaced data with the desired number of elements. The following statements produce vectors over the range of the random data with the same resolution as that generated by the -8:.5:8 statement in the previous `sinc` example.

```
xlin = linspace(min(x),max(x),33);
ylin = linspace(min(y),max(y),33);
```

3 Now use these points to generate a uniformly spaced grid.

```
[X,Y] = meshgrid(xlin,ylin);
```

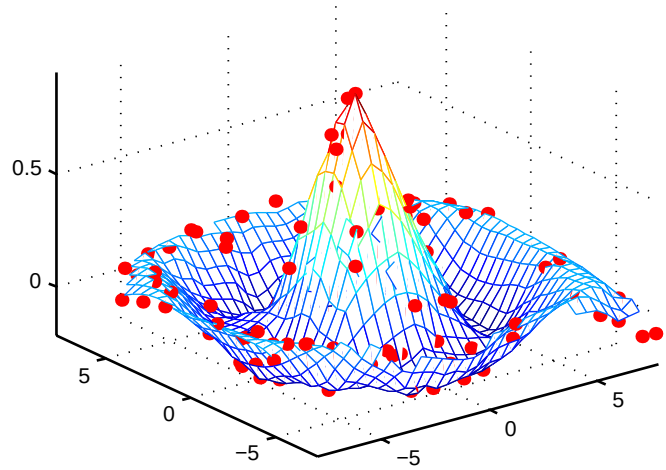
4 The key to this process is to use `griddata` to interpolate the values of the function at the uniformly spaced points, based on the values of the function at the original data points (which are random in this example). This statement uses a triangle-based cubic interpolation to generate the new data.

```
Z = griddata(x,y,z,X,Y,'cubic');
```

5 Plotting the interpolated and the nonuniform data produces

```
mesh(X,Y,Z) %interpolated
axis tight; hold on
```

```
plot3(x,y,z,'.','MarkerSize',15) %nonuniform
```



Parametric Surfaces

The functions that draw surfaces can take two additional vector or matrix arguments to describe surfaces with specific *x* and *y* data. If `Z` is an *m*-by-*n* matrix, `x` is an *n*-vector, and `y` is an *m*-vector, then

```
mesh(x,y,Z,C)
```

describes a mesh surface with vertices having color `C(i,j)` and located at the points

```
(x(j), y(i), Z(i,j))
```

where `x` corresponds to the columns of `Z` and `y` to its rows.

More generally, if `X`, `Y`, `Z`, and `C` are matrices of the same dimensions, then

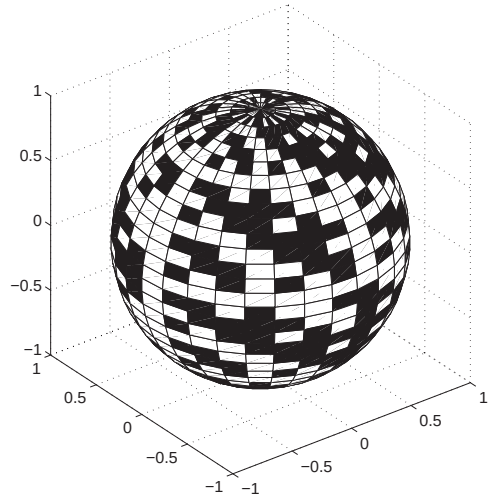
```
mesh(X,Y,Z,C)
```

describes a mesh surface with vertices having color `C(i,j)` and located at the points

```
(X(i,j), Y(i,j), Z(i,j))
```

This example uses spherical coordinates to draw a sphere and color it with the pattern of pluses and minuses in a Hadamard matrix, an orthogonal matrix used in signal processing coding theory. The vectors `theta` and `phi` are in the range $-\pi \le$ `theta` $\le \pi$ and $-\pi/2 \le$ `phi` $\le \pi/2$. Because `theta` is a row vector and `phi` is a column vector, the multiplications that produce the matrices `X`, `Y`, and `Z` are vector outer products.

```
k = 5;
n = 2^k-1;
theta = pi*(-n:2:n)/n;
phi = (pi/2)*(-n:2:n)'/n;
X = cos(phi)*cos(theta);
Y = cos(phi)*sin(theta);
Z = sin(phi)*ones(size(theta));
colormap([0 0 0;1 1 1])
C = hadamard(2^k);
surf(X,Y,Z,C)
axis square
```

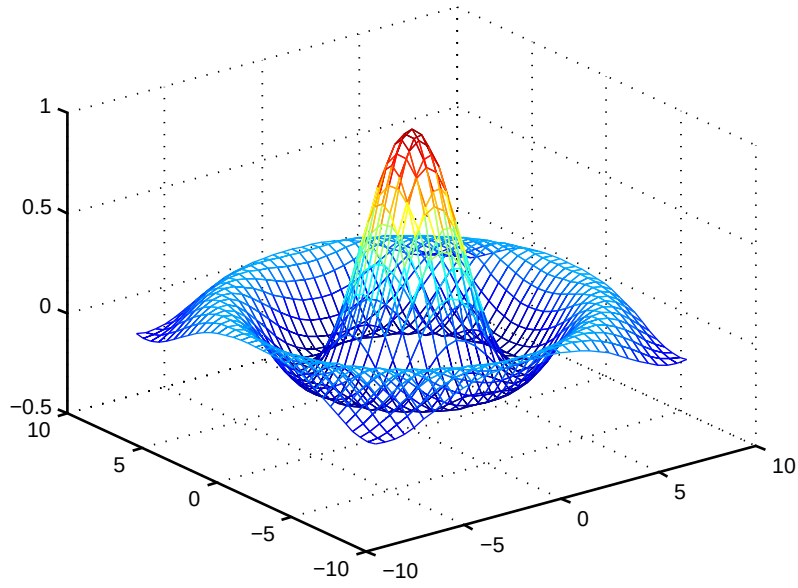


Hidden Line Removal

By default, MATLAB removes lines that are hidden from view in mesh plots, even though the faces of the plot are not colored. You can disable hidden line removal and allow the faces of a mesh plot to be transparent with the command

```
hidden off
```

This is the surface plot with `hidden` set to `off`.


1
0.5
0
−0.5
10
5
0
−5
−10
−10
−5
0
5
10

Coloring Mesh and Surface Plots

In this section...

Coloring Techniques

You can enhance the information content of surface plots by controlling the way MATLAB applies color to these plots. MATLAB can map particular data values to colors specified explicitly or can map the entire range of data to a predefined range of colors called a *colormap*.

You can apply three different coloring techniques:

- Indexed Color -- MATLAB colors the surface plot by assigning each data point an index into the figure's colormap. The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated).
- Truecolor — MATLAB colors the surface plot using the explicitly specified colors (i.e., the RGB triplets). The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated). To be rendered accurately, truecolor requires computers with 24-bit displays; however, MATLAB simulates truecolor on indexed systems. See the `shading` command for information on the types of shading.
- Texture Mapping -- Texture mapping displays a 2-D image mapped onto a 3-D surface.

Types of Color Data

The type of color data you specify (i.e., single values or RGB triplets) determines how MATLAB interprets it. When you create a surface plot, you can

- Provide no explicit color data, in which case MATLAB generates colormap indices from the *z*-data.
- Specify an array of color data that is equal in size to the *z*-data and is used for indexed colors.
- Specify an *m*-by-*n*-by-3 array of color data that defines an RGB triplet for each element in the *m*-by-*n* *z*-data array and is used for truecolor.

Colormaps

Each MATLAB figure window has a colormap associated with it. A colormap is simply a three-column matrix whose length is equal to the number of colors it defines. Each row of the matrix defines a particular color by specifying three values in the range 0 to 1. These values define the RGB components (i.e., the intensities of the red, green, and blue video components).

The `colormap` function, with no arguments, returns the current figure's colormap.

For example, the MATLAB default colormap contains 64 colors and the 57th color is red.

```
cm = colormap;
cm(57,:)
ans =
      1  0  0
```

RGB Color Components

This table lists some representative RGB color definitions.

Red	Green	Blue	Color
0	0	0	Black
1	1	1	White
1	0	0	Red
0	1	0	Green
0	0	1	Blue
1	1	0	Yellow
1	0	1	Magenta
0	1	1	Cyan
0.5	0.5	0.5	Gray
0.5	0	0	Dark red
1	0.62	0.40	Copper
0.49	1	0.83	Aquamarine

You can create colormaps with MATLAB array operations or you can use any of several functions that generate useful maps, including `hsv`, `hot`, `cool`, `summer`, and `gray`. Each function has an optional parameter that specifies the number of rows in the resulting map.

For example,

```
hot(m)
```

creates an *m*-by-3 matrix whose rows specify the RGB intensities of a map that varies from black, through shades of red, orange, and yellow, to white.

If you do not specify the colormap length, MATLAB creates a colormap the same length as the current colormap. The default colormap is `jet(64)`.

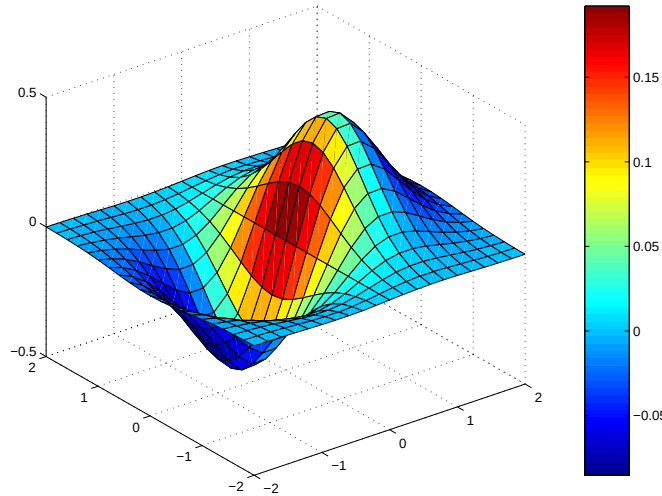
If you use long colormaps (> 64 colors) in each of several figure windows, it might become necessary for the operating system to swap in different color lookup tables as the active focus is moved among the windows.

Displaying Colormaps

The `colorbar` function displays the current colormap, either vertically or horizontally, in the figure window along with your graph. For example, the statements

```
[x,y] = meshgrid([-2:.2:2]);
Z = x.*exp(-x.^2-y.^2);
surf(x,y,Z,gradient(Z))
colorbar
```

produce a surface plot and a vertical strip of color corresponding to the colormap. Note how the colorbar indicates the mapping of data value to color with the axis labels.



Indexed Color Surfaces — Direct and Scaled Color Mapping

MATLAB can use two different methods to map indexed color data to the colormap — direct and scaled.

Direct Mapping

Direct mapping uses the color data directly as indices into the colormap. For example, a value of 1 points to the first color in the colormap, a value of 2 points to the second color, and so on. If the color data is noninteger, MATLAB rounds it toward zero. Values greater than the number of colors in the colormap are set equal to the last color in the colormap (i.e., the number `length(colormap)`). Values less than 1 are set to 1.

Scaled Mapping

Scaled mapping uses a two-element vector `[cmin cmax]` (specified with the `caxis` command) to control the mapping of color data to the figure colormap. `cmin` specifies the data value to map to the first color in the colormap and `cmax` specifies the data value to map to the last color in the colormap. Data values in between are linearly transformed from the second to the next-to-last color, using the expression

```
colormap_index = fix((color_data-cmin)/(cmax-cmin)*cm_length)+1
```

`cm_length` is the length of the colormap.

By default, MATLAB sets `cmin` and `cmax` to span the range of the color data of all graphics objects within the axes. However, you can set these limits to any range of values. This enables you to display multiple axes within a single figure window and use different portions of the figure's colormap for each one. See Calculating Color Limits in Axes Properties of the Using MATLAB Graphics documentation for an example that uses color limits.

By default, MATLAB uses scaled mapping. To use direct mapping, you must turn off scaling when you create the plot. For example,

```
surf(Z,C,'CDataMapping','direct')
```

See `surface` for more information on specifying color data.

Specifying Indexed Colors

When creating a surface plot with a single matrix argument, `surf(Z)` for example, the argument `Z` specifies both the height and the color of the surface. MATLAB transforms `Z` to obtain indices into the current colormap.

With two matrix arguments, the statement

```
surf(Z,C)
```

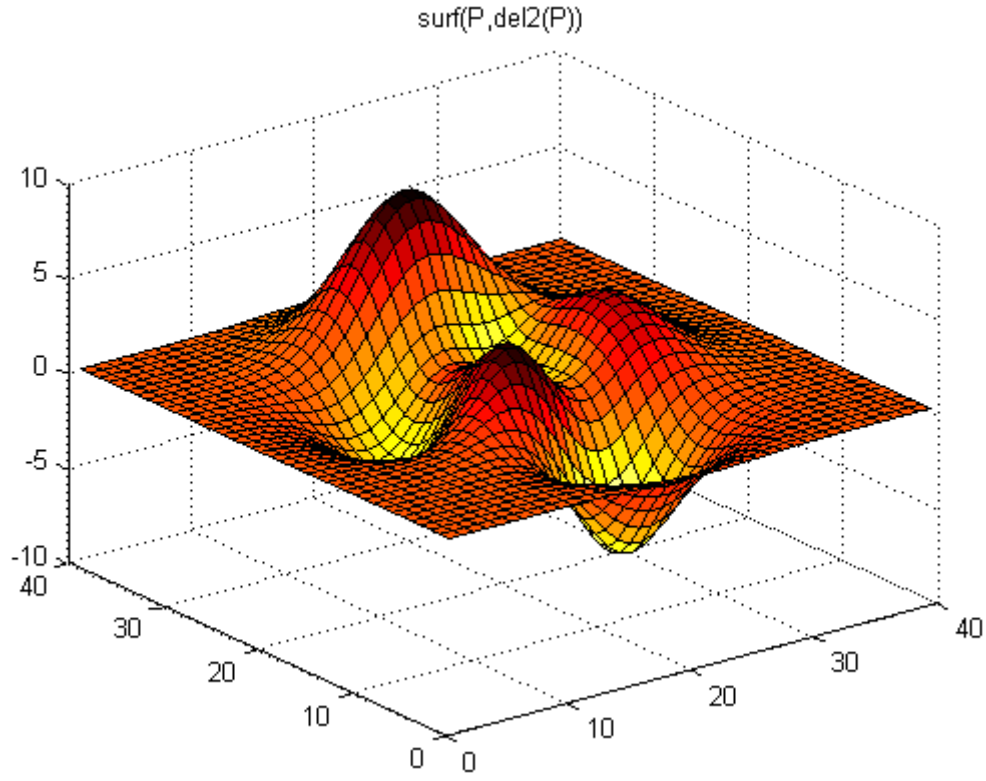
independently specifies the color using the second argument.

Example — Mapping Surface Curvature to Color

The Laplacian of a surface plot is related to its curvature; it is positive for functions shaped like `i^2 + j^2` and negative for functions shaped like `-(i^2 + j^2)`. The function `del2` computes the discrete Laplacian of any matrix. For example, use `del2` to determine the color for the data returned by `peaks`.

```
P = peaks(40);
C = del2(P);
surf(P,C)
colormap hot
```

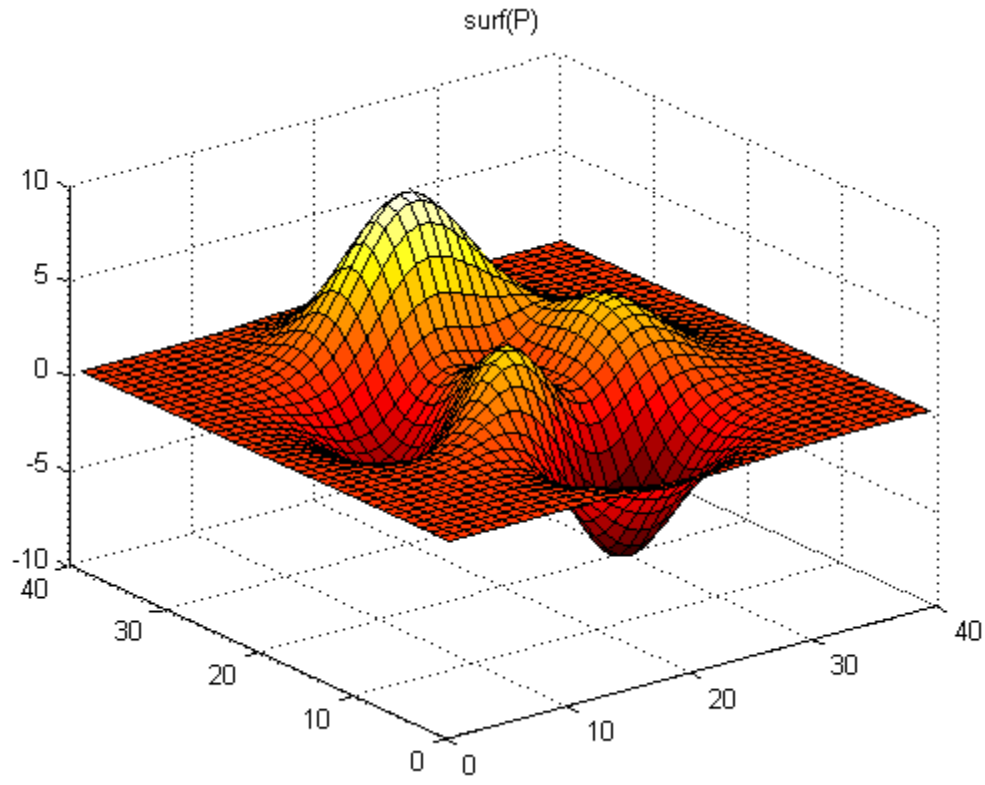
Creating a color array by applying the Laplacian to the data is useful because it causes regions with similar curvature to be drawn in the same color.



Compare this surface coloring with that produced by the statements

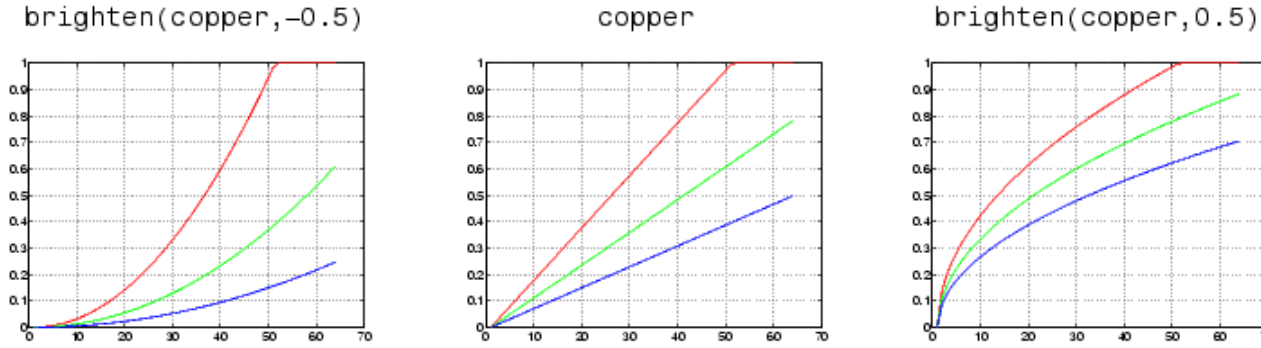
```
surf(P)
colormap hot
```

which use the same colormap, but map regions with similar z value (height above the x-y plane) to the same color.



Altering Colormaps

Because colormaps are matrices, you can manipulate them like other arrays. The `brighten` function takes advantage of this fact to increase or decrease the intensity of the colors. Plotting the values of the R, G, and B components of a colormap using `rgbplot` illustrates the effects of `brighten`.



NTSC Color Encoding

The brightness component of television signals uses the NTSC color encoding scheme.

```
b = .30*red + .59*green + .11*blue
  = sum(diag([.30 .59 .11])*map')';
```

Using the nonlinear greyscale map,

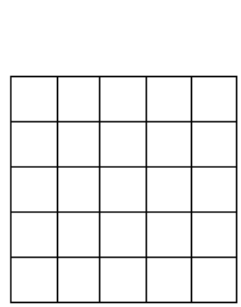
```
colormap([b b b])
```

effectively converts a color image to its NTSC black-and-white equivalent.

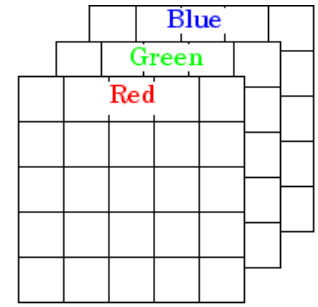
Truecolor Surfaces

Computer systems with 24-bit displays are capable of displaying over 16 million (2^{24}) colors, as opposed to the 256 colors available on 8-bit displays. You can take advantage of this capability by defining color data directly as RGB values and eliminating the step of mapping numerical values to locations in a colormap.

Specify truecolor using an *m*-by-*n*-by-3 array, where the size of `Z` is *m*-by-*n*.



m-by-*n* matrix defining surface plot

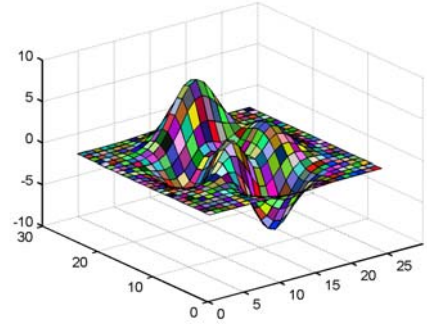


Corresponding *m*-by-*n*-by-3 matrix specifying truecolor for the surface plot

For example, the statements

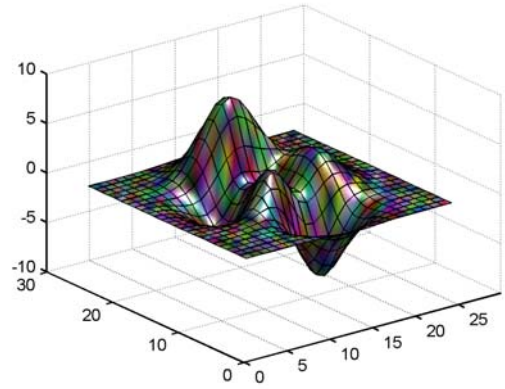
```
Z = peaks(25);
C(:,:,1) = rand(25);
C(:,:,2) = rand(25);
C(:,:,3) = rand(25);
surf(Z,C)
```

create a plot of the `peaks` matrix with random coloring.



You can set surface properties as with indexed color.

```
surf(Z,C,'FaceColor','interp','FaceLighting','phong')
camlight right
```



Rendering Methods for Truecolor

MATLAB always uses either OpenGL or the Z-buffer rendering method when displaying truecolor. If the figure `RendererMode` property is set to `auto`, MATLAB automatically switches the value of the `Renderer` property to `zbuffer` whenever you specify truecolor data.

If you explicitly set `Renderer` to `painters` (this sets `RendererMode` to `manual`) and attempt to define an image, patch, or surface object using truecolor, MATLAB returns a warning and does not render the object.

See the `image`, `patch`, and `surface` functions for information on defining truecolor for these objects.

Texture Mapping

Texture mapping is a technique for mapping a 2-D image onto a 3-D surface by transforming color data so that it conforms to the surface plot. It allows you to apply a "texture," such as bumps or wood grain, to a surface without performing the geometric modeling necessary to create a surface with these features. The color data can also be any image, such as a scanned photograph.

Texture mapping allows the dimensions of the color data array to be different from the data defining the surface plot. You can apply an image of arbitrary

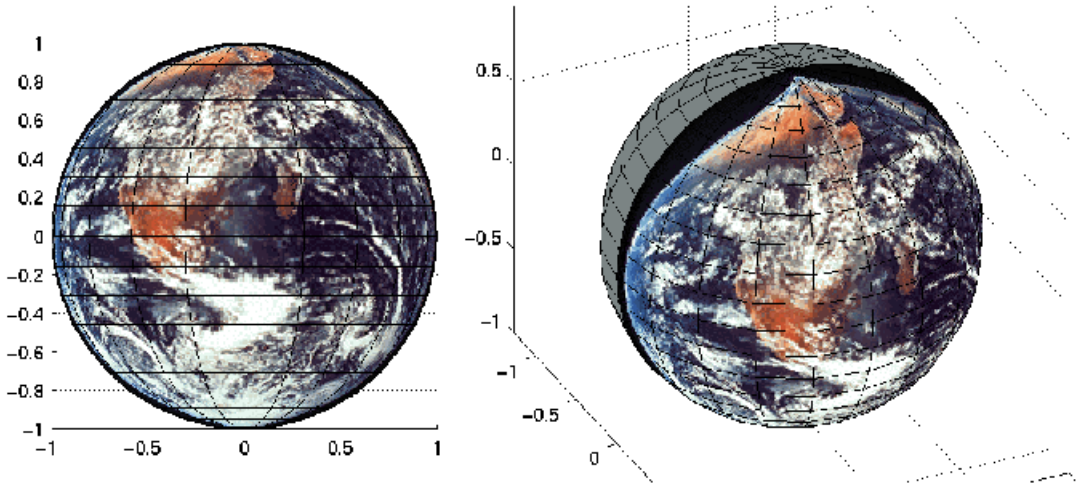
size to any surface. MATLAB interpolates texture color data so that it is mapped to the entire surface.

Example — Texture Mapping a Surface

This example creates a spherical surface using the `sphere` function and texture maps it with an image of the earth taken from space. Because the earth image is a view of earth from one side, this example maps the image to only one side of the sphere, padding the image data with 1s In this case, the image data is a 257-by-250 matrix, so it is padded equally on each side with two 257-by-125 matrices of 1s by concatenating the three matrices.

To use texture mapping, set the `FaceColor` to `texturemap` and assign the image to the surface's `CData`.

```
load earth % Load image data, X, and colormap, map
sphere; h = findobj('Type','surface');
hemisphere = [ones(257,125),...
              X,...
              ones(257,125)];
set(h,'CData',flipud(hemisphere),'FaceColor','texturemap')
colormap(map)
axis equal
view([90 0])
set(gca,'CameraViewAngleMode','manual')
view([65 30])
```


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

2

Defining the View

Viewing Overview

In this section...

Viewing 3-D Graphs and Scenes

The *view* is the particular orientation you select to display your graph or graphical scene. The term *viewing* refers to the process of displaying a graphical scene from various directions, zooming in or out, changing the perspective and aspect ratio, flying by, and so on.

This section describes how to define the various viewing parameters to obtain the view you want. Generally, viewing is applied to 3-D graphs or models, although you might want to adjust the aspect ratio of 2-D views to achieve specific proportions or make a graph fit in a particular shape.

MATLAB viewing is composed of two basic areas:

- Positioning the viewpoint to orient the scene
- Setting the aspect ratio and relative axis scaling to control the shape of the objects being displayed

Positioning the Viewpoint

- Setting the Viewpoint -- Discusses how to specify the point from which you view a graph in terms of azimuth and elevation. This is conceptually simple, but does have limitations.
- Defining Scenes with Camera Graphics, View Control with the Camera Toolbar, and Camera Graphics Functions — How to compose complex scenes using the MATLAB camera viewing model.
- Dollying the Camera and Moving the Camera Through a Scene -- Programming techniques for moving the view around and through scenes.

- Low-Level Camera Properties — The graphics properties that control the camera and illustrates the effects they cause.

Setting the Aspect Ratio

- View Projection Types -- Describes orthographic and perspective projection types and illustrates their use.
- Understanding Axes Aspect Ratio and Axes Aspect Ratio Properties — How MATLAB sets the aspect ratio of the axes and how you can select the most appropriate setting for your graphs.

Default Views

MATLAB automatically sets the view when you create a graph. The actual view that MATLAB selects depends on whether you are creating a 2- or 3-D graph. See “Default Viewpoint Selection” on page 2-31 and “Default Aspect Ratio Selection” on page 2-48 for a description of how MATLAB defines the standard view.

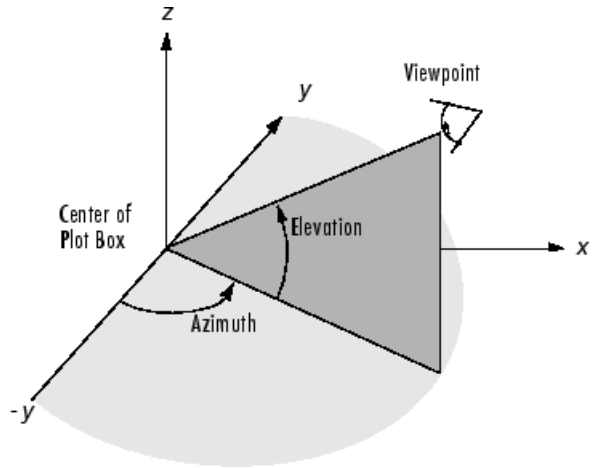
Setting the Viewpoint with Azimuth and Elevation

Azimuth and Elevation

MATLAB enables you to control the orientation of the graphics displayed in an axes. You can specify the viewpoint, view target, orientation, and extent of the view displayed in a figure window. These viewing characteristics are controlled by a set of graphics properties. You can specify values for these properties directly or you can use the `view` command and rely on MATLAB automatic property selection to define a reasonable view.

The `view` command specifies the viewpoint by defining azimuth and elevation with respect to the axis origin. Azimuth is a polar angle in the *x*-*y* plane, with positive angles indicating counterclockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the *x*-*y* plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions.



Default 2-D and 3-D Views

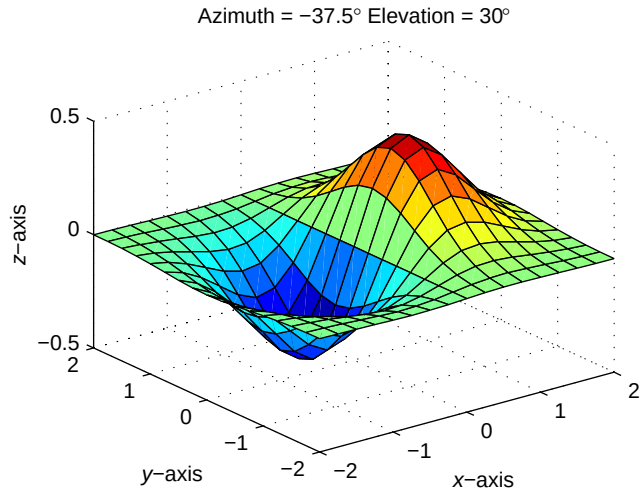
MATLAB automatically selects a viewpoint that is determined by whether the plot is 2-D or 3-D:

- For 2-D plots, the default is azimuth = 0° and elevation = 90°.
- For 3-D plots, the default is azimuth = -37.5° and elevation = 30°.

Examples of Views Specified with Azimuth and Elevation

For example, these statements create a 3-D surface plot and display it in the default 3-D view.

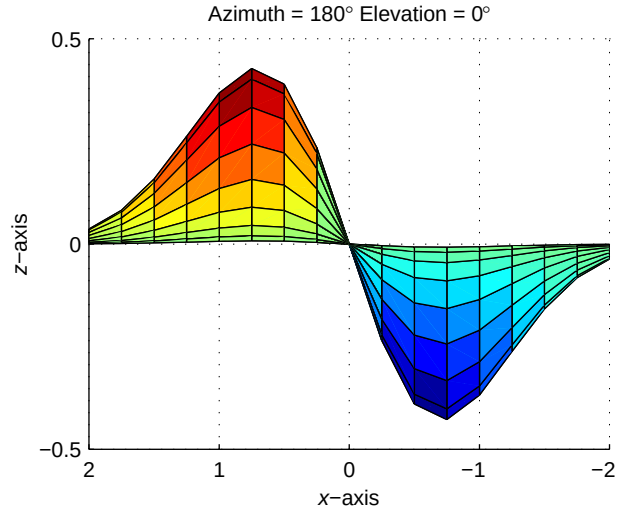
```
[X,Y] = meshgrid([-2:.25:2]);
Z = X.*exp(-X.^2 -Y.^2);
surf(X,Y,Z)
```



The statement

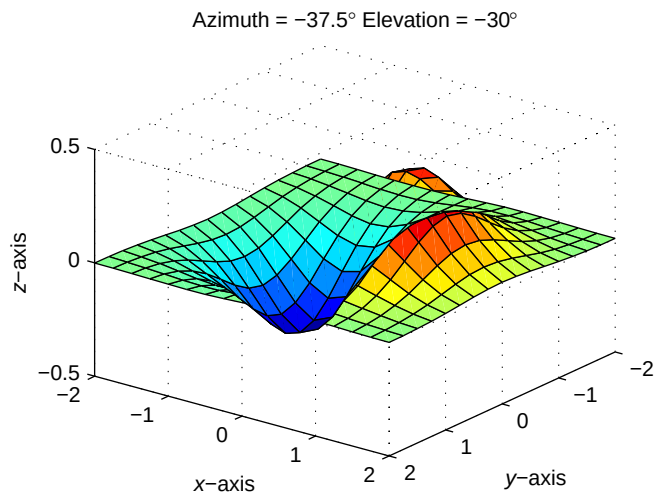
```
view([180 0])
```

sets the viewpoint so you are looking in the negative *y*-direction with your eye at the $z = 0$ elevation.



You can move the viewpoint to a location below the axis origin using a negative elevation.

```
view([-37.5 -30])
```



Limitations of Azimuth and Elevation

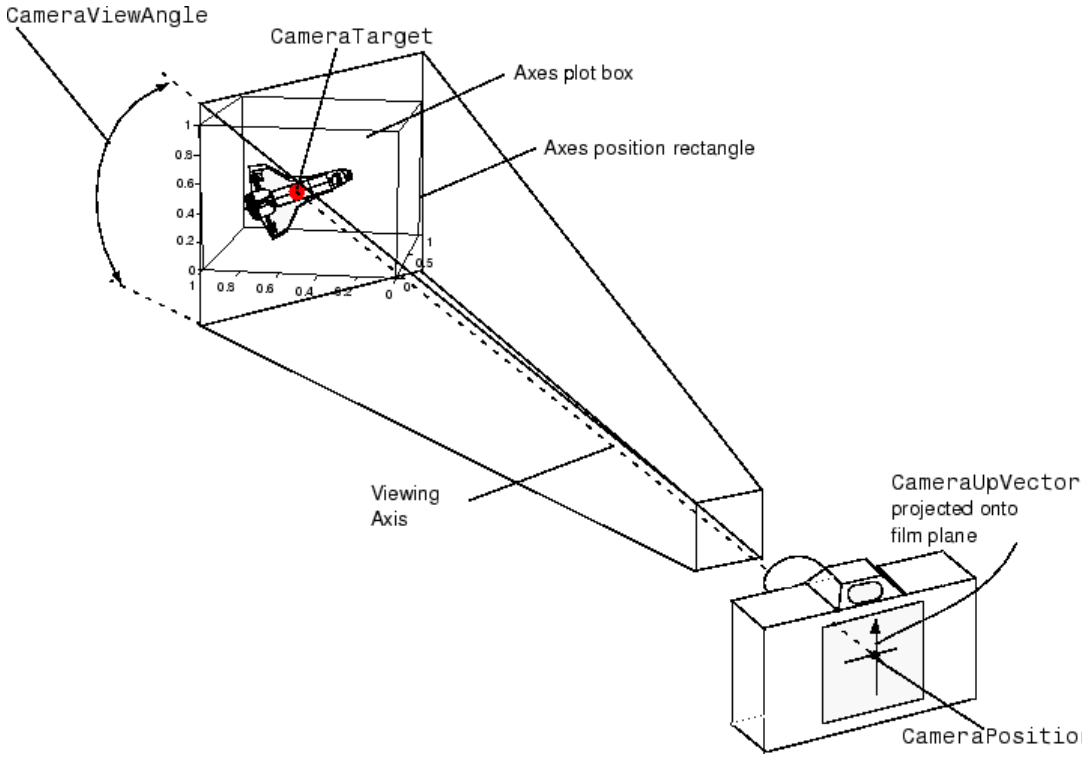
Specifying the viewpoint in terms of azimuth and elevation is conceptually simple, but it has limitations. It does not allow you to specify the actual position of the viewpoint, just its direction, and the *z*-axis is always pointing up. It does not allow you to zoom in and out on the scene or perform arbitrary rotations and translations.

MATLAB camera graphics provides greater control than the simple adjustments allowed with azimuth and elevation. The following sections discuss how to use camera properties to control the view.

Defining Scenes with Camera Graphics

When you look at the graphics objects displayed in an axes, you are viewing a scene from a particular location in space that has a particular orientation with regard to the scene. MATLAB provides functionality, analogous to that of a camera with a zoom lens, that enables you to control the view of the scene created by MATLAB.

This picture illustrates how the camera is defined in terms of properties of the axes.



View Control with the Camera Toolbar

In this section...

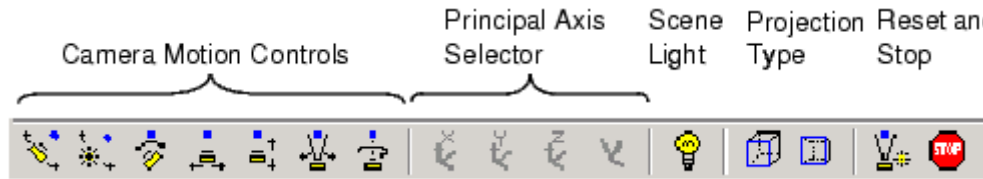
Camera Toolbar

The Camera toolbar enables you to perform a number of viewing operations interactively. To use the Camera toolbar,

- Display the toolbar by selecting **Camera Toolbar** from the figure window’s **View** menu.
- Select the type of camera motion control you want to use.
- Position the cursor over the figure window and click, hold down the right mouse button, then move the cursor in the desired direction.

MATLAB updates the display immediately as you move the mouse.

The toolbar contains the following parts:



- Camera Motion Controls — These tools select which camera motion function to enable. You can also access the camera motion controls from the **Tools** menu.
- Principal Axis Selector — Some camera controls operate with respect to a particular axis. These selectors enable you to select the principal axis or to select nonaxis constrained motion. The selectors are grayed out when not applicable to the currently selected function. You can also access the principal axis selector from the **Tools** menu.
- Scene Light — The scene light button toggles a light source on or off in the scene (one light per axes).
- Projection Type — You can select orthographic or perspective projection types.
- Reset and Stop — Reset returns the scene to the standard 3-D view. Stop causes the camera to stop moving (this can be useful if you apply too much cursor movement). You can also access an expanded set of reset functions from the **Tools** menu.

Principal Axes

The principal axis of a scene defines the direction that is oriented upward on the screen. For example, a MATLAB surface plot aligns the up direction along the positive *z*-axis.

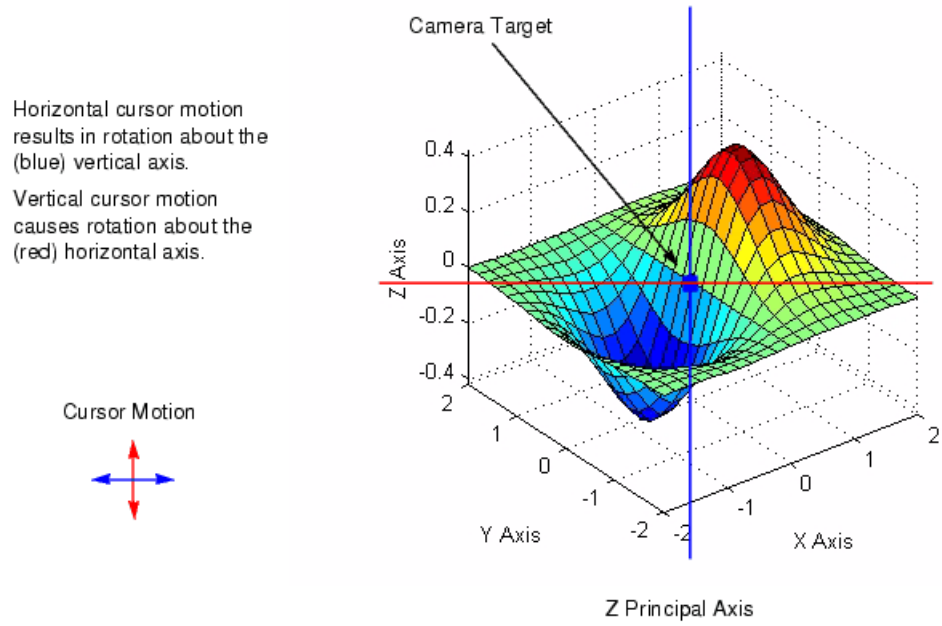
Principal axes constrain camera-tool motion along axes that are (on the screen) parallel and perpendicular to the principal axis that you select. Specifying a principal axis is useful if your data is defined with respect to a specific axis. Z is the default principal axis, because this matches the MATLAB default 3-D view.

Two of the camera tools (Orbit and Pan/Tilt) allow you to select a principal axis as well as axis-free motion. On the screen, the axes of rotation are determined by a vertical and a horizontal line, both of which pass through the point defined by the `CameraTarget` property and are parallel and perpendicular to the principal axis.

For example, when the principal axis is z, movement occurs about

- A vertical line that passes through the camera target and is parallel to the *z*-axis
- A horizontal line that passes through the camera target and is perpendicular to the *z*-axis

This means the scene (or camera, as the case may be) moves in an arc whose center is at the camera target. The following picture illustrates the rotation axes for a *z* principal axis.

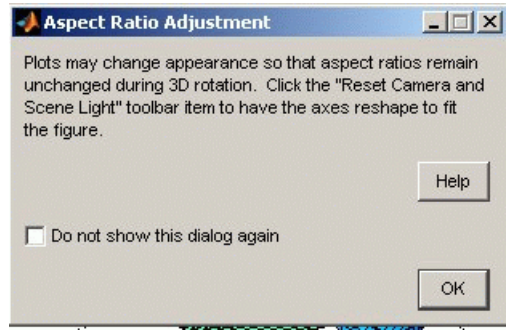


The axes of rotation always pass through the camera target.

Optimizing for 3-D Camera Motion

When you create a plot, MATLAB displays it with an aspect ratio that fits the figure window. This behavior might not create an optimum situation for the manipulation of 3-D graphics, as it can lead to distortion as you move the camera around the scene. To avoid possible distortion, it is best to switch to a 3-D visualization mode (enabled from the command line with the command

`axis vis3d`). When using the Camera toolbar, MATLAB automatically switches to the 3-D visualization mode, but warns you first with the following dialog box.



This dialog box appears only once per MATLAB session.

For more information about the underlying effects of related camera properties, see “Understanding Axes Aspect Ratio” on page 2-42. The next section, “Camera Motion Controls” on page 2-13, discusses how to use each tool.

Camera Motion Controls

This section discusses the individual camera motion functions selectable from the toolbar.

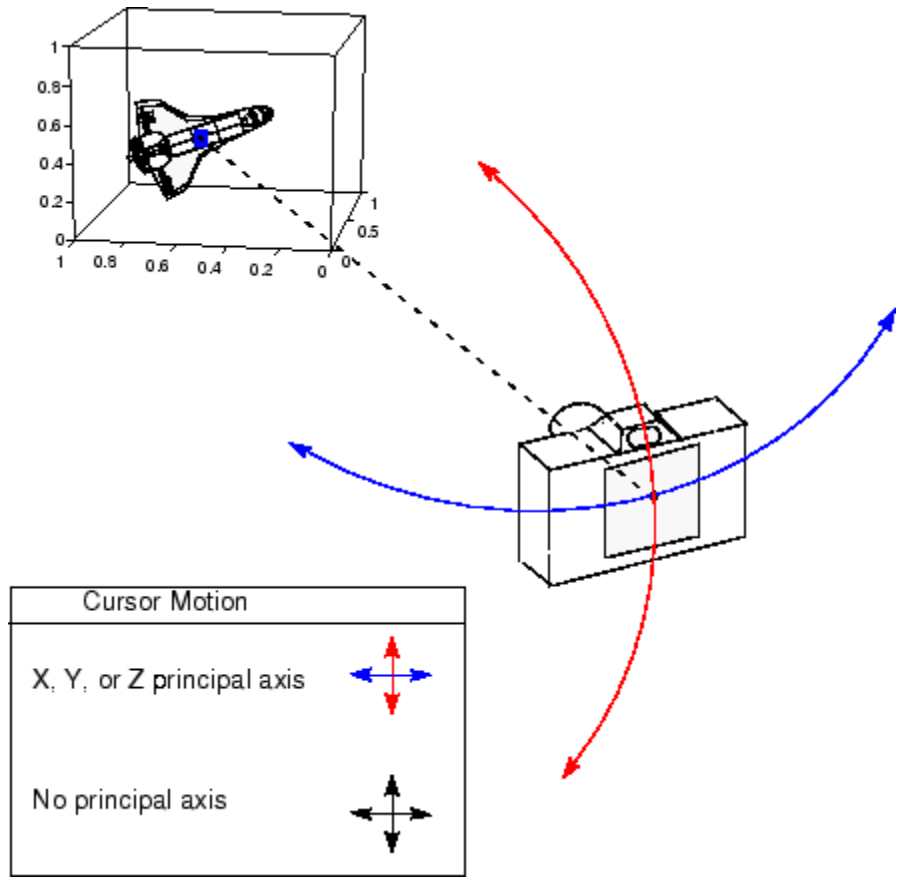
Note When interpreting the following diagrams, keep in mind that the camera always points towards the camera target. See “Defining Scenes with Camera Graphics” on page 2-9 for an illustration of the graphics properties involved in camera motion.

Orbit Camera

Orbit Camera rotates the camera about the *z*-axis (by default). You can select *x*-, *y*-, *z*-, or free-axis rotation using the Principal Axis Selectors. When using no principal axis, you can rotate about an arbitrary axis.

Graphics Properties

Orbit Camera changes the `CameraPosition` property while keeping the `CameraTarget` fixed.



Orbit Scene Light

The scene light is a light source that is placed with respect to the camera position. By default, the scene light is positioned to the right of the camera (i.e., `camlight right`). Orbit Scene Light changes the light's offset from the camera position. There is only one scene light; however, you can add other lights using the `light` command.

Toggle the scene light on and off by clicking the yellow light bulb icon.

Graphics Properties

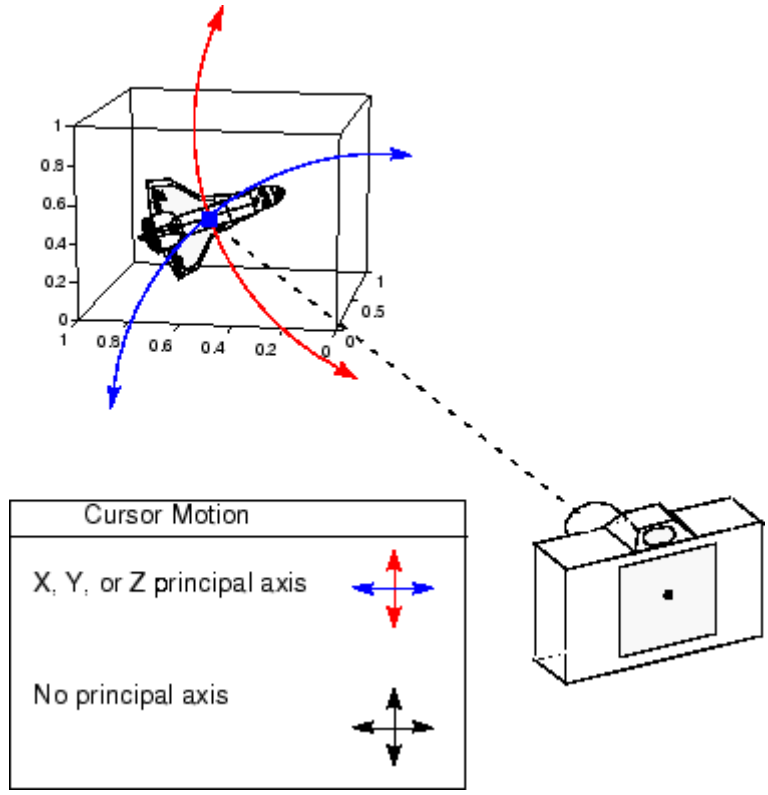
Orbit Scene Light moves the scene light by changing the light's `Position` property.

Pan/Tilt Camera

Pan/Tilt Camera moves the point in the scene that the camera points to while keeping the camera fixed. The movement occurs in an arc about the *z*-axis by default. You can select *x*-, *y*-, *z*-, or free-axis rotation using the Principal Axes Selectors.

Graphics Properties

Pan/Tilt Camera moves the point in the scene that the camera is pointing to by changing the `CameraTarget` property.

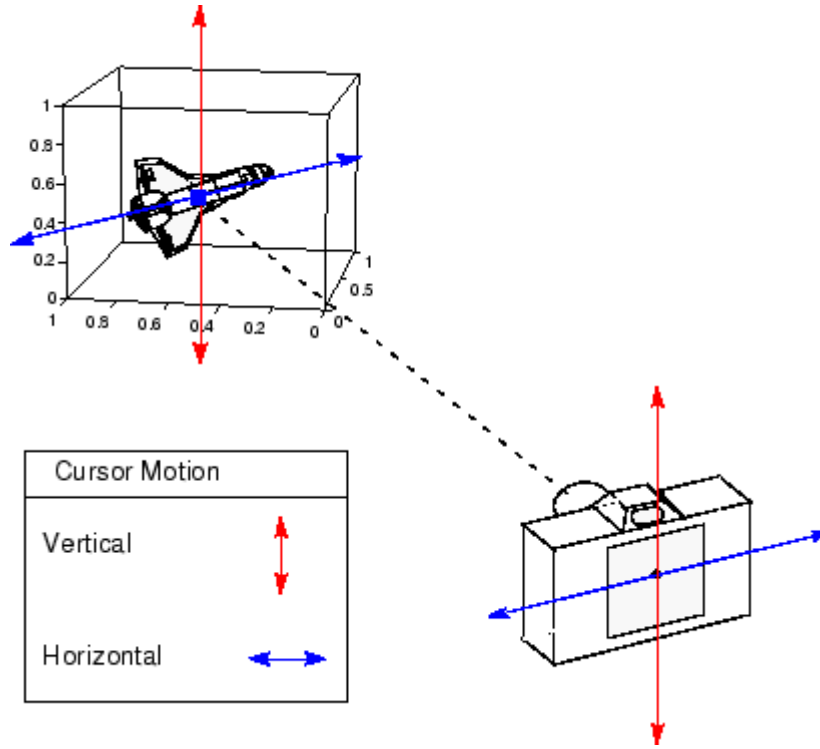


Move Camera Horizontally/Vertically

Moving the cursor horizontally or vertically (or any combination of the two) moves the scene in the same direction.

Graphics Properties

The horizontal and vertical movement is achieved by moving the `CameraPosition` and the `CameraTarget` in unison along parallel lines.

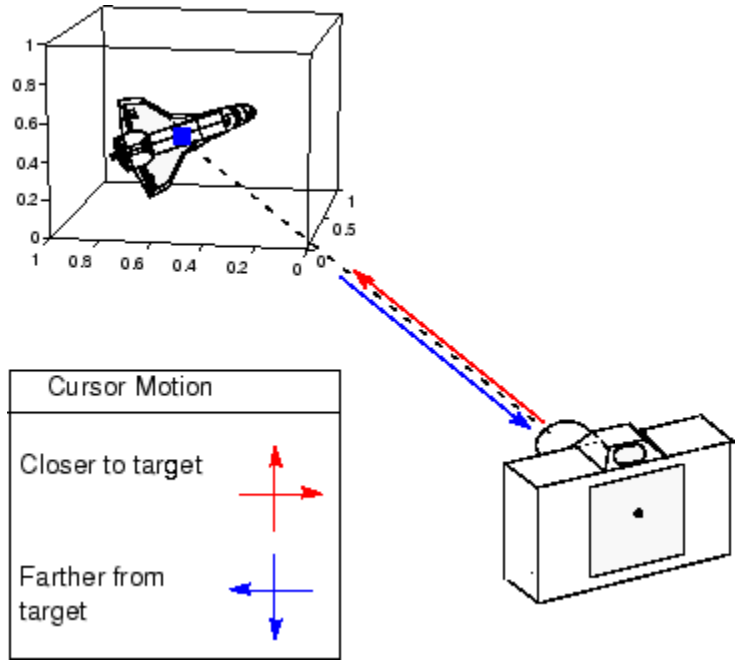


Move Camera Forward and Backward

Moving the cursor up or to the right moves the camera toward the scene. Moving the cursor down or to the left moves the camera away from the scene. It is possible to move the camera through objects in the scene and to the other side of the camera target.

Graphics Properties

This function moves the `CameraPosition` along the line connecting the camera position and the camera target.

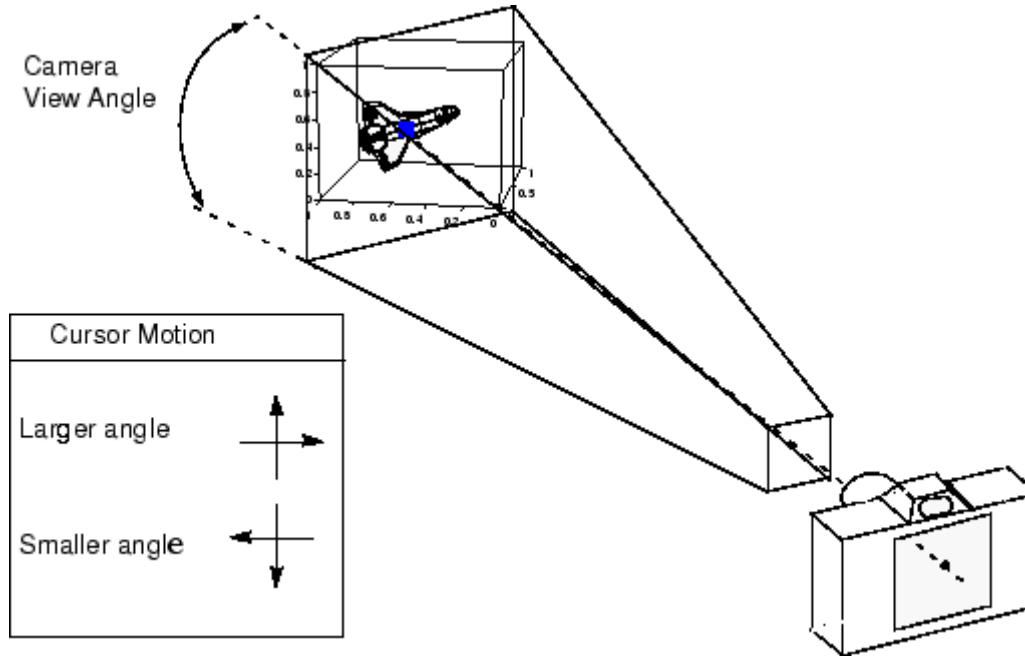


Zoom Camera

Zoom Camera makes the scene larger as you move the cursor up or to the right and smaller as you move the cursor down or to the left. Zooming does not move the camera and therefore cannot move the viewpoint through objects in the scene.

Graphics Properties

Zoom is implemented by changing the `CameraViewAngle`. The larger the angle, the smaller the scene appears, and vice versa.

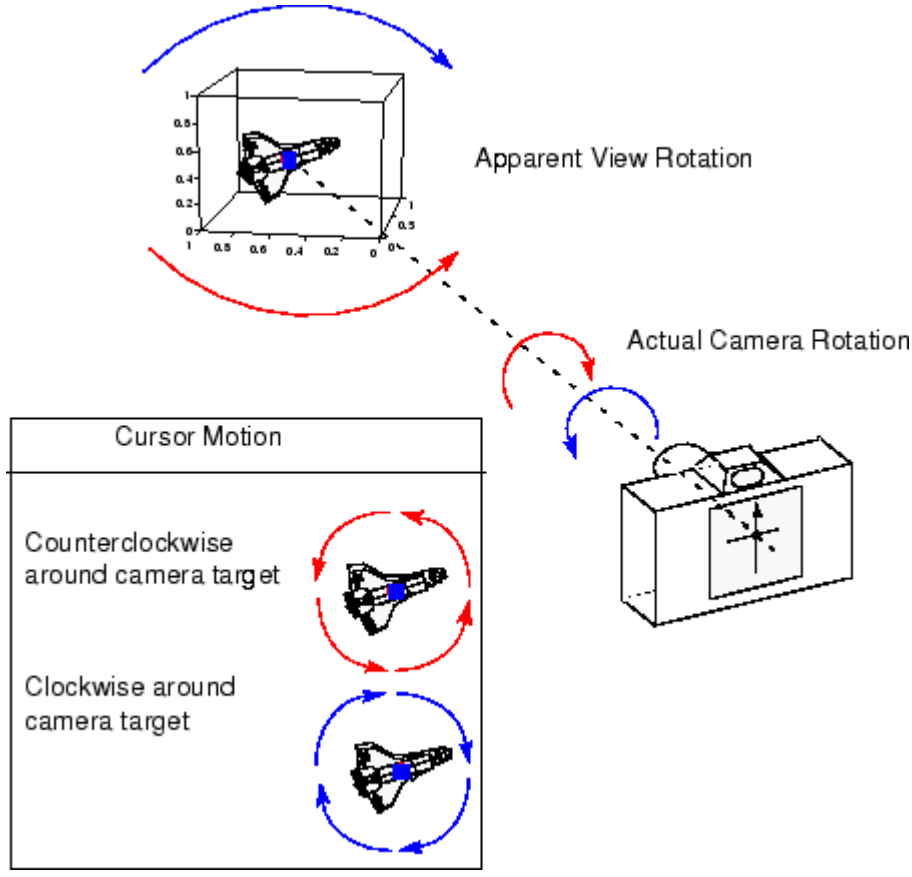


Camera Roll

Camera Roll rotates the camera about the viewing axis, thereby rotating the view on the screen.

Graphics Properties

Camera Roll changes the `CameraUpVector`.


Apparent View Rotation
Actual Camera Rotation
Cursor Motion
Counterclockwise around camera target
Clockwise around camera target

Camera Graphics Functions

The following table lists MATLAB functions that enable you to perform a number of useful camera maneuvers. The individual command descriptions provide information on using each one.

Function	Purpose
`camdolly`	Move camera position and target
`camlookat`	View specific objects
`camorbit`	Orbit the camera about the camera target
`campan`	Rotate the camera target about the camera position
`campos`	Set or get the camera position
`camproj`	Set or get the projection type (orthographic or perspective)
`camroll`	Rotate the camera about the viewing axis
`camtarget`	Set or get the camera target location
`camup`	Set or get the value of the camera up vector
`camva`	Set or get the value of the camera view angle
`camzoom`	Zoom the camera in or out on the scene

Example — Dollying the Camera

In this section...

Summary of Techniques

In the camera metaphor, a dolly is a stage that enables movement of the camera from side to side with respect to the scene. The `camdolly` command implements similar behavior by moving both the position of the camera and the position of the camera target in unison (or just the camera position if you so desire).

This example illustrates how to use `camdolly` to explore different regions of an image. It shows how to use the following functions:

- `ginput` to obtain the coordinates of locations on the image
- The `camdolly` `data` coordinates option to move the camera and target to the new position based on coordinates obtained from `ginput`
- `camva` to zoom in and to fix the camera view angle, which is otherwise under automatic control

Implementation

First load the Cape Cod image and zoom in by setting the camera view angle (using `camva`).

```
load cape
image(X)
colormap(map)
axis image
camva(camva/2.5)
```

Then use `ginput` to select the *x*- and *y*-coordinates of the camera target and camera position.

```
while 1
```

```
  [x,y] = ginput(1);
  if ~strcmp(get(gcf,'SelectionType'),'normal')
    break
  end
  ct = camtarget;
  dx = x - ct(1);
  dy = y - ct(2);
  camdolly(dx,dy,ct(3),'movetarget','data')
  drawnow
end
```

Example — Moving the Camera Through a Scene

In this section...

Summary of Techniques

A fly-through is an effect created by moving the camera through three-dimensional space, giving the impression that you are flying along with the camera as if in an aircraft. You can fly through regions of a scene that might be otherwise obscured by objects in the scene or you can fly by a scene by keeping the camera focused on a particular point.

To accomplish these effects you move the camera along a particular path, the *x*-axis for example, in a series of steps. To produce a fly-through, move both the camera position and the camera target at the same time.

The following example makes use of the fly-though effect to view the interior of an isosurface drawn within a volume defined by a vector field of wind velocities. This data represents air currents over North America.

This example employs a number of visualization techniques. It uses

- Isosurfaces and cone plots to illustrate the flow through the volume
- Lighting to illuminate the isosurface and cones in the volume
- Stream lines to define a path for the camera through the volume
- Coordinated motion of the camera position, camera target, and light

See `coneplot` for a fixed visualization of the same data.

Graphing the Volume Data

The first step is to draw the isosurface and plot the air flow using cone plots.

See `isosurface`, `isonormals`, `reducepatch`, and `coneplot` for information on using these commands.

Setting the data aspect ratio (`daspect`) to `[1,1,1]` before drawing the cone plot enables MATLAB to calculate the size of the cones correctly for the final view.

```
load wind
wind_speed = sqrt(u.^2 + v.^2 + w.^2);

hpatch = patch(isosurface(x,y,z,wind_speed,35));
isonormals(x,y,z,wind_speed,hpatch)
set(hpatch,'FaceColor','red','EdgeColor','none');

[f vt] = reducepatch(isosurface(x,y,z,wind_speed,45),0.05);
daspect([1,1,1]);
hcone = coneplot(x,y,z,u,v,w,vt(:,1),vt(:,2),vt(:,3),2);
set(hcone,'FaceColor','blue','EdgeColor','none');
```

Setting Up the View

You need to define viewing parameters to ensure the scene is displayed correctly:

- Selecting a perspective projection provides the perception of depth as the camera passes through the interior of the isosurface (`camproj`).
- Setting the camera view angle to a fixed value prevents MATLAB from automatically adjusting the angle to encompass the entire scene as well as zooming in the desired amount (`camva`).

  ```
  camproj perspective
  camva(25)
  ```

Specifying the Light Source

Positioning the light source at the camera location and modifying the reflectance characteristics of the isosurface and cones enhances the realism of the scene:

- Creating a light source at the camera position provides a "headlight" that moves along with the camera through the isosurface interior (`camlight`).
- Setting the reflection properties of the isosurface gives the appearance of a dark interior (`AmbientStrength` set to 0.1) with highly reflective material (`SpecularStrength` and `DiffuseStrength` set to 1).
- Setting the `SpecularStrength` of the cones to 1 makes them highly reflective.

```
hlight = camlight('headlight');
set(hpatch,'AmbientStrength',.1,...
      'SpecularStrength',1,...
      'DiffuseStrength',1);
set(hcone,'SpecularStrength',1);
set(gcf,'Color','k')
```

Selecting a Renderer

Because this example uses lighting, MATLAB must use either `zbuffer` or, if available, `OpenGL` renderer settings. The `OpenGL` renderer is likely to be much faster displaying the animation; however, you need to use `gouraud` lighting with OpenGL, which is not as smooth as Phong lighting, which you can use with the `zbuffer` renderer. The two choices are

```
lighting gouraud
set(gcf,'Renderer','OpenGL')
```

or for `zbuffer`

```
lighting phong
set(gcf,'Renderer','zbuffer')
```

Defining the Camera Path as a Stream Line

Stream lines indicate the direction of flow in the vector field. This example uses the *x*-, *y*-, and *z*-coordinate data of a single stream line to map a path

through the volume. The camera is then moved along this path. The tasks include

- Create a stream line starting at the point x = 80, y = 30, z = 11.
- Get the *x*-, *y*-, and *z*-coordinate data of the stream line.
- Delete the stream line (note that you could also use `stream3` to calculate the stream line data without actually drawing the stream line).

```
hsline = streamline(x,y,z,u,v,w,80,30,11);
xd = get(hsline,'XData');
yd = get(hsline,'YData');
zd = get(hsline,'ZData');
delete(hsline)
```

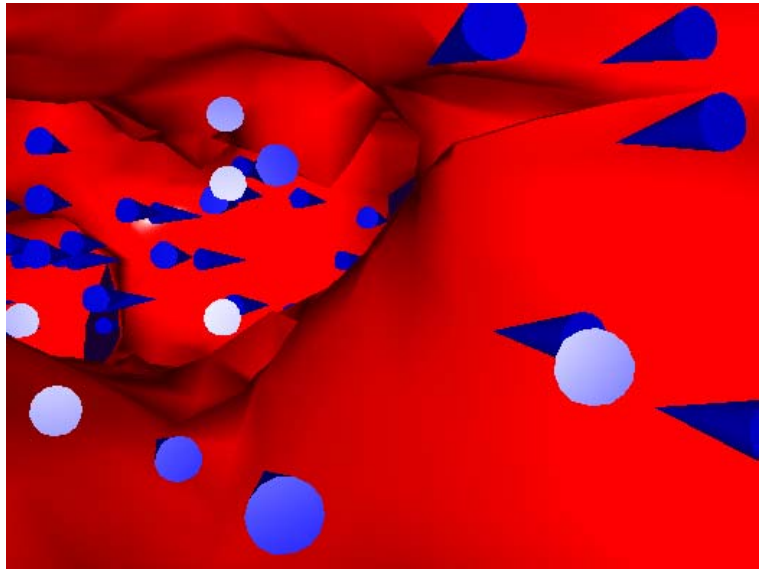
Implementing the Fly-Through

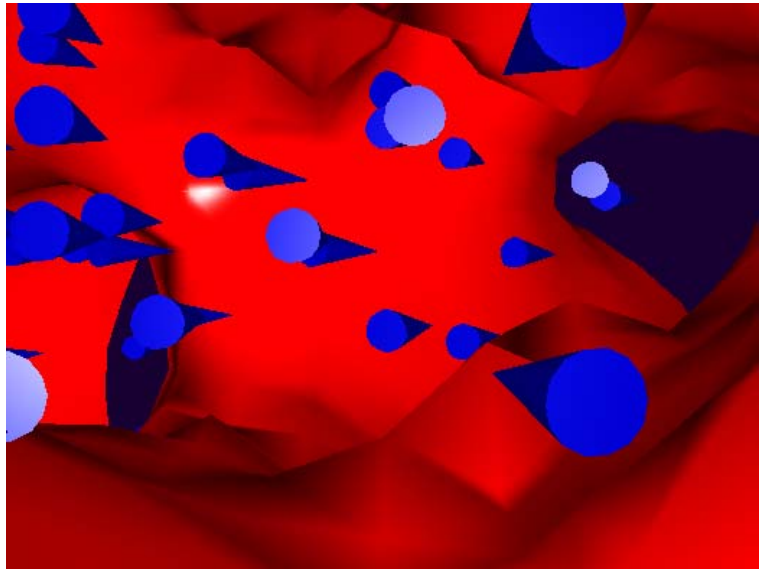
To create a fly-through, move the camera position and camera target along the same path. In this example, the camera target is placed five elements further along the *x*-axis than the camera. A small value is added to the camera target x position to prevent the position of the camera and target from becoming the same point if the condition `xd(n) = xd(n+5)` should occur:

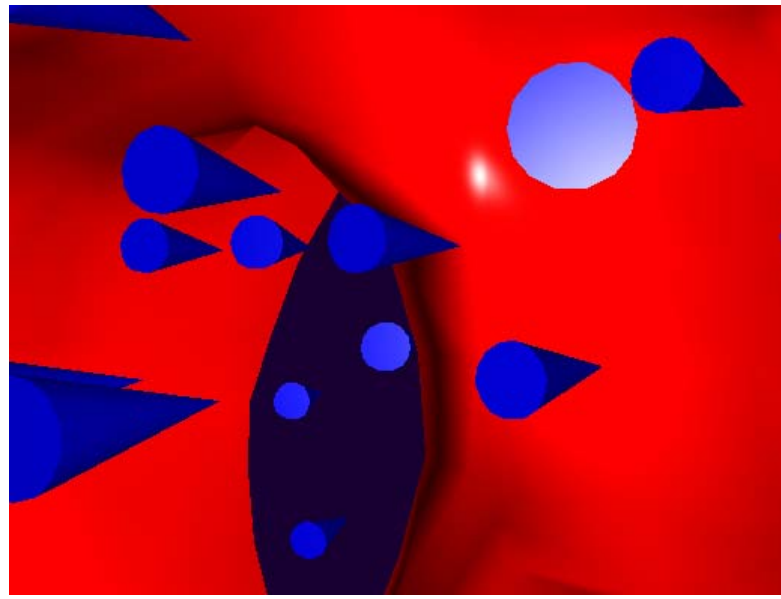
- Update the camera position and camera target so that they both move along the coordinates of the stream line.
- Move the light along with the camera.
- Call `drawnow` to display the results of each move.

```
for i=1:length(xd)-50
  campos([xd(i),yd(i),zd(i)])
  camtarget([xd(i+5)+min(xd)/100,yd(i),zd(i)])
  camlight(hlight,'headlight')
  drawnow
end
```

These snapshots illustrate the view at values of `i` equal to `10`, `110`, and `185`.







Low-Level Camera Properties

In this section...

Camera Properties You Can Set

Camera graphics is based on a group of axes properties that control the position and orientation of the camera. In general, the camera commands make it unnecessary to access these properties directly.

Property	Description
`CameraPosition`	Specifies the location of the viewpoint in axes units.
`CameraPositionMode`	In `automatic` mode, MATLAB determines the position based on the scene. In `manual` mode, you specify the viewpoint location.
`CameraTarget`	Specifies the location in the axes pointed to by the camera. Together with the `CameraPosition`, it defines the viewing axis.
`CameraTargetMode`	In `automatic` mode, MATLAB specifies the `CameraTarget` as the center of the axes plot box. In `manual` mode, you specify the location.
`CameraUpVector`	The rotation of the camera around the viewing axis is defined by a vector indicating the direction taken as up.
`CameraUpVectorMode`	In `automatic` mode, MATLAB orients the up vector along the positive *y*-axis for 2-D views and along the positive *z*-axis for 3-D views. In `manual` mode, you specify the direction.

Property	Description
`CameraViewAngle`	Specifies the field of view of the "lens." If you specify a value for CameraViewAngle, MATLAB overrides stretch-to-fill behavior (see “Understanding Axes Aspect Ratio” on page 2-42).
`CameraViewAngleMode`	In `automatic` mode, MATLAB adjusts the view angle to the smallest angle that captures the entire scene. In `manual` mode, you specify the angle. Setting `CameraViewAngleMode` to `manual` overrides stretch-to-fill behavior.
`Projection`	Selects either an orthographic or perspective projection.

Default Viewpoint Selection

When all the camera mode properties are set to `auto` (the default), MATLAB automatically controls the view, selecting appropriate values based on the assumption that you want the scene to fill the position rectangle (which is defined by the width and height components of the axes `Position` property).

By default, MATLAB

- Sets the `CameraPosition` so the orientation of the scene is the standard MATLAB 2-D or 3-D view (see the `view` command)
- Sets the `CameraTarget` to the center of the plot box
- Sets the `CameraUpVector` so the *y*-direction is up for 2-D views and the *z*-direction is up for 3-D views
- Sets the `CameraViewAngle` to the minimum angle that makes the scene fill the position rectangle (the rectangle defined by the axes `Position` property)
- Uses orthographic projection

This default behavior generally produces desirable results. However, you can change these properties to produce useful effects.

Moving In and Out on the Scene

You can move the camera anywhere in the 3-D space defined by the axes. The camera continues to point towards the target regardless of its position. When the camera moves, MATLAB varies the camera view angle to ensure the scene fills the position rectangle.

Moving Through a Scene

You can create a fly-by effect by moving the camera through the scene. To do this, continually change `CameraPosition` property, moving it toward the target. Because the camera is moving through space, it turns as it moves past the camera target. Override the MATLAB automatic resizing of the scene each time you move the camera by setting the `CameraViewAngleMode` to `manual`.

If you update the `CameraPosition` and the `CameraTarget`, the effect is to pass through the scene while continually facing the direction of movement.

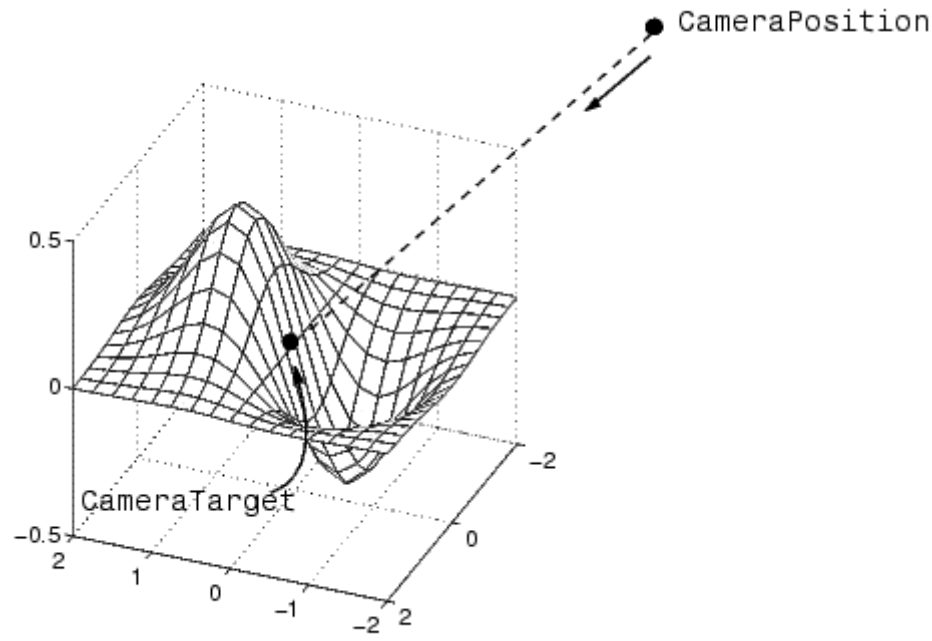
If the `Projection` is set to `perspective`, the amount of perspective distortion increases as the camera gets closer to the target and decreases as it gets farther away.

Example — Moving Toward or Away from the Target

To move the camera along the viewing axis, you need to calculate new coordinates for the `CameraPosition` property. This is accomplished by subtracting (to move closer to the target) or adding (to move away from the target) some fraction of the total distance between the camera position and the camera target.

The function `movecamera` calculates a new `CameraPosition` that moves in on the scene if the argument `dist` is positive and moves out if `dist` is negative.

```
function movecamera(dist) %dist in the range [-1 1]
set(gca,'CameraViewAngleMode','manual')
newcp = cpos - dist * (cpos - ctarg);
set(gca,'CameraPosition',newcp)
function out = cpos
out = get(gca,'CameraPosition');
function out = ctarg
out = get(gca,'CameraTarget');
```



Note that setting the `CameraViewAngleMode` to `manual` overrides MATLAB stretch-to-fill behavior and can cause an abrupt change in the aspect ratio. See “Understanding Axes Aspect Ratio” on page 2-42 for more information on stretch-to-fill.

Making the Scene Larger or Smaller

Adjusting the `CameraViewAngle` property makes the view of the scene larger or smaller. Larger angles cause the view to encompass a larger area, thereby making the objects in the scene appear smaller. Similarly, smaller angles make the objects appear larger.

Changing `CameraViewAngle` makes the scene larger or smaller without affecting the position of the camera. This is desirable if you want to zoom in without moving the viewpoint past objects that will then no longer be in the scene (as could happen if you changed the camera position). Also, changing

the CameraViewAngle does not affect the amount of perspective applied to the scene, as changing CameraPosition does when the figure Projection property is set to perspective.

Revolving Around the Scene

You can use the view command to revolve the viewpoint about the *z*-axis by varying the azimuth, and about the azimuth by varying the elevation. This has the effect of moving the camera around the scene along the surface of a sphere whose radius is the length of the viewing axis. You could create the same effect by changing the CameraPosition, but doing so requires you to perform calculations that MATLAB performs for you when you call view.

For example, the function orbit moves the camera around the scene.

```
function orbit(deg)
[az el] = view;
rotvec = 0:deg/10:deg;
for i = 1:length(rotvec)
    view([az+rotvec(i) el])
    drawnow
end
```

Rotation Without Resizing of Graphics Objects

When CameraViewAngleMode is auto, MATLAB calculates the CameraViewAngle so that the scene is as large as can fit in the axes position rectangle. This causes an apparent size change during rotation of the scene. To prevent resizing during rotation, you need to set the CameraViewAngleMode to manual (which happens automatically when you specify a value for the CameraViewAngle property). To do this in the orbit function, add the statement

```
set(gca,'CameraViewAngleMode','manual')
```

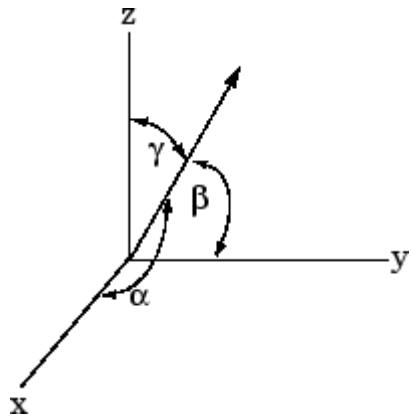
Rotation About the Viewing Axis

You can change the orientation of the scene by specifying the direction defined as up. By default, MATLAB defines *up* as the *y*-axis in 2-D views (the CameraUpVector is [0 1 0]) and the *z*-axis for 3-D views (the

`CameraUpVector` is `[0 0 1]`). However, you can specify up as any arbitrary direction.

The vector defined by the `CameraUpVector` property forms one axis of the camera's coordinate system. Internally, MATLAB determines the actual orientation of the camera up vector by projecting the specified vector onto the plane that is normal to the camera direction (i.e., the viewing axis). This simplifies the specification of the `CameraUpVector` property, because it need not lie in this plane.

In many cases, you might find it convenient to visualize the desired up vector in terms of angles with respect to the axes *x*-, *y*-, and *z*-axis. You can then use *direction cosines* to convert from angles to vector components. For a unit vector, the expression simplifies to



where the angles α, β, and γ are specified in degrees.

```
XComponent = cos(α x (pi   180));
YComponent = cos(β x (pi   180));
ZComponent = cos([[GAMMA]] x (pi   180));
```

(Consult a mathematics book on vector analysis for a more detailed explanation of direction cosines.)

Example — Calculating a Camera Up Vector

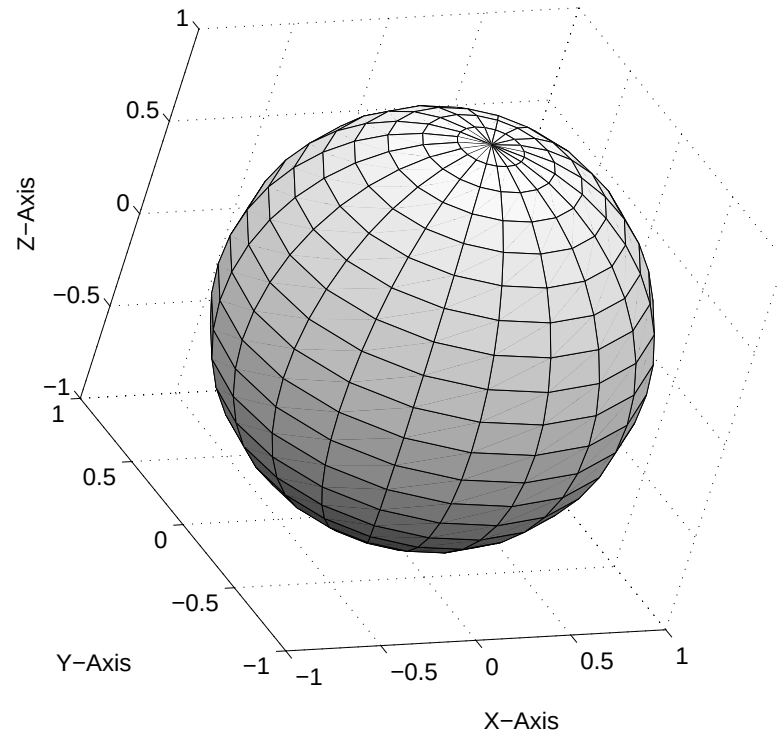
To specify an up vector that makes an angle of 30° with the z-axis and lies in the y-z plane, use the expression

```
upvec = [cos(90*(pi/180)),cos(60*(pi/180)),cos(30*(pi/180))];
```

and then set the `CameraUpVector` property.

```
set(gca,'CameraUpVector',upvec)
```

Drawing a sphere with this orientation produces



Understanding View Projections

In this section...

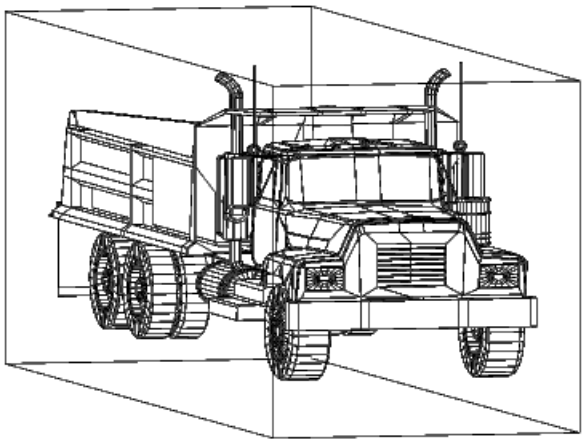
The Two Types of Projections

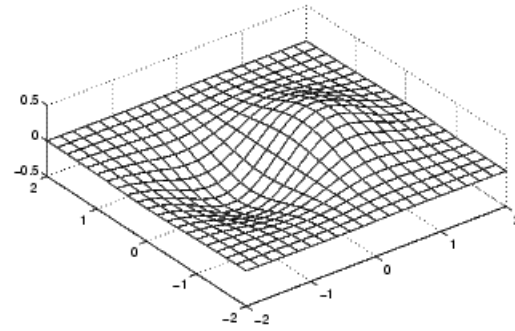
MATLAB supports both orthographic and perspective projection types for displaying 3-D graphics. The one you select depends on the type of graphics you are displaying:

- `orthographic` projects the viewing volume as a rectangular parallelepiped (i.e., a box whose opposite sides are parallel). Relative distance from the camera does not affect the size of objects. This projection type is useful when it is important to maintain the actual size of objects and the angles between objects.
- `perspective` projects the viewing volume as the frustum of a pyramid (a pyramid whose apex has been cut off parallel to the base). Distance causes foreshortening; objects further from the camera appear smaller. This projection type is useful when you want to display realistic views of real objects.

By default, MATLAB displays objects using orthographic projection. You can set the projection type using the `camproj` command.

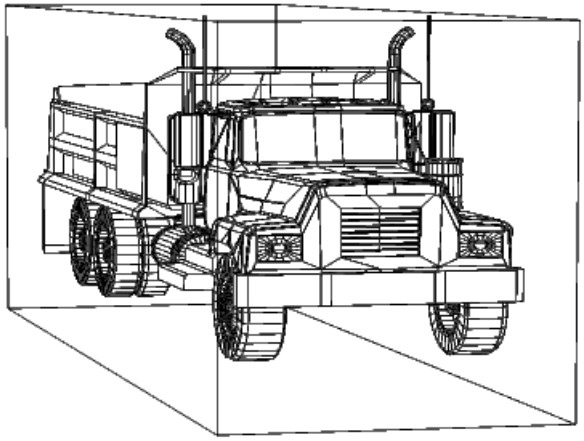
These pictures show a drawing of a dump truck (created with `patch`) and a surface plot of a mathematical function, both using orthographic projection.

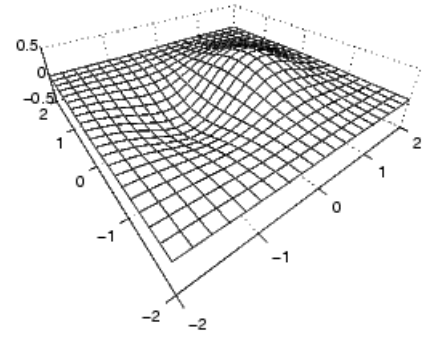




If you measure the width of the front and rear faces of the box enclosing the dump truck, you'll see they are the same size. This picture looks unnatural because it lacks the apparent perspective you see when looking at real objects with depth. On the other hand, the surface plot accurately indicates the values of the function within rectangular space.

Now look at the same graphics objects with perspective added. The dump truck looks more natural because portions of the truck that are farther from the viewer appear smaller. This projection mimics the way human vision works. The surface plot, on the other hand, looks distorted.



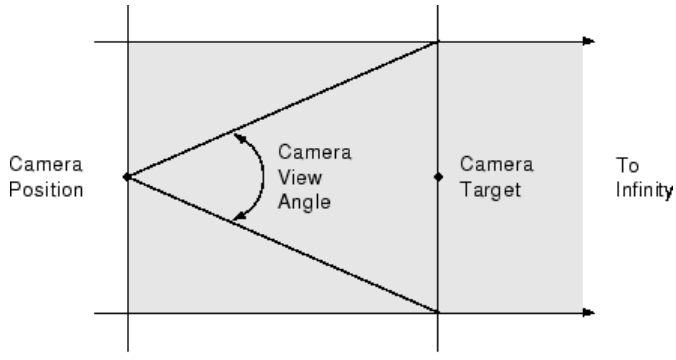


Projection Types and Camera Location

By default, MATLAB adjusts the `CameraPosition`, `CameraTarget`, and `CameraViewAngle` properties to point the camera at the center of the scene and to include all graphics objects in the axes. If you position the camera so that there are graphics objects behind the camera, the scene displayed can be affected by both the axes `Projection` property and the figure `Renderer` property. The following summarizes the interactions between projection type and rendering method.

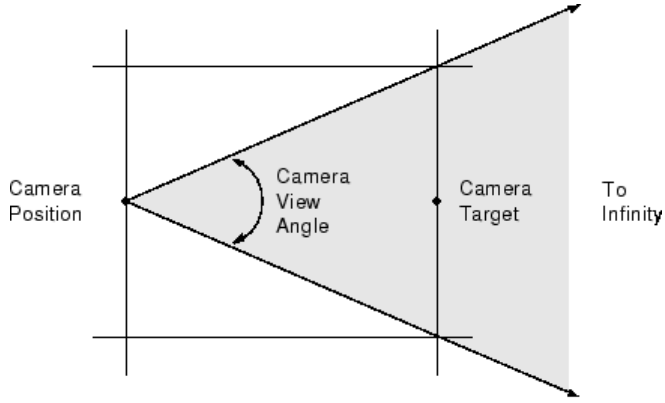
	Orthographic	**Perspective**
Z-buffer	`CameraViewAngle` determines extent of scene at `CameraTarget`.	`CameraViewAngle` determines extent of scene from `CameraPosition` to infinity.
Painters	All objects are displayed regardless of `CameraPosition`.	Not recommended if graphics objects are behind the `CameraPosition`.

This diagram illustrates what you see (gray area) when using orthographic projection and Z-buffer. Anything in front of the camera is visible.



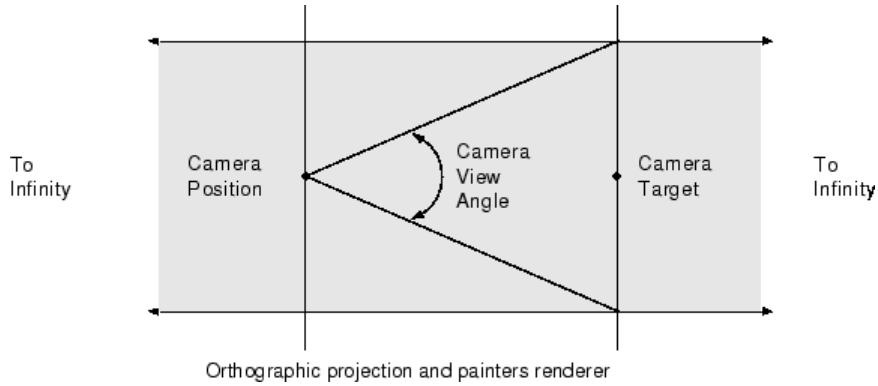
Orthographic projection and Z-buffer renderer

In perspective projection, you see only what is visible in the cone of the camera view angle.



Perspective projection and Z-buffer renderer

Painters rendering method is less suited to moving the camera in 3-D space because MATLAB does not clip along the viewing axis. Orthographic projection in painters method results in all objects contained in the scene being visible regardless of the camera position.



Orthographic projection and painters renderer

Printing 3-D Scenes

The same effects described in the previous section occur in hardcopy output. However, because of the differences in the process of rendering to the screen and to a printing format, MATLAB might render using Z-buffer and generate printed output using painters. You might need to specify Z-buffer printing explicitly to obtain the results displayed on the screen (use the `-zbuffer` option with the `print` command).

Additional Information

See Basic Printing and Exporting and Selecting a Renderer in Figure Properties in the Using MATLAB Graphics documentation for information on printing and rendering methods.

Understanding Axes Aspect Ratio

In this section...

Stretch-to-Fill

Axes shape graphics objects by setting the scaling and limits of each axis. When you create a graph, MATLAB automatically determines axis scaling based on the values or size of the plotted data, and then draws the axes to fit the space available for display. Axes aspect ratio properties control how MATLAB performs the scaling required to create a graph.

By default, the size of the axes MATLAB creates for plotting is normalized to the size of the figure window (but is slightly smaller to allow for borders). If you resize the figure, the size and possibly the aspect ratio (the ratio of width to height) of the axes changes proportionally. This enables the axes to always fill the available space in the window. MATLAB also sets the *x*-, *y*-, and *z*-axis limits to provide the greatest resolution in each direction, again optimizing the use of available space.

This stretch-to-fill behavior is generally desirable; however, you might want to control this process to produce specific results. For example, images need to be displayed in correct proportions regardless of the aspect ratio of the figure window, or you might want graphs always to be a particular size on a printed page.

Specifying Axis Scaling

The `axis` command enables you to adjust the scaling of graphs. By default, MATLAB finds the maxima and minima of the plotted data and chooses appropriate axes ranges. You can override the defaults by setting axis limits.

```
axis([xmin xmax ymin ymax zmin zmax])
```

You can control how MATLAB scales the axes with predefined `axis` options:

- `axis auto` returns the axis scaling to its default, automatic mode. `v = axis` saves the scaling of the axes of the current plot in vector `v`. For subsequent graphics commands to have these same axis limits, follow them with `axis(v)`.
- `axis manual` freezes the scaling at the current limits. If you then set `hold on`, subsequent plots use the current limits. Specifying values for axis limits also sets axis scaling to manual.
- `axis tight` sets the axis limits to the range of the data.
- `axis ij` places MATLAB into its "matrix" axes mode. The coordinate system origin is at the upper left corner. The *i*-axis is vertical and is numbered from top to bottom. The *j*-axis is horizontal and is numbered from left to right.
- `axis xy` places MATLAB into its default Cartesian axes mode. The coordinate system origin is at the lower left corner. The *x*-axis is horizontal and is numbered from left to right. The *y*-axis is vertical and is numbered from bottom to top.

Specifying Aspect Ratio

The `axis` command enables you to adjust the aspect ratio of graphs. Normally MATLAB stretches the axes to fill the window. In many cases, it is more useful to specify the aspect ratio of the axes based on a particular characteristic such as the relative length or scaling of each axis. The `axis` command provides a number of useful options for adjusting the aspect ratio:

- `axis equal` changes the current axes scaling so that equal tick mark increments on the *x*-, *y*-, and *z*-axis are equal in length. This makes the surface displayed by `sphere` look like a sphere instead of an ellipsoid. `axis equal` overrides stretch-to-fill behavior.
- `axis square` makes each axis the same length and overrides stretch-to-fill behavior.
- `axis vis3d` freezes aspect ratio properties to enable rotation of 3-D objects and overrides stretch-to-fill. Use this option after other `axis` options to keep settings from changing while you rotate the scene.

- `axis image` makes the aspect ratio of the axes the same as the image.
- axis auto returns the *x*-, *y*-, and *z*-axis limits to automatic selection mode.
- `axis normal` restores the current axis box to full size and removes any restrictions on the scaling of the units. It undoes the effects of `axis square`. Used in conjunction with `axis auto`, it undoes the effects of `axis equal`.

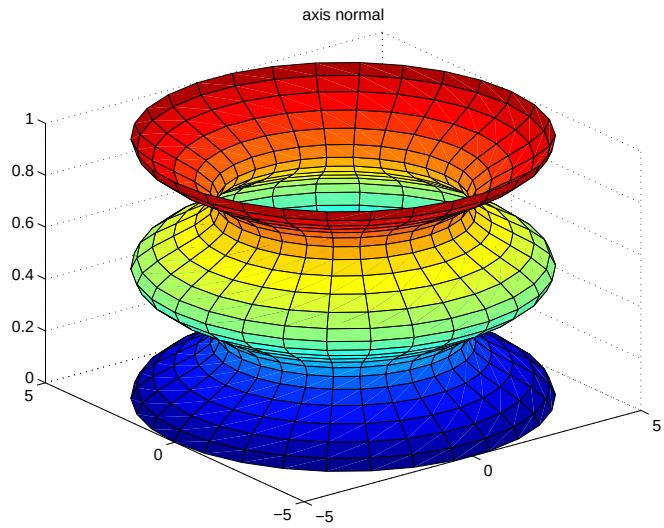
The `axis` command works by manipulating axes graphics object properties.

Example — axis Command Options

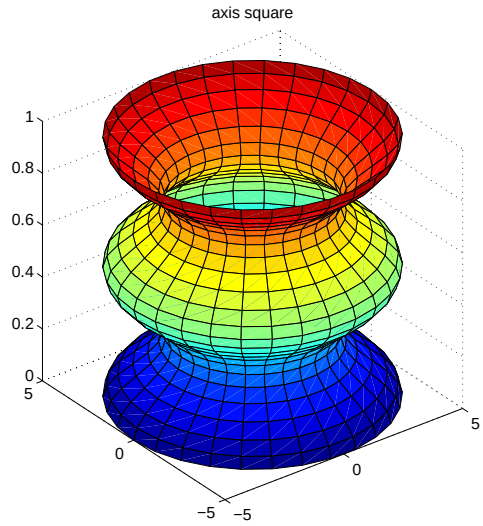
The following three pictures illustrate the effects of three `axis` options on a cylindrical surface created with the statements

```
t = 0:pi/6:4*pi;
[x,y,z] = cylinder(4+cos(t),30);
surf(x,y,z)
```

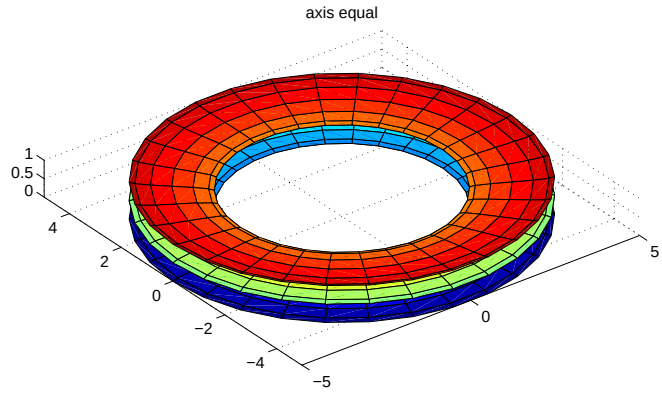
`axis normal` is the default behavior. MATLAB automatically sets the axis limits to span the data range along each axis and stretches the plot to fit the figure window.



`axis square` creates an axes that is square regardless of the shape of the figure window. The cylindrical surface is no longer distorted because it is not warped to fit the window. However, the size of one data unit is not equal along all axes (the *z*-axis spans only one unit while the *x*-axes and *y*-axes span 10 units each).



`axis equal` makes the length of one data unit equal along each axis while maintaining a nearly square plot box. It also prevents warping of the axis to fill the window's shape.



Additional Commands for Setting Aspect Ratio

You can control the aspect ratio of your graph in three ways:

- Specifying the relative scales of the *x*-, *y*-, and *z*-axes (data aspect ratio)
- Specifying the shape of the space defined by the axes (plot box aspect ratio)
- Specifying the axis limits

The following commands enable you to set these values.

Command	Purpose
`daspect`	Set or query the data aspect ratio
`pbaspect`	Set or query the plot box aspect ratio
`xlim`	Set or query *x*-axis limits
`ylim`	Set or query *y*-axis limits
`zlim`	Set or query *z*-axis limits

See “Manipulating Axes Aspect Ratio” on page 2-47 for a list of the axes properties that control aspect ratio.

Manipulating Axes Aspect Ratio

In this section...

Axes Aspect Ratio Properties

The `axis` command works by setting various axes object properties. You can set these properties directly to achieve precisely the effect you want.

Property	Description
`DataAspectRatio`	Sets the relative scaling of the individual axis data values. Set `DataAspectRatio` to `[1 1 1]` to display real-world objects in correct proportions. Specifying a value for DataAspectRatio overrides stretch-to-fill behavior.
`DataAspectRatioMode`	In `auto`, MATLAB selects axis scales that provide the highest resolution in the space available.
`PlotBoxAspectRatio`	Sets the proportions of the axes plot box (set `box` to `on` to see the box). Specifying a value for PlotBoxAspectRatio overrides stretch-to-fill behavior.
`PlotBoxAspectRatioMode`	In `auto`, MATLAB sets the `PlotBoxAspectRatio` to `[1 1 1]` unless you explicitly set the `DataAspectRatio` and/or the axis limits.
`Position`	Defines the location and size of the axes with a four-element vector: [*left offset*, *bottom offset*, *width*, *height*].
`XLim, YLim, ZLim`	Sets the minimum and maximum limits of the respective axes.
`XLimMode, YLimMode, ZLimMode`	In `auto`, MATLAB selects the axis limits.

By default, MATLAB automatically determines values for all of these properties (i.e., all the modes are auto) and then applies stretch-to-fill. You can override any property's automatic operation by specifying a value for the property or setting its mode to manual. The value you select for a particular property depends primarily on what type of data you want to display.

Much of the data visualized with MATLAB is either

- Numerical data displayed as line or mesh plots
- Representations of real-world objects (e.g., a dump truck or a section of the earth's topography)

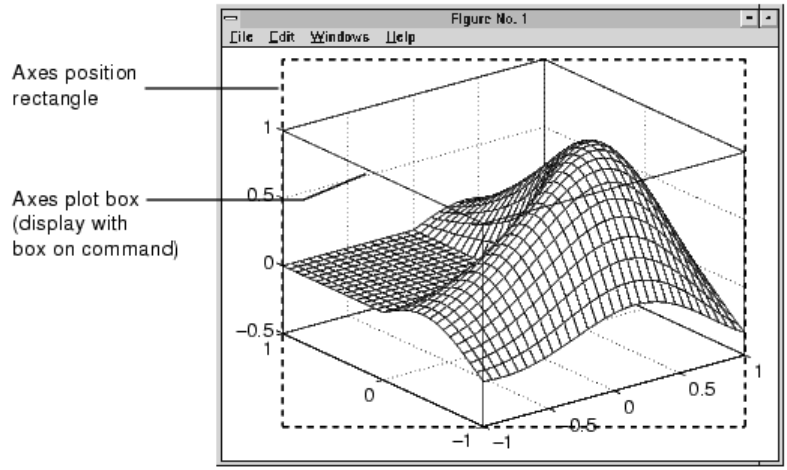
In the first case, it is generally desirable to select axis limits that provide good resolution in each axis direction and to fill the available space. Real-world objects, on the other hand, need to be represented accurately in proportion, regardless of the angle of view.

Default Aspect Ratio Selection

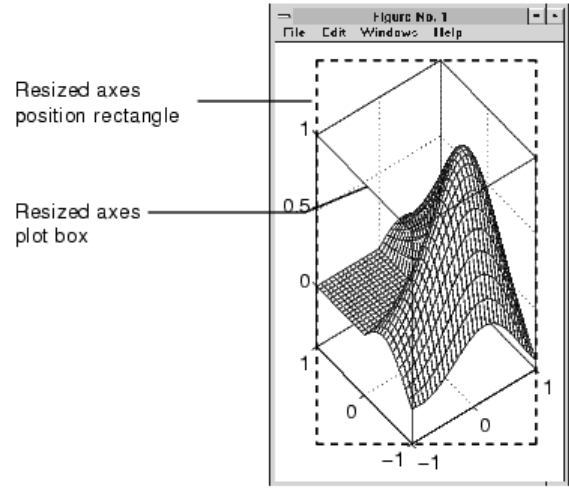
There are two key elements to MATLAB default behavior — normalizing the axes size to the window size and stretch-to-fill.

The axes `Position` property specifies the location and dimensions of the axes. The third and fourth elements of the `Position` vector (`width` and `height`) define a rectangle in which MATLAB draws the axes (indicated by the dotted line in the following pictures). MATLAB stretches the axes to fill this rectangle.

The default value for the axes `Units` property is `normalized` to the parent figure dimensions. This means the shape of the figure window determines the shape of the position rectangle. As you change the size of the window, MATLAB reshapes the position rectangle to fit it.



The view is the 2-D projection of the plot box onto the screen.



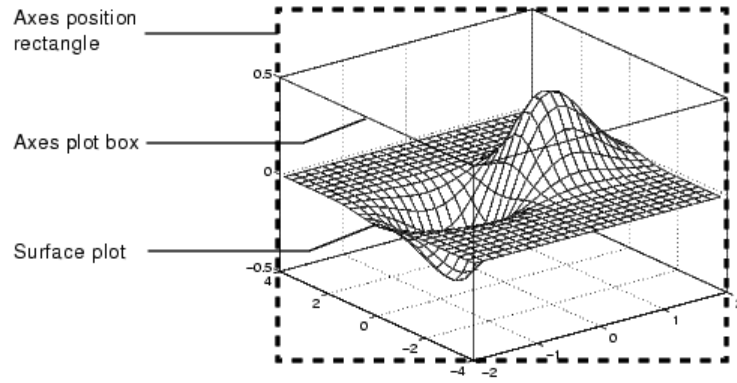
As you can see, reshaping the axes to fit into the figure window can change the aspect ratio of the graph. MATLAB applies stretch-to-fill so the axes fill the position rectangle and in the process can distort the shape. This is

generally desirable for graphs of numeric data, but not for displaying objects realistically.

Example — MATLAB Defaults

MATLAB surface plots are well suited for visualizing mathematical functions of two variables. For example, to display a mesh plot of the function $z = xe^{(-x^2 - y^2)}$ evaluated over the range $-2 \le x \le 2$, $-4 \le y \le 4$, use the statements

```
[X,Y] = meshgrid([-2:.15:2],[-4:.3:4]);
Z = X.*exp(-X.^2 - Y.^2);
mesh(X,Y,Z)
```



The MATLAB default property values are designed to

- Select axis limits to span the range of the data (`XLimMode`, `YLimMode`, and `ZLimMode` are set to `auto`).
- Provide the highest resolution in the available space by setting the scale of each axis independently (`DataAspectRatioMode` and the `PlotBoxAspectRatioMode` are set to `auto`).
- Draw axes that fit the position rectangle by adjusting the `CameraViewAngle` and then stretch-to-fill the axes if necessary.

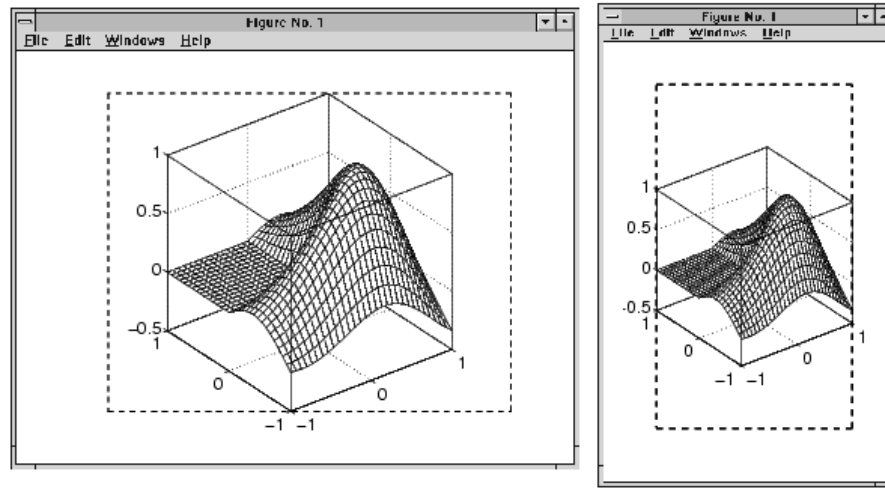
Overriding Stretch-to-Fill

To maintain a particular shape, you can specify the size of the axes in absolute units such as inches, which are independent of the figure window size. However, this is not a good approach if you are writing an M-file that you want to work with a figure window of any size. A better approach is to specify the aspect ratio of the axes and override automatic stretch-to-fill.

In cases where you want a specific aspect ratio, you can override stretching by specifying a value for these axes properties:

- `DataAspectRatio` or `DataAspectRatioMode`
- `PlotBoxAspectRatio` or `PlotBoxAspectRatioMode`
- `CameraViewAngle` or `CameraViewAngleMode`

The first two sets of properties affect the aspect ratio directly. Setting either of the mode properties to `manual` simply disables stretch-to-fill while maintaining all current property values. In this case, MATLAB enlarges the axes until one dimension of the position rectangle constrains it.



Setting the `CameraViewAngle` property disables stretch-to-fill, and also prevents MATLAB from readjusting the size of the axes if you change the view.

Effects of Setting Aspect Ratio Properties

It is important to understand how properties interact with each other, in order to obtain the results you want. The `DataAspectRatio`, `PlotBoxAspectRatio`, and the *x*-, *y*-, and *z*-axis limits (`XLim`, `YLim`, and `ZLim` properties) all place constraints on the shape of the axes.

Data Aspect Ratio

The `DataAspectRatio` property controls the ratio of the axis scales. For a mesh plot of the function $z = xe^{(-x^2 - y^2)}$ evaluated over the range $-2 \le x \le 2$, $-4 \le y \le 4$

```
[X,Y] = meshgrid([-2:.15:2],[-4:.3:4]);
Z = X.*exp(-X.^2 - Y.^2);
mesh(X,Y,Z)
```

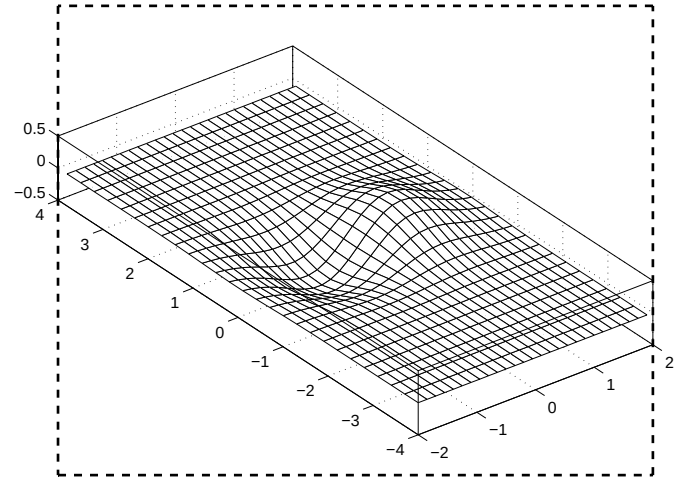
the values are

```
get(gca,'DataAspectRatio')
ans =
     4 8 1
```

This means that four units in length along the *x*-axis cover the same data values as eight units in length along the *y*-axis and one unit in length along the *z*-axis. The axes fill the plot box, which has an aspect ratio of [1 1 1] by default.

If you want to view the mesh plot so that the relative magnitudes along each axis are equal with respect to each other, you can set the `DataAspectRatio` to `[1 1 1]`.

```
set(gca,'DataAspectRatio',[1 1 1])
```



Setting the value of the `DataAspectRatio` property also sets the `DataAspectRatioMode` to `manual` and overrides stretch-to-fill so the specified aspect ratio is achieved.

Plot Box Aspect Ratio

Looking at the value of the `PlotBoxAspectRatio` for the graph in the previous section shows that it has now taken on the former value of the `DataAspectRatio`.

```
get(gca,'PlotBoxAspectRatio')
ans =
     4 8 1
```

MATLAB has rescaled the plot box to accommodate the graph using the specified `DataAspectRatio`.

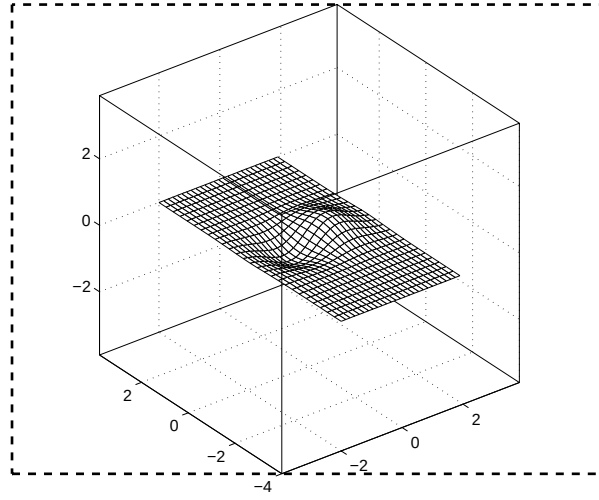
The `PlotBoxAspectRatio` property controls the shape of the axes plot box. MATLAB sets this property to `[1 1 1]` by default and adjusts the `DataAspectRatio` property so that graphs fill the plot box if stretching is on, or until reaching a constraint if stretch-to-fill has been overridden.

When you set the value of the DataAspectRatio and thereby prevent it from changing, MATLAB varies the PlotBoxAspectRatio instead. If you specify both the DataAspectRatio and the PlotBoxAspectRatio, MATLAB is forced to change the axis limits to obey the two constraints you have already defined.

Continuing with the mesh example, if you set both properties,

```
set(gca,'DataAspectRatio',[1 1 1],...
        'PlotBoxAspectRatio',[1 1 1])
```

MATLAB changes the axis limits to satisfy the two constraints placed on the axes.



Adjusting Axis Limits

MATLAB enables you to set the axis limits to the values you want. However, specifying a value for DataAspectRatio, PlotBoxAspectRatio, and the axis limits overconstrains the axes definition. For example, it is not possible for MATLAB to draw the axes if you set these values:

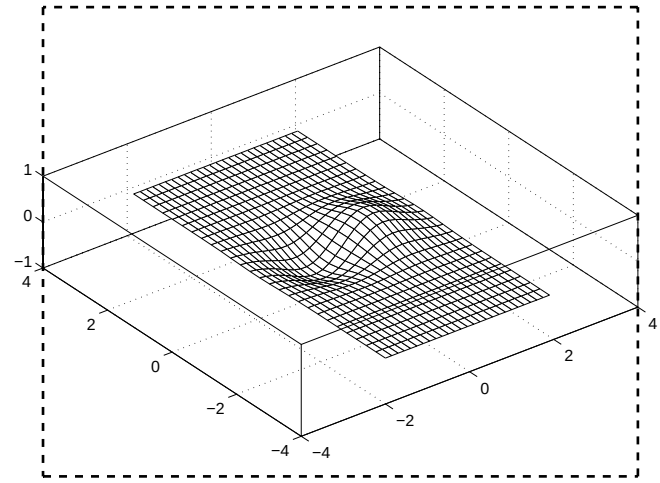
```
set(gca,'DataAspectRatio',[1 1 1],...
        'PlotBoxAspectRatio',[1 1 1],...
```

```
        'XLim',[-4 4],...
        'YLim',[-4 4],...
        'ZLim',[-1 1])
```

In this case, MATLAB ignores the setting of the `PlotBoxAspectRatio` and determines its value automatically. These particular values cause the `PlotBoxAspectRatio` to return to its calculated value.

```
get(gca,'PlotBoxAspectRatio')
ans =
      4 8 1
```

MATLAB can now draw the axes using the specified `DataAspectRatio` and axis limits.



Example — Displaying Cross-Sections of Surfaces

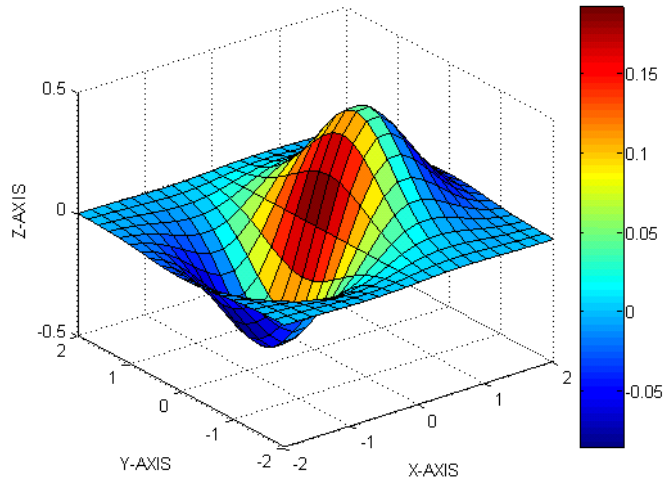
Sometimes projecting a 3-D surface onto an *x*-, *y*-, or *z*-axis can aid visualization. To do this, you might change the aspect ratio, in order to make space for the projection. The following example illustrates how to do this:

1 Create an *x*-*y* grid and *z*-values for it:

```
[x,y] = meshgrid([-2:.2:2]);
Z = x.*exp(-x.^2-y.^2);
```

2 Plot the surface in 3-D; annotate with a colorbar and axis labels:

```
surf(x,y,Z,gradient(Z))
colorbar
xlabel('X-AXIS')
ylabel('Y-AXIS')
zlabel('Z-AXIS')
```



3 Use `axis` to change the `Ymax` value in to 3, stretching the plot in one direction:

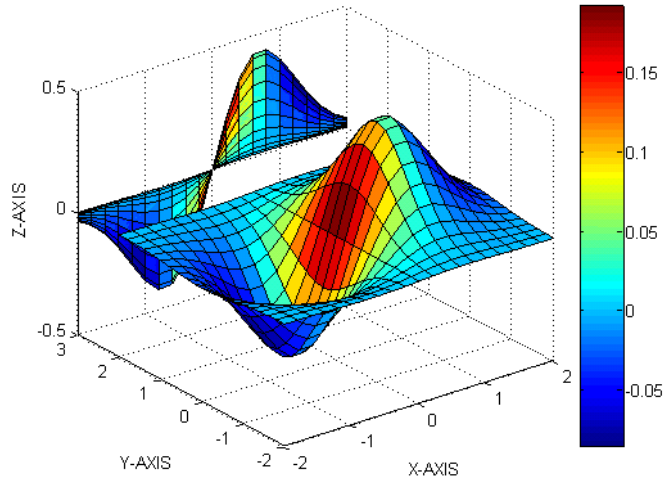
```
axis([-2 2 -2 3 -0.5 0.5]) %
```

4 Regrid the surface, setting all Y-values equal to 3:

```
y = 3*ones(21);
```

5 Plaster a plot of the surface onto the Y-axis:

```
hold on
surf(x,y,Z,gradient(Z))
```



Example — Displaying Real Objects

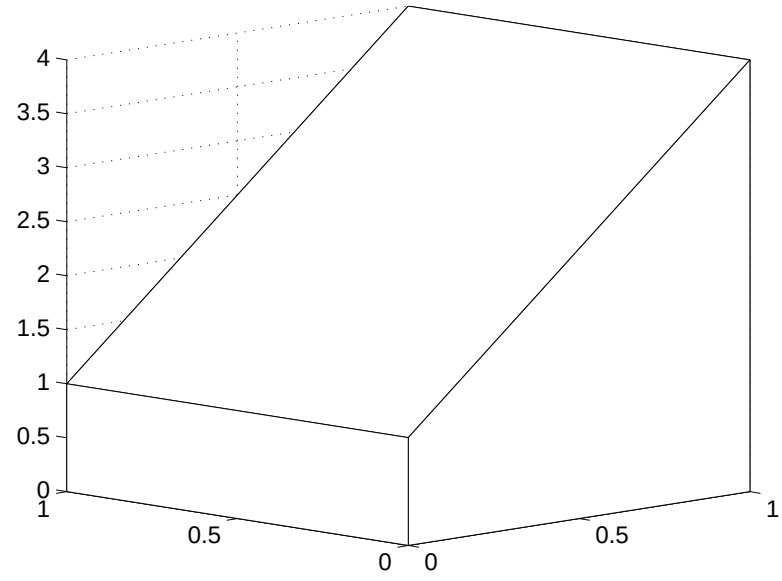
If you want to display an object so that it looks realistic, you need to change MATLAB defaults. For example, this data defines a wedge-shaped patch object.

```
vertex_list =

     0     0     0
     0     1     0
     1     1     0
     1     0     0
     0     0     1
     0     1     1
     1     1     4
     1     0     4
```

```
vertex_connection =

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

```
patch('Vertices',vertex_list,'Faces',vertex_connection,...
```

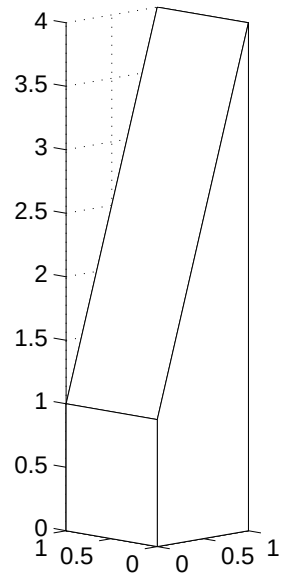
```
 'FaceColor','w','EdgeColor','k')
view(3)
```



However, this axes distorts the actual shape of the solid object defined by the data. To display it in correct proportions, set the `DataAspectRatio`.

```
set(gca,'DataAspectRatio',[1 1 1])
```

The units are now equal in the *x*-, *y*-, and *z*-directions and the axes is not being stretched to fill the position rectangle, revealing the true shape of the object.


4
3.5
3
2.5
2
1.5
1
0.5
0
1
0.5
0
0
0.5
1

3

Lighting as a Visualization Tool

Lighting Overview

In this section...

Lighting Commands

MATLAB provides commands that enable you to position light sources and adjust the characteristics of lit objects. These commands include the following.

Command	Purpose
`camlight`	Create or move a light with respect to the camera position
`lightangle`	Create or position a light in spherical coordinates
`light`	Create a light object
`lighting`	Select a lighting method
`material`	Set the reflectance properties of lit objects

You might find it useful to set light or lit-object properties directly to achieve specific results. In addition to the material in this topic area, you can explore the following lighting examples as an introduction to lighting for visualization.

Light Objects

You create a light object using the `light` function. Three important light object properties are

- `Color` — Color of the light cast by the light object
- `Style` — Either infinitely far away (the default) or local
- `Position` — Direction (for infinite light sources) or the location (for local light sources)

The `Color` property determines the color of the directional light from the light source. The color of an object in a scene is determined by the color of the object and the light source.

The `Style` property determines whether the light source is a point source (`Style` set to `local`), which radiates from the specified position in all directions, or a light source placed at infinity (`Style` set to `infinite`), which shines from the direction of the specified position with parallel rays.

The `Position` property specifies the location of the light source in axes data units. In the case of a light source at infinity, `Position` specifies the direction to the light source.

Lights affect surface and patch objects that are in the same axes as the light. These objects have a number of properties that alter the way they look when illuminated by lights.

Properties That Affect Lighting

You cannot see light objects themselves, but you can see their effects on any patch and surface objects present in the axes containing the light. A number of functions create these objects, including `surf`, `mesh`, `pcolor`, `fill`, and `fill3` as well as the `surface` and `patch` functions.

You control lighting effects by setting various axes, light, patch, and surface object properties. All properties have default values that generally produce desirable results. However, you can achieve the specific effect you want by adjusting the values of these properties.

Property	Effect
`AmbientLightColor`	An axes property that specifies the color of the background light in the scene, which has no direction and affects all objects uniformly. Ambient light effects occur only when there is a visible light object in the axes.
`AmbientStrength`	A patch and surface property that determines the intensity of the ambient component of the light reflected from the object.
`DiffuseStrength`	A patch and surface property that determines the intensity of the diffuse component of the light reflected from the object.

Property	Effect
`SpecularStrength`	A patch and surface property that determines the intensity of the specular component of the light reflected from the object.
`SpecularExponent`	A patch and surface property that determines the size of the specular highlight.
`SpecularColorReflectance`	A patch and surface property that determines the degree to which the specularly reflected light is colored by the object color or the light source color.
`FaceLighting`	A patch and surface property that determines the method used to calculate the effect of the light on the faces of the object. Choices are either no lighting, or flat, Gouraud, or Phong lighting algorithms.
`EdgeLighting`	A patch and surface property that determines the method used to calculate the effect of the light on the edges of the object. Choices are either no lighting, or flat, Gouraud, or Phong lighting algorithms.
`BackFaceLighting`	A patch and surface property that determines how faces are lit when their vertex normals point away from the camera. This property is useful for discriminating between the internal and external surfaces of an object.
`FaceColor`	A patch and surface property that specifies the color of the object faces.
`EdgeColor`	A patch and surface property that specifies the color of the object edges.
`VertexNormals`	A patch and surface property that contains normal vectors for each vertex of the object. MATLAB uses vertex normal vectors to perform lighting calculations. While MATLAB automatically generates this data, you can also specify your own vertex normals.
`NormalMode`	A patch and surface property that determines whether MATLAB recalculates vertex normals if you change object data (`auto`) or uses the current values of the `VertexNormals` property (`manual`). If you specify values for `VertexNormals`, MATLAB sets this property to `manual`.

For more information, see descriptions of `axes`, `surface`, and `patch` object properties.

Examples of Lighting Control

Lighting is a technique for adding realism to a graphical scene. It does this by simulating the highlights and dark areas that occur on objects under natural lighting (e.g., the directional light that comes from the sun). To create lighting effects, MATLAB defines a graphics object called a light. MATLAB applies lighting to surface and patch objects.

These examples illustrate the use of lighting in a visualization context.

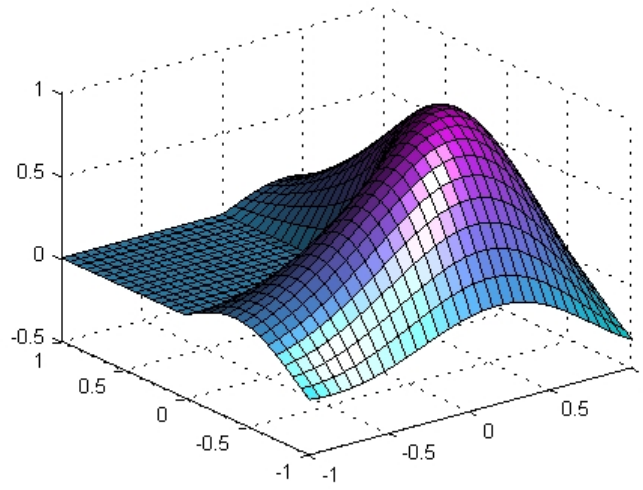
- Tracing a stream line through a volume — Sets properties of surfaces, patches, and lights (MATLAB Graphics documentation).
- Using slice planes and cone plots — Sets lighting characteristics of objects in a scene independently to achieve a desired result (MATLAB `coneplot` function).
- Lighting multiple slice planes independently to visualize fluid flow (MATLAB Graphics documentation).
- Combining single-color lit surfaces with interpolated coloring. See "Example — Visualizing MRI Data" (3–D Visualization documentation).
- Employing lighting to reveal surface shape. The fluid flow isosurface example and the surface plot of the `sinc` function examples illustrate this technique (3–D Visualization documentation).

Example — Adding Lights to a Scene

This example displays the membrane surface and illuminates it with a light source emanating from the direction defined by the position vector `[0 -2 1]`. This vector defines a direction from the axes origin passing through the point with the coordinates 0, -2, 1. The light shines from this direction toward the axes origin.

```
membrane
light('Position',[0 -2 1])
```

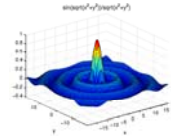
Creating a light activates a number of lighting-related properties controlling characteristics such as the ambient light and reflectance properties of objects. It also switches to Z-buffer renderer if not already in that mode.



Example — Illuminating Mathematical Functions

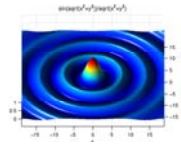
Lighting can enhance surface graphs of mathematical functions. For example, use the `ezsurf` command to evaluate the expression $\sin(\sqrt{x^2+y^2}) \div \sqrt{x^2+y^2}$ over the region -6π to 6π.

```
ezsurf('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)',[-6*pi,6*pi])
```



Now add lighting using the `lightangle` command, which accepts the light position in terms of azimuth and elevation.

```
view(0,75)
shading interp
lightangle(-45,30)
set(gcf,'Renderer','zbuffer')
set(findobj(gca,'type','surface'),...
    'FaceLighting','phong',...
    'AmbientStrength',.3,'DiffuseStrength',.8,...
    'SpecularStrength',.9,'SpecularExponent',25,...
    'BackFaceLighting','unlit')
```



After obtaining the surface object's handle using `findobj`, you can set properties that affect how the light reflects from the surface. See for more detailed descriptions of these properties.

Selecting a Lighting Method

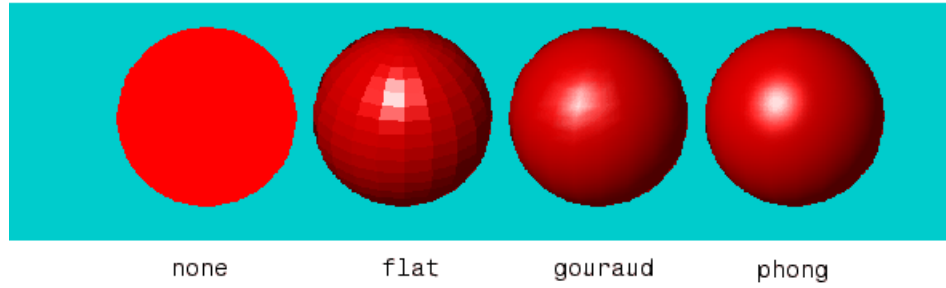
Face and Edge Lighting Methods

When you add lights to an axes, MATLAB determines the effects these lights have on the patch and surface objects that are displayed in that axes. There are different methods used to calculate the face and edge coloring of lit objects, and the one you select depends on the results you want to obtain.

MATLAB supports three different algorithms for lighting calculations, selected by setting the `FaceLighting` and `EdgeLighting` properties of each patch and surface object in the scene. Each algorithm produces somewhat different results:

- Flat lighting — Produces uniform color across each of the faces of the object. Select this method to view faceted objects.
- Gouraud lighting — Calculates the colors at the vertices and then interpolates colors across the faces. Select this method to view curved surfaces.
- Phong lighting — Interpolates the vertex normals across each face and calculates the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

This illustration shows how a red sphere looks using each of the lighting methods with one white light source.



The `lighting` command (as opposed to the `light` function) provides a convenient way to set the lighting method.

Reflectance Characteristics of Graphics Objects

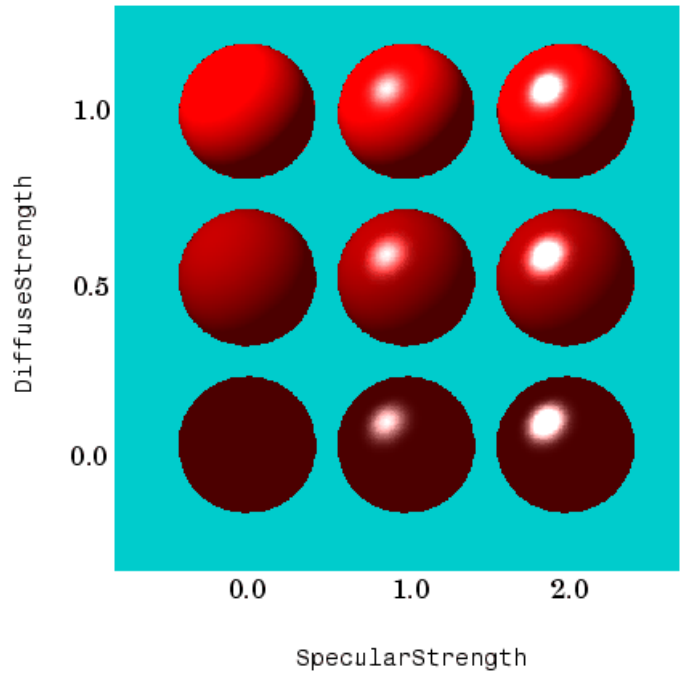
In this section...

Specular and Diffuse Reflection

You can specify the reflectance characteristics of patch and surface objects and thereby affect the way they look when lights are applied to the scene. It is likely you will adjust these characteristics in combination to produce particular results.

Also see the `material` command for a convenient way to produce certain lighting effects.

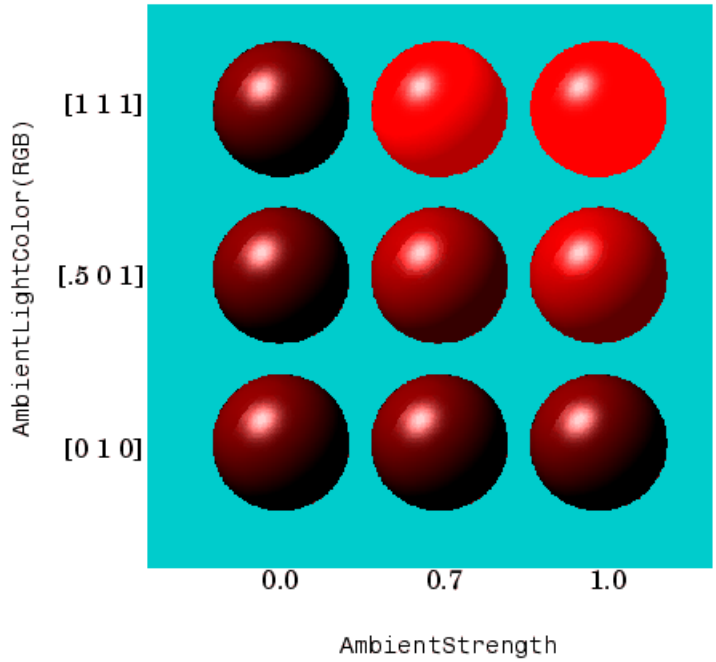
You can control the amount of specular and diffuse reflection from the surface of an object by setting the `SpecularStrength` and `DiffuseStrength` properties. This picture illustrates various settings.



Ambient Light

Ambient light is a directionless light that shines uniformly on all objects in the scene. Ambient light is visible only when there are light objects in the axes. There are two properties that control ambient light — `AmbientLightColor` is an axes property that sets the color, and `AmbientStrength` is a property of patch and surface objects that determines the intensity of the ambient light on the particular object.

This illustration shows three different ambient light colors at various intensities. The sphere is red and there is a white light object present.

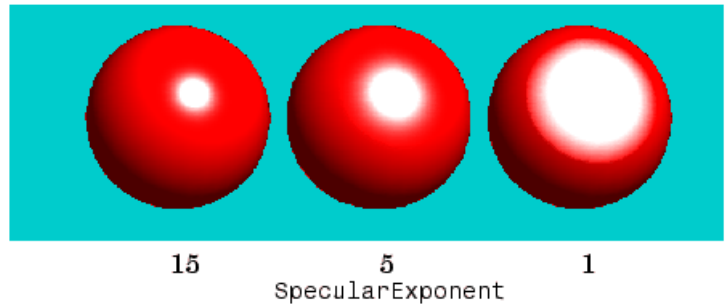


The green [0 1 0] ambient light does not affect the scene because there is no red component in green light. However, the color defined by the RGB values [.5 0 1] does have a red component, so it contributes to the light on the sphere (but less than the white [1 1 1] ambient light).

Specular Exponent

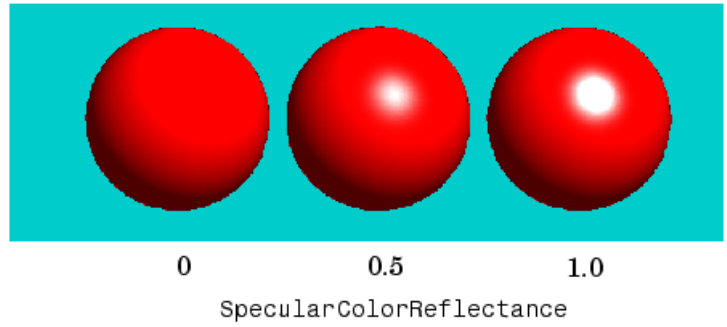
The size of the specular highlight spot depends on the value of the patch and surface object's `SpecularExponent` property. Typical values for this property range from 1 to 500, with normal objects having values in the range 5 to 20.

This illustration shows a red sphere illuminated by a white light with three different values for the `SpecularExponent` property.



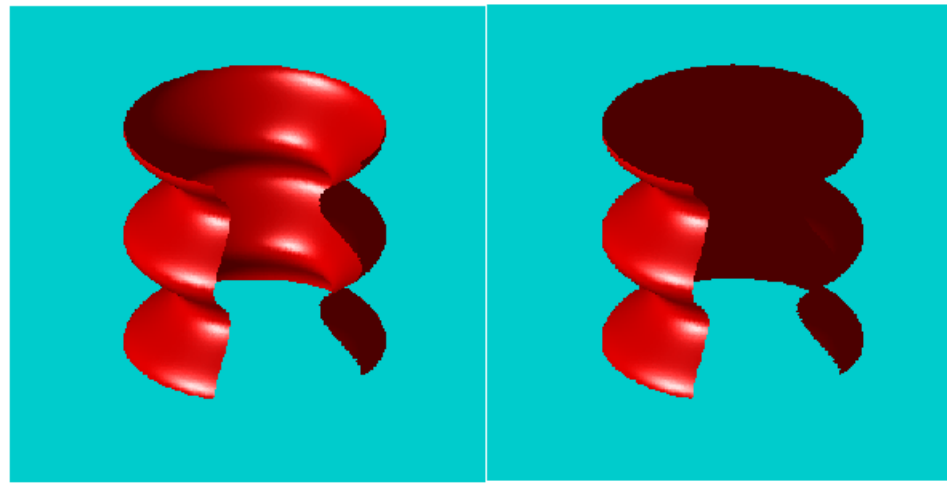
Specular Color Reflectance

The color of the specularly reflected light can range from a combination of the color of the object and the color of the light source to the color of the light source only. The patch and surface `SpecularColorReflectance` property controls this color. This illustration shows a red sphere illuminated by a white light. The values of the `SpecularColorReflectance` property range from 0 (object and light color) to 1 (light color).



Back Face Lighting

Back face lighting is useful for showing the difference between internal and external faces. These pictures of cut-away cylindrical surfaces illustrate the effects of back face lighting.

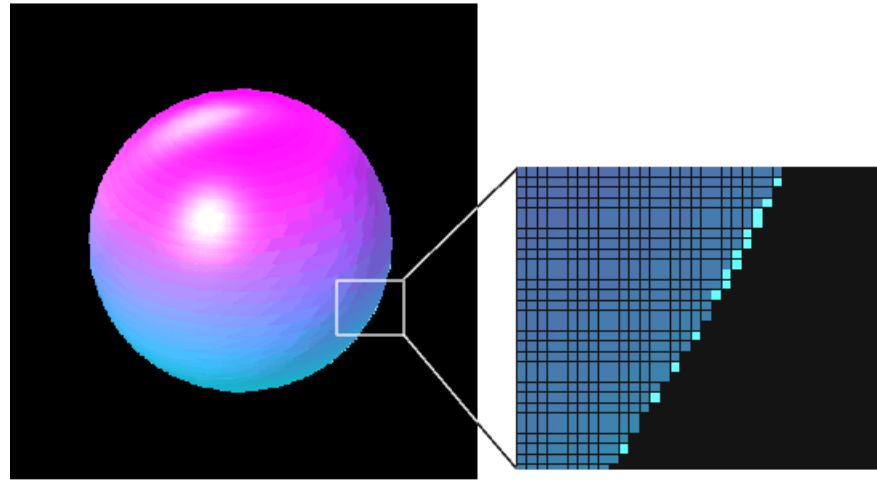


`BackFaceLighting = reverselit` `BackFaceLighting = unlit`

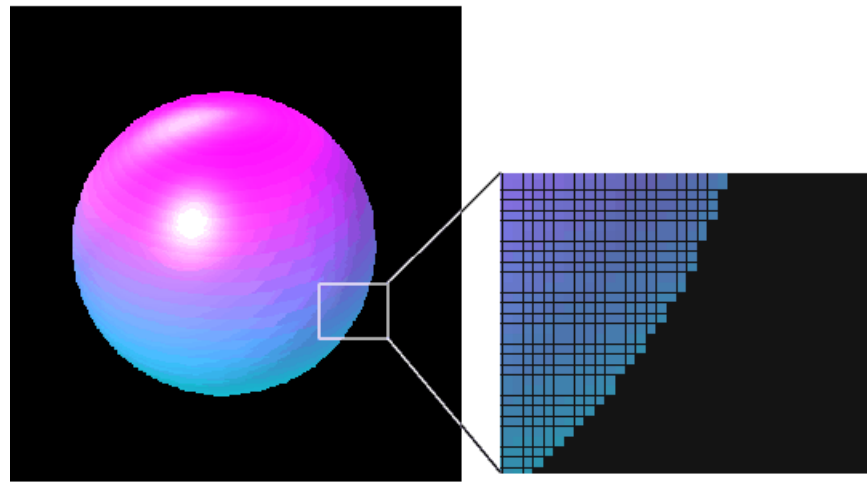
The default value for `BackFaceLighting` is `reverselit`. This setting reverses the direction of the vertex normals that face away from the camera, causing the interior surface to reflect light towards the camera. Setting `BackFaceLighting` to `unlit` disables lighting on faces with normals that point away from the camera.

You can also use `BackFaceLighting` to remove edge effects for closed objects. These effects occur when `BackFaceLighting` is set to `reverselit` and pixels along the edge of a closed object are lit as if their vertex normals faced the camera. This produces an improperly lit pixel because the pixel is visible but is really facing away from the camera.

To illustrate this effect, the next picture shows a blowup of the edge of a lit sphere. Setting `BackFaceLighting` to `lit` prevents the improper lighting of pixels.



BackFaceLighting = reverselit



BackFaceLighting = lit

Positioning Lights in Data Space

This example creates a sphere and a cube to illustrate the effects of various properties on lighting. The variables `vert` and `fac` define the cube using the `patch` function.

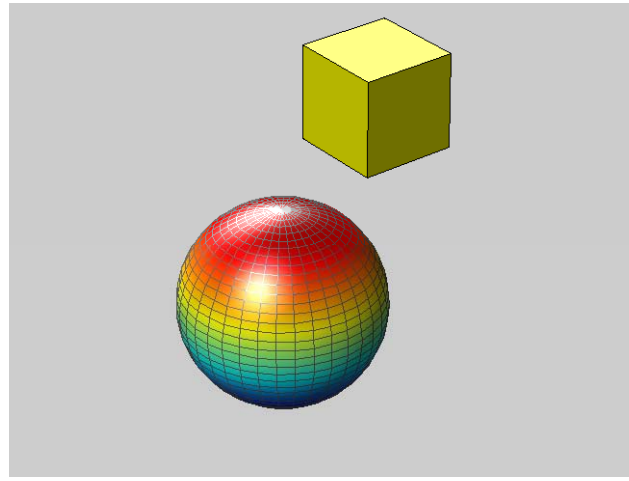
```
vert =                          fac =

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

```
sphere(36);
h = findobj('Type','surface');
set(h,'FaceLighting','phong',...
      'FaceColor','interp',...
      'EdgeColor',[.4 .4 .4],...
      'BackFaceLighting','lit')
hold on
patch('faces',fac,'vertices',vert,'FaceColor','y');
light('Position',[1 3 2]);
light('Position',[-3 -1 3]);
material shiny
axis vis3d off
hold off
```

All faces of the cube have `FaceColor` set to yellow. The `sphere` function creates a spherical surface and the handle of this surface is obtained using `findobj` to search for the object whose `Type` property is `surface`. The `light` functions define two white (the default color) light objects located at infinity in the direction specified by the `Position` vectors. These vectors are defined in axes coordinates [*x*, *y*, *z*].

The patch uses `flat` `FaceLighting` (the default) to enhance the visibility of each side. The surface uses `phong` `FaceLighting` because it produces the smoothest interpolation of lighting effects. The `material shiny` command affects the reflectance properties of both the cube and sphere (although its effects are noticeable only on the sphere because of the cube's flat shading).

Because the sphere is closed, the `BackFaceLighting` property is changed from its default setting, which reverses the direction of vertex normals that face away from the camera, to normal lighting, which removes undesirable edge effects.



Examining the code in the `lighting` and `material` M-files can help you understand how various properties affect lighting.

4

Transparency

Making Objects Transparent

In this section...

About Transparency

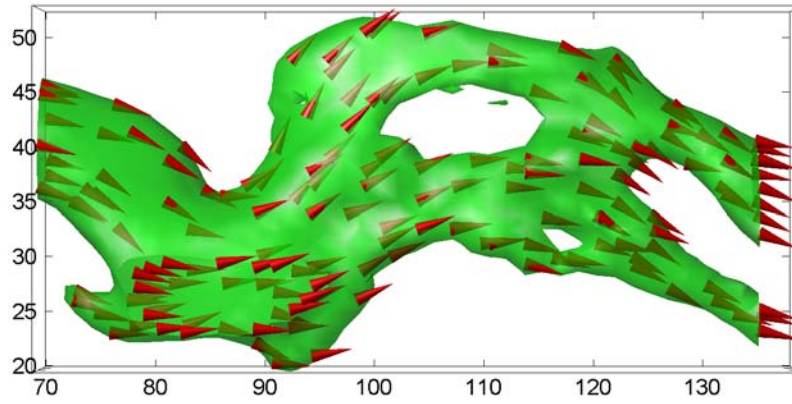
Making graphics objects semitransparent is a useful technique in 3-D visualization to make it possible to see an object, while at the same time, see what information the object would obscure if it was completely opaque. You can also use transparency as another dimension for displaying data, much the way color is used in MATLAB.

The transparency of a graphics object determines the degree to which you can see through the object. You can specify a continuous range of transparency varying from completely transparent (i.e., invisible) to completely opaque (i.e., no transparency).

Objects that support transparency are

- Image
- Patch
- Surface

The following picture illustrates the effect of transparency. The green isosurface (patch object) reveals the cone plot that lies in the interior.



Note You must have OpenGL available on your system to use transparency. When rendering transparency MATLAB automatically uses OpenGL if it is available. If it is not available, transparency does not display. See the figure property `RendererMode` for more information.

Specifying Transparency

Transparency values, which range from [0 1], are referred to as *alpha* values. An alpha value of 0 means completely transparent (i.e., invisible); an alpha value of 1 means completely opaque (i.e., no transparency).

MATLAB treats transparency in a way that is analogous to how it treats color for the respective objects:

- Patches and surfaces can define a single face and edge alpha value or use flat or interpolated transparency based on values in the figure's alphamap.
- Images, patches, and surfaces can define alpha data that is used as indices into the alphamap or directly as alpha values.
- Axes define alpha limits that control the mapping of object data to alpha values.
- Figures contain alphamaps, which are m-by-1 arrays of alpha values.

See the following sections for more information on color:

- "Specifying Patch Coloring" on page 5-14 in Creating 3-D Models with Patches in the Using MATLAB Graphics documentation
- "Coloring Mesh and Surface Plots" on page 1-16 in Creating 3-D Graphs in the Using MATLAB Graphics documentation

Transparency Properties

The following table summarizes the object properties that control transparency.

Property	Purpose
`AlphaData`	Transparency data for image and surface objects
`AlphaDataMapping`	Transparency data mapping method
`FaceAlpha`	Transparency of the faces (patch and surface only)
`EdgeAlpha`	Transparency of the edges (patch and surface only)
`FaceVertexAlphaData`	Patch only alpha data property
`ALim`	Alpha axis limits
`ALimMode`	Alpha axis limits mode
`Alphamap`	Figure alphamap

Transparency Functions

There are three functions that simplify the process of setting alpha properties.

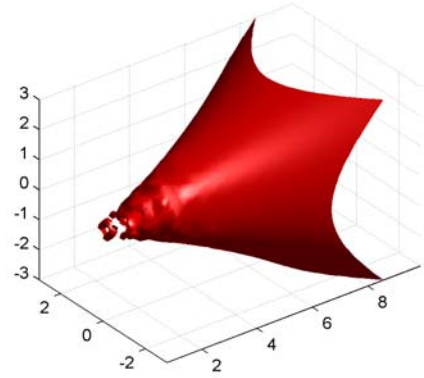
Function	Purpose
`alpha`	Set or query transparency properties for objects in current axes
`alphamap`	Specify the figure alphamap
`alim`	Set or query the axes alpha limits

Example — A Transparent Isosurface

Specifying a single transparency value for graphics objects is useful when you want to reveal structure that is obscured with opaque objects. For patches and surfaces, use the `FaceAlpha` and `EdgeAlpha` properties to specify the transparency of faces and edges. The following example illustrates this.

This example uses the `flow` function to generate data for the speed profile of a submerged jet within an infinite tank. One way to visualize this data is by creating an isosurface illustrating where the rate of flow is equal to a specified value.

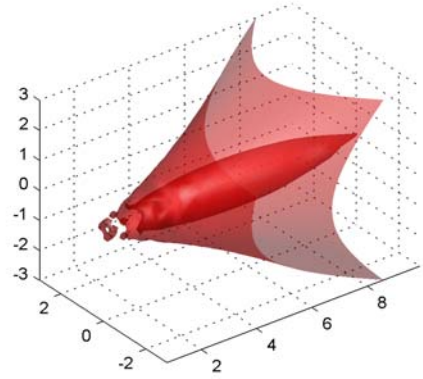
```
[x y z v] = flow;
p = patch(isosurface(x,y,z,v,-3));
isonormals(x,y,z,v,p);
set(p,'facecolor','red','edgecolor','none');
daspect([1 1 1]);
view(3); axis tight; grid on;
camlight; lighting gouraud;
```



Adding transparency to the isosurface reveals that there is greater complexity in the fluid flow than is visible using the opaque surface. The statement

```
alpha(.5)
```

sets the `FaceAlpha` value for the isosurface face to .5.



Setting a Single Transparency Value for Images

For images, the statement

```
alpha(.5)
```

sets `AlphaData` to .5. When the `AlphaDataMapping` property is set to `none` (the default), setting `AlphaData` on an image causes the entire image to be rendered with the specified alpha value.

Mapping Data to Transparency — Alpha Data

In this section...

What Is Alpha Data?

Alpha data is analogous to color data (e.g., the `CData` property of surfaces). When you create a surface, MATLAB maps each element in the color data array to a color in the colormap. Similarly, each element in the alpha data maps to a transparency value in the alphamap.

Specify surface and image alpha data with the `AlphaData` property. For patch objects, use the `FaceVertexAlphaData` property.

You can control how MATLAB interprets alpha data with the following properties:

- `FaceAlpha` and `EdgeAlpha` — Enable you to select flat or interpolated transparency rendering. If set to a single transparency value, MATLAB applies this value to all faces or edges and does not use the alpha data.
- `AlphaDataMapping` and `ALim` — Determine how MATLAB maps the alpha data to the alphamap. By default, MATLAB scales the alpha data to be within the range [0 1].
- `Alphamap` — Contains the actual transparency values to which the data is to be mapped.

Note that there are differences between the default values of equivalent color and alpha properties because, in contrast to color, transparency is not displayed by default. The following table highlights these differences.

Color Property	Default	Alpha Property	Default
`FaceColor`	Flat	`FaceAlpha`	1 (opaque)
`CData`	Equal to `ZData`	`AlphaData` and `FaceVertexAlphaData`	1 (scalar)

By default, objects have single-valued alpha data. Therefore you cannot specify `flat` or `interp` `FaceAlpha` or `EdgeAlpha` without first setting `AlphaData` to an array of the appropriate size.

The sections that follow illustrate how to use these properties to display object data as degrees of transparency.

Size of the Alpha Data Array

In order to use nonscalar alpha data, you need to specify the alpha data as an array equal in size to

- `CData` of images and surfaces
- The number of faces (flat) or the number of vertices (interpolated) defined in the `FaceVertexAlphaData` property of patches

Once you have specified an alpha data array of the proper size, you can select the face and edge rendering you want to use. Flat uses one transparency value per face, while interpolated performs bilinear interpolation of the values at each vertex.

Mapping Alpha Data to the Alphamap

You can control how MATLAB maps the alpha data to the alphamap using the `AlphaDataMapping` property. There are three possible mappings:

- `none` — Interpret the values in alpha data as transparency values (data values must be between 0 and 1, or will be clamped to 0 or 1). This is the default mapping.

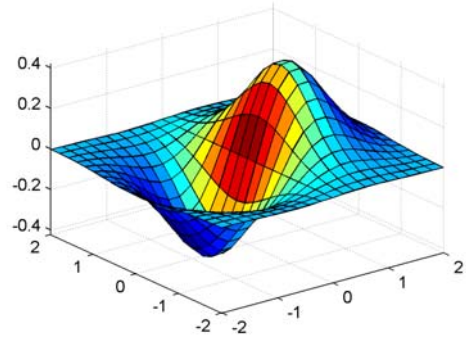
- `scaled` — Transform the alpha data to span the portion of the alphamap indicated by the axes `ALim` property, linearly mapping data values to alpha values. This is the same way color data is mapped to the colormap.
- `direct` — Use the alpha data directly as indices into the figure alphamap.

By default, objects have scalar alpha data (`AlphaData` and `FaceVertexAlphaData`) set to the value 1.

Example — Mapping Data to Color or Transparency

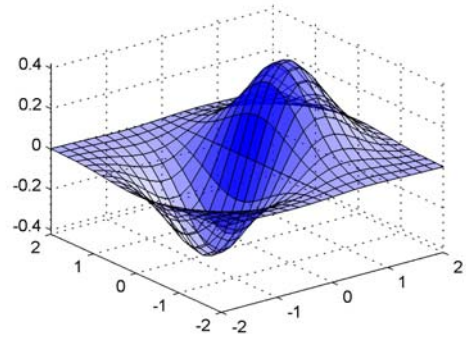
This example displays a surface plot of a function of two variables. The color is mapped to the gradient of the `z` data.

```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2-y.^2);
surf(x,y,z,gradient(z)); axis tight
```



You can map transparency to the gradient of `z` in a similar way.

```
surf(x,y,z,'FaceAlpha','flat',...
 'AlphaDataMapping','scaled',...
 'AlphaData',gradient(z),...
 'FaceColor','blue');
axis tight
```


0.4
0.2
0
-0.2
-0.4
2
1
0
-1
-2
-2
-1
0
1
2

Selecting an Alphamap

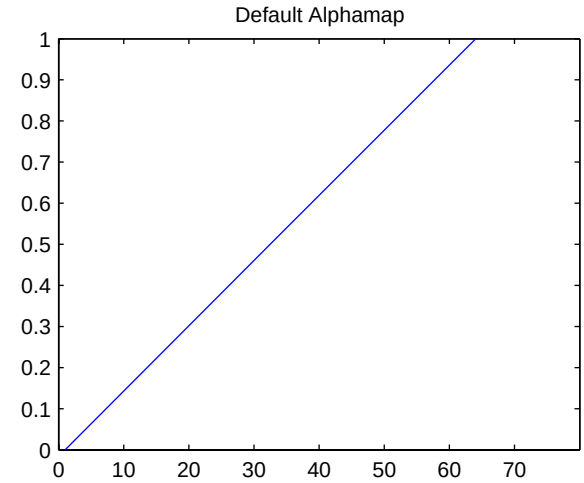
In this section...

What Is an Alphamap?

An alphamap is simply an array of values ranging from 0 to 1. The size of the array can be either m-by-1 or 1-by-m.

The default alphamap contains 64 values ranging linearly from 0 to 1, as you can see in the following plot.

```
plot(get(gcf,'Alphamap'))
```

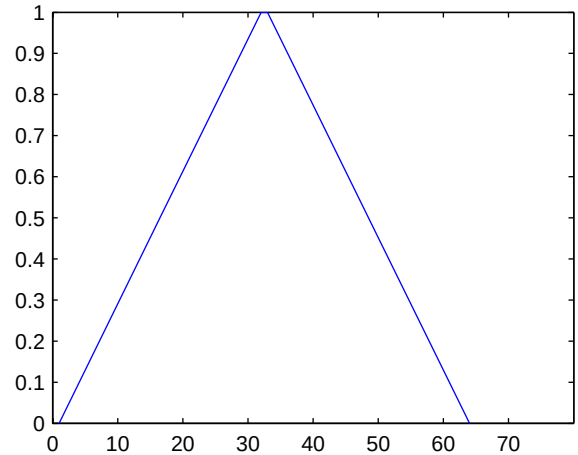


This alphamap displays the lowest alpha data values as completely transparent and the highest alpha data values as opaque.

The `alphamap` function creates some useful predefined alphamaps and also enables you to modify existing maps. For example,

```
plot(alphamap('vup'))
```

produces the following alphamap.

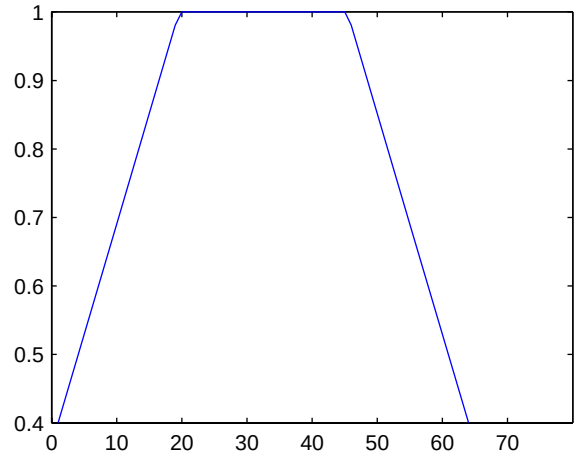


You can shift the values using the `increase` or `decrease` options. For example,

```
alphamap('increase',.4)
```

adds the value .4 to all values in the current figure's alphamap. Replotting the `'vup'` alphamap illustrates the change. Note how the values are clamped to the range [0 1].

```
plot(get(gcf,'Alphamap'))
```



Example — Modifying the Alphamap

This example uses slice planes to examine volume data. The slice planes use the color data for alpha data and employ a rampdown alphamap (the values range from 1 to 0):

1. Create the volume data by evaluating a function of three variables.

```
[x,y,z] = meshgrid(-1.25:.1:-.25,-2:.2:2,-2:.1:2);
v = x.*exp(-x.^2-y.^2-z.^2);
```

2. Create the slice planes, set the alpha data equal to the color data, and specify interpolated `FaceAlpha`.

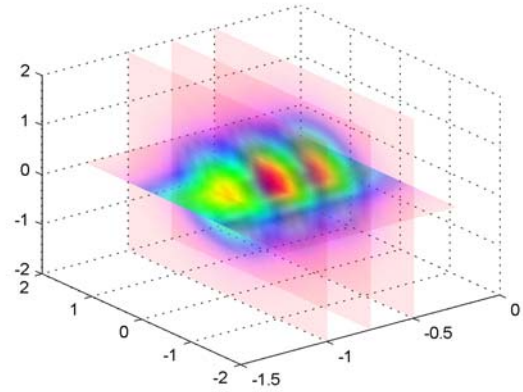
```
h = slice(x,y,z,v,[-1 -.75 -.5],[],[0]);
alpha('color')
set(h,'EdgeColor','none','FaceColor','interp',...
 'FaceAlpha','interp')
```

3. Install the rampdown alphamap and increase each value in the alphamap by .1 to achieve the desired degree of transparency. Specify the `hsv` colormap.

```
alphamap('rampdown')
```

```
alphamap('increase',.1)
colormap(hsv)
```

This alphamap causes the smallest values of the function (around zero) to be displayed with the least transparency and the greatest values to display with the most transparency. This enables you to see through the slice planes, while at the same time preserving the data around zero.



5

Creating 3-D Models with Patches

Introduction to Patch Objects

In this section...

What Are Patch Objects?

A patch graphics object is composed of one or more polygons that may or may not be connected. Patches are useful for modeling real-world objects such as airplanes or automobiles, and for drawing 2- or 3-D polygons of arbitrary shape.

In contrast, surface objects are rectangular grids of quadrilaterals and are better suited for displaying planar topographies such as the values of mathematical functions of two variables, the contours of data in a rectangular plane, or parameterized surfaces such as spheres.

A number of MATLAB functions create patch objects — `fill`, `fill3`, `isosurface`, `isocaps`, some of the `contour` functions, and `patch`. This section concentrates on use of the `patch` function.

You define a patch by specifying the coordinates of its vertices and some form of color data. Patches support a variety of coloring options that are useful for visualizing data superimposed on geometric shapes.

There are two ways to specify a patch:

- By specifying the coordinates of the vertices of each polygon, which MATLAB connects to form the patch
- By specifying the coordinates of each *unique* vertex and a matrix that specifies how to connect these vertices to form the faces

The second technique is preferred for multifaceted patches because it generally requires less data to define the patch; vertices shared by more than

one face need be defined only once. This section provides examples of both techniques.

Behavior of the patch Function

There are two forms of the `patch` function -- high-level syntax and low-level syntax. The behavior of the `patch` function differs somewhat depending on which syntax you use.

High-Level Syntax

When you use the high-level syntax, MATLAB automatically determines how to color each face based on the color data you specify. The high-level syntax enables you to omit the property names for the *x*-, *y*-, and *z*-coordinates and the color data, as long as you specify these arguments in the correct order.

```
patch(x-coordinates,y-coordinates,z-coordinates,colordata)
```

However, you must specify color data so MATLAB can determine what type of coloring to use. If you do not specify color data, MATLAB returns an error.

```
patch(sin(t),cos(t))
??? Error using ==> patch
Not enough input arguments.
```

Low-Level Syntax

The low-level syntax accepts only property name/property value pairs as arguments and does not automatically color the faces unless you also change the value of the `FaceColor` property. For example, the statement

```
patch('XData',sin(t),'YData',cos(t)) % Low-level syntax
```

draws a patch with white face color because the factory default value for the `FaceColor` property is the color white.

```
get(0,'FactoryPatchFaceColor')
ans =
      1     1     1
```

See the list of patch properties in the MATLAB Function Reference and the `get` command for information on how to obtain the factory and user default values for properties.

Interpreting the Color Argument

When you use the low-level syntax, MATLAB interprets the third (or fourth if there are *z*-coordinates) argument as color data. If you intend to define a patch with *x*-, *y*-, and *z*-coordinates, but leave out the color, MATLAB interprets the *z*-coordinates as color data, and then draws a 2-D patch. For example,

```
h = patch(sin(t),cos(t),1:length(t))
```

draws a patch with all vertices at z = 0, colored by interpolating the vertex colors (since there is one color for each vertex), whereas

```
h = patch(sin(t),cos(t),1:length(t),'y')
```

draws a patch with vertices at increasing values of *z*, colored yellow.

"Specifying Patch Coloring" on page 5-14 provides more information on options for coloring patches.

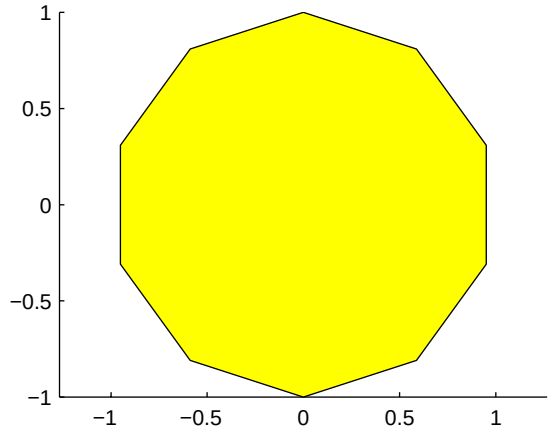
Creating a Single Polygon

A polygon is simply a patch with one face. To create a polygon, specify the coordinates of the vertices and color data with a statement of the form

```
patch(x-coordinates,y-coordinates,[z-coordinates],colordata)
```

For example, these statements display a 10-sided polygon with a yellow face enclosed by a black edge. The `axis equal` command produces a correctly proportioned polygon.

```
t = 0:pi/5:2*pi;
patch(sin(t),cos(t),'y')
axis equal
```

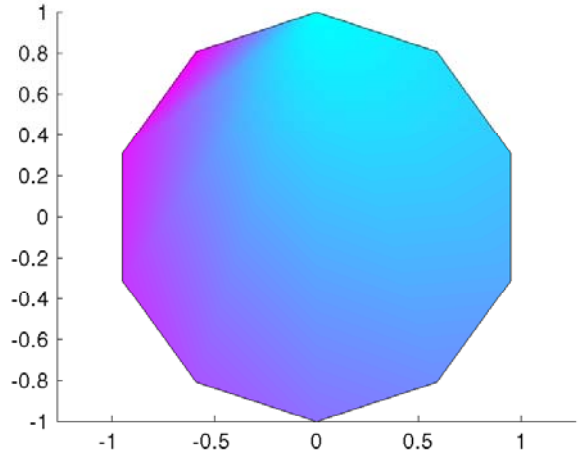


The first and last vertices need not coincide; MATLAB automatically closes each polygonal face of the patch. In fact, it is generally better to define each vertex only once, particularly if you are using interpolated face coloring.

Interpolated Face Colors

You can control many aspects of the patch coloring. For example, instead of specifying a single color, you can provide a range of numerical values that map the color at each vertex to a color in the figure colormap.

```
a = t(1:length(t)-1); %remove redundant vertex definition
patch(sin(a),cos(a),1:length(a),'FaceColor','interp')
colormap cool;
axis equal
```



MATLAB now interpolates the colors across the face of the patch. You can color the edges of the patch the same way, by setting the edge colors to be interpolated. The command is

```
patch(sin(t),cos(t),1:length(t),'EdgeColor','interp')
```

“Specifying Patch Coloring” on page 5-14 provides more information on options for coloring patches.

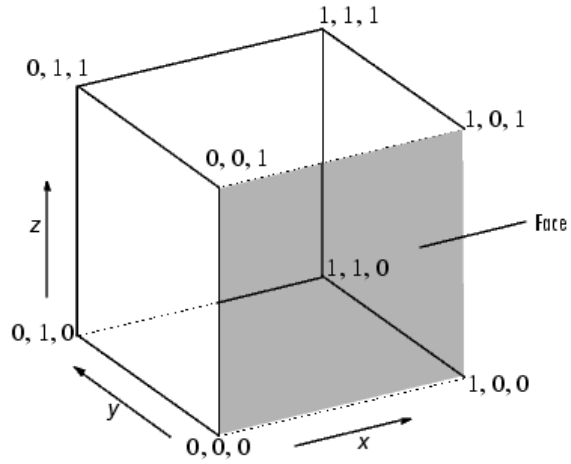
Multifaceted Patches

Example — Defining a Cube

A cube is defined by eight vertices that form six sides. This illustration shows the *x*-, *y*-, and *z*-coordinates of the vertices defining a cube in which the sides are one unit in length.

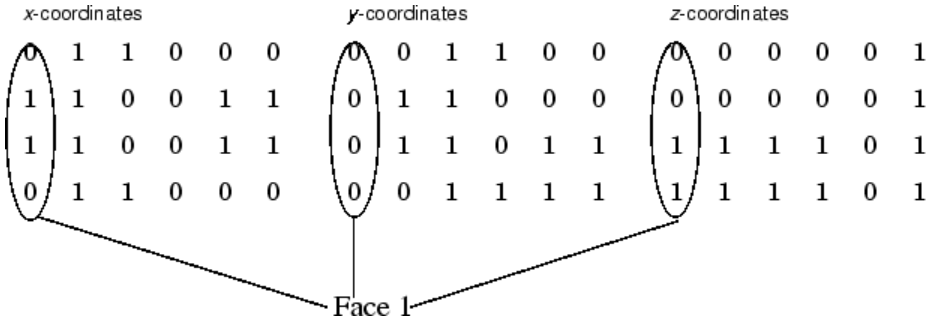
If you specify the *x*-, *y*-, and *z*-coordinate arguments as vectors, MATLAB draws a single polygon by connecting the points. If the arguments are matrices, MATLAB draws one polygon per column, producing a single patch with multiple faces. These faces need not be connected and can be self-intersecting.

Alternatively, you can specify the coordinates of each unique vertex and the order in which to connect them to form the faces. The examples in this section illustrate both techniques.



Specifying X, Y, and Z Coordinates

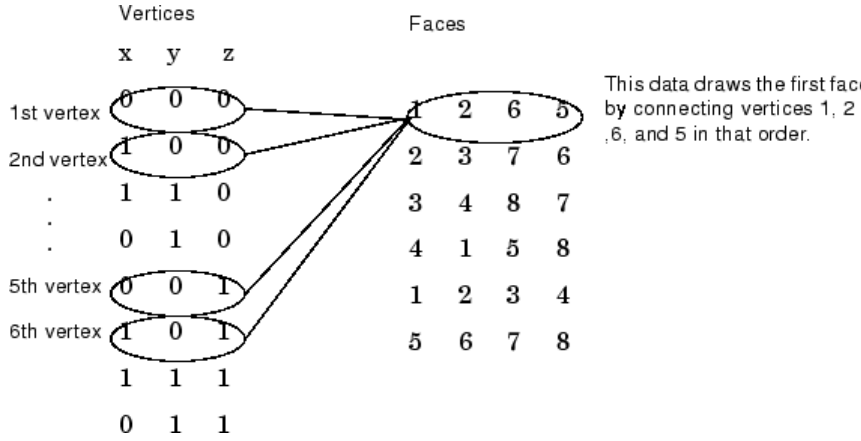
Each of the six faces has four vertices. Because you do not need to close each polygon (i.e., the first and last vertices do not need to be the same), you can define this cube using a 4-by-6 matrix for each of the *x*-, *y*-, and *z*-coordinates.



Each column of the matrices specifies a different face. Note that while there are only eight vertices, you must specify 24 vertices to define all six faces. Since each face shares vertices with four other faces, you can define the patch more efficiently by defining each vertex only once and then specifying the order in which to connect these vertices to form each face. The patch `Vertices` and `Faces` properties define patches in just this way.

Specifying Faces and Vertices

These matrices specify the cube using `Vertices` and `Faces`.



Using the vertices/faces technique can save a considerable amount of computer memory when patches contain a large number of faces. This technique requires the formal `patch` function syntax, which entails assigning values to the `Vertices` and `Faces` properties explicitly. For example,

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix)
```

Because the high-level syntax does not automatically assign face or edge colors, you must set the appropriate properties to produce patches with colors other than the default white face color and black edge color.

Flat Face Color

Flat face color is the result of specifying one color per face. For example, using the vertices/faces technique and the `FaceVertexCData` property to define color, this statement specifies one color per face and sets the `FaceColor` property to `flat`.

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix,...
      'FaceVertexCData',hsv(6),'FaceColor','flat')
```

Because truecolor specified with the `FaceVertexCData` property has the same format as a MATLAB colormap (i.e., an *n*-by-3 array of RGB values), this example uses the `hsv` colormap to generate the six colors required for flat shading.

Interpolated Face Color

Interpolated face color means the vertex colors of each face define a transition of color from one vertex to the next. To interpolate the colors between vertices, you must specify a color for each vertex and set the `FaceColor` property to `interp`.

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix,...
      'FaceVertexCData',hsv(8),'FaceColor','interp')
```

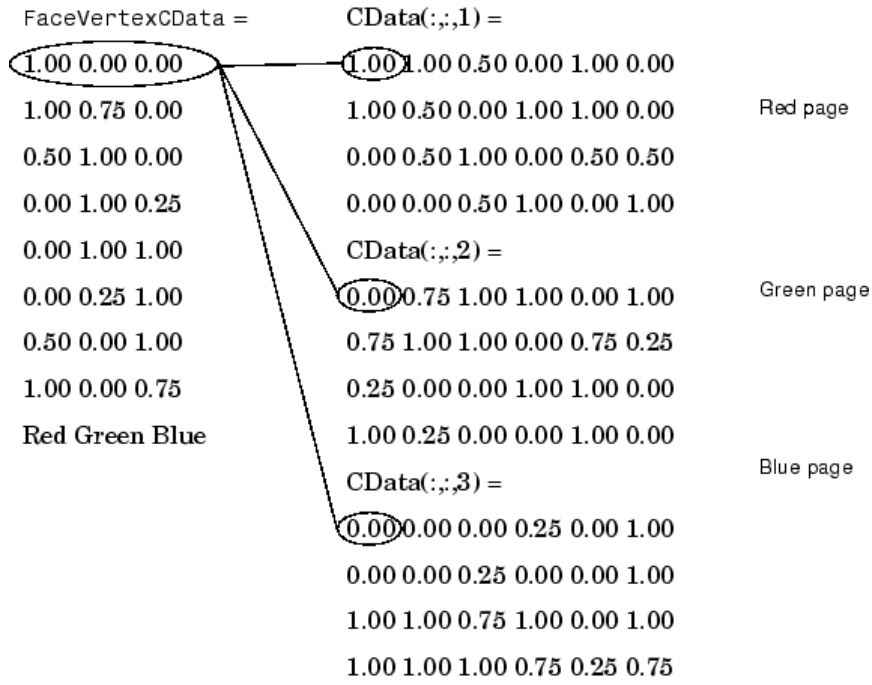
Changing to the standard 3-D view and making the axis square,

```
view(3); axis square
```

produces a cube with each face colored by interpolating the vertex colors.

To specify the same coloring using the x, y, z, c technique, c must be an *m*-by-*n*-by-3 array, where the dimensions of x, y, and z are *m*-by-*n*.

This diagram shows the correspondence between the `FaceVertexCData` and `CData` properties.



"Specifying Patch Coloring" on page 5-14 discusses coloring techniques in more detail.

Modifying Data on Existing Patch Objects

In this section...

Specifying Patch Data

In general, if you define a patch with `Faces` and `Vertices` data and then want to modify its data, you should continue to use these same properties. Do not switch modes and modify the `XData`, `YData`, `ZData`, or `CData` properties.

Handling Mixed Data Specification

When you create a patch specified with `Faces` and `Vertices` data, MATLAB constructs arrays of data for the `XData`, `YData`, `ZData` and `CData` properties when you query them. However, these arrays contain only enough data to define the same number of vertices as there are referred to in the `Faces` property. If the number of vertices in the `Vertices` property is greater than the number of vertices used by the `Faces` property, then MATLAB cannot generate complete x, y, and z data from the faces and vertex data.

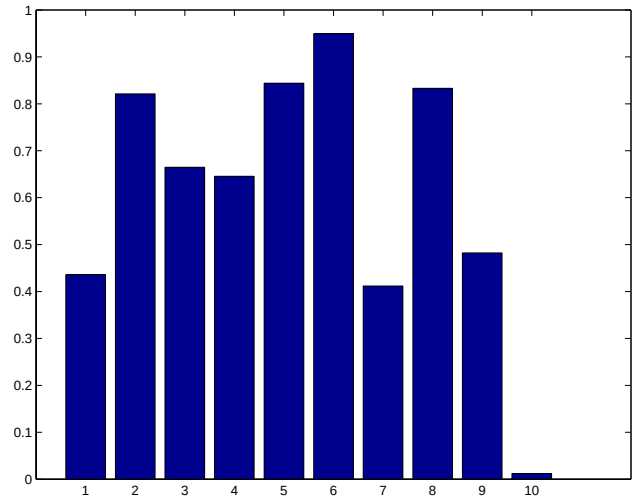
While you should not use mixed data specification when defining patch objects directly, you might need to modify patch data when using functions that themselves create patch objects. For example, the `bar` function creates patch objects to implement the bars in a graph.

Note The barseries `YData` property enables you to modify the bar graph without the need to use the following steps. See the `bar` function for more information on working with bar graphs.

The function uses y-data values to determine the height of each bar, but creates each bar as the face of a patch specified by faces and vertices. For example,

```
rand('state',4)
```

```
h = bar(rand(10,1));   % y data for each bar
p = get(h,'children'); % get the handle of the patch
cl = get(gca,'CLim');
```



Before you can change the patch `YData` property, you must switch the patch to x, y, and z data as follows:

```
xd = get(p,'XData');
yd = get(p,'YData');
zd = get(p,'ZData');
cd = get(p,'CData');
set(p,'XData',xd,'YData',yd,'ZData',zd,'CData',cd);
set(gca,'CLim',cl)
```

This setting of the `XData`, `YData`, `ZData` and `CData` properties causes the `patch` function to match the faces and vertex data with x, y, and z data. Note that because there is a change in the patch data, the color limits change, so you must use the original values for the axes `CLim` property.

You can now modify the y data values to change your graph. For example, the value of bar at x = 10 is 0.0122:

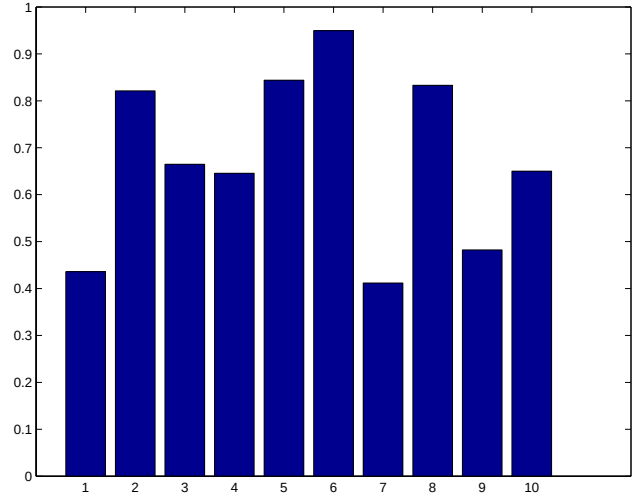
```
yd(:,10)
ans =
         0
    0.0122
    0.0122
         0
```

You can change this bar by changes rows 2 and 3:

```
yd(2:3,10) = [.65 .65];
```

Now reset the patch `YData` property:

```
set(p,'YData',yd)
```



Another reason you might want to modify face-vertex data for bar graphs or other objects is to modify their `CData` to customize how they are colored. Techniques for doing this for 2–D and 3–D bar graphs are explained in "Coloring 2-D Bars According to Height" and "Coloring 3-D Bars According to Height" in the MATLAB Graphics documentation.

Specifying Patch Coloring

In this section...

Patch Color Properties

Patch coloring is defined differently from surface object coloring in that patches do not automatically generate color data based on the value of the *z*-coordinate at each vertex. You must explicitly specify patch coloring, or MATLAB uses the default white face color and black edge color.

You can specify patch face coloring by defining

- A single color for all faces
- One color for each face, which is used for flat coloring
- One color for each vertex, which is used for interpolated coloring

Specify the face color using either the `CData` property, if you are using *x*-, *y*-, and *z*-coordinates, or the `FaceVertexCData` property, if you are specifying vertices and faces.

This table summarizes the patch properties that control color (exclusive of those used when light sources are present).

Property	Purpose
`CData`	Specify single, per face, or per vertex colors in conjunction with x, y, and z data
`CDataMapping`	Specifies whether color data is scaled or used directly as indices into the figure colormap
`FaceVertexCData`	Specify single, per face, or per vertex colors in conjunction with faces and vertices data

Property	Purpose
`EdgeColor`	Specifies whether edges are invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors
`FaceColor`	Specifies whether faces are invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors
`MarkerEdgeColor`	Specifies the color of the marker, or the edge color for filled markers
`MarkerFaceColor`	Specifies the fill color for markers that are closed shapes

Patch Edge Coloring

Each patch face has a bounding edge, which you can color as

- A single color for all edges
- A flat color defined by the color of the vertex that precedes the edge
- Interpolated colors determined by the two vertices that bound the edge

Note that patch edge colors can be flat or interpolated only when you specify a color for each vertex. For flat edge coloring, MATLAB uses the color of the vertex preceding the edge to determine the color of the edge. The order in which you specify the vertices establishes which vertex colors a particular edge.

The following examples illustrate patch edge coloring:

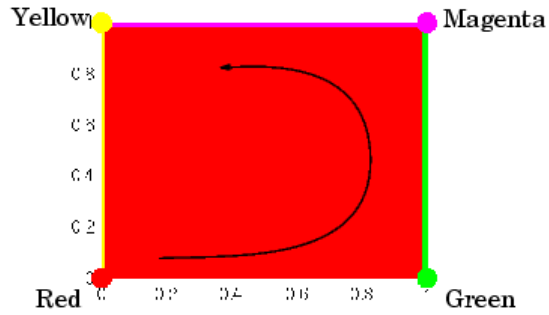
-
- “Coloring Edges with Shared Vertices” on page 5-17

Example — Specifying Flat Edge and Face Coloring

These statements create a square patch.

```
v  = [0 0 0;1 0 0;1 1 0;0 1 0];
```

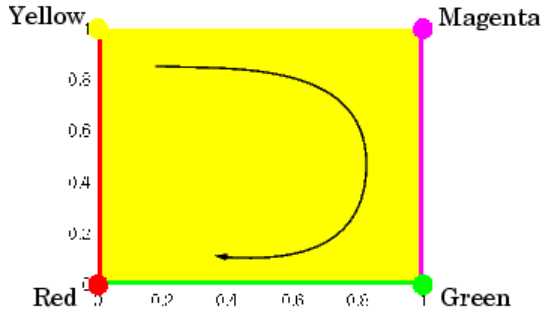
```
f = [1 2 3 4];
fvc = [1 0 0;0 1 0;1 0 1;1 1 0];
patch('Vertices',v,'Faces',f,'FaceVertexCData',fvc,...
      'FaceColor','flat','EdgeColor','flat',...
      'Marker','o','MarkerFaceColor','flat')
```



The `Faces` property value, `[1 2 3 4]`, determines the order in which MATLAB connects the vertices. In this case, the order is red, green, magenta, and yellow. If you change this order, the results can be quite different. For example, specifying the `Faces` property as

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

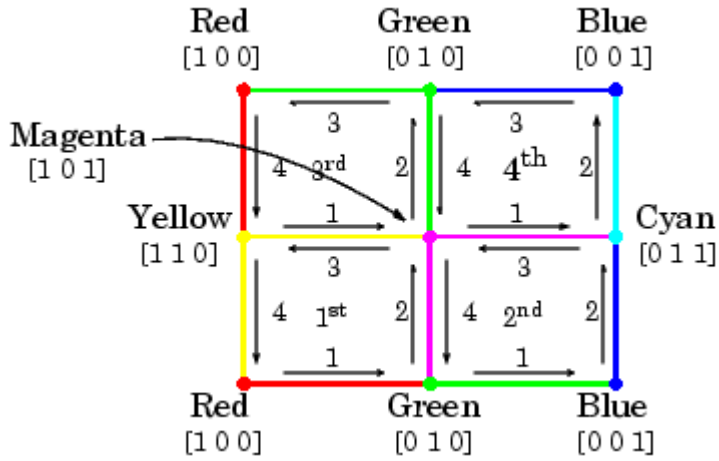
changes the order to yellow, magenta, green, and red. Note that changing the order not only changes the color of the edges, but also the color of the face, which is the color of the first vertex specified.



Coloring Edges with Shared Vertices

Each patch face is bound by edges, which are line segments that connect the vertices. When patches have multiple faces that share vertices, some of the edges might overlap. In such cases, the edges of the most recently drawn face overlie previously drawn edges.

For example, this illustration shows a patch with four faces and flat colored edges (`FaceColor` set to `none`, `EdgeColor` set to `flat`).



The arrows indicate the order in which each edge is drawn in the first, second, third, and fourth face. The color at each vertex determines the color of the edge that follows it. Notice how the second edge in the first face would be green except that the second face drew its fourth edge from the magenta vertex. You can see similar effects in all shared edges.

For `EdgeColor` set to `interp`, MATLAB interpolates colors between adjacent vertices. In this case, the order in which you specify the vertices does not affect the edge color.

Interpreting Indexed and Truecolor Data

In this section...

Introduction

MATLAB interprets the patch color data in either of two ways:

- Indexed Color Data — Numerical values that are mapped to colors defined in the figure colormap
- Truecolor Data — RGB triples that define colors explicitly and do not make use of the figure colormap

The dimensions of the color data (`CData` or `FaceVertexCData`) determine how MATLAB interprets it. If you specify only one numeric value per patch, per face, or per vertex, then MATLAB interprets the data as indexed. If there are three numeric values per patch, face, or vertex, then MATLAB interprets the data as RGB values.

Indexed Color Data

MATLAB interprets indexed color data as either values to scale before mapping to the colormap, or directly as indices into the colormap. You control the interpretation by setting the `CDataMapping` property. The default is to scale the data.

Scaled Color

By default, MATLAB scales the color data so that the minimum value maps to the first color in the colormap, the maximum value maps to the last color in the colormap, and values in between are linearly transformed to span the colormap. This enables you to use colormaps of different sizes without

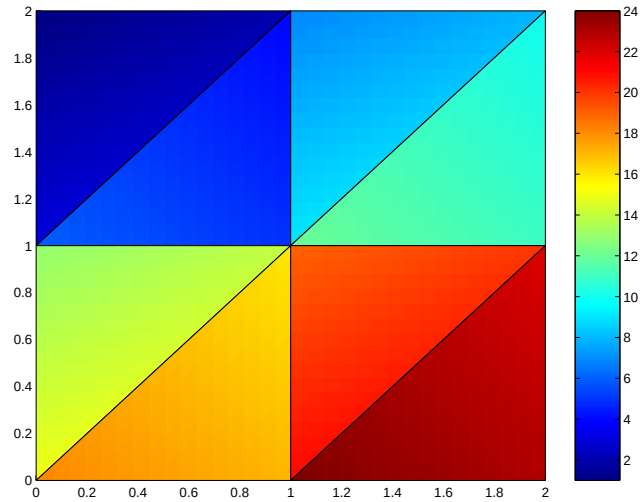
changing your data and to use data in any range of values without changing the colormap.

For example, the following patch has eight triangular faces with a total of 24 (nonunique) vertices. The color data are integers that range from one to 24, but could be any values.

The variable `c` contains the color data. It is a 3-by-8 matrix, with each column specifying the colors for the three vertices of each face.

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

The color bar (`colorbar`) on the right side of the patch illustrates the colormap used and indicates with the vertical axis which color is mapped to the respective data value.



You can alter the mapping of color data to colormap entry using the `caxis` command. This command uses a two-element vector `[cmin cmax]` to specify

what data values map to the beginning and end of the colormap, thereby shifting the color mapping.

By default, MATLAB sets `cmin` to the minimum value and `cmax` to the maximum value of the color data of all graphics objects within the axes. However, you can set these limits to span any range of values and thereby shift the color mapping. See Calculating Color Limits in "Axes Properties" in the Using MATLAB Graphics documentation for more information.

The color data does not need to be a sequential list of integers; it can be any matrix with dimensions matching the coordinate data. For example,

```
patch(x,y,z,rand(size(z)))
```

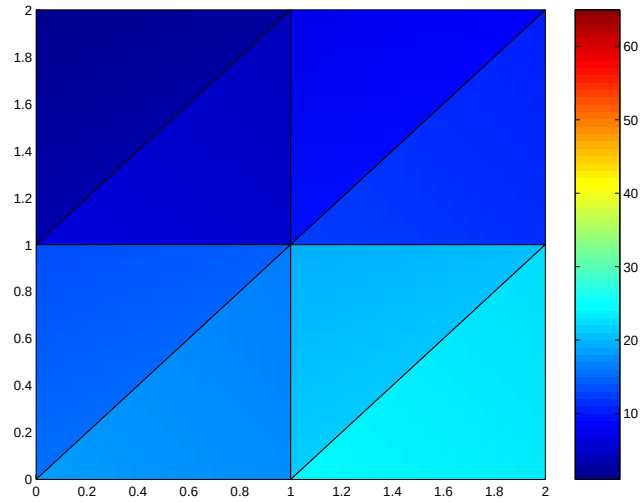
Direct Color

If you set the patch `CDataMapping` property to `direct`,

```
set(patch_handle,'CDataMapping','direct')
```

MATLAB interprets each color data value as a direct index into the colormap. That is, a value of 1 maps to the first color, a value of 2 maps to the second color, and so on.

The patch from the previous example would then use only the first 24 colors in the colormap.



This example uses integer color data. However, if the values are not integers, MATLAB converts them according to these rules:

- If value is < 1, it maps to the first color in the colormap.
- If value is not an integer, it is rounded to the nearest integer toward zero.
- If value > `length(colormap)`, it maps to the last color in the colormap.

Unscaled color data is more commonly used for images where there is typically a colormap associated with a particular image.

Truecolor Patches

Truecolor is a means to specify a color explicitly with RGB values rather than pointing to an entry in the figure colormap. Truecolor generally provides a greater range of colors than can be defined in a colormap.

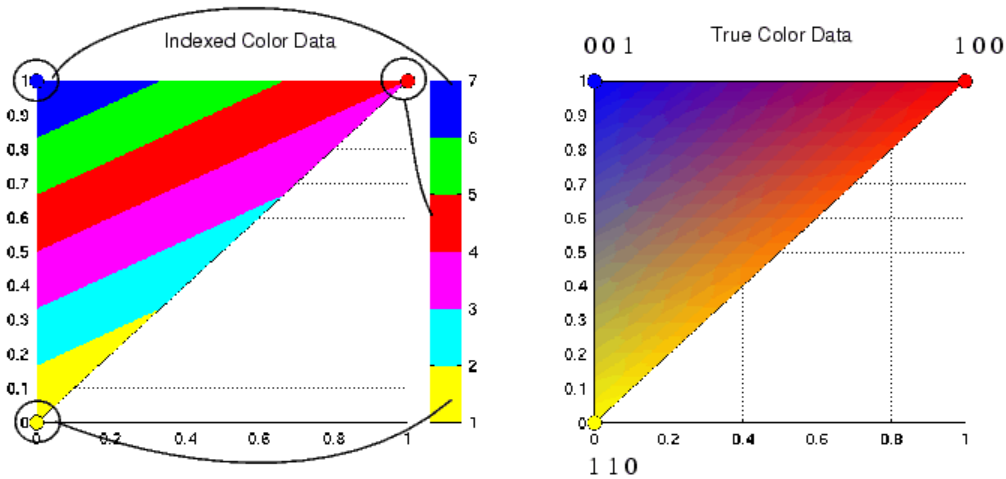
Using truecolor eliminates the mapping of data to colormap entries. On the other hand, you cannot change the coloring of the patch without redefining the color data (as opposed to just changing the colormap).

Interpolating in Indexed Color Versus Truecolor

When you specify interpolated face coloring, MATLAB determines the color of each face by interpolating the vertex colors. The method of interpolation depends on whether you specified truecolor data or indexed color data.

With truecolor data, MATLAB interpolates the numeric RGB values defined for the vertices. This generally produces a smooth variation of color across the face. In contrast, indexed color interpolation uses only colors that are defined in the colormap. With certain colormaps, the results can be quite different.

To illustrate this difference, these two patches are defined with the same vertex colors. Circular markers indicate the yellow, red, and blue vertex colors.



The patch on the left uses indexed colors obtained from the six-element colormap shown next to it. The color data maps the vertex colors to the colormap elements indicated in the picture. With this colormap, interpolating from the cyan vertex to the blue vertex can include only the colors green, red, yellow, and magenta, hence the banding.

Interpolation in RGB space makes no use of the colormap. It is simply the gradual transition from one numeric value to another. For example,

interpolating from the cyan vertex to the blue vertex follows a progression similar to these values.

```
0 1 1, 0 0.9 1, 0 0.8 1, ... 0 0.2 1, 0 0.1 1, 0 0 1
```

In reality each pixel would be a different color so the incremental change would be much smaller than illustrated here.

6

Volume Visualization Techniques

Overview of Volume Visualization

In this section...

Examples of Volume Data

Volume visualization is the creation of graphical representations of data sets that are defined on three-dimensional grids. Volume data sets are characterized by multidimensional arrays of scalar or vector data. These data are typically defined on lattice structures representing values sampled in 3-D space. There are two basic types of volume data:

- *Scalar volume data* contains single values for each point.
- *Vector volume data* contains two or three values for each point, defining the components of a vector.

An example of scalar volume data is that produced by the `flow` M-file. The flow data represents the speed profile of a submerged jet within an infinite tank. Typing

```
[x,y,z,v] = flow;
```

produces four 3-D arrays. The `x`, `y`, and `z` arrays specify the coordinates of the scalar values in the array `v`.

The `wind` data set is an example of vector volume data that represents air currents over North America. You can load this data in the MATLAB workspace with the command

```
load wind
```

This data set comprises six 3-D arrays: `x`, `y`, and `z` are the coordinate data for the arrays `u`, `v`, and `w`, which are the vector components for each point in the volume.

Selecting Visualization Techniques

The techniques you select to visualize volume data depend on what type of data you have and what you want to learn. In general,

- Scalar data is best viewed with isosurfaces, slice planes, and contour slices.
- Vector data represents both a magnitude and direction at each point, which is best displayed by stream lines (particles, ribbons, and tubes), cone plots, and arrow plots. Most visualizations, however, employ a combination of techniques to best reveal the content of the data.

The material in these sections describes how to apply a variety of techniques to typical volume data.

Steps to Create a Volume Visualization

Creating an effective visualization requires a number of steps to compose the final scene. These steps fall into four basic categories:

1. Determine the characteristics of your data. Graphing volume data usually requires knowledge of the range of both the coordinates and the data values.
2. Select an appropriate plotting routine. The information in this section helps you select the right methods.
3. Define the view. The information conveyed by a complex three-dimensional graph can be greatly enhanced through careful composition of the scene. Viewing techniques include adjusting camera position, specifying aspect ratio and project type, zooming in or out, and so on.
4. Add lighting and specify coloring. Lighting is an effective means to enhance the visibility of surface shape and to provide a three-dimensional perspective to volume graphs. Color can convey data values, both constant and varying.

Volume Visualization Functions

MATLAB provides functions that enable you to apply a variety of volume visualization techniques. The following tables group these functions into two categories based on the type of data (scalar or vector) that each is designed to work with. The reference page for each function provides examples of the intended use.

Functions for Scalar Data

Function	Purpose
`contourslice`	Draw contours in volume slice planes
`isocaps`	Compute isosurface end-cap geometry
`isocolors`	Compute the colors of isosurface vertices
`isonormals`	Compute normals of isosurface vertices
`isosurface`	Extract isosurface data from volume data
`patch`	Create a patch (multipolygon) graphics object
`reducepatch`	Reduce the number of patch faces
`reducevolume`	Reduce the number of elements in a volume data set
`shrinkfaces`	Reduce the size of each patch face
`slice`	Draw slice planes in volume
`smooth3`	Smooth 3-D data
`surf2patch`	Convert surface data to patch data
`subvolume`	Extract subset of volume data set

Functions for Vector Data

Function	Purpose
`coneplot`	Plot velocity vectors as cones in 3-D vector fields
`curl`	Compute the curl and angular velocity of a 3-D vector field

Function	Purpose
`divergence`	Compute the divergence of a 3-D vector field
`interpstreamspeed`	Interpolate streamline vertices from vector-field magnitudes
`streamline`	Draw stream lines from 2-D or 3-D vector data
`streamparticles`	Draw stream particles from vector volume data
`streamribbon`	Draw stream ribbons from vector volume data
`streamslice`	Draw well-spaced stream lines from vector volume data
`streamtube`	Draw stream tubes from vector volume data
`stream2`	Compute 2-D stream line data
`stream3`	Compute 3-D stream line data
`volumebounds`	Return coordinate and color limits for volume (scalar and vector)

Techniques for Visualizing Scalar Volume Data

In this section...

What Is Scalar Volume Data?

Typical scalar volume data is composed of a 3-D array of data and three coordinate arrays of the same dimensions. The coordinate arrays specify the *x*-, *y*-, and *z*-coordinates for each data point.

The units of the coordinates depend on the type of data. For example, flow data might have coordinate units of inches and data units of psi.

MATLAB supports a number of functions that are useful for visualizing scalar data:

- Slice planes provide a way to explore the distribution of data values within the volume by mapping values to colors. You can orient slice planes at arbitrary angles, as well as use nonplanar slices. (For illustrations of how to use slice planes, see `slice`, a volume slicing example, and slice planes used to show context.) You can specify the data used to color isosurfaces, enabling you to display different information in color and surface shape (see `isocolors`).
- Contour slices are contour plots drawn at specific coordinates within the volume. Contour plots enable you to see where in a given plane the data values are equal. See `contourslice` for an example
- Isosurfaces are surfaces constructed by using points of equal value as the vertices of `patch` graphics objects.

Example — Ways to Display MRI Data

An example of scalar data includes Magnetic Resonance Imaging (MRI) data. This data typically contains a number of slice planes taken through a volume, such as the human body. MATLAB includes an MRI data set that contains 27 image slices of a human head. This example illustrate the following techniques applied to MRI data:

- A series of 2-D images representing slices through the head
- 2-D and 3-D contour slices taken at arbitrary locations within the data
- An isosurface with isocaps showing a cross section of the interior

Changing the Data Format

The MRI data, `D`, is stored as a 128-by-128-by-1-by-27 array. The third array dimension is used typically for the image color data. However, since these are indexed images (a colormap, `map`, is also loaded) there is no information in the third dimension, which you can remove using the `squeeze` command. The result is a 128-by-128-by-27 array.

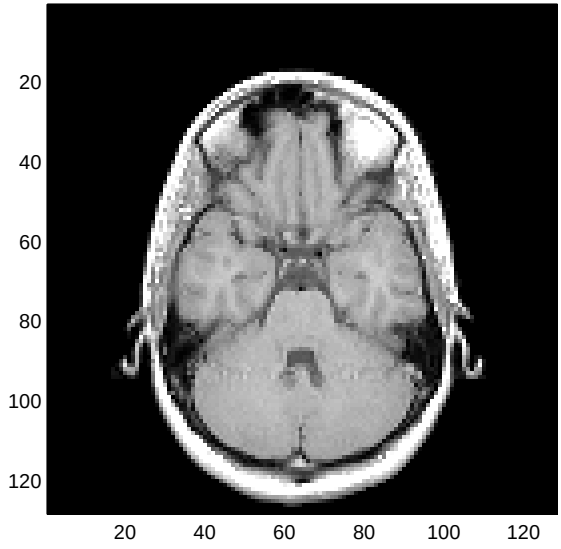
The first step is to load the data and transform the data array from 4-D to 3-D.

```
load mri
D = squeeze(D);
```

Displaying Images of MRI Data

To display one of the MRI images, use the `image` command, indexing into the data array to obtain the eighth image. Then adjust `axis` scaling, and install the MRI `colormap`, which was loaded along with the data.

```
image_num = 8;
image(D(:,:,image_num))
axis image
colormap(map)
```



Save the *x*- and *y*-axis limits for use in the next part of the example.

```
x = xlim;
y = ylim;
```

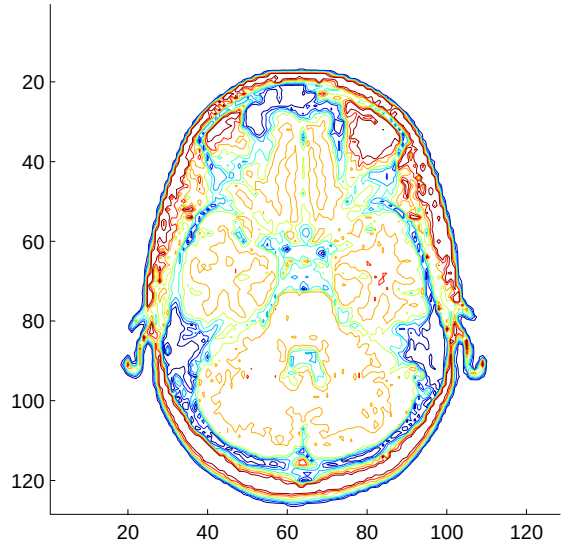
Displaying a 2-D Contour Slice

You can treat this MRI data as a volume because it is a collection of slices taken progressively through the 3-D object. Use `contourslice` to display a contour plot of a slice of the volume. To create a contour plot with the same

orientation and size as the image created in the first part of this example, adjust the *y*-axis direction (`axis`), set the limits (`xlim`, `ylim`), and set the data aspect ratio (`daspect`).

```
contourslice(D,[],[],image_num)
axis ij
xlim(x)
ylim(y)
daspect([1,1,1])
colormap('default')
```

This contour plot uses the figure colormap to map color to contour value.

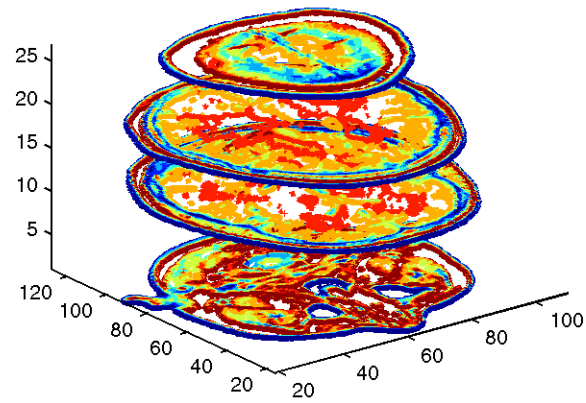


Displaying 3-D Contour Slices

Unlike images, which are 2-D objects, contour slices are 3-D objects that you can display in any orientation. For example, you can display four contour slices in a 3-D view. To improve the visibility of the contour line, increase the `LineWidth` to 2 points (one point equals 1/72 of an inch).

```
phandles = contourslice(D,[],[],[1,12,19,27],8);
```

```
view(3); axis tight
set(phandles,'LineWidth',2)
```



Displaying an Isosurface

You can use isosurfaces to display the overall structure of a volume. When combined with isocaps, this technique can reveal information about data on the interior of the isosurface.

First, smooth the data with `smooth3`; then use `isosurface` to calculate the isodata. Use `patch` to display this data as a graphics object.

```
Ds = smooth3(D);
hiso = patch(isosurface(Ds,5),...
 'FaceColor',[1,.75,.65],...
 'EdgeColor','none');
```

Adding an Isocap to Show a Cutaway Surface

Use `isocaps` to calculate the data for another patch that is displayed at the same isovalue (`5`) as the surface. Use the unsmoothed data (`D`) to show details of the interior. You can see this as the sliced-away top of the head.

```
hcap = patch(isocaps(D,5),...
 'FaceColor','interp',...
 'EdgeColor','none');
colormap(map)
```

Defining the View

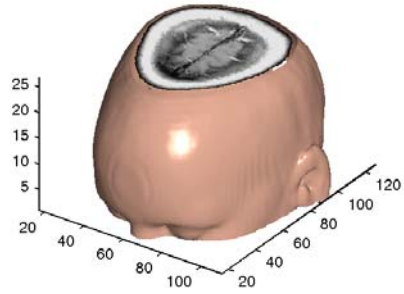
Define the view and set the aspect ratio (`view`, `axis`, `daspect`).

```
view(45,30)
axis tight
daspect([1,1,.4])
```

Add Lighting

Add lighting and recalculate the surface normals based on the gradient of the volume data, which produces smoother lighting (`camlight`, `lighting`, `isonormals`). Increase the `AmbientStrength` property of the isocap to brighten the coloring without affecting the isosurface. Set the `SpecularColorReflectance` of the isosurface to make the color of the specular reflected light closer to the color of the isosurface; then set the `SpecularExponent` to reduce the size of the specular spot.

```
lightangle(45,30);
set(gcf,'Renderer','zbuffer'); lighting phong
isonormals(Ds,hiso)
set(hcap,'AmbientStrength',.6)
set(hiso,'SpecularColorReflectance',0,'SpecularExponent',50)
```



Example of an Isocap

The isocap uses interpolated face coloring, which means the figure colormap determines the coloring of the patch. This example uses the colormap supplied with the data.

To display isocaps at other data values, try changing the isosurface value or use the `subvolume` command. See the `isocaps` and `subvolume` reference pages for examples.

Exploring Volumes with Slice Planes

In this section...

Example — Slicing Fluid Flow Data

A slice plane (which does not have to be planar) is a surface that takes on coloring based on the values of the volume data in the region where the slice is positioned. Slice planes are useful for probing volume data sets to discover where interesting regions exist, which you can then visualize with other types of graphs (see the `slice` example). Slice planes are also useful for adding a visual context to the bound of the volume when other graphing methods are also used (see `coneplot` and "Example — Stream Line Plots of Vector Data" on page 6-32 for examples).

Use the `slice` function to create slice planes. This example slices through a volume generated by the `flow` M-file.

1. Investigate the Data

Generate the volume data with the command

```
[x,y,z,v] = flow;
```

Determine the range of the volume by finding the minimum and maximum of the coordinate data.

```
xmin = min(x(:));
ymin = min(y(:));
zmin = min(z(:));

xmax = max(x(:));
ymax = max(y(:));
zmax = max(z(:));
```

2. Create a Slice Plane at an Angle to the X-Axes

To create a slice plane that does not lie in an axes plane, first define a surface and rotate it to the desired orientation. This example uses a surface that has the same *x*- and *y*- coordinates as the volume.

```
hslice = surf(linspace(xmin,xmax,100),...
 linspace(ymin,ymax,100),...
 zeros(100));
```

Rotate the surface by -45 degrees about the *x*-axis and save the surface `XData`, `YData`, and `ZData` to define the slice plane; then delete the surface.

```
rotate(hslice,[-1,0,0],-45)
xd = get(hslice,'XData');
yd = get(hslice,'YData');
zd = get(hslice,'ZData');
delete(hslice)
```

3. Draw the Slice Planes

Draw the rotated slice plane, setting the `FaceColor` to `interp` so that it is colored by the figure colormap, and set the `EdgeColor` to `none`. Increase the `DiffuseStrength` to `.8` to make this plane shine more brightly after adding a light source.

```
h = slice(x,y,z,v,xd,yd,zd);
set(h,'FaceColor','interp',...
 'EdgeColor','none',...
 'DiffuseStrength',.8)
```

Set `hold` to on and add three more orthogonal slice planes at `xmax`, `ymax`, and `zmin` to provide a context for the first plane, which slices through the volume at an angle.

```
hold on
hx = slice(x,y,z,v,xmax,[],[]);
set(hx,'FaceColor','interp','EdgeColor','none')

hy = slice(x,y,z,v,[],ymax,[]);
set(hy,'FaceColor','interp','EdgeColor','none')
```

```
hz = slice(x,y,z,v,[],[],zmin);
set(hz,'FaceColor','interp','EdgeColor','none')
```

4. Define the View

To display the volume in correct proportions, set the data aspect ratio to `[1,1,1]` (`daspect`). Adjust the axis to fit tightly around the volume (`axis`) and turn on the `box` to provide a sense of a 3-D object. The orientation of the axes can be selected initially using `rotate3d` to determine the best `view`.

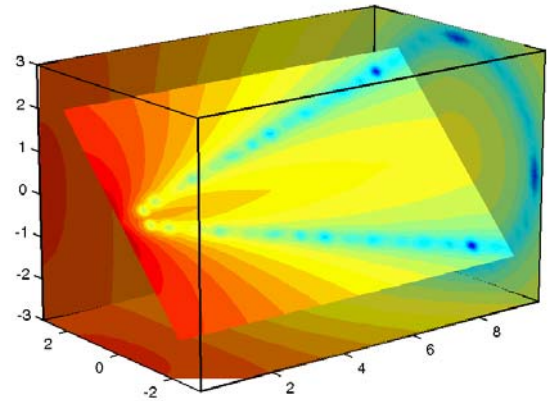
Zooming in on the scene provides a larger view of the volume (`camzoom`). Selecting a projection type of `perspective` gives the rectangular solid more natural proportions than the default orthographic projection (`camproj`).

```
daspect([1,1,1])
axis tight
box on
view(-38.5,16)
camzoom(1.4)
camproj perspective
```

5. Add Lighting and Specify Colors

Adding a light to the scene makes the boundaries between the four slice planes more obvious because each plane forms a different angle with the light source (`lightangle`). Selecting a colormap with only 24 colors (the default is 64) creates visible gradations that help indicate the variation within the volume.

```
lightangle(-45,45)
colormap (jet(24))
set(gcf,'Renderer','zbuffer')
```



The “Modifying the Color Mapping” on page 6-17 section shows how to modify how the data is mapped to color.

Modifying the Color Mapping

The current colormap determines the coloring of the slice planes. This enables you to change the slice plane coloring by

- Changing the colormap
- Changing the mapping of data value to color

Suppose, for example, you are interested in data values only between -5 and 2.5 and would like to use a colormap that mapped lower values to reds and higher values to blues (that is, the opposite of the default `jet` colormap).

Customizing the Colormap

The first step is to flip the colormap (`colormap`, `flipud`).

```
colormap (flipud(jet(24)))
```

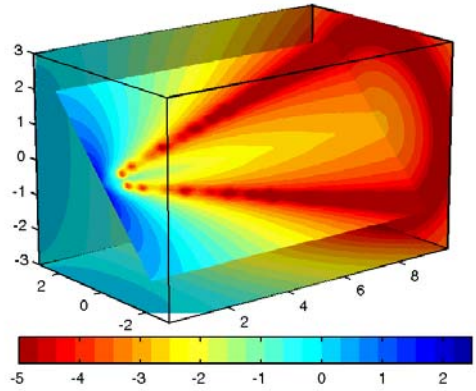
Adjusting the Color Limits

Adjusting the color limits enables you to emphasize any particular data range of interest. Adjust the color limits to range from -5 to 2.4832 so that any value lower than the value -5 (the original data ranged from -11.5417 to 2.4832) is mapped into the same color. (See `caxis` and Axis Color Limits - The CLim Property in Axes Properties in the MATLAB documentation for an explanation of color mapping.)

```
caxis([-5,2.4832])
```

Adding a color bar provides a key for the data-to-color mapping.

```
colorbar('horiz')
```



Connecting Equal Values with Isosurfaces

Example — Isosurfaces in Fluid Flow Data

Isosurfaces are constructed by creating a surface within the volume that has the same value at each vertex. Isosurface plots are similar to contour plots in that they both indicate where values are equal.

Isosurfaces are useful to determine where in a volume a certain threshold value is reached or to observe the spatial distribution of data by selecting various isovalues at which to generate a plot. The isovalue must lie within the range of the volume data.

Create isosurfaces with the `isosurface` and `patch` commands.

This example creates isosurfaces in a volume generated by the `flow` M-file. Generate the volume data with the command

```
[x,y,z,v] = flow;
```

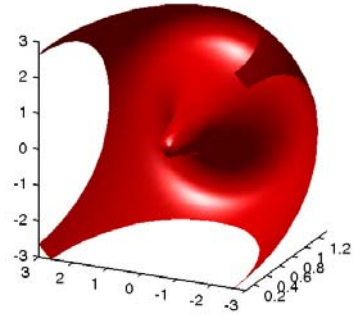
To select the isovalue, determine the range of values in the volume data.

```
min(v(:))
ans =
 -11.5417
max(v(:))
ans =
 2.4832
```

Through exploration, you can select isovalues that reveal useful information about the data. Once selected, use the isovalue to create the isosurface:

- Use `isosurface` to generate data that you can pass directly to `patch`.
- Recalculate the surface normals from the gradient of the volume data to produce better lighting characteristics (`isonormals`).
- Set the patch `FaceColor` to `red` and the `EdgeColor` to `none` to produce a smoothly lit surface.
- Adjust the view and add lighting (`daspect`, `view`, `camlight`, `lighting`).

```
hpatch = patch(isosurface(x,y,z,v,0));
isonormals(x,y,z,v,hpatch)
set(hpatch,'FaceColor','red','EdgeColor','none')
daspect([1,4,4])
view([-65,20])
axis tight
camlight left;
set(gcf,'Renderer','zbuffer'); lighting phong
```



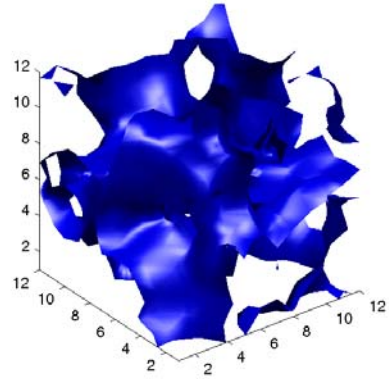
Isocaps Add Context to Visualizations

In this section...

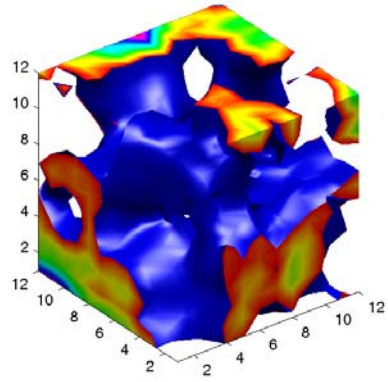
What Are Isocaps?

Isocaps are planes that are fitted to the limits of an isosurface to provide a visual context for the isosurface. Isocaps show a cross-sectional view of the interior of the isosurface for which the isocap provides an *end cap*.

The following two pictures illustrate the use of isocaps. The first is an isosurface without isocaps.



The second picture shows the effect of adding isocaps to the same isosurface.



Other Isocap Applications

Some additional applications of isocaps are shown in the following examples.

- Isocaps show the interior of a cut-away volume.
- Isocaps cap the end of a volume that would otherwise appear empty.
- Isocaps enhance the visibility of the isosurface limits.

Defining Isocaps

Isocaps, like isosurfaces, are created as `patch` graphics objects. Use the `isocaps` command to generate the data to pass to `patch`. For example,

```
patch(isocaps(voldata,isoval),...
 'FaceColor','interp',...
 'EdgeColor','none')
```

creates isocaps for the scalar volume data `voldata` at the value `isoval`. You should create the isosurface using the same volume data and isovalue to ensure that the edges of the isocaps *fit* the isosurface.

Setting the patch `FaceColor` property to `interp` results in a coloring that maps the data values spanned by the isocap to colormap entries. You can also set other patch properties to control the effects of lighting and coloring on the isocaps.

Example — Adding Isocaps to an Isosurface

This example illustrates how to set coloring and lighting characteristics when working with isocaps. There are five basic steps:

- Generate and process your volume data.
- Create the isosurface and isocaps and set patch properties to control the coloring and lighting.
- Create the isocaps and set properties.
- Specify the view.
- Add lights to the scene.

1. Prepare the Data

This example uses a 3-D array of random (`rand`) data to define the volume data. The data is then smoothed (`smooth3`).

```
data = rand(12,12,12);
data = smooth3(data,'box',5);
```

2. Create the Isosurface and Set Properties

Use `isosurface` and `patch` to create the isosurface and set coloring and lighting properties. Reduce the `AmbientStrength`, `SpecularStrength`, and `DiffuseStrength` of the reflected light to compensate for the brightness of the two light sources used to provide more uniform lighting.

Recalculate the vertex normals of the isosurface to produce smoother lighting (`isonormals`).

```
isoval = .5;
h = patch(isosurface(data,isoval),...
 'FaceColor','blue',...
 'EdgeColor','none',...
 'AmbientStrength',.2,...
 'SpecularStrength',.7,...
 'DiffuseStrength',.4);
isonormals(data,h)
```

3. Create the Isocaps and Set Properties

Define the `isocaps` using the same data and isovalue as the isosurface. Specify interpolated coloring and select a colormap that provides better contrasting colors with the blue isosurface than those in the default colormap (`colormap`).

```
patch(isocaps(data,isoval),...
 'FaceColor','interp',...
 'EdgeColor','none')
colormap hsv
```

4. Define the View

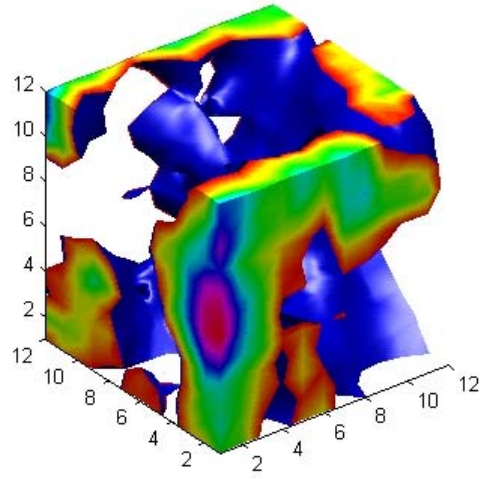
Set the data aspect ratio to `[1,1,1]` so that the display is in correct proportions (`daspect`). Eliminate white space within the axes and set the view to 3-D (`axis tight`, `view`).

```
daspect([1,1,1])
axis tight
view(3)
```

5. Add Lighting

To add fairly uniform lighting, but still take advantage of the ability of light sources to make visible subtle variations in shape, this example uses two lights, one to the left and one to the right of the camera (`camlight`). Use Phong lighting to produce the smoothest variation of color (`lighting`). Phong lighting requires the `zbuffer` renderer.

```
camlight right
camlight left
set(gcf,'Renderer','zbuffer');
lighting phong
```


12
10
8
6
4
2
12
10
8
6
4
2
2
4
6
8
10
12

Visualizing Vector Volume Data

In this section...

Lines, Particles, Ribbons, Streams, Tubes, and Cones

Vector volume data contains more information than scalar data because each coordinate point in the data set has three values associated with it. These values define a vector that represents both a magnitude and a direction. The velocity of fluid flow is an example of vector data.

MATLAB supports a number of techniques that are useful for visualizing vector data:

- Stream lines trace the path that a massless particle immersed in the vector field would follow.
- Stream particles are markers that trace stream lines and are useful for creating stream line animations.
- Stream ribbons are similar to stream lines, except that the width of the ribbons enables them to indicate twist. Stream ribbons are useful to indicate curl angular velocity.
- Stream tubes are similar to stream lines, but you can also control the width of the tube. Stream tubes are useful for displaying the divergence of a vector field.
- Cone plots represent the magnitude and direction of the data at each point by displaying a conical arrowhead or an arrow.

It is typically the case that these functions best elucidate the data when used in conjunction with other visualization techniques, such as contours, slice planes, and isosurfaces. The examples in this section illustrate some of these techniques.

Using Scalar Techniques with Vector Data

Visualization techniques such as contour slices, slice planes, and isosurfaces require scalar volume data. You can use these techniques with vector data by taking the magnitude of the vectors. For example, the `wind` data set returns three coordinate arrays and three vector component arrays, `u`, `v`, `w`. In this case, the magnitude of the velocity vectors equals the wind speed at each corresponding coordinate point in the volume.

```
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

The array `wind_speed` contains scalar values for the volume data. The usefulness of the information produced by this approach, however, depends on what physical phenomenon is represented by the magnitude of your vector data.

Specifying Starting Points for Stream Plots

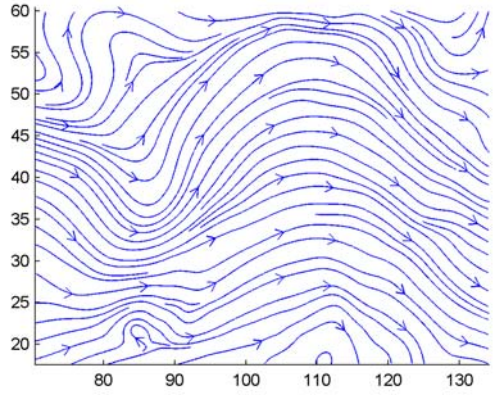
Stream plots (stream lines, ribbons, tubes, and cones or arrows) illustrate the flow of a 3-D vector field. The MATLAB stream plotting routines (`streamline`, `streamribbon`, `streamtube`, `coneplot`, `stream2`, `stream3`) all require you to specify the point at which you want to begin each stream trace.

Determining the Starting Points

Generally, knowledge of your data's characteristics helps you select the starting points. Information such as the primary direction of flow and the range of the data coordinates helps you decide where to evaluate the data.

The `streamslice` function is useful for exploring your data. For example, these statements draw a slice through the vector field at a z value midway in the range.

```
load wind
zmax = max(z(:)); zmin = min(z(:));
streamslice(x,y,z,u,v,w,[],[],(zmax-zmin)/2)
```



This stream slice plot indicates that the flow is in the positive *x* direction and also enables you to select starting points in both *x* and *y*. You could create similar plots that slice the volume in the *x*-*z* plane or the *y*-*z* plane to gain further insight into your data's range and orientation.

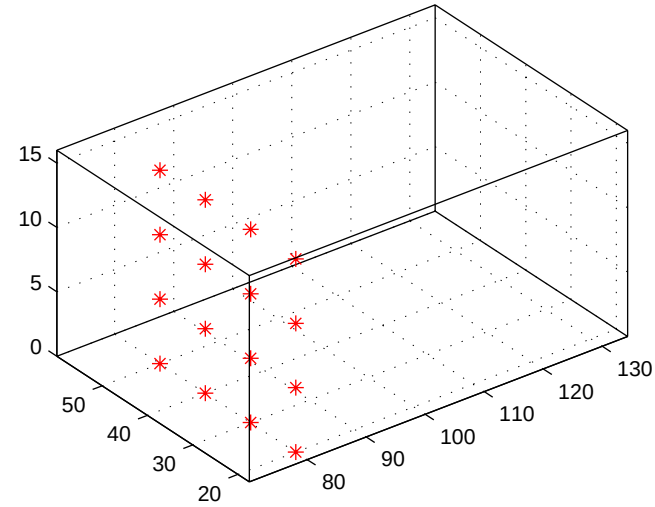
Specifying Arrays of Starting-Point Coordinates

To specify the starting point for one stream line, you need the *x*-, *y*-, and *z*-coordinates of the point. The `meshgrid` command provides a convenient way to create arrays of starting points. For example, you could select the following starting points from the wind data displayed in the previous stream slice.

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
```

This statement defines the starting points as all lying on x = 80, y ranging from 20 to 50, and z ranging from 0 to 15. You can use `plot3` to display the locations.

```
plot3(sx(:),sy(:),sz(:),'*r');
axis(volumebounds(x,y,z,u,v,w))
grid; box; daspect([2 2 1])
```



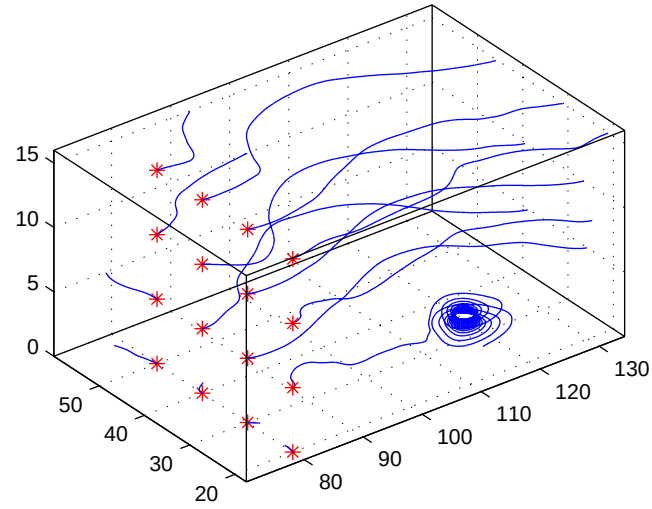
You do not need to use 3-D arrays, such as those returned by `meshgrid`, but the size of each array must be the same, and `meshgrid` provides a convenient way to generate arrays when you do not have an equal number of unique values in each coordinate. You can also define starting-point arrays as column vectors. For example, `meshgrid` returns 3-D arrays.

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
whos
Name      Size          Bytes  Class
sx        4x1x4           128  double array
sy        4x1x4           128  double array
sz        4x1x4           128  double array
```

In addition, you could use 16-by-1 column vectors with the corresponding elements of the three arrays composing the coordinates of each starting point. (This is the equivalent of indexing the values returned by `meshgrid` as `sx(:)`, `sy(:)`, and `sz(:)`.)

For example, adding the stream lines produces

```
streamline(x,y,z,u,v,w,sx(:),sy(:),sz(:))
```



Accessing Subregions of Volume Data

The `subvolume` function provides a simple way to access subregions of a volume data set. `subvolume` enables you to select regions of interest based on limits rather than using the colon operator to index into the 3-D arrays that define volumes. Consider the following two approaches to creating the data for a subvolume — indexing with the colon operator and using `subvolume`.

Indexing with the Colon Operator

When you index the arrays, you work with values that specify the elements in each dimension of the array.

```
load wind
xsub = x(1:10,20:30,1:7);
ysub = y(1:10,20:30,1:7);
zsub = z(1:10,20:30,1:7);
usub = u(1:10,20:30,1:7);
vsub = v(1:10,20:30,1:7);
wsub = w(1:10,20:30,1:7);
```

Using the subvolume Function

`subvolume` enables you to use coordinate values that you can read from the axes. For example,

```
lims = [100.64 116.67 17.25 28.75 -0.02 6.86];
[xsub,ysub,zsub,usub,vsub,wsub] = subvolume(x,y,z,u,v,w,lims);
```

You can then use the subvolume data as inputs to any function requiring vector volume data.

Example — Stream Line Plots of Vector Data

In this section...

Wind Mapping Data

MATLAB includes a vector data set called `wind` that represents air currents over North America. This example uses a combination of techniques:

- Stream lines to trace the wind velocity
- Slice planes to show cross-sectional views of the data
- Contours on the slice planes to improve the visibility of slice-plane coloring

1. Determine the Range of the Coordinates

Load the data and determine minimum and maximum values to locate the slice planes and contour plots (`load`, `min`, `max`).

```
load wind
xmin = min(x(:));
xmax = max(x(:));
ymax = max(y(:));
zmin = min(z(:));
```

2. Add Slice Planes for Visual Context

Calculate the magnitude of the vector field (which represents wind speed) to generate scalar data for the `slice` command. Create slice planes along the *x*-axis at `xmin`, `100`, and `xmax`, along the *y*-axis at `ymax`, and along the *z*-axis at `zmin`. Specify interpolated face coloring so the slice coloring indicates wind speed, and do not draw edges (`sqrt`, `slice`, `FaceColor`, `EdgeColor`).

```
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
hsurfaces = slice(x,y,z,wind_speed,[xmin,100,xmax],ymax,zmin);
set(hsurfaces,'FaceColor','interp','EdgeColor','none')
```

3. Add Contour Lines to the Slice Planes

Draw light gray contour lines on the slice planes to help quantify the color mapping (`contourslice`, `EdgeColor`, `LineWidth`).

```
hcont = ...
contourslice(x,y,z,wind_speed,[xmin,100,xmax],ymax,zmin);
set(hcont,'EdgeColor',[.7,.7,.7],'LineWidth',.5)
```

4. Define the Starting Points for the Stream Lines

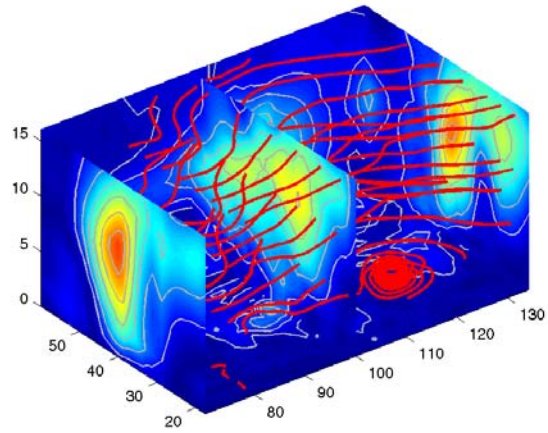
In this example, all stream lines start at an *x*-axis value of 80 and span the range 20 to 50 in the *y* direction and 0 to 15 in the *z* direction. Save the handles of the stream lines and set the line width and color (`meshgrid`, `streamline`, `LineWidth`, `Color`).

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
hlines = streamline(x,y,z,u,v,w,sx,sy,sz);
set(hlines,'LineWidth',2,'Color','r')
```

5. Define the View

Set up the view, expanding the *z*-axis to make it easier to read the graph (`view`, `daspect`, `axis`).

```
view(3)
daspect([2,2,1])
axis tight
```



See `coneplot` for an example of the same data plotted with cones.

Example — Displaying Curl with Stream Ribbons

In this section...

What Stream Ribbons Can Show

Stream ribbons illustrate direction of flow, similar to stream lines, but can also show rotation about the flow axis by twisting the ribbon-shaped flow line. The `streamribbon` function enables you to specify a twist angle (in radians) for each vertex in the stream ribbons.

When used in conjunction with the `curl` function, `streamribbon` is useful for displaying the curl angular velocity of a vector field. The following example illustrates this technique:

1. Select a Subset of Data to Plot

Load and select a region of interest in the `wind` data set using `subvolume`. Plotting the full data set first can help you select a region of interest.

```
load wind
lims = [100.64 116.67 17.25 28.75 -0.02 6.86];
[x,y,z,u,v,w] = subvolume(x,y,z,u,v,w,lims);
```

2. Calculate Curl Angular Velocity and Wind Speed

Calculate the `curl` angular velocity and the wind speed.

```
cav = curl(x,y,z,u,v,w);
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

3. Create the Stream Ribbons

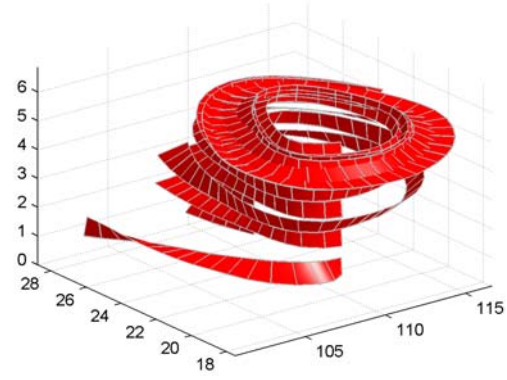
- Use `meshgrid` to create arrays of starting points for the stream ribbons. See "Starting Points for Stream Plots" in this chapter for information on specifying the arrays of starting points.
- `stream3` calculates the stream line vertices with a step size of `.5`.
- `streamribbon` scales the width of the ribbon by a factor of `2` to enhance the visibility of the twisting (which indicates curl angular velocity).
- `streamribbon` returns the handles of the surface objects it creates, which are then used to set the color to red (`FaceColor`), the color of the surface edges to light gray (`EdgeColor`), and slightly increase the brightness of the ambient light reflected when lighting is applied (`AmbientStrength`).

```
[sx sy sz] = meshgrid(110,20:5:30,1:5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz,.5);
h = streamribbon(verts,x,y,z,cav,wind_speed,2);
set(h,'FaceColor','r',...
 'EdgeColor',[.7 .7 .7],...
 'AmbientStrength',.6)
```

4. Define the View and Add Lighting

- The `volumebounds` command provides a convenient way to set `axis` and color limits.
- Add a `grid` and set the `view` for 3-D (`streamribbon` does not change the current view).
- `camlight` creates a light positioned to the right of the viewpoint and `lighting` sets the lighting method to Phong (which requires the Z-buffer renderer).

```
axis(volumebounds(x,y,z,wind_speed))
grid on
view(3)
camlight right;
set(gcf,'Renderer','zbuffer'); lighting phong
```


6
5
4
3
2
1
0
28
26
24
22
20
18
105
110
115

Example — Displaying Divergence with Stream Tubes

In this section...

What Stream Tubes Can Show

Stream tubes are similar to stream lines, except the tubes have width, providing another dimension that you can use to represent information.

By default, MATLAB indicates the divergence of the vector field by the width of the tube. You can also define widths for each tube vertex and thereby map other data to width.

This example uses the following techniques:

- Stream tubes to indicate flow direction and divergence of the vector field in the `wind` data set
- Slice planes colored to indicate the speed of the wind currents overlaid with contour line to enhance visibility

Inputs include the coordinates of the volume, vector field components, and starting locations for the stream tubes.

1. Load Data and Calculate Required Values

The first step is to load the data and calculate values needed to make the plots. These values include

- The location of the slice planes (maximum `x`, minimum `y`, and a value for the altitude)

- The minimum `x` value for the start of the stream tubes
- The speed of the wind (magnitude of the vector field)

```
load wind
xmin = min(x(:));
xmax = max(x(:));
ymin = min(y(:));
alt = 7.356; % z-value for slice and streamtube plane
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

2. Draw the Slice Planes

Draw the slice planes (`slice`) and set `surface` properties to create a smoothly colored slice. Use 16 colors from the `hsv colormap`.

```
hslice = slice(x,y,z,wind_speed,xmax,ymin,alt);
set(hslice,'FaceColor','interp','EdgeColor','none')
colormap hsv(16)
```

3. Add Contour Lines to Slice Planes

Add contour lines (`contourslice`) to the slice planes. Adjust the contour interval so the lines match the color boundaries in the slice planes:

- Call `caxis` to get the current color limits.
- Set the interpolation method used by `contourslice` to `linear` to match the default used by `slice`.

```
color_lim = caxis;
cont_intervals = linspace(color_lim(1),color_lim(2),17);
hcont = contourslice(x,y,z,wind_speed,xmax,ymin,...
 alt,cont_intervals,'linear');
set(hcont,'EdgeColor',[.4 .4 .4],'LineWidth',1)
```

4. Create the Stream Tubes

Use `meshgrid` to create arrays for the starting points for the stream tubes, which begin at the minimum `x` value, range from 20 to 50 in `y`, and lie in a single plane in `z` (corresponding to one of the slice planes).

The stream tubes (`streamtube`) are drawn at the specified locations and scaled to be 1.25 times the default width to emphasize the variation in divergence (width). The second element in the vector [1.25 30] specifies the number of points along the circumference of the tube (the default is 20). You might want to increase this value as the tube size increases, to maintain a smooth-looking tube.

Set the data aspect ratio (`daspect`) before calling `streamtube`.

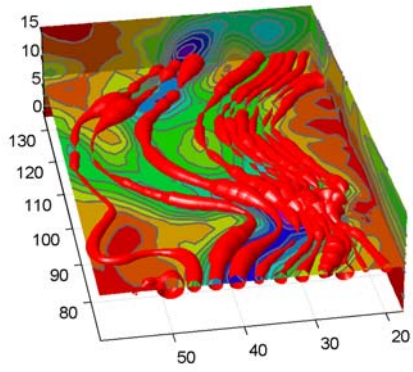
Stream tubes are surface objects, therefore you can control their appearance by setting surface properties. This example sets surface properties to give a brightly lit, red surface.

```
[sx,sy,sz] = meshgrid(xmin,20:3:50,alt);
daspect([1,1,1]) % set DAR before calling streamtube
htubes = streamtube(x,y,z,u,v,w,sx,sy,sz,[1.25 30]);
set(htubes,'EdgeColor','none','FaceColor','r',...
 'AmbientStrength',.5)
```

5. Define the View

The final step is to define the view and add lighting (`view`, `axis volumebounds`, `Projection`, `camlight`).

```
view(-100,30)
axis(volumebounds(x,y,z,wind_speed))
set(gca,'Projection','perspective')
camlight left
```


15
10
5
0
130
120
110
100
90
80
50
40
30
20

Example — Creating Stream Particle Animations

In this section...

What Particle Animations Can Show

A stream particle animation is useful for visualizing the flow direction and speed of a vector field. The "particles" (represented by any of the line markers) trace the flow along a particular stream line. The speed of each particle in the animation is proportional to the magnitude of the vector field at any given point along the stream line:

1. Specify the Starting Points of the Data Range to Plot

This example determines the region of the volume to plot by specifying the appropriate starting points. In this case, the stream plots begin at x = 100, y spans 20 to 50 and in the z = 5 plane. Note that this is not the full volume bounds.

```
load wind
[sx sy sz] = meshgrid(100,20:2:50,5);
```

2. Create Stream Lines to Indicate the Particle Paths

This example uses stream lines (`stream3`, `streamline`) to trace the path of the animated particles. This adds a visual context for the animation. Another possibility is to set the `EraseMode` property of the stream particle to none, which would be useful for a single trace through the volume.

```
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
sl = streamline(verts);
```

3. Define the View

While all the stream lines start in the z = 5 plane, the values of some spiral down to lower values. The following settings provide a clear view of the animation:

- The viewpoint (`view`) selected shows both the plane containing most stream lines and the spiral.
- Selecting a data aspect ratio (`daspect`) of `[2 2 0.125]` provides greater resolution in the *z* direction to make the stream particles more easily visible in the spiral.
- Set the axes limits to match the data limits (`axis`) and draw the axis box (`box`).

```
view(-10.5,18)
daspect([2 2 0.125])
axis tight; box on
```

4. Calculate the Stream Particle Vertices

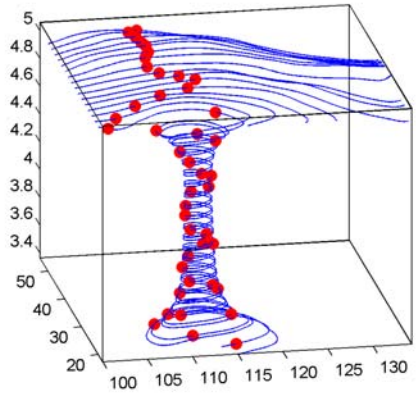
The first step is to determine the vertices along the stream line where a particle should be drawn. The `interpstreamspeed` function returns this data based on the stream line vertices and the speed of the vector data. This example scales the velocities by 0.05 to increase the number of interpolated vertices.

Setting the axes `DrawMode` property to `fast` enables the animation to run faster.

The `streamparticles` function sets the following properties:

- `Animate` to `10` to run the animation 10 times
- `ParticleAlignment` to `on` to start all particle traces together
- `MarkerEdgeColor` to `none` to draw only the face of the circular marker. Animations usually run faster when marker edges are not drawn.
- MarkerFaceColor to `red`
- `Marker` to `o`, which draws a circular marker. You can use other line markers as well.

```
iverts = interpstreamspeed(x,y,z,u,v,w,verts,0.05);
set(gca,'drawmode','fast');
streamparticles(iverts,15,...
 'Animate',10,...
 'ParticleAlignment','on',...
 'MarkerEdgeColor','none',...
 'MarkerFaceColor','red',...
 'Marker','o');
```



Example — Vector Field Displayed with Cone Plots

In this section...

What Cone Plots Can Show

This example plots the velocity vector cones for the `wind` data. The graph produced employs a number of visualization techniques:

- An isosurface is used to provide visual context for the cone plots and to provide means to select a specific data value for a set of cones.
- Lighting enables the shape of the isosurface to be clearly visible.
- The use of perspective projection, camera positioning, and view angle adjustments composes the final view.

1. Create an Isosurface

Displaying an isosurface within the rectangular space of the data provides a visual context for the cone plot. Creating the isosurface requires a number of steps:

- Calculate the magnitude of the vector field, which represents the speed of the wind.
- Use `isosurface` and `patch` to draw an isosurface illustrating where in the rectangular space the wind speed is equal to a particular value. Regions inside the isosurface have higher wind speeds, regions outside the isosurface have lower wind speeds.

- Use `isonormals` to compute vertex normals of the isosurface from the volume data rather than calculate the normals from the triangles used to render the isosurface. These normals generally produce more accurate results.
- Set visual properties of the isosurface, making it red and without drawing edges (`FaceColor`, `EdgeColor`).

```
load wind
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
hiso = patch(isosurface(x,y,z,wind_speed,40));
isonormals(x,y,z,wind_speed,hiso)
set(hiso,'FaceColor','red','EdgeColor','none');
```

2. Add Isocaps to the Isosurface

Isocaps are similar to slice planes in that they show a cross section of the volume. They are designed to be the end caps of isosurfaces. Using interpolated face color on an isocap causes a mapping of data value to color in the current colormap. To create isocaps for the isosurface, define them at the same isovalue (`isocaps`, `patch`, `colormap`).

```
hcap = patch(isocaps(x,y,z,wind_speed,40),...
    'FaceColor','interp',...
    'EdgeColor','none');
colormap hsv
```

3. Create First Set of Cones

- Use `daspect` to set the data aspect ratio of the axes before calling `coneplot` so MATLAB can determine the proper size of the cones.
- Determine the points at which to place cones by calculating another isosurface that has a smaller isovalue (so the cones are displayed outside the first isosurface) and use `reducepatch` to reduce the number of faces and vertices (so there are not too many cones on the graph).
- Draw the cones and set the face color to blue and the edge color to none.

```
daspect([1,1,1]);
[f verts] = reducepatch(isosurface(x,y,z,wind_speed,30),0.07);
h1 = coneplot(x,y,z,u,v,w,verts(:,1),verts(:,2),verts(:,3),3);
```

```
set(h1,'FaceColor','blue','EdgeColor','none');
```

4. Create Second Set of Cones

- Create a second set of points at values that span the data range (`linspace`, `meshgrid`).
- Draw a second set of cones and set the face color to green and the edge color to none.

```
xrange = linspace(min(x(:)),max(x(:)),10);
yrange = linspace(min(y(:)),max(y(:)),10);
zrange = 3:4:15;
[cx,cy,cz] = meshgrid(xrange,yrange,zrange);
h2 = coneplot(x,y,z,u,v,w,cx,cy,cz,2);
set(h2,'FaceColor','green','EdgeColor','none');
```

5. Define the View

- Use the `axis` command to set the axis limits equal to the minimum and maximum values of the data and enclose the graph in a box to improve the sense of a volume (`box`).
- Set the projection type to perspective to create a more natural view of the volume. Set the viewpoint and zoom in to make the scene larger (`camproj`, `camzoom`, `view`).

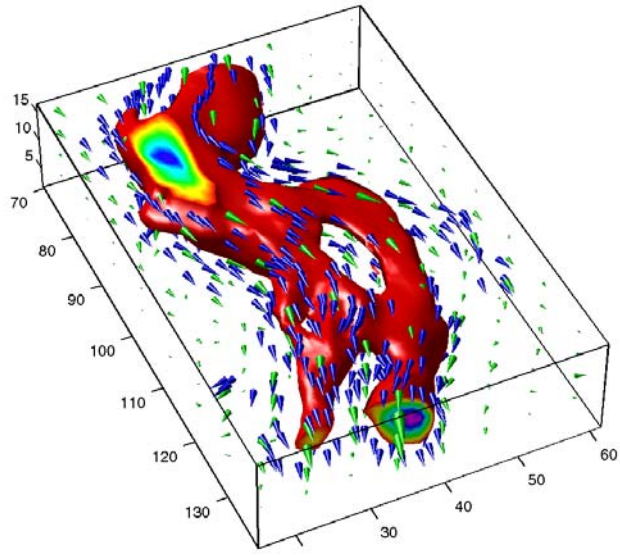
```
axis tight
box on
camproj perspective
camzoom(1.25)
view(65,45)
```

6. Add Lighting

Add a light source and use Phong lighting for the smoothest lighting of the isosurface (Phong lighting requires the Z-buffer renderer). Increase the strength of the background lighting on the isocaps to make them brighter (`camlight`, `lighting`, `AmbientStrength`).

```
camlight(-45,45)
```

```
set(gcf,'Renderer','zbuffer');
lighting phong
set(hcap,'AmbientStrength',.6)
```



Index

A

B

C

D

E

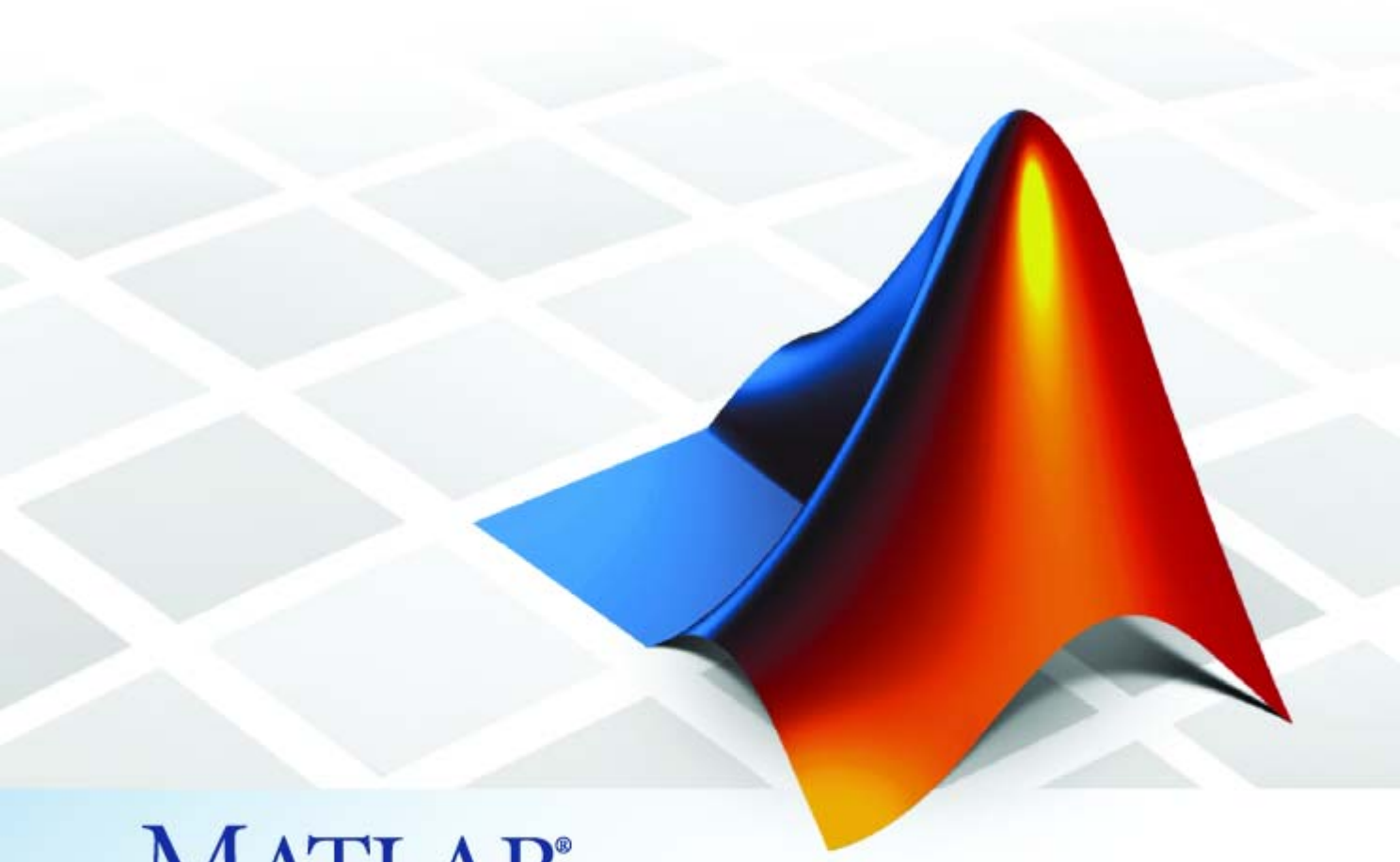
T

V

W

Z

MATLAB® 7
C and Fortran API Reference



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB C and Fortran API Reference

Revision History

December 1996	First Printing	New for MATLAB 5 (Release 8)
May 1997	Online only	Revised for MATLAB 5.1 (Release 9)
January 1998	Online Only	Revised for MATLAB 5.2 (Release 10)
January 1999	Online Only	Revised for MATLAB 5.3 (Release 11)
September 2000	Online Only	Revised for MATLAB 6.0 (Release 12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
January 2003	Online only	Revised for MATLAB 6.5.1 (Release 13SP1)
June 2004	Online only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised and renamed for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised and renamed for MATLAB 7.5 (Release 2007b)

Contents

1

API — By Category

MAT-File Access (p. 1-2)	Incorporate and use MATLAB® data in C and Fortran programs
MX Array Manipulation (p. 1-2)	Create and manipulate MATLAB arrays from C and Fortran MEX and engine routines
MEX-Files (p. 1-9)	Perform operations in MATLAB environment from C and Fortran MEX-files
MATLAB Engine (p. 1-11)	Call MATLAB from C and Fortran programs

See also “External Interfaces” in MATLAB Function Reference for MATLAB interfaces to DLLs, Java, COM and ActiveX, Web services, and serial port devices.

MAT-File Access

matClose (C and Fortran)	Close MAT-file
matDeleteVariable (C and Fortran)	Delete named mxArray from MAT-file
MATFile (C and Fortran)	Type for a MAT-file
matGetDir (C and Fortran)	Get directory of mxArrays in MAT-file
matGetFp (C)	Get file pointer to MAT-file
matGetNextVariable (C and Fortran)	Read next mxArray from MAT-file
matGetNextVariableInfo (C and Fortran)	Load array header information only
matGetVariable (C and Fortran)	Read mxArrays from MAT-files
matGetVariableInfo (C and Fortran)	Load array header information only
matOpen (C and Fortran)	Open MAT-file
matPutVariable (C and Fortran)	Write mxArrays to MAT-files
matPutVariableAsGlobal (C and Fortran)	Put mxArrays into MAT-files as originating from global workspace

MX Array Manipulation

mwIndex (C and Fortran)	Type for index values
mwPointer (Fortran)	Declare appropriate pointer type for platform
mwSize (C and Fortran)	Type for size values
mxAddField (C and Fortran)	Add field to structure array
mxArray (C and Fortran)	Type for a MATLAB array
mxArrayToString (C)	Convert array to string

mxAssert (C)	Check assertion value for debugging purposes
mxAssertS (C)	Check assertion value without printing assertion text
mxCalcSingleSubscript (C and Fortran)	Offset from first element to desired element
mxCalloc (C and Fortran)	Allocate dynamic memory for array using MATLAB memory manager
mxChar (C)	Type for string mxArray
mxClassID (C)	Enumerated value identifying class of mxArray
mxClassIDFromClassName (Fortran)	Identifier corresponding to class
mxComplexity (C)	Flag specifying whether mxArray has imaginary components
mxCopyCharacterToPtr (Fortran)	Copy character values from Fortran array to pointer array
mxCopyComplex16ToPtr (Fortran)	Copy COMPLEX*16 values from Fortran array to pointer array
mxCopyComplex8ToPtr (Fortran)	Copy COMPLEX*8 values from Fortran array to pointer array
mxCopyInteger1ToPtr (Fortran)	Copy INTEGER*1 values from Fortran array to pointer array
mxCopyInteger2ToPtr (Fortran)	Copy INTEGER*2 values from Fortran array to pointer array
mxCopyInteger4ToPtr (Fortran)	Copy INTEGER*4 values from Fortran array to pointer array
mxCopyPtrToCharacter (Fortran)	Copy character values from pointer array to Fortran array
mxCopyPtrToComplex16 (Fortran)	Copy COMPLEX*16 values from pointer array to Fortran array

mxCopyPtrToComplex8 (Fortran)	Copy COMPLEX*8 values from pointer array to Fortran array
mxCopyPtrToInteger1 (Fortran)	Copy INTEGER*1 values from pointer array to Fortran array
mxCopyPtrToInteger2 (Fortran)	Copy INTEGER*2 values from pointer array to Fortran array
mxCopyPtrToInteger4 (Fortran)	Copy INTEGER*4 values from pointer array to Fortran array
mxCopyPtrToPtrArray (Fortran)	Copy pointer values from pointer array to Fortran array
mxCopyPtrToReal4 (Fortran)	Copy REAL*4 values from pointer array to Fortran array
mxCopyPtrToReal8 (Fortran)	Copy REAL*8 values from pointer array to Fortran array
mxCopyReal4ToPtr (Fortran)	Copy REAL*4 values from Fortran array to pointer array
mxCopyReal8ToPtr (Fortran)	Copy REAL*8 values from Fortran array to pointer array
mxCreateCellArray (C and Fortran)	Create unpopulated N-D cell mxArray
mxCreateCellMatrix (C and Fortran)	Create unpopulated 2-D cell mxArray
mxCreateCharArray (C and Fortran)	Create unpopulated N-D string mxArray
mxCreateCharMatrixFromStrings (C and Fortran)	Create unpopulated 2-D string mxArray
mxCreateDoubleMatrix (C and Fortran)	Create 2-D, double-precision, floating-point mxArray initialized to 0
mxCreateDoubleScalar (C and Fortran)	Create scalar, double-precision array initialized to specified value
mxCreateLogicalArray (C)	Create N-D logical mxArray initialized to false

`mxCreateLogicalMatrix (C)`	Create 2-D, logical `mxArray` initialized to `false`
`mxCreateLogicalScalar (C)`	Create scalar, logical `mxArray` initialized to `false`
`mxCreateNumericArray (C and Fortran)`	Create unpopulated N-D numeric `mxArray`
`mxCreateNumericMatrix (C and Fortran)`	Create numeric matrix and initialize data elements to 0
`mxCreateSparse (C and Fortran)`	Create 2-D unpopulated sparse `mxArray`
`mxCreateSparseLogicalMatrix (C)`	Create unpopulated 2-D, sparse, logical `mxArray`
`mxCreateString (C and Fortran)`	Create 1-by-N string `mxArray` initialized to specified string
`mxCreateStructArray (C and Fortran)`	Create unpopulated N-D structure `mxArray`
`mxCreateStructMatrix (C and Fortran)`	Create unpopulated 2-D structure `mxArray`
`mxDestroyArray (C and Fortran)`	Free dynamic memory allocated by `mxCreate*` functions
`mxDuplicateArray (C and Fortran)`	Make deep copy of array
`mxFree (C and Fortran)`	Free dynamic memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`
`mxGetCell (C and Fortran)`	Get contents of `mxArray` cell
`mxGetChars (C)`	Get pointer to character array data
`mxGetClassID (C and Fortran)`	Get class of `mxArray`
`mxGetClassName (C and Fortran)`	Get class of `mxArray` as string
`mxGetData (C and Fortran)`	Get pointer to data
`mxGetDimensions (C andFortran)`	Get pointer to dimensions array

mxGetElementSize (C and Fortran)	Get number of bytes required to store each data element
mxGetEps (C and Fortran)	Get value of eps
mxGetField (C and Fortran)	Get field value, given field name and index into structure array
mxGetFieldByNumber (C and Fortran)	Get field value, given field number and index into structure array
mxGetFieldNameByNumber (C and Fortran)	Get field name, given field number in structure array
mxGetFieldNumber (C and Fortran)	Get field number, given field name in structure array
mxGetImagData (C and Fortran)	Get pointer to imaginary data of mxArray
mxGetInf (C and Fortran)	Get value of infinity
mxGetIr (C and Fortran)	Get ir array of sparse matrix
mxGetJc (C and Fortran)	Get jc array of sparse matrix
mxGetLogicals (C)	Get pointer to logical array data
mxGetM (C and Fortran)	Get number of rows in mxArray
mxGetN (C and Fortran)	Get number of columns in mxArray
mxGetNaN (C and Fortran)	Get value of NaN (Not-a-Number)
mxGetNumberOfDimensions (C and Fortran)	Get number of dimensions in mxArray
mxGetNumberOfElements (C and Fortran)	Get number of elements in mxArray
mxGetNumberOfFields (C and Fortran)	Get number of fields in structure mxArray
mxGetNzmax (C and Fortran)	Get number of elements in ir, pr, and pi arrays
mxGetPi (C and Fortran)	Get imaginary data elements in mxArray
mxGetPr (C and Fortran)	Get real data elements in mxArray

`mxGetScalar (C and Fortran)`	Get real component of first data element in `mxArray`
`mxGetString (C and Fortran)`	Copy string `mxArray` to C-style string
`mxIsCell (C and Fortran)`	Determine whether input is cell `mxArray`
`mxIsChar (C and Fortran)`	Determine whether input is string `mxArray`
`mxIsClass (C and Fortran)`	Determine whether `mxArray` is member of specified class
`mxIsComplex (C and Fortran)`	Determine whether data is complex
`mxIsDouble (C and Fortran)`	Determine whether `mxArray` represents data as double-precision, floating-point numbers
`mxIsEmpty (C and Fortran)`	Determine whether `mxArray` is empty
`mxIsFinite (C and Fortran)`	Determine whether input is finite
`mxIsFromGlobalWS (C and Fortran)`	Determine whether `mxArray` was copied from MATLAB global workspace
`mxIsInf (C and Fortran)`	Determine whether input is infinite
`mxIsInt16 (C and Fortran)`	Determine whether `mxArray` represents data as signed 16-bit integers
`mxIsInt32 (C and Fortran)`	Determine whether `mxArray` represents data as signed 32-bit integers
`mxIsInt64 (C and Fortran)`	Determine whether `mxArray` represents data as signed 64-bit integers
`mxIsInt8 (C and Fortran)`	Determine whether `mxArray` represents data as signed 8-bit integers

`mxIsLogical (C and Fortran)`	Determine whether `mxArray` is of type `mxLogical`
`mxIsLogicalScalar (C)`	Determine whether scalar `mxArray` is of type `mxLogical`
`mxIsLogicalScalarTrue (C)`	Determine whether scalar `mxArray` of type `mxLogical` is `true`
`mxIsNaN (C and Fortran)`	Determine whether input is `NaN` (Not-a-Number)
`mxIsNumeric (C and Fortran)`	Determine whether `mxArray` is numeric
`mxIsSingle (C and Fortran)`	Determine whether `mxArray` represents data as single-precision, floating-point numbers
`mxIsSparse (C and Fortran)`	Determine whether input is sparse `mxArray`
`mxIsStruct (C and Fortran)`	Determine whether input is structure `mxArray`
`mxIsUint16 (C and Fortran)`	Determine whether `mxArray` represents data as unsigned 16-bit integers
`mxIsUint32 (C and Fortran)`	Determine whether `mxArray` represents data as unsigned 32-bit integers
`mxIsUint64 (C and Fortran)`	Determine whether `mxArray` represents data as unsigned 64-bit integers
`mxIsUint8 (C and Fortran)`	Determine whether `mxArray` represents data as unsigned 8-bit integers
`mxLogical (C)`	Type for logical `mxArray`
`mxMalloc (C and Fortran)`	Allocate dynamic memory using MATLAB memory manager
`mxRealloc (C and Fortran)`	Reallocate memory

`mxRemoveField (C and Fortran)`	Remove field from structure array
`mxSetCell (C and Fortran)`	Set value of one cell of `mxArray`
`mxSetClassName (C)`	Convert structure array to MATLAB object array
`mxSetData (C and Fortran)`	Set pointer to data
`mxSetDimensions (C and Fortran)`	Modify number of dimensions and size of each dimension
`mxSetField (C and Fortran)`	Set structure array field, given field name and index
`mxSetFieldByNumber (C and Fortran)`	Set structure array field, given field number and index
`mxSetImagData (C and Fortran)`	Set imaginary data pointer for `mxArray`
`mxSetIr (C and Fortran)`	Set `ir` array of sparse `mxArray`
`mxSetJc (C and Fortran)`	Set `jc` array of sparse `mxArray`
`mxSetM (C and Fortran)`	Set number of rows in `mxArray`
`mxSetN (C and Fortran)`	Set number of columns in `mxArray`
`mxSetNzmax (C and Fortran)`	Set storage space for nonzero elements
`mxSetPi (C and Fortran)`	Set new imaginary data for `mxArray`
`mxSetPr (C and Fortran)`	Set new real data for `mxArray`

MEX-Files

`mexAtExit (C and Fortran)`	Register function to call when MEX-function is cleared or MATLAB terminates
`mexCallMATLAB (C and Fortran)`	Call MATLAB function or user-defined M-file or MEX-file

`mexErrMsgIdAndTxt (C and Fortran)`	Issue error message with identifier and return to MATLAB prompt
`mexErrMsgTxt (C and Fortran)`	Issue error message and return to MATLAB prompt
`mexEvalString (C and Fortran)`	Execute MATLAB command in caller's workspace
`mexFunction (C and Fortran)`	Entry point to C MEX-file
`mexFunctionName (C and Fortran)`	Name of current MEX-function
`mexGet (C)`	Get value of specified Handle Graphics® property
`mexGetVariable (C and Fortran)`	Get copy of variable from specified workspace
`mexGetVariablePtr (C and Fortran)`	Get read-only pointer to variable from another workspace
`mexIsGlobal (C and Fortran)`	Determine whether `mxArray` has global scope
`mexIsLocked (C and Fortran)`	Determine whether MEX-file is locked
`mexLock (C and Fortran)`	Prevent MEX-file from being cleared from memory
`mexMakeArrayPersistent (C and Fortran)`	Make `mxArray` persist after MEX-file completes
`mexMakeMemoryPersistent (C and Fortran)`	Make allocated memory MATLAB persist after MEX-function completes
`mexPrintf (C and Fortran)`	ANSI C `printf`-style output routine
`mexPutVariable (C and Fortran)`	Copy `mxArray` from MEX-function into specified workspace
`mexSet (C)`	Set value of specified Handle Graphics property

`mexSetTrapFlag (C and Fortran)`	Control response of `mexCallMATLAB` to errors
`mexUnlock (C and Fortran)`	Allow MEX-file to be cleared from memory
`mexWarnMsgIdAndTxt (C and Fortran)`	Issue warning message with identifier
`mexWarnMsgTxt (C and Fortran)`	Issue warning message

MATLAB Engine

`engClose (C and Fortran)`	Quit MATLAB engine session
`engEvalString (C and Fortran)`	Evaluate expression in string
`engGetVariable (C and Fortran)`	Copy variable from MATLAB engine workspace
`engGetVisible (C)`	Determine visibility of MATLAB engine session
`Engine (C)`	Type for a MATLAB engine
`engOpen (C and Fortran)`	Start MATLAB engine session
`engOpenSingleUse (C)`	Start MATLAB engine session for single, nonshared use
`engOutputBuffer (C and Fortran)`	Specify buffer for MATLAB output
`engPutVariable (C and Fortran)`	Put variables into MATLAB engine workspace
`engSetVisible (C)`	Show or hide MATLAB engine session

2

API — Alphabetical List

engClose (C and Fortran)

Purpose

Quit MATLAB engine session

C Syntax

```
#include "engine.h"
int engClose(Engine *ep);
```

Fortran Syntax

```
integer*4 engClose(ep)
mwPointer ep
```

Arguments

`ep`

Engine pointer

Returns

`0` on success, and `1` otherwise. Possible failure includes attempting to terminate a MATLAB engine session that was already terminated.

Description

This routine sends a quit command to the MATLAB engine session and closes the connection.

C Examples

UNIX

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

Windows

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

Fortran Examples

See `fengdemo.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

See Also

`engOpen`

Purpose

Evaluate expression in string

C Syntax

```
#include "engine.h"
int engEvalString(Engine *ep,const char *string);
```

Fortran Syntax

```
integer*4 engEvalString(ep, string)
mwPointer ep
character*(*) string
```

Arguments

`ep`
Engine pointer

`string`
String to execute

Returns

`0` if the command was evaluated by the MATLAB engine session, and nonzero otherwise. Possible reasons for failure include the MATLAB engine session is no longer running or the engine pointer is invalid or `NULL`.

Description

`engEvalString` evaluates the expression contained in `string` for the MATLAB engine session, `ep`, previously started by `engOpen`.

UNIX

On UNIX systems, `engEvalString` sends commands to MATLAB by writing down a pipe connected to the MATLAB *`stdin`*. Any output resulting from the command that ordinarily appears on the screen is read back from *`stdout`* into the buffer defined by `engOutputBuffer`.

To turn off output buffering in C, use

```
engOutputBuffer(ep, NULL, 0);
```

To turn off output buffering in Fortran, use

```
engOutputBuffer(ep, '')
```

Windows

On a PC, `engEvalString` communicates with MATLAB using a Component Object Model (COM) interface.

C Examples

UNIX

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

Windows

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

Fortran Examples

See `fengdemo.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

See Also

`engOpen`, `engOutputBuffer`

Purpose

Copy variable from MATLAB engine workspace

C Syntax

```
#include "engine.h"
mxArray *engGetVariable(Engine *ep, const char *name);
```

Fortran Syntax

```
mwPointer engGetVariable(ep, name)
mwPointer ep
character*(*) name
```

Arguments

`ep`
: Engine pointer

`name`
: Name of `mxArray` to get from MATLAB

Returns

A pointer to a newly allocated `mxArray` structure, or `NULL` if the attempt fails. `engGetVariable` fails if the named variable does not exist.

Description

`engGetVariable` reads the named `mxArray` from the MATLAB engine session associated with `ep`.

Be careful in your code to free the `mxArray` created by this routine when you are finished with it.

C Examples

UNIX

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

Windows

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

See Also

`engPutVariable`

engGetVisible (C)

Purpose

Determine visibility of MATLAB engine session

C Syntax

```
#include "engine.h"
int engGetVisible(Engine *ep, bool *value);
```

Arguments

`ep`
Engine pointer

`value`
Pointer to value returned from `engGetVisible`

Returns

Windows Only

`0` on success, and `1` otherwise.

Description

`engGetVisible` returns the current visibility setting for MATLAB engine session, `ep`. A *visible* engine session runs in a window on the Windows desktop, thus making the engine available for user interaction. An invisible session is hidden from the user by removing it from the desktop.

Examples

The following code opens engine session `ep` and disables its visibility.

```
Engine *ep;
bool vis;

ep = engOpen(NULL);
engSetVisible(ep, 0);
```

To determine the current visibility setting, use

```
engGetVisible(ep, &vis);
```

See Also

`engSetVisible`

Purpose

Type for a MATLAB engine

Description

A handle to a MATLAB engine object.

`Engine` is a C language opaque type.

You can call MATLAB as a computational engine by writing C and Fortran programs that use the MATLAB engine library, described in "MATLAB Engine" on page 1-11. `Engine` is the link between your program and the separate MATLAB engine process.

The header file containing this type is

```
#include "engine.h"
```

Examples

The example `engwindemo.c` (in your *`matlabroot`*`/extern/examples/eng_mat` directory) shows how to plot position versus time for a falling object in a MATLAB figure window.

The `engOpen` function starts the MATLAB process, returning an `Engine` variable. You use this handle for all calls to MATLAB.

The `mxCreateDoubleMatrix` function creates an `mxArray` named `T`. The C function `memcpy` copies your `time` data (initialized in `engwindemo.c`) into `T`.

The `engPutVariable` function puts `T` into MATLAB. Now you can use this variable to calculate distance `D`. The `engEvalString` function evaluates the expression `D = .5.*(-9.8).*T.^2`.

Next, various MATLAB plot functions, like `plot(T,D)`, display the graph.

Calls to the `engClose` and `mxDestroyArray` functions complete the procedure.

Other sample programs, also found in your *`matlabroot`*`\extern\examples\eng_mat` directory, that show you how to use `Engine` are:

- `engdemo.c` shows how to call the MATLAB engine functions from a C program.
- `engwindemo.c` show how to call the MATLAB engine functions from a C program for Windows.
- `fengdemo.F` shows how to call the MATLAB engine functions from a Fortran program.

See Also

`engOpen`

Purpose

Start MATLAB engine session

C Syntax

```
#include "engine.h"
Engine *engOpen(const char *startcmd);
```

Fortran Syntax

```
mwPointer engOpen(startcmd)
character*(*) startcmd
```

Arguments

`startcmd`
: String to start the MATLAB process. On Windows, the `startcmd` string must be `NULL`.

Returns

A pointer to an engine handle or `NULL` if the open fails.

Description

This routine allows you to start a MATLAB process for the purpose of using MATLAB as a computational engine.

`engOpen` starts a MATLAB process using the command specified in the string `startcmd`, establishes a connection, and returns a unique engine identifier, or `NULL` if the open fails.

On UNIX systems, if `startcmd` is `NULL` or the empty string, `engOpen` starts MATLAB on the current host using the command `matlab`. If `startcmd` is a `hostname`, `engOpen` starts MATLAB on the designated host by embedding the specified `hostname` string into the larger string:

```
"rsh hostname \"/bin/csh -c 'setenv DISPLAY\
 hostname:0; matlab'\""
```

If `startcmd` is any other string (has white space in it, or nonalphanumeric characters), the string is executed literally to start MATLAB.

On UNIX systems, `engOpen` performs the following steps:

1. Creates two pipes.

2 Forks a new process and sets up the pipes to pass *stdin* and *stdout* from MATLAB (parent) to two file descriptors in the engine program (child).

3 Executes a command to run MATLAB (`rsh` for remote execution).

Under Windows on a PC, `engOpen` opens a COM channel to MATLAB. This starts the MATLAB that was registered during installation. If you did not register during installation, on the command line you can enter the command

```
matlab /regserver
```

See "Introducing MATLAB COM Integration" for additional details.

C Examples

UNIX

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

Windows

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

Fortran Examples

See `fengdemo.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

Purpose

Start MATLAB engine session for single, nonshared use

C Syntax

```
#include "engine.h"
Engine *engOpenSingleUse(const char *startcmd, void *dcom,
  int *retstatus);
```

Arguments

`startcmd`
: String to start MATLAB process. On Windows, the `startcmd` string must be `NULL`.

`dcom`
: Reserved for future use; must be `NULL`.

`retstatus`
: Return status; possible cause of failure.

Returns

Windows Only

A pointer to an engine handle or `NULL` if the open fails.

UNIX

This routine is not supported and simply returns.

Description

This routine allows you to start multiple MATLAB processes for the purpose of using MATLAB as a computational engine. `engOpenSingleUse` starts a MATLAB process, establishes a connection, and returns a unique engine identifier, or `NULL` if the open fails. `engOpenSingleUse` starts a new MATLAB process each time it is called.

`engOpenSingleUse` opens a COM channel to MATLAB. This starts the MATLAB that was registered during installation. If you did not register during installation, on the command line you can enter the command

```
matlab /regserver
```

`engOpenSingleUse` allows single-use instances of a MATLAB engine server. `engOpenSingleUse` differs from `engOpen`, which allows multiple users to use the same MATLAB engine server.

See “Introducing MATLAB COM Integration” for additional details.

Purpose

Specify buffer for MATLAB output

C Syntax

```
#include "engine.h"
int engOutputBuffer(Engine *ep, char *p, int n);
```

Fortran Syntax

```
integer*4 engOutputBuffer(ep, p)
mwPointer ep
character*n p
```

Arguments

`ep`
: Engine pointer

`p`
: Pointer to character buffer

`n`
: Length of buffer `p`

Returns

`1` if you pass it a NULL engine pointer. Otherwise, it returns `0`.

Description

`engOutputBuffer` defines a character buffer for `engEvalString` to return any output that ordinarily appears on the screen.

The default behavior of `engEvalString` is to discard any standard output caused by the command it is executing. A call to `engOutputBuffer` with a buffer of nonzero length tells any subsequent calls to `engEvalString` to save output in the character buffer pointed to by `p`.

To turn off output buffering in C, use

```
engOutputBuffer(ep, NULL, 0);
```

To turn off output buffering in Fortran, use

```
engOutputBuffer(ep, '')
```

Note The buffer returned by `engEvalString` is not guaranteed to be `NULL` terminated.

C Examples

UNIX

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

Windows

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

Fortran Examples

See `fengdemo.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a Fortran program.

See Also

`engOpen`, `engEvalString`

Purpose

Put variables into MATLAB engine workspace

C Syntax

```
#include "engine.h"
int engPutVariable(Engine *ep, const char *name, const mxArray
     *pm);
```

Fortran Syntax

```
integer*4 engPutVariable(ep, name, pm)
mwPointer ep, pm
character*(*) name
```

Arguments

`ep`
: Engine pointer

`name`
: Name given to the `mxArray` in the engine's workspace

`pm`
: `mxArray` pointer

Returns

`0` if successful and `1` if an error occurs.

Description

`engPutVariable` writes `mxArray` `pm` to the engine `ep`, giving it the variable name `name`. If the `mxArray` does not exist in the workspace, it is created. If an `mxArray` with the same name already exists in the workspace, the existing `mxArray` is replaced with the new `mxArray`.

C Examples

UNIX

See `engdemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program.

Windows

See `engwindemo.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to call the MATLAB engine functions from a C program for Windows.

See Also

engGetVariable

Purpose

Show or hide MATLAB engine session

C Syntax

```
#include "engine.h"
int engSetVisible(Engine *ep, bool value);
```

Arguments

`ep`
: Engine pointer

`value`
: Value to set the `Visible` property to. Set `value` to `1` to make the engine window visible, or to `0` to make it invisible.

Returns

Windows Only

`0` on success, and `1` otherwise.

Description

`engSetVisible` makes the window for the MATLAB engine session, `ep`, either visible or invisible on the Windows desktop. You can use this function to enable or disable user interaction with the MATLAB engine session.

Examples

The following code opens engine session `ep` and disables its visibility.

```
Engine *ep;
bool vis;

ep = engOpen(NULL);
engSetVisible(ep, 0);
```

To determine the current visibility setting, use

```
engGetVisible(ep, &vis);
```

See Also

`engGetVisible`

matClose (C and Fortran)

Purpose

Close MAT-file

C Syntax

```
#include "mat.h"
int matClose(MATFile *mfp);
```

Fortran Syntax

```
integer*4 matClose(mfp)
mwPointer mfp
```

Arguments

`mfp`
Pointer to MAT-file information

Returns

`EOF` in C (`-1` in Fortran) for a write error, and `0` if successful.

Description

`matClose` closes the MAT-file associated with `mfp`.

C Examples

See `matcreat.c` and `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

Fortran Examples

See `matdemo1.F` and `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use this MAT-file routine in a Fortran program.

Purpose

Delete named mxArray from MAT-file

C Syntax

```
#include "mat.h"
int matDeleteVariable(MATFile *mfp, const char *name);
```

Fortran Syntax

```
integer*4 matDeleteVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Name of mxArray to delete

Returns

0 if successful, and nonzero otherwise.

Description

matDeleteVariable deletes the named mxArray from the MAT-file pointed to by mfp.

C Examples

See matcreat.c and matdgns.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

MATFile (C and Fortran)

Purpose

Type for a MAT-file

Description

A handle to a MAT-file object. A MAT-file is the data file format MATLAB uses for saving data to your disk.

`MATFile` is a C language opaque type.

The MAT-file interface library contains routines for reading and writing MAT-files. These routines are listed in "MAT-File Access" on page 1-2. You call these routines from your own C and Fortran programs, using `MATFile` to access your data file.

The header file containing this type is

```
#include "mat.h"
```

Examples

The example `matcreat.c` in your *`matlabroot`*`/extern/examples/eng_mat` directory shows how to create and use a MAT-file.

The `matOpen` function creates the file `mattest.mat`.

The `mxCreateDoubleMatrix` and `mxCreateString` functions create `mxArrays` `pa1`, `pa2`, and `pa3`. `mxCreateString` also initializes `pa3` using the literal string `"MATLAB: the language of technical computing"`. The C function `memcpy` copies `data` (initialized in `matcreat.c`) into `pa2`.

The `matPutVariable` and `matPutVariableAsGlobal` functions write the data to `mattest.mat`.

Calls to the `matClose` and `mxDestroyArray` functions complete the procedure.

Other examples, also found in your *`matlabroot`*`\extern\examples\eng_mat` directory, that show you how to use `MATFile` are:

- `matdgns.c` shows how to use MAT-file routines in a C program.
- `matdemo1.F` and `matdemo2.F` show how to use MAT-file routines in a Fortran program.

See Also

matOpen, matClose, matPutVariable, matGetVariable, mxDestroyArray

matGetDir (C and Fortran)

Purpose

Get directory of mxArrays in MAT-file

C Syntax

```
#include "mat.h"
char **matGetDir(MATFile *mfp, int *num);
```

Fortran Syntax

```
mwPointer matGetDir(mfp, num)
mwPointer mfp
integer*4 num
```

Arguments

mfp
: Pointer to MAT-file information

num
: Address of the variable to contain the number of mxArrays in the MAT-file

Returns

A pointer to an internal array containing pointers to the names of the mxArrays in the MAT-file pointed to by mfp. In C, each name is a NULL-terminated string. The length of the internal array (number of mxArrays in the MAT-file) is placed into num. If num is zero, mfp contains no arrays.

matGetDir returns NULL in C (0 in Fortran) and sets num to a negative number if it fails.

Description

This routine allows you to get a list of the names of the mxArrays contained within a MAT-file.

The internal array of strings that matGetDir returns is allocated using a single mxCalloc and must be freed using mxFree when you are finished with it.

MATLAB variable names can be up to length mxMAXNAM, where mxMAXNAM is defined in the C header file matrix.h.

C Examples

See matcreat.c and matdgns.c in the eng_mat subdirectory of the examples directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

Fortran Examples

See `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this MAT-file routine in a Fortran program.

matGetFp (C)

Purpose

Get file pointer to MAT-file

C Syntax

```
#include "mat.h"
FILE *matGetFp(MATFile *mfp);
```

Arguments

`mfp`
Pointer to MAT-file information

Returns

A C file handle to the MAT-file with handle `mfp`. Returns `NULL` if `mfp` is a handle to a MAT-file in HDF5-based format.

Description

Use `matGetFp` to obtain a C file handle to a MAT-file. This can be useful for using standard C library routines like `ferror` and `feof` to investigate error situations.

Examples

See `matcreat.c` and `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

Purpose

Read next mxArray from MAT-file

C Syntax

```
#include "mat.h"
mxArray *matGetNextVariable(MATFile *mfp, const char **name);
```

Fortran Syntax

```
mwPointer matGetNextVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

mfp
: Pointer to MAT-file information

name
: Address of the variable to contain the mxArray name

Returns

A pointer to a newly allocated mxArray structure representing the next mxArray from the MAT-file pointed to by mfp. The function returns the name of the mxArray in name.

matGetNextVariable returns NULL in C (0 in Fortran) when the end-of-file is reached or if there is an error condition. In C, use feof and ferror from the Standard C Library to determine status.

Description

matGetNextVariable allows you to step sequentially through a MAT-file and read all the mxArrays in a single pass. The function reads and returns the next mxArray from the MAT-file pointed to by mfp.

Use matGetNextVariable immediately after opening the MAT-file with matOpen and not in conjunction with other MAT-file routines. Otherwise, the concept of the *next* mxArray is undefined.

Free the memory used by the mxArray created by this routine when you are finished with it.

The order of variables returned from successive calls to matGetNextVariable is not guaranteed to be the same order in which the variables were written.

C Examples

See `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use the MATLAB MAT-file routines in a C program.

Purpose

Load array header information only

C Syntax

```
#include "mat.h"
mxArray *matGetNextVariableInfo(MATFile *mfp, const char **name);
```

Fortran Syntax

```
mwPointer matGetNextVariableInfo(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Address of the variable to contain the `mxArray` name

Returns

A pointer to a newly allocated `mxArray` structure representing header information for the next `mxArray` from the MAT-file pointed to by `mfp`. The function returns the name of the `mxArray` in `name`.

`matGetNextVariableInfo` returns `NULL` in C (`0` in Fortran) when the end-of-file is reached or if there is an error condition. In C, use `feof` and `ferror` from the Standard C Library to determine status.

Description

`matGetNextVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`, from the file's current file offset.

If `pr`, `pi`, `ir`, and `jc` are set to nonzero values when loaded with `matGetVariable`, `matGetNextVariableInfo` sets them to `-1` instead. These headers are for informational use only and should *never* be passed back to MATLAB or saved to MAT-files.

Free the memory used by the `mxArray` created by this routine when you are finished with it.

The order of variables returned from successive calls to `matGetNextVariableInfo` is not guaranteed to be the same order in which the variables were written.

C Examples

See `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use the MATLAB MAT-file routines in a C program.

See Also

`matGetNextVariable`, `matGetVariableInfo`

Purpose

Read mxArrays from MAT-files

C Syntax

```
#include "mat.h"
mxArray *matGetVariable(MATFile *mfp, const char *name);
```

Fortran Syntax

```
mwPointer matGetVariable(mfp, name)
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Name of `mxArray` to get from MAT-file

Returns

A pointer to a newly allocated `mxArray` structure representing the `mxArray` named by `name` from the MAT-file pointed to by `mfp`.

`matGetVariable` returns `NULL` in C (`0` in Fortran) if the attempt to return the `mxArray` named by `name` fails.

Description

This routine allows you to copy an `mxArray` out of a MAT-file.

Free the memory used by the `mxArray` created by this routine when you are finished with it.

C Examples

See `matcreat.c` and `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

matGetVariableInfo (C and Fortran)

Purpose

Load array header information only

C Syntax

```
#include "mat.h"
mxArray *matGetVariableInfo(MATFile *mfp, const char *name);
```

Fortran Syntax

```
mwPointer matGetVariableInfo(mfp, name);
mwPointer mfp
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Name of `mxArray` to get from MAT-file

Returns

A pointer to a newly allocated `mxArray` structure representing header information for the `mxArray` named by `name` from the MAT-file pointed to by `mfp`.

`matGetVariableInfo` returns `NULL` in C (`0` in Fortran) if the attempt to return header information for the `mxArray` named by `name` fails.

Description

`matGetVariableInfo` loads only the array header information, including everything except `pr`, `pi`, `ir`, and `jc`. It recursively creates the cells and structures through their leaf elements, but does not include `pr`, `pi`, `ir`, and `jc`.

If `pr`, `pi`, `ir`, and `jc` are set to nonzero values when loaded with `matGetVariable`, `matGetVariableInfo` sets them to `-1` instead. These headers are for informational use only and should *never* be passed back to MATLAB or saved to MAT-files.

Free the memory used by the `mxArray` created by this routine when you are finished with it.

C Examples

See `matcreat.c` and `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

See Also `matGetVariable`

matOpen (C and Fortran)

Purpose

Open MAT-file

C Syntax

```
#include "mat.h"
MATFile *matOpen(const char *filename, const char *mode);
```

Fortran Syntax

```
mwPointer matOpen(filename, mode)
character*(*) filename, mode
```

Arguments

`filename`

Name of file to open

`mode`

File opening mode. Valid values for `mode` are listed in the following table.

`r`	Opens file for reading only; determines the current version of the MAT-file by inspecting the files and preserves the current version.
`u`	Opens file for update, both reading and writing, but does not create the file if the file does not exist (equivalent to the `r+` mode of `fopen`); determines the current version of the MAT-file by inspecting the files and preserves the current version.
`w`	Opens file for writing only; deletes previous contents, if any.
`w4`	Creates a Level 4 MAT-file, compatible with MATLAB Versions 4 and earlier.
`wL`	Opens file for writing character data using the default character set for your system. The resulting MAT-file can be read with MATLAB Version 6 or 6.5. If you do not use the `wL` mode switch, MATLAB writes character data to the MAT-file using Unicode character encoding by default.

`wz`	Opens file for writing compressed data.
`w7.3`	Creates a MAT-file in an HDF5-based format that can store objects occupy more than 2 GB.

Returns

A file handle, or `NULL` in C (`0` in Fortran) if the open fails.

Description

This routine opens a MAT-file for reading and writing.

See "Writing Character Data" in the External Interfaces documentation for more information on how MATLAB uses character encodings.

C Examples

See `matcreat.c` and `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

Fortran Examples

See `matdemo1.F` and `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a Fortran program.

matPutVariable (C and Fortran)

Purpose

Write mxArrays to MAT-files

C Syntax

```
#include "mat.h"
int matPutVariable(MATFile *mfp, const char *name, const mxArray
  *pm);
```

Fortran Syntax

```
integer*4 matPutVariable(mfp, name, pm)
mwPointer mfp, pm
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Name of `mxArray` to put into MAT-file

`pm`
: `mxArray` pointer

Returns

`0` if successful and nonzero if an error occurs. In C, use `feof` and `ferror` from the Standard C Library along with `matGetFp` to determine status.

Description

This routine allows you to put an `mxArray` into a MAT-file.

`matPutVariable` writes `mxArray pm` to the MAT-file `mfp`. If the `mxArray` does not exist in the MAT-file, it is appended to the end. If an `mxArray` with the same name already exists in the file, the existing `mxArray` is replaced with the new `mxArray` by rewriting the file. The size of the new `mxArray` can be different from the existing `mxArray`.

C Examples

See `matcreat.c` and `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

Purpose

Put mxArrays into MAT-files as originating from global workspace

C Syntax

```
#include "mat.h"
int matPutVariableAsGlobal(MATFile *mfp, const char *name, const
  mxArray *pm);
```

Fortran Syntax

```
integer*4 matPutVariableAsGlobal(mfp, name, pm)
mwPointer mfp, pm
character*(*) name
```

Arguments

`mfp`
: Pointer to MAT-file information

`name`
: Name of `mxArray` to put into MAT-file

`pm`
: `mxArray` pointer

Returns

`0` if successful and nonzero if an error occurs. In C, use `feof` and `ferror` from the Standard C Library with `matGetFp` to determine status.

Description

This routine puts an `mxArray` into a MAT-file. `matPutVariableAsGlobal` is similar to `matPutVariable`, except that the array, when loaded by MATLAB, is placed into the global workspace and a reference to it is set in the local workspace. If you write to a MATLAB 4 format file, `matPutVariableAsGlobal` does not load it as global and has the same effect as `matPutVariable`.

`matPutVariableAsGlobal` writes `mxArray pm` to the MAT-file `mfp`. If the `mxArray` does not exist in the MAT-file, it is appended to the end. If an `mxArray` with the same name already exists in the file, the existing `mxArray` is replaced with the new `mxArray` by rewriting the file. The size of the new `mxArray` can be different from the existing `mxArray`.

C Examples

See `matcreat.c` and `matdgns.c` in the `eng_mat` subdirectory of the `examples` directory for sample programs that illustrate how to use the MATLAB MAT-file routines in a C program.

Purpose

Register function to call when MEX-function is cleared or MATLAB terminates

C Syntax

```
#include "mex.h"
int mexAtExit(void (*ExitFcn)(void));
```

Fortran Syntax

```
integer*4 mexAtExit(ExitFcn)
subroutine ExitFcn()
```

Arguments

`ExitFcn`
Pointer to function you want to run on exit

Returns

Always returns `0`.

Description

Use `mexAtExit` to register a function to be called just before the MEX-function is cleared or MATLAB is terminated. `mexAtExit` gives your MEX-function a chance to perform tasks such as freeing persistent memory and closing files. Typically, the named `ExitFcn` performs tasks like closing streams or sockets.

Each MEX-function can register only one active exit function at a time. If you call `mexAtExit` more than once, MATLAB uses the `ExitFcn` from the more recent `mexAtExit` call as the exit function.

If a MEX-function is locked, all attempts to clear the MEX-file will fail. Consequently, if a user attempts to clear a locked MEX-file, MATLAB does not call the `ExitFcn`.

In Fortran, you must declare the `ExitFcn` as `external` in the Fortran routine that calls `mexAtExit` if it is not within the scope of the file.

C Examples

See `mexatexit.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexLock`, `mexUnlock`, `mexSetTrapFlag`

mexCallMATLAB (C and Fortran)

Purpose

Call MATLAB function or user-defined M-file or MEX-file

C Syntax

```
#include "mex.h"
int mexCallMATLAB(int nlhs, mxArray *plhs[], int nrhs,
  mxArray *prhs[], const char *name);
```

Fortran Syntax

```
integer*4 mexCallMATLAB(nlhs, plhs, nrhs, prhs, name)
integer*4 nlhs, nrhs
mwPointer plhs(*), prhs(*)
character*(*) name
```

Arguments

`nlhs`

Number of desired output arguments. This value must be less than or equal to 50.

`plhs`

Array of pointers to `mxArrays`. The called command puts pointers to the resultant `mxArrays` into `plhs` and allocates dynamic memory to store the resultant `mxArrays`. By default, MATLAB automatically deallocates this dynamic memory when you clear the MEX-file. However, if heap space is at a premium, you may want to call `mxDestroyArray` as soon as you are finished with the `mxArrays` that `plhs` points to.

`nrhs`

Number of input arguments. This value must be less than or equal to `50`.

`prhs`

Array of pointers to input arguments.

`name`

Character string containing the name of the MATLAB built-in, operator, M-file, or MEX-file that you are calling. If `name` is an operator, just place the operator inside a pair of single quotes, for example, `'+'`.

Returns

`0` if successful, and a nonzero value if unsuccessful.

Description

Call `mexCallMATLAB` to invoke internal MATLAB numeric functions, MATLAB operators, M-files, or other MEX-files. See `mexFunction` for a complete description of the arguments.

By default, if `name` detects an error, MATLAB terminates the MEX-file and returns control to the MATLAB prompt. If you want a different error behavior, turn on the trap flag by calling `mexSetTrapFlag`.

It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. For example, if you create an M-file that returns two variables but assigns only one of them a value,

```
function [a,b]=foo(c)
a=2*c;
```

you get this warning message in MATLAB:

```
Warning: One or more output arguments not assigned
during call to 'foo'.
```

MATLAB assigns output `b` to an empty matrix. If you then call `foo` using `mexCallMATLAB`, the unassigned output variable is given type `mxUNKNOWN_CLASS`.

C Examples

See `mexcallmatlab.c` in the `mex` subdirectory of the `examples` directory.

Additional examples:

- `sincall.c` in the `refbook` subdirectory of the `examples` directory
- `mexevalstring.c` and `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory
- `mxcreatecellmatrix.c` and `mxisclass.c` in the `mx` subdirectory of the `examples` directory

See Also

`mexFunction`, `mexSetTrapFlag`

mexErrMsgIdAndTxt (C and Fortran)

Purpose

Issue error message with identifier and return to MATLAB prompt

C Syntax

```
#include "mex.h"
void mexErrMsgIdAndTxt(const char *errorid,
const char *errormsg, ...);
```

Fortran Syntax

```
mexErrMsgIdAndTxt(errorid, errormsg)
character*(*) errorid, errormsg
```

Arguments

`errorid`
: String containing a MATLAB message identifier. See "Message Identifiers" in the MATLAB documentation for information on this topic.

`errormsg`
: String containing the error message to be displayed. In C, the string may include formatting conversion characters, such as those used with the ANSI C `sprintf` function.

`...`
: In C, any additional arguments needed to translate formatting conversion characters used in `errormsg`. Each conversion character in `errormsg` is converted to one of these values.

Description

Call `mexErrMsgIdAndTxt` to write an error message and its corresponding identifier to the MATLAB window. After the error message prints, MATLAB terminates the MEX-file and returns control to the MATLAB prompt.

Calling `mexErrMsgIdAndTxt` does not clear the MEX-file from memory. Consequently, `mexErrMsgIdAndTxt` does not invoke the function registered through `mexAtExit`.

If your application called `mxCalloc` or one of the `mxCreate*` routines to allocate memory, `mexErrMsgIdAndTxt` automatically frees the allocated memory.

Note If you get warnings when using `mexErrMsgIdAndTxt`, you may have a memory management compatibility problem. For more information, see "Memory Management Compatibility Issues" in the External Interfaces documentation.

Remarks

In addition to the `errorid` and `errormsg`, the `mexerrmsgtxt` function determines where the error occurred, and displays the following information. For example, in the function `foo`, `mexerrmsgtxt` displays:

```
??? Error using ==> foo
```

See Also

`mexErrMsgTxt`, `mexWarnMsgIdAndTxt`, `mexWarnMsgTxt`

mexErrMsgTxt (C and Fortran)

Purpose

Issue error message and return to MATLAB prompt

C Syntax

```
#include "mex.h"
void mexErrMsgTxt(const char *errormsg);
```

Fortran Syntax

```
mexErrMsgTxt(errormsg)
character*(*) errormsg
```

Arguments

`errormsg`
: String containing the error message to be displayed

Description

Call `mexErrMsgTxt` to write an error message to the MATLAB window. After the error message prints, MATLAB terminates the MEX-file and returns control to the MATLAB prompt.

Calling `mexErrMsgTxt` does not clear the MEX-file from memory. Consequently, `mexErrMsgTxt` does not invoke the function registered through `mexAtExit`.

If your application called `mxCalloc` or one of the `mxCreate*` routines to allocate memory, `mexErrMsgTxt` automatically frees the allocated memory.

Note If you get warnings when using `mexErrMsgTxt`, you may have a memory management compatibility problem. For more information, see "Memory Management Compatibility Issues".

Remarks

In addition to the `errormsg`, the `mexerrmsgtxt` function determines where the error occurred, and displays the following information. If an error labeled `Print my error message` occurs in the function `foo`, `mexerrmsgtxt` displays:

```
??? Error using ==> foo
Print my error message
```

C Examples

See `xtimesy.c` in the `refbook` subdirectory of the `examples` directory.

For additional examples, see `convec.c`, `findnz.c`, `fulltosparse.c`, `phonebook.c`, `revord.c`, and `timestwo.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`, `mexWarnMsgTxt`

mexEvalString (C and Fortran)

Purpose

Execute MATLAB command in caller's workspace

C Syntax

```
#include "mex.h"
int mexEvalString(const char *command);
```

Fortran Syntax

```
integer*4 mexEvalString(command)
character*(*) command
```

Arguments

`command`
A string containing the MATLAB command to execute

Returns

`0` if successful, and a nonzero value if unsuccessful.

Description

Call `mexEvalString` to invoke a MATLAB `command` in the workspace of the caller.

`mexEvalString` and `mexCallMATLAB` both execute MATLAB commands. However, `mexCallMATLAB` provides a mechanism for returning results (left-hand side arguments) back to the MEX-file; `mexEvalString` provides no way for return values to be passed back to the MEX-file.

All arguments that appear to the right of an equal sign in the `command` string must already be current variables of the caller's workspace.

Examples

See `mexevalstring.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexCallMATLAB`

Purpose

Entry point to C MEX-file

C Syntax

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
     const mxArray *prhs[]);
```

Fortran Syntax

```
mexFunction(nlhs, plhs, nrhs, prhs)
integer*4 nlhs, nrhs
mwPointer plhs(*), prhs(*)
```

Arguments

`nlhs`
: The number of expected output `mxArrays`

`plhs`
: Array of pointers to the expected output `mxArrays`

`nrhs`
: The number of input `mxArrays`

`prhs`
: Array of pointers to the input `mxArrays`. These `mxArrays` are read only and should not be modified by your MEX-file. Changing the data in these `mxArrays` may produce undesired side effects.

Description

`mexFunction` is not a routine you call. Rather, `mexFunction` is the name of a function in C (subroutine in Fortran) that you must write in every MEX-file. When you invoke a MEX-function, MATLAB finds and loads the corresponding MEX-file of the same name. MATLAB then searches for a symbol named `mexFunction` within the MEX-file. If it finds one, it calls the MEX-function using the address of the `mexFunction` symbol. If MATLAB cannot find a routine named `mexFunction` inside the MEX-file, it issues an error message.

When you invoke a MEX-file, MATLAB automatically seeds `nlhs`, `plhs`, `nrhs`, and `prhs` with the caller's information. In the syntax of the MATLAB language, functions have the general form

```
[a,b,c,...] = fun(d,e,f,...)
```

where the ... denotes more items of the same format. The a,b,c... are left-hand side arguments, and the d,e,f... are right-hand side arguments. The arguments nlhs and nrhs contain the number of left-hand side and right-hand side arguments, respectively, with which the MEX-function is called. prhs is an array of mxArray pointers whose length is nrhs. plhs is an array whose length is nlhs, where your function must set pointers for the returned left-hand side mxArrays.

C Examples

See mexfunction.c in the mex subdirectory of the examples directory.

Purpose

Name of current MEX-function

C Syntax

```
#include "mex.h"
const char *mexFunctionName(void);
```

Fortran Syntax

```
character*(*) mexFunctionName()
```

Returns

The name of the current MEX-function.

Description

`mexFunctionName` returns the name of the current MEX-function.

C Examples

See `mexgetarray.c` in the `mex` subdirectory of the `examples` directory.

mexGet (C)

Purpose

Get value of specified Handle Graphics® property

C Syntax

```
#include "mex.h"
const mxArray *mexGet(double handle, const char *property);
```

Arguments

`handle`
Handle to a particular graphics object

`property`
A Handle Graphics property

Returns

The value of the specified property in the specified graphics object on success. Returns `NULL` on failure. The return argument from `mexGet` is declared as `constant`, meaning that it is read only and should not be modified. Changing the data in these `mxArrays` may produce undesired side effects.

Description

Call `mexGet` to get the value of the property of a certain graphics object. `mexGet` is the API equivalent of the MATLAB `get` function. To set a graphics property value, call `mexSet`.

Examples

See `mexget.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexSet`

Purpose

Get copy of variable from specified workspace

C Syntax

```
#include "mex.h"
mxArray *mexGetVariable(const char *workspace, const char
  *varname);
```

Fortran Syntax

```
mwPointer mexGetVariable(workspace, varname)
character*(*) workspace, varname
```

Arguments

`workspace`

Specifies where `mexGetVariable` should search in order to find array `varname`. The possible values are

`base`	Search for the variable in the base workspace.
`caller`	Search for the variable in the caller's workspace.
`global`	Search for the variable in the global workspace.

`varname`

Name of the variable to copy

Returns

A copy of the variable on success. Returns `NULL` in C (`0` on Fortran) on failure. A common cause of failure is specifying a variable that is not currently in the workspace. Perhaps the variable was in the workspace at one time but has since been cleared.

Description

Call `mexGetVariable` to get a copy of the specified variable. The returned `mxArray` contains a copy of all the data and characteristics that the variable had in the other workspace. Modifications to the returned `mxArray` do not affect the variable in the workspace unless you write the copy back to the workspace with `mexPutVariable`.

C Examples

See `mexgetarray.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexGetVariablePtr`, `mexPutVariable`

mexGetVariablePtr (C and Fortran)

Purpose

Get read-only pointer to variable from another workspace

C Syntax

```
#include "mex.h"
const mxArray *mexGetVariablePtr(const char *workspace,
  const char *varname);
```

Fortran Syntax

```
mwPointer mexGetVariablePtr(workspace, varname)
character*(*) workspace, varname
```

Arguments

`workspace`

Specifies which workspace you want `mexGetVariablePtr` to search. The possible values are

`base`	Search for the variable in the base workspace.
`caller`	Search for the variable in the caller's workspace.
`global`	Search for the variable in the global workspace.

`varname`

Name of a variable in another workspace. This is a variable name, not an `mxArray` pointer.

Returns

A read-only pointer to the `mxArray` on success. Returns `NULL` in C (`0` in Fortran) on failure.

Description

Call `mexGetVariablePtr` to get a read-only pointer to the specified variable, `varname`, into your MEX-file's workspace. This command is useful for examining an `mxArray`'s data and characteristics. If you need to change data or characteristics, use `mexGetVariable` (along with `mexPutVariable`) instead of `mexGetVariablePtr`.

If you simply need to examine data or characteristics, `mexGetVariablePtr` offers superior performance because the caller needs to pass only a pointer to the array.

C Examples

See `mxislogical.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mexGetVariable`

mexIsGlobal (C and Fortran)

Purpose

Determine whether mxArray has global scope

C Syntax

```
#include "matrix.h"
bool mexIsGlobal(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mexIsGlobal(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the mxArray has global scope, and logical 0 (false) otherwise.

Description

Use mexIsGlobal to determine whether the specified mxArray has global scope.

C Examples

See mxislogical.c in the mx subdirectory of the examples directory.

See Also

mexGetVariable, mexGetVariablePtr, mexPutVariable, global

Purpose

Determine whether MEX-file is locked

C Syntax

```
#include "mex.h"
bool mexIsLocked(void);
```

Fortran Syntax

```
integer*4 mexIsLocked()
```

Returns

Logical `1` (`true`) if the MEX-file is locked; logical `0` (`false`) if the file is unlocked.

Description

Call `mexIsLocked` to determine whether the MEX-file is locked. By default, MEX-files are unlocked, meaning that users can clear the MEX-file at any time.

To unlock a MEX-file, call `mexUnlock`.

C Examples

See `mexlock.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexLock`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mexUnlock`

mexLock (C and Fortran)

Purpose

Prevent MEX-file from being cleared from memory

C Syntax

```
#include "mex.h"
void mexLock(void);
```

Fortran Syntax

```
mexLock()
```

Description

By default, MEX-files are unlocked, meaning that a user can clear them at any time. Call `mexLock` to prohibit a MEX-file from being cleared.

To unlock a MEX-file, you must call `mexUnlock`. Do not use the `munlock` function.

`mexLock` increments a lock count. If you call `mexLock` `n` times, you must call `mexUnlock` `n` times to unlock your MEX-file.

C Examples

See `mexlock.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexIsLocked`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mexUnlock`

Purpose

Make mxArray persist after MEX-file completes

C Syntax

```
#include "mex.h"
void mexMakeArrayPersistent(mxArray *pm);
```

Fortran Syntax

```
mexMakeArrayPersistent(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray created by an mxCreate* function

Description

By default, mxArrays allocated by mxCreate* functions are not persistent. The MATLAB memory management facility automatically frees nonpersistent mxArrays when the MEX-function finishes. If you want the mxArray to persist through multiple invocations of the MEX-function, you must call mexMakeArrayPersistent.

Note If you create a persistent mxArray, you are responsible for destroying it when the MEX-file is cleared. If you do not destroy a persistent mxArray, MATLAB leaks memory. See mexAtExit to see how to register a function that gets called when the MEX-file is cleared. See mexLock to see how to lock your MEX-file so that it is never cleared.

See Also

mexAtExit, mexLock, mexMakeMemoryPersistent, and the mxCreate* functions

mexMakeMemoryPersistent (C and Fortran)

Purpose

Make allocated memory MATLAB persist after MEX-function completes

C Syntax

```
#include "mex.h"
void mexMakeMemoryPersistent(void *ptr);
```

Fortran Syntax

```
mexMakeMemoryPersistent(ptr)
mwPointer ptr
```

Arguments

`ptr`

Pointer to the beginning of memory allocated by one of the MATLAB memory allocation routines

Description

By default, memory allocated by MATLAB is nonpersistent, so it is freed automatically when the MEX-function finishes. If you want the memory to persist, you must call `mexMakeMemoryPersistent`.

Note If you create persistent memory, you are responsible for freeing it when the MEX-function is cleared. If you do not free the memory, MATLAB leaks memory. To free memory, use `mxFree`. See `mexAtExit` to see how to register a function that gets called when the MEX-function is cleared. See `mexLock` to see how to lock your MEX-function so that it is never cleared.

See Also

`mexAtExit`, `mexLock`, `mexMakeArrayPersistent`, `mxCalloc`, `mxFree`, `mxMalloc`, `mxRealloc`

Purpose

ANSI C `printf`-style output routine

C Syntax

```
#include "mex.h"
int mexPrintf(const char *message, ...);
```

Fortran Syntax

```
integer*4 mexPrintf(message)
character*(*) message
```

Arguments

`message`
: String to be displayed. In C, the string may include formatting conversion characters, such as those used with the ANSI C `printf` function.

`...`
: In C, any additional arguments needed to translate formatting conversion characters used in `message`. Each conversion character in `message` is converted to one of these values.

Returns

The number of characters printed. This includes characters specified with backslash codes, such as `\n` and `\b`.

Description

This routine prints a string on the screen and in the diary (if the diary is in use). It provides a callback to the standard C `printf` routine already linked inside MATLAB, and avoids linking the entire `stdio` library into your MEX-file.

In a C MEX-file, you must call `mexPrintf` instead of `printf` to display a string.

Note If you want the literal `%` in your message, you must use `%%` in your message string since `%` has special meaning to `mexPrintf`. Failing to do so causes unpredictable results.

C Examples

See

- `mexfunction.c` in the `mex` subdirectory of the `examples` directory
- `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mexErrMsgIdAndTxt`, `mexErrMsgTxt`, `mexWarnMsgIdAndTxt`, `mexWarnMsgTxt`

Purpose

Copy mxArray from MEX-function into specified workspace

C Syntax

```
#include "mex.h"
int mexPutVariable(const char *workspace, const char *varname,
  const mxArray *pm);
```

Fortran Syntax

```
integer*4 mexPutVariable(workspace, varname, pm)
character*(*) workspace, varname
mwPointer pm
```

Arguments

workspace
: Specifies the scope of the array that you are copying. The possible values are

base	Copy mxArray to the base workspace.
caller	Copy mxArray to the caller's workspace.
global	Copy mxArray to the list of global variables.

varname
: Name given to the mxArray in the workspace

pm
: Pointer to the mxArray

Returns

0 on success; 1 on failure. A possible cause of failure is that pm is NULL in C (0 in Fortran).

Description

Call mexPutVariable to copy the mxArray, at pointer pm, from your MEX-function into the specified workspace. MATLAB gives the name, varname, to the copied mxArray in the receiving workspace.

mexPutVariable makes the array accessible to other entities, such as MATLAB, M-files, or other MEX-functions.

If a variable of the same name already exists in the specified workspace, mexPutVariable overwrites the previous contents of the variable with

the contents of the new `mxArray`. For example, suppose the MATLAB workspace defines variable `Peaches` as

```
Peaches
1     2     3     4
```

and you call `mexPutVariable` to copy `Peaches` into the same workspace:

```
mexPutVariable("base", "Peaches", pm)
```

Then the old value of `Peaches` disappears and is replaced by the value passed in by `mexPutVariable`.

C Examples

See `mexgetarray.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexGetVariable`

Purpose

Set value of specified Handle Graphics property

C Syntax

```
#include "mex.h"
int mexSet(double handle, const char *property,
           mxArray *value);
```

Arguments

`handle`
: Handle to a particular graphics object

`property`
: String naming a Handle Graphics property

`value`
: Pointer to an `mxArray` holding the new value to assign to the property

Returns

`0` on success; `1` on failure. Possible causes of failure include:

- Specifying a nonexistent property.
- Specifying an illegal value for that property, for example, specifying a string value for a numerical property.

Description

Call `mexSet` to set the value of the property of a certain graphics object. `mexSet` is the API equivalent of the MATLAB `set` function. To get the value of a graphics property, call `mexGet`.

Examples

See `mexget.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexGet`

mexSetTrapFlag (C and Fortran)

Purpose

Control response of `mexCallMATLAB` to errors

C Syntax

```
#include "mex.h"
void mexSetTrapFlag(int trapflag);
```

Fortran Syntax

```
mexSetTrapFlag(trapflag)
integer*4 trapflag
```

Arguments

`trapflag`
Control flag. Possible values are

`0`	On error, control returns to the MATLAB prompt.
`1`	On error, control returns to your MEX-file.

Description

Call `mexSetTrapFlag` to control the MATLAB response to errors in `mexCallMATLAB`.

If you do not call `mexSetTrapFlag`, then whenever MATLAB detects an error in a call to `mexCallMATLAB`, MATLAB automatically terminates the MEX-file and returns control to the MATLAB prompt. Calling `mexSetTrapFlag` with `trapflag` set to `0` is equivalent to not calling `mexSetTrapFlag` at all.

If you call `mexSetTrapFlag` and set the `trapflag` to `1`, then whenever MATLAB detects an error in a call to `mexCallMATLAB`, MATLAB does not automatically terminate the MEX-file. Rather, MATLAB returns control to the line in the MEX-file immediately following the call to `mexCallMATLAB`. The MEX-file is then responsible for taking an appropriate response to the error.

If you call `mexSetTrapFlag`, the value of the `trapflag` you set remains in effect until the next call to `mexSetTrapFlag` within that MEX-file or, if there are no more calls to `mexSetTrapFlag`, until the MEX-file exits. If a routine defined in a MEX-file calls another MEX-file,

1. The current value of the `trapflag` in the first MEX-file is saved.

2 The second MEX-file is called with the `trapflag` initialized to `0` within that file.

3 When the second MEX-file exits, the saved value of the `trapflag` in the first MEX-file is restored within that file.

C Examples

See `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexAtExit`, `mexErrMsgTxt`

mexUnlock (C and Fortran)

Purpose

Allow MEX-file to be cleared from memory

C Syntax

```
#include "mex.h"
void mexUnlock(void);
```

Fortran Syntax

```
mexUnlock()
```

Description

By default, MEX-files are unlocked, meaning that a user can clear them at any time. Calling `mexLock` locks a MEX-file so that it cannot be cleared. Calling `mexUnlock` removes the lock so that the MEX-file can be cleared.

`mexLock` increments a lock count. If you called `mexLock` `n` times, you must call `mexUnlock` `n` times to unlock your MEX-file.

C Examples

See `mexlock.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mexIsLocked`, `mexLock`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`

Purpose

Issue warning message with identifier

C Syntax

```
#include "mex.h"
void mexWarnMsgIdAndTxt(const char *warningid,
  const char *warningmsg, ...);
```

Fortran Syntax

```
mexWarnMsgIdAndTxt(warningid, warningmsg)
character*(*) warningid, warningmsg
```

Arguments

`warningid`
: String containing a MATLAB message identifier. See "Message Identifiers" in the MATLAB documentation for information on this topic.

`warningmsg`
: String containing the warning message to be displayed. In C, the string may include formatting conversion characters, such as those used with the ANSI C `sprintf` function.

`...`
: In C, any additional arguments needed to translate formatting conversion characters used in `warningmsg`. Each conversion character in `warningmsg` is converted to one of these values.

Description

Call `mexWarnMsgIdAndTxt` to write a warning message and its corresponding identifier to the MATLAB window.

Unlike `mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt` does not cause the MEX-file to terminate.

See Also

`mexErrMsgTxt`, `mexErrMsgIdAndTxt`, `mexWarnMsgTxt`

mexWarnMsgTxt (C and Fortran)

Purpose

Issue warning message

C Syntax

```
#include "mex.h"
void mexWarnMsgTxt(const char *warningmsg);
```

Fortran Syntax

```
mexWarnMsgTxt(warningmsg)
character*(*) warningmsg
```

Arguments

`warningmsg`
String containing the warning message to be displayed

Description

`mexWarnMsgTxt` causes MATLAB to display the contents of `warningmsg`.

Unlike `mexErrMsgTxt`, `mexWarnMsgTxt` does not cause the MEX-file to terminate.

C Examples

See `yprime.c` in the `mex` subdirectory of the `examples` directory.

Additional examples:

- `explore.c` in the `mex` subdirectory of the `examples` directory
- `fulltosparse.c` in the `refbook` subdirectory of the `examples` directory
- `mxisfinite.c` and `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory

See Also

`mexErrMsgTxt`, `mexErrMsgIdAndTxt`, `mexWarnMsgIdAndTxt`

Purpose

Type for index values

Description

`mwIndex` is a type that represents index values, such as indices into arrays. This function is provided for purposes of cross-platform flexibility. By default, `mwIndex` is equivalent to `int` in C. When using the mex `-largeArrayDims` switch, `mwIndex` is equivalent to `size_t` in C. `mwIndex` is equivalent to `INTEGER*4` in Fortran. The C header file containing this type is

```
#include "matrix.h"
```

In Fortran, `mwIndex` is implemented as a preprocessor macro. The Fortran header file containing this type is

```
#include "fintrf.h"
```

See Also

`mex`, `mwSize`

mwPointer (Fortran)

Purpose

Declare appropriate pointer type for platform

Description

`mwPointer` is a preprocessor macro that declares the appropriate Fortran type representing a pointer to an `mxArray` or to other data that is not of a native Fortran type, such as memory allocated by `mxMalloc`. On 32-bit platforms, the Fortran type that represents a pointer is `INTEGER*4`; on 64-bit platforms, it is `INTEGER*8`. The Fortran preprocessor translates `mwPointer` to the Fortran declaration that is appropriate for the platform on which you compile your file.

If your Fortran compiler supports preprocessing, you can use `mwPointer` to declare functions, arguments, and variables that represent pointers. If you cannot use `mwPointer`, you must ensure that your declarations have the correct size for the platform on which you are compiling Fortran code.

The Fortran header file containing this type is

```
#include "fintrf.h"
```

Examples

This example declares the arguments for `mexFunction` in a Fortran MEX-file:

```
SUBROUTINE MEXFUNCTION(NLHS, PLHS, NRHS, PRHS)
MWPOINTER PLHS(*), PRHS(*)
INTEGER NLHS, NRHS
```

For additional examples, see the Fortran files with names ending in `.F` in the `$MATLAB/extern/examples` directory, where `$MATLAB` is the string returned by the `matlabroot` command.

Purpose

Type for size values

Description

`mwSize` is a type that represents size values, such as array dimensions. This function is provided for purposes of cross-platform flexibility. By default, `mwSize` is equivalent to `int` in C. When using the mex `-largeArrayDims` switch, `mwSize` is equivalent to `size_t` in C. `mwSize` is equivalent to `INTEGER*4` in Fortran.

In Fortran, `mwSize` is implemented as a preprocessor macro.

The C header file containing this type is

```
#include "matrix.h"
```

The Fortran header file containing this type is

```
#include "fintrf.h"
```

See Also

`mex`, `mwIndex`

mxAddField (C and Fortran)

Purpose

Add field to structure array

C Syntax

```
#include "matrix.h"
extern int mxAddField(mxArray pm, const char *fieldname);
```

Fortran Syntax

```
integer*4 mxAddField(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

Arguments

`pm`
: Pointer to a structure `mxArray`

`fieldname`
: The name of the field you want to add

Returns

Field number on success or `-1` if inputs are invalid or an out-of-memory condition occurs.

Description

Call `mxAddField` to add a field to a structure array. You must then create the values with the `mxCreate*` functions and use `mxSetFieldByNumber` to set the individual values for the field.

See Also

`mxRemoveField`, `mxSetFieldByNumber`

Purpose

Type for a MATLAB array

Description

The fundamental type underlying MATLAB data. For information on how the MATLAB array works with MATLAB-supported variables, see "MATLAB Data" in the External Interfaces documentation.

`mxArray` is a C language opaque type.

All C and Fortran MEX-files start with a gateway routine, called `mexFunction`, which requires `mxArray` for both input and output parameters. A C MEX-file gateway routine is described in "C MEX-Files". The Fortran version is described in "Fortran MEX-Files".

Once you have MATLAB data in your MEX-file, you can use the array access library routines (listed in "MX Array Manipulation" on page 1-2) to manipulate the data, and the MEX library routines (listed in "MEX-Files" on page 1-9) to perform operations in the MATLAB environment. You use `mxArray` to pass data to and from these functions.

Use any of the `mxcreate*` functions when you need to create data, and the corresponding `mxDestroyArray` function to free memory.

The header file containing this type is

```
#include "matrix.h"
```

Example

See `mxcreatecharmatrixfromstr.c` in your *`matlabroot`*`/extern/examples/mx` directory.

The input argument `prhs` contains two or more strings, defined as `mxArray`. Use the `mxIsChar` function to validate the input. Create a C variable `str` of type `char` using the `mxArrayToString` function. Now you can manipulate your data in C.

To set the return values in `plhs`, use the `mxCreateCharMatrixFromStrings` function.

Before you exit your routine, be sure to free memory using the `mxFree` function on `str`.

See Also

mexFunction, mxClassID, mxCreateDoubleMatrix, mxCreateNumericArray, mxCreateString, mxDestroyArray, mxGetData, mxSetData

Purpose

Convert array to string

C Syntax

```
#include "matrix.h"
char *mxArrayToString(const mxArray *array_ptr);
```

Arguments

`array_ptr`
Pointer to a string `mxArray`; that is, a pointer to an `mxArray` having the `mxCHAR_CLASS` class.

Returns

A C-style string. Returns NULL on failure. Possible reasons for failure include out of memory and specifying an `mxArray` that is not a string `mxArray`.

Description

Call `mxArrayToString` to copy the character data of a string `mxArray` into a C-style string. The C-style string is always terminated with a `NULL` character.

If the string array contains several rows, they are copied, one column at a time, into one long string array. This function is similar to `mxGetString`, except that

- It does not require the length of the string as an input.
- It supports multibyte character sets.

`mxArrayToString` does not free the dynamic memory that the `char` pointer points to. Consequently, you should typically free the string (using `mxFree`) immediately after you have finished using it.

Examples

See `mexatexit.c` in the `mex` subdirectory of the `examples` directory.

For additional examples, see `mxcreatecharmatrixfromstr.c` and `mxislogical.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCreateCharArray`, `mxCreateCharMatrixFromStrings`, `mxCreateString`, `mxGetString`

mxAssert (C)

Purpose

Check assertion value for debugging purposes

C Syntax

```
#include "matrix.h"
void mxAssert(int expr, char *error_message);
```

Arguments

`expr`
: Value of assertion

`error_message`
: Description of why assertion failed

Description

Similar to the ANSI C `assert` macro, `mxAssert` checks the value of an assertion, and continues execution only if the assertion holds. If `expr` evaluates to logical `1` (`true`), `mxAssert` does nothing. If `expr` evaluates to logical `0` (`false`), `mxAssert` prints an error to the MATLAB command window consisting of the failed assertion's expression, the filename and line number where the failed assertion occurred, and the `error_message` string. The `error_message` string allows you to specify a better description of why the assertion failed. Use an empty string if you don't want a description to follow the failed assertion message.

After a failed assertion, control returns to the MATLAB command line.

The `mex` script turns off these assertions when building optimized MEX-functions, so use this for debugging purposes only. Build the MEX-file using the syntax `mex -g filename` in order to use `mxAssert`.

Assertions are a way of maintaining internal consistency of logic. Use them to keep yourself from misusing your own code and to prevent logical errors from propagating before they are caught; do not use assertions to prevent users of your code from misusing it.

Assertions can be taken out of your code by the C preprocessor. You can use these checks during development and then remove them when the code works properly, letting you use them for troubleshooting during development without slowing down the final product.

Purpose

Check assertion value without printing assertion text

C Syntax

```
#include "matrix.h"
void mxAssertS(int expr, char *error_message);
```

Arguments

`expr`
Value of assertion

`error_message`
Description of why assertion failed

Description

`mxAssertS` is similar to `mxAssert`, except `mxAssertS` does not print the text of the failed assertion. `mxAssertS` checks the value of an assertion, and continues execution only if the assertion holds. If `expr` evaluates to logical `1` (`true`), `mxAssertS` does nothing. If `expr` evaluates to logical `0` (`false`), `mxAssertS` prints an error to the MATLAB command window consisting of the filename and line number where the assertion failed and the `error_message` string. The `error_message` string allows you to specify a better description of why the assertion failed. Use an empty string if you don't want a description to follow the failed assertion message.

After a failed assertion, control returns to the MATLAB command line.

Note that the `mex` script turns off these assertions when building optimized MEX-functions, so use this for debugging purposes only. Build the MEX-file using the syntax`mex -g filename` in order to use `mxAssertS`.

mxCalcSingleSubscript (C and Fortran)

Purpose

Offset from first element to desired element

C Syntax

```
#include "matrix.h"
mwIndex mxCalcSingleSubscript(const mxArray *pm, mwSize nsubs,
    mwIndex *subs);
```

Fortran Syntax

```
mwIndex mxCalcSingleSubscript(pm, nsubs, subs)
mwPointer pm
mwSize nsubs
mwIndex subs
```

Arguments

`pm`

Pointer to an `mxArray`

`nsubs`

The number of elements in the `subs` array. Typically, you set `nsubs` equal to the number of dimensions in the `mxArray` that `pm` points to.

subs

An array of integers. Each value in the array should specify that dimension's subscript. In C syntax, the value in `subs[0]` specifies the row subscript, and the value in `subs[1]` specifies the column subscript. Use zero-based indexing for subscripts. For example, to express the starting element of a two-dimensional `mxArray` in `subs`, set `subs[0]` to `0` and `subs[1]` to `0`.

In Fortran syntax, the value in `subs(1)` specifies the row subscript, and the value in `subs(2)` specifies the column subscript. Use 1-based indexing for subscripts. For example, to express the starting element of a two-dimensional `mxArray` in `subs`, set `subs(1)` to `1` and `subs(2)` to `1`.

Returns

The number of elements between the start of the `mxArray` and the specified subscript. This returned number is called an *index*; many `mx` routines (for example, `mxGetField`) require an index as an argument.

If subs describes the starting element of an mxArray, mxCalcSingleSubscript returns 0. If subs describes the final element of an mxArray, mxCalcSingleSubscript returns N-1 (where N is the total number of elements).

Description

Call mxCalcSingleSubscript to determine how many elements there are between the beginning of the mxArray and a given element of that mxArray. For example, given a subscript like (5,7), mxCalcSingleSubscript returns the distance from the first element of the array to the (5,7) element. Remember that the mxArray data type internally represents all data elements in a one-dimensional array no matter how many dimensions the MATLAB mxArray appears to have.

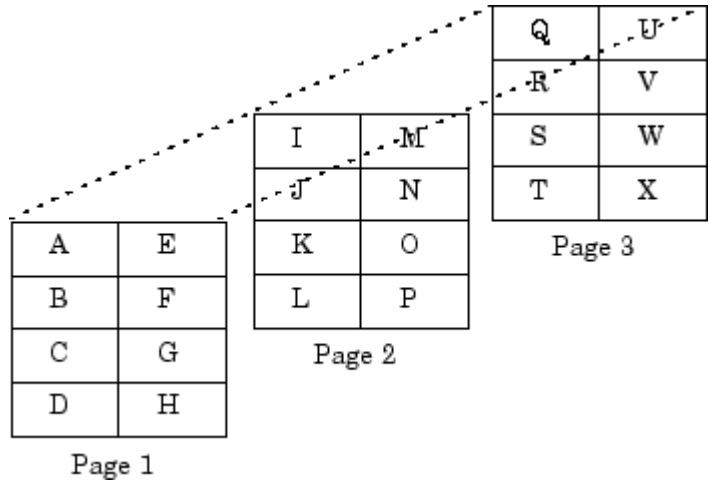
MATLAB uses a column-major numbering scheme to represent data elements internally. That means that MATLAB internally stores data elements from the first column first, then data elements from the second column second, and so on through the last column. For example, suppose you create a 4-by-2 variable. It is helpful to visualize the data as follows.

A	E
B	F
C	G
D	H

In fact, though, MATLAB internally represents the data as the following:

A	B	C	D	E	F	G	H
Index 0	Index 1	Index 2	Index 3	Index 4	Index 5	Index 6	Index 7

If an mxArray is N-dimensional, MATLAB represents the data in N-major order. For example, consider a three-dimensional array having dimensions 4-by-2-by-3. Although you can visualize the data as



MATLAB internally represents the data for this three-dimensional array in the following order:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Avoid using `mxCalcSingleSubscript` to traverse the elements of an array. In C, it is more efficient to do this by finding the array's starting address and then using pointer auto-incrementing to access successive elements. For example, to find the starting address of a numerical array, call `mxGetPr` or `mxGetPi`.

C Examples

See `mxcalcsinglesubscript.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxGetCell`, `mxSetCell`

Purpose

Allocate dynamic memory for array using MATLAB memory manager

C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxCalloc(mwSize n, mwSize size);
```

Fortran Syntax

```
mwPointer mxCalloc(n, size)
mwSize n, size
```

Arguments

`n`

Number of elements to allocate. This must be a nonnegative number.

`size`

Number of bytes per element. (The C `sizeof` operator calculates the number of bytes per element.)

Returns

A pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, `mxCalloc` returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt.

`mxCalloc` is unsuccessful when there is insufficient free heap space.

Description

MATLAB applications should always call `mxCalloc` rather than `calloc` to allocate memory. Note that `mxCalloc` works differently in MEX-files than in stand-alone MATLAB applications.

In MEX-files, `mxCalloc` automatically

- Allocates enough contiguous heap space to hold `n` elements.
- Initializes all `n` elements to `0`.
- Registers the returned heap space with the MATLAB memory management facility.

The MATLAB memory management facility maintains a list of all memory allocated by `mxCalloc`. The MATLAB memory management facility automatically frees (deallocates) all of a MEX-file's parcels when control returns to the MATLAB prompt.

In stand-alone MATLAB C applications, `mxCalloc` calls the ANSI C `calloc` function.

By default, in a MEX-file, `mxCalloc` generates nonpersistent `mxCalloc` data. In other words, the memory management facility automatically deallocates the memory as soon as the MEX-file ends. If you want the memory to persist after the MEX-file completes, call `mexMakeMemoryPersistent` after calling `mxCalloc`. If you write a MEX-file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX-file is cleared.

When you finish using the memory allocated by `mxCalloc`, call `mxFree`. `mxFree` deallocates the memory.

C Examples

See

- `explore.c` in the `mex` subdirectory of the `examples` directory
- `phonebook.c` and `revord.c` in the `refbook` subdirectory of the `examples` directory

For additional examples, see `mxcalcsinglesubscript.c` and `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxDestroyArray`, `mxFree`, `mxMalloc`, `mxRealloc`

Purpose

Type for string mxArray

Description

A string mxArray stores its data elements as mxChar rather than as char.

The header file containing this type is

```
#include "matrix.h"
```

Examples

See mxmalloc.c in the mx subdirectory of the examples directory.

Additional examples:

- explore.c in the mex subdirectory of the examples directory
- mxcreatecharmatrixfromstr.c in the mx subdirectory of the examples directory

See Also

mxCreateCharArray

mxClassID (C)

Purpose

Enumerated value identifying class of `mxArray`

C Syntax

```
typedef enum {
        mxUNKNOWN_CLASS,
        mxCELL_CLASS,
        mxSTRUCT_CLASS,
        mxLOGICAL_CLASS,
        mxCHAR_CLASS,
        mxDOUBLE_CLASS,
        mxSINGLE_CLASS,
        mxINT8_CLASS,
        mxUINT8_CLASS,
        mxINT16_CLASS,
        mxUINT16_CLASS,
        mxINT32_CLASS,
        mxUINT32_CLASS,
        mxINT64_CLASS,
        mxUINT64_CLASS,
        mxFUNCTION_CLASS
} mxClassID;
```

Constants

`mxUNKNOWN_CLASS`
: The class cannot be determined. You cannot specify this category for an `mxArray`; however, `mxGetClassID` can return this value if it cannot identify the class.

`mxCELL_CLASS`
: Identifies a cell `mxArray`.

`mxSTRUCT_CLASS`
: Identifies a structure `mxArray`.

`mxLOGICAL_CLASS`
: Identifies a logical `mxArray`, an `mxArray` whose data is represented as `mxLogical`.

`mxCHAR_CLASS`
: Identifies a string `mxArray`, an `mxArray` whose data is represented as `mxChar`.

`mxDOUBLE_CLASS`
: Identifies a numeric `mxArray` whose data is stored as double-precision, floating-point numbers.

`mxSINGLE_CLASS`
: Identifies a numeric `mxArray` whose data is stored as single-precision, floating-point numbers.

`mxINT8_CLASS`
: Identifies a numeric `mxArray` whose data is stored as signed 8-bit integers.

`mxUINT8_CLASS`
: Identifies a numeric `mxArray` whose data is stored as unsigned 8-bit integers.

`mxINT16_CLASS`
: Identifies a numeric `mxArray` whose data is stored as signed 16-bit integers.

`mxUINT16_CLASS`
: Identifies a numeric `mxArray` whose data is stored as unsigned 16-bit integers.

`mxINT32_CLASS`
: Identifies a numeric `mxArray` whose data is stored as signed 32-bit integers.

`mxUINT32_CLASS`
: Identifies a numeric `mxArray` whose data is stored as unsigned 32-bit integers.

`mxINT64_CLASS`
: Identifies a numeric `mxArray` whose data is stored as signed 64-bit integers.

`mxUINT64_CLASS`
: Identifies a numeric `mxArray` whose data is stored as unsigned 64-bit integers.

`mxFUNCTION_CLASS`
: Identifies a function handle `mxArray`.

mxClassID (C)

Description

Various `mx*` calls require or return an `mxClassID` argument. `mxClassID` identifies the way in which the `mxArray` represents its data elements.

Examples

See `explore.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mxGetClassID` , `mxCreateNumericArray`

Purpose

Identifier corresponding to class

Fortran Syntax

```
integer*4 mxClassIDFromClassName(classname)
character*(*) classname
```

Arguments

`classname`

A `character` array specifying a MATLAB class name. Use one of the strings from the following table.

Returns

A numeric identifier used internally by MATLAB to represent the MATLAB class, `classname`. Returns `unknown` if `classname` is not a recognized MATLAB class.

Description

Use `mxClassIDFromClassName` to obtain an identifier for any class that is recognized by MATLAB. This function is most commonly used to provide a `classid` argument to `mxCreateNumericArray` and `mxCreateNumericMatrix`.

Valid choices for `classname` are listed in the `mxIsClass` reference page.

See Also

`mxGetClassName`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxIsClass`

mxComplexity (C)

Purpose

Flag specifying whether `mxArray` has imaginary components

C Syntax

```
typedef enum mxComplexity {mxREAL=0, mxCOMPLEX};
```

Constants

`mxREAL`
: Identifies an `mxArray` with no imaginary components.

`mxCOMPLEX`
: Identifies an `mxArray` with imaginary components.

Description

Various `mx*` calls require an `mxComplexity` argument. You can set an `mxComplex` argument to either `mxREAL` or `mxCOMPLEX`.

Examples

See `mxcalcsinglesubscript.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCreateNumericArray`, `mxCreateDoubleMatrix`, `mxCreateSparse`

Purpose

Copy character values from Fortran array to pointer array

Fortran Syntax

```
mxCopyCharacterToPtr(y, px, n)
character*(*) y
mwPointer px
mwSize n
```

Arguments

`y`
: `character` Fortran array

`px`
: Pointer to `character` or `name` array

`n`
: Number of elements to copy

Description

`mxCopyCharacterToPtr` copies `n` `character` values from the Fortran `character` array `y` into the MATLAB string array pointed to by `px`. This subroutine is essential for copying character data between MATLAB pointer arrays and ordinary Fortran `character` arrays.

See Also

`mxCopyPtrToCharacter`, `mxCreateCharArray`, `mxCreateString`, `mxCreateCharMatrixFromStrings`

mxCopyComplex16ToPtr (Fortran)

Purpose

Copy COMPLEX*16 values from Fortran array to pointer array

Fortran Syntax

```
mxCopyComplex16ToPtr(y, pr, pi, n)
complex*16 y(n)
mwPointer pr, pi
mwSize n
```

Arguments

`y`
: COMPLEX*16 Fortran array

`pr`
: Pointer to the real data of a double-precision MATLAB array

`pi`
: Pointer to the imaginary data of a double-precision MATLAB array

`n`
: Number of elements to copy

Description

`mxCopyComplex16ToPtr` copies `n` `COMPLEX*16` values from the Fortran `COMPLEX*16` array `y` into the MATLAB arrays pointed to by `pr` and `pi`. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyPtrToComplex16`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Purpose

Copy COMPLEX*8 values from Fortran array to pointer array

Fortran Syntax

```
mxCopyComplex8ToPtr(y, pr, pi, n)
complex*8 y(n)
mwPointer pr, pi
mwSize n
```

Arguments

`y`
: COMPLEX*8 Fortran array

`pr`
: Pointer to the real data of a single-precision MATLAB array

`pi`
: Pointer to the imaginary data of a single-precision MATLAB array

`n`
: Number of elements to copy

Description

`mxCopyComplex8ToPtr` copies `n` `COMPLEX*8` values from the Fortran `COMPLEX*8` array `y` into the MATLAB arrays pointed to by `pr` and `pi`. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyPtrToComplex8`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

mxCopyInteger1ToPtr (Fortran)

Purpose

Copy INTEGER*1 values from Fortran array to pointer array

Fortran Syntax

```
mxCopyInteger1ToPtr(y, px, n)
integer*1 y(n)
mwPointer px
mwSize n
```

Arguments

`y`
: INTEGER*1 Fortran array

`px`
: Pointer to `ir` or `jc` array

`n`
: Number of elements to copy

Description

`mxCopyInteger1ToPtr` copies `n` INTEGER*1 values from the Fortran INTEGER*1 array `y` into the MATLAB array pointed to by `px`, either an `ir` or `jc` array. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyPtrToInteger1`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Purpose

Copy INTEGER*2 values from Fortran array to pointer array

Fortran Syntax

```
mxCopyInteger2ToPtr(y, px, n)
integer*2 y(n)
mwPointer px
mwSize n
```

Arguments

`y`
: INTEGER*2 Fortran array

`px`
: Pointer to `ir` or `jc` array

`n`
: Number of elements to copy

Description

`mxCopyInteger2ToPtr` copies `n` INTEGER*2 values from the Fortran INTEGER*2 array `y` into the MATLAB array pointed to by `px`, either an `ir` or `jc` array. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyPtrToInteger2`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

mxCopyInteger4ToPtr (Fortran)

Purpose

Copy INTEGER*4 values from Fortran array to pointer array

Fortran Syntax

```
mxCopyInteger4ToPtr(y, px, n)
integer*4 y(n)
mwPointer px
mwSize n
```

Arguments

`y`
: INTEGER*4 Fortran array

`px`
: Pointer to `ir` or `jc` array

`n`
: Number of elements to copy

Description

`mxCopyInteger4ToPtr` copies `n` INTEGER*4 values from the Fortran INTEGER*4 array `y` into the MATLAB array pointed to by `px`, either an `ir` or `jc` array. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyPtrToInteger4`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

Purpose

Copy character values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToCharacter(px, y, n)
mwPointer px
character*(*) y
mwSize n
```

Arguments

`px`
: Pointer to character or name array

`y`
: character Fortran array

`n`
: Number of elements to copy

Description

`mxCopyPtrToCharacter` copies `n` character values from the MATLAB array pointed to by `px` into the Fortran character array `y`. This subroutine is essential for copying character data from MATLAB pointer arrays into ordinary Fortran character arrays.

Examples

See `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

`mxCopyCharacterToPtr`, `mxCreateCharArray`, `mxCreateString`, `mxCreateCharMatrixFromStrings`

mxCopyPtrToComplex16 (Fortran)

Purpose

Copy COMPLEX*16 values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToComplex16(pr, pi, y, n)
mwPointer pr, pi
complex*16 y(n)
mwSize n
```

Arguments

`pr`
: Pointer to the real data of a double-precision MATLAB array

`pi`
: Pointer to the imaginary data of a double-precision MATLAB array

`y`
: COMPLEX*16 Fortran array

`n`
: Number of elements to copy

Description

`mxCopyPtrToComplex16` copies `n` `COMPLEX*16` values from the MATLAB arrays pointed to by `pr` and `pi` into the Fortran `COMPLEX*16` array `y`. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyComplex16ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`, `mxGetData`, `mxGetImagData`

Purpose

Copy COMPLEX*8 values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToComplex8(pr, pi, y, n)
mwPointer pr, pi
complex*8 y(n)
mwSize n
```

Arguments

`pr`
: Pointer to the real data of a single-precision MATLAB array

`pi`
: Pointer to the imaginary data of a single-precision MATLAB array

`y`
: COMPLEX*8 Fortran array

`n`
: Number of elements to copy

Description

mxCopyPtrToComplex8 copies n COMPLEX*8 values from the MATLAB arrays pointed to by pr and pi into the Fortran COMPLEX*8 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

mxCopyComplex8ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyPtrToInteger1 (Fortran)

Purpose

Copy INTEGER*1 values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToInteger1(px, y, n)
mwPointer px
integer*1 y(n)
mwSize n
```

Arguments

px
: Pointer to ir or jc array

y
: INTEGER*1 Fortran array

n
: Number of elements to copy

Description

mxCopyPtrToInteger1 copies n INTEGER*1 values from the MATLAB array pointed to by px, either an ir or jc array, into the Fortran INTEGER*1 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

mxCopyInteger1ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

Purpose

Copy INTEGER*2 values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToInteger2(px, y, n)
mwPointer px
integer*2 y(n)
mwSize n
```

Arguments

`px`
: Pointer to `ir` or `jc` array

`y`
: `INTEGER*2` Fortran array

`n`
: Number of elements to copy

Description

`mxCopyPtrToInteger2` copies `n` `INTEGER*2` values from the MATLAB array pointed to by `px`, either an `ir` or `jc` array, into the Fortran `INTEGER*2` array `y`. This subroutine is essential for use with Fortran compilers that do not support the `%VAL` construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

`mxCopyInteger2ToPtr`, `mxCreateNumericArray`, `mxCreateNumericMatrix`

mxCopyPtrToInteger4 (Fortran)

Purpose

Copy INTEGER*4 values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToInteger4(px, y, n)
mwPointer px
integer*4 y(n)
mwSize n
```

Arguments

px
: Pointer to ir or jc array

y
: INTEGER*4 Fortran array

n
: Number of elements to copy

Description

mxCopyPtrToInteger4 copies n INTEGER*4 values from the MATLAB array pointed to by px, either an ir or jc array, into the Fortran INTEGER*4 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

mxCopyInteger4ToPtr, mxCreateNumericArray, mxCreateNumericMatrix

Purpose

Copy pointer values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToPtrArray(px, y, n)
mwPointer px
mwPointer y(n)
mwSize n
```

Arguments

`px`
: Pointer to pointer array

`y`
: Fortran array of `mwPointer` values

`n`
: Number of pointers to copy

Description

`mxCopyPtrToPtrArray` copies `n` pointers from the MATLAB array pointed to by `px` into the Fortran array `y`. This subroutine is essential for copying the output of `matGetDir` into an array of pointers. After calling this function, each element of `y` contains a pointer to a string. You can convert these strings to Fortran character arrays by passing each element of `y` as the first argument to `mxCopyPtrToCharacter`.

Examples

See `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

`matGetDir`, `mxCopyPtrToCharacter`

mxCopyPtrToReal4 (Fortran)

Purpose

Copy REAL*4 values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToReal4(px, y, n)
mwPointer px
real*4 y(n)
mwSize n
```

Arguments

px
: Pointer to the real or imaginary data of a single-precision MATLAB array

y
: REAL*4 Fortran array

n
: Number of elements to copy

Description

mxCopyPtrToReal4 copies n REAL*4 values from the MATLAB array pointed to by px, either a pr or pi array, into the Fortran REAL*4 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

mxCopyReal4ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

Purpose

Copy REAL*8 values from pointer array to Fortran array

Fortran Syntax

```
mxCopyPtrToReal8(px, y, n)
mwPointer px
real*8 y(n)
mwSize n
```

Arguments

px
: Pointer to the real or imaginary data of a double-precision MATLAB array

y
: REAL*8 Fortran array

n
: Number of elements to copy

Description

mxCopyPtrToReal8 copies n REAL*8 values from the MATLAB array pointed to by px, either a pr or pi array, into the Fortran REAL*8 array y. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Examples

See fengdemo.F in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

mxCopyReal8ToPtr, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCopyReal4ToPtr (Fortran)

Purpose

Copy REAL*4 values from Fortran array to pointer array

Fortran Syntax

```
mxCopyReal4ToPtr(y, px, n)
real*4 y(n)
mwPointer px
mwSize n
```

Arguments

y
: REAL*4 Fortran array

px
: Pointer to the real or imaginary data of a single-precision MATLAB array

n
: Number of elements to copy

Description

mxCopyReal4ToPtr copies n REAL*4 values from the Fortran REAL*4 array y into the MATLAB array pointed to by px, either a pr or pi array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

See Also

mxCopyPtrToReal4, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

Purpose

Copy REAL*8 values from Fortran array to pointer array

Fortran Syntax

```
mxCopyReal8ToPtr(y, px, n)
real*8 y(n)
mwPointer px
mwSize n
```

Arguments

y
: REAL*8 Fortran array

px
: Pointer to the real or imaginary data of a double-precision MATLAB array

n
: Number of elements to copy

Description

mxCopyReal8ToPtr copies n REAL*8 values from the Fortran REAL*8 array y into the MATLAB array pointed to by px, either a pr or pi array. This subroutine is essential for use with Fortran compilers that do not support the %VAL construct in order to set up standard Fortran arrays for passing as arguments to the computation routine of a MEX-file.

Examples

See matdemo1.F and fengdemo.F in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

mxCopyPtrToReal8, mxCreateNumericArray, mxCreateNumericMatrix, mxGetData, mxGetImagData

mxCreateCellArray (C and Fortran)

Purpose

Create unpopulated N-D cell mxArray

C Syntax

```
#include "matrix.h"
mxArray *mxCreateCellArray(mwSize ndim, const mwSize *dims);
```

Fortran Syntax

```
mwPointer mxCreateCellArray(ndim, dims)
mwSize ndim, dims
```

Arguments

ndim

The desired number of dimensions in the created cell. For example, to create a three-dimensional cell mxArray, set ndim to 3.

dims

The dimensions array. Each element in the dimensions array contains the size of the mxArray in that dimension. For example, in C, setting dims[0] to 5 and dims[1] to 7 establishes a 5-by-7 mxArray. In Fortran, setting dims(1) to 5 and dims(2) to 7 establishes a 5-by-7 mxArray. In most cases, there should be ndim elements in the dims array.

Returns

A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateCellArray returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. The most common cause of failure is insufficient free heap space.

Description

Use mxCreateCellArray to create a cell mxArray whose size is defined by ndim and dims. For example, in C, to establish a three-dimensional cell mxArray having dimensions 4-by-8-by-7, set

```
ndim = 3;
dims[0] = 4; dims[1] = 8; dims[2] = 7;
```

In Fortran, to establish a three-dimensional cell mxArray having dimensions 4-by-8-by-7, set

```
ndim = 3;
```

```
dims(1) = 4; dims(2) = 8; dims(3) = 7;
```

The created cell `mxArray` is unpopulated; `mxCreateCellArray` initializes each cell to `NULL`. To put data into a cell, call `mxSetCell`.

Any trailing singleton dimensions specified in the `dims` argument are automatically removed from the resulting array. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, the resulting array is given the dimensions 4-by-1-by-7.

C Examples

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mxCreateCellMatrix`, `mxGetCell`, `mxSetCell`, `mxIsCell`

mxCreateCellMatrix (C and Fortran)

Purpose

Create unpopulated 2-D cell mxArray

C Syntax

```
#include "matrix.h"
mxArray *mxCreateCellMatrix(mwSize m, mwSize n);
```

Fortran Syntax

```
mwPointer mxCreateCellMatrix(m, n)
mwSize m, n
```

Arguments

m
: The desired number of rows

n
: The desired number of columns

Returns

A pointer to the created cell mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateCellMatrix returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the only reason for mxCreateCellMatrix to be unsuccessful.

Description

Use mxCreateCellMatrix to create an m-by-n two-dimensional cell mxArray. The created cell mxArray is unpopulated; mxCreateCellMatrix initializes each cell to NULL in C (0 in Fortran). To put data into cells, call mxSetCell.

mxCreateCellMatrix is identical to mxCreateCellArray except that mxCreateCellMatrix can create two-dimensional mxArrays only, but mxCreateCellArray can create mxArrays having any number of dimensions greater than 1.

C Examples

See mxcreatecellmatrix.c in the mx subdirectory of the examples directory.

See Also

mxCreateCellArray

Purpose

Create unpopulated N-D string mxArray

C Syntax

```
#include "matrix.h"
mxArray *mxCreateCharArray(mwSize ndim, const mwSize *dims);
```

Fortran Syntax

```
mwPointer mxCreateCharArray(ndim, dims)
mwSize ndim, dims
```

Arguments

`ndim`

The desired number of dimensions in the string mxArray. You must specify a positive number. If you specify 0, 1, or 2, mxCreateCharArray creates a two-dimensional mxArray.

`dims`

The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting `dims[0]` to `5` and `dims[1]` to `7` establishes a 5-by-7 mxArray. In Fortran, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a 5-by-7 character mxArray. The dims array must have at least ndim elements.

Returns

A pointer to the created string mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateCharArray returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the only reason for mxCreateCharArray to be unsuccessful.

Description

Call mxCreateCharArray to create an N-dimensional string mxArray. The created mxArray is unpopulated; that is, mxCreateCharArray initializes each cell to `NULL` in C (`0` in Fortran).

Any trailing singleton dimensions specified in the dims argument are automatically removed from the resulting array. For example, if ndim equals `5` and dims equals `[4 1 7 1 1]`, the resulting array is given the dimensions 4-by-1-by-7.

C Examples

See `mxcreatecharmatrixfromstr.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCreateCharMatrixFromStrings`, `mxCreateString`

Purpose

Create populated 2-D string mxArray

C Syntax

```
#include "matrix.h"
mxArray *mxCreateCharMatrixFromStrings(mwSize m, const char **str);
```

Fortran Syntax

```
mwPointer mxCreateCharMatrixFromStrings(m, str)
mwSize m
character*(*) str(m)
```

Arguments

`m`

The desired number of rows in the created string mxArray. The value you specify for m should equal the number of strings in str.

`str`

In C, an array of strings containing at least m strings. In Fortran, a character*n array of size m, where each element of the array is n bytes.

Returns

A pointer to the created string mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateCharMatrixFromStrings returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. Insufficient free heap space is the primary reason for mxCreateCharMatrixFromStrings to be unsuccessful. Another possible reason for failure is that str contains fewer than m strings.

Description

Use mxCreateCharMatrixFromStrings to create a two-dimensional string mxArray, where each row is initialized to a string from str. In C, the created mxArray has dimensions m-by-max, where max is the length of the longest string in str. In Fortran, the created mxArray has dimensions m-by-n, where n is the number of characters in str(i).

Note that string mxArrays represent their data elements as mxChar rather than as C char.

C Examples

See `mxcreatecharmatrixfromstr.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCreateCharArray`, `mxCreateString`, `mxGetString`

Purpose

Create 2-D, double-precision, floating-point mxArray initialized to 0

C Syntax

```
#include "matrix.h"
mxArray *mxCreateDoubleMatrix(mwSize m, mwSize n,
  mxComplexity ComplexFlag);
```

Fortran Syntax

```
mwPointer mxCreateDoubleMatrix(m, n, ComplexFlag)
mwSize m, n
integer*4 ComplexFlag
```

Arguments

m
: The desired number of rows

n
: The desired number of columns

ComplexFlag
: Specify either mxREAL or mxCOMPLEX. If the data you plan to put into the mxArray has no imaginary components, specify mxREAL in C (0 in Fortran). If the data has some imaginary components, specify mxCOMPLEX in C (1 in Fortran).

Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateDoubleMatrix returns NULL in C (0 in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateDoubleMatrix is unsuccessful when there is not enough free heap space to create the mxArray.

Description

Use mxCreateDoubleMatrix to create an m-by-n mxArray. mxCreateDoubleMatrix initializes each element in the pr array to 0. If you set ComplexFlag to mxCOMPLEX in C (1 in Fortran), mxCreateDoubleMatrix also initializes each element in the pi array to 0.

If you set ComplexFlag to mxREAL in C (0 in Fortran), mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements. If you set ComplexFlag to mxCOMPLEX in C (1 in Fortran),

mxCreateDoubleMatrix allocates enough memory to hold m-by-n real elements and m-by-n imaginary elements.

Call mxDestroyArray when you finish using the mxArray. mxDestroyArray deallocates the mxArray and its associated real and complex elements.

C Examples

See convec.c, findnz.c, sincall.c, timestwo.c, timestwoalt.c, and xtimesy.c in the refbook subdirectory of the examples directory.

See Also

mxCreateNumericArray

Purpose

Create scalar, double-precision array initialized to specified value

C Syntax

```
#include "matrix.h"
mxArray *mxCreateDoubleScalar(double value);
```

Fortran Syntax

```
mwPointer mxCreateDoubleScalar(value)
real*8 value
```

Arguments

`value`
The desired value to which you want to initialize the array

Returns

A pointer to the created `mxArray`, if successful. `mxCreateDoubleScalar` is unsuccessful if there is not enough free heap space to create the `mxArray`. If `mxCreateDoubleScalar` is unsuccessful in a MEX-file, the MEX-file prints an "Out of Memory" message, terminates, and control returns to the MATLAB prompt. If `mxCreateDoubleScalar` is unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateDoubleScalar` returns `NULL` in C (`0` in Fortran).

Description

Call `mxCreateDoubleScalar` to create a scalar double `mxArray`. `mxCreateDoubleScalar` is a convenience function that can be used in place of the following C code:

```
pa = mxCreateDoubleMatrix(1, 1, mxREAL);
*mxGetPr(pa) = value;
```

`mxCreateDoubleScalar` can be used in place of the following Fortran code:

```
pm = mxCreateDoubleMatrix(1, 1, 0)
mxCopyReal8ToPtr(value, mxGetPr(pm), 1)
```

When you finish using the `mxArray`, call `mxDestroyArray` to destroy it.

See Also

`mxGetPr`, `mxCreateDoubleMatrix`

mxCreateLogicalArray (C)

Purpose

Create N-D logical mxArray initialized to false

C Syntax

```
#include "matrix.h"
mxArray *mxCreateLogicalArray(mwSize ndim, const mwSize *dims);
```

Arguments

ndim

Number of dimensions. If you specify a value for ndim that is less than 2, mxCreateLogicalArray automatically sets the number of dimensions to 2.

dims

The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, setting dims[0] to 5 and dims[1] to 7 establishes a 5-by-7 mxArray. There should be ndim elements in the dims array.

Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateLogicalArray returns NULL. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateLogicalArray is unsuccessful when there is not enough free heap space to create the mxArray.

Description

Call mxCreateLogicalArray to create an N-dimensional mxArray of mxLogical elements. After creating the mxArray, mxCreateLogicalArray initializes all its elements to logical 0. mxCreateLogicalArray differs from mxCreateLogicalMatrix in that the latter can create two-dimensional arrays only.

mxCreateLogicalArray allocates dynamic memory to store the created mxArray. When you finish with the created mxArray, call mxDestroyArray to deallocate its memory.

Any trailing singleton dimensions specified in the dims argument are automatically removed from the resulting array. For example, if ndim equals 5 and dims equals [4 1 7 1 1], the resulting array is given the dimensions 4-by-1-by-7.

See Also

mxCreateLogicalMatrix, mxCreateSparseLogicalMatrix, mxCreateLogicalScalar

mxCreateLogicalMatrix (C)

Purpose

Create 2-D, logical mxArray initialized to false

C Syntax

```
#include "matrix.h"
mxArray *mxCreateLogicalMatrix(mwSize m, mwSize n);
```

Arguments

m
: The desired number of rows

n
: The desired number of columns

Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, mxCreateLogicalMatrix returns NULL. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateLogicalMatrix is unsuccessful when there is not enough free heap space to create the mxArray.

Description

Use mxCreateLogicalMatrix to create an m-by-n mxArray of mxLogical elements. mxCreateLogicalMatrix initializes each element in the array to logical 0.

Call mxDestroyArray when you finish using the mxArray. mxDestroyArray deallocates the mxArray.

See Also

mxCreateLogicalArray, mxCreateSparseLogicalMatrix, mxCreateLogicalScalar

Purpose

Create scalar, logical mxArray initialized to false

C Syntax

```
#include "matrix.h"
mxArray *mxCreateLogicalScalar(mxLogical value);
```

Arguments

value
: The desired logical value to which you want to initialize the array

Returns

A pointer to the created mxArray, if successful. mxCreateLogicalScalar is unsuccessful if there is not enough free heap space to create the mxArray. If mxCreateLogicalScalar is unsuccessful in a MEX-file, the MEX-file prints an "Out of Memory" message, terminates, and control returns to the MATLAB prompt. If mxCreateLogicalScalar is unsuccessful in a stand-alone (non-MEX-file) application, the function returns NULL.

Description

Call mxCreateLogicalScalar to create a scalar logical mxArray. mxCreateLogicalScalar is a convenience function that can be used in place of the following code:

```
pa = mxCreateLogicalMatrix(1, 1);
*mxGetLogicals(pa) = value;
```

When you finish using the mxArray, call mxDestroyArray to destroy it.

See Also

mxCreateLogicalArray, mxCreateLogicalMatrix, mxIsLogicalScalar, mxIsLogicalScalarTrue, mxGetLogicals

mxCreateNumericArray (C and Fortran)

Purpose

Create unpopulated N-D numeric mxArray

C Syntax

```
#include "matrix.h"
mxArray *mxCreateNumericArray(mwSize ndim, const mwSize *dims,
         mxClassID classid, mxComplexity ComplexFlag);
```

Fortran Syntax

```
mwPointer mxCreateNumericArray(ndim, dims, classid,
  ComplexFlag)
mwSize ndim, dims
integer*4 classid, ComplexFlag
```

Arguments

`ndim`

Number of dimensions. If you specify a value for `ndim` that is less than 2, `mxCreateNumericArray` automatically sets the number of dimensions to 2.

`dims`

The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting `dims[0]` to 5 and `dims[1]` to 7 establishes a 5-by-7 `mxArray`. In Fortran, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a 5-by-7 `mxArray`. In most cases, there should be `ndim` elements in the `dims` array.

`classid`

An identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` in C causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer. In Fortran, use the function `mxClassIDFromClassName` to derive the `classid` value from a MATLAB class name. See the Description section for more information.

`ComplexFlag`

If the data you plan to put into the `mxArray` has no imaginary components, specify `mxREAL` in C (`0` in Fortran). If the data has some imaginary components, specify `mxCOMPLEX` in C (`1` in Fortran).

Returns

A pointer to the created `mxArray`, if successful. If unsuccessful in a stand-alone (non-MEX-file) application, `mxCreateNumericArray` returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. `mxCreateNumericArray` is unsuccessful when there is not enough free heap space to create the `mxArray`.

Description

Call `mxCreateNumericArray` to create an N-dimensional `mxArray` in which all data elements have the numeric data type specified by `classid`. After creating the `mxArray`, `mxCreateNumericArray` initializes all its real data elements to 0. If `ComplexFlag` equals `mxCOMPLEX` in C (`1` in Fortran), `mxCreateNumericArray` also initializes all its imaginary data elements to 0. `mxCreateNumericArray` differs from `mxCreateDoubleMatrix` in two important respects:

- All data elements in `mxCreateDoubleMatrix` are double-precision, floating-point numbers. The data elements in `mxCreateNumericArray` could be any numerical type, including different integer precisions.
- `mxCreateDoubleMatrix` can create two-dimensional arrays only; `mxCreateNumericArray` can create arrays of two or more dimensions.

`mxCreateNumericArray` allocates dynamic memory to store the created `mxArray`. When you finish with the created `mxArray`, call `mxDestroyArray` to deallocate its memory.

Any trailing singleton dimensions specified in the `dims` argument are automatically removed from the resulting array. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, the resulting array is given the dimensions 4-by-1-by-7.

The following table shows the C `classid` values and the Fortran data types that are equivalent to MATLAB classes.

MATLAB Class Name	C classid Value	Fortran Type
int8	mxINT8_CLASS	BYTE
uint8	mxUINT8_CLASS	
int16	mxUINT16_CLASS	INTEGER*2
uint16	mxUINT16_CLASS	
int32	mxINT32_CLASS	INTEGER*4
uint32	mxUINT32_CLASS	
int64	mxINT64_CLASS	INTEGER*8
uint64	mxUINT64_CLASS	
single	mxSINGLE_CLASS	REAL*4
double	mxDOUBLE_CLASS	REAL*8
single, with imaginary components	mxSINGLE_CLASS	COMPLEX*8
double, with imaginary components	mxDOUBLE_CLASS	COMPLEX*16

C Examples

See phonebook.c and doubleelement.c in the refbook subdirectory of the examples directory. For an additional example, see mxisfinite.c in the mx subdirectory of the examples directory.

Fortran Examples

To create a 4-by-4-by-2 array of REAL*8 elements having no imaginary components, use

```
C       Create 4x4x2 mxArray of REAL*8
        data dims / 4, 4, 2 /
        mxCreateNumericArray(3, dims,
     +                mxClassIDFromClassName('double'), 0)
```

See Also

mxClassId, mxClassIdFromClassName, mxComplexity, mxCreateNumericMatrix

mxCreateNumericMatrix (C and Fortran)

Purpose

Create numeric matrix and initialize data elements to 0

C Syntax

```
#include "matrix.h"
mxArray *mxCreateNumericMatrix(mwSize m, mwSize n,
  mxClassID classid, mxComplexity ComplexFlag);
```

Fortran Syntax

```
mwPointer mxCreateNumericMatrix(m, n, classid,
  ComplexFlag)
mwSize m, n
integer*4 classid, ComplexFlag
```

Arguments

`m`
: The desired number of rows.

`n`
: The desired number of columns.

`classid`
: An identifier for the class of the array, which determines the way the numerical data is represented in memory. For example, specifying `mxINT16_CLASS` in C causes each piece of numerical data in the `mxArray` to be represented as a 16-bit signed integer. In Fortran, use the function `mxClassIDFromClassName` to derive the `classid` value from a MATLAB class name. See the Description section for more information.

`ComplexFlag`
: If the data you plan to put into the `mxArray` has no imaginary components, specify `mxREAL` in C (`0` in Fortran). If the data has some imaginary components, specify `mxCOMPLEX` in C (`1` in Fortran).

Returns

A pointer to the created `mxArray`, if successful. `mxCreateNumericMatrix` is unsuccessful if there is not enough free heap space to create the `mxArray`. If `mxCreateNumericMatrix` is unsuccessful in a MEX-file, the MEX-file prints an “Out of Memory” message, terminates, and control returns to the MATLAB prompt. If `mxCreateNumericMatrix`

is unsuccessful in a stand-alone (nonMEX-file) application, `mxCreateNumericMatrix` returns `NULL` in C (`0` in Fortran).

Description

Call `mxCreateNumericMatrix` to create a 2-D `mxArray` in which all data elements have the numeric data type specified by `classid`. After creating the `mxArray`, `mxCreateNumericMatrix` initializes all its real data elements to 0. If `ComplexFlag` equals `mxCOMPLEX` in C (`1` in Fortran), `mxCreateNumericMatrix` also initializes all its imaginary data elements to 0. `mxCreateNumericMatrix` allocates dynamic memory to store the created `mxArray`. When you finish using the `mxArray`, call `mxDestroyArray` to destroy it.

The following table shows the C `classid` values and the Fortran data types that are equivalent to MATLAB classes.

MATLAB Class Name	C `classid` Value	Fortran Type
int8	mxINT8_CLASS	BYTE
uint8	mxUINT8_CLASS	
int16	mxUINT16_CLASS	INTEGER*2
uint16	mxUINT16_CLASS	
int32	mxINT32_CLASS	INTEGER*4
uint32	mxUINT32_CLASS	
int64	mxINT64_CLASS	INTEGER*8
uint64	mxUINT64_CLASS	
single	mxSINGLE_CLASS	REAL*4
double	mxDOUBLE_CLASS	REAL*8

MATLAB Class Name	C classid Value	Fortran Type
single, with imaginary components	mxSINGLE_CLASS	COMPLEX*8
double, with imaginary components	mxDOUBLE_CLASS	COMPLEX*16

Fortran Examples

To create a 4-by-3 matrix of REAL*4 elements having no imaginary components, use

```
C      Create 4x3 mxArray of REAL*4
       mxCreateNumericMatrix(4, 3,
     +                mxClassIDFromClassName('single'), 0)
```

See Also

mxClassId, mxClassIdFromClassName, mxComplexity, mxCreateNumericArray

Purpose

Create 2-D unpopulated sparse mxArray

C Syntax

```
#include "matrix.h"
mxArray *mxCreateSparse(mwSize m, mwSize n, mwSize nzmax,
         mxComplexity ComplexFlag);
```

Fortran Syntax

```
mwPointer mxCreateSparse(m, n, nzmax, ComplexFlag)
mwSize m, n, nzmax
integer*4 ComplexFlag
```

Arguments

`m`
: The desired number of rows

`n`
: The desired number of columns

`nzmax`
: The number of elements that `mxCreateSparse` should allocate to hold the `pr`, `ir`, and, if `ComplexFlag` is `mxCOMPLEX` in C (`1` in Fortran), `pi` arrays. Set the value of `nzmax` to be greater than or equal to the number of nonzero elements you plan to put into the `mxArray`, but make sure that `nzmax` is less than or equal to `m*n`.

`ComplexFlag`
: If the `mxArray` you are creating is to contain imaginary data, set `ComplexFlag` to `mxCOMPLEX` in C (`1` in Fortran). Otherwise, set `ComplexFlag` to `mxREAL` in C (`0` in Fortran).

Returns

A pointer to the created sparse double `mxArray` if successful, and `NULL` in C (`0` in Fortran) otherwise. The most likely reason for failure is insufficient free heap space. If that happens, try reducing `nzmax`, `m`, or `n`.

Description

Call `mxCreateSparse` to create an unpopulated sparse double `mxArray`. The returned sparse `mxArray` contains no sparse information and cannot be passed as an argument to any MATLAB sparse functions. To make the returned sparse `mxArray` useful, you must initialize the `pr`, `ir`, `jc`, and (if it exists) `pi` arrays.

mxCreateSparse allocates space for

- A pr array of length nzmax.
- A pi array of length nzmax, but only if ComplexFlag is mxCOMPLEX in C (1 in Fortran).
- An ir array of length nzmax.
- A jc array of length n+1.

When you finish using the sparse mxArray, call mxDestroyArray to reclaim all its heap space.

C Examples

See fulltosparse.c in the refbook subdirectory of the examples directory.

See Also

mxDestroyArray, mxSetNzmax, mxSetPr, mxSetPi, mxSetIr, mxSetJc, mxComplexity

Purpose

Create unpopulated 2-D, sparse, logical mxArray

C Syntax

```
#include "matrix.h"
mxArray *mxCreateSparseLogicalMatrix(mwSize m, mwSize n,
  mwSize nzmax);
```

Arguments

m
: The desired number of rows

n
: The desired number of columns

nzmax
: The number of elements that mxCreateSparseLogicalMatrix should allocate to hold the data. Set the value of nzmax to be greater than or equal to the number of nonzero elements you plan to put into the mxArray, but make sure that nzmax is less than or equal to m*n.

Returns

A pointer to the created mxArray, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, mxCreateSparseLogicalMatrix returns NULL. If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt. mxCreateSparseLogicalMatrix is unsuccessful when there is not enough free heap space to create the mxArray.

Description

Use mxCreateSparseLogicalMatrix to create an m-by-n mxArray of mxLogical elements. mxCreateSparseLogicalMatrix initializes each element in the array to logical 0.

Call mxDestroyArray when you finish using the mxArray. mxDestroyArray deallocates the mxArray and its elements.

See Also

mxCreateLogicalArray, mxCreateLogicalMatrix, mxCreateLogicalScalar, mxCreateSparse, mxIsLogical

mxCreateString (C and Fortran)

Purpose

Create 1-by-N string mxArray initialized to specified string

C Syntax

```
#include "matrix.h"
mxArray *mxCreateString(const char *str);
```

Fortran Syntax

```
mwPointer mxCreateString(str)
character*(*) str
```

Arguments

str

The string that is to serve as the mxArray's initial data

Returns

A pointer to the created string mxArray if successful, and NULL in C (0 in Fortran) otherwise. The most likely cause of failure is insufficient free heap space.

Description

Use mxCreateString to create a string mxArray initialized to str. Many MATLAB functions (for example, strcmp and upper) require string array inputs.

Free the string mxArray when you are finished using it. To free a string mxArray, call mxDestroyArray.

C Examples

See revord.c in the refbook subdirectory of the examples directory.

For additional examples, see mxcreatestructarray.c and mxisclass.c in the mx subdirectory of the examples directory.

Fortran Examples

See matdemo1.F in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

mxCreateCharMatrixFromStrings, mxCreateCharArray

Purpose

Create unpopulated N-D structure `mxArray`

C Syntax

```
#include "matrix.h"
mxArray *mxCreateStructArray(mwSize ndim, const mwSize *dims,
  int nfields, const char **fieldnames);
```

Fortran Syntax

```
mwPointer mxCreateStructArray(ndim, dims, nfields,
  fieldnames)
mwSize ndim, dims
integer*4 nfields
character*(*) fieldnames(nfields)
```

Arguments

`ndim`
: Number of dimensions. If you set `ndim` to be less than `2`, `mxCreateStructArray` creates a two-dimensional `mxArray`.

`dims`
: The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting `dims[0]` to `5` and `dims[1]` to `7` establishes a 5-by-7 `mxArray`. In Fortran, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a 5-by-7 `mxArray`. Typically, the `dims` array should have `ndim` elements.

`nfields`
: The desired number of fields in each element

`fieldnames`
: The desired list of field names

Each structure field name must begin with a letter and is case sensitive. The rest of the name may contain letters, numerals, and underscore characters. Use the `namelengthmax` function to determine the maximum length of a field name.

Returns

A pointer to the created structure `mxArray` if successful, and `NULL` in C (`0` in Fortran) otherwise. The most likely cause of failure is insufficient heap space to hold the returned `mxArray`.

Description

Call `mxCreateStructArray` to create an unpopulated structure `mxArray`. Each element of a structure `mxArray` contains the same number of fields (specified in `nfields`). Each field has a name; the list of names is specified in `fieldnames`. A structure `mxArray` in MATLAB is conceptually identical to an array of `structs` in the C language.

Each field holds one `mxArray` pointer. `mxCreateStructArray` initializes each field to `NULL` in C (`0` in Fortran). Call `mxSetField` or `mxSetFieldByNumber` to place a non-`NULL` `mxArray` pointer in a field.

When you finish using the returned structure `mxArray`, call `mxDestroyArray` to reclaim its space.

Any trailing singleton dimensions specified in the `dims` argument are automatically removed from the resulting array. For example, if `ndim` equals `5` and `dims` equals `[4 1 7 1 1]`, the resulting array is given the dimensions 4-by-1-by-7.

C Examples

See `mxcreatestructarray.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxDestroyArray`, `mxAddField`, `mxRemoveField`, `mxSetField`, `mxSetFieldByNumber`

Purpose

Create unpopulated 2-D structure mxArray

C Syntax

```
#include "matrix.h"
mxArray *mxCreateStructMatrix(mwSize m, mwSize n, int nfields,
         const char **fieldnames);
```

Fortran Syntax

```
mwPointer mxCreateStructMatrix(m, n, nfields, fieldnames)
mwSize m, n
integer*4 nfields
character*(*) fieldnames(nfields)
```

Arguments

`m`
: The desired number of rows. This must be a positive integer.

`n`
: The desired number of columns. This must be a positive integer.

`nfields`
: The desired number of fields in each element.

`fieldnames`
: The desired list of field names.

Each structure field name must begin with a letter and is case sensitive. The rest of the name may contain letters, numerals, and underscore characters. Use the `namelengthmax` function to determine the maximum length of a field name.

Returns

A pointer to the created structure `mxArray` if successful, and `NULL` in C (`0` in Fortran) otherwise. The most likely cause of failure is insufficient heap space to hold the returned `mxArray`.

Description

`mxCreateStructMatrix` and `mxCreateStructArray` are almost identical. The only difference is that `mxCreateStructMatrix` can create only two-dimensional `mxArrays`, while `mxCreateStructArray` can create `mxArrays` having two or more dimensions.

C Examples

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mxCreateStructArray`

Purpose

Free dynamic memory allocated by mxCreate* functions

C Syntax

```
#include "matrix.h"
void mxDestroyArray(mxArray *pm);
```

Fortran Syntax

```
mxDestroyArray(pm)
mwPointer pm
```

Arguments

pm

Pointer to the mxArray you want to free

Description

mxDestroyArray deallocates the memory occupied by the specified mxArray. mxDestroyArray not only deallocates the memory occupied by the mxArray's characteristics fields (such as m and n), but also deallocates all the mxArray's associated data arrays, such as pr and pi for complex arrays, ir and jc for sparse arrays, fields of structure arrays, and cells of cell arrays. Do not call mxDestroyArray on an mxArray you are returning on the left-hand side.

C Examples

See sincall.c in the refbook subdirectory of the examples directory.

Additional examples:

- mexcallmatlab.c and mexgetarray.c in the mex subdirectory of the examples directory
- mxisclass.c in the mx subdirectory of the examples directory

See Also

mxCalloc, mxMalloc, mxFree, mexMakeArrayPersistent, mexMakeMemoryPersistent

mxDuplicateArray (C and Fortran)

Purpose

Make deep copy of array

C Syntax

```
#include "matrix.h"
mxArray *mxDuplicateArray(const mxArray *in);
```

Fortran Syntax

```
mwPointer mxDuplicateArray(in)
mwPointer in
```

Arguments

`in`

Pointer to the `mxArray` you want to copy

Returns

Pointer to a copy of the array.

Description

`mxDuplicateArray` makes a deep copy of an array, and returns a pointer to the copy. A deep copy refers to a copy in which all levels of data are copied. For example, a deep copy of a cell array copies each cell and the contents of each cell (if any), and so on.

C Examples

See

- `mexget.c` in the `mex` subdirectory of the `examples` directory
- `phonebook.c` in the `refbook` subdirectory of the `examples` directory

For additional examples, see `mxcreatecellmatrix.c`, `mxgetinf.c`, and `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory.

Purpose

Free dynamic memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`

C Syntax

```
#include "matrix.h"
void mxFree(void *ptr);
```

Fortran Syntax

```
mxFree(ptr)
mwPointer ptr
```

Arguments

`ptr`

Pointer to the beginning of any memory parcel allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`.

Description

`mxFree` deallocates heap space using the MATLAB memory management facility. This ensures correct memory management in error and abort (**Ctrl+C**) conditions.

To deallocate heap space, MATLAB applications in C should always call `mxFree` rather than the ANSI C `free` function.

The MATLAB memory management facility maintains a list of all memory allocated by `mxCalloc`, `mxMalloc`, `mxRealloc`, and the `mxCreate*` calls. The MATLAB memory management facility automatically deallocates all of a MEX-file's managed parcels when the MEX-file completes and control returns to the MATLAB prompt.

When `mxFree` appears in a stand-alone MATLAB application, `mxFree` simply deallocates the contiguous heap space that begins at address `ptr`. In a MEX-file, `mxFree` also removes the memory parcel from the MATLAB memory management facility's list of memory parcels.

In a MEX-file, your use of `mxFree` depends on whether the specified memory parcel is persistent or nonpersistent. By default, memory parcels created by `mxCalloc`, `mxMalloc`, and `mxRealloc` are nonpersistent. The MATLAB memory management facility automatically frees all nonpersistent memory whenever a MEX-file completes. Thus, even if you do not call `mxFree`, MATLAB takes care of freeing the memory for you. Nevertheless, it is good programming

practice to deallocate memory as soon as you are through using it. Doing so generally makes the entire system run more efficiently.

If an application calls `mexMakeMemoryPersistent`, the specified memory parcel becomes persistent. When a MEX-file completes, the MATLAB memory management facility does not free persistent memory parcels. Therefore, the only way to free a persistent memory parcel is to call `mxFree`. Typically, MEX-files call `mexAtExit` to register a cleanup handler. The cleanup handler calls `mxFree`.

C Examples

See `mxcalcsinglesubscript.c` in the `mx` subdirectory of the `examples` directory.

Additional examples:

- `phonebook.c` in the `refbook` subdirectory of the `examples` directory
- `explore.c` and `mexatexit.c` in the `mex` subdirectory of the `examples` directory
- `mxcreatecharmatrixfromstr.c`, `mxisfinite.c`, `mxmalloc.c`, and `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory

See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxMalloc`, `mxRealloc`

Purpose

Get contents of mxArray cell

C Syntax

```
#include "matrix.h"
mxArray *mxGetCell(const mxArray *pm, mwIndex index);
```

Fortran Syntax

```
mwPointer mxGetCell(pm, index)
mwPointer pm
mwIndex index
```

Arguments

`pm`
: Pointer to a cell mxArray

`index`
: The number of elements in the cell mxArray between the first element and the desired one. See mxCalcSingleSubscript for details on calculating an index in a multidimensional cell array.

Returns

A pointer to the ith cell mxArray if successful, and NULL in C (0 in Fortran) otherwise. Causes of failure include

- Specifying the index of a cell array element that has not been populated.
- Specifying a pm that does not point to a cell mxArray.
- Specifying an index greater than the number of elements in the cell.
- Insufficient free heap space to hold the returned cell mxArray.

Description

Call mxGetCell to get a pointer to the mxArray held in the indexed element of the cell mxArray.

Note Inputs to a MEX-file are constant read-only mxArrays and should not be modified. Using mxSetCell* or mxSetField* to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

C Examples

See `explore.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mxCreateCellArray`, `mxIsCell`, `mxSetCell`

Purpose

Get pointer to character array data

C Syntax

```
#include "matrix.h"
mxChar *mxGetChars(const mxArray *array_ptr);
```

Arguments

`array_ptr`
Pointer to an `mxArray`

Returns

The address of the first character in the `mxArray`. Returns `NULL` if the specified array is not a character array.

Description

Call `mxGetChars` to determine the address of the first character in the `mxArray` that `array_ptr` points to. Once you have the starting address, you can access any other element in the `mxArray`.

See Also

`mxGetString`

mxGetClassID (C and Fortran)

Purpose

Get class of mxArray

C Syntax

```
#include "matrix.h"
mxClassID mxGetClassID(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxGetClassID(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

A numeric identifier of the class (category) of the mxArray that pm points to. The C-language class identifiers are listed in the mxClassID reference page.

Description

Use mxGetClassId to determine the class of an mxArray. The class of an mxArray identifies the kind of data the mxArray is holding. For example, if pm points to a logical mxArray, then mxGetClassId returns mxLOGICAL_CLASS (in C).

mxGetClassId is similar to mxGetClassName, except that the former returns the class as an integer identifier and the latter returns the class as a string.

C Examples

See

- phonebook.c in the refbook subdirectory of the examples directory
- explore.c in the mex subdirectory of the examples directory

See Also

mxClassID, mxGetClassName

Purpose

Get class of mxArray as string

C Syntax

```
#include "matrix.h"
const char *mxGetClassName(const mxArray *pm);
```

Fortran Syntax

```
character*(*) mxGetClassName(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

The class (as a string) of the mxArray pointed to by pm.

Description

Call mxGetClassName to determine the class of an mxArray. The class of an mxArray identifies the kind of data the mxArray is holding. For example, if pm points to a logical mxArray, mxGetClassName returns logical.

mxGetClassID is similar to mxGetClassName, except that the former returns the class as an integer identifier, as listed in the mxClassID reference page, and the latter returns the class as a string, as listed in the mxIsClass reference page.

C Examples

See mexfunction.c in the mex subdirectory of the examples directory. For an additional example, see mxisclass.c in the mx subdirectory of the examples directory.

See Also

mxGetClassID, mxIsClass

mxGetData (C and Fortran)

Purpose

Get pointer to data

C Syntax

```
#include "matrix.h"
void *mxGetData(const mxArray *pm);
```

Fortran Syntax

```
mwPointer mxGetData(pm)
mwPointer pm
```

Arguments

`pm`
Pointer to an `mxArray`

Returns

The address of the first element of the real data. Returns `NULL` in C (`0` in Fortran) if there is no real data.

Description

Similar to `mxGetPr`, except that in C, `mxGetData` returns a `void *`.

To copy values from the returned pointer to Fortran, use one of the `mxCopyPtrTo*` functions in the following manner:

```
C       Get the data in mxArray, pm
        mxCopyPtrToReal8(mxGetData(pm), data,
     +                       mxGetNumberOfElements(pm))
```

C Examples

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

For additional examples, see `mxcreatecharmatrixfromstr.c` and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxGetImagData`, `mxGetPr`

Purpose

Get pointer to dimensions array

C Syntax

```
#include "matrix.h"
const mwSize *mxGetDimensions(const mxArray *pm);
```

Fortran Syntax

```
mwPointer mxGetDimensions(pm)
mwPointer pm
```

Arguments

`pm`
Pointer to an `mxArray`.

Returns

The address of the first element in the dimensions array. Each integer in the dimensions array represents the number of elements in a particular dimension. The array is not `NULL` terminated.

Description

Use `mxGetDimensions` to determine how many elements are in each dimension of the `mxArray` that `pm` points to. Call `mxGetNumberOfDimensions` to get the number of dimensions in the `mxArray`.

To copy the values to Fortran, use `mxCopyPtrToInteger4` in the following manner:

```
C      Get dimensions of mxArray, pm
       mxCopyPtrToInteger4(mxGetDimensions(pm), dims,
     +                     mxGetNumberOfDimensions(pm))
```

C Examples

See `mxcalcsinglesubscript.c` in the `mx` subdirectory of the `examples` directory.

Additional examples:

- `findnz.c` and `phonebook.c` in the `refbook` subdirectory of the `examples` directory
- `explore.c` in the `mex` subdirectory of the `examples` directory

- `mxgeteps.c` and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory

See Also

`mxGetNumberOfDimensions`

Purpose

Get number of bytes required to store each data element

C Syntax

```
#include "matrix.h"
mwSize mxGetElementSize(const mxArray *pm);
```

Fortran Syntax

```
mwSize mxGetElementSize(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

The number of bytes required to store one element of the specified `mxArray`, if successful. Returns 0 on failure. The primary reason for failure is that `pm` points to an `mxArray` having an unrecognized class. If `pm` points to a cell `mxArray` or a structure `mxArray`, `mxGetElementSize` returns the size of a pointer (not the size of all the elements in each cell or structure field).

Description

Call `mxGetElementSize` to determine the number of bytes in each data element of the `mxArray`. For example, if the MATLAB class of an `mxArray` is `int16`, the `mxArray` stores each data element as a 16-bit (2-byte) signed integer. Thus, `mxGetElementSize` returns `2`.

`mxGetElementSize` is particularly helpful when using a non-MATLAB routine to manipulate data elements. For example, the C function `memcpy` requires (for its third argument) the size of the elements you intend to copy.

C Examples

See `doubleelement.c` and `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mxGetM`, `mxGetN`

mxGetEps (C and Fortran)

Purpose

Get value of eps

C Syntax

```
#include "matrix.h"
double mxGetEps(void);
```

Fortran Syntax

```
real*8 mxGetEps
```

Returns

The value of the MATLAB eps variable

Description

Call mxGetEps to return the value of the MATLAB eps variable. This variable holds the distance from 1.0 to the next largest floating-point number. As such, it is a measure of floating-point accuracy. The MATLAB PINV and RANK functions use eps as a default tolerance.

C Examples

See mxgeteps.c in the mx subdirectory of the examples directory.

See Also

mxGetInf, mxGetNan

Purpose

Get field value, given field name and index into structure array

C Syntax

```
#include "matrix.h"
mxArray *mxGetField(const mxArray *pm, mwIndex index,
         const char *fieldname);
```

Fortran Syntax

```
mwPointer mxGetField(pm, index, fieldname)
mwPointer pm
mwIndex index
character*(*) fieldname
```

Arguments

`pm`
: Pointer to a structure `mxArray`

`index`
: The desired element. In C, the first element of an `mxArray` has an `index` of `0`, the second element has an `index` of `1`, and the last element has an `index` of `N-1`, where `N` is the total number of elements in the structure `mxArray`. In Fortran, the first element of an `mxArray` has an `index` of `1`, the second element has an `index` of `2`, and the last element has an `index` of `N`, where N is the total number of elements in the structure `mxArray`.

`fieldname`
: The name of the field whose value you want to extract.

Returns

A pointer to the `mxArray` in the specified field at the specified `fieldname`, on success. Returns `NULL` in C (`0` in Fortran) if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. To determine whether `pm` points to a structure `mxArray`, call `mxIsStruct`.
- Specifying an `index` to an element outside the bounds of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an `index` greater than `9` in C (`10` in Fortran).

- Specifying a nonexistent `fieldname`. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to get existing field names.
- Insufficient heap space to hold the returned `mxArray`.

Description

Call `mxGetField` to get the value held in the specified element of the specified field. In pseudo-C terminology, `mxGetField` returns the value at

```
pm[index].fieldname
```

`mxGetFieldByNumber` is similar to `mxGetField`. Both functions return the same value. The only difference is in the way you specify the field. `mxGetFieldByNumber` takes a field number as its third argument, and `mxGetField` takes a field name as its third argument.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

In C, calling

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "field_name");
mxGetFieldByNumber(pa, index, field_num);
```

where `index` is `0` if you have a 1-by-1 structure.

In Fortran, calling

```
mxGetField(pm, index, 'fieldname')
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxGetFieldByNumber(pm, index, fieldnum)
```

where `index` is `1` if you have a 1-by-1 structure.

See Also

`mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

mxGetFieldByNumber (C and Fortran)

Purpose

Get field value, given field number and index into structure array

C Syntax

```
#include "matrix.h"
mxArray *mxGetFieldByNumber(const mxArray *pm, mwIndex index,
         int fieldnumber);
```

Fortran Syntax

```
mwPointer mxGetFieldByNumber(pm, index, fieldnumber)
mwPointer pm
mwIndex index
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`index`

The desired element. In C, the first element of an `mxArray` has an `index` of `0`, the second element has an `index` of `1`, and the last element has an `index` of `N-1`, where `N` is the total number of elements in the structure `mxArray`. In Fortran, the first element of an `mxArray` has an `index` of `1`, the second element has an `index` of `2`, and the last element has an `index` of `N`, where `N` is the total number of elements in the structure `mxArray`. See `mxCalcSingleSubscript` for more details on calculating an `index`.

`fieldnumber`

The position of the field whose value you want to extract. In C, the first field within each element has a field number of `0`, the second field has a field number of `1`, and so on. The last field has a field number of `N-1`, where `N` is the number of fields. In Fortran, the first field within each element has a field number of `1`, the second field has a field number of `2`, and so on. The last field has a field number of `N`, where `N` is the number of fields.

Returns

A pointer to the `mxArray` in the specified field for the desired element, on success. Returns `NULL` in C (`0` in Fortran) if passed an invalid argument or if there is no value assigned to the specified field. Common causes of failure include

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStructto` determine whether `pm` points to a structure `mxArray`.
- Specifying an `index` to an element outside the bounds of the `mxArray`. For example, given a structure `mxArray` that contains 10 elements, you cannot specify an `index` greater than `9` in C (`10` in Fortran).
- Specifying a nonexistent field number. Call `mxGetFieldNumber` to determine the field number that corresponds to a given field name.

Description

Call `mxGetFieldByNumber` to get the value held in the specified `fieldnumber` at the indexed element.

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

In C, calling

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "field_name");
mxGetFieldByNumber(pa, index, field_num);
```

where `index` is `0` if you have a 1-by-1 structure.

In Fortran, calling

```
mxGetField(pm, index, 'fieldname')
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxGetFieldByNumber(pm, index, fieldnum)
```

where `index` is `1` if you have a 1-by-1 structure.

C Examples

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- `mxisclass.c` in the `mx` subdirectory of the `examples` directory
- `explore.c` in the `mex` subdirectory of the `examples` directory

See Also

`mxGetField`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

Purpose

Get field name, given field number in structure array

C Syntax

```
#include "matrix.h"
const char *mxGetFieldNameByNumber(const mxArray *pm,
            int fieldnumber);
```

Fortran Syntax

```
character*(*) mxGetFieldNameByNumber(pm, fieldnumber)
mwPointer pm
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`fieldnumber`

The position of the desired field. For instance, in C, to get the name of the first field, set `fieldnumber` to 0; to get the name of the second field, set `fieldnumber` to 1; and so on. In Fortran, to get the name of the first field, set `fieldnumber` to `1`; to get the name of the second field, set `fieldnumber` to `2`; and so on.

Returns

A pointer to the nth field name, on success. Returns `NULL` in C (`0` in Fortran) on failure. Common causes of failure include

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.
- Specifying a value of `fieldnumber` outside the bounds of the number of fields in the structure `mxArray`. In C, `fieldnumber` 0 represents the first field, and `fieldnumber` `N-1` represents the last field, where `N` is the number of fields in the structure `mxArray`. In Fortran, `fieldnumber` `1` represents the first field, and `fieldnumber` `N` represents the last field.

Description

Call `mxGetFieldNameByNumber` to get the name of a field in the given structure `mxArray`. A typical use of `mxGetFieldNameByNumber` is to call

it inside a loop in order to get the names of all the fields in a given `mxArray`.

Consider a MATLAB structure initialized to

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

In C, the field number `0` represents the field `name`; field number `1` represents field `billing`; field number `2` represents field `test`. A field number other than `0`, `1`, or `2` causes `mxGetFieldNameByNumber` to return `NULL`.

In Fortran, the field number `1` represents the field `name`; field number `2` represents field `billing`; field number `3` represents field `test`. A field number other than `1`, `2`, or `3` causes `mxGetFieldNameByNumber` to return `0`.

C Examples

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- `mxisclass.c` in the `mx` subdirectory of the `examples` directory
- `explore.c` in the `mex` subdirectory of the `examples` directory

See Also

`mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

Purpose

Get field number, given field name in structure array

C Syntax

```
#include "matrix.h"
int mxGetFieldNumber(const mxArray *pm,
    const char *fieldname);
```

Fortran Syntax

```
integer*4 mxGetFieldNumber(pm, fieldname)
mwPointer pm
character*(*) fieldname
```

Arguments

`pm`
Pointer to a structure `mxArray`.

`fieldname`
The name of a field in the structure `mxArray`.

Returns

The field number of the specified `fieldname`, on success. In C, the first field has a field number of `0`, the second field has a field number of `1`, and so on. In Fortran, the first field has a field number of `1`, the second field has a field number of `2`, and so on. Returns `-1` in C (`0` in Fortran) on failure. Common causes of failure include

- Specifying an array pointer `pm` that does not point to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.
- Specifying the `fieldname` of a nonexistent field.

Description

If you know the name of a field but do not know its field number, call `mxGetFieldNumber`. Conversely, if you know the field number but do not know its field name, call `mxGetFieldNameByNumber`.

For example, consider a MATLAB structure initialized to

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

In C, the field `name` has a field number of `0`; the field `billing` has a field number of `1`; and the field `test` has a field number of `2`. If you call `mxGetFieldNumber` and specify a field name of anything other than `name`, `billing`, or `test`, `mxGetFieldNumber` returns `-1`.

Calling

```
mxGetField(pa, index, "field_name");
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "field_name");
mxGetFieldByNumber(pa, index, field_num);
```

where `index` is `0` if you have a 1-by-1 structure.

In Fortran, the field `name` has a field number of `1`; the field `billing` has a field number of `2`; and the field `test` has a field number of `3`. If you call `mxGetFieldNumber` and specify a field name of anything other than `name`, `billing`, or `test`, `mxGetFieldNumber` returns `0`.

Calling

```
mxGetField(pm, index, 'fieldname');
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname');
mxGetFieldByNumber(pm, index, fieldnum);
```

where `index` is `1` if you have a 1-by-1 structure.

C Examples

See `mxcreatestructarray.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetField`, `mxSetFieldByNumber`

Purpose

Get pointer to imaginary data of `mxArray`

C Syntax

```
#include "matrix.h"
void *mxGetImagData(const mxArray *pm);
```

Fortran Syntax

```
mwPointer mxGetImagData(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

The address of the first element of the imaginary data, on success. Returns `NULL` in C (`0` in Fortran) if there is no imaginary data or if there is an error.

Description

This function is similar to `mxGetPi`, except that in C it returns a `void *`.

C Examples

See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxGetData`, `mxGetPi`

mxGetInf (C and Fortran)

Purpose

Get value of infinity

C Syntax

```
#include "matrix.h"
double mxGetInf(void);
```

Fortran Syntax

```
real*8 mxGetInf
```

Returns

The value of infinity on your system.

Description

Call `mxGetInf` to return the value of the MATLAB internal `inf` variable. `inf` is a permanent variable representing IEEE arithmetic positive infinity. The value of `inf` is built into the system; you cannot modify it.

Operations that return infinity include

- Division by 0. For example, 5/0 returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

C Examples

See `mxgetinf.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxGetEps`, `mxGetNaN`

Purpose

Get ir array of sparse matrix

C Syntax

```
#include "matrix.h"
mwIndex *mxGetIr(const mxArray *pm);
```

Fortran Syntax

```
mwPointer mxGetIr(pm)
mwPointer pm
```

Arguments

pm

Pointer to a sparse mxArray

Returns

A pointer to the first element in the ir array, if successful, and NULL in C (0 in Fortran) otherwise. Possible causes of failure include

- Specifying a full (nonsparse) mxArray.
- Specifying a value for pm that is NULL in C (0 in Fortran). This usually means that an earlier call to mxCreateSparse failed.

Description

Use mxGetIr to obtain the starting address of the ir array. The ir array is an array of integers; the length of the ir array is typically nzmax values. For example, if nzmax equals 100, the ir array should contain 100 integers.

Each value in an ir array indicates a row (offset by 1) at which a nonzero element can be found. (The jc array is an index that indirectly specifies a column where nonzero elements can be found.)

For details on the ir and jc arrays, see mxSetIr and mxSetJc.

C Examples

See fulltosparse.c in the refbook subdirectory of the examples directory.

Additional examples:

- explore.c in the mex subdirectory of the examples directory

- `mxsetdimensions.c` and `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory

See Also

`mxGetJc`, `mxGetNzmax`, `mxSetIr`, `mxSetJc`, `mxSetNzmax`

Purpose

Get jc array of sparse matrix

C Syntax

```
#include "matrix.h"
mwIndex *mxGetJc(const mxArray *pm);
```

Fortran Syntax

```
mwPointer mxGetJc(pm)
mwPointer pm
```

Arguments

pm

Pointer to a sparse mxArray

Returns

A pointer to the first element in the jc array, if successful, and NULL in C (0 in Fortran) otherwise. Possible causes of failure include

- Specifying a full (nonsparse) mxArray.
- Specifying a value for pm that is NULL in C (0 in Fortran). This usually means that an earlier call to mxCreateSparse failed.

Description

Use mxGetJc to obtain the starting address of the jc array. The jc array is an integer array having n+1 elements, where n is the number of columns in the sparse mxArray. The values in the jc array indirectly indicate columns containing nonzero elements. For a detailed explanation of the jc array, see mxSetJc.

C Examples

See fulltosparse.c in the refbook subdirectory of the examples directory.

Additional examples:

- explore.c in the mex subdirectory of the examples directory
- mxgetnzmax.c, mxsetdimensions.c, and mxsetnzmax.c in the mx subdirectory of the examples directory

See Also

mxGetIr, mxGetNzmax, mxSetIr, mxSetJc, mxSetNzmax

mxGetLogicals (C)

Purpose

Get pointer to logical array data

C Syntax

```
#include "matrix.h"
mxLogical *mxGetLogicals(const mxArray *array_ptr);
```

Arguments

`array_ptr`
Pointer to an `mxArray`

Returns

The address of the first logical element in the `mxArray`. The result is unspecified if the `mxArray` is not a logical array.

Description

Call `mxGetLogicals` to determine the address of the first logical element in the `mxArray` that `array_ptr` points to. Once you have the starting address, you can access any other element in the `mxArray`.

See Also

`mxCreateLogicalArray`, `mxCreateLogicalMatrix`, `mxCreateLogicalScalar`, `mxIsLogical`, `mxIsLogicalScalar`, `mxIsLogicalScalarTrue`

Purpose

Get number of rows in mxArray

C Syntax

```
#include "matrix.h"
mwSize mxGetM(const mxArray *pm);
```

Fortran Syntax

```
mwSize mxGetM(pm)
mwPointer pm
```

Arguments

pm
: Pointer to an mxArray

Returns

The number of rows in the mxArray to which pm points.

Description

mxGetM returns the number of rows in the specified array. The term *rows* always means the first dimension of the array, no matter how many dimensions the array has. For example, if pm points to a four-dimensional array having dimensions 8-by-9-by-5-by-3, mxGetM returns 8.

C Examples

See convec.c in the refbook subdirectory of the examples directory.

Additional examples:

- fulltosparse.c, revord.c, timestwo.c, and xtimesy.c in the refbook subdirectory of the examples directory
- explore.c, mexget.c, mexlock.c, mexsettrapflag.c and yprime.c in the mex subdirectory of the examples directory
- mxmalloc.c, mxsetdimensions.c, mxgetnzmax.c, and mxsetnzmax.c in the mx subdirectory of the examples directory

Fortran Examples

See matdemo2.F in the eng_mat subdirectory of the examples directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

mxGetN, mxSetM, mxSetN

mxGetN (C and Fortran)

Purpose

Get number of columns in mxArray

C Syntax

```
#include "matrix.h"
mwSize mxGetN(const mxArray *pm);
```

Fortran Syntax

```
mwSize mxGetN(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

The number of columns in the mxArray.

Description

Call mxGetN to determine the number of columns in the specified mxArray.

If pm is an N-dimensional mxArray, mxGetN is the product of dimensions 2 through N. For example, if pm points to a four-dimensional mxArray having dimensions 13-by-5-by-4-by-6, mxGetN returns the value 120 (5 × 4 × 6). If the specified mxArray has more than two dimensions and you need to know exactly how many elements are in each dimension, call mxGetDimensions.

If pm points to a sparse mxArray, mxGetN still returns the number of columns, not the number of occupied columns.

C Examples

See convec.c in the refbook subdirectory of the examples directory.

Additional examples:

- fulltosparse.c, revord.c, timestwo.c, and xtimesy.c in the refbook subdirectory of the examples directory
- explore.c, mexget.c, mexlock.c, mexsettrapflag.c and yprime.c in the mex subdirectory of the examples directory
- mxmalloc.c, mxsetdimensions.c, mxgetnzmax.c, and mxsetnzmax.c in the mx subdirectory of the examples directory

Fortran Examples

See `matdemo2.F` in the `eng_mat` subdirectory of the `examples` directory for a sample program that illustrates how to use this routine in a Fortran program.

See Also

`mxGetM`, `mxGetDimensions`, `mxSetM`, `mxSetN`

mxGetNaN (C and Fortran)

Purpose

Get value of NaN (Not-a-Number)

C Syntax

```
#include "matrix.h"
double mxGetNaN(void);
```

Fortran Syntax

```
real*8 mxGetNaN
```

Returns

The value of NaN (Not-a-Number) on your system

Description

Call mxGetNaN to return the value of NaN for your system. NaN is the IEEE arithmetic representation for Not-a-Number. Certain mathematical operations return NaN as a result, for example,

- `0.0/0.0`
- `Inf-Inf`

The value of Not-a-Number is built in to the system. You cannot modify it.

C Examples

See `mxgetinf.c` in the `mx` subdirectory of the `examples` directory.

See Also

mxGetEps, mxGetInf

Purpose

Get number of dimensions in `mxArray`

C Syntax

```
#include "matrix.h"
mwSize mxGetNumberOfDimensions(const mxArray *pm);
```

Fortran Syntax

```
mwSize mxGetNumberOfDimensions(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

The number of dimensions in the specified `mxArray`. The returned value is always `2` or greater.

Description

Use `mxGetNumberOfDimensions` to determine how many dimensions are in the specified array. To determine how many elements are in each dimension, call `mxGetDimensions`.

C Examples

See `explore.c` in the `mex` subdirectory of the `examples` directory.

Additional examples:

- `findnz.c`, `fulltosparse.c`, and `phonebook.c` in the `refbook` subdirectory of the `examples` directory
- `mxcalcsinglesubscript.c`, `mxgeteps.c`, and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxSetM`, `mxSetN`, `mxGetDimensions`

mxGetNumberOfElements (C and Fortran)

Purpose

Get number of elements in `mxArray`

C Syntax

```
#include "matrix.h"
mwSize mxGetNumberOfElements(const mxArray *pm);
```

Fortran Syntax

```
mwSize mxGetNumberOfElements(pm)
mwPointer pm
```

Arguments

`pm`
Pointer to an `mxArray`

Returns

Number of elements in the specified `mxArray`

Description

`mxGetNumberOfElements` tells you how many elements an array has. For example, if the dimensions of an array are 3-by-5-by-10, `mxGetNumberOfElements` returns the number `150`.

C Examples

See `findnz.c` and `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- `explore.c` in the `mex` subdirectory of the `examples` directory
- `mxcalcsinglesubscript.c`, `mxgeteps.c`, `mxgetinf.c`, `mxisfinite.c`, and `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory

See Also

`mxGetDimensions`, `mxGetM`, `mxGetN`, `mxGetClassID`, `mxGetClassName`

Purpose

Get number of fields in structure mxArray

C Syntax

```
#include "matrix.h"
int mxGetNumberOfFields(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxGetNumberOfFields(pm)
mwPointer pm
```

Arguments

pm

Pointer to a structure mxArray

Returns

The number of fields, on success. Returns 0 on failure. The most common cause of failure is that pm is not a structure mxArray. Call mxIsStruct to determine whether pm is a structure.

Description

Call mxGetNumberOfFields to determine how many fields are in the specified structure mxArray.

Once you know the number of fields in a structure, you can loop through every field in order to set or to get field values.

C Examples

See phonebook.c in the refbook subdirectory of the examples directory.

Additional examples:

- mxisclass.c in the mx subdirectory of the examples directory
- explore.c in the mex subdirectory of the examples directory.

See Also

mxGetField, mxIsStruct, mxSetField

mxGetNzmax (C and Fortran)

Purpose

Get number of elements in `ir`, `pr`, and `pi` arrays

C Syntax

```
#include "matrix.h"
mwSize mxGetNzmax(const mxArray *pm);
```

Fortran Syntax

```
mwSize mxGetNzmax(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to a sparse `mxArray`

Returns

The number of elements allocated to hold nonzero entries in the specified sparse `mxArray`, on success. Returns an indeterminate value on error. The most likely cause of failure is that `pm` points to a full (nonsparse) `mxArray`.

Description

Use `mxGetNzmax` to get the value of the `nzmax` field. The `nzmax` field holds an integer value that signifies the number of elements in the `ir`, `pr`, and, if it exists, the `pi` arrays. The value of `nzmax` is always greater than or equal to the number of nonzero elements in a sparse `mxArray`. In addition, the value of `nzmax` is always less than or equal to the number of rows times the number of columns.

As you adjust the number of nonzero elements in a sparse `mxArray`, MATLAB often adjusts the value of the `nzmax` field. MATLAB adjusts `nzmax` in order to reduce the number of costly reallocations and in order to optimize its use of heap space.

C Examples

See `mxgetnzmax.c` and `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxSetNzmax`

Purpose

Get imaginary data elements in mxArray

C Syntax

```
#include "matrix.h"
double *mxGetPi(const mxArray *pm);
```

Fortran Syntax

```
mwPointer mxGetPi(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

The imaginary data elements of the specified mxArray, on success. Returns NULL in C (0 in Fortran) if there is no imaginary data or if there is an error.

Description

The pi field points to an array containing the imaginary data of the mxArray. Call mxGetPi to get the contents of the pi field, that is, to get the starting address of this imaginary data.

The best way to determine whether an mxArray is purely real is to call mxIsComplex.

The imaginary parts of all input matrices to a MATLAB function are allocated if any of the input matrices are complex.

C Examples

See convec.c, findnz.c, and fulltosparse.c in the refbook subdirectory of the examples directory.

Additional examples:

- explore.c and mexcallmatlab.c in the mex subdirectory of the examples directory
- mxcalcsinglesubscript.c, mxgetinf.c, mxisfinite.c, and mxsetnzmax.c in the mx subdirectory of the examples directory

See Also

mxGetPr, mxSetPi, mxSetPr

mxGetPr (C and Fortran)

Purpose

Get real data elements in mxArray

C Syntax

```
#include "matrix.h"
double *mxGetPr(const mxArray *pm);
```

Fortran Syntax

```
mwPointer mxGetPr(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

The address of the first element of the real data. Returns NULL in C (0 in Fortran) if there is no real data.

Description

Call mxGetPr to determine the starting address of the real data in the mxArray that pm points to. Once you have the starting address, you can access any other element in the mxArray.

C Examples

See convec.c, doubleelement.c, findnz.c, fulltosparse.c, sincall.c, timestwo.c, timestwoalt.c, and xtimesy.c in the refbook subdirectory of the examples directory.

See Also

mxGetPi, mxSetPi, mxSetPr

Purpose

Get real component of first data element in mxArray

C Syntax

```
#include "matrix.h"
double mxGetScalar(const mxArray *pm);
```

Fortran Syntax

```
real*8 mxGetScalar(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray; cannot be a cell mxArray, a structure mxArray, or an empty mxArray

Returns

The value of the first real (nonimaginary) element of the mxArray. Notice that in C, mxGetScalar returns a double. Therefore, if real elements in the mxArray are stored as something other than double, mxGetScalar automatically converts the scalar value into a double. To preserve the original data representation of the scalar, you must cast the return value to the desired data type.

mxGetScalar should only be called when pm points to a non-empty numeric, logical, or char mxArray. Use mx functions such as mxIsEmpty, mxIsLogical, mxIsNumeric, or mxIsChar to test for this condition before calling mxGetScalar.

If pm points to a sparse mxArray, mxGetScalar returns the value of the first nonzero real element in the mxArray.

Description

Call mxGetScalar to get the value of the first real (nonimaginary) element of the mxArray.

In most cases, you call mxGetScalar when pm points to an mxArray containing only one element (a scalar). However, pm can point to an mxArray containing many elements. If pm points to an mxArray containing multiple elements, mxGetScalar returns the value of the first real element. If pm points to a two-dimensional mxArray, mxGetScalar returns the value of the (1,1) element; if pm points to

a three-dimensional `mxArray`, `mxGetScalar` returns the value of the `(1,1,1)` element; and so on.

C Examples

See timestwoalt.c and xtimesy.c in the `refbook` subdirectory of the `examples` directory.

Additional examples:

- mxsetdimensions.c in the `mx` subdirectory of the `examples` directory
- mexlock.c and mexsettrapflag.c in the `mex` subdirectory of the `examples` directory

See Also

`mxGetM`, `mxGetN`

Purpose

Copy string mxArray to C-style string

C Syntax

```
#include "matrix.h"
int mxGetString(const mxArray *pm, char *str, mwSize strlen);
```

Fortran Syntax

```
integer*4 mxGetString(pm, str, strlen)
mwPointer pm
character*(*) str
mwSize strlen
```

Arguments

pm
: Pointer to a string mxArray; that is, a pointer to an mxArray having the mxCHAR_CLASS class.

str
: The starting location into which the string should be written. mxGetString writes the character data into str and then, in C, terminates the string with a NULL character (in the manner of C strings). str can point to either dynamic or static memory.

strlen
: Maximum number of characters to read into str. Typically, in C, you set strlen to 1 plus the number of elements in the string mxArray to which pm points. See the mxGetM and mxGetN reference pages to find out how to get the number of elements.

Returns

0 on success, and 1 on failure. Possible reasons for failure include

- Specifying an mxArray that is not a string mxArray.
- Specifying strlen with less than the number of characters needed to store the entire mxArray pointed to by pm. If this is the case, 1 is returned and the string is truncated.

Description

Call mxGetString to copy the character data of a string mxArray into a C-style string in C or a character array in Fortran. The copied string starts at str and contains no more than strlen-1 characters in C (no

more than strlen characters in Fortran). In C, the C-style string is always terminated with a NULL character.

If the string array contains several rows, they are copied—one column at a time—into one long string array.

Multibyte Character Sets

This function is for use only with strings that represent single-byte character sets. For strings that represent multibyte character sets, use the C function mxArrayToString. Fortran users must allocate sufficient space for the return string to avoid possible truncation.

```
strlen = (mxGetM(prhs[0]) * mxGetN(prhs[0]) * sizeof(mxChar)) + 1
```

C Examples

Examples:

- explore.c in the mex subdirectory of the examples directory
- mxmalloc.c in the mx subdirectory of the examples directory

See Also

mxArrayToString, mxCreateCharArray, mxCreateCharMatrixFromStrings, mxCreateString

Purpose

Determine whether input is cell mxArray

C Syntax

```
#include "matrix.h"
bool mxIsCell(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsCell(pm)
mwPointer pm
```

Arguments

pm
: Pointer to an mxArray

Returns

Logical 1 (true) if pm points to an array having the class mxCELL_CLASS, and logical 0 (false) otherwise.

Description

Use mxIsCell to determine whether the specified array is a cell array.

In C, calling mxIsCell is equivalent to calling

```
mxGetClassID(pm) == mxCELL_CLASS
```

In Fortran, calling mxIsCell is equivalent to calling

```
mxGetClassName(pm) .eq. 'cell'
```

Note mxIsCell does not answer the question "Is this mxArray a cell of a cell array?" An individual cell of a cell array can be of any type.

See Also

mxIsClass

mxIsChar (C and Fortran)

Purpose

Determine whether input is string mxArray

C Syntax

```
#include "matrix.h"
bool mxIsChar(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsChar(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if pm points to an array having the class mxCHAR_CLASS, and logical 0 (false) otherwise.

Description

Use mxIsChar to determine whether pm points to string mxArray.

In C, calling mxIsChar is equivalent to calling

```
mxGetClassID(pm) == mxCHAR_CLASS
```

In Fortran, calling mxIsChar is equivalent to calling

```
mxGetClassName(pm) .eq. 'char'
```

C Examples

See phonebook.c and revord.c in the refbook subdirectory of the examples directory.

For additional examples, see mxcreatecharmatrixfromstr.c, mxislogical.c, and mxmalloc.c in the mx subdirectory of the examples directory.

See Also

mxIsClass, mxGetClassID

Purpose

Determine whether mxArray is member of specified class

C Syntax

```
#include "matrix.h"
bool mxIsClass(const mxArray *pm, const char *classname);
```

Fortran Syntax

```
integer*4 mxIsClass(pm, classname)
mwPointer pm
character*(*) classname
```

Arguments

`pm`

Pointer to an mxArray

`classname`

The array category that you are testing. Specify classname as a string (not as an integer identifier). You can specify any one of the following predefined constants:

Value of classname	Corresponding Class
cell	mxCELL_CLASS
char	mxCHAR_CLASS
double	mxDOUBLE_CLASS
function_handle	mxFUNCTION_CLASS
int8	mxINT8_CLASS
int16	mxINT16_CLASS
int32	mxINT32_CLASS
int64	mxINT64_CLASS
logical	mxLOGICAL_CLASS
single	mxSINGLE_CLASS
struct	mxSTRUCT_CLASS
uint8	mxUINT8_CLASS

Value of classname	Corresponding Class
uint16	mxUINT16_CLASS
uint32	mxUINT32_CLASS
uint64	mxUINT64_CLASS
<class_name>	*<class_id>*
unknown	mxUNKNOWN_CLASS

In the table, *<class_name>* represents the name of a specific MATLAB custom object. You can also specify one of your own class names.

Returns

Logical `1` (`true`) if `pm` points to an array having category `classname`, and logical `0` (`false`) otherwise.

Description

Each `mxArray` is tagged as being a certain type. Call `mxIsClass` to determine whether the specified `mxArray` has this type.

In C,

```
mxIsClass("double");
```

is equivalent to calling either of these forms:

```
mxIsDouble(pm);
```

```
strcmp(mxGetClassName(pm), "double");
```

In Fortran,

```
mxIsClass(pm, 'double')
```

is equivalent to calling either one of the following

```
mxIsDouble(pm)
```

```
mxGetClassName(pm) .eq. 'double'
```

It is most efficient to use the `mxIsDouble` form.

C Examples

See `mxisclass.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxClassID`, `mxGetClassID`, `mxIsEmpty`, `mxGetClassName`

mxIsComplex (C and Fortran)

Purpose

Determine whether data is complex

C Syntax

```
#include "matrix.h"
bool mxIsComplex(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsComplex(pm)
mwPointer pm
```

Arguments

`pm`
Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` is a numeric array containing complex data, and logical `0` (`false`) otherwise. If `pm` points to a cell array or a structure array, `mxIsComplex` returns `false`.

Description

Use `mxIsComplex` to determine whether or not an imaginary part is allocated for an `mxArray`. The imaginary pointer `pi` is `NULL` in C (`0` in Fortran) if an `mxArray` is purely real and does not have any imaginary data. If an `mxArray` is complex, `pi` points to an array of numbers.

C Examples

See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

Additional examples:

- `convec.c`, `phonebook.c`, `timestwo.c`, and `xtimesy.c` in the `refbook` subdirectory of the `examples` directory
- `explore.c`, `yprime.c`, `mexlock.c`, and `mexsettrapflag.c` in the `mex` subdirectory of the `examples` directory
- `mxcalcsinglesubscript.c`, `mxgeteps.c`, and `mxgetinf.c` in the `mx` subdirectory of the `examples` directory

See Also

`mxIsNumeric`

Purpose

Determine whether mxArray represents data as double-precision, floating-point numbers

C Syntax

```
#include "matrix.h"
bool mxIsDouble(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsDouble(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the mxArray stores its data as double-precision, floating-point numbers, and logical 0 (false) otherwise.

Description

Call mxIsDouble to determine whether or not the specified mxArray represents its real and imaginary data as double-precision, floating-point numbers.

Older versions of MATLAB store all mxArray data as double-precision, floating-point numbers. However, starting with MATLAB Version 5, MATLAB can store real and imaginary data in a variety of numerical formats.

In C, calling mxIsDouble is equivalent to calling

```
mxGetClassID(pm) == mxDOUBLE_CLASS
```

In Fortran, calling mxIsDouble is equivalent to calling

```
mxGetClassName(pm) .eq. 'double'
```

C Examples

See findnz.c, fulltosparse.c, timestwo.c, and xtimesy.c in the refbook subdirectory of the examples directory.

Additional examples:

- `mexget.c`, `mexlock.c`, `mexsettrapflag.c`, and `yprime.c` in the `mex` subdirectory of the `examples` directory
- `mxcalcsinglesubscript.c`, `mxgeteps.c`, `mxgetinf.c`, and `mxisfinite.c` in the `mx` subdirectory of the `examples` directory

See Also

`mxIsClass`, `mxGetClassID`

Purpose

Determine whether mxArray is empty

C Syntax

```
#include "matrix.h"
bool mxIsEmpty(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsEmpty(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the mxArray is empty, and logical 0 (false) otherwise.

Description

Use mxIsEmpty to determine whether an mxArray contains no data. An mxArray is empty if the size of any of its dimensions is 0.

C Examples

See mxisfinite.c in the mx subdirectory of the examples directory.

See Also

mxIsClass

mxIsFinite (C and Fortran)

Purpose

Determine whether input is finite

C Syntax

```
#include "matrix.h"
bool mxIsFinite(double value);
```

Fortran Syntax

```
integer*4 mxIsFinite(value)
real*8 value
```

Arguments

`value`
The double-precision, floating-point number that you are testing

Returns

Logical `1` (`true`) if `value` is finite, and logical `0` (`false`) otherwise.

Description

Call `mxIsFinite` to determine whether or not `value` is finite. A number is finite if it is greater than `-Inf` and less than `Inf`.

C Examples

See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxIsInf`, `mxIsNan`

Purpose

Determine whether mxArray was copied from MATLAB global workspace

C Syntax

```
#include "matrix.h"
bool mxIsFromGlobalWS(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsFromGlobalWS(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the array was copied out of the global workspace, and logical 0 (false) otherwise.

Description

mxIsFromGlobalWS is useful for stand-alone MAT programs. mexIsGlobal tells you whether the pointer you pass actually points into the global workspace.

C Examples

See matdgns.c and matcreat.c in the eng_mat subdirectory of the examples directory.

See Also

mexIsGlobal

mxIsInf (C and Fortran)

Purpose

Determine whether input is infinite

C Syntax

```
#include "matrix.h"
bool mxIsInf(double value);
```

Fortran Syntax

```
integer*4 mxIsInf(value)
real*8 value
```

Arguments

`value`
The double-precision, floating-point number that you are testing

Returns

Logical `1` (`true`) if value is infinite, and logical `0` (`false`) otherwise.

Description

Call `mxIsInf` to determine whether or not `value` is equal to infinity or minus infinity. MATLAB stores the value of infinity in a permanent variable named `Inf`, which represents IEEE arithmetic positive infinity. The value of the variable `Inf` is built into the system; you cannot modify it.

Operations that return infinity include

- Division by 0. For example, `5/0` returns infinity.
- Operations resulting in overflow. For example, `exp(10000)` returns infinity because the result is too large to be represented on your machine.

If `value` equals `NaN` (Not-a-Number), `mxIsInf` returns `false`. In other words, `NaN` is not equal to infinity.

C Examples

See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxIsFinite`, `mxIsNaN`

Purpose

Determine whether mxArray represents data as signed 16-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsInt16(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsInt16(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the array stores its data as signed 16-bit integers, and logical 0 (false) otherwise.

Description

Use mxIsInt16 to determine whether or not the specified array represents its real and imaginary data as 16-bit signed integers.

In C, calling mxIsInt16 is equivalent to calling

```
mxGetClassID(pm) == mxINT16_CLASS
```

In Fortran, calling mxIsInt16 is equivalent to calling

```
mxGetClassName(pm) == 'int16'
```

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

mxIsInt32 (C and Fortran)

Purpose

Determine whether mxArray represents data as signed 32-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsInt32(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsInt32(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the array stores its data as signed 32-bit integers, and logical 0 (false) otherwise.

Description

Use mxIsInt32 to determine whether or not the specified array represents its real and imaginary data as 32-bit signed integers.

In C, calling mxIsInt32 is equivalent to calling

```
mxGetClassID(pm) == mxINT32_CLASS
```

In Fortran, calling mxIsInt32 is equivalent to calling

```
mxGetClassName(pm) == 'int32'
```

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

Purpose

Determine whether mxArray represents data as signed 64-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsInt64(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsInt64(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the array stores its data as signed 64-bit integers, and logical 0 (false) otherwise.

Description

Use mxIsInt64 to determine whether or not the specified array represents its real and imaginary data as 64-bit signed integers.

In C, calling mxIsInt64 is equivalent to calling

```
mxGetClassID(pm) == mxINT64_CLASS
```

In Fortran, calling mxIsInt64 is equivalent to calling

```
mxGetClassName(pm) == 'int64'
```

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

mxIsInt8 (C and Fortran)

Purpose

Determine whether mxArray represents data as signed 8-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsInt8(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsInt8(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the array stores its data as signed 8-bit integers, and logical 0 (false) otherwise.

Description

Use mxIsInt8 to determine whether or not the specified array represents its real and imaginary data as 8-bit signed integers.

In C, calling mxIsInt8 is equivalent to calling

```
mxGetClassID(pm) == mxINT8_CLASS
```

In Fortran, calling mxIsInt8 is equivalent to calling

```
mxGetClassName(pm) .eq. 'int8'
```

See Also

mxIsClass, mxGetClassID, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32, mxIsUint64

Purpose

Determine whether `mxArray` is of type `mxLogical`

C Syntax

```
#include "matrix.h"
bool mxIsLogical(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsLogical(pm)
mwPointer pm
```

Arguments

`pm`

Pointer to an `mxArray`

Returns

Logical `1` (`true`) if `pm` points to a logical `mxArray`, and logical `0` (`false`) otherwise.

Description

Use `mxIsLogical` to determine whether MATLAB treats the data in the `mxArray` as Boolean (logical). If an `mxArray` is logical, MATLAB treats all zeros as meaning `false` and all nonzero values as meaning `true`. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

C Examples

See `mxislogical.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxIsClass`

mxIsLogicalScalar (C)

Purpose

Determine whether scalar `mxArray` is of type `mxLogical`

C Syntax

```
#include "matrix.h"
bool mxIsLogicalScalar(const mxArray *array_ptr);
```

Arguments

`array_ptr`
Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the `mxArray` is of class `mxLogical` and has 1-by-1 dimensions, and logical `0` (`false`) otherwise.

Description

Use `mxIsLogicalScalar` to determine whether MATLAB treats the scalar data in the `mxArray` as logical or numerical. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

`mxIsLogicalScalar(pa)` is equivalent to

```
mxIsLogical(pa) && mxGetNumberOfElements(pa) == 1
```

See Also

`mxIsLogical`, `mxIsLogicalScalarTrue`, `mxGetLogicals`, `mxGetScalar`

Purpose

Determine whether scalar `mxArray` of type `mxLogical` is `true`

C Syntax

```
#include "matrix.h"
bool mxIsLogicalScalarTrue(const mxArray *array_ptr);
```

Arguments

`array_ptr`
Pointer to an `mxArray`

Returns

Logical `1` (`true`) if the value of the `mxArray`'s logical, scalar element is `true`, and logical `0` (`false`) otherwise.

Description

Use `mxIsLogicalScalarTrue` to determine whether the value of a scalar `mxArray` is true or false. For additional information on the use of logical variables in MATLAB, type `help logical` at the MATLAB prompt.

`mxIsLogicalScalarTrue(pa)` is equivalent to

```
mxIsLogical(pa) && mxGetNumberOfElements(pa) == 1 &&
mxGetLogicals(pa)[0] == true
```

See Also

`mxIsLogical`, `mxIsLogicalScalar`, `mxGetLogicals`, `mxGetScalar`

mxIsNaN (C and Fortran)

Purpose

Determine whether input is NaN (Not-a-Number)

C Syntax

```
#include "matrix.h"
bool mxIsNaN(double value);
```

Fortran Syntax

```
integer*4 mxIsNaN(value)
real*8 value
```

Arguments

`value`
The double-precision, floating-point number that you are testing

Returns

Logical `1` (`true`) if `value` is `NaN` (Not-a-Number), and logical `0` (`false`) otherwise.

Description

Call `mxIsNaN` to determine whether or not `value` is `NaN`. `NaN` is the IEEE arithmetic representation for Not-a-Number. A `NaN` is obtained as a result of mathematically undefined operations such as

- `0.0/0.0`
- `Inf-Inf`

The system understands a family of bit patterns as representing `NaN`. In other words, `NaN` is not a single value; rather, it is a family of numbers that MATLAB (and other IEEE-compliant applications) use to represent an error condition or missing data.

C Examples

See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

For additional examples, see `findnz.c` and `fulltosparse.c` in the `refbook` subdirectory of the `examples` directory.

See Also

`mxIsFinite`, `mxIsInf`

Purpose

Determine whether mxArray is numeric

C Syntax

```
#include "matrix.h"
bool mxIsNumeric(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsNumeric(pm)
mwPointer pm
```

Arguments

pm
: Pointer to an mxArray

Returns

Logical 1 (true) if the array can contain numeric data. The following class IDs represent storage types for arrays that can contain numeric data:

- mxDOUBLE_CLASS
- mxSINGLE_CLASS
- mxINT8_CLASS
- mxUINT8_CLASS
- mxINT16_CLASS
- mxUINT16_CLASS
- mxINT32_CLASS
- mxUINT32_CLASS
- mxINT64_CLASS
- mxUINT64_CLASS

Logical 0 (false) if the array cannot contain numeric data.

Description

Call mxIsNumeric to determine whether the specified array contains numeric data. If the specified array has a storage type that represents

numeric data, `mxIsNumeric` returns logical `1` (`true`). Otherwise, `mxIsNumeric` returns logical `0` (`false`).

Call `mxGetClassID` to determine the exact storage type.

C Examples

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory.

Fortran Examples

See `matdemo1.F` in the `eng_mat` subdirectory of the `examples` directory.

See Also

`mxGetClassID`

Purpose

Determine whether mxArray represents data as single-precision, floating-point numbers

C Syntax

```
#include "matrix.h"
bool mxIsSingle(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsSingle(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the array stores its data as single-precision, floating-point numbers, and logical 0 (false) otherwise.

Description

Use mxIsSingle to determine whether or not the specified array represents its real and imaginary data as single-precision, floating-point numbers.

In C, calling mxIsSingle is equivalent to calling

```
mxGetClassID(pm) == mxSINGLE_CLASS
```

In Fortran, calling mxIsSingle is equivalent to calling

```
mxGetClassName(pm) .eq. 'single'
```

See Also

mxIsClass, mxGetClassID

mxIsSparse (C and Fortran)

Purpose

Determine whether input is sparse mxArray

C Syntax

```
#include "matrix.h"
bool mxIsSparse(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsSparse(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if pm points to a sparse mxArray, and logical 0 (false) otherwise. A false return value means that pm points to a full mxArray or that pm does not point to a legal mxArray.

Description

Use mxIsSparse to determine whether pm points to a sparse mxArray. Many routines (for example, mxGetIr and mxGetJc) require a sparse mxArray as input.

C Examples

See phonebook.c in the refbook subdirectory of the examples directory.

For additional examples, see mxgetnzmax.c, mxsetdimensions.c, and mxsetnzmax.c in the mx subdirectory of the examples directory.

See Also

mxGetIr, mxGetJc, mxCreateSparse

Purpose

Determine whether input is structure mxArray

C Syntax

```
#include "matrix.h"
bool mxIsStruct(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsStruct(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if pm points to a structure mxArray, and logical 0 (false) otherwise.

Description

Use mxIsStruct to determine whether pm points to a structure mxArray. Many routines (for example, mxGetFieldName and mxSetField) require a structure mxArray as an argument.

C Examples

See phonebook.c in the refbook subdirectory of the examples directory.

See Also

mxCreateStructArray, mxCreateStructMatrix, mxGetNumberOfFields, mxGetField, mxSetField

mxIsUint16 (C and Fortran)

Purpose

Determine whether mxArray represents data as unsigned 16-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsUint16(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsUint16(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the mxArray stores its data as unsigned 16-bit integers, and logical 0 (false) otherwise.

Description

Use mxIsUint16 to determine whether or not the specified mxArray represents its real and imaginary data as 16-bit unsigned integers.

In C, calling mxIsUint16 is equivalent to calling

```
mxGetClassID(pm) == mxUINT16_CLASS
```

In Fortran, calling mxIsUint16 is equivalent to calling

```
mxGetClassName(pm) .eq. 'uint16'
```

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint32, mxIsUint64

Purpose

Determine whether mxArray represents data as unsigned 32-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsUint32(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsUint32(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the mxArray stores its data as unsigned 32-bit integers, and logical 0 (false) otherwise.

Description

Use mxIsUint32 to determine whether or not the specified mxArray represents its real and imaginary data as 32-bit unsigned integers.

In C, calling mxIsUint32 is equivalent to calling

```
mxGetClassID(pm) == mxUINT32_CLASS
```

In Fortran, calling mxIsUint32 is equivalent to calling

```
mxGetClassName(pm) .eq. 'uint32'
```

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint64

mxIsUint64 (C and Fortran)

Purpose

Determine whether mxArray represents data as unsigned 64-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsUint64(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsUint64(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the mxArray stores its data as unsigned 64-bit integers, and logical 0 (false) otherwise.

Description

Use mxIsUint64 to determine whether or not the specified mxArray represents its real and imaginary data as 64-bit unsigned integers.

In C, calling mxIsUint64 is equivalent to calling

```
mxGetClassID(pm) == mxUINT64_CLASS
```

In Fortran, calling mxIsUint64 is equivalent to calling

```
mxGetClassName(pm) .eq. 'uint64'
```

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint8, mxIsUint16, mxIsUint32

Purpose

Determine whether mxArray represents data as unsigned 8-bit integers

C Syntax

```
#include "matrix.h"
bool mxIsUint8(const mxArray *pm);
```

Fortran Syntax

```
integer*4 mxIsUint8(pm)
mwPointer pm
```

Arguments

pm

Pointer to an mxArray

Returns

Logical 1 (true) if the mxArray stores its data as unsigned 8-bit integers, and logical 0 (false) otherwise.

Description

Use mxIsUint8 to determine whether or not the specified mxArray represents its real and imaginary data as 8-bit unsigned integers.

In C, calling mxIsUint8 is equivalent to calling

```
mxGetClassID(pm) == mxUINT8_CLASS
```

In Fortran, calling mxIsUint8 is equivalent to calling

```
mxGetClassName(pm) .eq. 'uint8'
```

See Also

mxIsClass, mxGetClassID, mxIsInt8, mxIsInt16, mxIsInt32, mxIsInt64, mxIsUint16, mxIsUint32, mxIsUint64

mxLogical (C)

Purpose

Type for logical `mxArray`

Description

All logical `mxArrays` store their data elements as `mxLogical` rather than as `bool`.

The header file containing this type is

```
#include "matrix.h"
```

Examples

See `mxislogical.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCreateLogicalArray`

Purpose

Allocate dynamic memory using MATLAB memory manager

C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxMalloc(mwSize n);
```

Fortran Syntax

```
mwPointer mxMalloc(n)
mwSize n
```

Arguments

`n`

Number of bytes to allocate

Returns

A pointer to the start of the allocated dynamic memory, if successful. If unsuccessful in a stand-alone (nonMEX-file) application, `mxMalloc` returns `NULL` in C (`0` in Fortran). If unsuccessful in a MEX-file, the MEX-file terminates and control returns to the MATLAB prompt.

`mxMalloc` is unsuccessful when there is insufficient free heap space.

Description

MATLAB applications should always call `mxMalloc` rather than `malloc` to allocate memory.

`mxMalloc` works differently in MEX-files than in stand-alone MATLAB applications. In MEX-files, `mxMalloc` automatically

- Allocates enough contiguous heap space to hold `n` bytes.
- Registers the returned heap space with the MATLAB memory management facility.

The MATLAB memory management facility maintains a list of all memory allocated by `mxMalloc`. The MATLAB memory management facility automatically frees (deallocates) all of a MEX-file's parcels when control returns to the MATLAB prompt.

In stand-alone MATLAB C applications, `mxMalloc` calls the ANSI C `malloc` function.

By default, in a MEX-file, `mxMalloc` generates nonpersistent `mxMalloc` data. In other words, the memory management facility automatically deallocates the memory as soon as the MEX-file ends. If you want the memory to persist after the MEX-file completes, call `mexMakeMemoryPersistent` after calling `mxMalloc`. If you write a MEX-file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory in the event your MEX-file is cleared.

When you finish using the memory allocated by `mxMalloc`, call `mxFree`. `mxFree` deallocates the memory.

C Examples

See `mxmalloc.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxFree`, `mxRealloc`

Purpose

Reallocate memory

C Syntax

```
#include "matrix.h"
#include <stdlib.h>
void *mxRealloc(void *ptr, mwSize size);
```

Fortran Syntax

```
mwPointer mxRealloc(ptr, size)
mwPointer ptr
mwSize size
```

Arguments

ptr
: Pointer to a block of memory allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc`

`size`
: New size of allocated memory, in bytes

Returns

A pointer to the reallocated block of memory, or `NULL` in C (`0` in Fortran) if `size` is `0`. In a stand-alone (non-MEX-file) application, if not enough memory is available to expand the block to the given size, `mxRealloc` returns `NULL` in C (`0` in Fortran). In a MEX-file, if not enough memory is available to expand the block to the given size, the MEX-file terminates and control returns to the MATLAB prompt.

Description

`mxRealloc` changes the size of a memory block that has been allocated with `mxCalloc`, `mxMalloc`, or `mxRealloc`.

If `size` is `0` and `ptr` is not `NULL` in C (`0` in Fortran), `mxRealloc` frees the memory pointed to by `ptr` and returns `NULL` in C (`0` in Fortran).

If `size` is greater than `0` and `ptr` is `NULL` in C (`0` in Fortran), `mxRealloc` behaves like `mxMalloc`, allocating a new block of memory of `size` bytes and returning a pointer to the new block.

Otherwise, `mxRealloc` changes the size of the memory block pointed to by `ptr` to `size` bytes. The contents of the reallocated memory are unchanged up to the smaller of the new and old sizes. The reallocated memory may be in a different location from the original memory, so

the returned pointer can be different from `ptr`. If the memory location changes, `mxRealloc` frees the original memory block pointed to by `ptr`.

In a stand-alone (non-MEX-file) application, if not enough memory is available to expand the block to the given size, `mxRealloc` returns `NULL` in C (`0` in Fortran) and leaves the original memory block unchanged. You must use `mxFree` to free the original memory block.

MATLAB maintains a list of all memory allocated by `mxRealloc`. By default, in a MEX-file, `mxRealloc` generates nonpersistent `mxRealloc` data. The memory management facility automatically deallocates the memory as soon as the MEX-file ends.

If you want the memory to persist after a MEX-file completes, call `mexMakeMemoryPersistent` after calling `mxRealloc`. If you write a MEX-file with persistent memory, be sure to register a `mexAtExit` function to free allocated memory when your MEX-file is cleared.

When you finish using the memory allocated by `mxRealloc`, call `mxFree`. `mxFree` deallocates the memory.

C Examples

See `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mexAtExit`, `mexMakeArrayPersistent`, `mexMakeMemoryPersistent`, `mxCalloc`, `mxDestroyArray`, `mxFree`, `mxMalloc`

Purpose

Remove field from structure array

C Syntax

```
#include "matrix.h"
void mxRemoveField(mxArray pm, int fieldnumber);
```

Fortran Syntax

```
subroutine mxRemoveField(pm, fieldnumber)
mwPointer pm
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`

`fieldnumber`

The number of the field you want to remove. In C, to remove the first field, set `fieldnumber` to `0`; to remove the second field, set `fieldnumber` to `1`; and so on. In Fortran, to remove the first field, set `fieldnumber` to `1`; to remove the second field, set `fieldnumber` to `2`; and so on.

Description

Call `mxRemoveField` to remove a field from a structure array. If the field does not exist, nothing happens. This function does not destroy the field values. Use `mxDestroyArray` to destroy the actual field values.

Consider a MATLAB structure initialized to

```
patient.name = 'John Doe';
patient.billing = 127.00;
patient.test = [79 75 73; 180 178 177.5; 220 210 205];
```

In C, the field number `0` represents the field `name`; field number `1` represents field `billing`; field number `2` represents field `test`. In Fortran, the field number `1` represents the field `name`; field number `2` represents field `billing`; field number `3` represents field `test`.

See Also

`mxAddField`, `mxDestroyArray`, `mxGetFieldByNumber`

mxSetCell (C and Fortran)

Purpose

Set value of one cell of mxArray

C Syntax

```
#include "matrix.h"
void mxSetCell(mxArray *pm, mwIndex index, mxArray *value);
```

Fortran Syntax

```
mxSetCell(pm, index, value)
mwPointer pm, value
mwIndex index
```

Arguments

pm
: Pointer to a cell mxArray

index
: Index from the beginning of the mxArray. Specify the number of elements between the first cell of the mxArray and the cell you want to set. The easiest way to calculate index in a multidimensional cell array is to call mxCalcSingleSubscript.

value
: The new value of the cell. You can put any kind of mxArray into a cell. In fact, you can even put another cell mxArray into a cell.

Description

Call mxSetCell to put the designated value into a particular cell of a cell mxArray.

Note Inputs to a MEX-file are constant read-only mxArrays and should not be modified. Using mxSetCell* or mxSetField* to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetCell before you call mxSetCell.

C Examples

See `phonebook.c` in the `refbook` subdirectory of the `examples` directory. For an additional example, see `mxcreatecellmatrix.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCreateCellArray`, `mxCreateCellMatrix`, `mxGetCell`, `mxIsCell`, `mxFree`

mxSetClassName (C)

Purpose

Convert structure array to MATLAB object array

C Syntax

```
#include "matrix.h"
int mxSetClassName(mxArray *array_ptr, const char *classname);
```

Arguments

`array_ptr`
Pointer to an `mxArray` of class `mxSTRUCT_CLASS`

`classname`
The object class to which to convert `array_ptr`

Returns

`0` if successful, and nonzero otherwise. One cause of failure is that `array_ptr` is not a structure `mxArray`. Call `mxIsStruct` to determine whether `array_ptr` is a structure.

Description

`mxSetClassName` converts a structure array to an object array, to be saved subsequently to a MAT-file. The object is not registered or validated by MATLAB until it is loaded via the `LOAD` command. If the specified `classname` is an undefined class within MATLAB, `LOAD` converts the object back to a simple structure array.

See Also

`mxIsClass`, `mxGetClassID`

Purpose

Set pointer to data

C Syntax

```
#include "matrix.h"
void mxSetData(mxArray *pm, void *pr);
```

Fortran Syntax

```
mxSetData(pm, pr)
mwPointer pm, pr
```

Arguments

`pm`

Pointer to an `mxArray`

`pr`

Pointer to an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this memory.

Description

`mxSetData` is similar to `mxSetPr`, except that in C, its second argument is a `void *`. Use this on numeric arrays with contents other than `double`.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetData` before you call `mxSetData`.

See Also

`mxCalloc`, `mxFree`, `mxGetData`, `mxSetPr`

mxSetDimensions (C and Fortran)

Purpose

Modify number of dimensions and size of each dimension

C Syntax

```
#include "matrix.h"
int mxSetDimensions(mxArray *pm, const mwSize *dims,
  mwSize ndim);
```

Fortran Syntax

```
integer*4 mxSetDimensions(pm, dims, ndim)
mwPointer pm
mwSize dims, ndim
```

Arguments

`pm`
: Pointer to an `mxArray`

`dims`
: The dimensions array. Each element in the dimensions array contains the size of the array in that dimension. For example, in C, setting `dims[0]` to `5` and `dims[1]` to `7` establishes a 5-by-7 `mxArray`. In Fortran, setting `dims(1)` to `5` and `dims(2)` to `7` establishes a 5-by-7 `mxArray`. In most cases, there should be `ndim` elements in the `dims` array.

`ndim`
: The desired number of dimensions

Returns

`0` on success, and `1` on failure. `mxSetDimensions` allocates heap space to hold the input size array. So it is possible (though extremely unlikely) that increasing the number of dimensions can cause the system to run out of heap space.

Description

Call `mxSetDimensions` to reshape an existing `mxArray`. `mxSetDimensions` is similar to `mxSetM` and `mxSetN`; however, `mxSetDimensions` provides greater control for reshaping `mxArrays` that have more than two dimensions.

`mxSetDimensions` does not allocate or deallocate any space for the `pr` or `pi` arrays. Consequently, if your call to `mxSetDimensions` increases the number of elements in the `mxArray`, you must enlarge the `pr` (and `pi`, if it exists) arrays accordingly.

If your call to mxSetDimensions reduces the number of elements in the mxArray, you can optionally reduce the size of the pr and pi arrays using mxRealloc.

Any trailing singleton dimensions specified in the dims argument are automatically removed from the resulting array. For example, if ndim equals 5 and dims equals [4 1 7 1 1], the resulting array is given the dimensions 4-by-1-by-7.

C Examples

See mxsetdimensions.c in the mx subdirectory of the examples directory.

See Also

mxGetNumberOfDimensions, mxSetM, mxSetN, mxRealloc

mxSetField (C and Fortran)

Purpose

Set structure array field, given field name and index

C Syntax

```
#include "matrix.h"
void mxSetField(mxArray *pm, mwIndex index,
  const char *fieldname, mxArray *value);
```

Fortran Syntax

```
mxSetField(pm, index, fieldname, value)
mwPointer pm, value
mwIndex index
character*(*) fieldname
```

Arguments

`pm`

Pointer to a structure `mxArray`. Call `mxIsStruct` to determine whether `pm` points to a structure `mxArray`.

`index`

Index of the desired element. In C, the first element of an `mxArray` has an index of `0`, the second element has an index of `1`, and the last element has an index of `N-1`, where `N` is the total number of elements in the structure `mxArray`. In Fortran, the first element of an `mxArray` has an index of `1`, the second element has an index of `2`, and the last element has an index of `N`. See `mxCalcSingleSubscript` for details on calculating an index.

`fieldname`

The name of the field whose value you are assigning. Call `mxGetFieldNameByNumber` or `mxGetFieldNumber` to determine existing field names.

value

Pointer to the `mxArray` you are assigning.

Description

Use `mxSetField` to assign a `value` to the specified element of the specified field. In pseudo-C terminology, `mxSetField` performs the assignment

```
pm[index].fieldname = value;
```

Note Inputs to a MEX-file are constant read-only `mxArrays` and should not be modified. Using `mxSetCell*` or `mxSetField*` to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

In C, calling

```
mxSetField(pa, index, "fieldname", new_value_pa);
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "fieldname");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

In Fortran, calling

```
mxSetField(pm, index, 'fieldname', newvalue)
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetField` before you call `mxSetField`.

C Examples

See `mxcreatestructarray.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCreateStructArray`, `mxCreateStructMatrix`, `mxGetField`, `mxGetFieldByNumber`, `mxGetFieldNameByNumber`, `mxGetFieldNumber`, `mxGetNumberOfFields`, `mxIsStruct`, `mxSetFieldByNumber`, `mxFree`

mxSetFieldByNumber (C and Fortran)

Purpose

Set structure array field, given field number and index

C Syntax

```
#include "matrix.h"
void mxSetFieldByNumber(mxArray *pm, mwIndex index,
  int fieldnumber, mxArray *value);
```

Fortran Syntax

```
mxSetFieldByNumber(pm, index, fieldnumber, value)
mwPointer pm, value
mwIndex index
integer*4 fieldnumber
```

Arguments

`pm`

Pointer to a structure `mxArray`. Call `mxIsStruct` to determine whether`pm` points to a structure `mxArray`.

`index`

The desired element. In C, the first element of an `mxArray` has an index of `0`, the second element has an index of `1`, and the last element has an index of `N-1`, where `N` is the total number of elements in the structure `mxArray`. In Fortran, the first element of an `mxArray` has an index of `1`, the second element has an index of `2`, and the last element has an index of `N`. See `mxCalcSingleSubscript` for details on calculating an index.

`fieldnumber`

The position of the field whose value you want to extract. In C, the first field within each element has a `fieldnumber` of `0`, the second field has a `fieldnumber` of `1`, and so on. The last field has a `fieldnumber` of `N-1`, where `N` is the number of fields. In Fortran, the first field within each element has a `fieldnumber` of `1`, the second field has a `fieldnumber` of `2`, and so on. The last field has a `fieldnumber` of `N`.

`value`

The value you are assigning.

Description

Use `mxSetFieldByNumber` to assign a `value` to the specified element of the specified field. `mxSetFieldByNumber` is almost identical to

mxSetField; however, the former takes a field number as its third argument and the latter takes a field name as its third argument.

Note Inputs to a MEX-file are constant read-only mxArrays and should not be modified. Using mxSetCell* or mxSetField* to modify the cells or fields of an argument passed from MATLAB causes unpredictable results.

In C, calling

```
mxSetField(pa, index, "field_name", new_value_pa);
```

is equivalent to calling

```
field_num = mxGetFieldNumber(pa, "field_name");
mxSetFieldByNumber(pa, index, field_num, new_value_pa);
```

In Fortran, calling

```
mxSetField(pm, index, 'fieldname', newvalue)
```

is equivalent to calling

```
fieldnum = mxGetFieldNumber(pm, 'fieldname')
mxSetFieldByNumber(pm, index, fieldnum, newvalue)
```

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetFieldByNumber before you call mxSetFieldByNumber.

C Examples

See mxcreatestructarray.c in the mx subdirectory of the examples directory. For an additional example, see phonebook.c in the refbook subdirectory of the examples directory.

See Also

mxCreateStructArray, mxCreateStructMatrix, mxGetField, mxGetFieldByNumber, mxGetFieldNameByNumber, mxGetFieldNumber, mxGetNumberOfFields, mxIsStruct, mxSetField, mxFree

Purpose

Set imaginary data pointer for `mxArray`

C Syntax

```
#include "matrix.h"
void mxSetImagData(mxArray *pm, void *pi);
```

Fortran Syntax

```
mxSetImagData(pm, pi)
mwPointer pm, pi
```

Arguments

`pm`

Pointer to an `mxArray`

`pi`

Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call `mxCalloc` to allocate this dynamic memory. If `pi` points to static memory, memory errors will result when the array is destroyed.

Description

`mxSetImagData` is similar to `mxSetPi`, except that in C, its `pi` argument is a `void *`. Use this on numeric arrays with contents other than `double`.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetImagData` before you call `mxSetImagData`.

C Examples

See `mxisfinite.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxCalloc`, `mxFree`, `mxGetImagData`, `mxSetPi`

mxSetIr (C and Fortran)

Purpose

Set ir array of sparse mxArray

C Syntax

```
#include "matrix.h"
void mxSetIr(mxArray *pm, mwIndex *ir);
```

Fortran Syntax

```
mxSetIr(pm, ir)
mwPointer pm, ir
```

Arguments

pm

Pointer to a sparse mxArray

ir

Pointer to the ir array. The ir array must be sorted in column-major order.

Description

Use mxSetIr to specify the ir array of a sparse mxArray. The ir array is an array of integers; the length of the ir array should equal the value of nzmax.

Each element in the ir array indicates a row (offset by 1) at which a nonzero element can be found. (The jc array is an index that indirectly specifies a column where nonzero elements can be found. See mxSetJc for more details on jc.)

For example, suppose you create a 7-by-3 sparse mxArray named Sparrow containing six nonzero elements by typing

```
Sparrow = zeros(7,3);
Sparrow(2,1) = 1;
Sparrow(5,1) = 1;
Sparrow(3,2) = 1;
Sparrow(2,3) = 2;
Sparrow(5,3) = 1;
Sparrow(6,3) = 1;
Sparrow = sparse(Sparrow);
```

The pr array holds the real data for the sparse matrix, which in Sparrow is the five 1s and the one 2. If there is any nonzero imaginary data, it is in a pi array.

Subscript	ir	pr	jc	Comments
(2,1)	1	1	0	Column 1; ir is 1 because row is 2.
(5,1)	4	1	2	Column 1; ir is 4 because row is 5.
(3,2)	2	1	3	Column 2; ir is 2 because row is 3.
(2,3)	1	2	6	Column 3; ir is 1 because row is 2.
(5,3)	4	1		Column 3; ir is 4 because row is 5.
(6,3)	5	1		Column 3; ir is 5 because row is 6.

Notice how each element of the ir array is always 1 less than the row of the corresponding nonzero element. For instance, the first nonzero element is in row 2; therefore, the first element in ir is 1 (that is, 2 – 1). The second nonzero element is in row 5; therefore, the second element in ir is 4 (5 – 1).

The ir array must be in column-major order. That means that the ir array must define the row positions in column 1 (if any) first, then the row positions in column 2 (if any) second, and so on through column N. Within each column, row position 1 must appear prior to row position 2, and so on.

mxSetIr does not sort the ir array for you; you must specify an ir array that is already sorted.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetIr before you call mxSetIr.

C Examples

See mxsetnzmax.c in the mx subdirectory of the examples directory. For an additional example, see explore.c in the mex subdirectory of the examples directory.

See Also mxCreateSparse, mxGetIr, mxGetJc, mxSetJc, mxFree

Purpose

Set jc array of sparse mxArray

C Syntax

```
#include "matrix.h"
void mxSetJc(mxArray *pm, mwIndex *jc);
```

Fortran Syntax

```
mxSetJc(pm, jc)
mwPointer pm, jc
```

Arguments

pm
: Pointer to a sparse mxArray

jc
: Pointer to the jc array

Description

Use mxSetJc to specify a new jc array for a sparse mxArray. The jc array is an integer array having n+1 elements, where n is the number of columns in the sparse mxArray.

If the jth column of the sparse mxArray has any nonzero elements:

- jc[j] is the index in ir, pr, and pi (if it exists) of the first nonzero element in the jth column.
- jc[j+1]-1 is the index of the last nonzero element in the jth column.

The number of nonzero elements in the jth column of the sparse mxArray is

```
jc[j+1] - jc[j];
```

For the jth column of the sparse mxArray, jc[j] is the total number of nonzero elements in all preceding columns. The last element of the jc array, jc[number of columns], is equal to nnz, which is the number of nonzero elements in the entire sparse mxArray.

For example, consider a 7-by-3 sparse mxArray named Sparrow containing six nonzero elements, created by typing

```
Sparrow = zeros(7,3);
```

```
Sparrow(2,1) = 1;
Sparrow(5,1) = 1;
Sparrow(3,2) = 1;
Sparrow(2,3) = 2;
Sparrow(5,3) = 1;
Sparrow(6,3) = 1;
Sparrow = sparse(Sparrow);
```

The contents of the `ir`, `jc`, and `pr` arrays are listed in this table.

Subscript	ir	pr	jc	Comment
(2,1)	1	1	0	Column 1 contains two nonzero elements, with rows designated by `ir[0]` and `ir[1]`
(5,1)	4	1	2	Column 2 contains one nonzero element, with row designated by `ir[2]`
(3,2)	2	1	3	Column 3 contains three nonzero elements, with rows designated by `ir[3]`,`ir[4]`, and `ir[5]`
(2,3)	1	2	6	There are six nonzero elements in all.
(5,3)	4	1		
(6,3)	5	1		

As an example of a much sparser `mxArray`, consider a 1,000-by-8 sparse `mxArray` named `Spacious` containing only three nonzero elements. The `ir`, `pr`, and `jc` arrays contain the values listed in this table.

Subscript	ir	pr	jc	Comment
(73,2)	72	1	0	Column 1 contains no nonzero elements.
(50,3)	49	1	0	Column 2 contains one nonzero element, with row designated by `ir[0]`.
(64,5)	63	1	1	Column 3 contains one nonzero element, with row designated by `ir[1]`.
			2	Column 4 contains no nonzero elements.
			2	Column 5 contains one nonzero element, with row designated by `ir[2]`.
			3	Column 6 contains no nonzero elements.
			3	Column 7 contains no nonzero elements.
			3	Column 8 contains no nonzero elements.
			3	There are three nonzero elements in all.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call `mxFree` on the pointer returned by `mxGetJc` before you call `mxSetJc`.

C Examples

See `mxsetdimensions.c` in the `mx` subdirectory of the `examples` directory. For an additional example, see `explore.c` in the `mex` subdirectory of the `examples` directory.

See Also

`mxCreateSparse`, `mxGetIr`, `mxGetJc`, `mxSetIr`, `mxFree`

mxSetM (C and Fortran)

Purpose

Set number of rows in mxArray

C Syntax

```
#include "matrix.h"
void mxSetM(mxArray *pm, mwSize m);
```

Fortran Syntax

```
mxSetM(pm, m)
mwPointer pm
mwSize m
```

Arguments

pm
: Pointer to an mxArray

m
: The desired number of rows

Description

Call mxSetM to set the number of rows in the specified mxArray. The term *rows* means the first dimension of an mxArray, regardless of the number of dimensions. Call mxSetN to set the number of columns.

You typically use mxSetM to change the shape of an existing mxArray. Note that mxSetM does not allocate or deallocate any space for the pr, pi, ir, or jc arrays. Consequently, if your calls to mxSetM and mxSetN increase the number of elements in the mxArray, you must enlarge the pr, pi, ir, and/or jc arrays. Call mxRealloc to enlarge them.

If your calls to mxSetM and mxSetN end up reducing the number of elements in the mxArray, you may want to reduce the sizes of the pr, pi, ir, and/or jc arrays in order to use heap space more efficiently. However, reducing the size is not mandatory.

C Examples

See mxsetdimensions.c in the mx subdirectory of the examples directory. For an additional example, see sincall.c in the refbook subdirectory of the examples directory.

See Also

mxGetM, mxGetN, mxSetN

Purpose

Set number of columns in mxArray

C Syntax

```
#include "matrix.h"
void mxSetN(mxArray *pm, mwSize n);
```

Fortran Syntax

```
mxSetN(pm, n)
mwPointer pm
mwSize n
```

Arguments

pm
: Pointer to an mxArray

n
: The desired number of columns

Description

Call mxSetN to set the number of columns in the specified mxArray. The term *columns* always means the second dimension of a matrix. Calling mxSetN forces an mxArray to have two dimensions. For example, if pm points to an mxArray having three dimensions, calling mxSetN reduces the mxArray to two dimensions.

You typically use mxSetN to change the shape of an existing mxArray. Note that mxSetN does not allocate or deallocate any space for the pr, pi, ir, or jc arrays. Consequently, if your calls to mxSetN and mxSetM increase the number of elements in the mxArray, you must enlarge the pr, pi, ir, and/or jc arrays.

If your calls to mxSetM and mxSetN end up reducing the number of elements in the mxArray, you may want to reduce the sizes of the pr, pi, ir, and/or jc arrays in order to use heap space more efficiently. However, reducing the size is not mandatory.

C Examples

See mxsetdimensions.c in the mx subdirectory of the examples directory. For an additional example, see sincall.c in the refbook subdirectory of the examples directory.

See Also

mxGetM, mxGetN, mxSetM

mxSetNzmax (C and Fortran)

Purpose

Set storage space for nonzero elements

C Syntax

```
#include "matrix.h"
void mxSetNzmax(mxArray *pm, mwSize nzmax);
```

Fortran Syntax

```
mxSetNzmax(pm, nzmax)
mwPointer pm
mwSize nzmax
```

Arguments

`pm`

Pointer to a sparse `mxArray`.

`nzmax`

The number of elements that `mxCreateSparse` should allocate to hold the arrays pointed to by `ir`, `pr`, and `pi` (if it exists). Set `nzmax` greater than or equal to the number of nonzero elements in the `mxArray`, but set it to be less than or equal to the number of rows times the number of columns. If you specify an `nzmax` value of `0`, `mxSetNzmax` sets the value of `nzmax` to `1`.

Description

Use `mxSetNzmax` to assign a new value to the `nzmax` field of the specified sparse `mxArray`. The `nzmax` field holds the maximum possible number of nonzero elements in the sparse `mxArray`.

The number of elements in the `ir`, `pr`, and `pi` (if it exists) arrays must be equal to `nzmax`. Therefore, after calling `mxSetNzmax`, you must change the size of the `ir`, `pr`, and `pi` arrays. To change the size of one of these arrays:

1. Call `mxRealloc` with a pointer to the array, setting the size to the new value of `nzmax`.
2. Call the appropriate `mxSet` routine (`mxSetIr`, `mxSetPr`, or `mxSetPi`) to establish the new memory area as the current one.

Two ways of determining how big you should make `nzmax` are

- Set `nzmax` equal to or slightly greater than the number of nonzero elements in a sparse `mxArray`. This approach conserves precious heap space.
- Make `nzmax` equal to the total number of elements in an `mxArray`. This approach eliminates (or, at least reduces) expensive reallocations.

C Examples

See `mxsetnzmax.c` in the `mx` subdirectory of the `examples` directory.

See Also

`mxGetNzmax`, `mxRealloc`

mxSetPi (C and Fortran)

Purpose

Set new imaginary data for mxArray

C Syntax

```
#include "matrix.h"
void mxSetPi(mxArray *pm, double *pi);
```

Fortran Syntax

```
mxSetPi(pm, pi)
mwPointer pm, pi
```

Arguments

pm

Pointer to a full (nonsparse) mxArray

pi

Pointer to the first element of an array. Each element in the array contains the imaginary component of a value. The array must be in dynamic memory; call mxCalloc to allocate this dynamic memory. If pi points to static memory, memory leaks and other memory errors may result.

Description

Use mxSetPi to set the imaginary data of the specified mxArray.

Most mxCreate* functions optionally allocate heap space to hold imaginary data. If you tell an mxCreate* function to allocate heap space—for example, by setting the ComplexFlag to mxCOMPLEX in C (1 in Fortran) or by setting pi to a non-NULL value in C (a nonzero value in Fortran)—you do not ordinarily use mxSetPi to initialize the created mxArray's imaginary elements. Rather, you call mxSetPi to replace the initial imaginary values with new ones.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetPi before you call mxSetPi.

C Examples

See mxisfinite.c and mxsetnzmax.c in the mx subdirectory of the examples directory.

See Also

mxGetPi, mxGetPr, mxSetImagData, mxSetPr, mxFree

Purpose

Set new real data for mxArray

C Syntax

```
#include "matrix.h"
void mxSetPr(mxArray *pm, double *pr);
```

Fortran Syntax

```
mxSetPr(pm, pr)
mwPointer pm, pr
```

Arguments

pm

Pointer to a full (nonsparse) mxArray

pr

Pointer to the first element of an array. Each element in the array contains the real component of a value. The array must be in dynamic memory; call mxCalloc to allocate this dynamic memory. If pr points to static memory, memory leaks and other memory errors can result.

Description

Use mxSetPr to set the real data of the specified mxArray.

All mxCreate* calls allocate heap space to hold real data. Therefore, you do not ordinarily use mxSetPr to initialize the real elements of a freshly created mxArray. Rather, you call mxSetPr to replace the initial real values with new ones.

This function does not free any memory allocated for existing data that it displaces. To free existing memory, call mxFree on the pointer returned by mxGetPr before you call mxSetPr.

C Examples

See mxsetnzmax.c in the mx subdirectory of the examples directory.

See Also

mxGetPi, mxGetPr, mxSetData, mxSetPi, mxFree

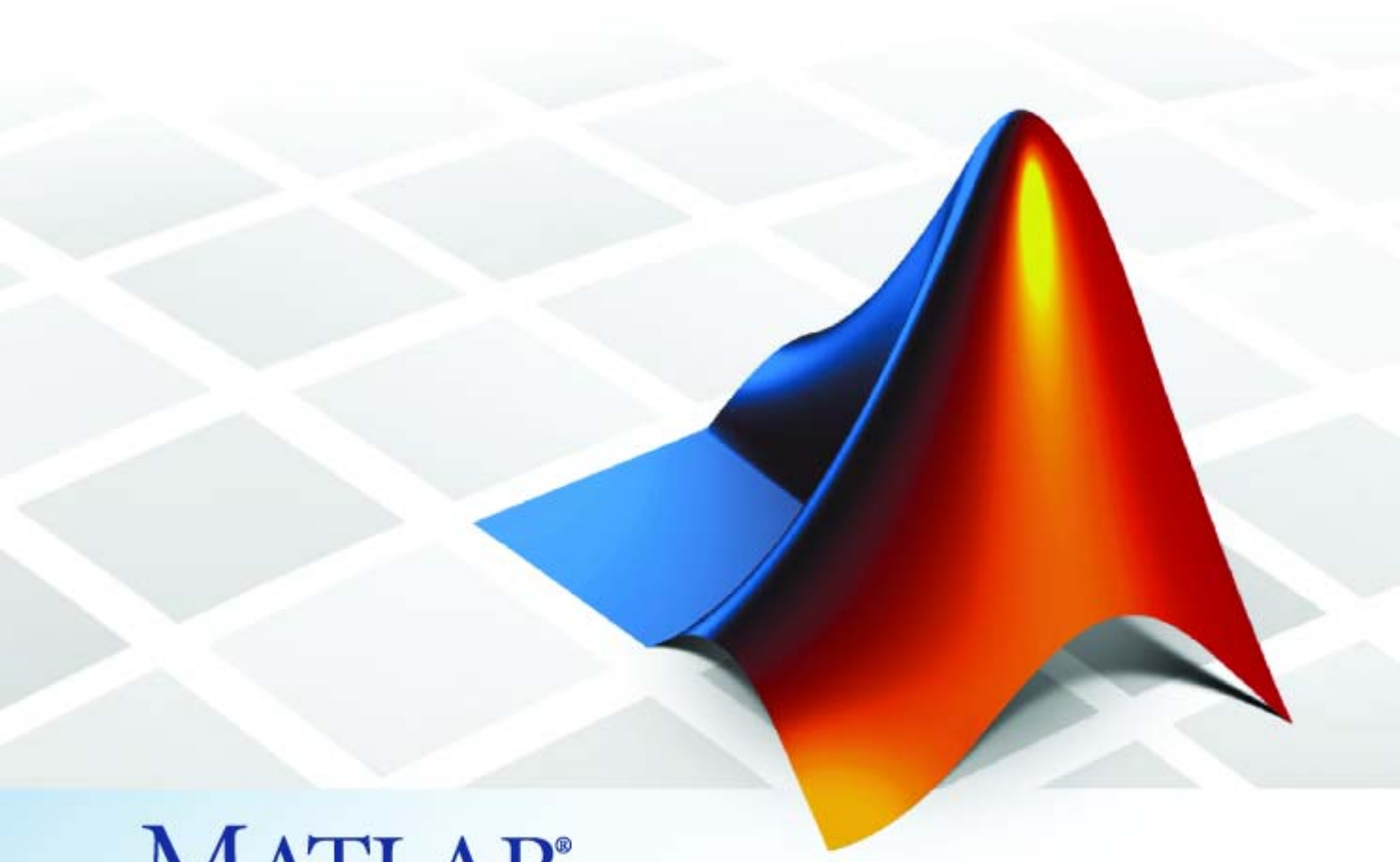
Index

O

P

S

MATLAB® 7
Creating Graphical User Interfaces



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Creating Graphical User Interfaces

Revision History

November 2000	Online Only	New for MATLAB 6.0 (Release 12)
June 2001	Online Only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online Only	Revised for MATLAB 6.6 (Release 13)
June 2004	Online Only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online Only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online Only	Revised for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online Only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online Only	Revised for MATLAB 7.2 (Release 2006a)
May 2006	Online Only	Revised for MATLAB 7.2
September 2006	Online Only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online Only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online Only	Revised for MATLAB 7.5 (Release 2007b)

Contents

1 About GUIs in MATLAB

2 Creating a Simple GUI with GUIDE

3 Creating a Simple GUI Programmatically

4 What Is GUIDE?

5 GUIDE Preferences and Options

6 Laying Out a GUIDE GUI

7 Saving and Running a GUIDE GUI

8 Programming a GUIDE GUI

9 Managing and Sharing Application Data in GUIDE

10 Examples of GUIDE GUIs

11 Laying Out a GUI

12 Programming the GUI

13 Managing Application-Defined Data

14 Managing Callback Execution

15 Examples of GUIs Created Programmatically

A Examples

Introduction to Creating GUIs

Chapter 1, About GUIs in MATLAB (p. 1-1)	Explains what a GUI is, how a GUI works, and how to get started creating a GUI.
Chapter 2, Creating a Simple GUI with GUIDE (p. 2-1)	Steps you through the process of creating a simple GUI using GUIDE.
Chapter 3, Creating a Simple GUI Programmatically (p. 3-1)	Steps you through the process of creating a simple GUI programmatically.

1

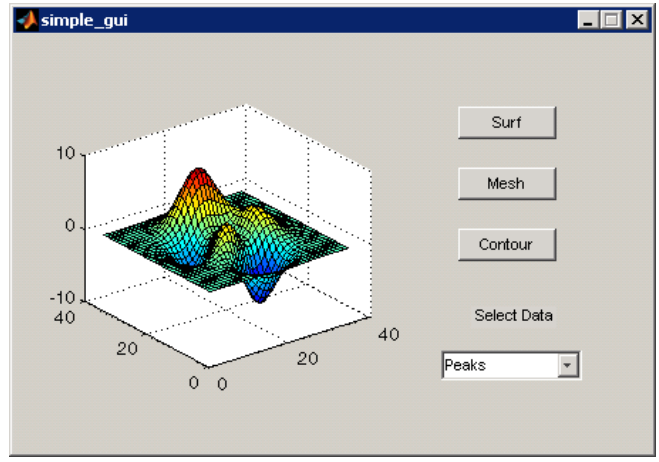
About GUIs in MATLAB

What Is a GUI?

A graphical user interface (GUI) is a graphical display that contains devices, or components, that enable a user to perform interactive tasks. To perform these tasks, the user of the GUI does not have to create a script or type commands at the command line. Often, the user does not have to know the details of the task at hand.

The GUI components can be menus, toolbars, push buttons, radio buttons, list boxes, and sliders—just to name a few. In MATLAB, a GUI can also display data in tabular form or as plots, and can group related components.

The following figure illustrates a simple GUI.



The GUI contains

- An axes component
- A pop-up menu listing three data sets that correspond to MATLAB functions: `peaks`, `membrane`, and `sinc`
- A static text component to label the pop-up menu
- Three buttons that provide different kinds of plots: surface, mesh, and contour

When you click a push button, the axes component displays the selected data set using the specified plot.

How Does a GUI Work?

Each component, and the GUI itself, is associated with one or more user-written routines known as callbacks. The execution of each callback is triggered by a particular user action such as a button push, mouse click, selection of a menu item, or the cursor passing over a component. You, as the creator of the GUI, provide these callbacks.

In the GUI described in “What Is a GUI?” on page 1-2, the user selects a data set from the pop-up menu, then clicks one of the plot type buttons. Clicking the button triggers the execution of a callback that plots the selected data in the axes.

This kind of programming is often referred to as event-driven programming. The event in the example is a button click. In event-driven programming, callback execution is asynchronous, controlled by events external to the software. In the case of MATLAB GUIs, these events usually take the form of user interactions with the GUI.

The writer of a callback has no control over the sequence of events that leads to its execution or, when the callback does execute, what other callbacks might be running simultaneously.

Where Do I Start?

First you have to design your GUI. You have to decide what you want it to do, how you want the user to interact with it, and what components you need. “Designing a GUI” on page 6-3 lists references that may be of help.

Next, you must decide what technique you want to use to create your GUI. MATLAB enables you to create GUIs programmatically or with GUIDE, an interactive GUI builder. It also provides functions that simplify the creation of standard dialog boxes. The technique you choose depends on your experience, your preferences, and the kind of GUI you want to create. This table outlines some possibilities.

GUI	Technique
Dialog box	MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For links to these functions, see “Predefined Dialog Boxes” in the MATLAB Function Reference documentation.
GUI containing just a few components	It is often simpler to create GUIs that contain only a few components programmatically. Each component can be fully defined with a single function call.
Moderately complex GUIs	GUIDE simplifies the creation of such GUIs.
Complex GUIs with many components, and GUIs that require interaction with other GUIs	Creating such GUIs programmatically lets you control exact placement of the components and provides reproducibility.

Once you have decided which technique you want to use, you can continue to learn about creating GUIs in MATLAB by following the examples in these topics:

- Chapter 2, “Creating a Simple GUI with GUIDE”
- Chapter 3, “Creating a Simple GUI Programmatically”

2

Creating a Simple GUI with GUIDE

GUIDE: A Brief Introduction

In this section...

Laying Out a GUI

GUIDE, the MATLAB graphical user interface development environment, provides a set of tools for creating graphical user interfaces (GUIs). These tools simplify the process of laying out and programming GUIs.

The GUIDE Layout Editor enables you to populate a GUI by clicking and dragging GUI components — such as buttons, text fields, sliders, axes, and so on — into the layout area. It also enables you to create menus and context menus for the GUI.

Other tools, which are accessible from the Layout Editor, enable you to size the GUI, modify component look and feel, align components, set tab order, view a hierarchical list of the component objects, and set GUI options.

The following topic, "Laying Out a Simple GUI" on page 2-5, uses some of these tools to show you the basics of laying out a GUI. "GUIDE Tools Summary" on page 4-3 describes the tools.

Programming a GUI

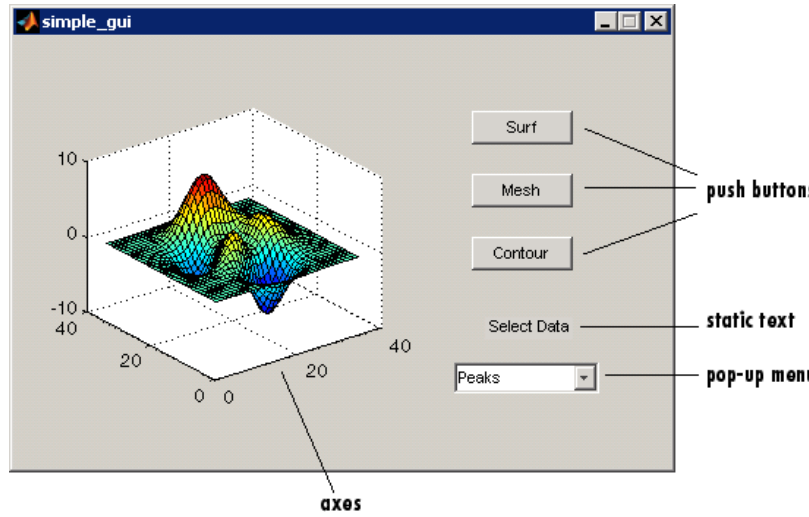
When you save your GUI layout, GUIDE automatically generates an M-file that you can use to control how the GUI works. This M-file provides code to initialize the GUI and contains a framework for the GUI callbacks—the routines that execute in response to user-generated events such as a mouse click. Using the M-file editor, you can add code to the callbacks to perform the functions you want. "Programming a Simple GUI" on page 2-21 shows you what code to add to the example M-file to make the GUI work.

Example: Simple GUI

In this section...

Simple GUI Overview

This section shows you how to use GUIDE to create the graphical user interface (GUI) shown in the following figure.



The GUI contains

- An axes component
- A pop-up menu listing three different data sets that correspond to MATLAB functions: `peaks`, `membrane`, and `sinc`
- A static text component to label the pop-up menu
- Three push buttons, each of which provides a different kind of plot: surface, mesh, and contour

To use the GUI, select a data set from the pop-up menu, then click one of the plot-type buttons. Clicking the button triggers the execution of a callback that plots the selected data in the axes.

Subsequent topics, starting with “Laying Out a Simple GUI” on page 2-5, guide you through the steps to create this GUI. We recommend that you create the GUI for yourself, as this is the best way to learn how to use GUIDE.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

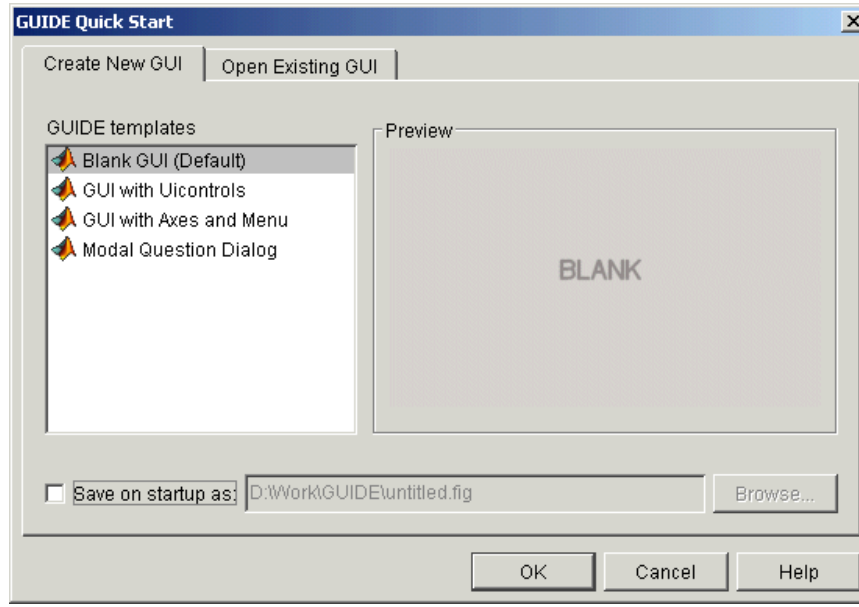
- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the MATLAB Editor.

Laying Out a Simple GUI

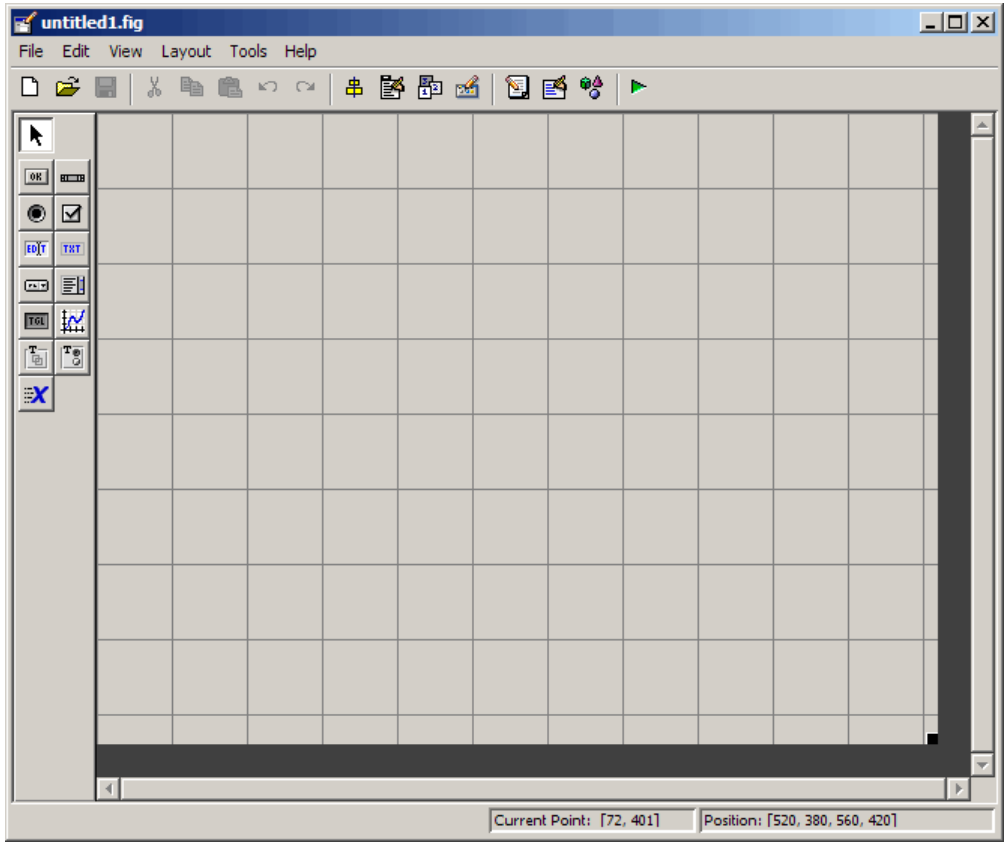
In this section...

Opening a New GUI in the Layout Editor

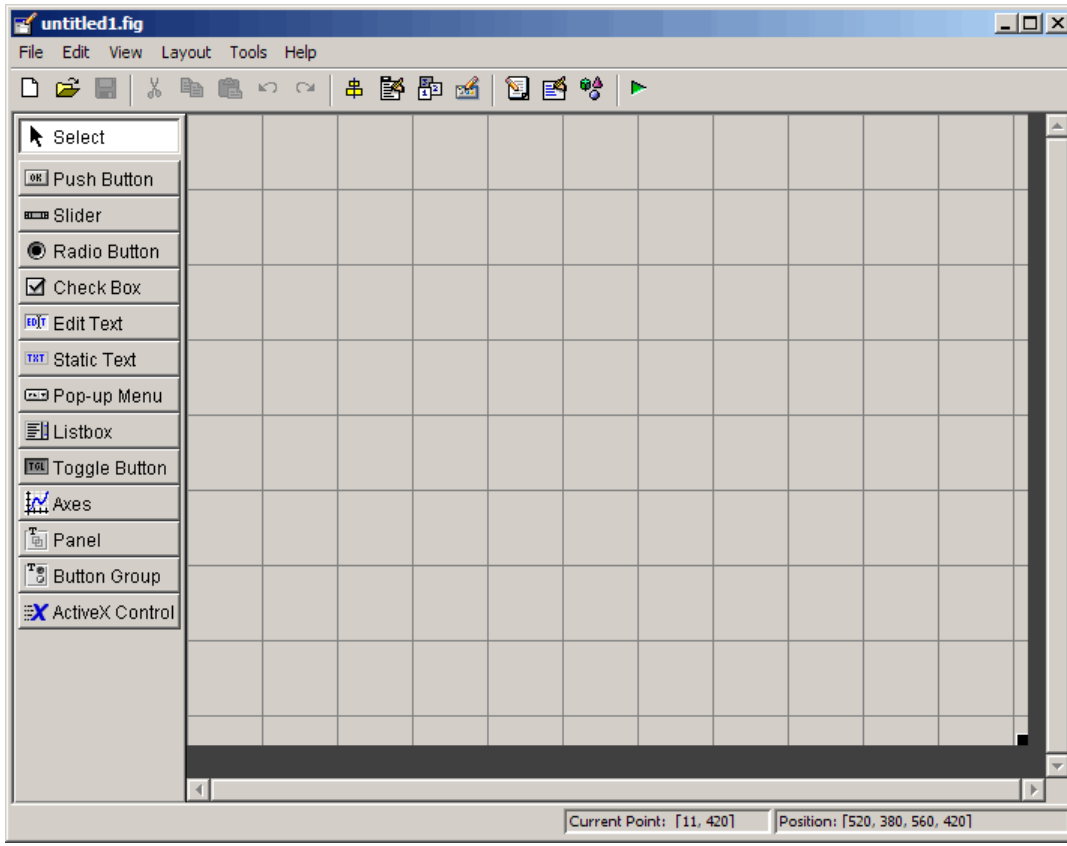
1 Start GUIDE by typing `guide` at the MATLAB prompt. This displays the GUIDE Quick Start dialog shown in the following figure.



2 In the Quick Start dialog, select the **Blank GUI (Default)** template. Click **OK** to display the blank GUI in the Layout Editor, as shown in the following figure.

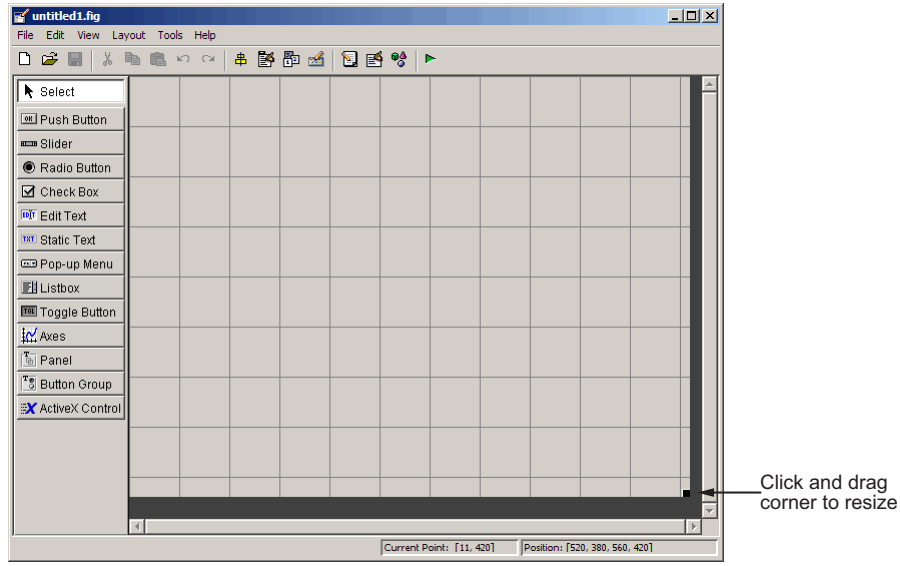


3 Display the names of the GUI components in the component palette. Select **Preferences** from the MATLAB **File** menu. Then select **GUIDE > Show names in component palette**, and click **OK**. The Layout Editor then appears as shown in the following figure.



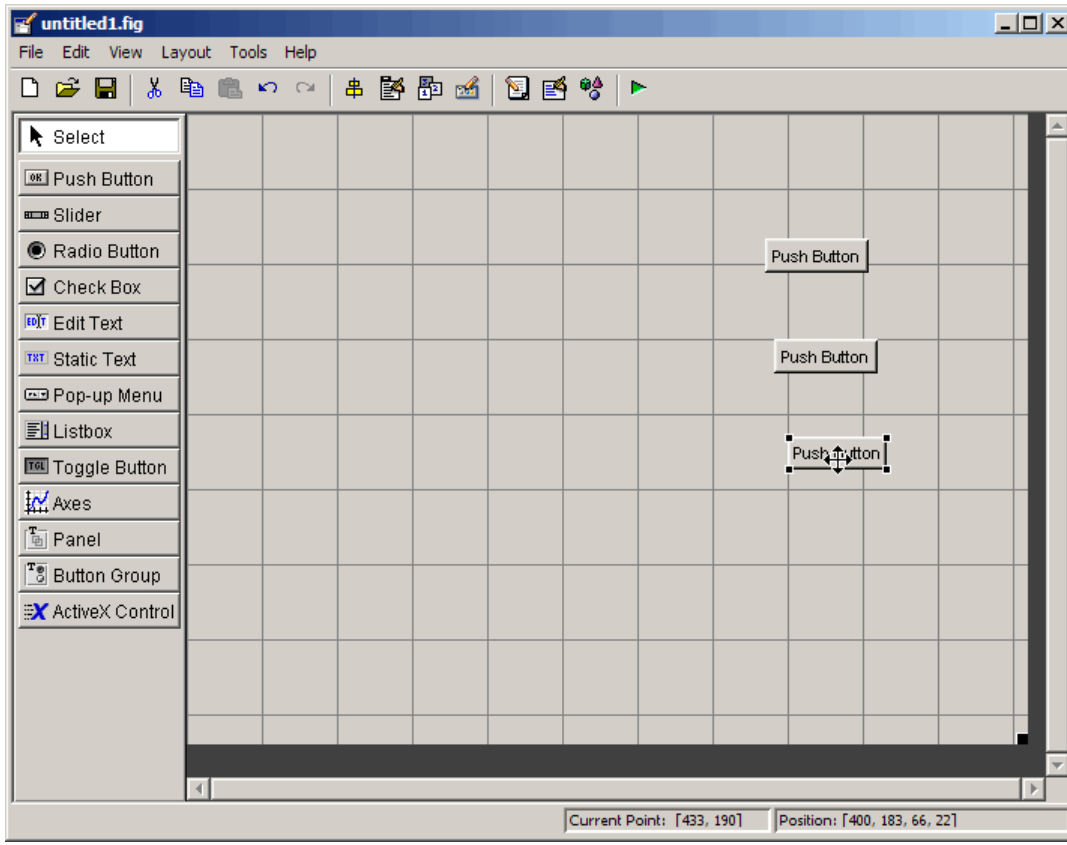
Setting the GUI Figure Size

Set the size of the GUI by resizing the grid area in the Layout Editor. Click the lower-right corner and drag it until the GUI is approximately 3 inches high and 4 inches wide. If necessary, make the window larger.



Adding the Components

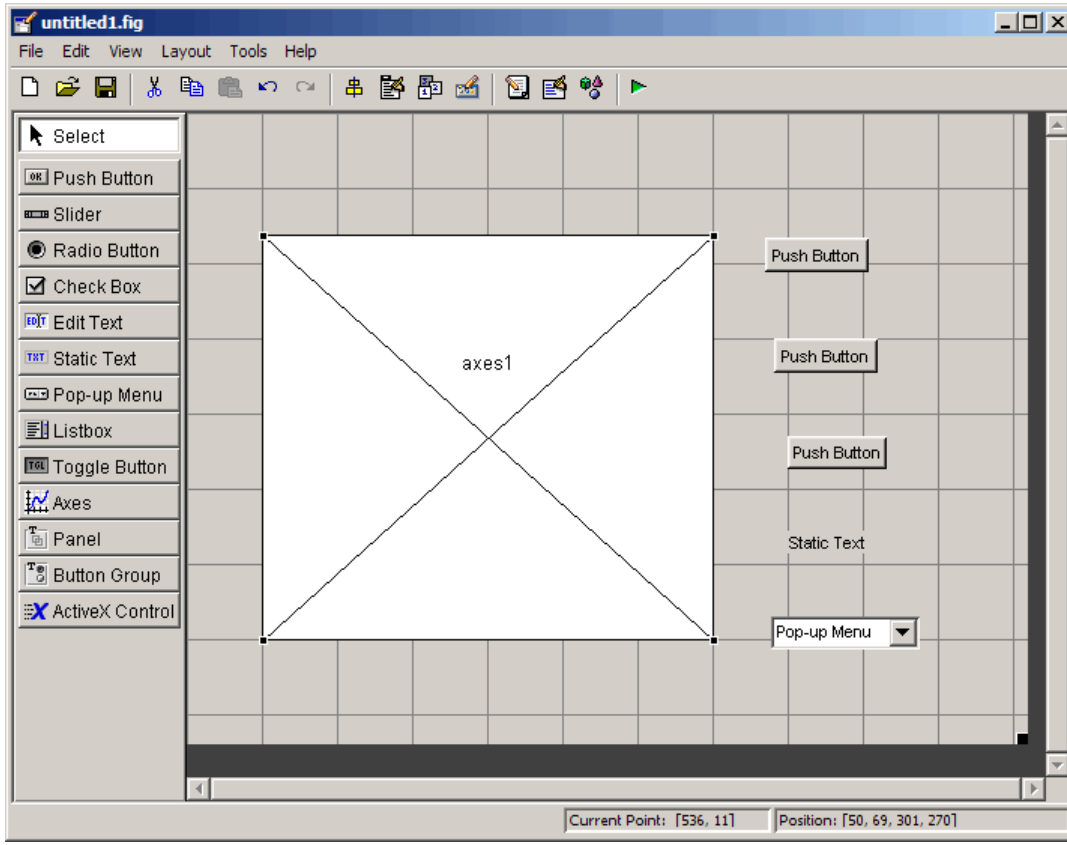
1 Add the three push buttons to the GUI. For each push button, select the push button from the component palette at the left of the Layout Editor and drag it into the layout area. Position them approximately as shown in the following figure.



2 Add the remaining components to the GUI.

- A static text area
- A pop-up menu
- An axes

Arrange the components as shown in the following figure. Resize the axes component to approximately 2-by-2 inches.



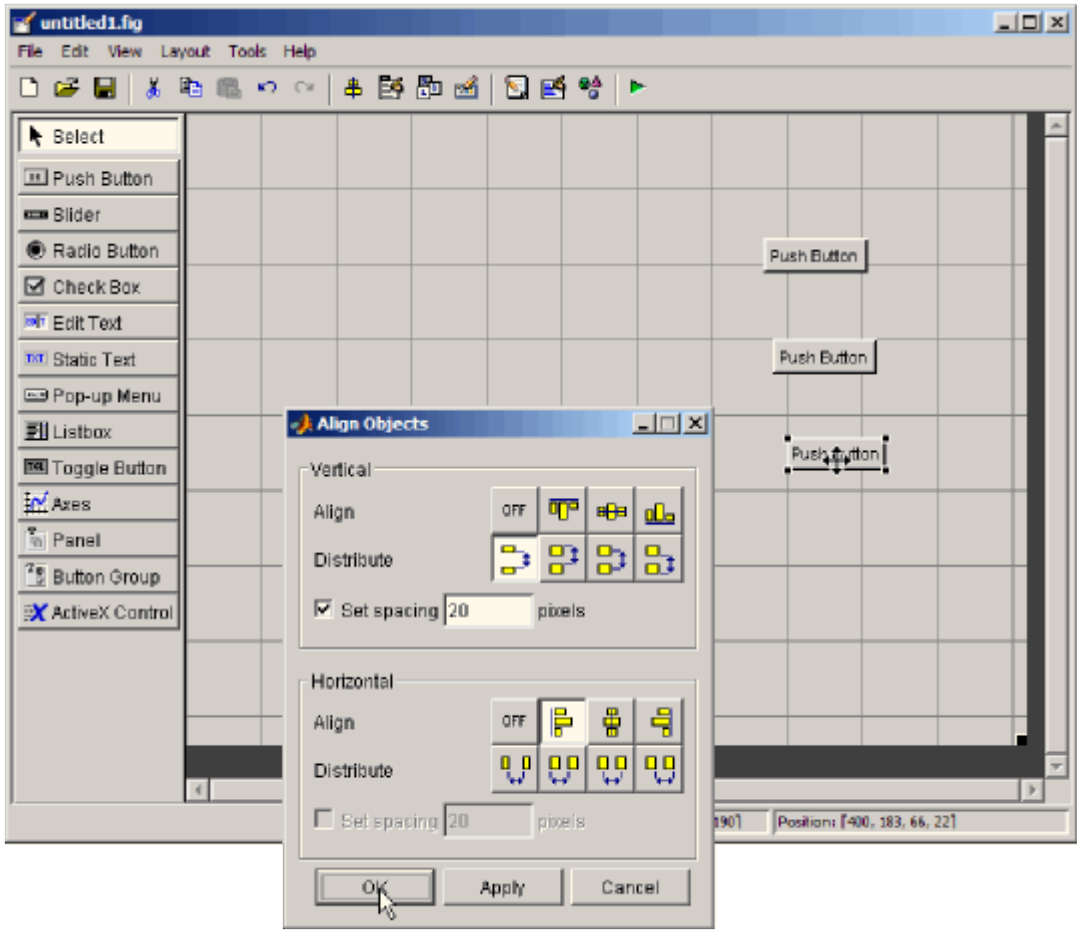
Aligning the Components

You can use the Alignment Tool to align components with respect to one another, if they have the same parent. To align the three push buttons:

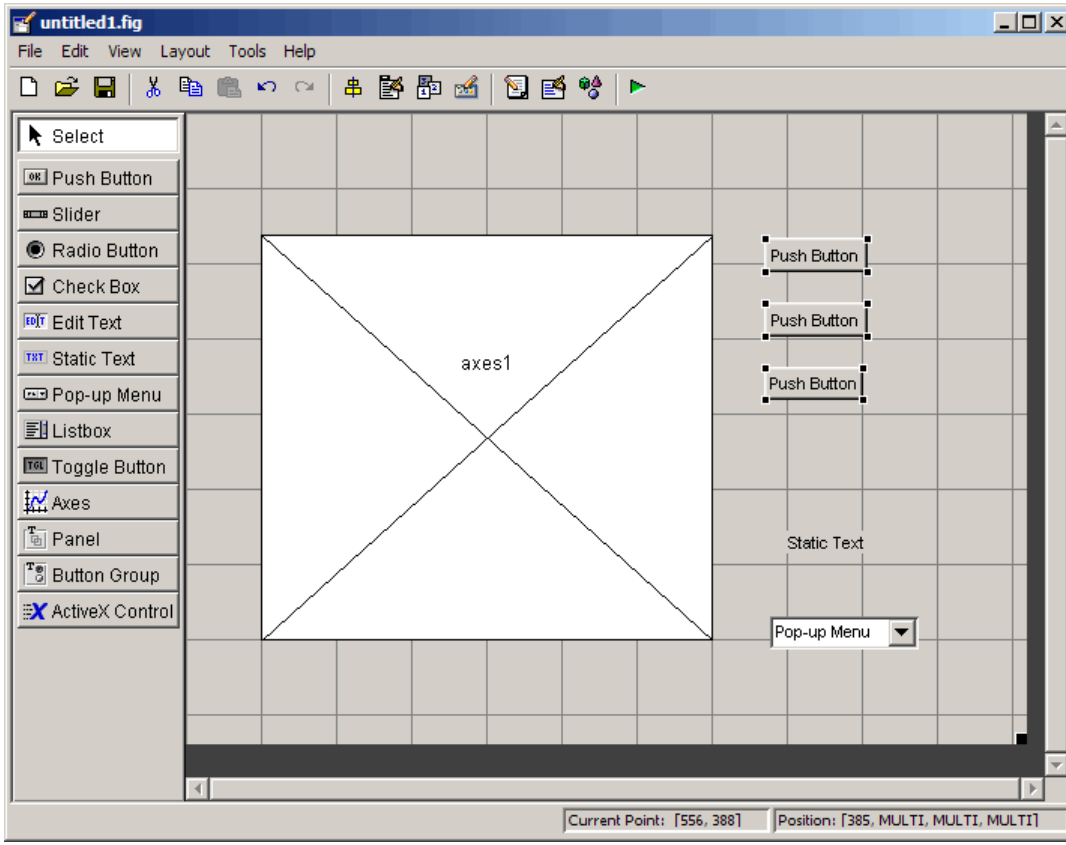
1. Select all three push buttons by pressing **Ctrl** and clicking them.
2. Select **Align Objects** from the **Tools** menu to display the Alignment Tool.

3 Make these settings in the Alignment Tool, as shown in the following figure:

- 20 pixels spacing between push buttons in the vertical direction.
- Left-aligned in the horizontal direction.



4 Click **OK**. Your GUI now looks like this in the Layout Editor.

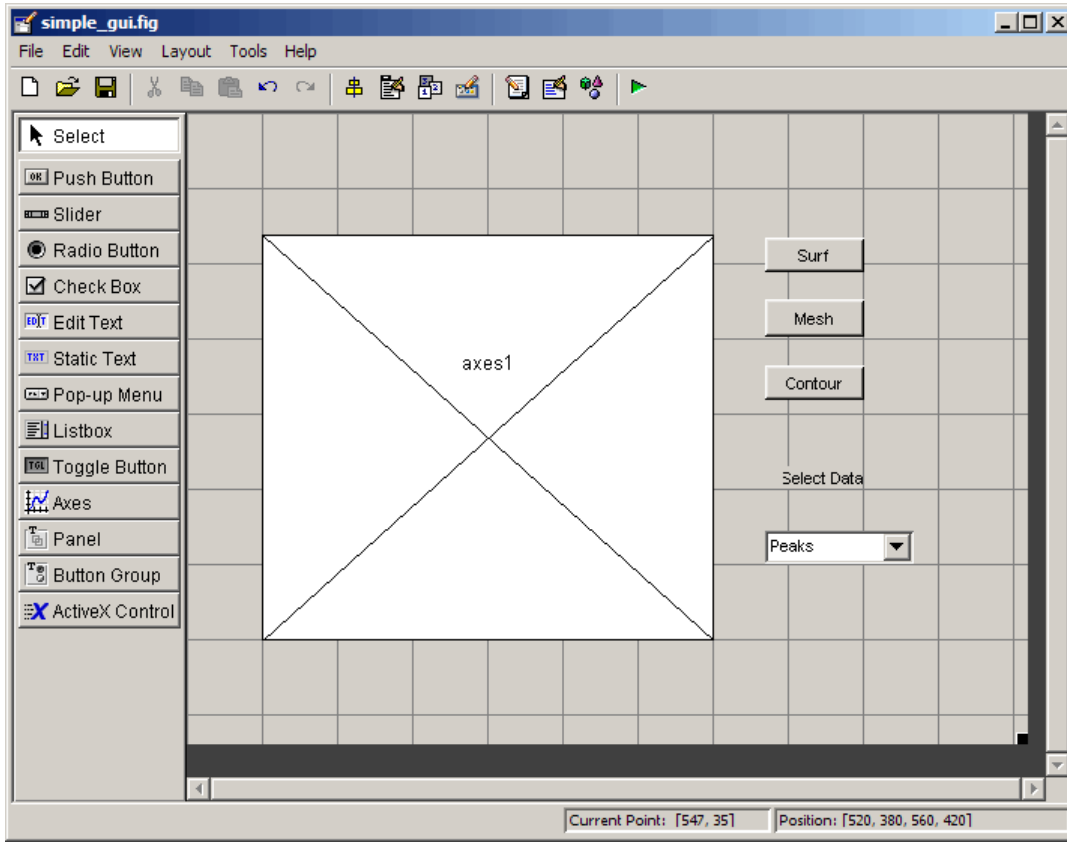


Adding Text to the Components

Although the push buttons, pop-up menu, and static text show some text in the Layout Editor, the text is not appropriate to the GUI being created. This topic shows you how to modify the default text.

- "Labeling the Push Buttons" on page 2-13
- "Entering Pop-Up Menu Items" on page 2-15
- "Modifying the Static Text" on page 2-17

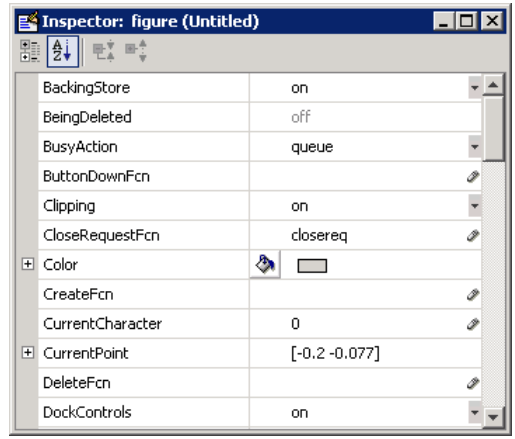
After you have added the appropriate text, the GUI will look like this in the Layout Editor.



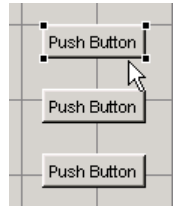
Labeling the Push Buttons

Each of the three push buttons lets the user choose a plot type: surf, mesh, and contour. This topic shows you how to label the buttons with those choices.

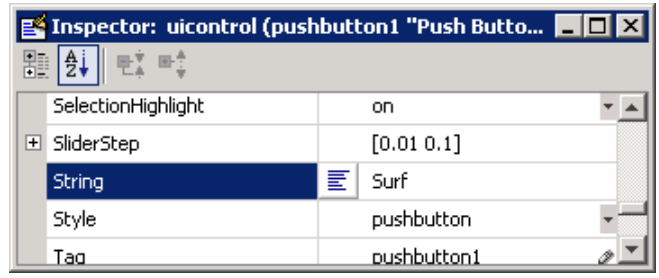
1 Select **Property Inspector** from the **View** menu.



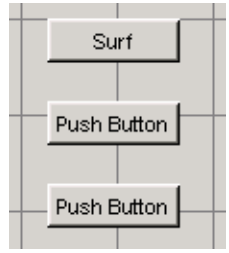
2 In the layout area, select the top push button by clicking it.



3 In the Property Inspector, select the `String` property and then replace the existing value with the word `Surf`.



4 Click outside the `String` field. The push button label changes to **Surf**.



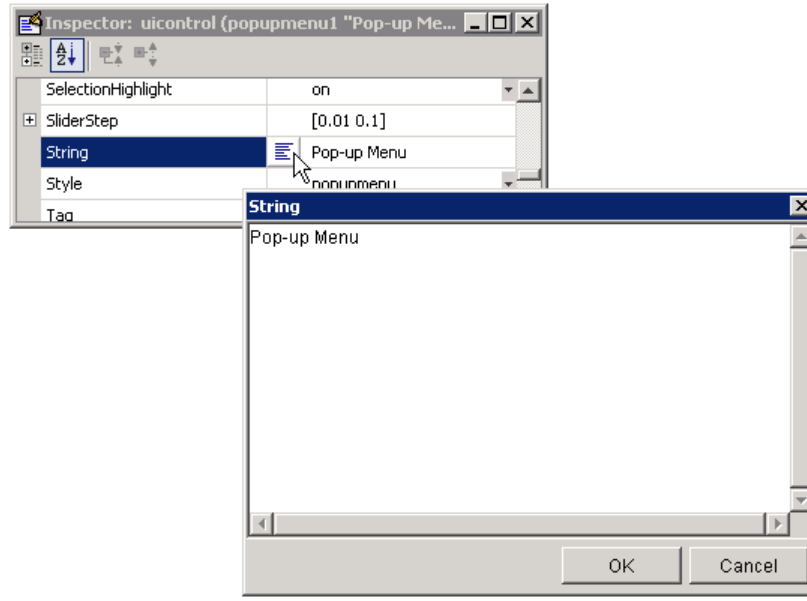
5 Select each of the remaining push buttons in turn and repeat steps 3 and 4. Label the middle push button **Mesh**, and the bottom button **Contour**.

Entering Pop-Up Menu Items

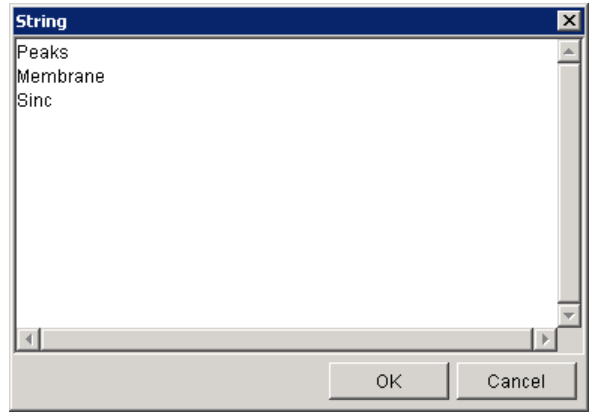
The pop-up menu provides a choice of three data sets: peaks, membrane, and sinc. These data sets correspond to MATLAB functions of the same name. This topic shows you how to list those data sets as choices in the pop-menu.

1 In the layout area, select the pop-up menu by clicking it.

2 In the Property Inspector, click the button next to `String`. The String dialog box displays.



3 Replace the existing text with the names of the three data sets: Peaks, Membrane, and Sinc. Press **Enter** to move to the next line.



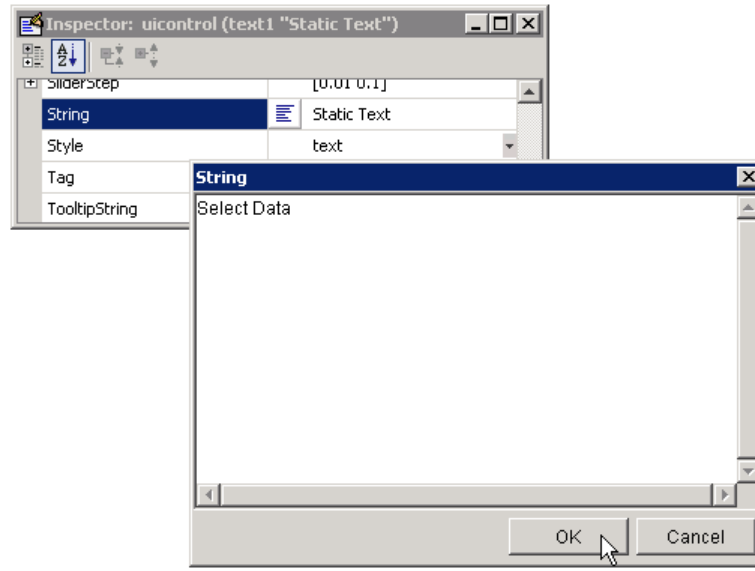
4 When you are done, click **OK**. The first item in your list, `Peaks`, appears in the pop-up menu in the layout area.

Modifying the Static Text

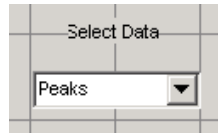
In this GUI, the static text serves as a label for the pop-up menu. The user cannot change this text. This topic shows you how to change the static text to read `Select Data`.

1 In the layout area, select the static text by clicking it.

2 In the Property Inspector, click the button next to `String`. In the String dialog box that displays, replace the existing text with the phrase `Select Data`.

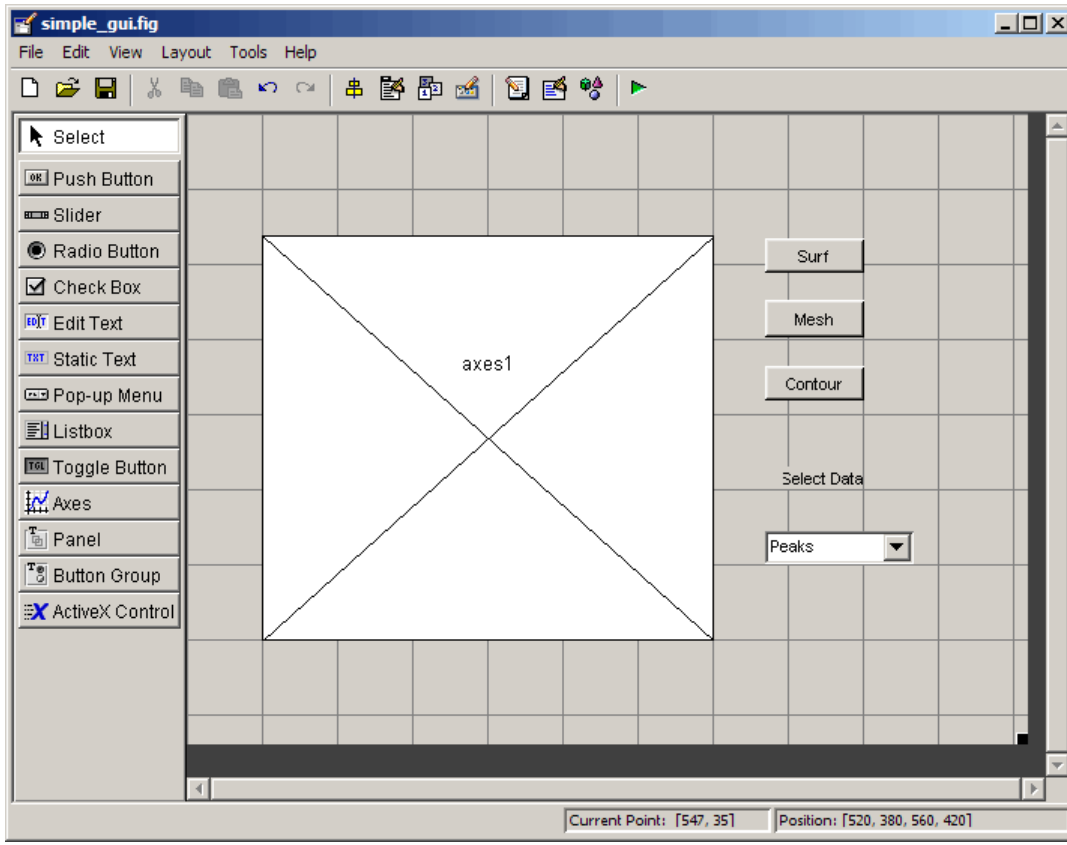


3 Click **OK**. The phrase `Select Data` appears in the static text component above the pop-up menu.



Completed Layout

In the Layout Editor, your GUI now looks like this and the next step is to save the layout. The next topic, “Saving the GUI Layout” on page 2-19, tells you how to do this.

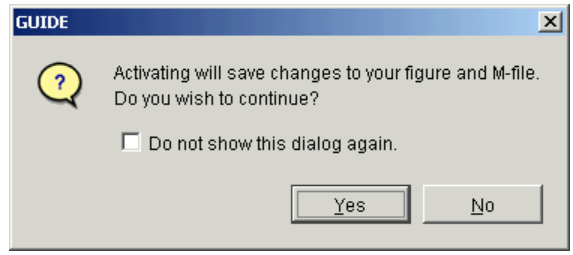


Saving the GUI Layout

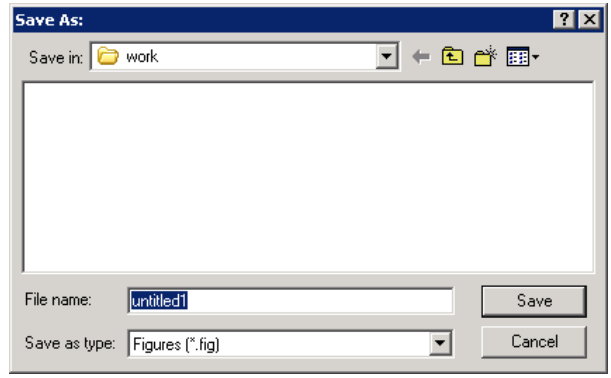
When you save a GUI, GUIDE creates two files, a FIG-file and an M-file. The FIG-file, with extension `.fig`, is a binary file that contains a description of the layout. The M-file, with extension `.m`, contains the code that controls the GUI.

1. Save and activate your GUI by selecting **Run** from the **Tools** menu.

2. GUIDE displays the following dialog box. Click **Yes** to continue.



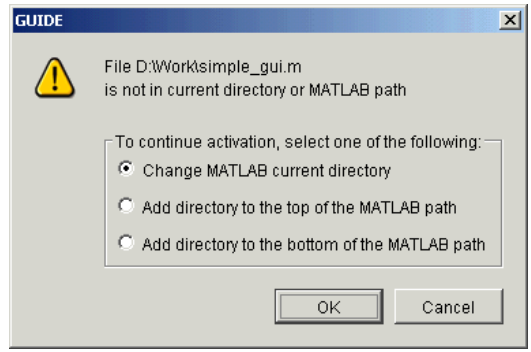
3. GUIDE opens a **Save As** dialog box in your current directory and prompts you for a FIG-file name.



4. Browse to any directory for which you have write privileges, and then enter the filename `simple_gui` for the FIG-file. GUIDE saves both the FIG-file and the M-file using this name.

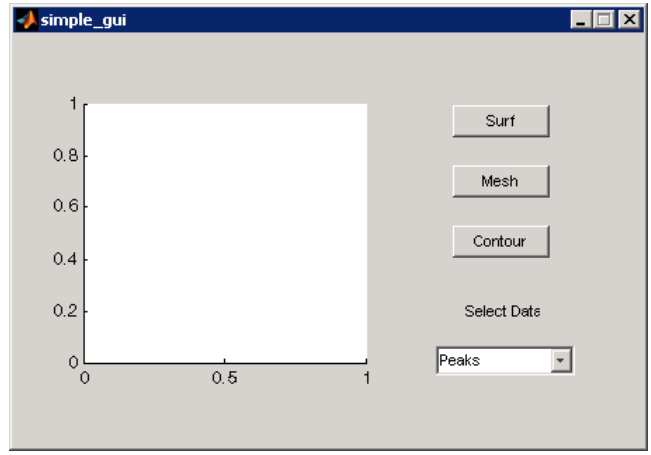
5. If the directory in which you save the GUI is not on the MATLAB path, GUIDE opens a dialog box, giving you the option of changing the current

working directory to the directory containing the GUI files, or adding that directory to the top or bottom of the MATLAB path.



6 GUIDE saves the files `simple_gui.fig` and `simple_gui.m` and activates the GUI. It also opens the GUI M-file in your default editor.

The GUI is active. You can select a data set in the pop-up menu and click the push buttons. But nothing happens. This is because there is no code in the M-file to service the pop-up menu and the buttons. The next step is to program the GUI. The next topic, “Programming a Simple GUI” on page 2-21, shows you how to do this.



Programming a Simple GUI

In this section...

Adding Code to the M-file

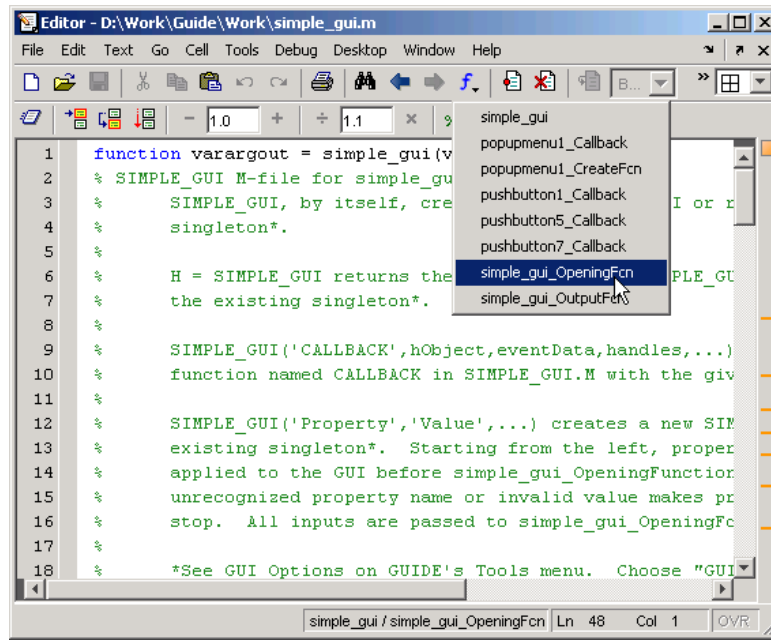
When you saved your GUI in the previous topic, “Saving the GUI Layout” on page 2-19, GUIDE created two files: a FIG-file `simple_gui.fig` that contains the GUI layout, and an M-file `simple_gui.m` that contains the code that controls the GUI. But the GUI didn’t do anything because there was no code in the M-file to make it work. This topic shows you how to add code to the M-file to make it work. There are three steps:

Generating Data to Plot

This topic shows you how to generate the data to be plotted when the user clicks a button. This data is generated in the *opening function*. The opening function is the first callback in every GUIDE-generated GUI M-file. You can use it to perform tasks that need to be done before the user has access to the GUI.

In this example, you add code that creates three data sets to the opening function. The code uses the MATLAB functions `peaks`, `membrane`, and `sinc`.

1. Display the opening function in the M-file editor. If the GUI M-file, `simple_gui.m`, is not already open in your editor, open it by selecting **M-file Editor** from the **View** menu. In the editor, click the function icon on the toolbar, then select **simple_gui_OpeningFcn** in the pop-up menu that displays.



The cursor moves to the opening function, which already contains this code:

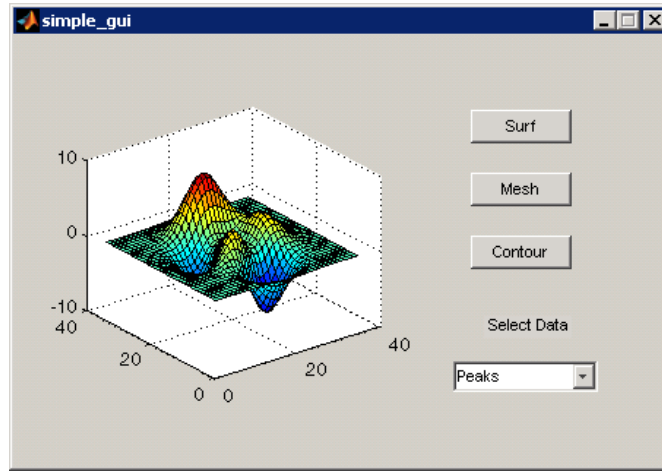
```
% --- Executes just before simple_gui is made visible.
function simple_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to simple_gui (see VARARGIN)

% Choose default command line output for simple_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes simple_gui wait for user response (see UIRESUME)
% uiwait(handles.figure1);
```

2 Create data for the GUI to plot by adding the following code to the opening function immediately after the comment that begins `% varargin...`

```
% Create the data to plot.
handles.peaks=peaks(35);
handles.membrane=membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc = sin(r)./r;
handles.sinc = sinc;
% Set the current data value.
handles.current_data = handles.peaks;
surf(handles.current_data)
```

The first six executable lines create the data using the MATLAB functions `peaks`, `membrane`, and `sinc`. They store the data in the `handles` structure, which is passed as an argument to all callbacks. Callbacks for the push buttons can retrieve the data from the `handles` structure.

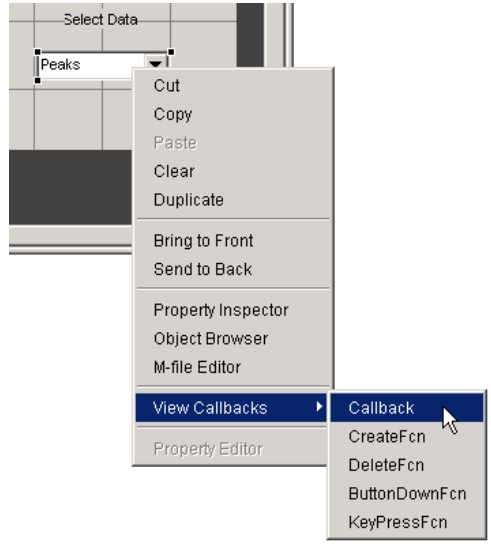
The last two lines create a current data value and set it to peaks, and then display the surf plot for peaks. The following figure shows how the GUI now looks when it first displays.



Programming the Pop-Up Menu

The pop-up menu enables the user to select the data to plot. When the GUI user selects one of the three plots, MATLAB sets the pop-up menu `Value` property to the index of the selected string. The pop-up menu callback reads the pop-up menu `Value` property to determine what item is currently displayed and sets `handles.current_data` accordingly.

1. Display the pop-up menu callback in the M-file editor. Right-click the pop-up menu component in the Layout Editor to display a context menu. From that menu, select **View Callbacks > Callback**.



   The GUI M-file opens in the editor if it is not already open, and the cursor moves to the pop-menu callback, which already contains this code:

```
% --- Executes on selection change in popupmenu1.
function popupmenu1_Callback(hObject, eventdata, handles)
% hObject    handle to popupmenu1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

2. Add the following code to the `popupmenu1_Callback` after the comment that begins `% handles...`

This code first retrieves two pop-up menu properties:

- `String` — a cell array that contains the menu contents
- `Value` — the index into the menu contents of the selected data set

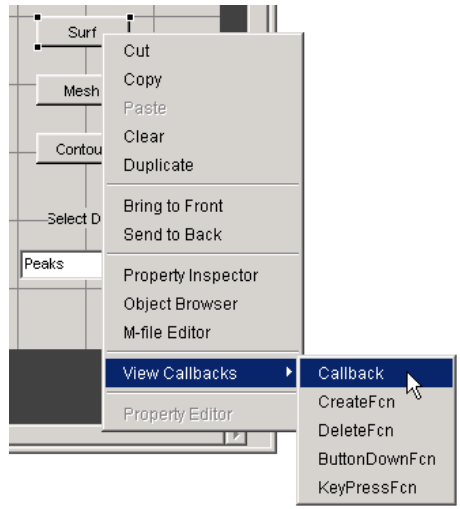
It then uses a `switch` statement to make the selected data set the current data. The last statement saves the changes to the `handles` structure.

```
% Determine the selected data set.
str = get(hObject, 'String');
val = get(hObject,'Value');
% Set current data to the selected data set.
switch str{val};
case 'Peaks' % User selects peaks.
   handles.current_data = handles.peaks;
case 'Membrane' % User selects membrane.
   handles.current_data = handles.membrane;
case 'Sinc' % User selects sinc.
   handles.current_data = handles.sinc;
end
% Save the handles structure.
guidata(hObject,handles)
```

Programming the Push Buttons

Each of the push buttons creates a different type of plot using the data specified by the current selection in the pop-up menu. The push button callbacks get data from the `handles` structure and then plot it.

1 Display the **Surf** push button callback in the M-file editor. Right-click the **Surf** push button in the Layout Editor to display a context menu. From that menu, select **View Callbacks > Callback**.



The GUI M-file opens in the editor if it is not already open, and the cursor moves to the **Surf** push button callback, which already contains this code:

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

2 Add the following code to the callback immediately after the comment that begins `% handles...`

```
% Display surf plot of the currently selected data.
surf(handles.current_data);
```

3 Repeat steps 1 and 2 to add similar code to the **Mesh** and **Contour** push button callbacks.

- Add this code to the **Mesh** push button callback, `pushbutton2_Callback`:

```
% Display mesh plot of the currently selected data.
mesh(handles.current_data);
```

- Add this code to the **Contour** push button callback, `pushbutton3_Callback`:

```
% Display contour plot of the currently selected data.
contour(handles.current_data);
```

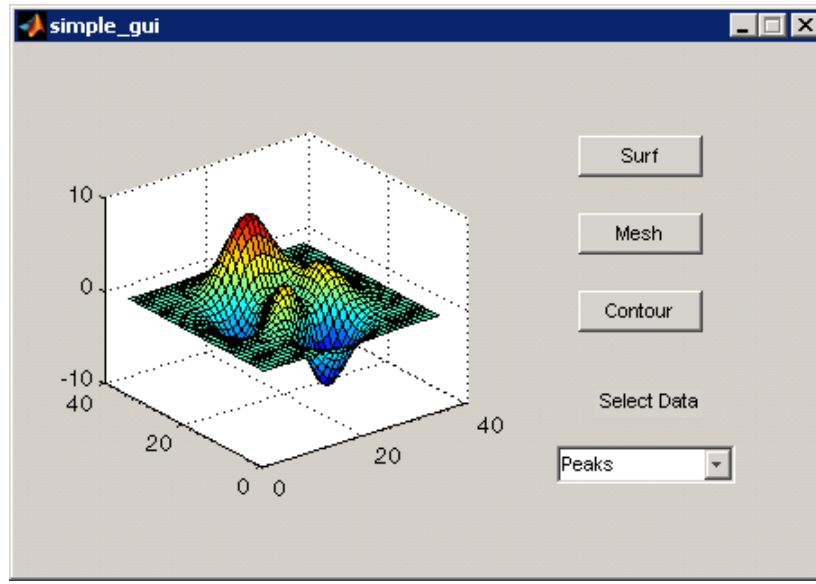
4 Save the M-file by selecting **Save** from the **File** menu.

Your GUI is ready to run. The next topic, “Running the GUI” on page 2-28, tells you how to do that.

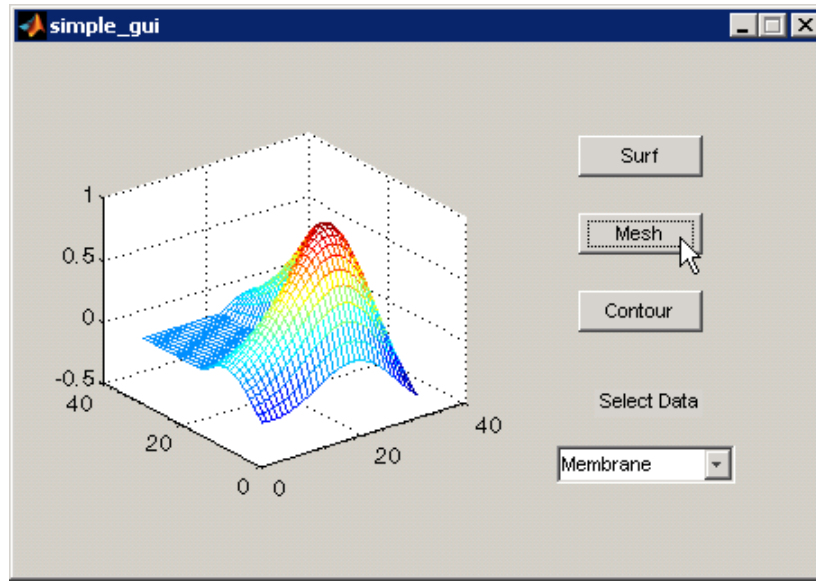
Running the GUI

In the previous topic, you programmed the pop-up menu and the push buttons. You also created data for them to use and initialized the display. Now you can run your GUI and see how it works.

1 Run your GUI by selecting **Run** from the Layout Editor **Tools** menu. If the GUI is on your MATLAB path or in your current directory, you can also run it by typing its name, `simple_gui`, at the prompt. The GUI looks like this when it first displays:



2 In the pop-up menu, select **Membrane**, then click the **Mesh** button. The GUI displays a mesh plot of the MATLAB logo.



3 Try other combinations before closing the GUI.

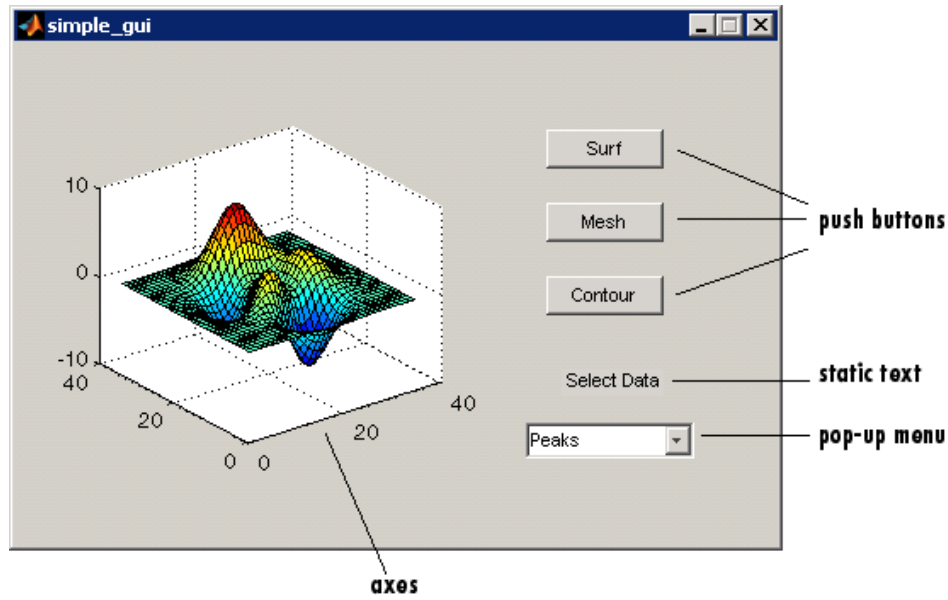
3

Creating a Simple GUI Programmatically

Example: Simple GUI

Simple GUI Overview

This section shows you how to write a script that creates the example graphical user interface (GUI) shown in the following figure.



The GUI contains

- An axes
- A pop-up menu listing three data sets that correspond to MATLAB functions: `peaks`, `membrane`, and `sinc`
- A static text component to label the pop-up menu
- Three push buttons, each of which provides a different kind of plot: surface, mesh, and contour

To use the GUI, the user selects a data set from the pop-up menu, then clicks one of the plot-type push buttons. Clicking the button triggers the execution of a callback that plots the selected data in the axes.

The next topic, “Function Summary” on page 3-4, summarizes the functions used to create this example GUI.

Subsequent topics guide you through the process of creating the GUI. This process begins with “Creating a GUI M-File” on page 3-5. We recommend that you create the GUI for yourself.

View Completed Example

If you are reading this in the MATLAB Help browser, you can click the following links to display the example GUI and its M-file.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display the example GUI.
- Click here to display the GUI M-file in the MATLAB Editor.

Function Summary

MATLAB provides a suite of functions for creating GUIs. This topic introduces you to the functions you need to create the example GUI.

Functions Used to Create the Simple GUI

Function	Description
`align`	Align GUI components such as user interface controls and axes.
`axes`	Create axes objects.
`figure`	Create figure objects. A GUI is a figure object.
`movegui`	Move GUI figure to specified location on screen.
`uicontrol`	Create user interface control objects, such as push buttons, static text, and pop-up menus.

Other MATLAB Functions Used to Program the GUI

Function	Description
`contour`	Contour graph of a matrix
`eps`	Floating point relative accuracy
`get`	Query object properties
`membrane`	Generate the MATLAB logo
`mesh`	Mesh plot
`meshgrid`	Generate X and Y arrays for 3-D plots
`peaks`	Example function of two variables.
`set`	Set object properties
`sin`	Sine; result in radians
`sqrt`	Square root
`surf`	3-D shaded surface plot

Creating a GUI M-File

Start by creating an M-file for the example GUI.

1. At the MATLAB prompt, type `edit`. MATLAB opens the editor.

2. Type or copy the following statement into the editor. This function statement is the first line in the file.

```
function simple_gui2
```

3. Add these comments to the M-file following the function statement. They are displayed at the command line in response to the `help` command. They must be followed by a blank line.

```
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.
(Leave a blank line here)
```

4. Add an `end` statement at the end of the file. This `end` statement matches the `function` statement. Your file now looks like this.

```
function simple_gui2
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

end
```

Note You need the `end` statement because the example is written using nested functions. For information about using nested functions, see "Nested Functions" in the MATLAB Programming documentation.

---

5. Save the file in your current directory or at a location that is on your MATLAB path.

The next section, "Laying Out a Simple GUI" on page 3-6, shows you how to add components to your GUI.

Laying Out a Simple GUI

In this section...

Creating the Figure

In MATLAB, a GUI is a figure. This first step creates the figure and positions it on the screen. It also makes the GUI invisible so that the GUI user cannot see the components being added or initialized. When the GUI has all its components and is initialized, the example makes it visible.

```
%  Initialize and hide the GUI as it is being constructed.
f = figure('Visible','off','Position',[360,500,450,285]);
```

The call to the `figure` function uses two property/value pairs. The `Position` property is a four-element vector that specifies the location of the GUI on the screen and its size: [distance from left, distance from bottom, width, height]. Default units are pixels.

The next topic, "Adding the Components" on page 3-6, shows you how to add the push buttons, axes, and other components to the GUI.

Adding the Components

The example GUI has six components: three push buttons, one static text, one pop-up menu, and one axes. Start by writing statements that add these components to the GUI. Create the push buttons, static text, and pop-up menu with the `uicontrol` function. Use the `axes` function to create the axes.

1. Add the three push buttons to your GUI by adding these statements to your M-file following the call to `figure`.

```
% Construct the components.
hsurf    = uicontrol('Style','pushbutton',...
             'String','Surf','Position',[315,220,70,25]);
hmesh    = uicontrol('Style','pushbutton',...
             'String','Mesh','Position',[315,180,70,25]);
```

```
hcontour = uicontrol('Style','pushbutton',...
             'String','Countour','Position',[315,135,70,25]);
```

These statements use the `uicontrol` function to create the push buttons. Each statement uses a series of property/value pairs to define a push button.

Property	Description
`Style`	In the example, `pushbutton` specifies the component as a push button.
`String`	Specifies the label that appears on each push button. Here, there are three types of plots: `Surf`, `Mesh`, `Contour`.
`Position`	Uses a four-element vector to specify the location of each push button within the GUI and its size: [distance from left, distance from bottom, width, height]. Default units for push buttons are pixels.

Each call returns the handle of the component that is created.

2 Add the pop-up menu and its label to your GUI by adding these statements to the M-file following the push button definitions.

```
hpopup = uicontrol('Style','popupmenu',...
           'String',{'Peaks','Membrane','Sinc'},...
           'Position',[300,50,100,25]);
htext  = uicontrol('Style','text','String','Select Data',...
           'Position',[325,90,60,15]);
```

For the pop-up menu, the `String` property uses a cell array to specify the three items in the pop-up menu: `Peaks`, `Membrane`, `Sinc`. The static text component serves as a label for the pop-up menu. Its `String` property tells the GUI user to `Select Data`. Default units for these components are pixels.

3 Add the axes to the GUI by adding this statement to the M-file. Set the `Units` property to pixels so that it has the same units as the other components.

```
ha = axes('Units','pixels','Position',[50,60,200,185]);
```

4 Align all components except the axes along their centers with the following statement. Add it to the M-file following all the component definitions.

```
align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');
```

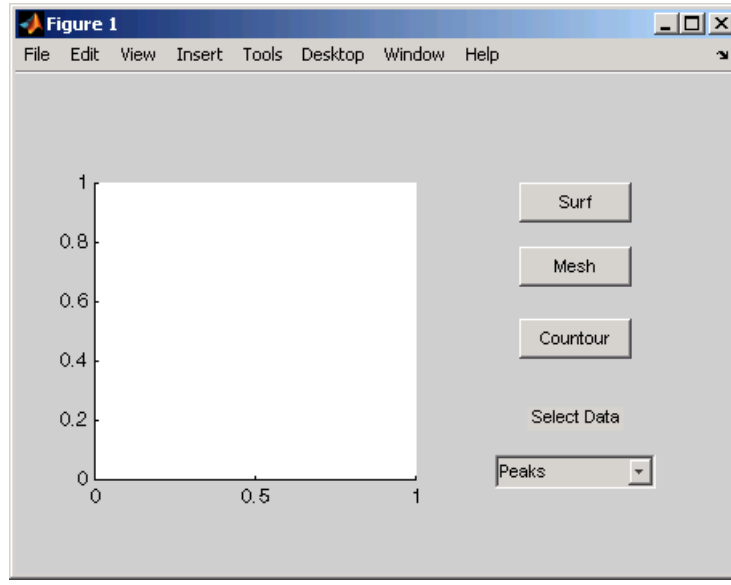
5 Make your GUI visible by adding this command following the `align` command.

```
set(f,'Visible','on')
```

6 This is what your M-file should now look like:

```
function simple_gui2
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

   %  Create and hide the GUI as it is being constructed.
   f = figure('Visible','off','Position',[360,500,450,285]);

   %  Construct the components.
   hsurf = uicontrol('Style','pushbutton','String','Surf',...
           'Position',[315,220,70,25]);
   hmesh = uicontrol('Style','pushbutton','String','Mesh',...
           'Position',[315,180,70,25]);
   hcontour = uicontrol('Style','pushbutton',...
           'String','Countour',...
           'Position',[315,135,70,25]);
   htext = uicontrol('Style','text','String','Select Data',...
           'Position',[325,90,60,15]);
   hpopup = uicontrol('Style','popupmenu',...
           'String',{'Peaks','Membrane','Sinc'},...
           'Position',[300,50,100,25]);
   ha = axes('Units','Pixels','Position',[50,60,200,185]);
   align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');

   %Make the GUI visible.
   set(f,'Visible','on')

end
```

7 Run your script by typing `simple_gui2` at the command line. This is what your GUI now looks like. Note that you can select a data set in the pop-up menu and click the push buttons. But nothing happens. This is because there is no code in the M-file to service the pop-up menu and the buttons.



8 Type `help simple_gui2` at the command line. MATLAB displays this help text.

```
help simple_gui2
  SIMPLE_GUI2 Select a data set from the pop-up menu, then
  click one of the plot-type push buttons. Clicking the button
  plots the selected data in the axes.
```

The next topic, “Initializing the GUI” on page 3-10, shows you how to initialize the GUI.

Initializing the GUI

When you make the GUI visible, it should be initialized so that it is ready for the user. This topic shows you how to

- Make the GUI behave properly when it is resized by changing the component and figure units to `normalized`. This causes the components to resize when the GUI is resized. Normalized units map the lower-left corner of the figure window to `(0,0)` and the upper-right corner to `(1.0, 1.0)`.
- Generate the data to plot. The example needs three sets of data: `peaks_data`, `membrane_data`, and `sinc_data`. Each set corresponds to one of the items in the pop-up menu.
- Create an initial plot in the axes
- Assign the GUI a name that appears in the window title
- Move the GUI to the center of the screen
- Make the GUI visible

1 Replace this code in your M-file:

```
% Make the GUI visible.
set(f,'Visible','on')
```

with this code:

```
% Initialize the GUI.
% Change units to normalized so components resize automatically.
set([f,hsurf,hmesh,hcontour,htext,hpopup],'Units','normalized');
% Generate the data to plot.
peaks_data = peaks(35);
membrane_data = membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc_data = sin(r)./r;
% Create a plot in the axes.
current_data = peaks_data;
surf(current_data);
% Assign the GUI a name to appear in the window title.
set(f,'Name','Simple GUI')
```

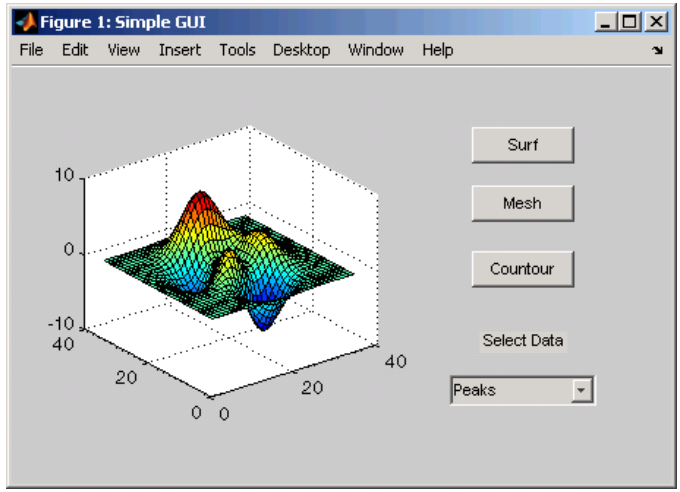
```
% Move the GUI to the center of the screen.
movegui(f,'center')
% Make the GUI visible.
set(f,'Visible','on');
```

2 Verify that your M-file now looks like this:

```
function simple_gui2
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

   % Create and hide the GUI figure as it is being constructed.
   f = figure('Visible','off','Position',[360,500,450,285]);

   % Construct the components
   hsurf = uicontrol('Style','pushbutton','String','Surf',...
           'Position',[315,220,70,25]);
   hmesh = uicontrol('Style','pushbutton','String','Mesh',...
           'Position',[315,180,70,25]);
   hcontour = uicontrol('Style','pushbutton',...
           'String','Countour',...
           'Position',[315,135,70,25]);
   htext = uicontrol('Style','text','String','Select Data',...
           'Position',[325,90,60,15]);
   hpopup = uicontrol('Style','popupmenu',...
           'String',{'Peaks','Membrane','Sinc'},...
           'Position',[300,50,100,25]);
   ha = axes('Units','Pixels','Position',[50,60,200,185]);
   align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');

   % Create the data to plot
   peaks_data = peaks(35);
   membrane_data = membrane;
   [x,y] = meshgrid(-8:.5:8);
   r = sqrt(x.^2+y.^2) + eps;
   sinc_data = sin(r)./r;

   % Initialize the GUI.
   % Change units to normalized so components resize
```

```
    % automatically.
    set([f,hsurf,hmesh,hcontour,htext,hpopup],...
        'Units','normalized');
    %Create a plot in the axes.
    current_data = peaks_data;
    surf(current_data);
    % Assign the GUI a name to appear in the window title.
    set(f,'Name','Simple GUI')
    % Move the GUI to the center of the screen.
    movegui(f,'center')
    % Make the GUI visible.
    set(f,'Visible','on');

end
```

3 Run your script by typing `simple_gui2` at the command line. This is what your GUI should now look like:



The next topic, “Programming the GUI” on page 3-13, shows you how to program the push buttons and pop-up menu so you can interactively generate different plots in the axes.

Programming the GUI

In this section...

Programming the Pop-Up Menu

The pop-up menu enables users to select the data to plot. When a GUI user selects one of the three data sets, MATLAB sets the pop-up menu `Value` property to the index of the selected string. The pop-up menu callback reads the pop-up menu `Value` property to determine which item is currently displayed and sets `current_data` accordingly.

Add the following callback to your file following the initialization code and before the final `end` statement.

```
%  Pop-up menu callback. Read the pop-up menu Value property to
%  determine which item is currently displayed and make it the
%  current data. This callback automatically has access to
%  current_data because this function is nested at a lower level.
   function popup_menu_Callback(source,eventdata)
      % Determine the selected data set.
      str = get(source, 'String');
      val = get(source,'Value');
      % Set current data to the selected data set.
      switch str{val};
      case 'Peaks' % User selects Peaks.
         current_data = peaks_data;
      case 'Membrane' % User selects Membrane.
         current_data = membrane_data;
      case 'Sinc' % User selects Sinc.
         current_data = sinc_data;
      end
   end
```

The next topic, "Programming the Push Buttons" on page 3-14, shows you how to write callbacks for the three push buttons.

Programming the Push Buttons

Each of the three push buttons creates a different type of plot using the data specified by the current selection in the pop-up menu. The push button callbacks plot the data in `current_data`. They automatically have access to `current_data` because they are nested at a lower level.

Add the following callbacks to your file following the pop-up menu callback and before the final `end` statement.

```
% Push button callbacks. Each callback plots current_data in the
% specified plot type.

function surfbutton_Callback(source,eventdata)
% Display surf plot of the currently selected data.
     surf(current_data);
end

function meshbutton_Callback(source,eventdata)
% Display mesh plot of the currently selected data.
     mesh(current_data);
end

function contourbutton_Callback(source,eventdata)
% Display contour plot of the currently selected data.
     contour(current_data);
end
```

The next topic shows you how to associate each callback with its specific component.

Associating Callbacks with Their Components

When the GUI user selects a data set from the pop-up menu or clicks one of the push buttons, MATLAB executes the callback associated with that particular event. But how does MATLAB know which callback to execute?

You must use each component's `Callback` property to specify the name of the callback with which it is associated.

1. To the `uicontrol` statement that defines the **Surf** push button, add the property/value pair

   ```
   'Callback',{@surfbutton_Callback}
   ```

   so that the statement looks like this:

   ```
   hsurf = uicontrol('Style','pushbutton','String','Surf',...
           'Position',[315,220,70,25],...
           'Callback',{@surfbutton_Callback});
   ```

   `Callback` is the name of the property. `surfbutton_Callback` is the name of the callback that services the **Surf** push button.

2. Similarly, to the `uicontrol` statement that defines the **Mesh** push button, add the property/value pair

   ```
   'Callback',{@meshbutton_Callback}
   ```

3. To the `uicontrol` statement that defines the **Contour** push button, add the property/value pair

   ```
   'Callback',{@contourbutton_Callback}
   ```

4. To the `uicontrol` statement that defines the pop-up menu, add the property/value pair

   ```
   'Callback',{@popup_menu_Callback}
   ```

The next topic, "Running the Final GUI" on page 3-16, shows the final M-file and runs the GUI.

Running the Final GUI

In this section...

Final M-File

This is what your final M-file should now look like:

```
function simple_gui2
% SIMPLE_GUI2 Select a data set from the pop-up menu, then
% click one of the plot-type push buttons. Clicking the button
% plots the selected data in the axes.

   %  Create and then hide the GUI as it is being constructed.
   f = figure('Visible','off','Position',[360,500,450,285]);

   %  Construct the components.
   hsurf = uicontrol('Style','pushbutton','String','Surf',...
          'Position',[315,220,70,25],...
          'Callback',{@surfbutton_Callback});
   hmesh = uicontrol('Style','pushbutton','String','Mesh',...
          'Position',[315,180,70,25],...
          'Callback',{@meshbutton_Callback});
   hcontour = uicontrol('Style','pushbutton',...
          'String','Countour',...
          'Position',[315,135,70,25],...
          'Callback',{@contourbutton_Callback});
   htext = uicontrol('Style','text','String','Select Data',...
          'Position',[325,90,60,15]);
   hpopup = uicontrol('Style','popupmenu',...
          'String',{'Peaks','Membrane','Sinc'},...
          'Position',[300,50,100,25],...
          'Callback',{@popup_menu_Callback});
   ha = axes('Units','Pixels','Position',[50,60,200,185]);
   align([hsurf,hmesh,hcontour,htext,hpopup],'Center','None');
```

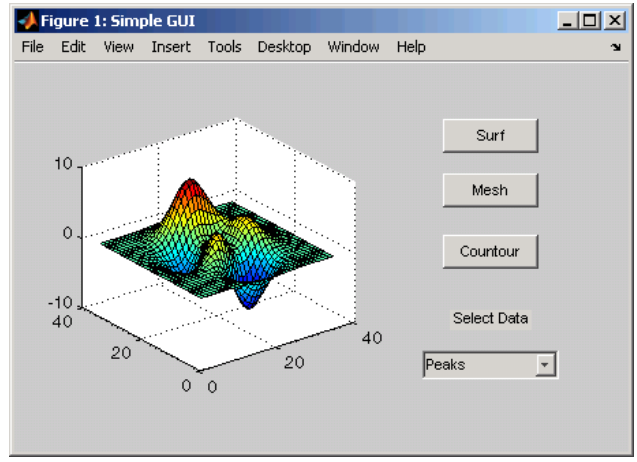
```
% Create the data to plot.
peaks_data = peaks(35);
membrane_data = membrane;
[x,y] = meshgrid(-8:.5:8);
r = sqrt(x.^2+y.^2) + eps;
sinc_data = sin(r)./r;

% Initialize the GUI.
% Change units to normalized so components resize
% automatically.
set([f,ha,hsurf,hmesh,hcontour,htext,hpopup],...
'Units','normalized');
%Create a plot in the axes.
current_data = peaks_data;
surf(current_data);
% Assign the GUI a name to appear in the window title.
set(f,'Name','Simple GUI')
% Move the GUI to the center of the screen.
movegui(f,'center')
% Make the GUI visible.
set(f,'Visible','on');

%  Callbacks for simple_gui2. These callbacks automatically
%  have access to component handles and initialized data
%  because they are nested at a lower level.

%  Pop-up menu callback. Read the pop-up menu Value property
%  to determine which item is currently displayed and make it
%  the current data.
   function popup_menu_Callback(source,eventdata)
      % Determine the selected data set.
      str = get(source, 'String');
      val = get(source,'Value');
      % Set current data to the selected data set.
      switch str{val};
      case 'Peaks' % User selects Peaks.
         current_data = peaks_data;
      case 'Membrane' % User selects Membrane.
         current_data = membrane_data;
      case 'Sinc' % User selects Sinc.
```

```
            current_data = sinc_data;
         end
      end

   % Push button callbacks. Each callback plots current_data in
   % the specified plot type.

   function surfbutton_Callback(source,eventdata)
   % Display surf plot of the currently selected data.
      surf(current_data);
   end

   function meshbutton_Callback(source,eventdata)
   % Display mesh plot of the currently selected data.
      mesh(current_data);
   end

   function contourbutton_Callback(source,eventdata)
   % Display contour plot of the currently selected data.
      contour(current_data);
   end

end
```

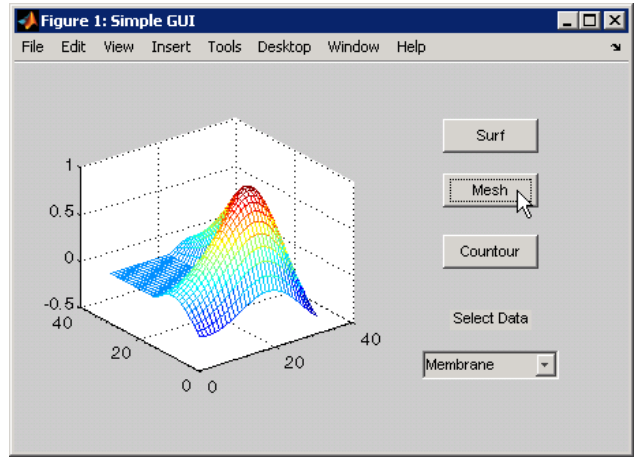
Running the GUI

1 Run the simple GUI by typing the name of the M-file at the command line.

```
simple_gui2
```



2 In the pop-up menu, select **Membrane**, and then click the **Mesh** button. The GUI displays a mesh plot of the MATLAB logo.



3 Try other combinations before closing the GUI.

Creating GUIs with GUIDE

Chapter 4, What Is GUIDE? (p. 4-1)	Introduces GUIDE
Chapter 5, GUIDE Preferences and Options (p. 5-1)	Describes briefly the available MATLAB preferences and GUI options.
Chapter 6, Laying Out a GUIDE GUI (p. 6-1)	Shows you how to start GUIDE and from there how to populate the GUI and create menus. Provides guidance in designing a GUI for cross-platform compatibility.
Chapter 7, Saving and Running a GUIDE GUI (p. 7-1)	Describes the files used to store the GUI. Steps you through the process for saving a GUI, and lists the different ways in which you can activate a GUI.
Chapter 8, Programming a GUIDE GUI (p. 8-1)	Explains how user-written callback routines control GUI behavior. Shows you how to associate callbacks with specific components and explains callback syntax and arguments. Provides simple programming examples for each kind of component.
Chapter 9, Managing and Sharing Application Data in GUIDE (p. 9-1)	Explains the mechanisms for managing application-defined data and explains how to share data among a GUIs callbacks.
Chapter 10, Examples of GUIDE GUIs (p. 10-1)	Illustrates techniques for programming various behaviors.

4

What Is GUIDE?

GUIDE: An Overview

In this section...

GUI Layout

GUIDE, the MATLAB graphical user interface development environment, provides a set of tools for creating graphical user interfaces (GUIs). These tools simplify the process of laying out and programming GUIs.

Using the GUIDE Layout Editor, you can populate a GUI by clicking and dragging GUI components—such as axes, panels, buttons, text fields, sliders, and so on—into the layout area. You can also create menus and context menus for the GUI. From the Layout Editor, you can size the GUI, modify component look and feel, align components, set tab order, view a hierarchical list of the component objects, and set GUI options.

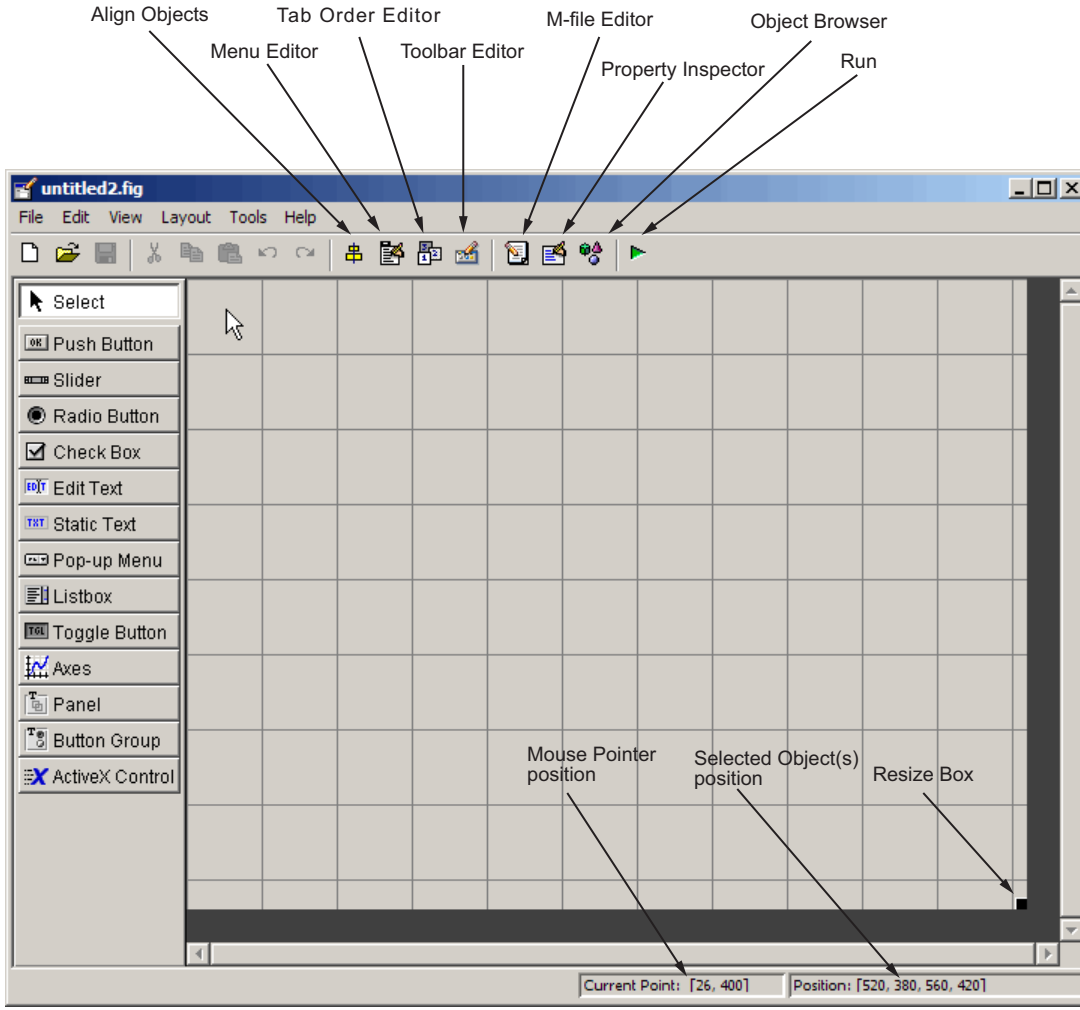
GUI Programming

GUIDE automatically generates an M-file that controls how the GUI operates. This M-file provides code to initialize the GUI and contains a framework for the GUI callbacks—the routines that execute when a user interacts with a GUI component. Using the M-file editor, you can add code to the callbacks to perform the functions you want.

Note MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see "Predefined Dialog Boxes" in the MATLAB Function Reference documentation.

GUIDE Tools Summary

The GUIDE tools are available from the Layout Editor shown in the figure below. The tools are called out in the figure and described briefly below. Subsequent sections show you how to use them.



Use This Tool...	To...
Layout Editor	Select components from the component palette, at the left side of the Layout Editor, and arrange them in the layout area. See “Adding Components to the GUI” on page 6-18 for more information.
Figure Resize Tab	Set the size at which the GUI is initially displayed when you run it. See “Setting the GUI Size” on page 6-16 for more information.
Menu Editor	Create menus and context, i.e., pop-up, menus. See “Creating Menus” on page 6-70 for more information.
Align Objects	Align and distribute groups of components. Grids and rulers also enable you to align components on a grid with an optional snap-to-grid capability. See “Aligning Components” on page 6-62 for more information.
Tab Order Editor	Set the tab and stacking order of the components in your layout. See “Setting Tab Order” on page 6-67 for more information.
Toolbar Editor	Create Toolbars containing predefined and custom push buttons and toggle buttons. See “Creating Toolbars” on page 6-84 for more information.
Icon Editor	Create and modify icons for tools in a toolbar. See “Creating Toolbars” on page 6-84 for more information.
Property Inspector	Set the properties of the components in your layout. It provides a list of all the properties you can set and displays their current values.
Object Browser	Display a hierarchical list of the objects in the GUI. See “Viewing the Object Hierarchy” on page 6-100 for more information.
Run	Save and run the current GUI. See Chapter 7, “Saving and Running a GUIDE GUI” for more information.

Use This Tool...	To...
M-File Editor	Display, in your default editor, the M-file associated with the GUI. See "GUI Files: An Overview" on page 8-5 for more information.
Position Readouts	Continuously display the mouse cursor position and the positions of selected objects

You can also set preferences that apply to all GUIs at creation, and options that are GUI-specific. See Chapter 5, "GUIDE Preferences and Options" for more information.

5

GUIDE Preferences and Options

GUIDE Preferences

In this section...

Setting Prefernces

You can set preferences for GUIDE by selecting **Preferences** from the **File** menu. These preferences apply to GUIDE and to all GUIs you create.

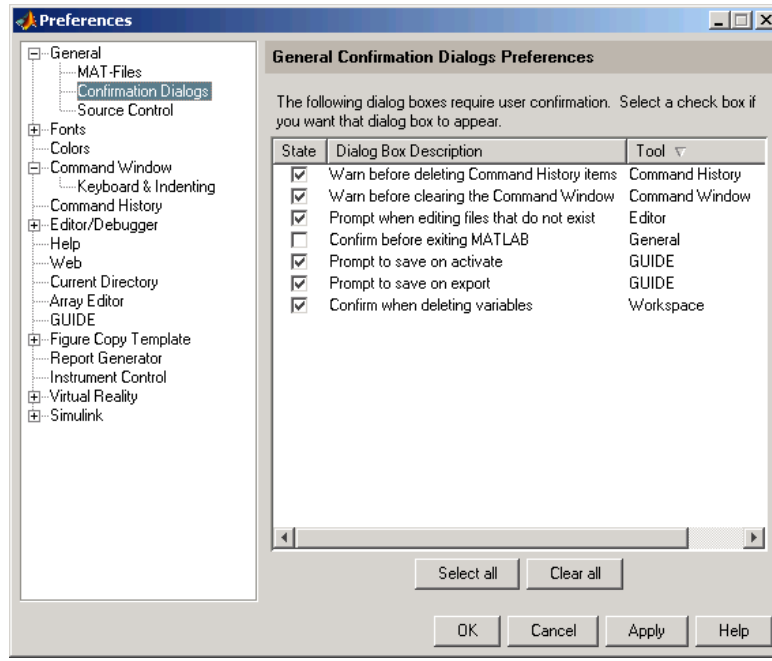
The preferences are in different locations within the Preferences dialog box:

Confirmation Preferences

GUIDE provides two confirmation preferences. You can choose whether you want to display a confirmation dialog box when you

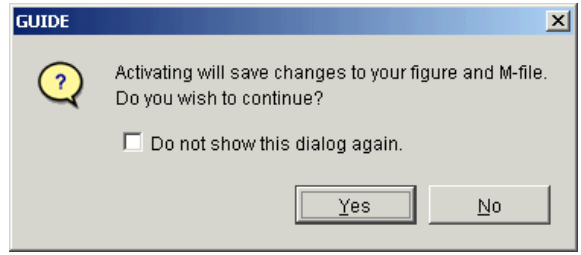
- Activate a GUI from GUIDE
- Export a GUI from GUIDE

In the Preferences dialog box, click **General > Confirmation Dialogs** to access the GUIDE confirmation preferences. Look for the word `GUIDE` in the **Tool** column.



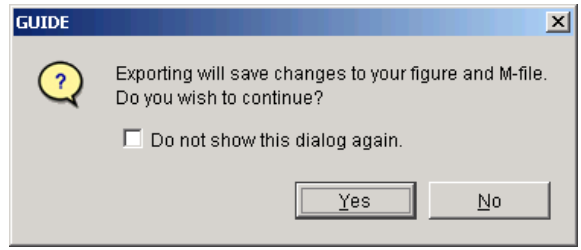
Prompt to Save on Activate

When you activate a GUI by clicking the **Run** button in the Layout Editor, a dialog box informs you of the impending save and lets you choose whether or not you want to continue.



Prompt to Save on Export

When you select **Export** from the Layout Editor **File** menu, a dialog box informs you of the impending save and lets you choose whether or not you want to continue.

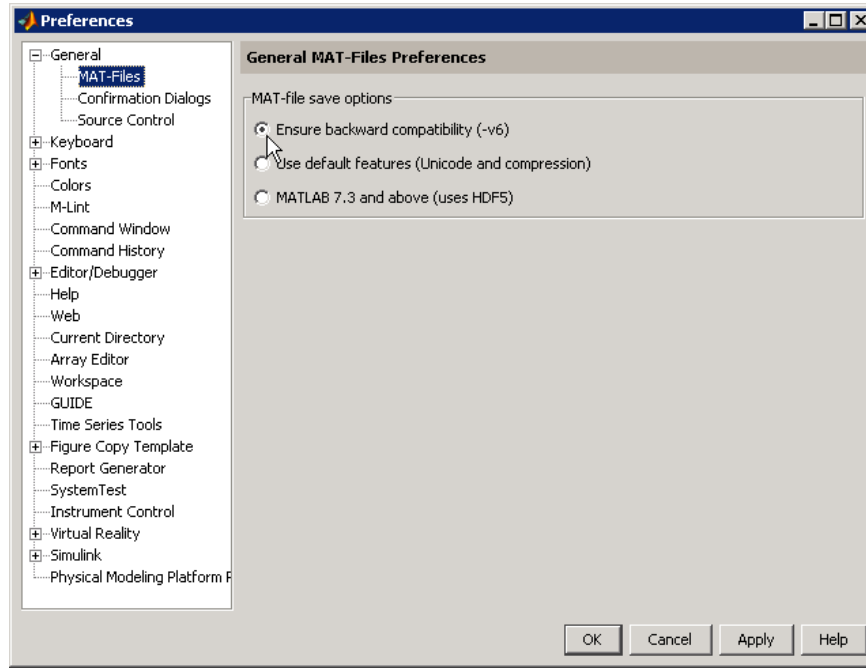


Backward Compatibility Preference

Ensure Backward Compatibility (-v6)

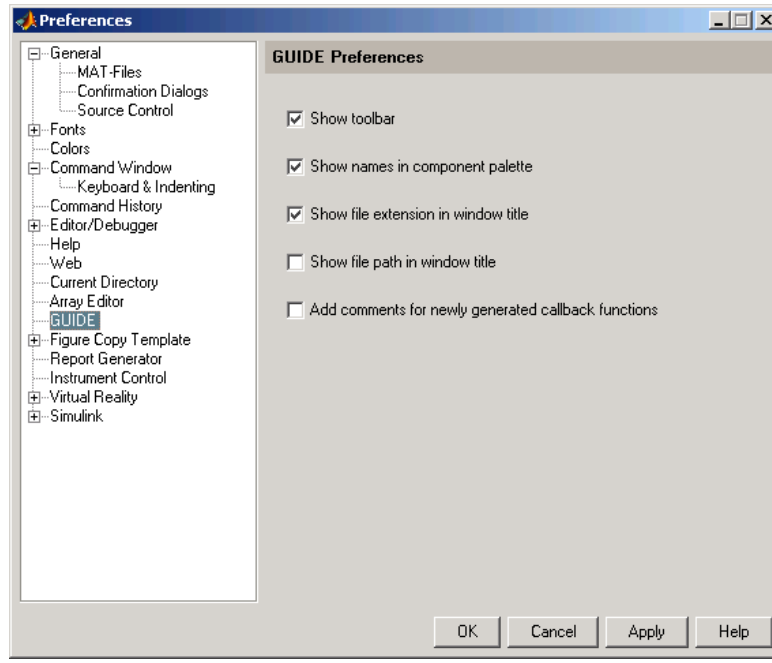
GUI FIG-files created or modified with MATLAB 7.0 or a later MATLAB version are not automatically compatible with Version 6.5 and earlier versions. GUIDE automatically generates FIG-files, which are a kind of MAT-file, to hold layout information for GUIs.

To make a FIG-file backward compatible, you must select **Ensure backward compatibility (-v6)** in the Preferences dialog box under **General > MAT-Files**. This is shown in the figure below.



All Other Preferences

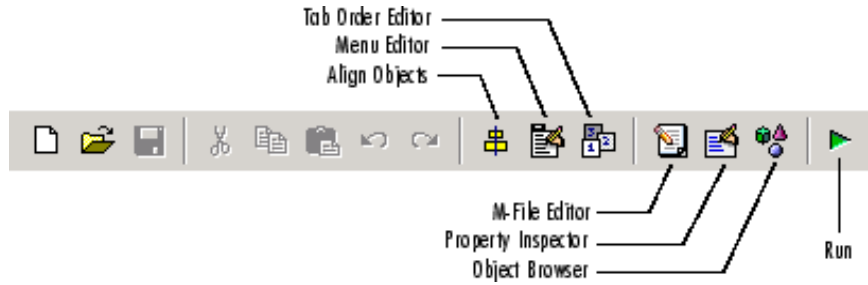
GUIDE provides several other preferences for the Layout Editor interface and M-file comments. In the Preferences dialog box, click **GUIDE** to access these preferences.



The following topics describe the preferences in this dialog:

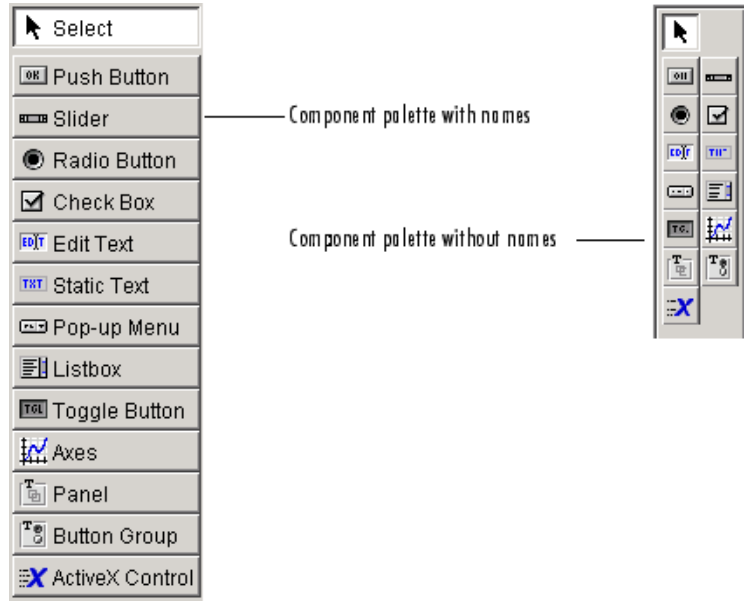
Show Toolbar

Displays the following toolbar in the Layout Editor window.



Show Names in Component Palette

Displays both icons and names in the component palette, as shown below. When unchecked, the icons alone are displayed in two columns.



Show File Extension in Window Title

Displays the GUI FIG-file filename with its file extension, `.fig`, in the Layout Editor window title. If unchecked, only the filename is displayed.

Show File Path in Window Title

Displays the full file path in the Layout Editor window title. If unchecked, the file path is not displayed.

Add Comments for Newly Generated Callback Functions

When this preference is checked, GUIDE includes the comment lines shown in the following example to all callbacks that are added to the M-file.

```
% --- Executes during object deletion, before destroying properties.
function figure1_DeleteFcn(hObject, eventdata, handles)
% hObject    handle to figure1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Some callbacks are added automatically because their associated components are part of the original GUIDE template that you chose. Other commonly used callbacks are added automatically when you add components. You can also add callbacks explicitly by selecting them from **View Callbacks** on the **View** menu or on the component's context menu.

If this preference is unchecked, GUIDE includes comments only for callbacks that are automatically included to support the original GUIDE template. No comments are included for any other callbacks that are added to the M-file.

See "Callback Syntax and Arguments" on page 8-12 for more information about callbacks and about the arguments described in the comments above.

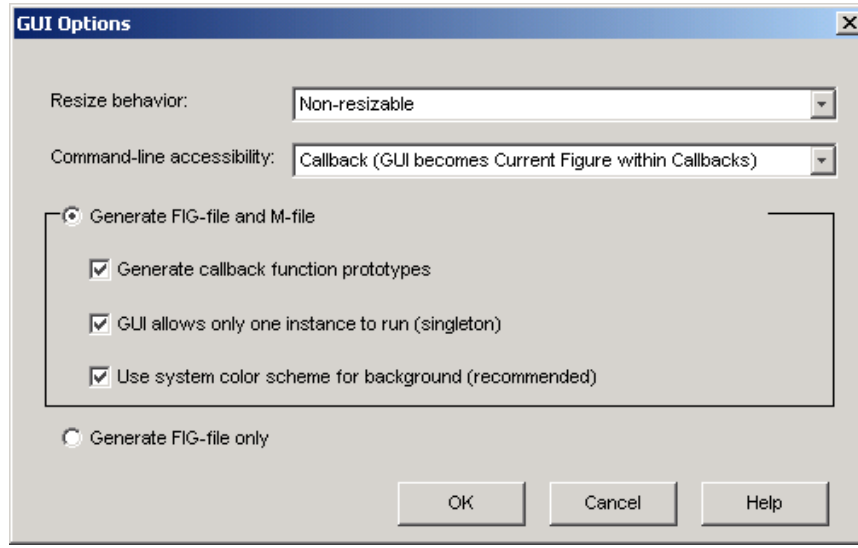
GUI Options

In this section...

The GUI Options Dialog Box

You can use the GUI Options dialog box to configure various behaviors that are specific to the GUI you are creating. These options take effect when you next save the GUI.

Access the dialog box by selecting **GUI Options** from the Layout Editor **Tools** menu.



The following sections describe the options in this dialog box:

Resize Behavior

You can control whether users can resize the figure window containing your GUI and how MATLAB handles resizing. GUIDE provides three options:

- **Non-resizable** — Users cannot change the window size (default).
- **Proportional** — MATLAB automatically rescales the components in the GUI in proportion to the new figure window size.
- **Other (Use ResizeFcn)** — Program the GUI to behave in a certain way when users resize the figure window.

The first two options set figure and component properties appropriately and require no other action. **Other (Use ResizeFcn)** requires you to write a callback routine that recalculates sizes and positions of the components based on the new figure size.

Command-Line Accessibility

You can restrict access to a GUI figure from the command line or from an M-file by using the GUIDE **Command-line accessibility** options.

Unless you explicitly specify a figure handle, many commands, such as `plot`, alter the current figure, i.e., the figure specified by the root `CurrentFigure` property and returned by the `gcf` command. The current figure is usually the figure that is most recently created or clicked in. However, a figure can also become the current figure with the statement

```
figure(h)
```

or by setting the `CurrentFigure` property to the figure's handle.

The `gcf` function returns the handle of the current figure.

```
h = gcf
```

For a GUI created in GUIDE, set the **Command-line accessibility** option to prevent users from inadvertently changing the appearance or content of a GUI by executing commands at the command line or from an M-file, such as `plot`. The following table briefly describes the four options for **Command-line accessibility**.

Option	Description
Callback (GUI becomes Current Figure within Callbacks)	The GUI can be accessed only from within a GUI callback. The GUI cannot be accessed from the command line or from an M-script. This is the default.
Off (GUI never becomes Current Figure)	The GUI can not be accessed from a callback, the command line, or an M-script, without the handle.
On (GUI may become Current Figure from Command Line)	The GUI can be accessed from a callback, from the command line, and from an M-script.
Other (Use settings from Property Inspector)	You control accessibility by setting the `HandleVisibility` and `IntegerHandle` properties from the Property Inspector.

Generate FIG-File and M-File

Select **Generate FIG-file and M-file** in the **GUI Options** dialog box if you want GUIDE to create both the FIG-file and the GUI M-file (this is the default). Once you have selected this option, you can select any of the following items in the frame to configure the M-file:

- "Generate Callback Function Prototypes" on page 5-11
- "GUI Allows Only One Instance to Run (Singleton)" on page 5-12
- "Use System Color Scheme for Background" on page 5-12

See "GUI Files: An Overview" on page 8-5 for information about these files.

Generate Callback Function Prototypes

If you select **Generate callback function prototypes** in the **GUI Options** dialog, GUIDE adds templates for the most commonly used callbacks to the GUI M-file for most components you add to the GUI. You must then write the code for these callbacks.

GUIDE also adds a callback whenever you edit a callback routine from the Layout Editor's right-click context menu and when you add menus to the GUI using the Menu Editor.

See "Callback Syntax and Arguments" on page 8-12 for general information about callbacks.

Note This option is available only if you first select the **Generate FIG-file and M-File** option.

GUI Allows Only One Instance to Run (Singleton)

This option allows you to select between two behaviors for the GUI figure:

- Allow MATLAB to display only one instance of the GUI at a time.
- Allow MATLAB to display multiple instances of the GUI.

If you allow only one instance, MATLAB reuses the existing GUI figure whenever the command to run the GUI is issued. If a GUI already exists, MATLAB brings it to the foreground rather than creating a new figure.

If you clear this option, MATLAB creates a new GUI figure whenever you issue the command to run the GUI.

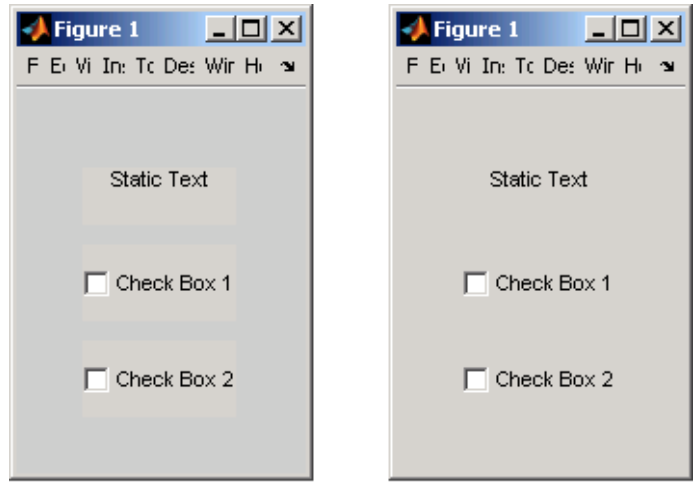
Note This option is available only if you first select the **Generate FIG-file and M-File** option.

Use System Color Scheme for Background

The default color used for GUI components is system dependent. This option enables you to make the figure background color the same as the default component background color.

If you select **Use system color scheme for background** (the default), GUIDE changes the figure background color to match the color of the GUI components.

The following figures illustrate the results with and without system color matching.



Without system color matching

With system color matching

Note This option is available only if you first select the **Generate FIG-file and M-File** option.

Generate FIG-File Only

The **Generate FIG-file only** option enables you to open figures and GUIs to perform limited editing. These can be any figures and need not be GUIs. GUIs need not have been generated using GUIDE. This mode provides limited editing capability and may be useful for GUIs generated in MATLAB Versions 5.3 and earlier. See the `guide` function for more information.

GUIDE selects **Generate FIG-file only** as the default if you do one of the following:

- Start GUIDE from the command line and provide one or more figure handles as arguments.

  ```
  guide(fh)
  ```

In this case, GUIDE selects **Generate FIG-file only** even though there may be a corresponding M-file in the same location.

- Start GUIDE from the command line and provide the name of a FIG-file for which no M-file with the same name exists in the same location.

```
guide('myfig.fig')
```

- Use the GUIDE **Open Existing GUI** tab to open a FIG-file for which no M-file with the same name exists in the same location.

When you save the figure or GUI with **Generate FIG-file only** selected, GUIDE saves only the FIG-file. You must update any corresponding M-files as appropriate.

If you want GUIDE to manage the GUI M-file for you, change the selection to **Generate FIG-file and M-file** before saving the GUI. If there is no corresponding M-file in the same location, GUIDE creates one. If an M-file with the same name as the original figure or GUI exists in the same location, GUIDE overwrites it. To prevent this, save the GUI using **Save As** from the **File** menu and select another filename. You must update the new M-file as appropriate.

6

Laying Out a GUIDE GUI

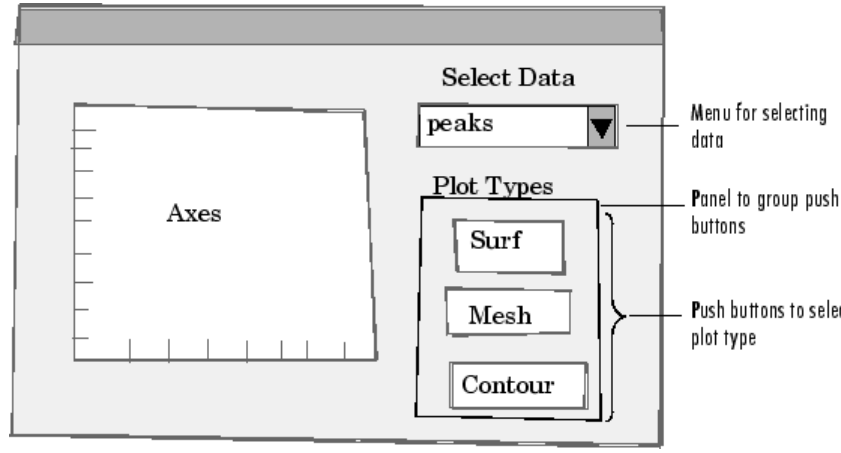
Viewing the Object Hierarchy (p. 6-100)	Describes use of the Object Browser to view the hierarchy of objects, including menus, in your GUI.
Designing for Cross-Platform Compatibility (p. 6-101)	Provides pointers for creating GUIs that behave more consistently when run on different platforms.

Designing a GUI

Before creating the actual GUI, it is important to decide what it is you want your GUI to do and how you want it to work. It is helpful to draw your GUI on paper and envision what the user sees and what actions the user takes.

Note MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see "Predefined Dialog Boxes" in the MATLAB Function Reference documentation.

The GUI used in this example contains an axes component that displays either a surface, mesh, or contour plot of data selected from the pop-up menu. The following picture shows a sketch that you might use as a starting point for the design.



A panel contains three push buttons that enable you to choose the type of plot you want. The pop-up menu contains three strings — `peaks`, `membrane`, and `sinc`, which correspond to MATLAB functions. You can select the data to plot from this menu.

Many Web sites and commercial publications such as the following provide guidelines for designing GUIs:

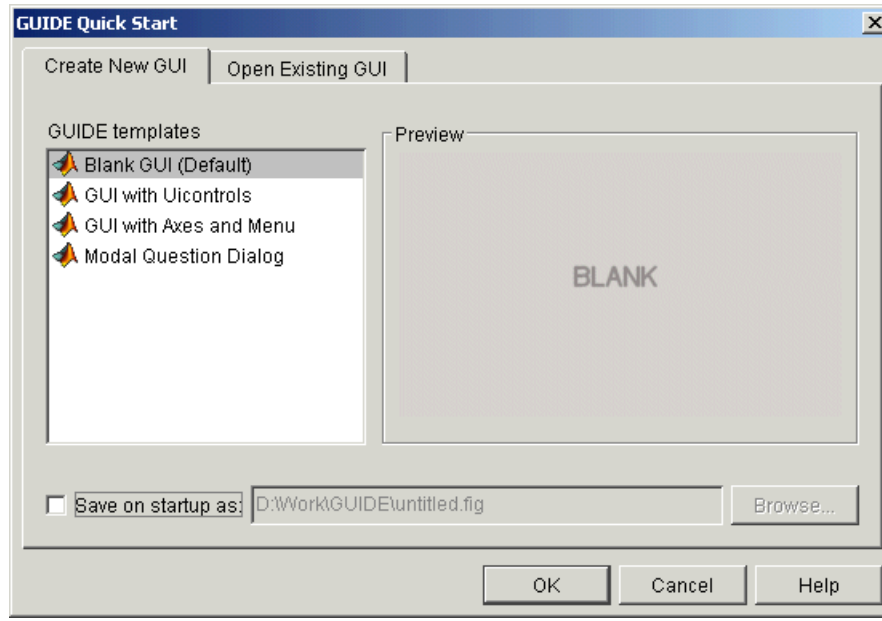
- AskTog — Essays on good design and a list of First Principles for good user interface design. The author, Tognazzini, is a well-respected user interface designer. `http://www.asktog.com/basics/firstPrinciples.html`.
- Galitz, Wilbert, O., *Essential Guide to User Interface Design*. Wiley, New York, NY, 2002.
- GUI Design Handbook — A detailed guide to the use of GUI controls. `http://www.fast-consulting.com/GUI%20Design%20Handbook/GDH_FRNTMTR.htm`.
- Johnson, J., *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann, San Francisco, CA, 2000.
- Usability Glossary — An extensive glossary of terms related to GUI design, usability, and related topics. `http://www.usabilityfirst.com/glossary/main.cgi`.
- UsabilityNet — Covers design principles, user-centered design, and other usability and design-related topics. `http://www.usabilitynet.org/management/b_design.htm`.

Starting GUIDE

There are many ways to start GUIDE. You can start GUIDE from the:

- Command line by typing `guide`
- **Start** menu by selecting **MATLAB > GUIDE (GUI Builder)**
- **MATLAB File** menu by selecting **New > GUI**
- MATLAB toolbar by clicking the **GUIDE** button 

However you start GUIDE, it displays the GUIDE Quick Start dialog box shown in the following figure.



The GUIDE Quick Start dialog box contains two tabs:

- **Create New GUI** — Asks you to start creating your new GUI by choosing a template for it. You can also specify the name by which the GUI is saved.

See “Selecting a GUI Template” on page 6-7 for information about the templates.

- **Open Existing GUI** — Enables you to open an existing GUI in GUIDE. You can choose a GUI from your current directory or browse other directories.

Selecting a GUI Template

In this section...

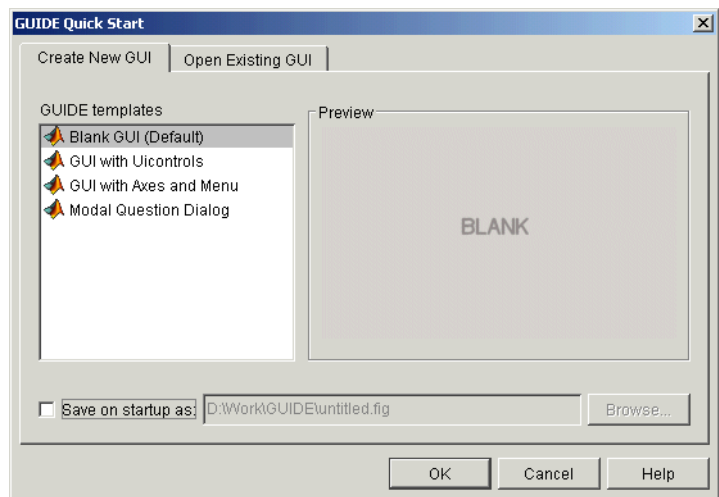
Accessing the Templates

GUIDE provides several templates that you can modify to create your own GUIs. The templates are fully functional GUIs; they are already programmed.

You can access the templates in two ways:

- Start GUIDE. See “Starting GUIDE” on page 6-5 for information.
- If GUIDE is already open, select **New** from the **File** menu in the Layout Editor.

In either case, GUIDE displays the **GUIDE Quick Start** dialog box with the **Create New GUI** tab selected as shown in the following figure. This tab contains a list of the available templates.



To use a template:

1. Select a template in the left pane. A preview displays in the right pane.
2. Optionally, name your GUI now by selecting **Save on startup as** and typing the name in the field to the right. GUIDE saves the GUI before opening it in the Layout Editor. If you choose not to name the GUI at this point, GUIDE prompts you to save it and give it a name the first time you run the GUI.
3. Click **OK** to open the GUI template in the Layout Editor.

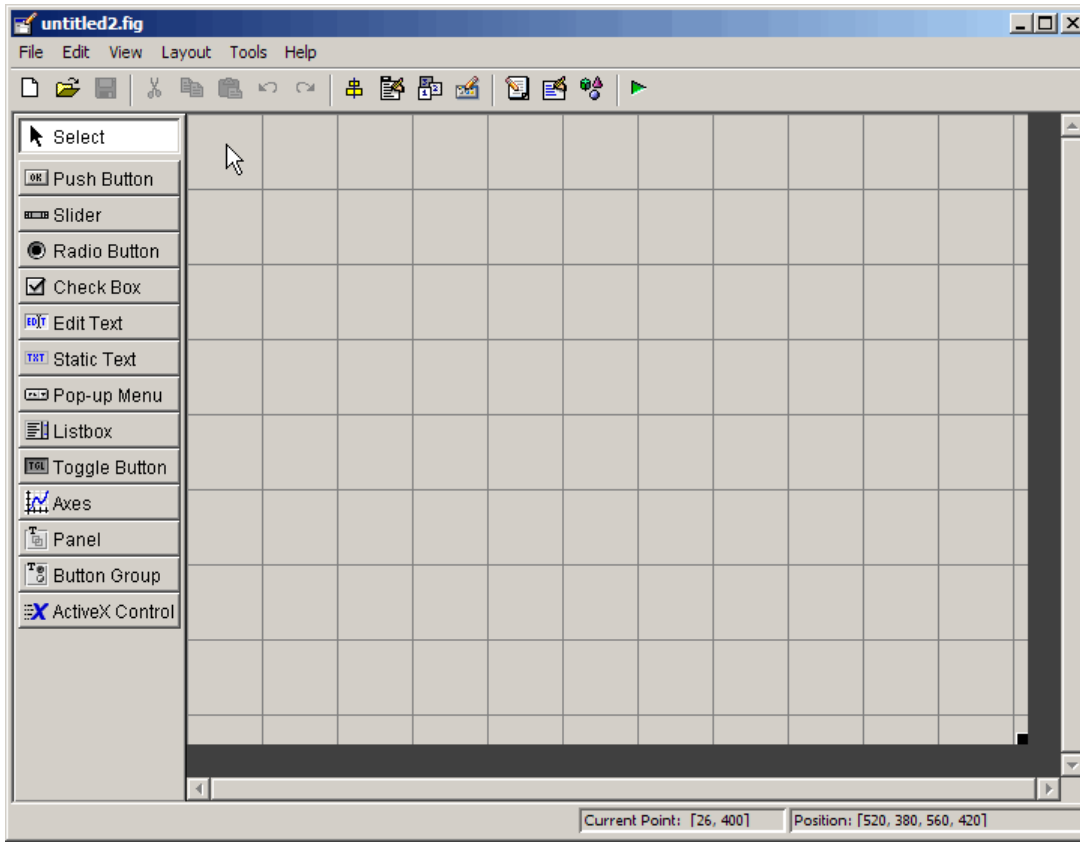
Template Descriptions

GUIDE provides four fully functional templates. They are described in the following sections:

Note To see how the template GUIs work, you can view their M-file code and look at their callbacks. You can also modify the callbacks for your own purposes. To view the M-file for any of these templates, open the template in the Layout Editor and click the **M-file Editor** button on the toolbar. For information about using callbacks, see Chapter 8, "Programming a GUIDE GUI".

Blank GUI

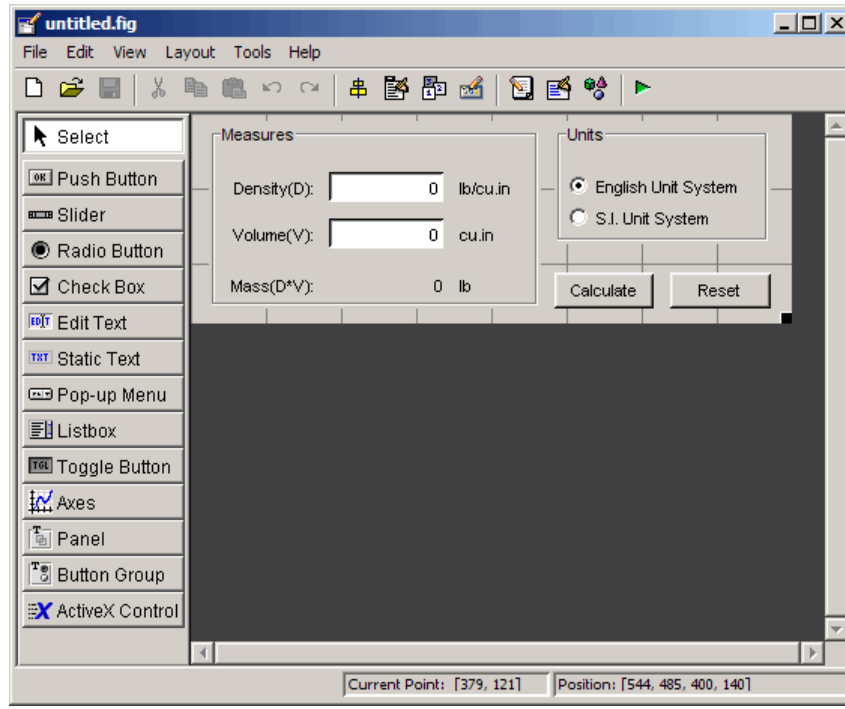
The blank GUI template displayed in the Layout Editor is shown in the following figure.



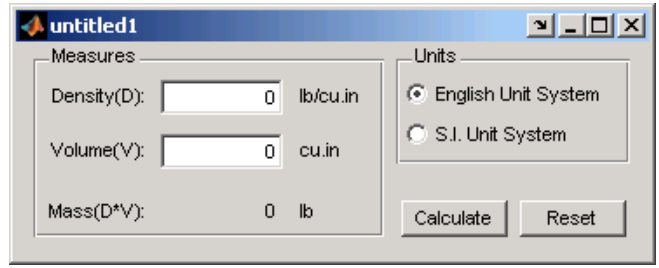
Select the blank GUI if the other templates are not suitable starting points for the GUI you are creating, or if you prefer to start with an empty GUI.

GUI with Uicontrols

The following figure shows the template for a GUI with user interface controls (uicontrols) displayed in the Layout Editor. User interface controls include push buttons, sliders, radio buttons, check boxes, editable and static text components, list boxes, and toggle buttons.



When you run the GUI by clicking the **Run** button, the GUI appears as shown in the following figure.

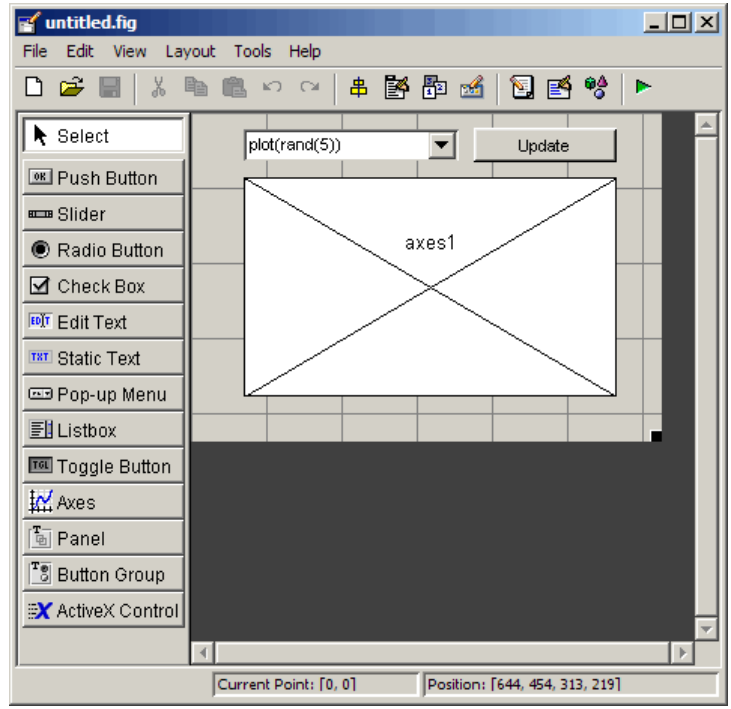


When a user enters values for the density and volume of an object, and clicks the **Calculate** button, the GUI calculates the mass of the object and displays the result next to **Mass(D*V)**.

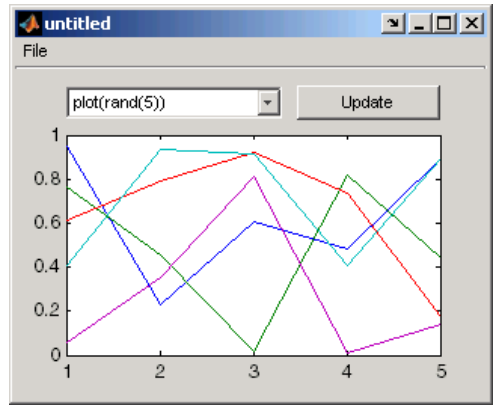
To view the M-file code for these user interface controls, open the template in the Layout Editor and click the **M-file Editor** button on the toolbar.

GUI with Axes and Menu

The template for a GUI with axes and menu is shown in the following figure.


untitled.fig
File Edit View Layout Tools Help
Select
Push Button
Slider
Radio Button
Check Box
Edit Text
Static Text
Pop-up Menu
Listbox
Toggle Button
Axes
Panel
Button Group
ActiveX Control
plot(rand(5))
Update
axes1
Current Point: [0, 0]
Position: [644, 454, 313, 219]

When you run the GUI by clicking the **Run** button on the toolbar, the GUI displays a plot of five lines, each of which is generated from random numbers using the MATLAB `rand(5)` command. The following figure shows an example.



You can select other plots in the pop-up menu. Clicking the **Update** button displays the currently selected plot on the axes.

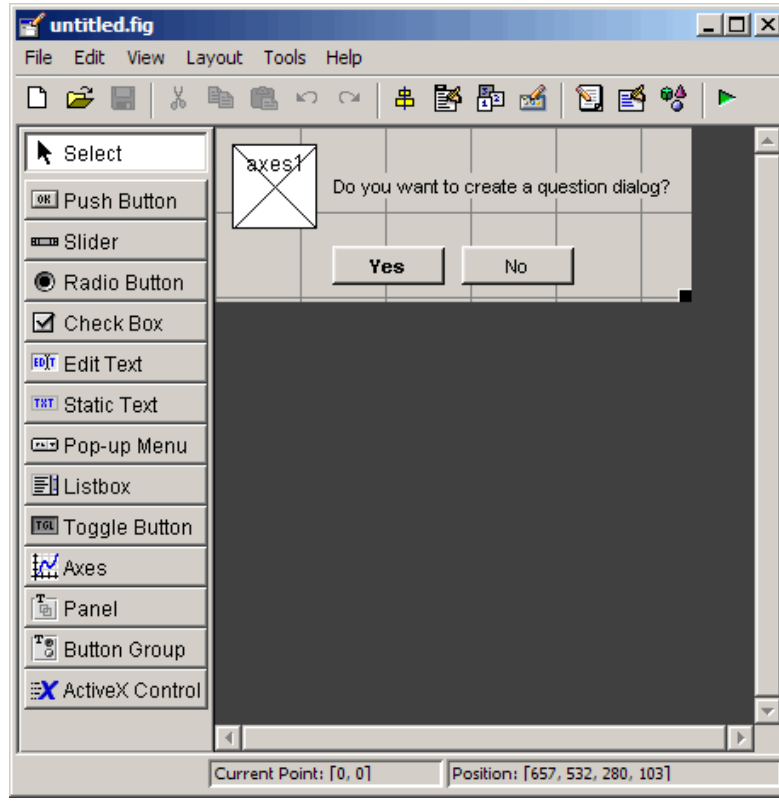
The GUI also has a **File** menu with three items:

- **Open** displays a dialog box from which you can open files on your computer.
- **Print** opens the Print dialog box. Clicking **OK** in the Print dialog box prints the figure.
- **Close** closes the GUI.

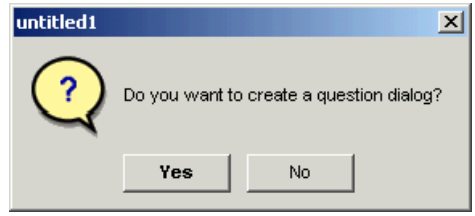
To view the M-file code for these menu choices, open the template in the Layout Editor and click the **M-file Editor** button on the toolbar.

Modal Question Dialog

The modal question dialog template displayed in the Layout Editor is shown in the following figure.



Running the GUI displays the dialog box shown in the following figure:



The GUI returns the text string `Yes` or `No`, depending on which button you click.

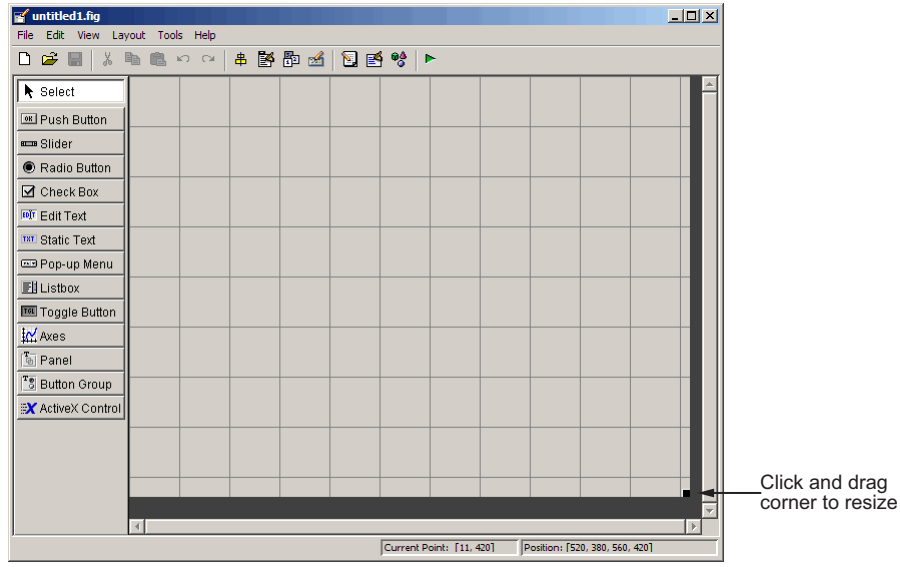
The GUI is *blocking*, which means that the current M-file stops executing until the GUI restores execution. The GUI is also *modal*, which means that the user cannot interact with other MATLAB windows until one of the buttons is clicked.

Select this template if you want your GUI to return a string or to be modal.

To view the M-file code for these capabilities, open the template in the Layout Editor and click the **M-file Editor** button  on the toolbar. See "Using a Modal Dialog to Confirm an Operation" on page 10-52 for an example of using this template with another GUI. Also see the figure `WindowStyle` property for more information.

Setting the GUI Size

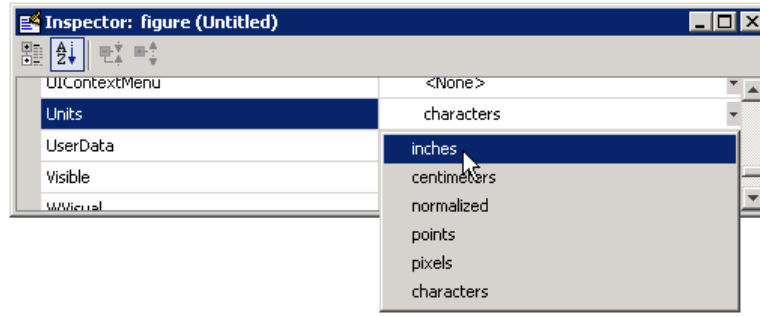
Set the size of the GUI by resizing the grid area in the Layout Editor. Click the lower-right corner and drag it until the GUI is the desired size. If necessary, make the window larger.



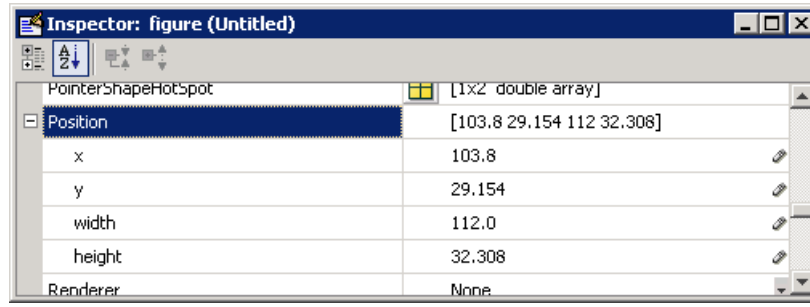
If you want to set the position or size of the GUI to an exact value, do the following:

1 Select **Property Inspector** from the **View** menu or click the **Property Inspector** button .

2 Scroll to the `Units` property and note whether the current setting is `characters` or `normalized`. Click the button next to `Units` and then change the setting to `inches` from the pop-up menu.



3 In the Property Inspector, click the **+** sign next to `Position`. The elements of the component's `Position` property are displayed.



4 Type the `x` and `y` coordinates of the point where you want the lower-left corner of the GUI to appear, and its width and height.

5 Reset the `Units` property to its previous setting, either `characters` or `normalized`.

Note Setting the `Units` property to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) gives the GUI a more consistent appearance across platforms. See "Cross-Platform Compatible Units" on page 6-103 for more information.

Adding Components to the GUI

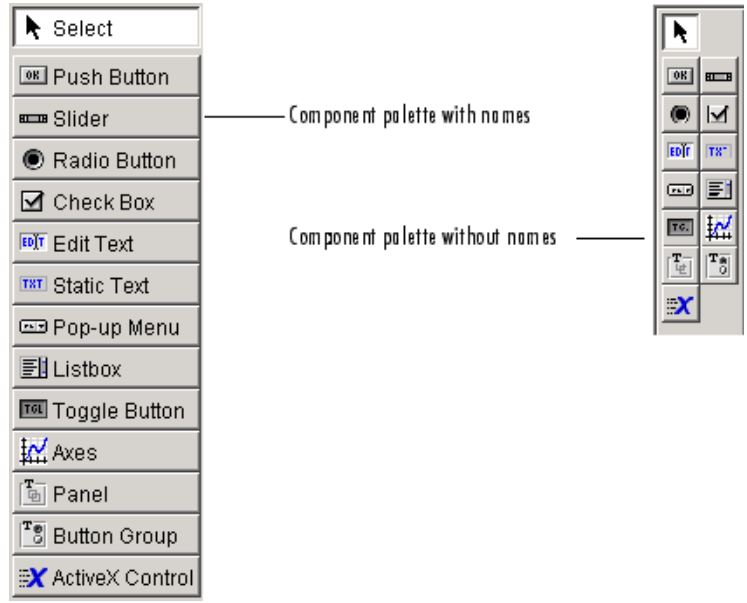
In this section...

Other topics that may be of interest:

- "Aligning Components" on page 6-62
- "Setting Tab Order" on page 6-67

Available Components

The component palette at the left side of the Layout Editor contains the components that you can add to your GUI. You can display it with or without names.



When you first open the Layout Editor, the component palette contains only icons. To display the names of the GUI components, select **Preferences** from the **File** menu, check the box next to **Show names in component palette**, and click **OK**.

See “Creating Menus” on page 6-70 for information about adding menus to a GUI.

Note This section provides information about using components to lay out a GUI. For information about programming these components see Chapter 8, “Programming a GUIDE GUI”.

Component	Icon	Description
Push Button		Push buttons generate an action when clicked. For example, an **OK** button might apply settings and close a dialog box. When you click a push button, it appears depressed; when you release the mouse button, the push button appears raised.
Toggle Button		Toggle buttons generate an action and indicate whether they are turned on or off. When you click a toggle button, it appears depressed, showing that it is on. When you release the mouse button, the toggle button remains depressed until you click it a second time. When you do so, the button returns to the raised state, showing that it is off. Use a button group to manage mutually exclusive toggle buttons.
Radio Button		Radio buttons are similar to check boxes, but radio buttons are typically mutually exclusive within a group of related radio buttons. That is, when you select one button the previously selected button is deselected. To activate a radio button, click the mouse button on the object. The display indicates the state of the button. Use a button group to manage mutually exclusive radio buttons.
Check Box		Check boxes can generate an action when checked and indicate their state as checked or not checked. Check boxes are useful when providing the user with a number of independent choices, for example, displaying a toolbar.
Edit Text		Edit text components are fields that enable users to enter or modify text strings. Use edit text when you want text as input. Users can enter numbers but you must convert them to their numeric equivalents.

Component	Icon	Description
Static Text		Static text controls display lines of text. Static text is typically used to label other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively.
Slider		Sliders accept numeric input within a specified range by enabling the user to move a sliding bar, which is called a slider or thumb. Users move the slider by clicking the slider and dragging it, by clicking in the trough, or by clicking an arrow. The location of the slider indicates the relative location within the specified range.
List Box		List boxes display a list of items and enable users to select one or more items.
Pop-Up Menu		Pop-up menus open to display a list of choices when users click the arrow.
Axes		Axes enable your GUI to display graphics such as graphs and images. Like all graphics objects, axes have properties that you can set to control many aspects of its behavior and appearance. See “Axes Properties” in the MATLAB Graphics documentation and commands such as the following for more information on axes objects: `plot`, `surf`, `line`, `bar`, `polar`, `pie`, `contour`, and `mesh`. See Functions — By Category in the MATLAB Function Reference documentation for a complete list.

Component	Icon	Description
Panel		Panels arrange GUI components into groups. By visually grouping related controls, panels can make the user interface easier to understand. A panel can have a title and various borders. Panel children can be user interface controls and axes as well as button groups and other panels. The position of each component within a panel is interpreted relative to the panel. If you move the panel, its children move with it and maintain their positions on the panel.
Button Group		Button groups are like panels but are used to manage exclusive selection behavior for radio buttons and toggle buttons.
ActiveX Component		ActiveX components enable you to display ActiveX controls in your GUI. They are available only on the Microsoft Windows platform. An ActiveX control can be the child only of a figure, i.e., of the GUI itself. It cannot be the child of a panel or button group. See “ActiveX Control” on page 8-33 in this document for an example. See “MATLAB COM Client Support” in the MATLAB External Interfaces documentation to learn more about ActiveX controls.

Adding Components to the GUIDE Layout Area

This topic tells you how to place components in the GUIDE layout area and give each component a unique identifier.

Note See “Creating Menus” on page 6-70 for information about adding menus to a GUI. See “Creating Toolbars” on page 6-84 for information about working with the toolbar.

---

1 Place components in the layout area according to your design.

- Drag a component from the palette and drop it in the layout area.
- Click a component in the palette and move the cursor over the layout area. The cursor changes to a cross. Click again to add the component in its default size, or click and drag to size the component as you add it.

The components listed in the following table need additional considerations.

If You Are Adding...	Then...
Panels or button groups	See "Adding a Component to a Panel or Button Group" on page 6-25.
ActiveX controls	See "Adding ActiveX Controls" on page 6-51.

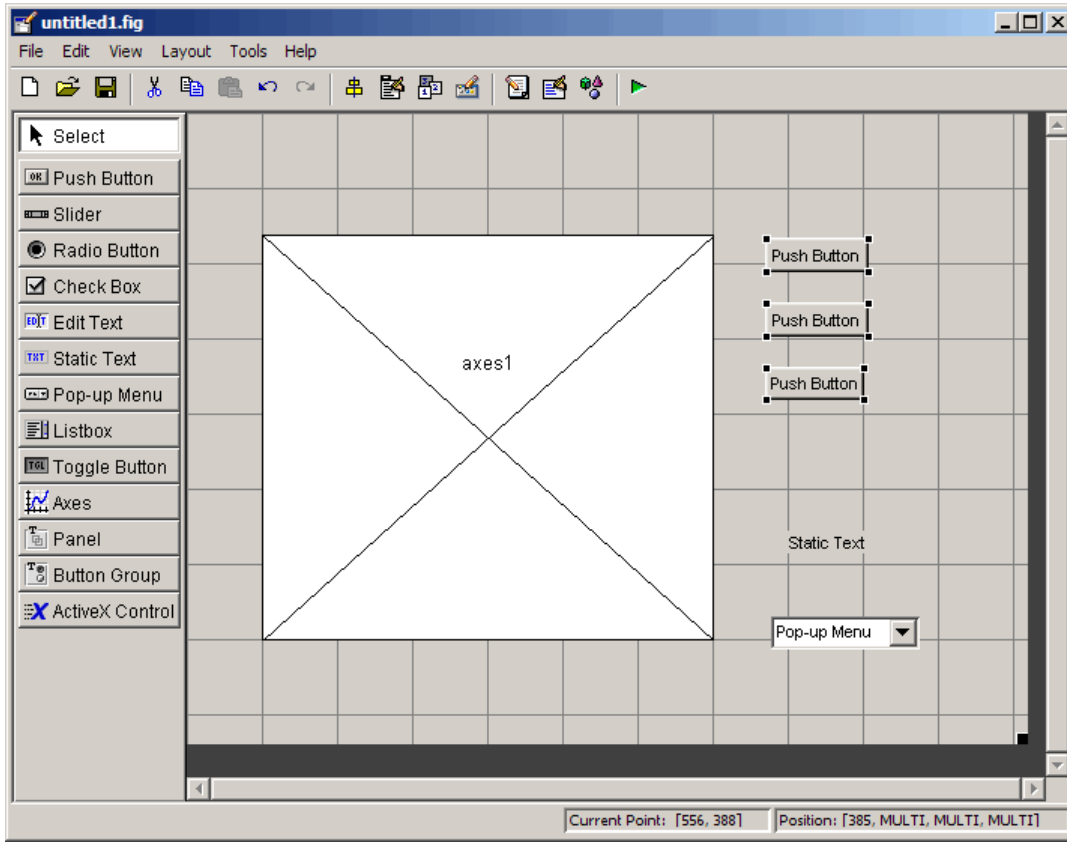
See "Grid and Rulers" on page 6-65 for information about using the grid.

2 Assign a unique identifier to each component. Do this by setting the value of the component `Tag` properties. See"Assigning an Identifier to Each Component" on page 6-27 for more information.

3 Specify the look and feel of each component by setting the appropriate properties. The following topics contain specific information.

- "Defining User Interface Controls" on page 6-27
- "Defining Panels and Button Groups" on page 6-43
- "Defining Axes" on page 6-48
- "Adding ActiveX Controls" on page 6-51

This is an example of a GUI in the Layout Editor. Components in the Layout Editor are not active. Chapter 7, “Saving and Running a GUIDE GUI” describes how to generate a functioning GUI.



Using Coordinates to Place Components

The status bar at the bottom of the GUIDE Layout Editor displays:

- **Current Point** — The current location of the mouse relative to the lower left corner of the grid area in the Layout Editor.
- **Position** — The `Position` property of the selected component, a 4-element vector: [distance from left, distance from bottom, width, height], where

distances are relative to the parent figure, panel, or button group. All values are given in pixels. Rulers also display pixels.

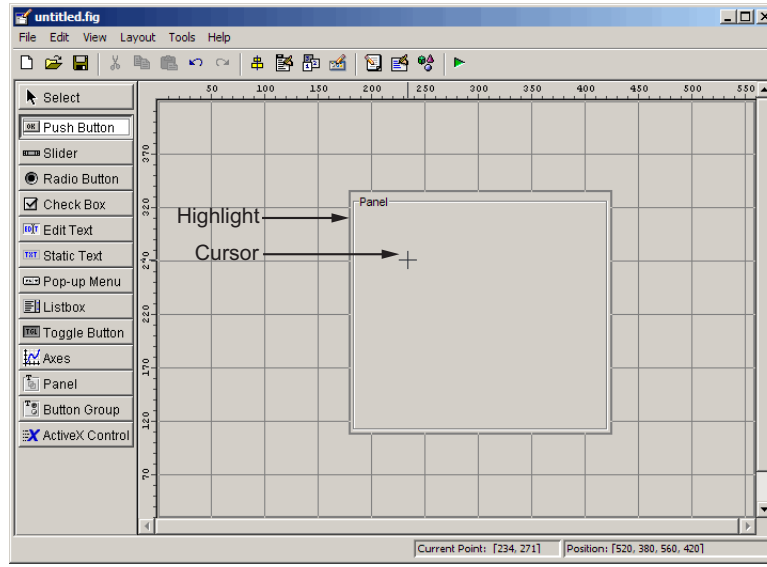
If you select a single component and move it, the first two elements of the position vector (distance from left, distance from bottom) are updated as you move the component. If you resize the component, the last two elements of the position vector (width, height) are updated as you change the size. The first two elements may also change if you resize the component such that the position of its lower left corner changes. If no components are selected, the position vector is that of the figure.

For more information, see "Using Coordinate Readouts" on page 6-57.

Adding a Component to a Panel or Button Group

To add a component to a panel or button group, select the component in the component palette then move the cursor over the desired panel or button group. The position of the cursor determines the component's parent.

GUIDE highlights the potential parent as shown in the following figure. The highlight indicates that if you drop the component or click the cursor, the component will be a child of the highlighted panel, button group, or figure.



Note If the component is not entirely contained in the panel or button group, it appears to be clipped in the Layout Editor. When you run the GUI, the entire component is displayed and straddles the panel or button group border. The component is nevertheless a child of the panel and behaves accordingly. You can use the Object Browser to determine the child objects of a panel or button group. “Viewing the Object Hierarchy” on page 6-100 tells you how.

Note Assign a unique identifier to each component in your panel or button group by setting the value of its `Tag` property. See “Assigning an Identifier to Each Component” on page 6-27 for more information.

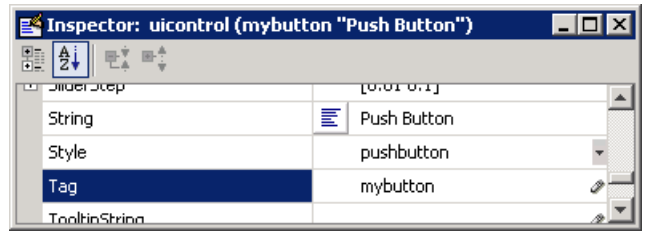
Assigning an Identifier to Each Component

Use the `Tag` property to assign each component a unique meaningful string identifier.

When you place a component in the layout area, GUIDE assigns a default value to the `Tag` property. Before saving the GUI, replace this value with a string that reflects the role of the component in the GUI.

The string value you assign `Tag` is used in the M-file code to identify the component and must be unique in the GUI. To set `Tag`:

1. Select **Property Inspector** from the **View** menu or click the **Property Inspector** button .
2. In the layout area, select the component for which you want to set `Tag`.
3. In the Property Inspector, select `Tag` and then replace the value with the string you want to use as the identifier. In the following figure, `Tag` is set to `mybutton`.



Defining User Interface Controls

User interface controls include push buttons, toggle buttons, sliders, radio buttons, edit text controls, static text controls, pop-up menus, check boxes, and list boxes.

To define user interface controls, you must set certain properties. To do this:

1. Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **Property Inspector** from the **View** menu or by clicking the Property Inspector button .

2 In the layout area, select the component you are defining.

Subsequent topics describe commonly used properties of user interface controls and offer a simple example for each kind of control:

Note See "Available Components" on page 6-19 for descriptions of these components. See "Examples: Programming GUIDE GUI Components" on page 8-20 for basic examples of programming these components.

Commonly Used Properties

The most commonly used properties needed to describe a user interface control are shown in the following table. Instructions for a particular control may also list properties that are specific to that control.

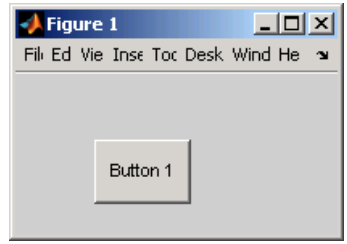
Property	Value	Description
`Enable`	`on`, `inactive`, `off`. Default is `on`.	Determines whether the control is available to the user

Property	Value	Description
`Max`	Scalar. Default is 1.	Maximum value. Interpretation depends on the type of component.
`Min`	Scalar. Default is 0.	Minimum value. Interpretation depends on the type of component.
`Position`	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
`String`	String. Can also be a cell array or character array of strings.	Component label. For list boxes and pop-up menus it is a list of the items.
`Units`	`characters`, `centimeters`, `inches`, `normalized`, `pixels`, `points`. Default is `characters`.	Units of measurement used to interpret the `Position` property vector
`Value`	Scalar or vector	Value of the component. Interpretation depends on the type of component.

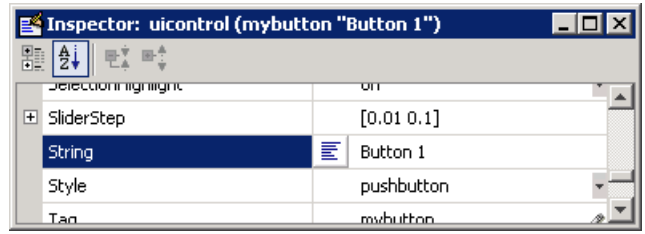
For a complete list of properties and for more information about the properties listed in the table, see Uicontrol Properties in the MATLAB documentation. Properties needed to control GUI behavior are discussed in Chapter 8, “Programming a GUIDE GUI”

Push Button

To create a push button with label **Button 1**, as shown in this figure:



- Specify the push button label by setting the `String` property to the desired label, in this case, `Button 1`.



To display the & character in a label, use two `&` characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

The push button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a push button that is too narrow to accommodate the specified `String`, MATLAB truncates the string with an ellipsis.

- If you want to set the position or size of the component to an exact value, then modify its `Position` property. See “Locating and Moving Components” on page 6-57 and “Resizing Components” on page 6-60 for details.
- To add an image to a push button, assign the button’s `CData` property an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the GUI M-file. For

example, the array `img` defines a 16-by-64-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

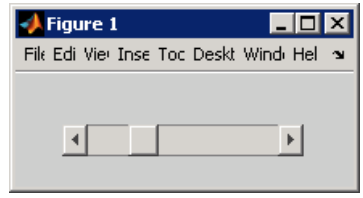
```
img = rand(16,64,3);
set(handles.pushbutton1,'CData',img);
```

where `pushbutton1` is the push button's `Tag` property.

Note Create your own icon with the icon editor described in "Icon Editor" on page 15-29. See `ind2rgb` for information on converting a matrix `X` and corresponding colormap, i.e., an `(X, MAP)` image, to RGB (truecolor) format.

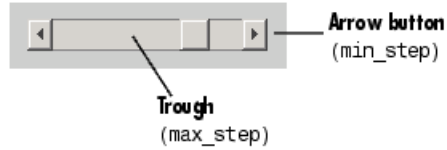
Slider

To create a slider as shown in this figure:



- Specify the range of the slider by setting its `Min` property to the minimum value of the slider and its `Max` property to the maximum value. The `Min` property must be less than `Max`.
- Specify the value indicated by the slider when it is created by setting the `Value` property to the appropriate number. This number must be less than or equal to `Max` and greater than or equal to `Min`. If you specify `Value` outside the specified range, the slider is not displayed.
- Control the amount the slider `Value` changes when a user clicks the arrow button to produce a minimum step or the slider trough to produce a

maximum step by setting the `SliderStep` property. Specify `SliderStep` as a two-element vector, `[min_step,max_step]`, where each value is in the range [0, 1] to indicate a percentage of the range.



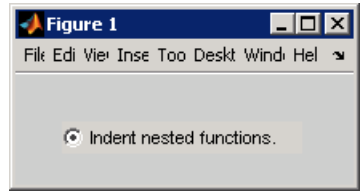
- If you want to set the position or size of the component to an exact value, then modify its `Position` property. See "Locating and Moving Components" on page 6-57 and "Resizing Components" on page 6-60 for details.

Note On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

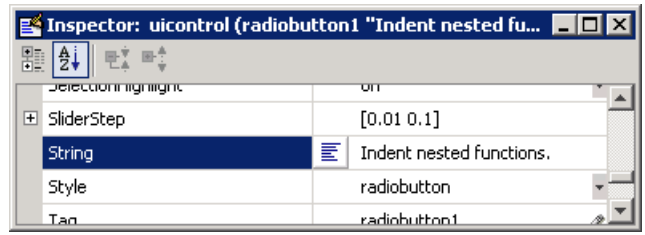
Note The slider component provides no text description or data entry capability. Use a "Static Text" on page 6-36 component to label the slider. Use an "Edit Text" on page 6-35 component to enable a user to provide a value for the slider.

Radio Button

To create a radio button with label **Indent nested functions**, as shown in this figure:



- Specify the radio button label by setting the `String` property to the desired label, in this case, `Indent nested functions`.



  To display the & character in a label, use two `&` characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

  The radio button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a radio button that is too narrow to accommodate the specified `String`, MATLAB truncates the string with an ellipsis.

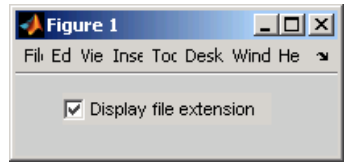
- Create the radio button with the button selected by setting its `Value` property to the value of its `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the radio button unselected. Correspondingly, when the user selects the radio button, MATLAB sets `Value` to `Max`. MATLAB sets `Value` to `Min` when the user deselects it.
- If you want to set the position or size of the component to an exact value, then modify its `Position` property. See “Locating and Moving Components” on page 6-57 and “Resizing Components” on page 6-60 for details.
- To add an image to a radio button, assign the button’s `CData` property an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the GUI M-file. For example, the array `img` defines a 16-by-24-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

```
img = rand(16,24,3);
set(handles.radiobutton1,'CData',img);
```

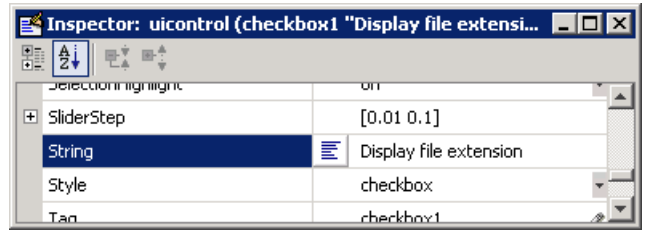
Note To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See “Button Group” on page 6-46 for more information.

Check Box

To create a check box with label **Display file extension** that is initially checked, as shown in this figure:



- Specify the check box label by setting the `String` property to the desired label, in this case, `Display file extension`.



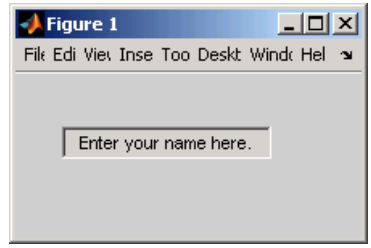
To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

The check box accommodates only a single line of text. If you specify a component width that is too small to accommodate the specified `String`, MATLAB truncates the string with an ellipsis.

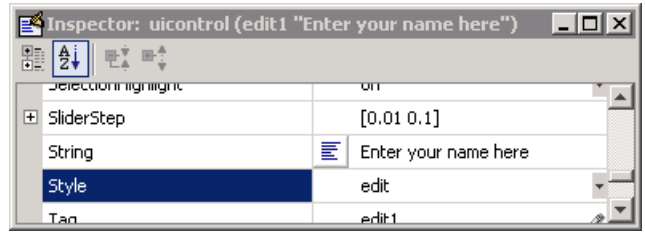
- Create the check box with the box checked by setting the `Value` property to the value of the `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the box unchecked. Correspondingly, when the user clicks the check box, MATLAB sets `Value` to `Max` when the user checks the box and to `Min` when the user unchecks it.
- If you want to set the position or size of the component to an exact value, then modify its `Position` property. See "Locating and Moving Components" on page 6-57 and "Resizing Components" on page 6-60 for details.

Edit Text

To create an edit text component that displays the initial text **Enter your name here**, as shown in this figure:

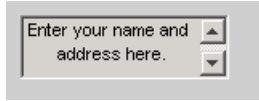


- Specify the text to be displayed when the edit text component is created by setting the `String` property to the desired string, in this case, `Enter your name here`.

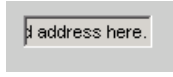


To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

- To enable multiple-line input, specify the `Max` and `Min` properties so that their difference is greater than 1. For example, `Max = 2`, `Min = 0`. `Max` default is `1`, `Min` default is `0`. MATLAB wraps the string and adds a scroll bar if necessary.



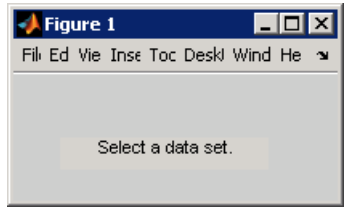
  If `Max-Min` is less than or equal to 1, the edit text component admits only a single line of input. If you specify a component width that is too small to accommodate the specified string, MATLAB displays only part of the string. The user can use the arrow keys to move the cursor over the entire string.



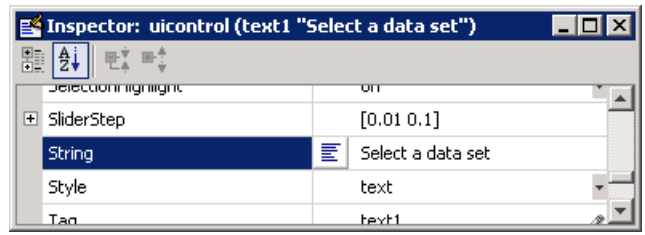
- If you want to set the position or size of the component to an exact value, then modify its `Position` property. See “Locating and Moving Components” on page 6-57 and “Resizing Components” on page 6-60 for details.

Static Text

To create a static text component with text **Select a data set**, as shown in this figure:

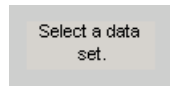


- Specify the text that appears in the component by setting the component `String` property to the desired text, in this case `Select a data set`.



To display the & character in a list item, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

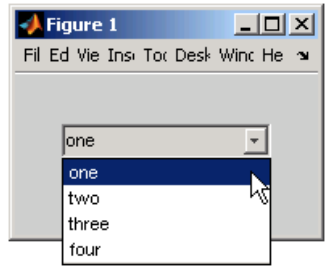
If your component is not wide enough to accommodate the specified `String`, MATLAB wraps the string.



- If you want to set the position or size of the component to an exact value, then modify its `Position` property. See “Locating and Moving Components” on page 6-57 and “Resizing Components” on page 6-60 for details.

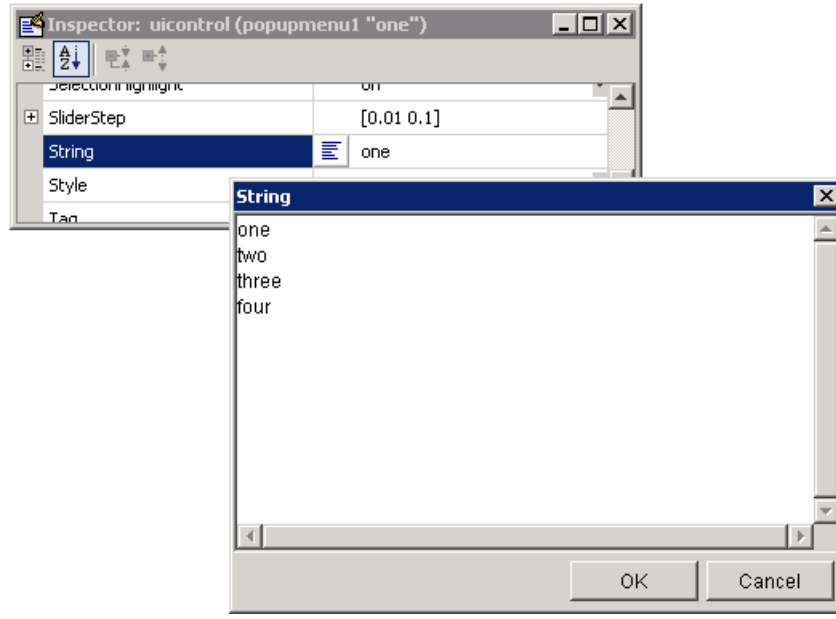
Pop-Up Menu

To create a pop-up menu (also known as a drop-down menu or combo box) with items **one**, **two**, **three**, and **four**, as shown in this figure:



- Specify the pop-up menu items to be displayed by setting the `String` property to the desired items. Click the

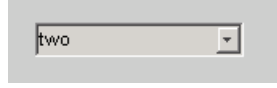
button to the right of the property name to open the Property Inspector editor.



To display the & character in a menu item, use two `&` characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

If the width of the component is too small to accommodate one or more of the specified strings, MATLAB truncates those strings with an ellipsis.

- To select an item when the component is created, set `Value` to a scalar that indicates the index of the selected list item, where `1` corresponds to the first item in the list. If you set `Value` to `2`, the menu looks like this when it is created:

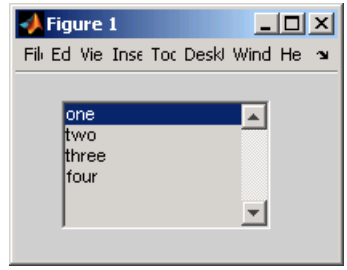


- If you want to set the position and size of the component to exact values, then modify its `Position` property. See “Locating and Moving Components” on page 6-57 and “Resizing Components” on page 6-60 for details. The height of a pop-up menu is determined by the font size. The height you set in the position vector is ignored.

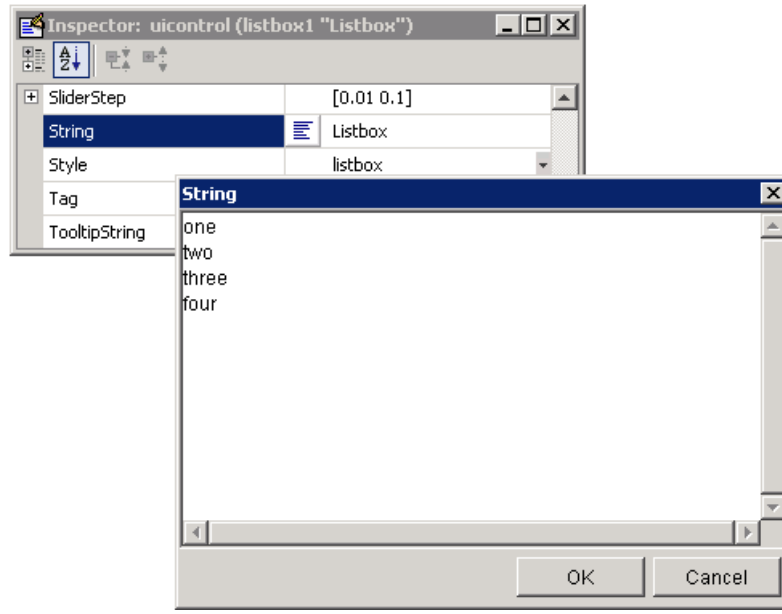
Note The pop-up menu does not provide for a label. Use a “Static Text” on page 6-36 component to label the pop-up menu.

List Box

To create a list box with items **one**, **two**, **three**, and **four**, as shown in this figure:



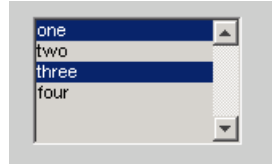
- Specify the list of items to be displayed by setting the `String` property to the desired list. Use the Property Inspector editor to enter the list. You can open the editor by clicking the  button to the right of the property name.



To display the & character in a label, use two `&` characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

If the width of the component is too small to accommodate one or more of the specified strings, MATLAB truncates those strings with an ellipsis.

- Specify selection by using the `Value` property together with the `Max` and `Min` properties.
  - To select a single item when the component is created, set `Value` to a scalar that indicates the index of the selected list item, where `1` corresponds to the first item in the list.
  - To select more than one item when the component is created, set `Value` to a vector of indices of the selected items. `Value = [1,3]` results in the following selection.



To enable selection of more than one item, you must specify the `Max` and `Min` properties so that their difference is greater than `1`. For example, `Max = 2`, `Min = 0`. `Max` default is `1`, `Min` default is `0`.

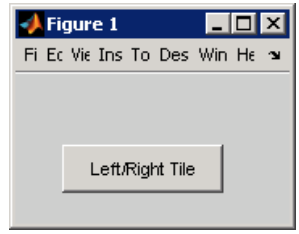
- If you want no initial selection, set the `Max` and `Min` properties to enable multiple selection, i.e., `Max - Min > 1`, and then set the `Value` property to an empty matrix `[]`.

- If the list box is not large enough to display all list entries, you can set the `ListBoxTop` property to the index of the item you want to appear at the top when the component is created.
- If you want to set the position or size of the component to an exact value, then modify its `Position` property. See "Locating and Moving Components" on page 6-57 and "Resizing Components" on page 6-60 for details.

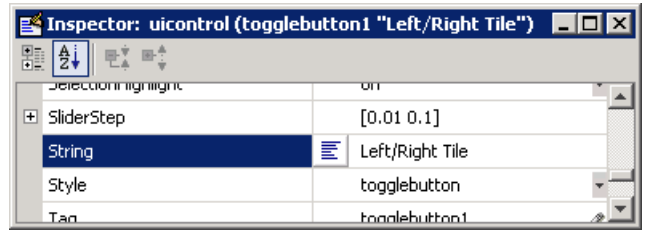
Note The list box does not provide for a label. Use a "Static Text" on page 6-36 component to label the list box.

Toggle Button

To create a toggle button with label **Left/Right Tile**, as shown in this figure:



- Specify the toggle button label by setting its `String` property to the desired label, in this case, `Left/Right Tile`.



To display the & character in a label, use two `&` characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

The toggle button accommodates only a single line of text. If you specify more than one line, only the first line is shown. If you create a toggle button that is too narrow to accommodate the specified `String`, MATLAB truncates the string with an ellipsis.

- Create the toggle button with the button selected (depressed) by setting its `Value` property to the value of its `Max` property (default is `1`). Set `Value` to `Min` (default is `0`) to leave the toggle button unselected (raised). Correspondingly, when the user selects the toggle button, MATLAB sets `Value` to `Max`. MATLAB sets `Value` to `Min` when the user deselects it. The following figure shows the toggle button in the depressed position.

- If you want to set the position or size of the component to an exact value, then modify its `Position` property. See “Locating and Moving Components” on page 6-57 and “Resizing Components” on page 6-60 for details.
- To add an image to a toggle button, assign the button’s `CData` property an m-by-n-by-3 array of RGB values that defines a truecolor image. You must do this programmatically in the opening function of the GUI M-file. For example, the array `img` defines a 16-by-64-by-3 truecolor image using random values between 0 and 1 (generated by `rand`).

```
img = rand(16,64,3);
set(handles.togglebutton1,'CData',img);
```

where `togglebutton1` is the toggle button's `Tag` property.

Note To manage exclusive selection of radio buttons and toggle buttons, put them in a button group. See "Button Group" on page 6-46 for more information.

Defining Panels and Button Groups

Panels and button groups are containers that arrange GUI components into groups. If you move the panel or button group, its children move with it and maintain their positions relative to the panel or button group.

To define panels and button groups, you must set certain properties. To do this:

1. Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **Property Inspector** from the **View** menu or by clicking the Property Inspector button . 
2. In the layout area, select the component you are defining.

Note See "Available Components" on page 6-19 for descriptions of these components. See "Examples: Programming GUIDE GUI Components" on page 8-20 for basic examples of programming these components.

Subsequent topics describe commonly used properties of panels and button groups and offer a simple example for each component.

Commonly Used Properties

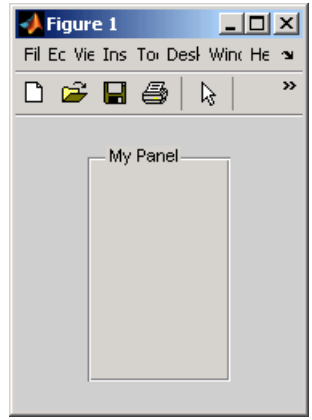
The most commonly used properties needed to describe a panel or button group are shown in the following table:

Property	Values	Description
Position	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
Title	String	Component label.
TitlePosition	`lefttop`, `centertop`, `righttop`, `leftbottom`, `centerbottom`, `rightbottom`. Default is `lefttop`.	Location of title string in relation to the panel or button group.
Units	`characters`, `centimeters`, `inches`, normalized, `pixels`, `points`. Default is `characters`.	Units of measurement used to interpret the `Position` property vector

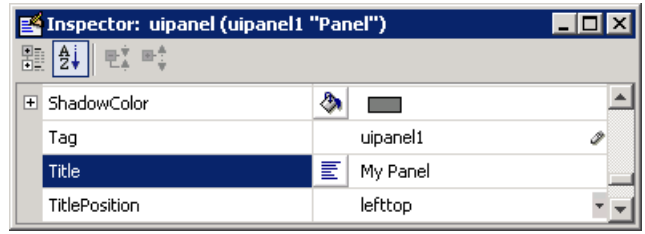
For a complete list of properties and for more information about the properties listed in the table, see the Uipanel Properties and Uibuttongroup Properties in the MATLAB reference documentation. Properties needed to control GUI behavior are discussed in theChapter 8, "Programming a GUIDE GUI".

Panel

To create a panel with title **My Panel** as shown in the following figure:

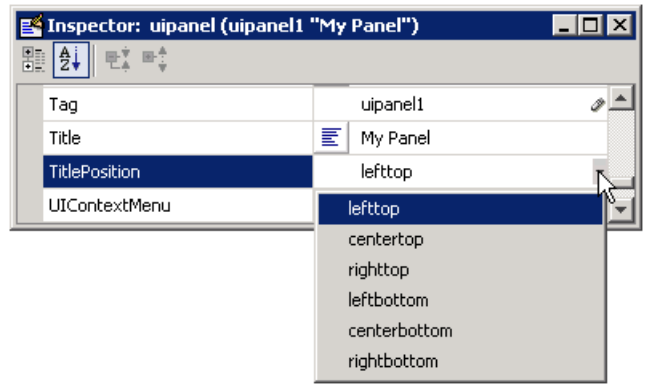


- Specify the panel title by setting the `Title` property to the desired string, in this case `My Panel`.



  To display the & character in the title, use two & characters in the string. The words remove, default, and factory (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove**.

- Specify the location of the panel title by selecting one of the available `TitlePosition` property values from the pop-up menu, in this case `lefttop`. You can position the title at the left, middle, or right of the top or bottom of the panel.

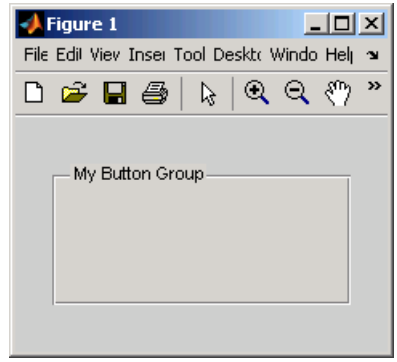


- If you want to set the position or size of the panel to an exact value, then modify its `Position` property. See "Locating and Moving Components" on page 6-57 and "Resizing Components" on page 6-60 for details.

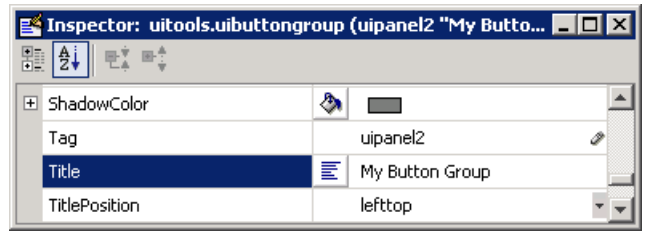
Note For information about adding components to a panel, see "Adding a Component to a Panel or Button Group" on page 6-25.

Button Group

To create a button group with title **My Button Group** as shown in the following figure:

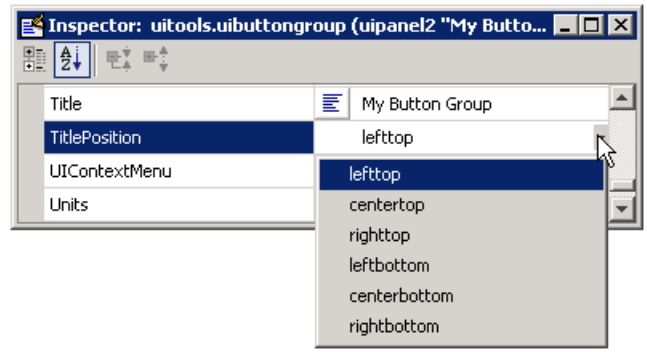


- Specify the button group title by setting the `Title` property to the desired string, in this case `My Button Group`.



  To display the & character in the title, use two & characters in the string. The words remove, default, and factory (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove**.

- Specify the location of the button group title by selecting one of the available `TitlePosition` property values from the pop-up menu, in this case `lefttop`. You can position the title at the left, middle, or right of the top or bottom of the button group.



- If you want to set the position or size of the button group to an exact value, then modify its `Position` property. See “Locating and Moving Components” on page 6-57 and “Resizing Components” on page 6-60 for details.

Note For information about adding components to a button group, see “Adding a Component to a Panel or Button Group” on page 6-25.

Defining Axes

Axes enable your GUI to display graphics such as graphs and images using commands such as: `plot`, `surf`, `line`, `bar`, `polar`, `pie`, `contour`, and `mesh`.

To define an axes, you must set certain properties. To do this:

1. Use the Property Inspector to modify the appropriate properties. Open the Property Inspector by selecting **Property Inspector** from the **View** menu or by clicking the Property Inspector button.
2. In the layout area, select the component you are defining.

Note See"Available Components" on page 6-19 for a description of this component.

Subsequent topics describe commonly used properties of axes and offer a simple example.

- "Commonly Used Properties" on page 6-48
- "Axes" on page 6-49

Commonly Used Properties

The most commonly used properties needed to describe an axes are shown in the following table:

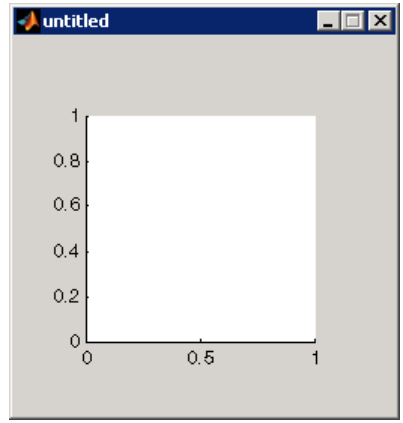
Property	Values	Description
Position	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
Units	`normalized`, `centimeters`, `characters`, `inches`, `pixels`, `points`. Default is `normalized`.	Units of measurement used to interpret position vector

For a complete list of properties and for more information about the properties listed in the table, see Axes Properties in the MATLAB documentation. Properties needed to control GUI behavior are discussed in Chapter 8, "Programming a GUIDE GUI".

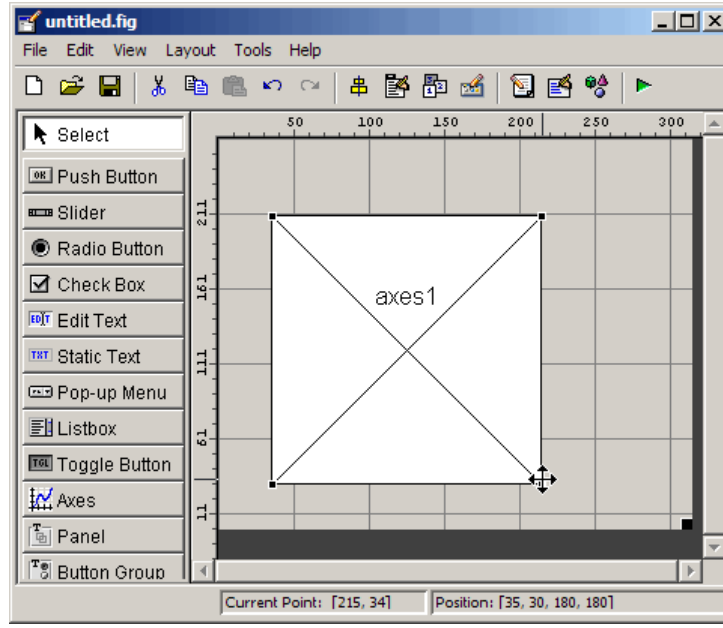
See commands such as the following for more information on axes objects: `plot`, `surf`, `line`, `bar`, `polar`, `pie`, `contour` and `mesh`. See Functions — By Category in the MATLAB Function Reference documentation for a complete list.

Axes

To create an axes as shown in the following figure:



- Allow for tick marks to be placed outside the box that appears in the Layout Editor. The axes above looks like this in the layout editor; placement allows space at the left and bottom of the axes for tick marks. Functions that draw in the axes update the tick marks appropriately.



- Use the `title`, `xlabel`, `ylabel`, `zlabel`, and `text` functions in the GUI M-file to label an axes component. For example,

  ```
  xlh = (axes_handle,'Years')
  ```

  labels the X-axis as `Years`. The handle of the X-axis label is `xlh`. See "Callback Syntax and Arguments" on page 8-12 for information about determining the axes handle.

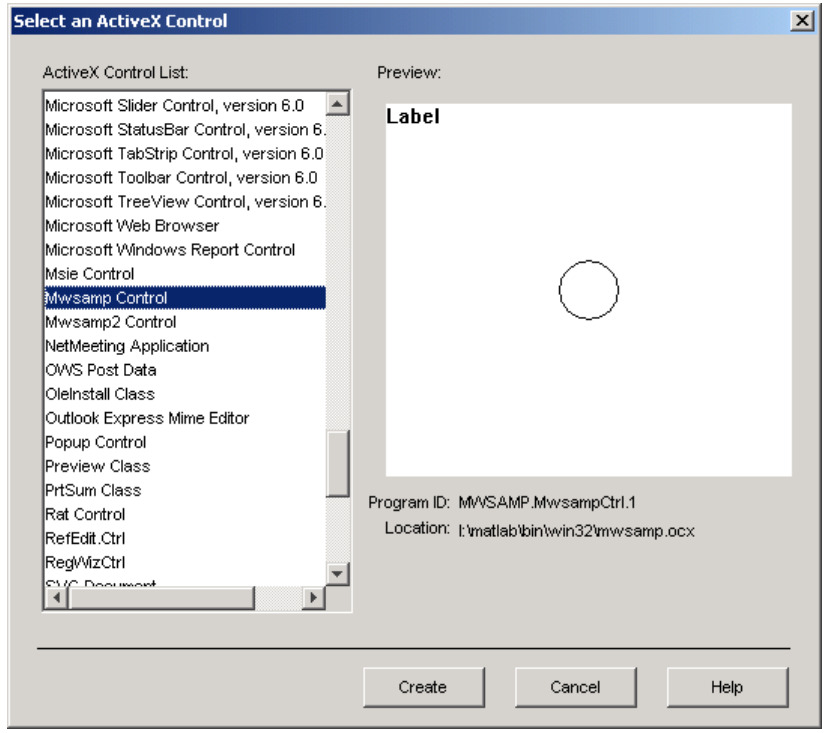
  The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these in component text, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

- If you want to set the position or size of the axes to an exact value, then modify its `Position` property. See "Locating and Moving Components" on page 6-57 and "Resizing Components" on page 6-60 for details.

Adding ActiveX Controls

When you drag an ActiveX component from the component palette into the layout area, GUIDE opens a dialog box, similar to the following, that lists the registered ActiveX controls on your system.

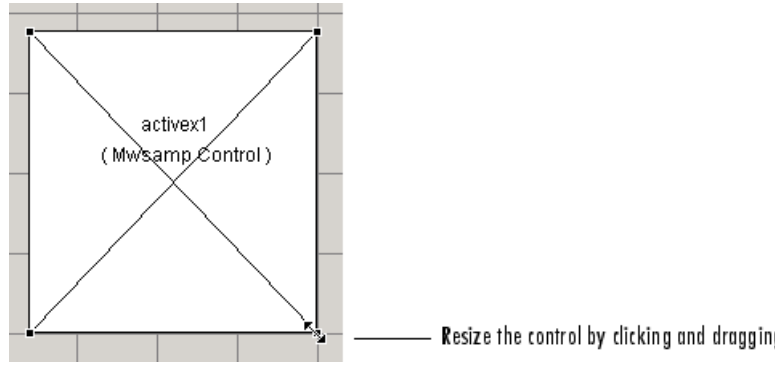
Note If MATLAB is not installed locally on your computer — for example, if you are running MATLAB over a network — you might not find the ActiveX control described in this example. To register the control, see "Registering Controls and Servers" in the MATLAB External Interfaces documentation.



1 Select the desired ActiveX control. The right panel shows a preview of the selected control.

2 Click **Create**. The control appears as a small box in the Layout Editor.

3 Resize the control to approximately the size of the square shown in the preview pane. You can do this by clicking and dragging a corner of the control, as shown in the following figure.



See “ActiveX Control” on page 8-33 for information about programming a sample ActiveX control and an example.

Working with Components in the Layout Area

This topic provides basic information about selecting, copying, pasting, and deleting components in the layout area.

Other topics that may be of interest are

Selecting Components

You can select components in the layout area in the following ways:

- Click a single component to select it.
- Press **Ctrl+A** to select all child objects of the figure. This does not select components that are child objects of panels or button groups.
- Click and drag the cursor to create a rectangle that encloses the components you want to select. If the rectangle encloses a panel or button group, only the panel or button group is selected, not its children. If the rectangle encloses part of a panel or button group, only the components within the rectangle that are child objects of the panel or button group are selected.
- Select multiple components using the **Shift** and **Ctrl** keys.

In some cases, a component may lie outside its parent's boundary. Such a component is not visible in the Layout Editor but can be selected by dragging a rectangle that encloses it or by selecting it in the Object Browser. Such a component is visible in the active GUI.

See "Viewing the Object Hierarchy" on page 6-100 for information about the Object Browser.

Note You can select multiple components only if they have the same parent. To determine the child objects of a figure, panel, or button group, use the Object Browser.

Copying, Cutting, and Clearing Components

Use standard menu and pop-up menu commands, toolbar icons, keyboard keys, and shortcut keys to copy, cut, and clear components.

Copying. Copying places a copy of the selected components on the clipboard. A copy of a panel or button group includes its children.

Cutting. Cutting places a copy of the selected components on the clipboard and deletes them from the layout area. If you cut a panel or button group, you also cut its children.

Clearing. Clearing deletes the selected components from the layout area. It does not place a copy of the components on the clipboard. If you clear a panel or button group, you also clear its children.

Pasting and Duplicating Components

Pasting. Use standard menu and pop-up menu commands, toolbar icons, and shortcut keys to paste components. GUIDE pastes the contents of the clipboard to the location of the last mouse click. It positions the upper-left corner of the contents at the mouse click.

Consecutive pastes place each copy to the lower right of the last one.

Duplicating. Select one or more components that you want to duplicate, then do one of the following:

- Copy and paste the selected components as described above.
- Select **Duplicate** from the **Edit** menu or the pop-up menu. **Duplicate** places the copy to the lower right of the original.
- Right-click and drag the component to the desired location. The position of the cursor when you drop the components determines the parent of all the selected components. Look for the highlight as described in “Adding a Component to a Panel or Button Group” on page 6-25.

Front-to-Back Positioning

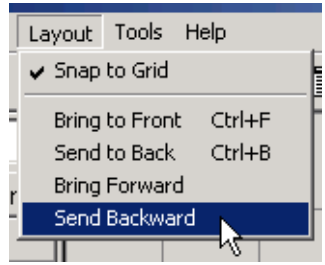
MATLAB figures maintain separate stacks that control the front-to-back positioning for different kinds of components:

- User interface controls such as buttons, sliders, and pop-up menus
- Panels, button groups, and axes
- ActiveX controls

You can control the front-to-back positioning of components that overlap only if those components are in the same stack. For overlapping components that are in different stacks:

- User interface controls always appear on top of panels, button groups, axes that they overlap. ActiveX controls appear on top of everything they overlap.
- Panels, button groups, and axes always appear on top of ActiveX controls.

The Layout Editor provides four operations that enable you to control front-to-back positioning. All are available from the **Layout** menu, which is shown in the following figure.



- **Bring to Front** — Move the selected object(s) in front of nonselected objects (available from the right-click context menu, the **Layout** menu, or the **Ctrl+F** shortcut).
- **Send to Back** — Move the selected object(s) behind nonselected objects (available from the right-click context menu, the **Layout** menu, or the **Ctrl+B** shortcut).
- **Bring Forward** — Move the selected object(s) forward by one level, i.e., in front of the object directly forward of it, but not in front of all objects that overlay it (available from the **Layout** menu).
- **Send Backward** — Move the selected object(s) back by one level, i.e., behind the object directly in back of it, but not behind all objects that are behind it (available from the **Layout** menu).

Note Changing front-to-back positioning of components also changes their tab order. See “Setting Tab Order” on page 6-67 for more information.

Locating and Moving Components

You can locate or move components in one of the following ways:

- "Using Coordinate Readouts" on page 6-57
- "Dragging Components" on page 6-58
- "Using Arrow Keys to Move Components" on page 6-58
- "Setting the Component's Position Property" on page 6-58

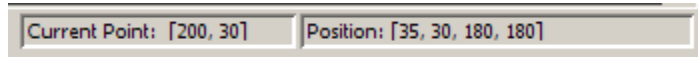
Another topic that may be of interest is

- "Aligning Components" on page 6-62

Using Coordinate Readouts

Coordinate readouts indicate where a component is placed and where the mouse pointer is located. Use these readouts to position and align components manually. The coordinate readout in the lower right corner of the Layout Editor shows the position of a selected component or components as `[xleft ybottom width height]`. These values are displayed in units of pixels, regardless of the coordinate units you select for components.

If you drag or resize the component, the readout updates accordingly. The readout to the left of the component position readout displays the current mouse position, also in pixels. The following readout example shows a selected component that has a position of `[35, 30, 180, 180]`, a 180-by-180 pixel object with a lower left corner at `x=35` and `y=30`, and locates the mouse position at `[200, 30]`.



When you select multiple objects, the **Position** readout displays numbers for `x`, `y`, `width` and `height` only if the objects have the same respective values; in all other cases it displays `'MULTI'`. For example, if you select two checkboxes, one with Position `[250, 140, 76, 20]` pixels and the other with position `[250, 190, 68, 20]` pixels, the **Position** readout indicates `[250, MULTI, MULTI, 20]`.

Dragging Components

Select one or more components that you want to move, then drag them to the desired position and drop them. You can move components from the figure into a panel or button group. You can move components from a panel or button group into the figure or into another panel or button group.

The position of the cursor when you drop the components also determines the parent of all the selected components. Look for the highlight as described in "Adding a Component to a Panel or Button Group" on page 6-25.

In some cases, one or more of the selected components may lie outside its parent's boundary. Such a component is not visible in the Layout Editor but can be selected by dragging a rectangle that encloses it or by selecting it in the Object Browser. Such a component is visible in the active GUI.

See "Viewing the Object Hierarchy" on page 6-100 for information about the Object Browser.

Note To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group.

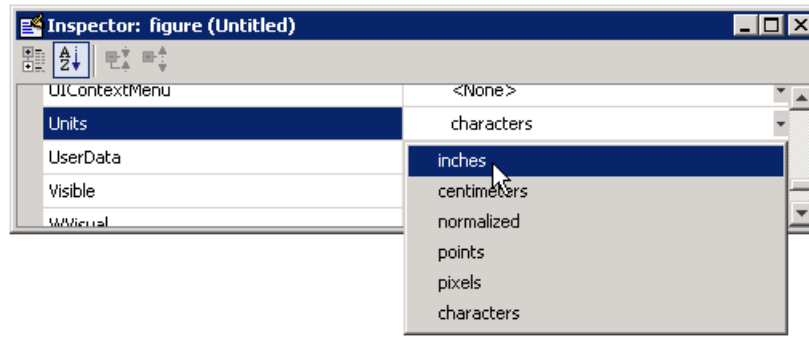
Using Arrow Keys to Move Components

Select one or more components that you want to move, then press and hold the arrow keys until the components have moved to the desired position. Note that the components remain children of the figure, panel, or button group from which you move them, even if they move outside its boundaries.

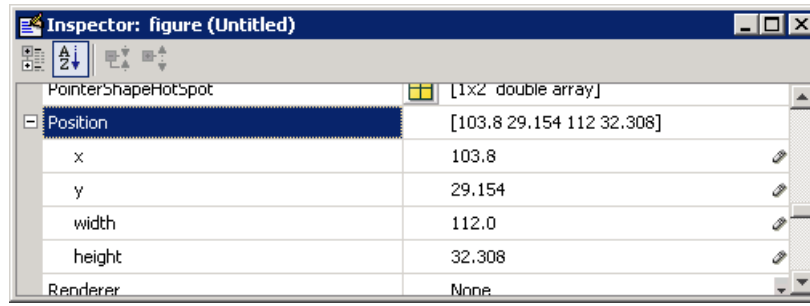
Setting the Component's Position Property

Select one or more components that you want to move. Then open the Property Inspector from the **View** menu or by clicking the Property Inspector button  .

1. In the Property Inspector, scroll to the `Units` property and note whether the current setting is `characters` or `normalized`. Click the button next to `Units` and then change the setting to `inches` from the pop-up menu.



2 Click the **+** sign next to `Position`. The Property Inspector displays the elements of the `Position` property.



3 If you have selected

- Only one component, type the `x` and `y` coordinates of the point where you want the lower-left corner of the component to appear.
- More than one component, type either the `x` or the `y` coordinate to align the components along that dimension.

4 Reset the `Units` property to its previous setting, either `characters` or `normalized`.

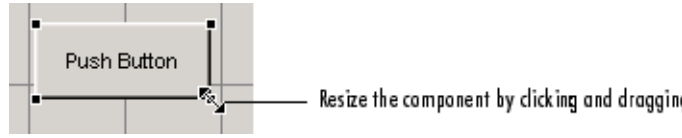
Note Setting the `Units` property to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) gives the GUI a more consistent appearance across platforms. See "Cross-Platform Compatible Units" on page 6-103 for more information.

Resizing Components

You can resize components in one of the following ways:

Dragging a Corner of the Component

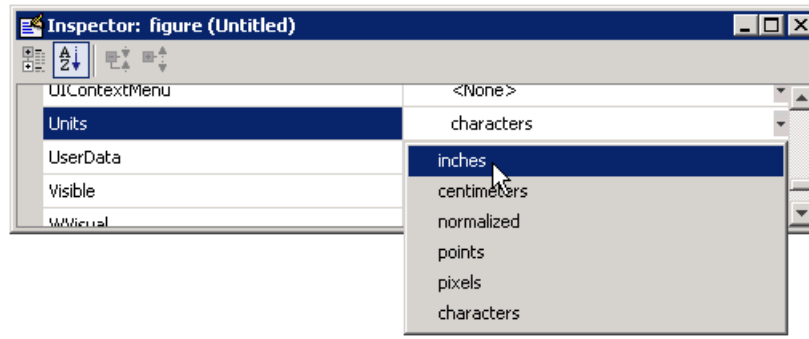
Select the component you want to resize. Click one of the corner handles and drag it until the component is the desired size.



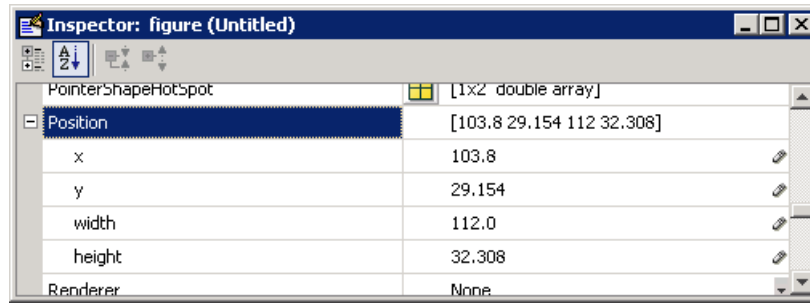
Setting the Component's Position Property

Select one or more components that you want to resize. Then open the Property Inspector from the **View** menu or by clicking the Property Inspector button .

1. In the Property Inspector, scroll to the `Units` property and note whether the current setting is `characters` or `normalized`. Click the button next to `Units` and then change the setting to `inches` from the pop-up menu.



2 Click the **+** sign next to `Position`. The Property Inspector displays the elements of the `Position` property.



3 Type the `width` and `height` you want the components to be.

4 Reset the `Units` property to its previous setting, either `characters` or `normalized`.

Note To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group. See "Selecting Components" on page 6-54 for more information. Setting the `Units` property to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) gives the GUI a more consistent appearance across platforms. See "Cross-Platform Compatible Units" on page 6-103 for more information.

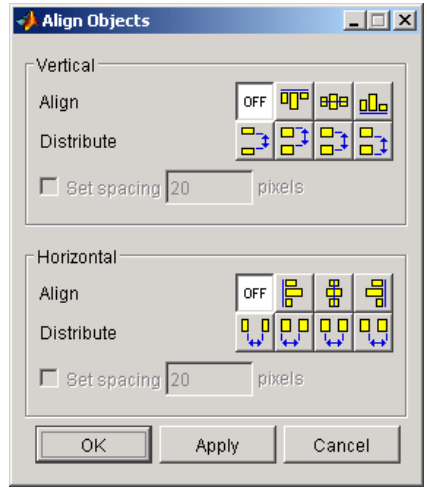
Aligning Components

In this section...

Alignment Tool

The Alignment Tool enables you to position objects with respect to each other and to adjust the spacing between selected objects. The specified alignment operations apply to all components that are selected when you press the **Apply** button.

Note To select multiple components, they must have the same parent. That is, they must be contained in the same figure, panel, or button group. See “Selecting Components” on page 6-54 for more information.



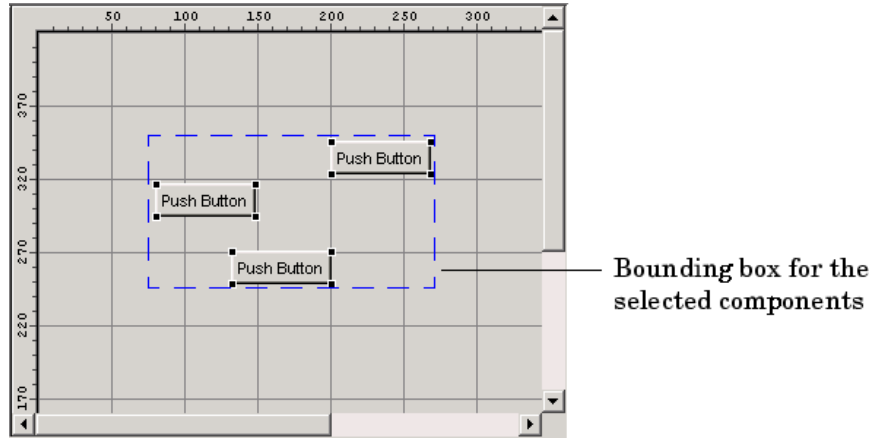
The alignment tool provides two types of alignment operations:

- **Align** — Align all selected components to a single reference line.
- **Distribute** — Space all selected components uniformly with respect to each other.

Both types of alignment can be applied in the vertical and horizontal directions. In many cases, it is better to apply alignments independently to the vertical and horizontal using two separate steps.

Align Options

There are both vertical and horizontal align options. Each option aligns selected components to a reference line, which is determined by the bounding box that encloses the selected objects. For example, the following picture of the layout area shows the bounding box (indicated by the dashed line) formed by three selected push buttons.



All of the align options (vertical top, center, bottom and horizontal left, center, right) place the selected components with respect to the corresponding edge (or center) of this bounding box.

Distribute Options

Distributing components adds equal space between all components in the selected group. The distribute options operate in two different modes:

- Equally space selected components within the bounding box (default)
- Space selected components to a specified value in pixels (check **Set spacing** and specify a pixel value)

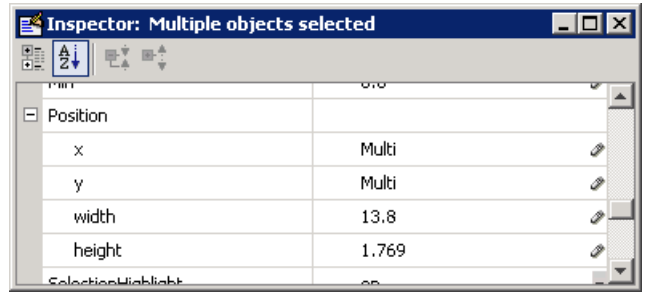
Both modes enable you to specify how the spacing is measured, as indicated by the button labels on the alignment tool. These options include spacing measured with respect to the following edges:

- Vertical — inner, top, center, and bottom
- Horizontal — inner, left, center, and right

Property Inspector

The Property Inspector enables you to align components by setting their `Position` properties. A component's `Position` property is a 4-element vector that specifies the location of the component on the GUI and its size: [distance from left, distance from bottom, width, height]. The values are given in the units specified by the `Units` property of the component.

1. Select the components you want to align. See "Selecting Components" on page 6-54 for information.
2. Select **Property Inspector** from the **View** menu or click the **Property Inspector** button  .
3. In the Property Inspector, scroll to the `Units` property and note its current setting, then change the setting to `inches`.
4. Scroll to the `Position` property. A `null` value means that the element differs in value for the different components. This figure shows the `Position` property for multiple components of the same size.

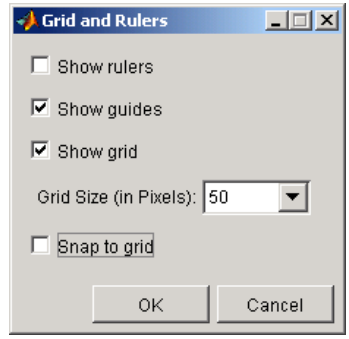


5 Change the value of `x` to align their left sides. Change the value of `y` to align their bottom edges. For example, setting `x` to 2.0 aligns the left sides of the components 2 inches from the left side of the GUI.

6 When the components are aligned, change the `Units` property back to its original setting.

Grid and Rulers

The layout area displays a grid and rulers to facilitate component layout. Grid lines are spaced at 50-pixel intervals by default and you can select from a number of other values ranging from 10 to 200 pixels. You can optionally enable *snap-to-grid*, which causes any object that is moved close to a grid line to jump to that line. Snap-to-grid works with or without a visible grid.



Use the Grid and Rulers dialog (select **Grid and Rulers** from the **Tools** menu) to:

- Control visibility of rulers, grid, and guide lines

- Set the grid spacing
- Enable or disable snap-to-grid

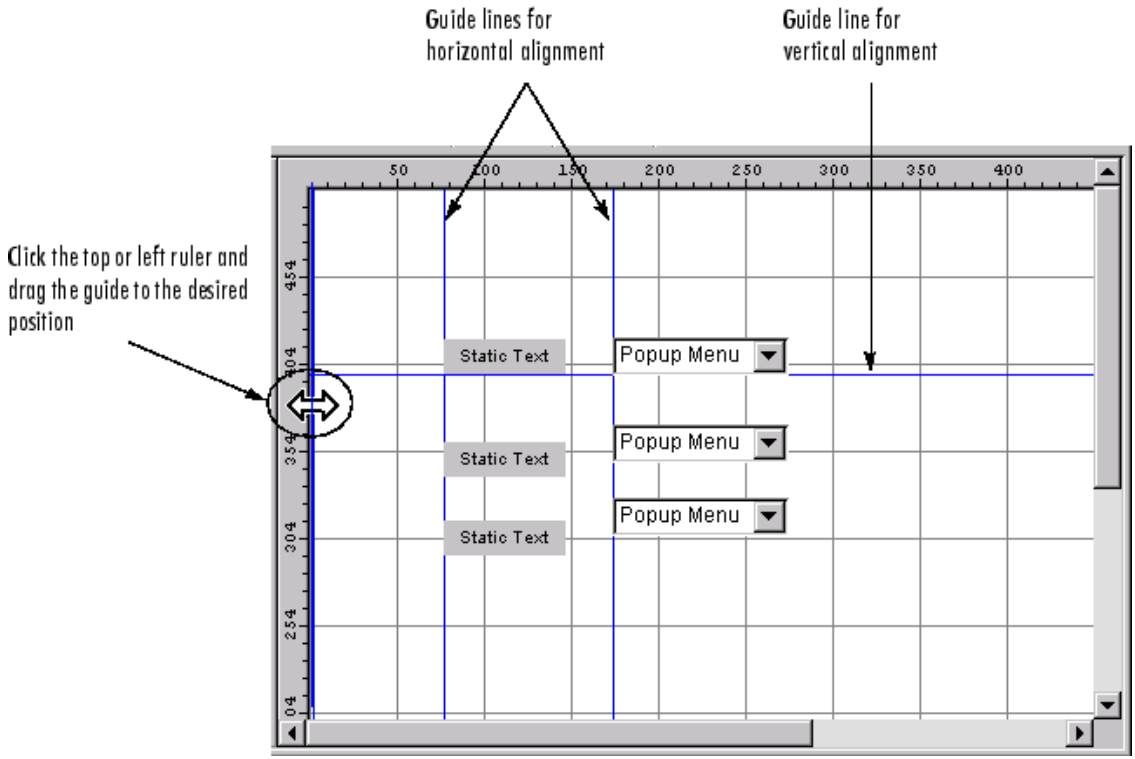
Guide Lines

The Layout Editor has both vertical and horizontal snap-to guide lines. Components snap to the line when you move them close to the line.

Guide lines are useful when you want to establish a reference for component alignment at an arbitrary location in the Layout Editor.

Creating Guide Lines

To create a guide line, click the top or left ruler and drag the line into the layout area.



Setting Tab Order

A GUI's tab order is the order in which components of the GUI acquire focus when a user presses the **Tab** key on the keyboard. Focus is generally denoted by a border or a dotted border.

You can set, independently, the tab order of components that have the same parent. The GUI figure and each panel and button group in it has its own tab order. For example, you can set the tab order of components that have the figure as a parent. You can also set the tab order of components that have a panel or button group as a parent.

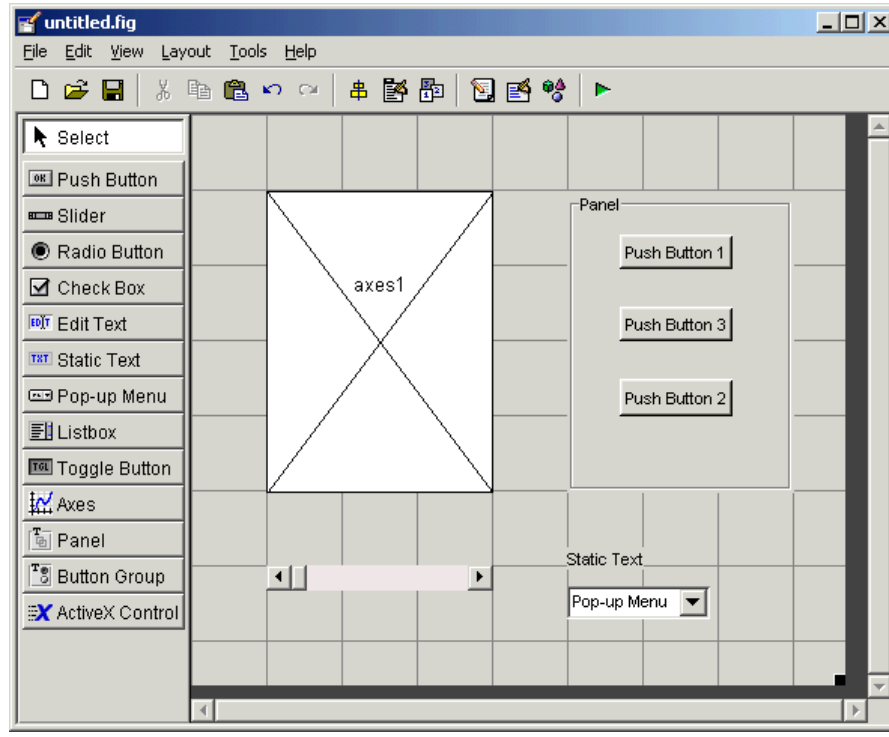
If, in tabbing through the components at the figure level, a user tabs to a panel or button group, then subsequent tabs sequence through the components of the panel or button group before returning to the level from which the panel or button group was reached.

Note Axes cannot be tabbed. From GUIDE, you cannot include ActiveX components in the tab order.

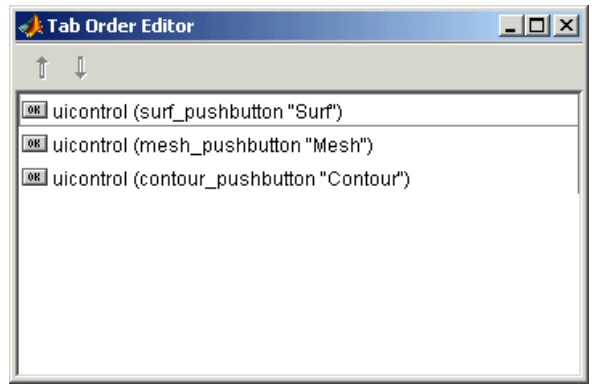
When you create a GUI, GUIDE sets the tab order at each level to be the order in which you add components to that level in the Layout Editor. This may not be the best order for the user.

Note Tab order also affects the stacking order of components. If components overlap, those that appear lower in the tabbing order, are drawn on top of those that appear higher in the order. See "Front-to-Back Positioning" on page 6-55 for more information.

The figure in the following GUI contains an axes component, a slider, a panel, static text, and a pop-up menu. Of these, only the slider, the panel, and the pop-up menu at the figure level can be tabbed. The panel contains three push buttons, which can all be tabbed.



To examine and change the tab order of the panel components, click the panel background to select it, then select **Tab Order Editor** in the **Tools** menu of the Layout Editor.



The Tab Order Editor displays the panel's components in their current tab order. To change the tab order, select a component and press the up or down arrow to move the component up or down in the list. If you set the tab order for the three components in the example to be

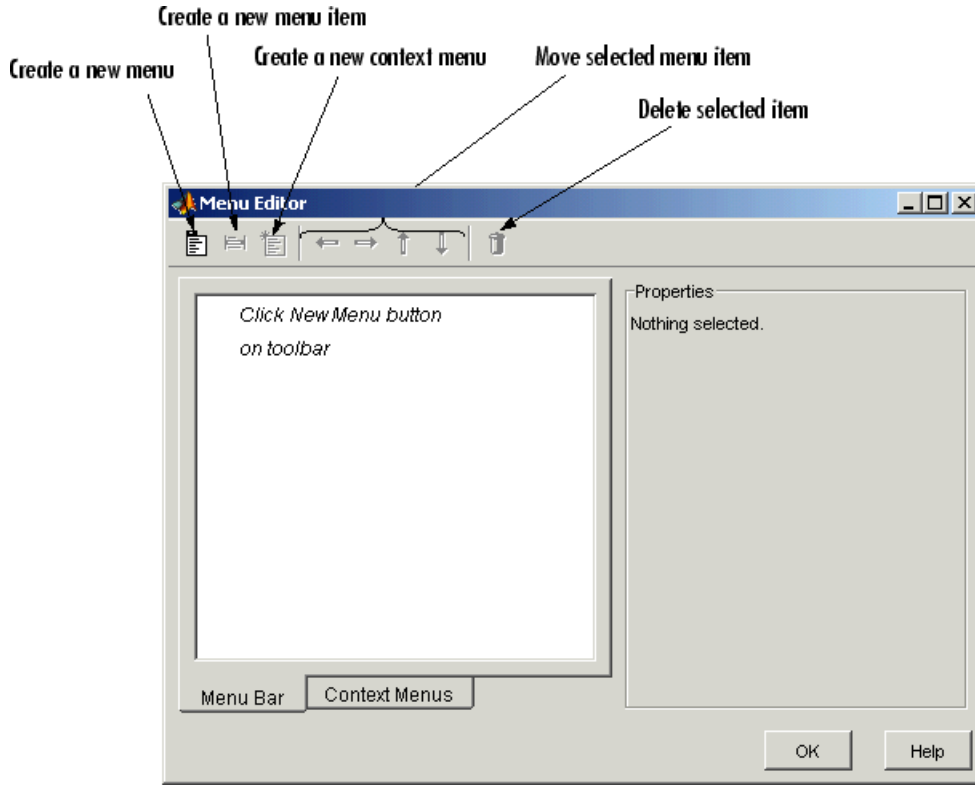
1. **Surf** push button
2. **Contour** push button
3. **Mesh** push button

the user first tabs to the **Surf** push button, then to the **Contour** push button, and then to the **Mesh** push button. Subsequent tabs sequence through the remaining components at the figure level.

Creating Menus

In this section...

You can create both types of menus using the Menu Editor. Access the Menu Editor from the **Tools** menu or click the **Menu Editor** button .



Note In general, programming conventions described for components in Chapter 8, "Programming a GUIDE GUI" also apply to menu items. See "Menu Item" on page 8-41 and "Updating a Menu Item Check" on page 8-42 for information about programming and basic examples.

Menus for the Menu Bar

When you create a drop-down menu, GUIDE adds its title to the GUI menu bar. You can then create menu items for that menu. Each menu item can have a cascading menu, also known as a submenu, and these items can have cascading menus, and so on.

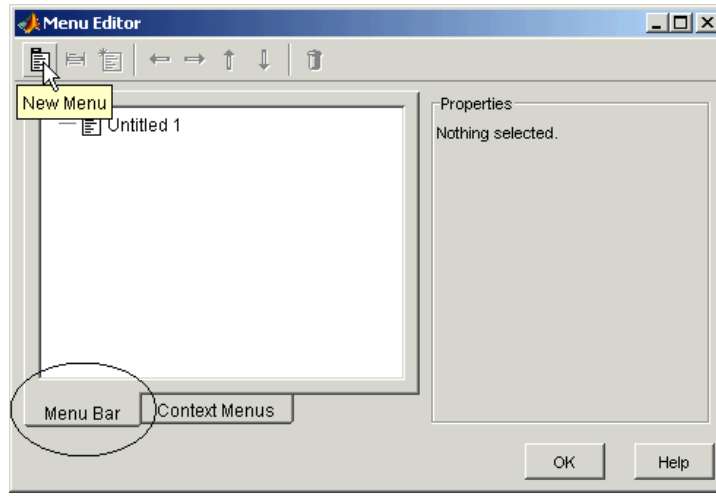
Adding Standard Menus to the Menu Bar

The figure `MenuBar` property controls whether your GUI displays the MATLAB standard menus on the menu bar. GUIDE initially sets the value of `MenuBar` to `none`. If you want your GUI to display the MATLAB standard menus, use the Property Inspector to set `MenuBar` to `figure`.

- If the value of `MenuBar` is `none`, GUIDE automatically adds a menu bar that displays only the menus you create.
- If the value of `MenuBar` is `figure`, the GUI displays the MATLAB standard menus and GUIDE adds the menus you create to this menu bar.

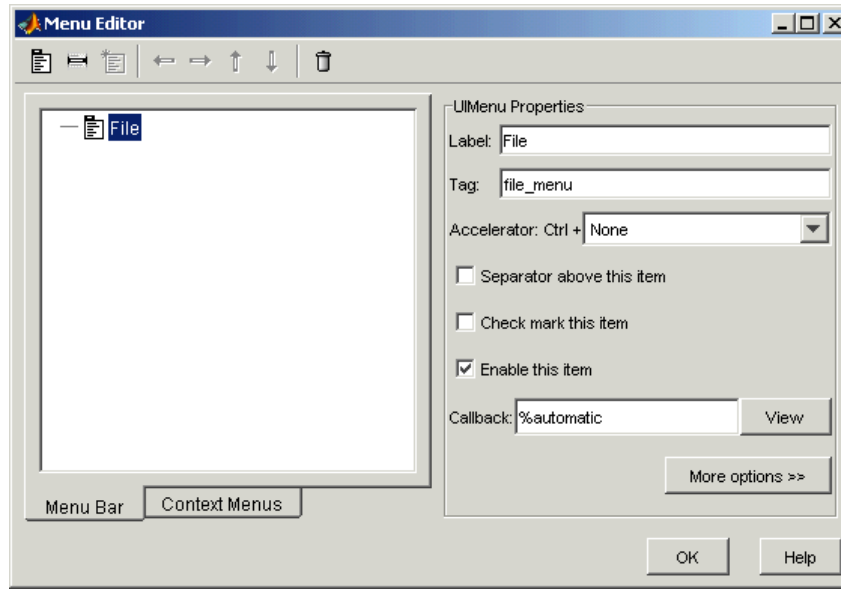
Creating a Menu

1 Start a new menu by clicking the New Menu button in the toolbar. A menu title, `Untitled 1`, appears in the left pane of the dialog box.



Note By default, GUIDE selects the **Menu Bar** tab when you open the Menu Editor.

2 Click the menu title to display a selection of menu properties in the right pane.



3 Fill in the **Label** and **Tag** fields for the menu. For example, set **Label** to `File` and set **Tag** to `file_menu`. Click outside the field for the change to take effect.

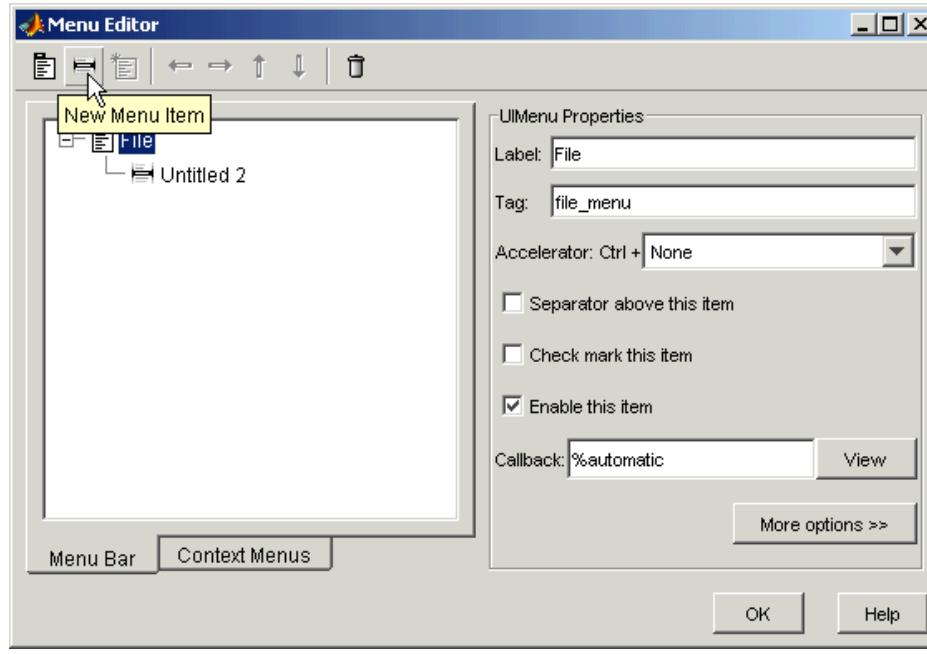
Label is a string that specifies the text label for the menu item. To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as labels, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove.**

Tag is a string that is an identifier for the menu object. It is used in the code to identify the menu item and must be unique in the GUI.

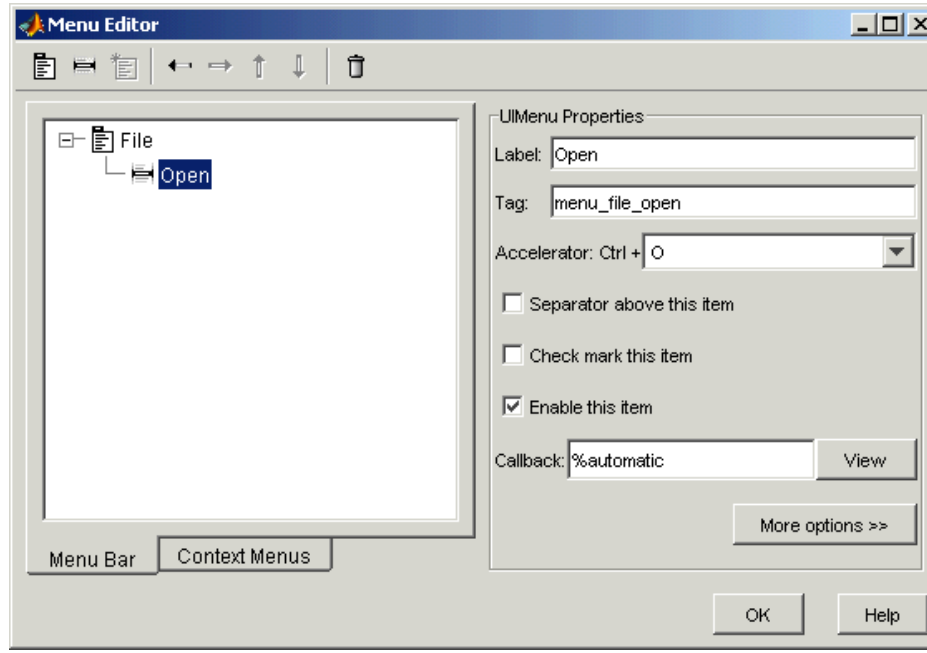
Adding Items to a Menu

Use the **New Menu Item** tool to create menu items that are displayed in the drop-down menu.

1. Add an **Open** menu item under `File`, by selecting `File` then clicking the **New Menu Item** button in the toolbar. A temporary numbered menu item label, `Untitled`, appears.



2 Fill in the **Label** and **Tag** fields for the new menu item. For example, set **Label** to `Open` and set **Tag** to `menu_file_open`. Click outside the field for the change to take effect.



You can also

- Choose an alphabetic keyboard accelerator for the menu item with the **Accelerator** pop-up menu. In combination with **Ctrl**, this is the keyboard equivalent for a menu item that does not have a child menu. Note that some accelerators may be used for other purposes on your system and that other actions may result.
- Display a separator above the menu item by checking **Separator above this item**.
- Display a check next to the menu item when the menu is first opened by checking **Check mark this item**. A check indicates the current state of the menu item. See the example in “Adding Items to the Context Menu” on page 6-80.

- Enable this item when the menu is first opened by checking **Enable this item**. This allows the user to select this item when the menu is first opened. If you uncheck this option, the menu item appears dimmed when the menu is first opened, and the user cannot select it.
- Specify a string for the routine, i.e., the **Callback**, that performs the action associated with the menu item. If you have not yet saved the GUI, the default value is `%automatic`. When you save the GUI, and if you have not changed this field, GUIDE automatically sets the value using a combination of the **Tag** field and the GUI filename. See "Menu Item" on page 8-41 for more information about specifying this field and for programming menu items.

  The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the GUI, GUIDE prompts you to save it.
- Open the Property Inspector, where you can change all menu properties, by clicking the **More options** button. For detailed information about the properties, see Uimenu Properties in the MATLAB documentation.

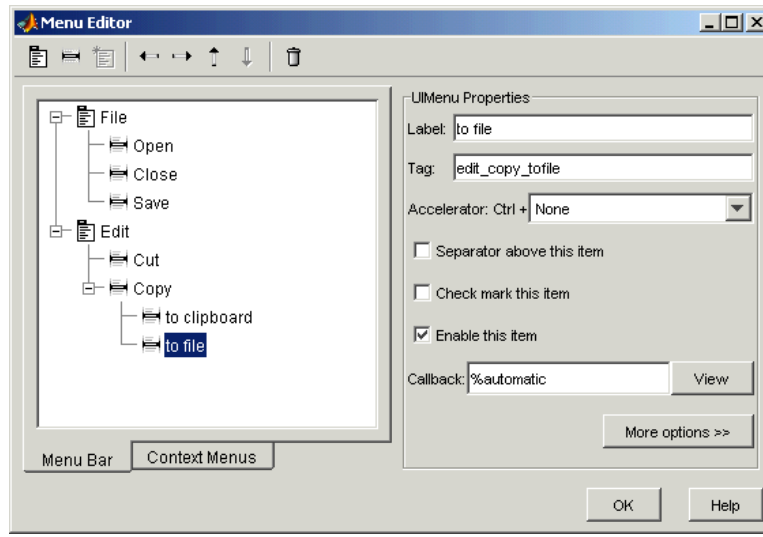
Note In general, programming conventions described for components in Chapter 8, "Programming a GUIDE GUI" also apply to menu items. See "Menu Item" on page 8-41 and "Updating a Menu Item Check" on page 8-42 for programming information and basic examples.

Additional Drop-Down Menus

To create additional drop-down menus, use the New Menu button in the same way you did to create the `File` menu. For example, the following figure also shows an `Edit` drop-down menu.

Cascading Menus

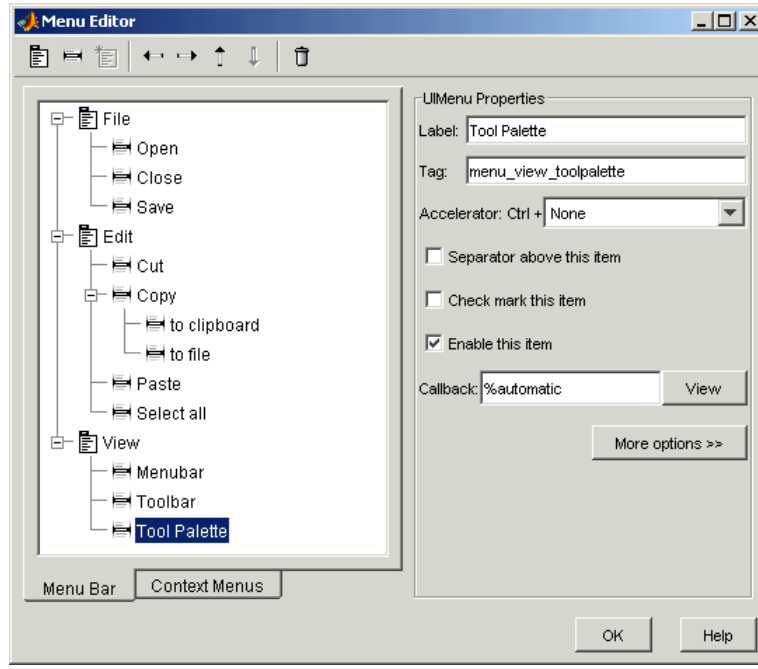
To create a cascading menu, select the menu item that will be the title for the cascading menu, then click the **New Menu Item** button. In the example below, `Copy` is a cascading menu.



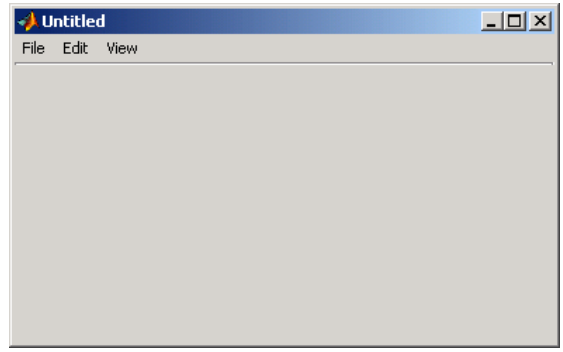
Note See “Menu Item” on page 8-41 for information about programming menu items.

Laying Out Three Menus

The following Menu Editor illustration shows three menus defined for the figure menu bar.



When you run the GUI, the menu titles appear in the menu bar.



Context Menus

A context menu is displayed when a user right-clicks the object for which the menu is defined. The Menu Editor enables you to define context menus and associate them with objects in the layout. The process has three steps:

1 “Creating the Parent Menu” on page 6-79

2 “Adding Items to the Context Menu” on page 6-80

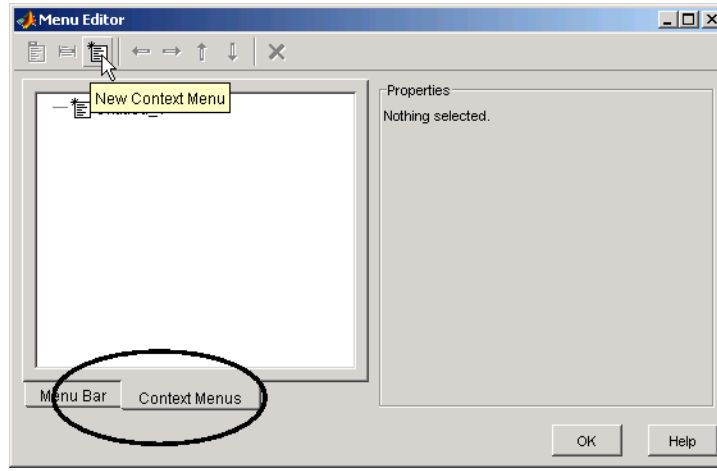
3 “Associating the Context Menu with an Object” on page 6-82

Note See “Menus for the Menu Bar” on page 6-71 for information about defining menus in general. See “Menu Item” on page 8-41 for information about defining callback subfunctions for your menus.

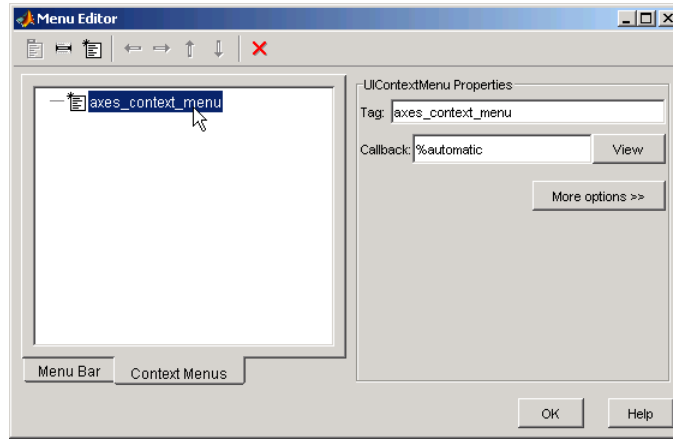
Creating the Parent Menu

All items in a context menu are children of a menu that is not displayed on the figure menu bar. To define the parent menu:

1 Select the Menu Editor’s **Context Menus** tab and select the New Context Menu button from the toolbar.



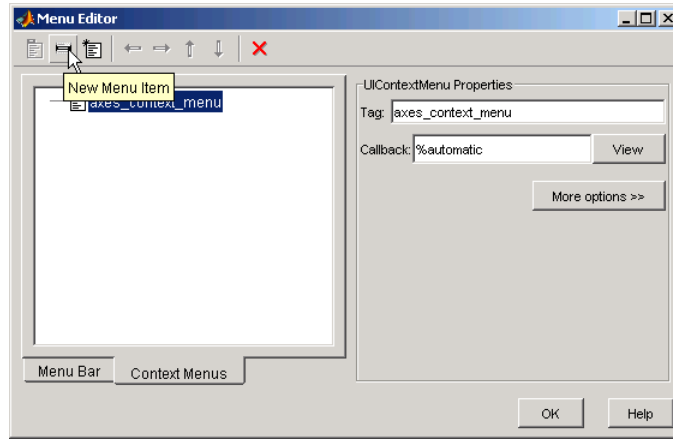
2 Select the menu and specify the **Tag** field to identify the context menu (`axes_context_menu` in this example).



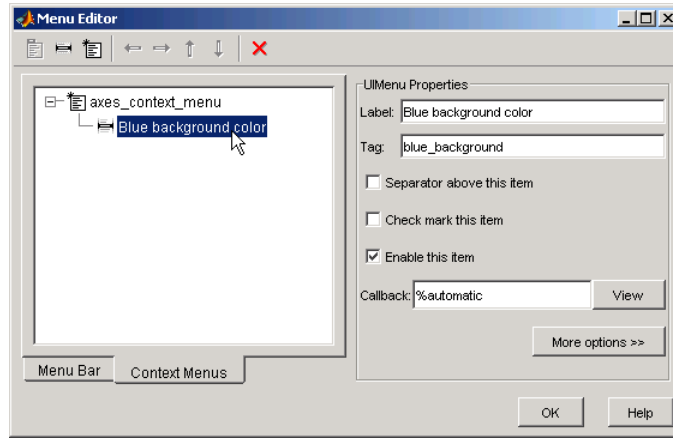
Adding Items to the Context Menu

Use the New Menu Item button to create menu items that are displayed in the context menu.

1 Add a **Blue background color** menu item to the menu by selecting `axes_context_menu` and clicking the **New Menu Item** tool. A temporary numbered menu item label, `Untitled`, appears.



2 Fill in the **Label** and **Tag** fields for the new menu item. For example, set **Label** to `Blue background color` and set **Tag** to `blue_background`. Click outside the field for the change to take effect.



You can also

- Display a separator above the menu item by checking **Separator above this item**.
- Display a check next to the menu item when the menu is first opened by checking **Check mark this item**. A check indicates the current state of

the menu item. See the example in "Adding Items to the Context Menu" on page 6-80. See "Updating a Menu Item Check" on page 8-42 for a code example.

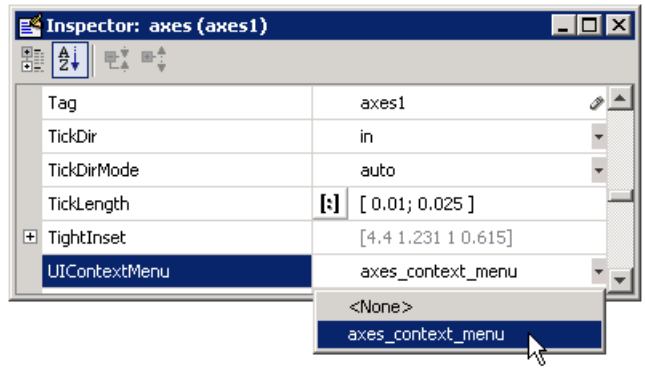
- Enable this item when the menu is first opened by checking **Enable this item**. This allows the user to select this item when the menu is first opened. If you uncheck this option, the menu item appears dimmed when the menu is first opened, and the user cannot select it.
- Specify a string for the routine, i.e., the **Callback**, that performs the action associated with the menu item. If you have not yet saved the GUI, the default value is `%automatic`. When you save the GUI, and if you have not changed this field, GUIDE automatically sets the value using a combination of the **Tag** field and the GUI filename. See "Menu Item" on page 8-41 for more information about specifying this field and for programming menu items.

  The **View** button displays the callback, if there is one, in an editor. If you have not yet saved the GUI, GUIDE prompts you to save it.
- Open the Property Inspector, where you can change all menu properties, by clicking the **More options** button. For detailed information about these properties, see Uicontextmenu Properties in the MATLAB documentation.

Associating the Context Menu with an Object

1. In the Layout Editor, select the object for which you are defining the context menu.
2. Use the Property Inspector to set this object's `UIContextMenu` property to the name of the desired context menu.

The following figure shows the `UIContextMenu` property for the `axes` object with `Tag` property `axes1`.



In the GUI M-file, complete the callback subfunction for each item in the context menu. Each callback executes when a user selects the associated context menu item. See “Menu Item” on page 8-41 for information on defining the syntax.

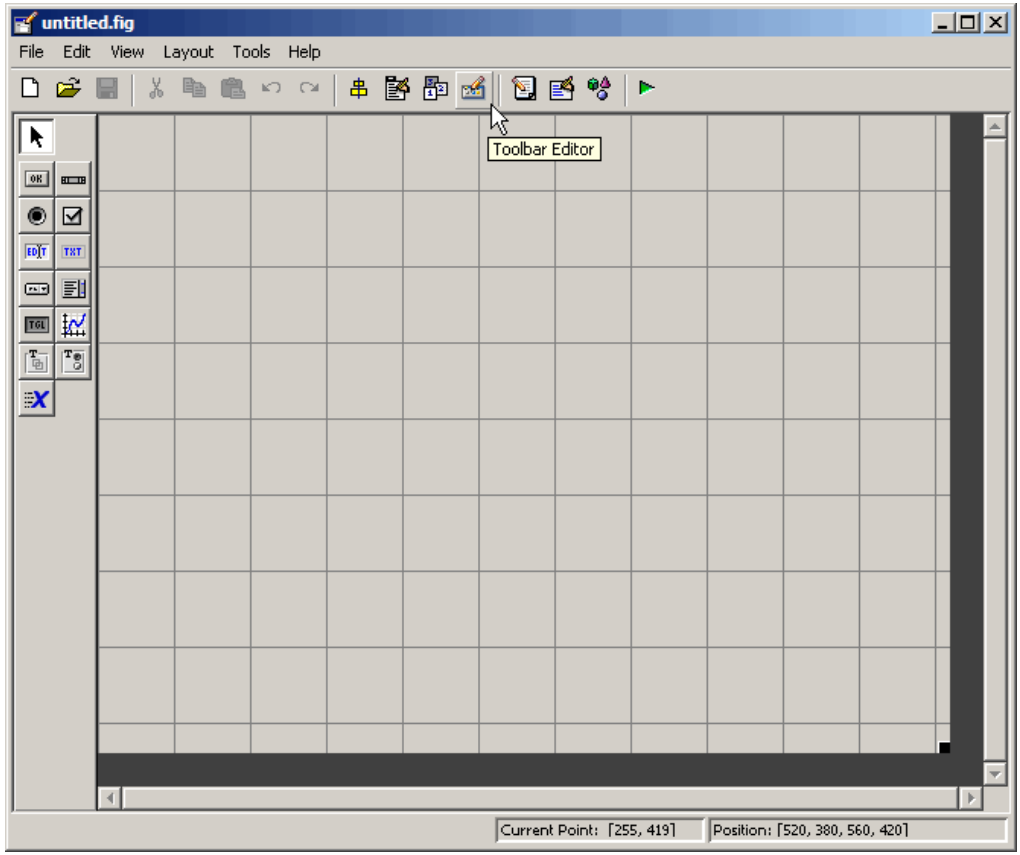
Note In general, programming conventions described for components in Chapter 8, “Programming a GUIDE GUI” also apply to menu items. See “Menu Item” on page 8-41 and “Updating a Menu Item Check” on page 8-42 for programming information and basic examples.

Creating Toolbars

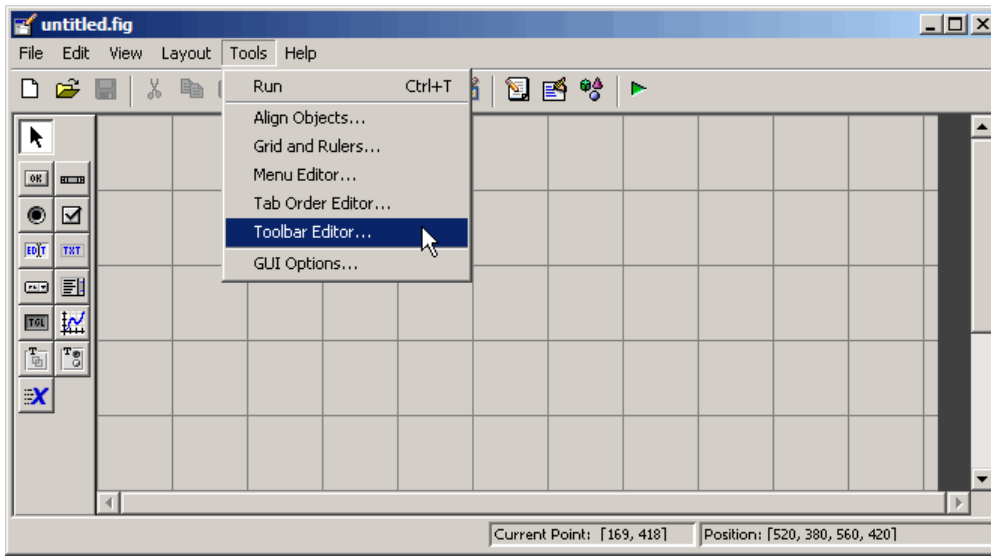
In this section...

Creating Toolbars with GUIDE

You can add a toolbar to a GUI you create in GUIDE with the Toolbar Editor, which you open from the GUIDE Layout Editor toolbar.



You can also open the Toolbar Editor from the Tools menu.



The Toolbar Editor gives you interactive access to all the features of the `uitoolbar`, `uipushtool`, and `uitoggletool` functions. It only operates in the context of GUIDE; you cannot use it to modify any of the built-in MATLAB toolbars. However, you can use the Toolbar Editor to add, modify, and delete a toolbar from any GUI in GUIDE.

Currently, you can add one toolbar to your GUI in GUIDE. However, your GUI can also include the standard MATLAB figure toolbar. If you need to, you can create a toolbar that looks like a normal figure toolbar, but customize its callbacks to make tools (such as pan, zoom, and open) behave in specific ways.

Note You do not need to use the Toolbar Editor if you simply want your GUI to have a standard figure toolbar. You can do this by setting the figure's `Toolbar` property to `'figure'`, as follows:

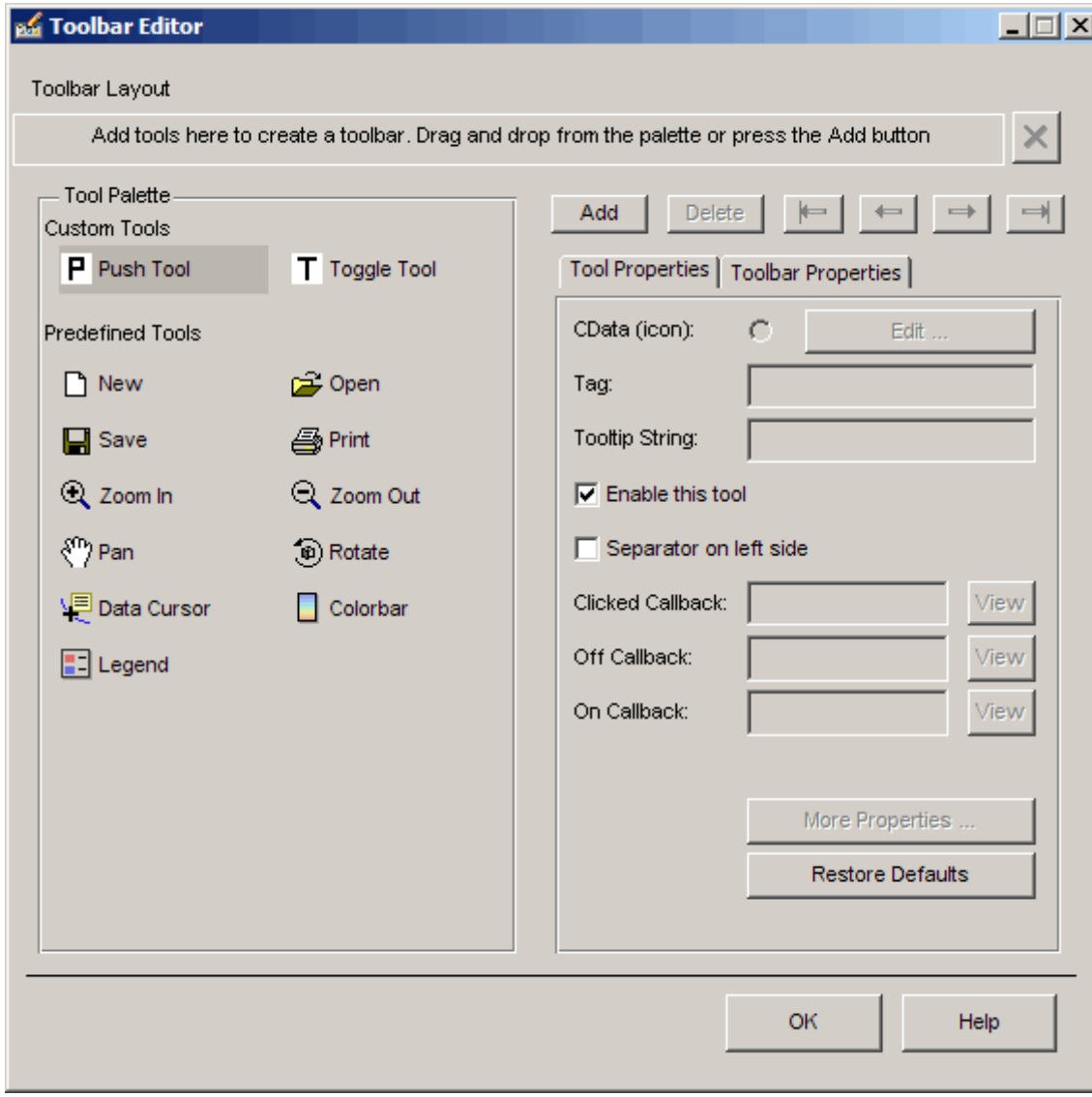
1. Open the GUI in GUIDE.
2. From the **View** menu, open **Property Inspector**.
3. Set the `Toolbar` property to `figure` using the drop-down menu.
4. Save the figure

If you later want to remove the figure toolbar, set the `Toolbar` property to `auto` and resave the GUI. This will not remove or hide your custom toolbar should the GUI have one. See "Creating Toolbars Programmatically" on page 6-98 for more information about creating a toolbar with M-code.

Using the Toolbar Editor

The Toolbar Editor contains three main parts:

- The **Toolbar Layout** preview area on the top
- The **Tool Palette** on the left
- Two tabbed property panes on the right



To add a tool, drag an icon from the **Tool Palette** into the **Toolbar Layout** (which initially contains the text prompt shown above), and edit the tool's properties in the **Tool Properties** pane.

When you first create a GUI, no toolbar exists on it. When you open the Toolbar Editor and place the first tool, a toolbar is created and a preview of the tool you just added appears in the top part of the window. If you later open a GUI that has a toolbar, the Toolbar Editor shows the existing toolbar, although the Layout Editor does not.

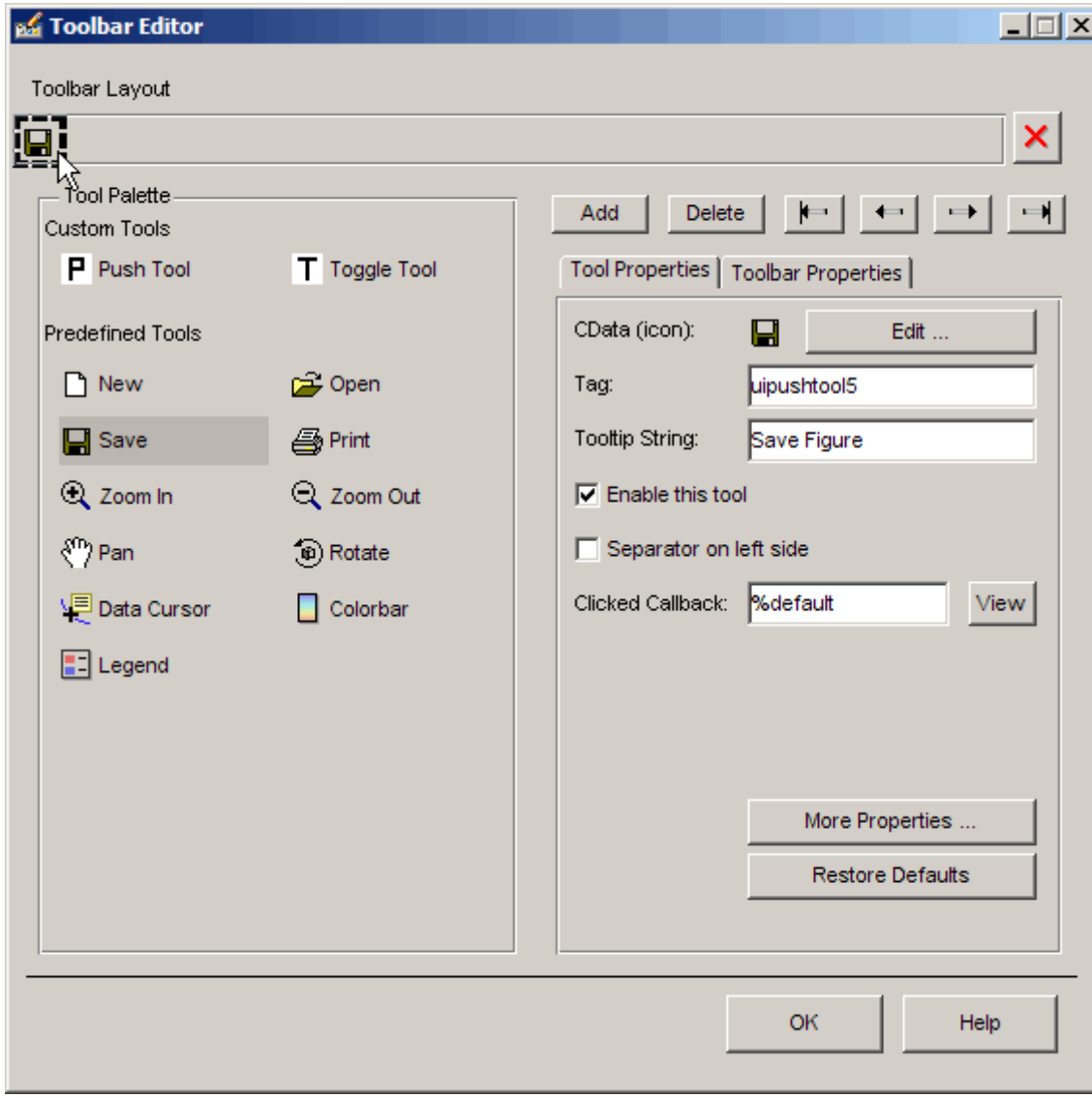
Adding Tools

You can add a tool to a toolbar in three ways:

- Drag and drop tools from the **Tool Palette**.
- Select a tool in the palette and click the **Add** button.
- Double-click a tool in the palette.

Dragging allows you to place a tool in any order on the toolbar. The other two methods place the tool to the right of the right-most tool on the **Toolbar Layout**. The new tool is selected (indicated by a dashed box around it) and its properties are shown in the **Tool Properties** pane. You can select only one tool at a time. You can cycle through the **Tool Palette** using the tab key or arrow keys on your computer keyboard. You must have placed at least one tool on the toolbar.

After you place tools from the **Tool Palette** into the **Toolbar Layout** area, the Toolbar Editor shows the properties of the currently selected tool, as the following illustration shows.



Predefined and Custom Tools

The Toolbar Editor provides two types of tools:

- Predefined tools, having standard icons and behaviors
- Custom tools, having generic icons and no behaviors

Predefined Tools. The set of icons on the bottom of the **Tool Palette** represent standard MATLAB figure tools. Their behavior is built in. Predefined tools that require an axes (such as pan and zoom) do not exhibit any behavior in GUIs lacking axes. The callback(s) defining the behavior of the predefined tool are shown as `%default`, which calls the same function that the tool calls in standard figure toolbars and menus (to open files, save figures, change modes, etc.). You can change `%default` to some other callback to customize the tool; GUIDE warns you that you will modify the behavior of the tool when you change a callback field or click the **View** button next to it, and asks if you want to proceed or not.

Custom Tools. The two icons at the top of the Tool Palette create pushtools and toggletools. These have no built-in behavior except for managing their appearance when clicked on and off. Consequently, you need to provide your own callback(s) when you add one to your toolbar. In order for custom tools to respond to clicks, you need to edit their callbacks to create the behaviors you desire. Do this by clicking the **View** button next to the callback in the **Tool Properties** pane, and then editing the callback in the Editor window.

Adding and Removing Separators

Separators are vertical bars that set off tools, enabling you to group them visually. You can add or remove a separator in any of three ways:

- Right-click on a tool's preview and select **Show Separator**, which toggles its separator on and off.
- Check or clear the checkbox **Separator** to the left in the tool's property pane.
- Change the `Separator` property of the tool from the Property Inspector

After adding a separator, that separator appears in the **Toolbar Layout** to the left of the tool. The separator is not a distinct object or icon; it is a property of the tool.

Moving Tools

You can reorder tools on the toolbar in two ways:

- Drag a tool to a new position.
- Select a tool in the toolbar and click one of the arrow buttons below the right side of the toolbar.

If a tool has a separator to its left, the separator moves with the tool.

Removing Tools

You can remove tools from the toolbar in three ways:

- Select a tool and press the **Delete** key.
- Select a tool and click the **Delete** button on the GUI.
- Right-click a tool and select **Delete** from the context menu.

You cannot undo any of these actions.

Editing a Tool's Properties

You edit the appearance and behavior of the currently selected tool using the **Tool Properties** pane, which includes controls for setting the most commonly used tool properties:

- CData — The tool's icon
- Tag — The internal name for the tool
- Enable — Whether users can click the tool
- Separator — A bar to the left of the icon for setting off and grouping tools
- Clicked Callback — The function called when users click the tool
- Off Callback (uitoggletool only) — The function called when the tool is put in the *off* state
- On Callback (uitoggletool only) — The function called when the tool is put in the *on* state

See “Callbacks: An Overview” on page 8-2 for details on programming the tool callbacks. You can also access these and other properties of the selected tool with the Property Inspector. To open the Property Inspector, clicki the **More Properties** button on the **Tool Properties** pane.

Editing Tool Icons

To edit a selected toolbar icon, click the **Edit** button in the **Tool Properties** pane, next to **CData (icon)** or right-click the **Toolbar Layout** and select **Edit Icon** from the context menu. The Icon Editor opens with the tool's `CData` loaded into it. For information about editing icons, see “Using the Icon Editor” on page 6-95.

Editing Toolbar Properties

If you click an empty part of the toolbar or click the **Toolbar Properties** tab, you can edit two of its properties:

- `Tag` — The internal name for the toolbar
- `Visible` — Whether the toolbar is displayed in your GUI

The `Tag` property is initially set to `uitoolbar1`. The `Visible` property is set to `on`. When on, the `Visible` property causes the toolbar to be displayed on the GUI regardless of the setting of the figure's `Toolbar` property. If you want to toggle a custom toolbar as you can built-in ones (from the **View** menu), you can create a menu item, a checkbox, or other control to control its Visible property.

To access nearly all the properties for the toolbar in the Property Inspector, click **More Properties**.

Testing Your Toolbar

To try out your toolbar, click the **Run** button in the Layout Editor. MATLAB asks if you want to save changes to its `.fig` file first.

Removing a Toolbar

You can remove a toolbar completely—destroying it—from the Toolbar Editor, leaving your GUI without a toolbar (other than the figure toolbar, which is not visible by default). The are two ways to remove a toolbar:

- Click the **Remove** button  on the right end of the toolbar.
- Right-click a blank area on the toolbar and select **Remove Toolbar** from the context menu.

If you remove all the individual tools in the ways shown in “Removing Tools” on page 6-92 without removing the toolbar itself, your GUI will contain an empty toolbar.

Closing the Toolbar Editor

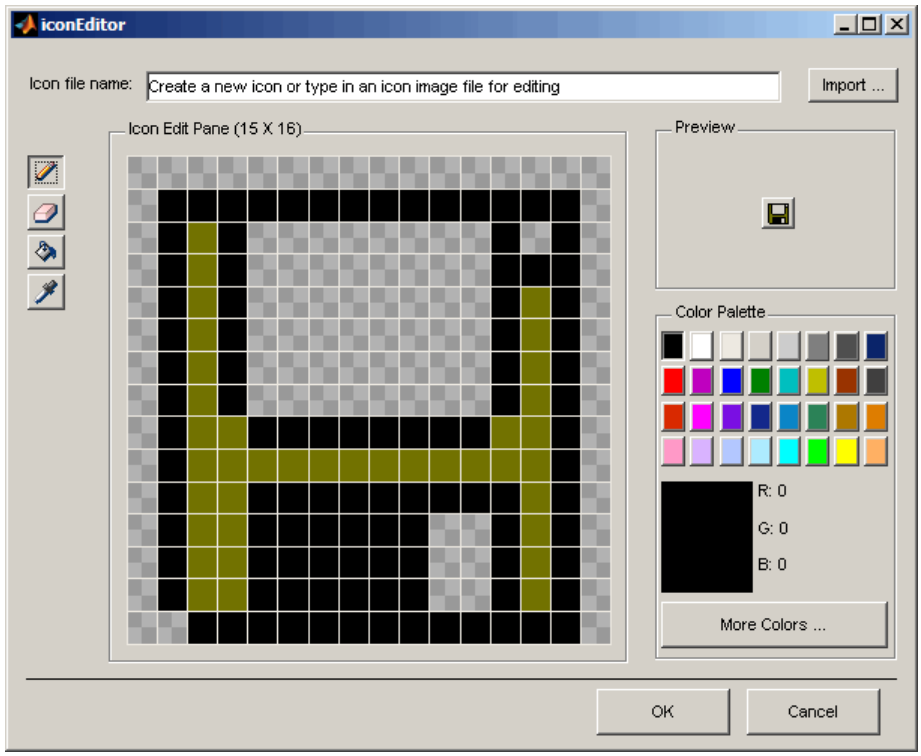
You can close the Toolbar Editor window in two ways:

- Press the **OK** button.
- Click the Close box in the title bar.

When you close the Toolbar Editor, the current state of your toolbar is saved with the GUI you are editing. You do not see the toolbar in the Layout Editor; you need to run the GUI to see or use it.

Editing Tool Icons

GUIDE includes its own Icon Editor, a GUI for creating and modifying icons such as icons on toolbars. You can access this editor only from the Toolbar Editor. This figure shows the Icon Editor loaded with a standard **Save** icon.



Note There are examples that show how to create your own icon editor. See the example in "Icon Editor" on page 15-29 and the discussion of sharing data among multiple GUIs in the portion of the GUI Building documentation.

Using the Icon Editor

The Icon Editor GUI includes the following components:

- **Icon file name** — The icon image file to be loaded for editing
- **Import** button — Opens a file dialog to select an existing icon file for editing
- Drawing tools — A group of four tools on the left side for editing icons

  - Pencil tool — Color icon pixels by clicking or dragging
  - Eraser tool — Erase pixels to be transparent by clicking or dragging
  - Paint bucket tool — Flood regions of same-color pixels with the current color
  - Pick color tool — Click a pixel or color palette swatch to define the current color
- **Icon Edit** pane — A n-by-m grid where you color an icon
- **Preview** pane — A button with a preview of current state of the icon
- **Color Palette** — Swatches of color that the pencil and paint tools can use
- **More Colors** button — Opens the Colors dialog box for choosing and defining colors
- **OK** button — Dismisses the GUI and returns the icon in its current state
- **Cancel** button — Closes the GUI without returning the icon

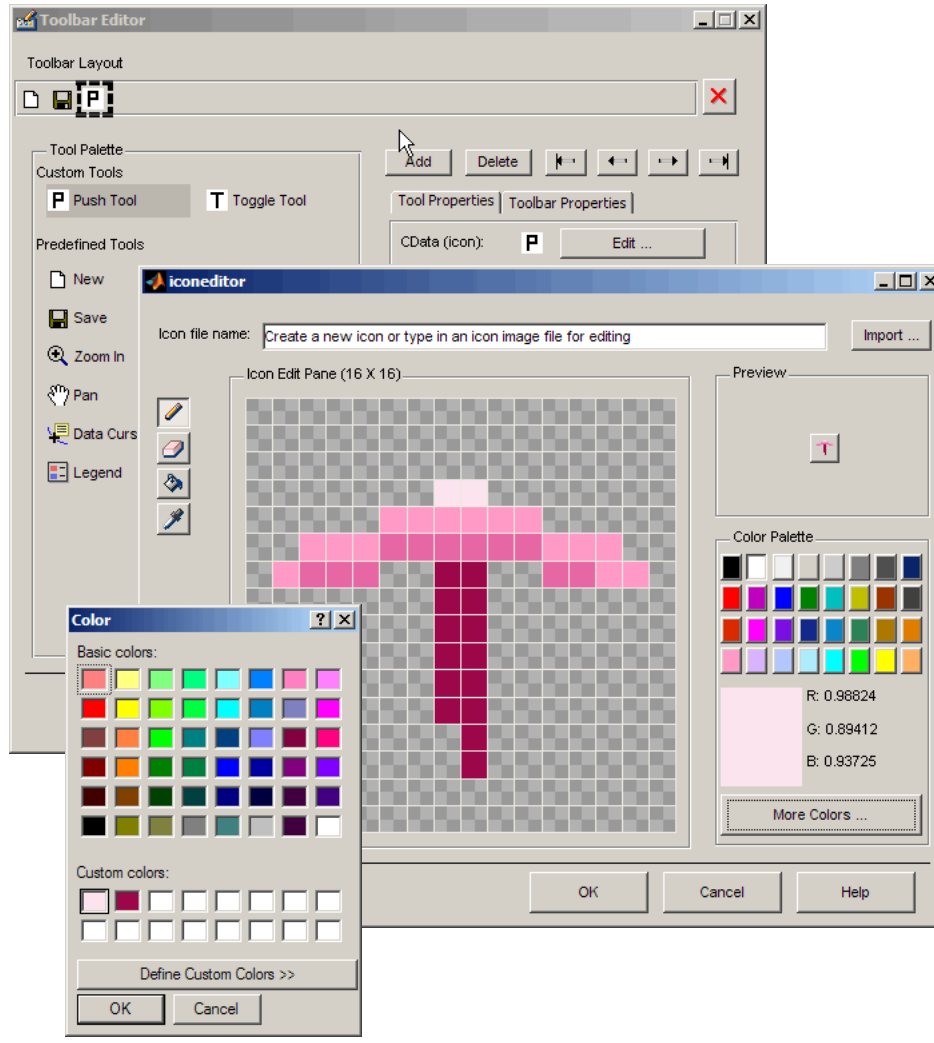
To work with the Icon Editor,

1. Open the Icon Editor for a selected tool's icon.
2. Using the Pencil tool, color the squares in the grid:
   - Click a color cell in the palette.
   - That color appears in the **Color Palette** preview swatch.
   - Click in specific squares of the grid to transfer the selected color to those squares.
   - Hold down the left mouse button and drag the mouse over the grid to transfer the selected color to the squares that you touch.
   - Change a color by writing over it with another color.
3. Using the Eraser tool, erase the color in some squares
   - Click the Eraser button on the palette.
   - Click in specific squares to erase those squares.

- Click and drag the mouse to erase the squares that you touch.
- Click a another drawing tool to disable the Eraser.

4 Click **OK** to close the GUI and return the icon you created or click **Cancel** to close the GUI without modifying the selected tool's icon.

The three GUIs are shown operating together below, before saving a uipushtool icon:



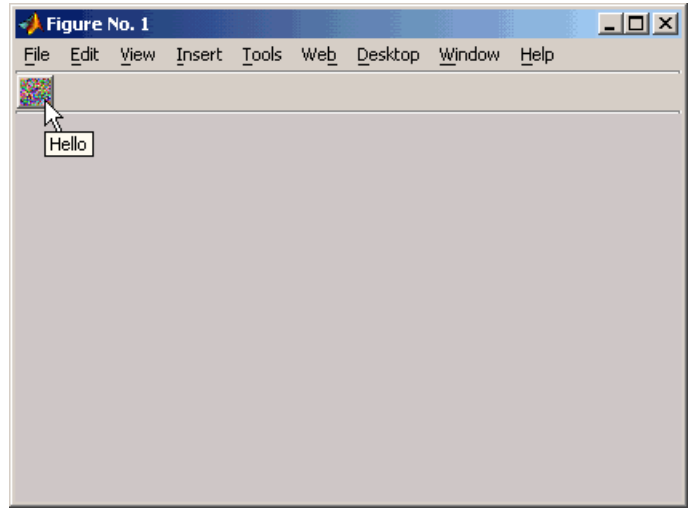
Creating Toolbars Programmatically

As described previously, GUIDE provides tools to enable you to add a toolbar to a GUI and add tools to it. You can also add a toolbar and tools programmatically by adding code to the opening function.

See “Initialization Callbacks” on page 8-16 for information about the opening function, and see the `uitoolbar`, `uipushtool`, and `uitoggletool` reference pages for information and examples.

This example creates a toolbar (`uitoolbar`) and places a toggle tool (`uitoggletool`) on it. Add the following code to the GUI's opening function to produce the toolbar shown:

```
ht = uitoolbar(hObject)
a = rand(16,16,3);
htt = uitoggletool(ht,'CData',a,'TooltipString','Hello')
```

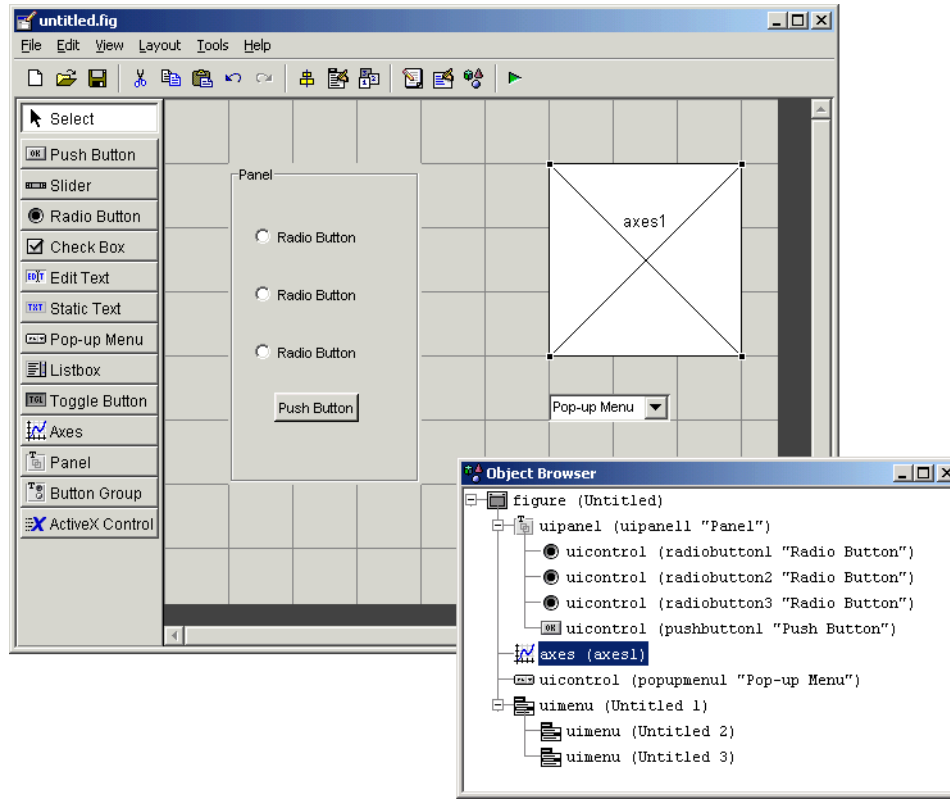


In the opening function, `hObject` is an input argument that holds the figure handle. The `CData` property enables you to display a truecolor image on the toggle tool.

Viewing the Object Hierarchy

The Object Browser displays a hierarchical list of the objects in the figure, including both components and menus. As you lay out your GUI, check the object hierarchy periodically, especially if your GUI contains menus, panes, or button groups.

The following illustration shows a figure object and its child objects. It also shows the child objects of the pane and a menu that was created.



To determine a component's place in the hierarchy, select it in the Layout Editor. It is automatically selected in the Object Browser. Similarly, if you select an object in the Object Browser, it is automatically selected in the Layout Editor.

Designing for Cross-Platform Compatibility

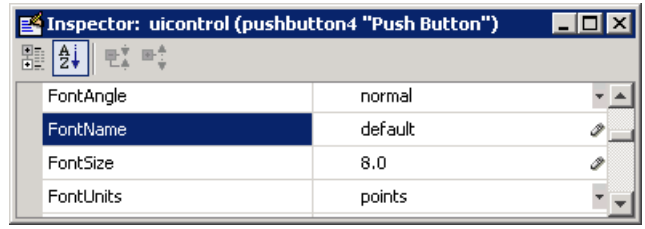
In this section...

Default System Font

By default, user interface controls (uicontrols) use the default font for the platform on which they are running. For example, when displaying your GUI on PCs, uicontrols use MS San Serif. When your GUI runs on a different platform, it uses that computer’s default font. This provides a consistent look with respect to your GUI and other application GUIs.

If you have set the `FontName` property to a named font and want to return to the default value, you can set the property to the string `default`. This ensures that MATLAB uses the system default at run-time.

You can use the Property Inspector to set this property:



Or you can use the `set` command to set the property in the GUI M-file. For example, if there is a push button in your GUI and its handle is stored in the `pushbutton1` field of the `handles` structure, then the statement

```
set(handles.pushbutton1,'FontName','default')
```

sets the `FontName` property to use the system default.

Specifying a Fixed-Width Font

If you want to use a fixed-width font for a user interface control, set its `FontName` property to the string `fixedwidth`. This special identifier ensures that your GUI uses the standard fixed-width font for the target platform.

You can find the name of the fixed-width font that is used on a given platform by querying the root `FixedWidthFontName` property.

```
get(0,'FixedWidthFontName')
```

Using a Specific Font Name

You can specify an actual font name (such as Times or Courier) for the `FontName` property. However, doing so may cause your GUI to not look as you intended when run on a different computer. If the target computer does not have the specified font, it will substitute another font that may not look good in your GUI or may not be the standard font used for GUIs on that system. Also, different versions of the same named font may have different size requirements for a given set of characters.

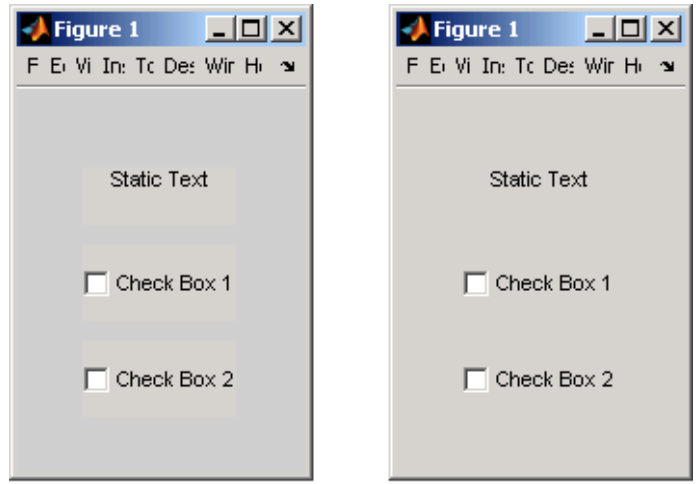
Standard Background Color

The default component background color is the standard system background color on which the GUI is running. This color varies on different computer systems, e.g., the standard shade of gray on the PC differs from that on UNIX, and may not match the default GUI background color.

If you use the default component background color, you can use that same color as the background color for your GUI. This provides a consistent look with respect to your GUI and other application GUIs. To do this in GUIDE, check **Options > Use system color scheme for background** on the Layout Editor **Tools** menu.

Note This option is available only if you first select the **Generate FIG-file and M-File** option.

The following figures illustrate the results with and without system color matching.



Without system color matching

With system color matching

Cross-Platform Compatible Units

Cross-platform compatible GUIs should look correct on computers having different screen sizes and resolutions. Since the size of a pixel can vary on different computer displays, using the default figure `Units` of `pixels` does not produce a GUI that looks the same on all platforms.

For this reason, GUIDE defaults the `Units` property for the figure to `characters`.

System-Dependent Units

Character units are defined by characters from the default system font. The width of a character unit equals the width of the letter `x` in the system font. The height of a character unit is the distance between the baselines of two lines of text. Note that character units are not square.

Units and Resize Behavior

If you set your GUI's resize behavior from the GUI Options dialog box, GUIDE automatically sets the units for the GUI's components in a way that maintains the intended look and feel across platforms. To specify the resize behavior option, select **GUI Options** from the **Tools** menu, then specify **Resize behavior** by selecting **Non-resizable**, **Proportional**, or **Other (Use ResizeFcn)**.

If you choose **Non-resizable**, GUIDE defaults the component units to `characters`. If you choose **Proportional**, it defaults the component units to `normalized`. In either case, these settings enable your GUI to automatically adjust the size and relative spacing of components as the GUI displays on different computers.

If you choose **Other (Use ResizeFcn)**, GUIDE defaults the component units to `characters`. However, you must provide a `ResizeFcn` callback to customize the GUI's resize behavior.

Note GUIDE does not automatically adjust component units if you modify the figure's `Resize` property programmatically or in the Property Inspector.

At times, it may be convenient to use a more familiar unit of measure, e.g., inches or centimeters, when you are laying out the GUI. However, to preserve the look of your GUI on different computers, remember to change the figure `Units` property back to `characters`, and the components' `Units` properties to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) before you save the GUI.

7

Saving and Running a GUIDE GUI

Naming a GUI and Its Files

In this section...

The GUI Files

By default, GUIDE stores a GUI in two files which are generated the first time you save or run the GUI:

- A FIG-file, with extension `.fig`, that contains a complete description of the GUI layout and the GUI components, such as push buttons, axes, panels, menus, and so on. The FIG-file is a binary file and you cannot modify it except by changing the layout in GUIDE. Note that a FIG-file is a kind of MAT-file. See "MAT-Files Preferences" in the MATLAB Desktop Tools and Development Environment documentation for more information.
- An M-file, with extension `.m`, that contains the code that controls the GUI, including the callbacks for its components.

These two files usually reside in the same directory. They correspond to the tasks of laying out and programming the GUI. When you lay out the GUI in the Layout Editor, your work is stored in the FIG-file. When you program the GUI, your work is stored in the corresponding M-file.

Note that if your GUI includes ActiveX components, GUIDE also generates a file for each ActiveX component. See "ActiveX Control" on page 8-33 for more information.

For more information about these files, see "GUI Files: An Overview" on page 8-5.

File and GUI Names

The M-file and the FIG-file that define your GUI must have the same name. This name is also the name of your GUI.

For example, if your files are named `mygui.fig` and `mygui.m`, then the name of the GUI is `mygui`, and you can run the GUI by typing `mygui` at the command line. This assumes that the M-file and FIG-file are in the same directory and that the directory is in your path.

Names are assigned when you save the GUI the first time. See "Ways to Save a GUI" on page 7-4 for information about saving GUIs.

Renaming GUIs and GUI Files

To rename a GUI, rename the GUI FIG-file using **Save As** from the Layout Editor **File** menu. When you do this, GUIDE renames both the FIG-file and the GUI M-file, updates any callback properties that contain the old name to use the new name, and updates all instances of the file name in the body of the M-file.

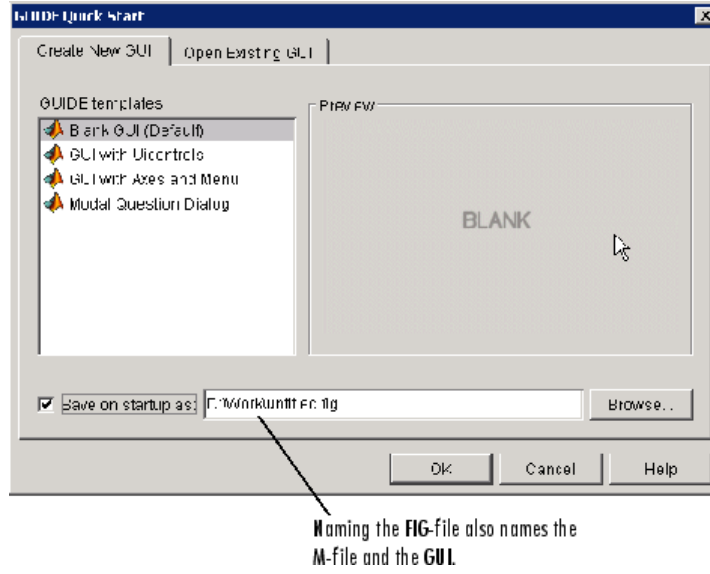
Saving a GUI

In this section...

Ways to Save a GUI

You can save a GUI in GUIDE in any of these ways:

- From the GUIDE Quick Start dialog box. Before you select a template, GUIDE lets you select a name for your GUI. When you click **OK**, GUIDE saves the GUI M-file and FIG-file using the name you specify.



- The first time you save the files by
  - Clicking the Save icon on the Layout Editor toolbar
  - Selecting the **Save** or **Save as** options on the **File** menu

In either case, GUIDE prompts you for a name before saving the GUI.

- The first time you run the GUI by
  - Clicking the Run icon on the Layout Editor toolbar
  - Selecting **Run** from the **Tools** menu

  In each case, GUIDE prompts you for a name and saves the GUI files before activating the GUI.

In all cases, GUIDE creates a template M-file and opens it in your default editor. See “Naming of Callback Functions” on page 8-13 for more information about the template M-file.

Note In most cases you should save your GUI to your current directory or to your path. GUIDE-generated GUIs cannot run correctly from a private directory. GUI FIG-files that are created or modified with MATLAB 7.0 or a later MATLAB version, are not automatically compatible with Version 6.5 and earlier versions. To make a FIG-file, which is a kind of MAT-file, backward compatible, you must check **General > MAT-Files > Ensure backward compatibility (-v6)** in the MATLAB **Preferences** dialog box before saving the file. Button groups and panels are introduced in MATLAB 7.0, and you should not use them in GUIs that you expect to run in earlier MATLAB versions.

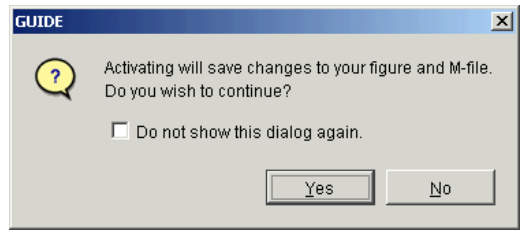
Saving a New GUI

Follow these steps if you are saving a GUI for the first time, or if you are using **Save as** from the **File** menu.

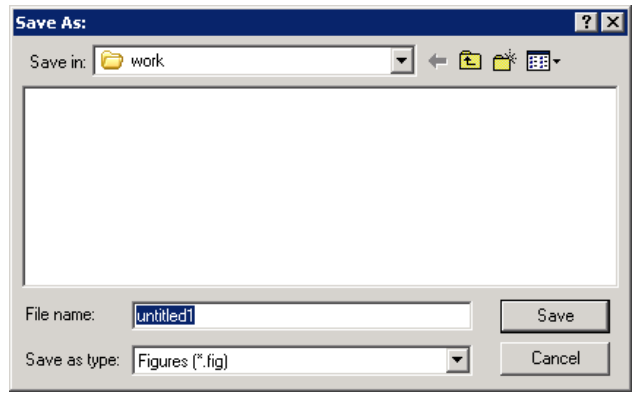
Note If you select **Save as** from the **File** menu or click the **Save** button on the toolbar, GUIDE saves the GUI without activating it. However, if you select **Run** from the **Tools** menu or click the **Run** icon on the toolbar, GUIDE saves the GUI before activating it.

---

1 If you have made changes to the GUI and elect to activate the GUI by selecting **Run** from the **Tools** menu or by clicking the **Run** icon on the toolbar, GUIDE displays the following dialog box. Click **Yes** to continue.



2 If you clicked **Yes** in the previous step, if you are saving the GUI without activating it, or if you are using **Save as** from the File menu, GUIDE opens a **Save As** dialog box and prompts you for a FIG-file name.

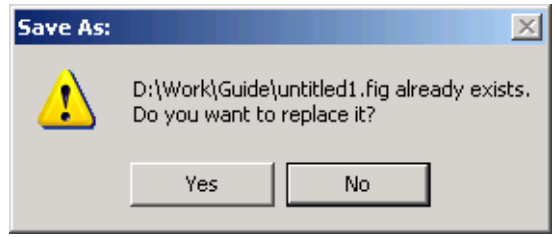


3 Change the directory if you choose, and then enter the name you want to use for the FIG-file. Be sure to choose a writable directory. GUIDE saves both the FIG-file and the M-file using this name.

Note In most cases you should save your GUI to your current directory or to your path. GUIDE-generated GUIs cannot run correctly from a private directory.

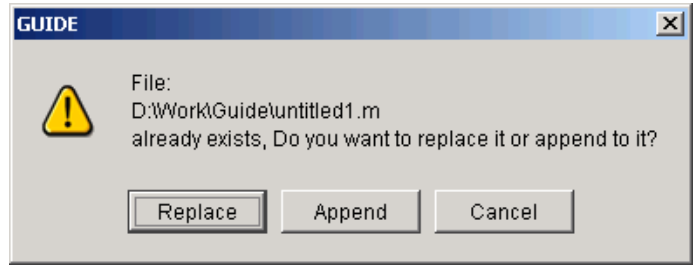
---

4 If you choose an existing filename, GUIDE displays a dialog box that asks you if you want to replace the existing FIG-file. Click **Yes** to continue.

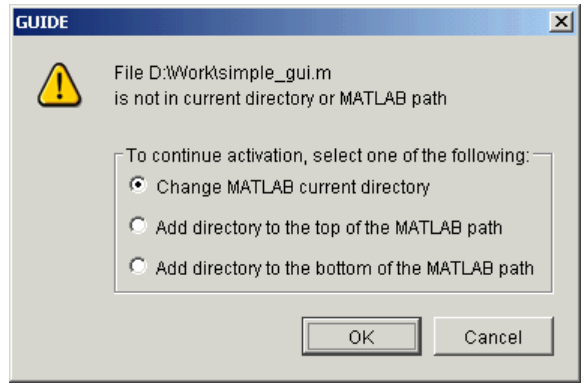


5 If you chose **Yes** in the previous step, GUIDE displays a dialog that asks if you want to replace the existing M-file or append to it. The most common choice is **Replace**.

If you choose **Append**, GUIDE adds callbacks to the existing M-file for components in the current layout that are not present in the existing M-file. Before you append the new components, ensure that their `Tag` properties do not duplicate `Tag` values that appear in callback function names in the existing M-file. See “Assigning an Identifier to Each Component” on page 6-27 for information about specifying the `Tag` property. See “Naming of Callback Functions” on page 8-13 for more information about callback function names.



6 If you chose to activate the GUI by selecting **Run** from the **Tools** menu or by clicking the **Run** button on the toolbar, and if the directory in which you save the GUI is not on the MATLAB path, GUIDE opens a dialog box, giving you the option of changing the current working directory to the directory containing the GUI files, or adding that directory to the top or bottom of the MATLAB path.



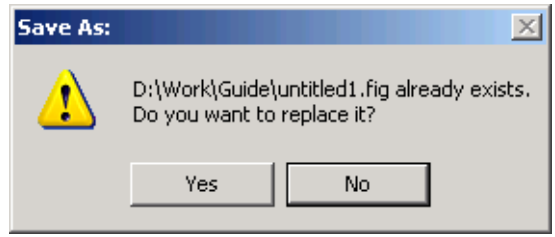
7 After you save the files, GUIDE opens the GUI M-file in your default editor. If you elected to run the GUI, it also activates the GUI.

Saving an Existing GUI

Follow these steps if you are saving an existing GUI to its current location. See "Saving a New GUI" on page 7-5 if you are using **Save as** from the **File** menu.

If you have made changes to a GUI and choose to save and activate the GUI by selecting **Run** from the **Tools** menu or by clicking the Run button on the toolbar, GUIDE saves the GUI and then activates it. It does not automatically open the M-file, even if you added new components.

If you select **Save** from the **File** menu or click the Save button on the toolbar, GUIDE saves the GUI without activating it.



Running a GUI

In this section...

Executing the M-file

Generally, you run your GUI by executing the M-file that GUIDE generates. This M-file contains the commands to load the GUI and provides a framework for the component callbacks. See "GUI Files: An Overview" on page 8-5 for more information about the M-file.

When you execute the M-file, a fully functional copy of the GUI displays on the screen. You can run a GUI:

Note You can display a copy of the GUI figure using the `openfig`, `open`, or `hgload` function. These commands load FIG-files into the MATLAB workspace. The displayed GUI is active, and you can manipulate the components. But nothing happens. This is because no corresponding M-file has been executed.

From the GUIDE Layout Editor

Run your GUI from the GUIDE Layout Editor by:

- Clicking the ▶ button on the Layout Editor toolbar
- Selecting **Run** from the **Tools** menu

In either case, if the GUI has changed or has never been saved, GUIDE saves the GUI files before activating it and opens the GUI M-file in your default editor. See "Saving a GUI" on page 7-4 for information about this process. See "GUI Files: An Overview" on page 8-5 for more information about GUI M-files.

From the Command Line

Run your GUI from its M-file by executing the GUI M-file. For example, if your GUI M-file is `mygui.m`, type

```
mygui
```

at the command line. The files must reside on your path or in your current directory.

If a GUI accepts arguments when it is run, they are passed to the GUI's opening function. See "Opening Function" on page 8-16 for more information.

Note Consider whether you want to allow more than one copy of the GUI to be active at the same time. If you want only one GUI to be active, select **Options > GUI Allows Only One Instance to Run (Singleton)** from the Layout Editor **View** menu. See "GUI Options" on page 5-9 for more information.

From an M-file

Run your GUI from an M-file by executing the GUI M-file. For example, if your GUI M-file is `mygui.m`, include the following statement in your M-file script.

```
mygui
```

The M-file must reside on the MATLAB path or in the current MATLAB directory where the GUI is run.

If a GUI accepts arguments when it is run, they are passed to the GUI's opening function. See "Opening Function" on page 8-16 for more information.

Note Consider whether you want to allow more than one copy of the GUI to be active at the same time. If you want only one GUI to be active, select **Options** from the Layout Editor **View** menu, then select **GUI Allows Only One Instance to Run (Singleton)**. See "GUI Options" on page 5-9 for more information.

8

Programming a GUIDE GUI

Callbacks: An Overview

In this section...

Programming of GUIs Created Using GUIDE

After you have laid out your GUI, you need to program its behavior. The code you write controls how the GUI responds to events such as button clicks, slider movement, menu item selection, or the creation and deletion of components. This programming takes the form of a set of functions, called callbacks, for each component and for the GUI figure itself.

What Is a Callback?

A callback is a function that you write and associate with a specific GUI component or with the GUI figure. It controls GUI or component behavior by performing some action in response to an event for its component. This kind of programming is often called event-driven programming.

When an event occurs for a component, MATLAB invokes the component's callback that is triggered by that event. As an example, suppose a GUI has a button that triggers the plotting of some data. When the user clicks the button, MATLAB calls the callback you associated with clicking that button, and the callback, which you have programmed, then gets the data and plots it.

A component can be any control device such as a push button, list box, or slider. For purposes of programming, it can also be a menu or a container such as a panel or button group. See "Available Components" on page 6-19 for a list and descriptions of components.

Kinds of Callbacks

The GUI figure and each type of component has specific kinds of callbacks with which it can be associated. The callbacks that are available for each component are defined as properties of that component. For example, a push

button has five callback properties: `ButtonDownFcn`, `Callback`, `CreateFcn`, `DeleteFcn`, and `KeyPressFcn`. A panel has four callback properties: `ButtonDownFcn`, `CreateFcn`, `DeleteFcn`, and `ResizeFcn`. You can, but are not required to, create a callback function for each of these properties. The GUI itself, which is a figure, also has certain kinds of callbacks with which it can be associated.

Each kind of callback has a triggering mechanism or event that causes it to be called. The following table lists the callback properties that GUIDE makes available, their triggering events, and the components to which they apply.

Callback Property	Triggering Event	Components
`ButtonDownFcn`	Executes when the user presses a mouse button while the pointer is on or within five pixels of a component or figure. If the component is a user interface control, its `Enable` property must be `on`.	Axes, figure, button group, panel, user interface controls
`Callback`	Component action. Executes, for example, when a user clicks a push button or selects a menu item.	Context menu, menu, user interface controls
`CloseRequestFcn`	Executes before the figure closes.	Figure
`CreateFcn`	Component creation. It can be use to initialize the component when it is created. It executes after the component or figure is created, but before it is displayed.	Axes, figure, button group, context menu, menu, panel, user interface controls
`DeleteFcn`	Component deletion. It can be used to perform cleanup operations just before the component or figure is destroyed.	Axes, figure, button group, context menu, menu, panel, user interface controls
`KeyPressFcn`	Executes when the user presses a keyboard key and the callback's component or figure has focus.	Figure, user interface controls
`KeyReleaseFcn`	Executes when the user releases a keyboard key and the figure has focus.	Figure

Callback Property	Triggering Event	Components
`ResizeFcn`	Executes when a user resizes a panel, button group, or figure whose figure `Resize` property is set to `On`.	Button group, figure, panel
`SelectionChangeFcn`	Executes when a user selects a different radio button or toggle button in a button group component.	Button group
`WindowButtonDownFcn`	Executes when you press a mouse button while the pointer is in the figure window.	Figure
`WindowButtonMotionFcn`	Executes when you move the pointer within the figure window.	Figure
`WindowButtonUpFcn`	Executes when you release a mouse button.	Figure
`WindowScrollWheelFcn`	Executes when the mouse wheel is scrolled while the figure has focus.	Figure

Note User interface controls include push buttons, sliders, radio buttons, check boxes, editable text boxes, static text boxes, list boxes, and toggle buttons. They are sometimes referred to as uicontrols.

Check the properties reference page for your component, e.g., `UicontrolProperties`, to get specific information for a given callback property.

GUI Files: An Overview

In this section...

M-Files and FIG-Files

By default, the first time you save or run a GUI, GUIDE stores the GUI in two files:

- A FIG-file, with extension `.fig`, that contains a complete description of the GUI layout and the GUI components, such as push buttons, axes, panels, menus, and so on. The FIG-file is a binary file and you cannot modify it except by changing the layout in GUIDE. Note that a FIG-file is a kind of MAT-file. See "MAT-Files Preferences" for more information.
- An M-file, with extension `.m`, that initially contains initialization code and templates for some callbacks that are needed to control GUI behavior. You must add the callbacks you write for your GUI components to this file.

  When you save your GUI the first time, GUIDE automatically opens the M-file in your default editor.

The FIG-file and the M-file, usually reside in the same directory. They correspond to the tasks of laying out and programming the GUI. When you lay out the GUI in the Layout Editor, your work is stored in the FIG-file. When you program the GUI, your work is stored in the corresponding M-file.

If your GUI includes ActiveX components, GUIDE also generates a file for each ActiveX component. See "ActiveX Control" on page 8-33 for more information.

For more information about naming and saving a GUI, see Chapter 7, "Saving and Running a GUIDE GUI". If you want to change the name of your GUI and its files, see "Renaming GUIs and GUI Files" on page 7-3.

GUI M-File Structure

The GUI M-file that GUIDE generates is a function file. The name of the main function is the same as the name of the M-file. For example, if the name of the M-file is `mygui.m`, then the name of the main function is `mygui`. Each callback in the file is a subfunction of the main function.

When GUIDE generates an M-file, it automatically includes templates for the most commonly used callbacks for each component. The M-file also contains initialization code, as well as an opening function callback and an output function callback. You must add code to the component callbacks for your GUI to work as you want. You may also want to add code to the opening function callback and the output function callback. The major sections of the GUI M-file are ordered as shown in the following table.

Section	Description
Comments	Displayed at the command line in response to the `help` command. Edit these as necessary for your GUI.
Initialization	GUIDE initialization tasks. *Do not edit this code.*
Opening function	Performs your initialization tasks before the user has access to the GUI.
Output function	Returns outputs to the MATLAB command line after the opening function returns control and before control returns to the command line.
Component and figure callbacks	Control the behavior of the GUI figure and of individual components. MATLAB calls a callback in response to a particular event for a component or for the figure itself.
Utility/helper functions	Perform miscellaneous functions not directly associated with an event for the figure or a component.

Adding Callback Templates to an Existing GUI M-File

When you save the GUI, GUIDE automatically adds templates for some callbacks to the M-file. However, you may want to add other callbacks to the M-file.

Within GUIDE, you can add a callback subfunction template to the GUI M-file in one of two ways. With the component selected for which you want to add the callback:

- Click the right mouse button to display the Layout Editor context menu. Select the desired callback from the **View callbacks** submenu. GUIDE adds the callback template to the GUI M-file and opens the M-file for editing at the callback it just added.
- In the **View** menu, select the desired callback from the **View callbacks** submenu. GUIDE adds the callback template to the GUI M-file and opens the M-file for editing at the callback you just added.

Note In either case, if you select a callback that already exists in the GUI M-file, GUIDE adds no callback, but opens the M-file for editing at the callback you select.

For more information, see “Associating Callbacks with Components” on page 8-8.

Associating Callbacks with Components

In this section...

GUI Components

A GUI can have many components and GUIDE provides a way of specifying which callback should run in response to a particular event for a particular component. The callback that runs when the user clicks a **Yes** button is not the one that runs for the **No** button. Similarly, each menu item usually performs a different function.

GUIDE uses each component's callback properties to associate specific callbacks with that component.

Note "Kinds of Callbacks" on page 8-2 provides a list of callback properties and the components to which each applies.

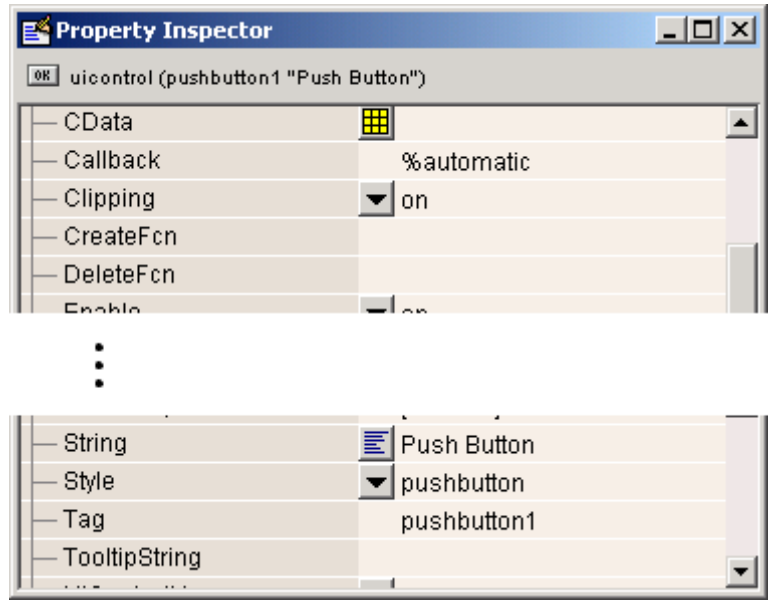
Setting Callback Properties Automatically

GUIDE initially sets the value of the most commonly used callback properties for each component to `%automatic`. For example, a push button has five callback properties, `ButtonDownFcn`, `Callback`, `CreateFcn`, `DeleteFcn`, and `KeyPressFcn`. GUIDE sets only the `Callback` property, the most commonly used callback, to `%automatic`. You can use the Property Inspector to set the other callback properties to `%automatic`.

When you next save the GUI, GUIDE replaces `%automatic` with a MATLAB expression that is the GUI calling sequence for the callback. Within the calling sequence, it constructs the callback name, i.e., the subfunction name, from the component's `Tag` property and the name of the callback property.

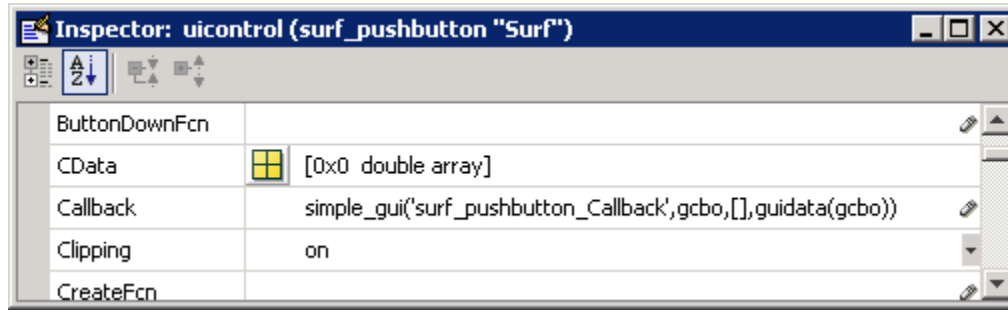
The following figure shows an example of a push button's `Callback` and `Tag` properties in the GUIDE Property Inspector before the GUI is saved.

Note If you change the string `%automatic` before saving the GUI, GUIDE does not automatically add a callback for that component or menu item.



When you save the GUI, GUIDE constructs the name of the callback by appending an underscore (_) and the name of the callback property to the value of the component's `Tag` property. For example, the MATLAB expression for the `Callback` property for a push button in the GUI `simple_gui` with `Tag` property `pushbutton1` is

```
simple_gui(pushbutton1_Callback,gcbo,[],guidata(gcbo))
```



`simple_gui` is the name of the GUI M-file as well as the name of the main function for that GUI. The remaining arguments generate input arguments for `pushbutton1_Callback`. Specifically,

- `gcbo` is a command that returns the handle of the callback object (i.e., `pushbutton1`).
- `[]` is a place holder for the currently unused `eventdata` argument.
- `guidata(gcbo)` returns the `handles` structure for this GUI.

See “Input Arguments” on page 8-14 for information about the callback input arguments.

When you save the GUI, GUIDE also opens the GUI M-file in your editor. The M-file then contains a template for the `Callback` callback for the component whose `Tag` is `pushbutton1`. If you activate the GUI, clicking the push button triggers the execution of the `Callback` callback for the component.

For information about changing the callback name after GUIDE assigns it, see “Changing Callback Names Assigned by GUIDE” on page 8-13. For information about adding callback templates to the GUI M-file, see “Adding Callback Templates to an Existing GUI M-File” on page 8-6.

The next topic, “Callback Syntax and Arguments” on page 8-12, provides more information about the callback template.

Deleting Callbacks from a GUI M-File

There are times when you want to delete a callback from a GUI M-file. The callback may have been automatically generated or you may have added it yourself. Some common reasons for wanting to delete a callback are:

- You have deleted the component for which the callback was generated.
- You want the component to use other code and you have already edited the appropriate callback property in the Property Inspector to point to the other code.

To delete a callback, whether it is automatically generated or whether you added it explicitly, you must first ensure that the callback is not used. Only then should you delete the callback.

To ensure that the callback is not used elsewhere in the GUI:

- Search for occurrences of the name of the callback in the GUI M-file.
- Open the GUI in GUIDE and use the Property Inspector to check for the name of the callback you want to delete in the callback properties of all the components.

In either case, if you find a reference to the callback, you must either remove the reference or retain the callback. Once you have assured yourself that the code is not used by the GUI, manually delete the entire callback subfunction from the M-file.

Callback Syntax and Arguments

In this section...

Callback Templates

GUIDE defines conventions for callback syntax and arguments and implements these conventions in the callback templates it adds to the M-file. Each template is similar to this one for the `Callback` subfunction for a push button.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

...
```

The first comment line describes the event that triggers execution of the callback. This is followed by the function definition line. The remaining comments describe the input arguments.

Insert your code after the last comment.

Note You can avoid automatic generation of the callback comment lines for new callbacks. In the Preferences dialog box, select **GUIDE** and uncheck **Add comments for newly generated callback functions**.

Naming of Callback Functions

The previous callback example shows the following function definition:

```
function pushbutton1_Callback(hObject,eventdata,handles)
```

When GUIDE generates the template, it creates the callback name by appending an underscore (_) and the name of the callback property to the component's `Tag` property. In the example above, `pushbutton1` is the `Tag` property for the push button, and `Callback` is one of the push button's callback properties. The `Tag` property uniquely identifies a component within the GUI.

The first time you save the GUI after adding a component, GUIDE adds callbacks for that component to the M-file and generates the callback names using the current value of the `Tag` property. If you want to change the default `Tag` value, you should do it before you save the GUI.

See "Associating Callbacks with Components" on page 8-8 for more information.

Changing Callback Names Assigned by GUIDE

You can change callback names assigned by GUIDE in either of the following ways:

Note If possible, change callback names for a component immediately after you add the component to the layout and before you save the GUI.

Changing the Tag Property

You can change `Tag` properties to give a component's callbacks more meaningful names, e.g., you might change the `Tag` property from `pushbutton1` to `closebutton`. If possible, change the `Tag` property before saving the GUI, then GUIDE automatically uses the new value when it names the callbacks. However, if you change the `Tag` property after saving the GUI,

GUIDE updates the following items according to the new `Tag`, provided that all components have distinct tags:

- The component's callback functions in the M-file
- The value of the component's callback properties, which you can view in the Property Inspector
- References in the M-file to the field of the `handles` structure that contains the component's handle. See "handles Structure" on page 8-15 for more information about the `handles` structure.

Changing the Callback Property

To rename a particular callback subfunction without changing the `Tag` property,

- Replace the name string in the callback property with the new name. For example, if the value of the callback property for a push button in `mygui` is

  ```
  mygui('pushbutton1_Callback',gcbo,[],guidata(gcbo))
  ```

  the string `pushbutton1_Callback` is the name of the callback function. Change the name to the desired name, for example, `closebutton_Callback`.
- As necessary, update instances of the callback function name in the M-file.

Input Arguments

All callbacks in the GUI M-file have the following input arguments:

- `hObject` — Handle of the object, e.g., the GUI component, for which the callback was triggered. For a button group `SelectionChangeFcn` callback, `hObject` is the handle of the selected radio button or toggle button.
- `eventdata` — Reserved for later use.
- `handles` — Structure that contains the handles of all the objects in the figure. It may also contain application-defined data. See "handles Structure" on page 8-15 for information about this structure.

handles Structure

GUIDE creates a `handles` structure that contains the handles of all the objects in the figure. For a GUI that contains an edit text, a panel, a pop-up menu, and a push button, the `handles` structure originally looks similar to this. GUIDE uses each component's `Tag` property to name the structure element for its handle.

```
handles =
        figure1: 160.0011
          edit1: 9.0020
         uipanel1: 8.0017
      popupmenu1: 7.0018
     pushbutton1: 161.0011
          output: 160.0011
```

GUIDE creates and maintains the `handles` structure as GUI data. It is passed as an input argument to all callbacks and enables a GUI's callbacks to share property values and application data.

For information about GUI data, see "Mechanisms for Managing Data" on page 9-2 and the `guidata` reference page.

For information about adding fields to the `handles` structure and instructions for correctly saving the structure, see Chapter 13, "Managing Application-Defined Data".

Initialization Callbacks

In this section...

Opening Function

The opening function is the first callback in every GUI M-file. It is executed just before the GUI is made visible to the user, but after all the components have been created, i.e., after the components' `CreateFcn` callbacks, if any, have been run.

You can use the opening function to perform your initialization tasks before the user has access to the GUI. For example, you can use it to create data or to read data from an external source. GUI command-line arguments are passed to the opening function.

Function Naming and Template

GUIDE names the opening function by appending `_OpeningFcn` to the name of the M-file. This is an example of an opening function template as it might appear in the `mygui` M-file.

```
% --- Executes just before mygui is made visible.
function mygui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to mygui (see VARARGIN)

% Choose default command line output for mygui
handles.output = hObject;
```

```
% Update handles structure
guidata(hObject, handles);

% UIWAIT makes mygui wait for user response (see UIRESUME)
% uiwait(handles.mygui);
```

Input Arguments

The opening function has four input arguments `hObject`, `eventdata`, `handles`, and `varargin`. The first three are the same as described in “Input Arguments” on page 8-14. the last argument, `varargin`, enables you to pass arguments from the command line to the opening function. The opening function can make such arguments available to the callbacks by adding them to the `handles` structure.

For more information about `varargin`, see the `varargin` reference page and“Passing Variable Numbers of Arguments” in the MATLAB Programming documentation.

All command-line arguments are passed to the opening function via `varargin`. If you open the GUI with a property name/property value pair as arguments, the GUI opens with the property set to the specified value. For example, `my_gui('Position', [71.8 44.9 74.8 19.7])` opens the GUI at the specified position, since `Position` is a valid figure property.

If the input argument is not a valid figure property, you must add code to the opening function to make use of the argument. For an example, look at the opening function for the **Modal Question Dialog** GUI template, available from the GUIDE Quick Start dialog box. The added code enables you to open the modal dialog with the syntax

```
mygui('String','Do you want to exit?')
```

which displays the text `'Do you want to exit?'` on the GUI. In this case, it is necessary to add code to the opening function because `'String'` is not a valid figure property.

Initial Template Code

Initially, the input function template contains these lines of code:

- `handles.output = hObject` adds a new element, `output`, to the `handles` structure and assigns it the value of the input argument `hObject`, which is the handle of the figure, i.e., the handle of the GUI. This handle is used later by the output function. For more information about the output function, see "Output Function" on page 8-18.
- `guidata(hObject,handles)` saves the `handles` structure. You must use `guidata` to save any changes that you make to the `handles` structure. It is not sufficient just to set the value of a `handles` field. See "handles Structure" on page 8-15 and "GUI Data" on page 9-2 for more information.
- `uiwait(handles.mygui)`, initially commented out, blocks GUI execution until `uiresume` is called or the GUI is deleted. Note that `uiwait` allows the user access to other MATLAB windows. Remove the comment symbol for this statement if you want the GUI to be blocking when it opens.

Output Function

The output function returns, to the command line, outputs that are generated during its execution. It is executed when the opening function returns control and before control returns to the command line. This means that you must generate the outputs in the opening function, or call `uiwait` in the opening function to pause its execution while other callbacks generate outputs.

Function Naming and Template

GUIDE names the output function by appending `_OutputFcn` to the name of the M-file. This is an example of an output function template as it might appear in the `mygui` M-file.

```
% --- Outputs from this function are returned to the command line.
function varargout = mygui_OutputFcn(hObject, eventdata,...
                     handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% Get default command line output from handles structure
varargout{1} = handles.output;
```

Input Arguments

The output function has three input arguments: `hObject`, `eventdata`, and `handles`. They are the same as described in "Input Arguments" on page 8-14.

Output Arguments

The output function has one output argument, `varargout`, which it returns to the command line. By default, the output function assigns `handles.output` to `varargout`. So the default output is the handle to the GUI, which was assigned to `handles.output` in the opening function.

You can change the output by

- Changing the value of `handles.output`. It can be any valid MATLAB value including a structure or cell array.
- Adding output arguments to `varargout`.

  `varargout` is a cell array. It can contain any number of output arguments. By default, GUIDE creates just one output argument, `handles.output`. To create an additional output argument, create a new field in the `handles` structure and add it to `varargout` using a command similar to

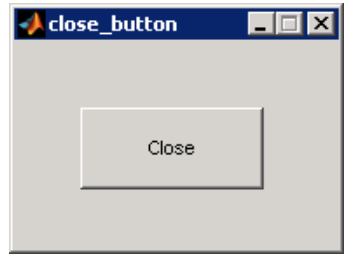
  ```
  varargout{2} = handles.second_output;
  ```

Examples: Programming GUIDE GUI Components

In this section...

Push Button

This example contains only a push button. Clicking the button, closes the GUI.



This is the push button’s `Callback` callback. It displays the string `Goodbye` at the command line and then closes the GUI.

```
function pushbutton1_Callback(hObject, eventdata, handles)
```

```
display Goodbye
close(handles.figure1);
```

Adding an Image to a Push Button or Toggle Button

To add an image to a push button or toggle button, assign the button's `CData` property an m-by-n-by-3 array of RGB values that defines "RGB (Truecolor) Images". For example, the array `a` defines 16-by-64 truecolor image using random values between 0 and 1 (generated by `rand`).

```
a(:,:,1) = rand(16,64);
a(:,:,2) = rand(16,64);
a(:,:,3) = rand(16,64);
set(hObject,'CData',a)
```

To add the image when the button is created, add the code to the button's `CreateFcn` callback. You may want to delete the value of the button's `String` property, which would usually be used as a label.

See `ind2rgb` for information on converting a matrix `X` and corresponding colormap, i.e., an `(X, MAP)` image, to RGB (truecolor) format.

Toggle Button

The callback for a toggle button needs to query the toggle button to determine what state it is in. MATLAB sets the `Value` property equal to the `Max` property when the toggle button is pressed (`Max` is 1 by default) and equal to the `Min` property when the toggle button is not pressed (`Min` is 0 by default).

The following code illustrates how to program the callback in the GUI M-file.

```
function togglebutton1_Callback(hObject, eventdata, handles)
button_state = get(hObject,'Value');
if button_state == get(hObject,'Max')
 % Toggle button is pressed-take approperiate action
```

```
   ...
elseif button_state == get(hObject,'Min')
 % Toggle button is not pressed-take appropriate action
    ...
end
```

You can also change the state of a toggle button programmatically by setting the toggle button `Value` property to the value of the `Max` or `Min` property. For example,

```
set(handles.togglebutton1,'Value','Max')
```

puts the toggle button with `Tag` property `togglebutton1` in the pressed state.

Note You can use a button group to manage exclusive selection behavior for toggle buttons. See "Button Group" on page 8-28 for more information.

Radio Button

You can determine the current state of a radio button from within its callback by querying the state of its `Value` property, as illustrated in the following example:

```
if (get(hObject,'Value') == get(hObject,'Max'))
 % Radio button is selected-take approriate action
else
 % Radio button is not selected-take approriate action
end
```

You can also change the state of a radio button programmatically by setting the radio button `Value` property to the value of the `Max` or `Min` property. For example,

```
set(handles.radiobutton1,'Value','Max')
```

puts the radio button with `Tag` property `radiobutton1` in the selected state.

Note You can use a button group to manage exclusive selection behavior for radio buttons. See “Button Group” on page 8-28 for more information.

Check Box

You can determine the current state of a check box from within its callback by querying the state of its `Value` property, as illustrated in the following example:

```
function checkbox1_Callback(hObject, eventdata, handles)
if (get(hObject,'Value') == get(hObject,'Max'))
 % Checkbox is checked-take approriate action
else
 % Checkbox is not checked-take approriate action
end
```

You can also change the state of a check box by programmatically by setting the check box `Value` property to the value of the `Max` or `Min` property. For example,

```
set(handles.checkbox1,'Value','Max')
```

puts the check box with `Tag` property `checkbox1` in the checked state.

Edit Text

To obtain the string a user types in an edit box, get the `String` property in the `Callback` callback.

```
function edittext1_Callback(hObject, eventdata, handles)
user_string = get(hObject,'String');
% Proceed with callback
```

If the edit text `Max` and `Min` properties are set such that `Max - Min > 1`, the user can enter multiple lines. For example, setting `Max` to `2`, with the default value of `0` for `Min`, enables users to enter multiple lines.

Retrieving Numeric Data from an Edit Text Component

MATLAB returns the value of the edit text `String` property as a character string. If you want users to enter numeric values, you must convert the characters to numbers. You can do this using the `str2double` command, which converts strings to doubles. If the user enters nonnumeric characters, `str2double` returns `NaN`.

You can use the following code in the edit text callback. It gets the value of the `String` property and converts it to a double. It then checks whether the converted value is `NaN` (`isnan`), indicating the user entered a nonnumeric character and displays an error dialog (`errordlg`).

```
function edittext1_Callback(hObject, eventdata, handles)
user_entry = str2double(get(hObject,'string'));
if isnan(user_entry)
 errordlg('You must enter a numeric value','Bad Input','modal')
 return
end
% Proceed with callback...
```

Triggering Callback Execution

If the contents of the edit text component have been changed, clicking inside the GUI but outside the edit text causes the edit text callback to execute. The user can also press **Enter** for an edit text that allows only a single line of text, or **Ctrl+Enter** for an edit text that allows multiple lines.

Available Keyboard Accelerators

GUI users can use the following keyboard accelerators to modify the content of an edit text. These accelerators are not modifiable.

- **Ctrl+X** — Cut
- **Ctrl+C** — Copy
- **Ctrl+V** — Paste
- **Ctrl+H** — Delete last character
- **Ctrl+A** — Select all

Slider

You can determine the current value of a slider from within its callback by querying its `Value` property, as illustrated in the following example:

```
function slider1_Callback(hObject, eventdata, handles)
slider_value = get(hObject,'Value');
% Proceed with callback...
```

The `Max` and `Min` properties specify the slider's maximum and minimum values. The slider's range is `Max - Min`.

List Box

When the list box `Callback` callback is triggered, the list box `Value` property contains the index of the selected item, where `1` corresponds to the first item in the list. The `String` property contains the list as a cell array of strings.

This example retrieves the selected string. It assumes `listbox1` is the value of the `Tag` property. Note that it is necessary to convert the value returned from the `String` property from a cell array to a string.

```
function listbox1_Callback(hObject, eventdata, handles)
index_selected = get(hObject,'Value');
list = get(hObject,'String');
item_selected = list{index_selected}; % Convert from cell array
                                      % to string
```

You can also select a list item programmatically by setting the list box `Value` property to the index of the desired item. For example,

```
set(handles.listbox1,'Value',2)
```

selects the second item in the list box with `Tag` property `listbox1`.

Triggering Callback Execution

MATLAB executes the list box's `Callback` callback after the mouse button is released or after certain key press events:

- The arrow keys change the `Value` property, trigger callback execution, and set the figure `SelectionType` property to `normal`.

- The **Enter** key and space bar do not change the `Value` property but trigger callback execution and set the figure `SelectionType` property to `open`.

If the user double-clicks, the callback executes after each click. MATLAB sets the figure `SelectionType` property to `normal` on the first click and to `open` on the second click. The callback can query the figure `SelectionType` property to determine if it was a single or double click.

List Box Examples

See the following examples for more information on using list boxes:

- "List Box Directory Reader" on page 10-9 — Shows how to creates a GUI that displays the contents of directories in a list box and enables users to open a variety of file types by double-clicking the filename.
- "Accessing Workspace Variables from a List Box" on page 10-16 — Shows how to access variables in the MATLAB base workspace from a list box GUI.

Pop-Up Menu

When the pop-up menu `Callback` callback is triggered, the pop-up menu `Value` property contains the index of the selected item, where `1` corresponds to the first item on the menu. The `String` property contains the menu items as a cell array of strings.

Note A pop-up menu is sometimes referred to as a drop-down menu or combo box.

Using Only the Index of the Selected Menu Item

This example retrieves only the index of the item selected. It uses a switch statement to take action based on the value. If the contents of the pop-up menu are fixed, then you can use this approach. Else, you can use the index to retrieve the actual string for the selected item.

```
function popupmenu1_Callback(hObject, eventdata, handles)
val = get(hObject,'Value');
switch val
case 1
```

```
% User selected the first item
case 2
% User selected the second item
% Proceed with callback...
```

You can also select a menu item programmatically by setting the pop-up menu `Value` property to the index of the desired item. For example,

```
set(handles.popupmenu1,'Value',2)
```

selects the second item in the pop-up menu with `Tag` property `popupmenu1`.

Using the Index to Determine the Selected String

This example retrieves the actual string selected in the pop-up menu. It uses the pop-up menu `Value` property to index into the list of strings. This approach may be useful if your program dynamically loads the contents of the pop-up menu based on user action and you need to obtain the selected string. Note that it is necessary to convert the value returned by the `String` property from a cell array to a string.

```
function popupmenu1_Callback(hObject, eventdata, handles)
val = get(hObject,'Value');
string_list = get(hObject,'String');
selected_string = string_list{val}; % Convert from cell array
                                    % to string
% Proceed with callback...
```

Panel

Panels group GUI components and can make a GUI easier to understand by visually grouping related controls. A panel can contain panels and button groups as well as axes and user interface controls such as push buttons, sliders, pop-up menus, etc. The position of each component within a panel is interpreted relative to the lower-left corner of the panel.

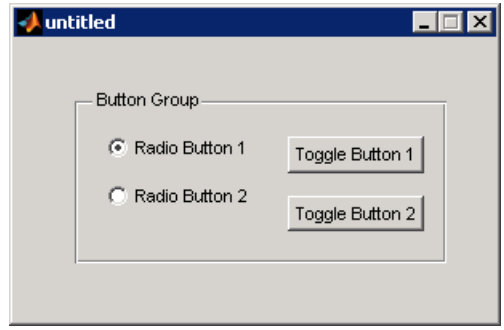
Generally, if the GUI is resized, the panel and its components are also resized. However, you can control the size and position of the panel and its components. You can do this by setting the GUI **Resize behavior** to **Other (Use ResizeFcn)** and providing a `ResizeFcn` callback for the panel.

Note To set **Resize behavior** for the figure to **Other (Use ResizeFcn)**, select **GUI Options** from the Layout Editor **Tools** menu. See "Cross-Platform Compatible Units" on page 6-103 for information about the effect of units on resize behavior.

Button Group

Button groups are like panels except that they manage exclusive selection behavior for radio buttons and toggle buttons. If a button group contains a set of radio buttons, toggle buttons, or both, the button group allows only one of them to be selected. When a user clicks a button, that button is selected and all others are deselected.

The following figure shows a button group with two radio buttons and two toggle buttons. **Radio Button 1** is selected.



If a user clicks the other radio button or one of the toggle buttons, it becomes selected and **Radio Button 1** is deselected. The following figure shows the result of clicking **Toggle Button 2**.

The button group's `SelectionChangeFcn` callback is called whenever a selection is made. Its `hObject` input argument contains the handle of the selected radio button or toggle button.

If you have a button group that contains a set of radio buttons and toggle buttons and you want:

- An immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group's `SelectionChangeFcn` callback function, not in the individual toggle button `Callback` functions. "Color Palette" on page 15-17 provides a practical example of a `SelectionChangeFcn` callback.
- Another component such as a push button to base its action on the selection, then that component's `Callback` callback can get the handle of the selected radio button or toggle button from the button group's `SelectedObject` property.

This example of a `SelectionChangeFcn` callback uses the `Tag` property of the selected object to choose the appropriate code to execute. For GUIDE GUIs, unlike other callbacks, the `hObject` argument of the `SelectionChangeFcn` callback contains the handle of the selected radio button or toggle button.

```
function uibuttongroup1_SelectionChangeFcn(hObject,...
    eventdata,handles)
```

```
switch get(hObject,'Tag')   % Get Tag of selected object
    case 'radiobutton1'
        % Code for when radiobutton1 is selected.
    case 'radiobutton2'
        % Code for when radiobutton2 is selected.
    case 'togglebutton1'
        % Code for when togglebutton1 is selected.
    case 'togglebutton2'
        % Code for when togglebutton2 is selected.
    % Continue with more cases as necessary.
    otherwise
        % Code for when there is no match.
end
```

See the `uibuttongroup` reference page for another example.

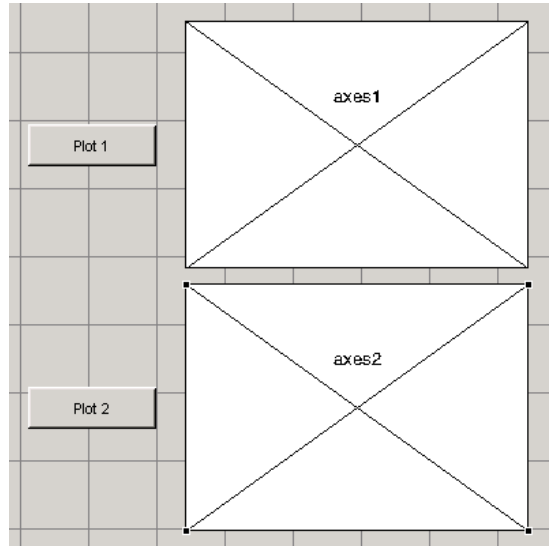
Axes

Axes components enable your GUI to display graphics, such as graphs and images. This topic briefly tells you how to plot to axes components in your GUI.

Plotting to an Axes

In most cases, you create a plot in an axes from a callback that belongs to some other component in the GUI. For example, pressing a button might trigger the plotting of a graph to an axes. In this case, the button's `Callback` callback contains the code that generates the plot.

The following example contains two axes and two buttons. Clicking one button generates a plot in one axes and clicking the other button generates a plot in the other axes. The following figure shows these components as they might appear in the Layout Editor.



1. Add this code to the **Plot 1** push button's `Callback` callback. The `surf` function produces a 3-D shaded surface plot. The `peaks` function returns a square matrix obtained by translating and scaling Gaussian distributions.

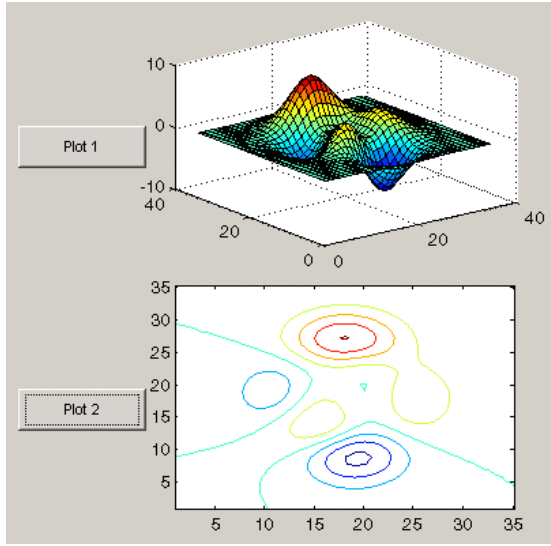
   ```
   surf(handles.axes1,peaks(35));
   ```

2. Add this code to the **Plot 2** push button's `Callback` callback. The `contour` function displays the contour plot of a matrix, in this case the output of `peaks`.

   ```
   contour(handles.axes2,peaks(35));
   ```

3. Run the GUI by selecting **Run** from the **Tools** menu.

4 Click the **Plot 1** button to display the surf plot in the first axes. Click the **Plot 2** button to display the contour plot in the second axes.



See “GUI with Multiple Axes” on page 10-2 for a more complex example that uses two axes.

Note For information about properties that you can set to control many aspects of axes behavior and appearance, see “Axes Properties” in the MATLAB Graphics documentation. For information about plotting in general, see “Plots and Plotting Tools” in the MATLAB Graphics documentation.

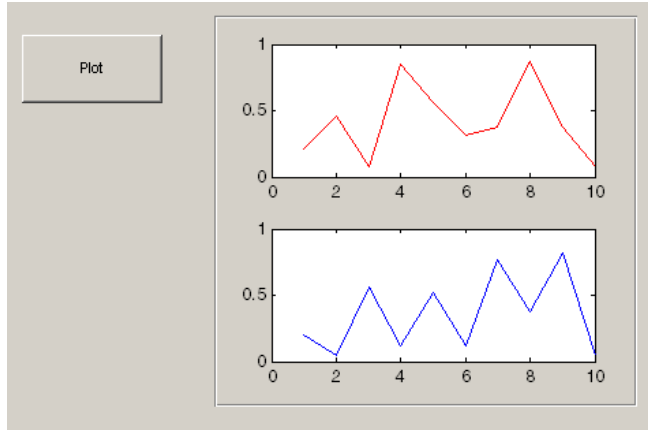
If your GUI contains axes, you should make sure that the **Command-line accessibility** option in the GUI Options dialog box is set to **Callback** (the default). From the Layout Editor select **Tools > GUI Options > Command Line Accessibility: Callback**. See “Command-Line Accessibility” on page 5-10 for more information about how this option works.

Creating Subplots

Use the `subplot` function to create axes in a tiled pattern. If your GUIDE-generated GUI contains components other than the subplots, the subplots must be contained in a panel.

As an example, the following code uses the `subplot` function to create an axes with two subplots in the panel with `Tag` property `uipanel1`. This code is part of the **Plot** push button `Callback` callback. Each time you press the **Plot** button, the code draws a line in each subplot. `a1` and `a2` are the handles of the subplots.

```
a1=subplot(2,1,1,'Parent',handles.uipanel1);
plot(a1,rand(1,10),'r');
a2=subplot(2,1,2,'Parent',handles.uipanel1);
plot(a2,rand(1,10),'b');
```



For more information about subplots, see the `subplot` reference page. For information about adding panels to your GUI, see “Adding Components to the GUIDE Layout Area” on page 6-22.

ActiveX Control

This example programs a sample ActiveX control **Mwsamp Control**. It first enables a user to change the radius of a circle by clicking on the circle. It then programs a slider on the GUI to do the same thing.

This topic also discusses:

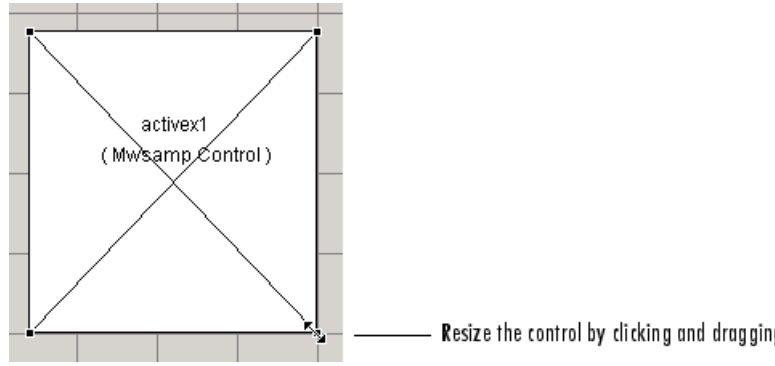
Note See "MATLAB COM Client Support" in the MATLAB External Interfaces documentation to learn more about ActiveX controls.

Programming an ActiveX Control

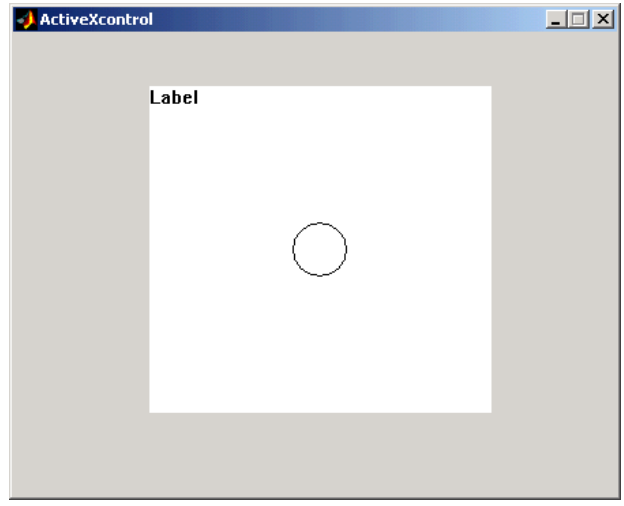
The sample ActiveX control **Mwsamp Control** contains a circle in the middle of a square. This example programs the control to change the circle radius when the user clicks the circle, and to update the label to display the new radius.

1 Add the sample ActiveX control **Mwsamp** to your GUI and resize it to approximately the size of the square shown in the preview pane. The following figure shows the ActiveX control as it appears in the Layout Editor.

If you need help adding the component, see “Adding Components to the GUIDE Layout Area” on page 6-22.

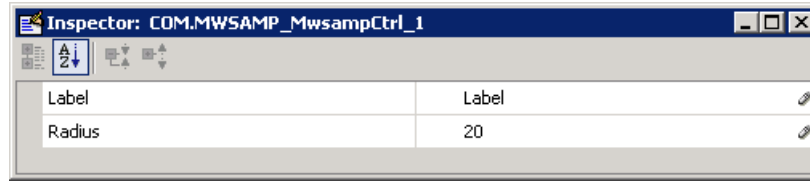


2 Activate the GUI by clicking the button on the toolbar and save the GUI when prompted. GUIDE displays the GUI shown in the following figure and opens the GUI M-file.



3 View the ActiveX Properties with the Property Inspector. Select the control in the Layout Editor, and then select **Property Inspector** from the **View** menu or by clicking the **Property Inspector** button on the toolbar.

The following figure shows properties of the `mwsamp` ActiveX control as they appear in the Property Inspector. The properties on your system may differ.



This ActiveX control `mwsamp` has two properties:

- `Label`, which contains the text that appears at the top of the control
- `Radius`, the default radius of the circle, which is 20

4 Add the following code to the `mswamp` control's `Click` callback. This code programs the ActiveX control to change the circle radius when the user clicks the circle, and updates the label to display the new radius.

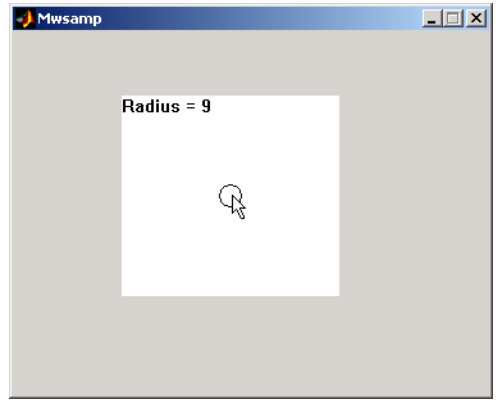
```
hObject.radius = .9*hObject.radius;
hObject.label = ['Radius = ' num2str(hObject.radius)];
refresh(handles.figure1);
```

To locate the `Click` callback in the GUI M-file, select **View Callbacks** from the **View** menu and then select **Click**.

5 Add the following commands to the opening function. This code initializes the label when you first open the GUI.

```
handles.activex1.label = ...
['Radius = ' num2str(handles.activex1.radius)];
```

Save the M-file. Now, when you open the GUI and click the ActiveX control, the radius of the circle is reduced by 10 percent and the new value of the radius is displayed. The following figure shows the GUI after clicking the circle six times.



If you click the GUI enough times, the circle disappears.

Programming a User Interface Control to Update an ActiveX Control

This topic continues the previous example by adding a slider to the GUI and programming the slider to change the circle radius. This example must also update the slider if the user clicks the circle.

1 Add a slider to your layout and then add the following code to the `slider1 Callback` callback:

```
handles.activex1.radius = ...
    get(hObject,'Value')*handles.default_radius;
handles.activex1.label = ...
    ['Radius = ' num2str(handles.activex1.radius)];
refresh(handles.figure1);
```

The first command

- Gets the `Value` of the slider, which in this example is a number between 0 and 1, the default values of the slider's `Min` and `Max` properties.

- Sets `handles.activex1.radius` equal to the `Value` times the default radius.

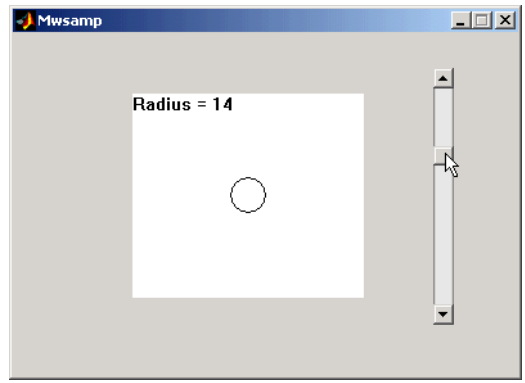
2. In the opening function, add the default radius to the `handles` structure. The `activex1_Click` callback uses the default radius to update the slider value if the user clicks the circle.

```
handles.default_radius = handles.activex1.radius;
```

3. In the `activex1_Click` callback, reset the slider's `Value` each time the user clicks the circle in the ActiveX control. The following command causes the slider to change position corresponding to the new value of the radius.

```
set(handles.slider1,'Value',...
    hObject.radius/handles.default_radius);
```

When you open the GUI and move the slider by clicking and dragging, the radius changes to a new value between 0 and the default radius of 20, as shown in the following figure.



Viewing the Methods for an ActiveX Control

To view the available methods for an ActiveX control, you first need to obtain the handle to the control. One way to do this is the following:

1. In the GUI M-file, add the command `keyboard` on a separate line of the `activex1_Click` callback. The command `keyboard` puts MATLAB in

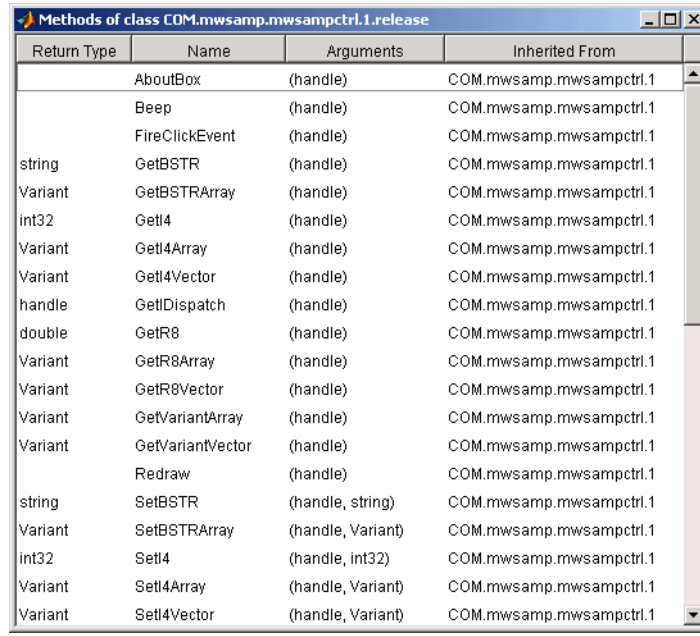
debug mode and pauses at the `activex1_Click` callback when you click the ActiveX control.

2 Run the GUI and click the ActiveX control. The handle to the control is now set to `hObject`.

3 To view the methods for the control, enter

```
methodsview(hObject)
```

This displays the available methods in a new window, as shown in the following figure.



Alternatively, you can enter

```
methods(hObject)
```

which displays the available methods in the MATLAB Command Window.

For more information about methods for ActiveX controls, see "Invoking Methods" in the External Interfaces documentation. See the reference pages for `methodsview` and `methods` for more information about these functions.

Saving a GUI That Contains an ActiveX Control

When you save a GUI that contains ActiveX controls, GUIDE creates a file in the current directory for each such control. The filename consists of the name of the GUI followed by an underscore (_) and `activex`*`n`*, where `n` is a sequence number. For example, if the GUI is named `mygui`, then the filename would be `mygui_activex1`. The filename does not have an extension.

Compiling a GUI That Contains an ActiveX Control

If you use the MATLAB Compiler `mcc` command to compile a GUI that contains an ActiveX control, you must use the `-a` flag to add the ActiveX file, which GUIDE saves in the current directory, to the CTF archive. Your command should be similar to

```
mcc -m mygui -a mygui_activex1
```

where `mygui_activex1` is the name of the ActiveX file. See the "MATLAB Compiler" documentation for more information. If you have more than one such file, use a separate `-a` flag for each file. You must have installed the MATLAB Compiler to compile a GUI.

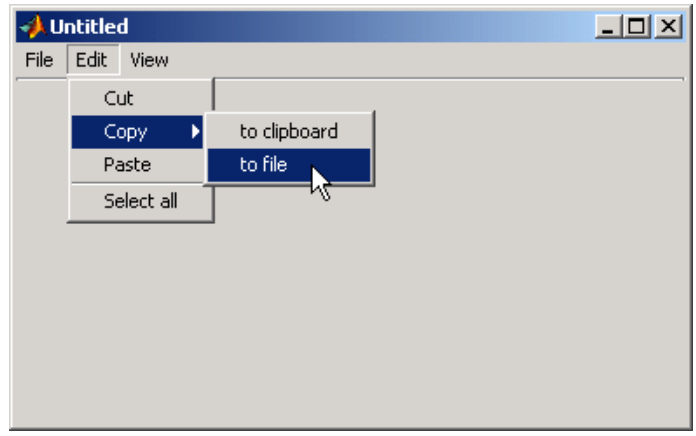
Menu Item

The Menu Editor generates an empty callback subfunction for every menu item, including menu titles.

Programming a Menu Title

Because clicking a menu title automatically displays the menu below it, you may not need to program callbacks at the title level. However, the callback associated with a menu title can be a good place to enable or disable menu items below it.

Consider the example illustrated in the following picture.



When a user selects the **to file** option under the **Edit** menu's **Copy** option, only the **to file** callback is required to perform the action.

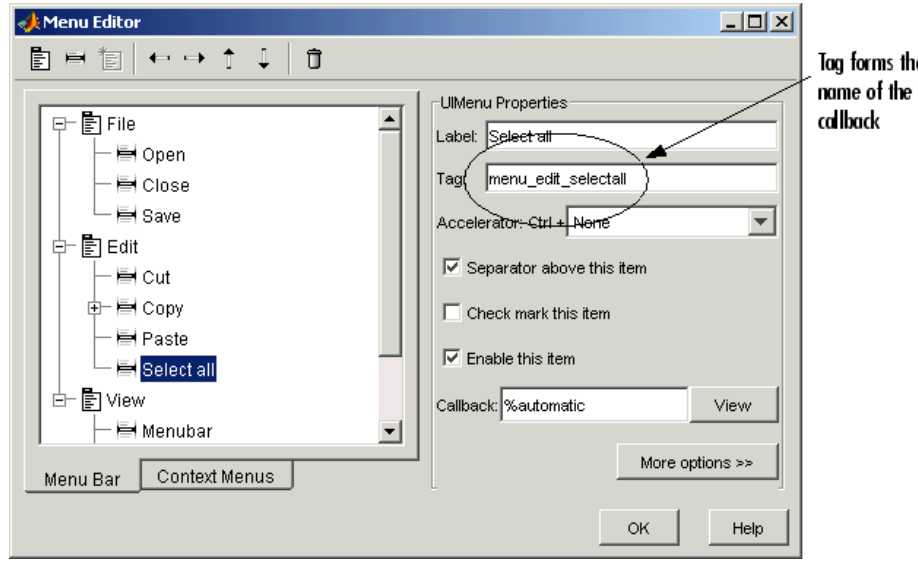
Suppose, however, that only certain objects can be copied to a file. You can use the **Copy** item `Callback` callback to enable or disable the **to file** item, depending on the type of object selected.

Opening a Dialog Box from a Menu Callback

The `Callback` callback for the **to file** menu item could contain code such as the following to display the standard dialog box for saving files.

```
[file,path] = uiputfile('animinit.m','Save file name');
```

`'Save file name'` is the dialog box title. In the dialog box, the filename field is set to `animinit.m`, and the filter set to M-files (`*.m`). For more information, see the `uiputfile` reference page.



Updating a Menu Item Check

A check is useful to indicate the current state of some menu items. If you selected **Check mark this item** in the Menu Editor, the item initially appears checked. Each time the user selects the menu item, the callback for that item must turn the check on or off. The following example shows you how to do this by changing the value of the menu item's `Checked` property.

```
if strcmp(get(hObject, 'Checked'),'on')
    set(hbject,'Checked','off');
else
    set(hObject,'Checked','on');
end
```

`hObject` is the handle of the component, for which the callback was triggered. The `strcmp` function compares two strings and returns logical `1` (true) if the two are identical and logical `0` (false) otherwise.

Use of checks when the GUI is first displayed should be consistent with the display. For example, if your GUI has an axes that is visible when a user first opens it and the GUI has a **Show axes** menu item, be sure to set the menu item's `Checked` property `on` when you create it so that a check appears next to the **Show axes** menu item initially.

Note From the Menu Editor, you can view a menu item's `Callback` callback in your editor by selecting the menu item and clicking the **View** button.

9

Managing and Sharing Application Data in GUIDE

Mechanisms for Managing Data (p. 9-2)	Describes various mechanisms for managing application-defined data. Explains how GUIDE uses several of these mechanisms.
Sharing Data Among a GUI's Callbacks (p. 9-8)	Shows how each mechanism for managing data can be used to share data among a GUI's callbacks.
Making Multiple GUIs Work Together (p. 9-15)	Ways and means to communicate application-defined data between multiple GUIs

Mechanisms for Managing Data

In this section...

Overview

Most GUIs generate or use data that is specific to the application. This topic describes the three mechanisms for managing application-defined data in the GUI environment. These mechanisms provide a way for applications to save and retrieve data stored with the GUI.

The GUI data and application data mechanisms are similar but GUI data can be simpler to use. GUIDE specifically uses GUI data to manage the `handles` structure, but you can use either the GUI data `handles` structure or application data to manage application-defined data. The `UserData` property can also hold application-defined data.

GUI Data

GUI data is managed using the `guidata` function. This function can store a single variable as GUI data. It is also used to retrieve the value of that variable.

About GUI Data

GUI data is always associated with the GUI figure. It is available to all callbacks of all components of the GUI. If you specify a component handle

when you save or retrieve GUI data, MATLAB automatically associates the data with the component's parent figure.

GUI data can contain only one variable at any time. Writing GUI data overwrites the existing GUI data. For this reason, GUI data is usually defined to be a structure to which you can add fields as you need them.

GUI data provides application developers with a convenient interface to a figure's application data:

- You do not need to create and maintain a hard-coded name for the data throughout your source code.
- You can access the data from within a callback routine using the component's handle, without having to find the figure handle. For GUIDE users, the object handle is automatically passed to each callback as `hObject`.

GUI Data in GUIDE

GUIDE uses `guidata` to create and maintain the `handles` structure. The `handles` structure contains the handles of all components in the GUI. GUIDE automatically passes the `handles` structure to every callback as an input argument.

In a GUI created using GUIDE, you cannot use `guidata` to manage any variable other than the `handles` structure. If you do, you may overwrite the `handles` structure and your GUI will not work. If you want to use GUI data to share application-defined data among callbacks, you must save the data in fields that you add to the `handles` structure.

The GUIDE templates use the `handles` structure to store application-defined data. See "Selecting a GUI Template" on page 6-7 for information about the templates.

Note For more information, see "handles Structure" on page 8-15.

Adding Fields to the handles Structure

To add a field to the `handles` structure, which is passed as an argument to every callback in GUIDE.

1. Assign a value to the new field. This adds the field to the structure. For example

   ```
   handles.number_errors = 0;
   ```

   adds the field `number_errors` to the structure `handles` and sets it to `0`.

2. Use the following command to save the data.

   ```
   guidata(hObject,handles)
   ```

   where `hObject` is the handle of the component for which the callback was triggered. It is passed automatically to every callback.

Changing GUI Data in an M-File Generated by GUIDE

In a GUIDE-generated M-file, GUI data is always represented by the `handles` structure. This example updates the `handles` structure and then saves it.

1. Assume that the `handles` structure contains an application-defined field `handles.when` whose value is `'now'`.

2. In a GUI callback, make the desired change to the `handles` structure. This step changes the value of `handles.when` to `'later'`, but does not save the `handles` structure.

   ```
   handles.when = 'later';
   ```

3. Save the changed version of the `handles` structure with the command

   ```
   guidata(hObject,handles)
   ```

   where `hObject`, which is passed automatically to every callback, is the handle of the component for which the callback was triggered. If you do not save the `handles` structure with `guidata`, the change you made to it in the previous step is lost.

Application Data

Application data provides a way for applications to save and retrieve data associated with a specified object. For a GUI, this is usually the GUI figure but can also be any component. The data is stored as name/value pairs. Application data enables you to create what are essentially user-defined properties for an object.

The following table summarizes the functions that provide access to application data. For more detailed information, see the individual function reference pages.

Functions for Managing Application Data

Function	Purpose
`setappdata`	Specify named application data for an object. The object does not have to be a figure. You can specify more than one named application data for an object. However, each name must be unique for that object and can be associated with only one value, usually a structure.
`getappdata`	Retrieve named application data. To retrieve named application data, you must know the name associated with the application data and the handle of the object with which it is associated.
`isappdata`	True if the named application data exists.
`rmappdata`	Remove the named application data.

Creating Application Data in GUIDE

Use the `setappdata` function to create application data. This example generates a 35-by-35 matrix of normally distributed random numbers in the opening function and creates application data `mydata` to manage it.

```
function mygui_OpeningFcn(hObject, eventdata, handles, varargin)
matrices.rand_35 = randn(35);
setappdata(hObject,'mydata',matrices);
```

Because this code appears in the opening function, `hObject` is the handle of the GUI figure, and the code associates `mydata` with the figure.

Adding Fields to an Application Data Structure in GUIDE

Application data is usually defined as a structure to enable you to add fields as necessary. In this example, a push button adds a field to the application data structure `mydata` created in the previous topic.

1. Use `getappdata` to retrieve the structure.

   From the example in the previous topic, the name of the application data structure is `mydata`. It is associated with the figure whose `Tag` is `figure1`. Since the `handles` structure is passed to every callback, the code can specify the figure's handle as `handles.figure1`.

   ```
   function mygui_pushbutton1(hObject, eventdata, handles)
   matrices = getappdata(handles.figure1,'mydata');
   ```

2. Create a new field and assign it a value. For example

   ```
   matrices.randn_50 = randn(50);
   ```

   adds the field `randn_50` to the `matrices` structure and sets it to a 50-by-50 matrix of normally distributed random numbers.

3. Use `setappdata` to save the data. This example uses `setappdata` to save the `matrices` structure as the application data structure `mydata`.

   ```
   setappdata(handles.figure1,'mydata',matrices);
   ```

UserData Property

All GUI components, including menus, and the figure have a `UserData` property. You can assign any valid MATLAB value to the `UserData` property. To retrieve the data, a callback must know the handle of the component with which the data is associated.

1. In this example, an edit text component stores the user-entered string in its `UserData` property.

   ```
   function mygui_edittext1(hObject, eventdata, handles)
   mystring = get(hObject,'String');
   set(hObject,'UserData',mystring);
   ```

2 A push button retrieves the string from the edit text component `UserData` property. The callback uses the `handles` structure and the edit text `Tag` property, `edittext1`, to specify the edit text handle.

```
function mygui_pushbutton1(hObject, eventdata, handles)
string = get(handles.edittext1,'UserData');
```

Sharing Data Among a GUI's Callbacks

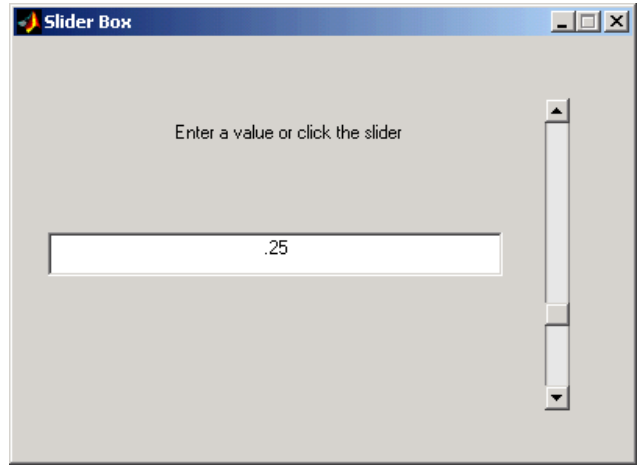
In this section...

GUI Data

GUI data, which you manage with the `guidata` function, is accessible to all callbacks of the GUI. A callback for one component can set a value in GUI data, which can then be read by a callback for another component. See "GUI Data" on page 9-2 for more information about GUI data.

GUI Data Example: Passing Data Between Components

This example uses a GUI that contains a slider and an edit text component as shown in the following figure. A static text component instructs the user to enter a value in the edit text or click the slider. The example uses GUI data to initialize and maintain an error counter.



The GUI behavior is as follows:

- When a user moves the slider, the edit text component displays the slider's current value.
- When a user types a value into the edit text component, the slider updates to this value.
- If a user enters a value in the edit text that is out of range for the slider — that is, a value that is not between 0 and 1 — the application returns a message in the edit text component indicating how many times the user has entered an erroneous value.

The commands given in the following steps initialize the error counter and implement the interchange between the slider and the edit text component.

1 Define the error counter in the opening function. The GUI records the number of times a user enters an erroneous value in the edit text component and stores this number in a field of the `handles` structure.

Start by defining this field, called `number_errors`, in the opening function as follows:

```
handles.number_errors = 0;
```

Type the preceding statement before the following line, which GUIDE automatically inserts into the opening function.

```
guidata(hObject,handles); % Save the updated handles structure.
```

The `guidata` command saves the modified `handles` structure so that it can be retrieved in the GUI's callbacks.

2 Set the value of the edit text component `String` property from the slider `Callback` callback. The following command in the slider `Callback` updates the value displayed in the edit text component when a user moves the slider and releases the mouse button.

```
set(handles.edittext1,'String',...
   num2str(get(handles.slider1,'Value')));
```

The code combines three commands:

- The `get` command obtains the current value of the slider.

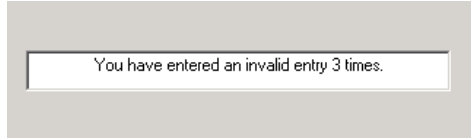
- The `num2str` command converts the value to a string.
- The `set` command sets the `String` property of the edit text to the updated value.

3. Set the slider value from the edit text component's `Callback` callback. The edit text `Callback` sets the slider's value to the number the user types in, after checking to see if it is a single numeric value between 0 and 1. If the value is out of range, then the error count is incremented and the edit text displays a message telling the user how many times they have entered an invalid number. Because this code appears in the edit text `Callback`, `hObject` is the handle of the edit text component.

```
val = str2double(get(hobject,'String'));
% Determine whether val is a number between 0 and 1.
if isnumeric(val) && length(val)==1 && ...
   val >= get(handles.slider1,'Min') && ...
   val <= get(handles.slider1,'Max')
   set(handles.slider1,'Value',val);
else
% Increment the error count, and display it.
   handles.number_errors = handles.number_errors+1;
   guidata(hObject,handles); % Store the changes.
   set(hobject,'String',...
   ['You have entered an invalid entry ',...
num2str(handles.number_errors),' times.']);
end
```

If the user types a number between 0 and 1 in the edit text component and then presses **Enter** or clicks outside the edit text, the `Callback` sets `handles.slider1` to the new value and the slider moves to the corresponding position.

If the entry is invalid — for example, `2.5` — the GUI increments the value of `handles.number_errors` and displays a message like the following in the edit text component:



Application Data

Application data can be associated with any object — a component, menu, or the figure itself. To access application data, a callback must know the name of the data and the handle of the component with which it is associated. Use the functions `setappdata`, `getappdata`, `isappdata`, and `rmappdata` to manage application data.

See "Application Data" on page 9-5 for more information about application data.

Application Data Example: Passing Data Between Components

The previous topic, "GUI Data Example: Passing Data Between Components" on page 9-8, uses GUI data to initialize and maintain an error counter. This example shows you how to do the same thing using application data. Refer to the previous topic for details of the example.

1. Define the error counter in the opening function. Add the following code to the opening function. This code first creates a structure `slider_data`, then assigns it to the named application data `slider`. Because this code appears in the opening function, using `hObject` associates the application data with the figure.

   ```
   slider_data.number_errors = 0;
   setappdata(hObject,'slider',slider_data);
   ```

2. Set the value of the edit text `String` property from the slider `Callback` callback. Before you can do this, you must convert the slider `Value` property to a string. Add this statement to the callback.

   ```
   set(handles.edittext1,'String',num2str(get(hObject,'Value')));
   ```

   Because this statement appears in the slider Callback, `hObject` is the handle of the slider.

3 Set the slider value from the edit text component's `Callback` callback. Add this code to the callback. It assumes the figure's `Tag` property is `figure1`.

To update the number of errors, the code must first retrieve the named application data `slider`, then increment the count. It then saves the application data and displays the new error count.

```
val = str2double(get(hObject,'String'));
% Determine whether val is a number between 0 and 1.
if isnumeric(val) && length(val)==1 && ...
   val >= get(handles.slider1,'Min') && ...
   val <= get(handles.slider1,'Max')
   set(handles.slider1,'Value',val);
else
% Retrieve and increment the error count.
   slider_data = getappdata(handles.figure1,'slider');
   slider_data.number_errors = slider_data.number_errors+1;
% Save the changes.
   setappdata(handles.figure1,'slider',slider_data);
% Display new total.
   set(hObject,'String',...
   ['You have entered an invalid entry ',...
num2str(slider_data.number_errors),' times.']);
end
```

UserData Property

Every GUI component, and the figure itself, has a `UserData` property that you can use to store application-defined data. To access `UserData`, a callback must know the handle of the component with which the property is associated.

Use the `get` function to retrieve `UserData`, and the `set` function to set it.

UserData Property Example: Passing Data Between Components

A previous topic, "GUI Data Example: Passing Data Between Components" on page 9-8, uses GUI data to initialize and maintain an error counter. This example shows you how to do the same thing using the edit text component's `UserData` property to store the error count. Refer to the GUI data example for example details.

1 Initialize the edit text component `UserData` property in the opening function by adding the following code to the opening function. This code initializes the data in a structure to allow for other data that may be needed.

```
data.number_errors = 0;
set(handles.edittext1,'UserData',data.number_errors)
```

Note Alternatively, you could add a `CreateFcn` callback for the edit text, and initialize the error counter there.

---

2 Set the edit text value from the slider `Callback` callback. Add this statement to the callback.

```
set(handles.edittext1,'String',...
   num2str(get(hObject,'Value')));
```

where `hObject` is the handle of the slider.

3 Set the slider value from the edit text `Callback` callback. To do this, add the following code to the callback.

To update the number or errors, the code must first retrieve the value of the edit text `UserData` property, then increment the count. It then saves the updated error count in the `UserData` property and displays the new count.

```
val = str2double(get(hObject,'String'));
% Determine whether val is a number between 0 and 1.
if isnumeric(val) && length(val)==1 && ...
   val >= get(handles.slider1,'Min') && ...
   val <= get(handles.slider1,'Max')
   set(handles.slider1,'Value',val);
else
% Retrieve and increment the error count.
   data = get(hObject,'UserData');
   data.number_errors = data.number_errors+1;
% Save the changes.
   set(hObject,'UserData',data);
% Display new total.
   set(hObject,'String',...
```

```
   ['You have entered an invalid entry ',...
num2str(number_errors),' times.']);
end
```

Because this code appears in the edit text `Callback`, `hObject` is the handle of the edit text component.

Making Multiple GUIs Work Together

In this section...

Overview of Data Sharing Techniques

Although most GUIs created in GUIDE use single figures, you can make several GUIDE-generated GUIs work together if your application requires more than a single figure. For example, your GUI may need to use several dialogs to display and obtain some of the parameters used by the GUI, or your GUI may include several individual tools that work together, either at the same time or in succession. This section describes the different techniques you can use to share data among multiple GUIDE GUIs to make them operate together. It also provides examples that show you how to use these techniques to make a set of GUIs cooperate with one another.

GUIs can share data in many ways. In a given application, more than one technique can be—and often is—used. Without resorting to communicating via files or workspace variables, you can use any of the approaches described in this table.

Data Sharing Method	How it Works	Use for
Property/Value pairs	Send data into a newly invoked or existing GUI by passing it along as input arguments.	Communicating data to new GUIs

Data Sharing Method	How it Works	Use for
Output	Return data from the invoked GUI.	Communicating data back to the invoking GUI, such as passing back the handles structure of the invoked GUI
Function Handles	Pass function handles as data through one of the three following methods.	Exposing functionality of the GUI within a GUI or between GUIs
userdata	Store data in a figure or component; communicate to other GUIs via handle references.	Communicating data within a GUI or between GUIs
getappdata/setappdata	Store data as a property in a figure or component; communicate to other GUIs via handle references	Communicating data within a GUI or across GUIs
guidata	Store data in the handles structure of a GUI; communicate to other GUIs via handle references.	Communicating data within a GUI or across GUIs; a convenient way to manage application data

The techniques described in “Sharing Data Among a GUI’s Callbacks” on page 9-8 that enable you to share data within a GUI—userdata, application data, and guidata—can also share data between several GUIs as long as the handles to objects in the first GUI are made available to other GUIs. The rest of this section provides two examples that illustrate these techniques. The first example describes how a simple GUI can open and receive data from a modal dialog. The second, more extensive, example illustrates how the three components of an icon editor are made to interact.

Note The examples that follow omit portions of code in order to more clearly convey data sharing techniques. The omissions are noted by ellipses like these:

.
.
.

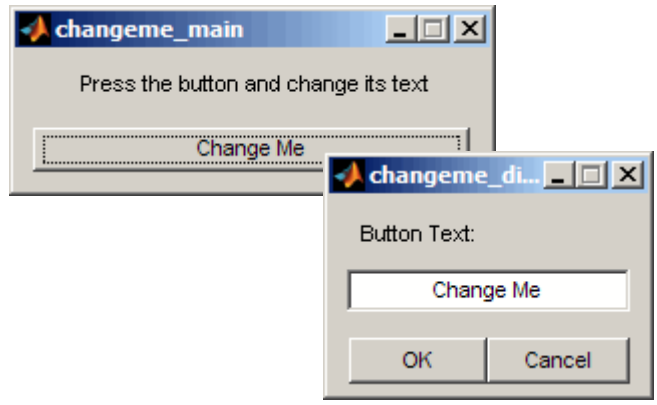
Complete M-files and FIG-files that you can run, view, and modify are provided for the examples.

Example — A GUIDE GUI with a Modal Dialog for User Input

This simple example demonstrates how data is passed to a modal dialog invoked from a GUIDE GUI. The dialog displays text data in an edit field in the dialog. Any changes to it that the user makes are passed back to the main GUI. That data can be used by the main GUI in various ways. In this example, the data updates the appearance of one of the components of the main GUI. The example illustrates how to do many common tasks involved in making multiple GUIs work together, for example, how to position a second GUI relative to the main GUI.

The main GUI contains one pushbutton and a static text field giving instructions. Clicking the button opens a modal dialog box. In it, the button's current string displays in an editable text field, and the user can change it. If the user clicks OK, the value of the text field is returned to the main GUI,

which sets the string property of its button to be that value. The main GUI and its modal dialog box are shown in the following figure.



Note The following links execute MATLAB commands and work only within the MATLAB Help browser. If you are reading this on the Web or in the PDF, go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display the `changeme` GUIs in the Layout Editor.
- Click here to display the `changeme` GUI M-files in the editor.

Opening the Text Change Dialog

Clicking the **Change Me** button causes the Text Change dialog to be invoked. When invoking the dialog, arguments include a property/value pair with name `'changeme_main'` (the main GUI's name) and value set to the handle to the main figure. This data enables the dialog to access the main GUI's data; if it is missing, the dialog displays an error that describes proper usage and exits.

```
function buttonChangeMe_Callback(hObject, ...
         eventdata, handles)
changeme_dialog('changeme_main', handles.figure);
```

Managing the Text Change Dialog

The Text Change dialog should be modal. In the Property Inspector for the Text Change dialog's figure, set the `'WindowStyle'` property to `'Modal'`. This ensures that the user can interact with no other figures while it is active.

To ensure proper behavior, use `uiwait` in the `OpeningFcn` of the dialog. Invoking `uiwait` puts off calling the output function until `uiresume` is called. This also keeps the invocation call of the GUI from returning until that time:

```
function changeme_dialog_OpeningFcn(hObject, ...
         eventdata, handles, varargin)
.
.
.
uiwait(hObject);
.
.
.
```

Every callback in which the GUI needs to close should call `uiresume`. In this example, it can happen in the `CloseRequestFcn` for the figure, the **Cancel** button, and the **OK** button:

```
function buttonCancel_Callback(hObject, ...
         eventdata, handles)
uiresume(handles.figure);

function figure_CloseRequestFcn(hObject, ...
         eventdata, handles)
uiresume(hObject);

function buttonOK_Callback(hObject, e...
         ventdata, handles)
.
.
.
uiresume(handles.figure);
```

In the `OutputFcn`, make sure to delete the figure, so that it closes:

```
function varargout = changeme_dialog_Dialog_OutputFcn(hObject, ...
```

```
         eventdata, handles)
varargout{1} = [];
delete(hObject);
```

Protecting the Text Change Dialog

If the Text Change dialog is not invoked from the main GUI, it displays an error and exits. The `OpeningFcn` for the dialog scans the input arguments for the `changeme_main` property. If it isn't found or has a value that is not valid, the modal dialog displays a message and then destroys itself. To be able to exit immediately, do not call `uiwait` until after validating the input. If `uiwait` is not called, the dialog immediately calls its `OutputFcn` and returns. As described earlier, the `OutputFcn` closes the figure.

```
function changeme_dialog_OpeningFcn(hObject, ...
         eventdata, handles, varargin)

% Check to see the changeme_main gui is passed in
dontOpen = false;
mainGuiInput = find(strcmp(varargin, 'changeme_main'));
if (isempty(mainGuiInput))
  || (length(varargin) <= mainGuiInput)
  || (~ishandle(varargin{mainGuiInput+1}))
    dontOpen = true;
else
.
.
.
end
.
.
.
if dontOpen
   disp('------------------------------------------------------');
   disp('Improper input arguments. Pass a property value pair')
   disp('whose name is "changeme_main" and value is the handle')
   disp('to the changeme_main figure.');
   disp('eg:');
   disp('   x = changeme_main()');
   disp('   changeme_dialog('changeme_main', x)');
```

```
   disp('--------------------------------------------------');
else
   uiwait(hObject);
end
```

Positioning the Text Change Dialog

The Text Change dialog (`changeme_dialog`) should position itself close to the invoking figure. To avoid distracting the user, the dialog box appears next to the main GUI. If the main figure is moved somewhere and the dialog is invoked, it opens in a different location from where it would have otherwise. Using the passed-in handle to the main figure, get the main figure's position and do some calculations to offset the dialog box to the right and down:

```
function changeme_dialog_OpeningFcn(hObject, ...
         eventdata, handles, varargin)
.
.
.
mainGuiInput = find(strcmp(varargin, 'changeme_main'));
.
.
.
handles.changeMeMain = varargin{mainGuiInput+1};
.
.
.
    % Position to be relative to parent:
    parentPosition = getpixelposition(handles.changeMeMain);
    currentPosition = get(hObject, 'Position');
    % Sets the position to be directly centered on the main figure
    newX = parentPosition(1) + (parentPosition(3)/2 ...
         - currentPosition(3)/2);
    newY = parentPosition(2) + (parentPosition(4)/2 ...
         - currentPosition(4)/2);
    newW = currentPosition(3);
    newH = currentPosition(4);

    set(hObject, 'Position', [newX, newY, newW, newH]);
.
```

```
.
.
```

Initializing the Text Change Dialog's Text

Initialize the Text Change dialog's text to the **Change Me** button's current text. From the main GUI's handle that was passed to the modal dialog, get the main GUI's handles structure. From that, get the **Change Me** button and get its `String` property. Then set the `String` property to the edit box's value in the dialog's `OpeningFcn`:

```
function changeme_dialog_OpeningFcn(hObject, ...
         eventdata, handles, varargin)

mainGuiInput = find(strcmp(varargin, 'changeme_main'));
.
.
.
% Remember the handle, and adjust our position
handles.changeMeMain = varargin{mainGuiInput+1};

% Set the initial text
mainHandles = guidata(handles.changeMeMain);
set(handles.editChangeMe, 'String',
    get(mainHandles.buttonChangeMe, 'String'));
.
.
.
```

Canceling the Text Change Dialog

If **Cancel** is clicked or the window is closed, do not modify the main GUI. There is really nothing to do, other than to call `uiresume` to close the modal dialog:

```
function buttonCancel_Callback(hObject, ...
         eventdata, handles)
uiresume(handles.figure);

function figure_CloseRequestFcn(hObject, ...
         eventdata, handles)
```

```
uiresume(hObject);
```

Applying the Text Change

If **OK** is clicked, set the main GUI's **Change Me** button label to the value of the textbox. This is where the main GUI gets modified. The modal dialog's `OpeningFcn` saved the reference to the main GUI in the handles structure. Now use that reference to get the main GUI's handles, and from that get the button's handle and modify its text:

```
function buttonOK_Callback(hObject, ...
         eventdata, handles)
text = get(handles.editChangeMe, 'String');
main = handles.changeMeMain;
mainHandles = guidata(main);
changeMeButton = mainHandles.buttonChangeMe;
set(changeMeButton, 'String', text);
uiresume(handles.figure);
```

Example — Individual GUIDE GUIs that Work Together as an Application

The following example demonstrates creating an icon editor application in GUIDE. The editor consists of three GUIs:

- The drawing area (Icon Editor)
- The tool selection toolbar (Tool Palette)
- The color picker (Color Palette)

These GUIs share data and expose functionality to one another using several different techniques.

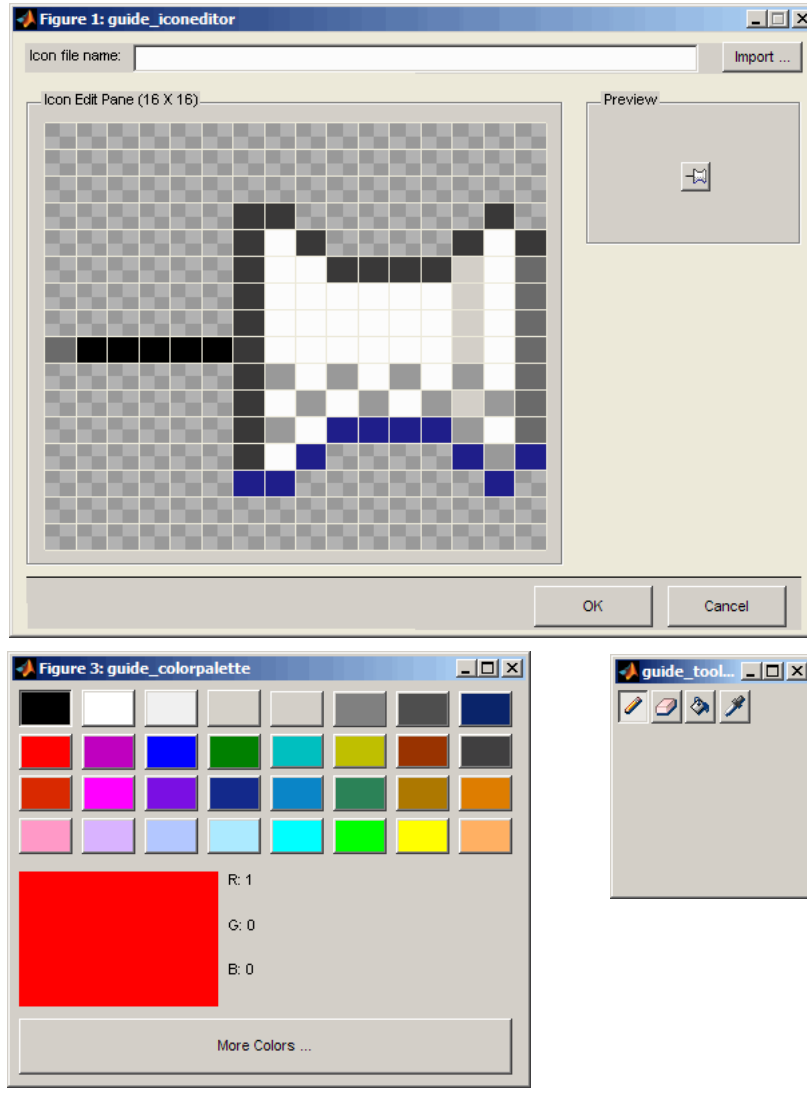


Figure 1: guide_iconeditor
Icon file name:
Import ...
Icon Edit Pane (16 X 16)
Preview
OK
Cancel
Figure 3: guide_colorpalette
R: 1
G: 0
B: 0
More Colors ...
guide_tool...

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this on the Web or in the PDF, go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display the icon editor GUIs in the Layout Editor.
- Click here to display the icon editor GUI M-files in the MATLAB editor.

Requirements for the GUIs

The Icon Editor application needs to behave as follows:

- When starting Icon Editor, create the Tool Palette and Color Palette.
- Set the initial color on the Color Palette when Icon Editor starts.
- Give the Icon Editor access to the Color Palette's current color.
- When clicking in the editing area, apply the currently selected tool from the Tool Palette.
- When the mouse pointer is over the edit area, display the current tool's cursor
- Close all windows only when the Icon Editor completes.

Click any item above to jump to that section.

M-file Implementations

This application uses three M-files and FIG-files that were fully implemented in GUIDE. You can modify and enhance them in the GUIDE environment should you choose to do so. The FIG-files are:

- `guide_iconeditor.fig` — Main GUI, for drawing and modifying icon files
- `guide_colorpalette.fig` — Palette for selecting a current color
- `guide_toolpalette.fig` — Palette for selecting one of four editing tools

The associated M-files contain the following functions and signatures:

- guide_iconeditor.m

```
guide_iconeditor(varargin)
guide_iconeditor_OpeningFcn(hObject, eventdata, handles, varargin)
guide_iconeditor_OutputFcn(hObject, eventdata, handles)
editFilename_CreateFcn(hObject, eventdata, handles)
buttonImport_Callback(hObject, eventdata, handles)
buttonOK_Callback(hObject, eventdata, handles
buttonCancel_Callback(hObject, eventdata, handles)
editFilename_ButtonDownFcn(hObject, eventdata, handles)
editFilename_Callback(hObject, eventdata, handles)
figure_CloseRequestFcn(hObject, eventdata, handles)
figure_WindowButtonDownFcn(hObject, eventdata, handles)
figure_WindowButtonUpFcn(hObject, eventdata, handles)
figure_WindowButtonMotionFcn(hObject, eventdata, handles)
getToolPalette(handles)
getColorPalette(handles)
setColor(hObject, color)
getColor(hObject)
updateCursor(hObject, overicon)
applyCurrentTool(handles)
localUpdateIconPlot(handles)
localGetIconCDataWithNaNs(handles)
```

- guide_colorpalette.m

```
guide_colorpalette(varargin)
guide_colorpalette_OpeningFcn(hObject, eventdata, handles, varargin)
guide_colorpalette_OutputFcn(hObject, eventdata, handles)
buttonMoreColors_Callback(hObject, eventdata, handles)
colorCellCallback(hObject, eventdata, handles)
figure_CloseRequestFcn(hObject, eventdata, handles)
localUpdateColor(handles)
setSelectedColor(hObject, color)
```

- guide_toolPalatte.m

```
guide_toolpalette(varargin)
guide_toolpalette_OpeningFcn(hObject, eventdata, handles, varargin)
guide_toolpalette_OutputFcn(hObject, eventdata, handles)
toolPencil_CreateFcn(hObject, eventdata, handles)
```

```
toolEraser_CreateFcn(hObject, eventdata, handles)
toolBucket_CreateFcn(hObject, eventdata, handles)
toolPicker_CreateFcn(hObject, eventdata, handles)
toolPalette_SelectionChangeFcn(hObject, eventdata, handles)
figure_CloseRequestFcn(hObject, eventdata, handles)
getIconEditor(handles)
pencilToolCallback(handles, toolstruct, cdata, point)
eraserToolCallback(handles, toolstruct, cdata, point)
bucketToolCallback(handles, toolstruct, cdata, point)
fillWithColor(cdata, rows, cols, color, row, col, seedcolor)
colorpickerToolCallback(handles, toolstruct, cdata, point)
```

1. When Icon Editor launches, create the Tool Palette and Color Palette

Starting the Icon Editor GUI should launch the Tool Palette and Color Palette. These GUIs are its children. The parent and children communicate using the following techniques:

- Property/Value pairs — Send data into a newly-invoked or existing GUI by passing it as input arguments.
- Guidata — Store data in the handles structure of a GUI; this can communicate data within one GUI or across several of them.
- Output — Returned data from the invoked GUI; this is used to communicate data, such as the handles structure of the invoked GUI, back to the invoking GUI.

The Icon Editor is passed into the Tool Palette and Color Palette as a Property/Value (p/v) pair in order to let the Tool Palette make calls back into Icon Editor. The output value from calling both of the palettes is the handle to their GUI figures. These figure handles are saved into the handles structure of Icon Editor:

```
% in Icon Editor
function guide_Icon Editor_OpeningFcn(hObject, ...
         eventdata, handles, varargin)
.
```

```
.
.
handles.colorPalette = guide_colorpalette('iconEditor', hObject);
handles.toolPalette = guide_toolpalette('iconEditor', hObject);
.
.
.
% Update handles structure
guidata(hObject, handles);
```

The Color Palette needs to remember the Icon Editor for later:

```
% in colorPalette
function guide_colorpalette_OpeningFcn(hObject, ...
         eventdata, handles, varargin)
handles.output = hObject;
.
.
.
handles.iconEditor = [];

iconEditorInput = find(strcmp(varargin, 'iconEditor'));
if ~isempty(iconEditorInput)
   handles.iconEditor = varargin{iconEditorInput+1};
end
.
.
.
% Update handles structure
guidata(hObject, handles);
```

The Tool Palette also needs to remember the Icon Editor:

```
% in toolPalette
function guide_toolpalette_OpeningFcn(hObject, ...
         eventdata, handles, varargin)
handles.output = hObject;
.
.
.
handles.iconEditor = [];
```

```
iconEditorInput = find(strcmp(varargin, 'iconEditor'));
if ~isempty(iconEditorInput)
   handles.iconEditor = varargin{iconEditorInput+1};
end
.
.
.
% Update handles structure
guidata(hObject, handles);
```

2. Set the initial color on the Color Palette when the Icon Editor starts

After all three GUIs have been created, set the initial color. When the Color Palette is invoked from the Icon Editor, the Color Palette needs to tell the Icon Editor how to set the initial color and provides the functionality via a function handle, which it stores in its handles structure. Color Palette outputs the handle to its figure, from which its handles structure can be retrieved:

```
% in colorPalette
function guide_colorpalette_OpeningFcn(hObject, ...
         eventdata, handles, varargin)
handles.output = hObject;
.
.
.
% Set the initial palette color to black
handles.mSelectedColor = [0 0 0];

% Publish the function setSelectedColor
handles.setColor = @setSelectedColor;
.
.
.
% Update handles structure
guidata(hObject, handles);


% in colorPalette
```

```
function setSelectedColor(hObject, color)
handles = guidata(hObject);
.
.
.
handles.mSelectedColor =color;
.
.
.
guidata(hObject, handles);
```

Call the publicized function from the Icon Editor, setting the initial color to `'red'`:

```
% in Icon Editor
function guide_iconeditor_OpeningFcn(hObject, ...
         eventdata, handles, varargin)
.
.
.
handles.colorPalette = guide_colorpalette('iconEditor', hObject);
.
.
.
colorPalette = handles.colorPalette;
colorPaletteHandles = guidata(colorPalette);
colorPaletteHandles.setColor(colorPalette, [1 0 0]);
.
.
.
% Update handles structure
guidata(hObject, handles);
```

3. Give the Icon Editor access to the Color Palette's current color

The Color Palette initializes the current color data:

```
%in colorPalette
function guide_colorpalette_OpeningFcn(hObject, ...
         eventdata, handles, varargin)
handles.output = hObject;
```

```
.
.
.
handles.mSelectedColor = [0 0 0];
.
.
.
% Update handles structure
guidata(hObject, handles);
```

The Icon Editor retrieves the inital color from the Color Palette's guidata via its handles structure:

```
% in Icon Editor
function color = getColor(hObject)
handles = guidata(hObject);
colorPalette = handles.colorPalette;
colorPaletteHandles = guidata(colorPalette);
color = colorPaletteHandles.mSelectedColor;
```

4. When clicking in the editing area, apply the currently selected tool from the Tool Palette

This example demonstrates how the `UserData` property of components in your GUIDE GUI can be used to share data. Every tool in the Tool Palette can modify the icon being edited, altering `CData` whatever tool is selected when the mouse is clicked in the icon editing area. The `UserData` property of each tool is used to record the function called when a tool is selected and applied to the icon editing area. Different tools do different things to the icon data. The following code shows how the *pencil tool* works.

In the `CreateFcn` for the pencil button, add the user data that points to the function for the pencil tool:

```
% in toolPalette
function toolPencil_CreateFcn(hObject, eventdata, handles)
set(hObject,'UserData', struct('Callback', @pencilToolCallback));
```

The currently selected tool is tracked by the Tool Palette in a field in its handles structure called `mCurrentTool`, which you can get from other GUIs

once you have the handles structure of the Tool Palette. The currently selected tool is set by calling `guidata` after you click a button in the Tool Palette:

```
% in toolPalette
function toolPalette_SelectionChangeFcn(hObject, ...
        eventdata, handles)
handles.mCurrentTool = hObject;
guidata(hObject, handles);
```

When you select the pencil tool and click in the icon editing area, the function of the pencil tool is eventually called by the Icon Editor:

```
% in iconEditor
function iconEditor_WindowButtonDownFcn(hObject,...
         eventdata, handles)
toolPalette = handles.toolPalette;
toolPaletteHandles = guidata(toolPalette);
 .
 .
 .
userData = get(toolPaletteHandles.mCurrentTool, 'UserData');
handles.mIconCData = userData.Callback(toolPaletteHandles, ...
        toolPaletteHandles.mCurrentTool, handles.mIconCData, ...);
```

If you are curious about what the pencil tool does, here is the code that shows how the pixel value in the icon editing area under the mouse click (the Tool icon's `CData`) is changed to the color currently selected in the Color Palette:

```
% in toolPalette
function cdata = pencilToolCallback(handles, toolstruct, cdata,...)
iconEditor = handles.iconEditor;
iconEditorHandles = guidata(iconEditor);
x = ...
y = ...
% update color of the selected block
color = iconEditorHandles.getColor(iconEditor);
cdata(y, x,:) = color;
```

5. When mouse pointer is in the edit area, display the current tool's cursor

Icon Editor must set the cursor with every mouse motion. If the mouse is not in the editing area, the pointer is the default arrow. Otherwise, it displays the currently selected tool's pointer icon. Identify the selected tool through the Tool Palette's handles:

```
% in Icon Editor
function iconEditor_WindowButtonMotionFcn(hObject, ...
        eventdata, handles)
.
.
.
rows = size(handles.mIconCData,1);
cols = size(handles.mIconCData,2);
pt = get(handles.icon,'currentpoint');
overicon = (pt(1,1)>=0 && pt(1,1)<=rows) && ...
           (pt(1,2)>=0 && pt(1,2)<=cols);
.
.
.
if ~overicon
    set(hObject,'pointer','arrow');
else
    toolPalette = handles.toolPalette;
    toolPaletteHandles = guidata(toolPalette);
    tool = toolPaletteHandles.mCurrentTool;
    cdata = round(mean(get(tool, 'cdata'),3))+1;
    if ~isempty(cdata)
        set(hObject,'pointer','custom','PointerShapeCData', ...
        cdata(1:16, 1:16),'PointerShapeHotSpot',[16 1]);
    end
end
.
.
.
```

6. Close all windows only when the Icon Editor completes

When launching Icon Editor, Color Palette and Tool Palette were also invoked and remembered within the handles structure of Icon Editor. However, Icon Editor also invokes uiwait to defer output until the GUI is finished, which complicates the shutdown sequence. Furthermore, neither the Color Palette nor Tool Palette is permitted to close independently of Icon Editor shutdown. The only ways out are the **OK** button, the **Cancel** button, or closing the Icon Editor's window directly. Closing the Color Palette and Tool Palette windows (by clicking their `X` box) has to be blocked.

Finally, upon closing, set the output of Icon Editor to be the `cdata` of the icon. The opening function for Icon Editor, with `uiwait`, contains this code:

```
% in Icon Editor
function guide_iconeditor_OpeningFcn(hObject, eventdata, ...
        handles, varargin)

.
.
.
handles.colorPalette = guide_colorpalette();
handles.toolPalette = guide_toolpalette('iconEditor', hObject);
.
.
.
% Update handles structure
guidata(hObject, handles);
uiwait(hObject);
```

Because Icon Editor calls `uiwait` to begin with, `uiresume` must be called on each exit path:

```
% in Icon Editor
function buttonOK_Callback(hObject, eventdata, handles)
uiresume(handles.figure);

function buttonCancel_Callback(hObject, eventdata, handles)
% Make sure the return data will be empty if we cancelled
handles.mIconCData =[];
guidata(handles.figure, handles);
```

```
uiresume(handles.figure);

function Icon Editor_CloseRequestFcn(hObject, eventdata, handles)
uiresume(hObject);
```

To ensure that the Color Palette is not closed any other way, override its `closerequestfcn` to do nothing:

```
% in colorPalette
function figure_CloseRequestFcn(hObject, eventdata, handles)
% Don't close this figure. It must be deleted from Icon Editor
```

Do the same for Tool Palette:

```
% in toolPalette
function figure_CloseRequestFcn(hObject, eventdata, handles)
% Don't close this figure. It must be deleted from Icon Editor
```

Finally, in the output function, destroy all three GUIs:

```
% in Icon Editor
function varargout = guide_iconeditor_OutputFcn(hObject, ...
        eventdata, handles)
% Return the cdata of the icon. If cancelled, this will be empty
varargout{1} = handles.mIconCData;
delete(handles.toolPalette);
delete(handles.colorPalette);
delete(hObject);
```

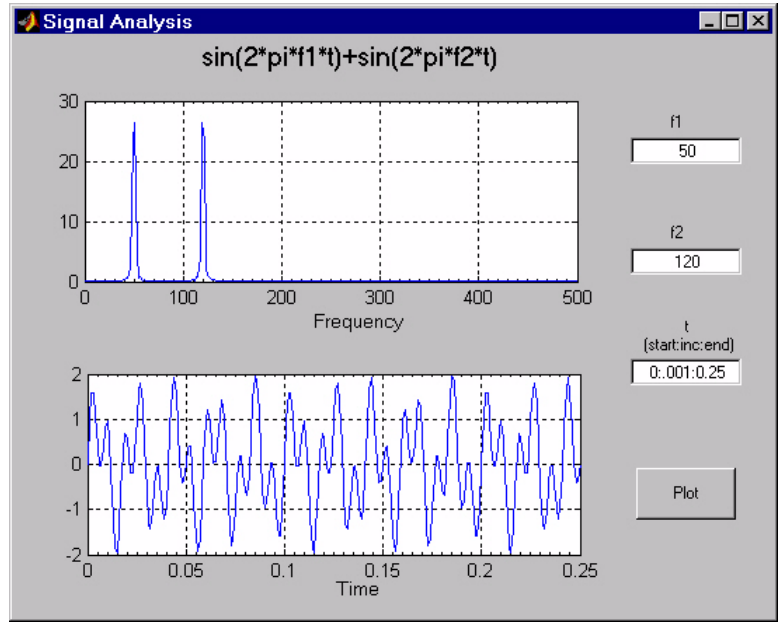
Examples of GUIDE GUIs

GUI with Multiple Axes

In this section...

Multiple Axes Example Outcome

This example creates a GUI that contains two axes for plotting data. For simplicity, this example obtains data by evaluating an expression using parameters entered by the user.



Techniques Used in the Example

GUI-building techniques illustrated in this example include

- Controlling which axes is the target for plotting commands.
- Using edit text controls to read numeric input and MATLAB expressions.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the MATLAB Editor.

Design of the GUI

This GUI requires three input values:

- Frequency one (`f1`)
- Frequency two (`f2`)
- A time vector (`t`)

When the user clicks the **Plot** button, the GUI puts these values into a MATLAB expression that is the sum of two sine function:

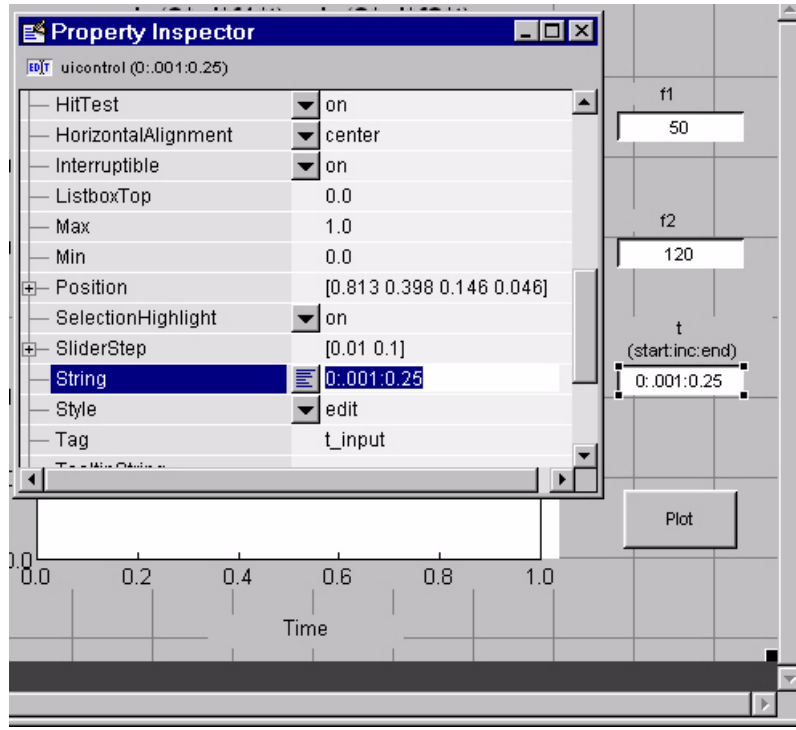
```
x = sin(2*pi*f1*t) + sin(2*pi*f2*t)
```

The GUI then calculates the FFT of `x` and creates two plots — one frequency domain and one time domain.

Specifying Default Values for the Inputs

The GUI uses default values for the three inputs. This enables users to click on the **Plot** button and see a result as soon as the GUI is run. It also helps to indicate what values the user might enter.

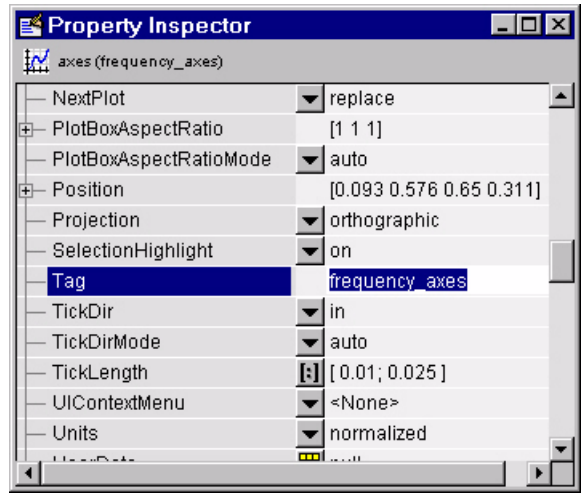
To create the default values, set the `String` property of the edit text. The following figure shows the value set for the time vector.



Identifying the Axes

Since there are two axes in this GUI, you must be able to specify which one you want to target when you issue the plotting commands. To do this, use the `handles` structure, which contains the handles of all components in the GUI.

The field name in the `handles` structure that contains the handle of any given component is derived from the component's `Tag` property. To make code more readable (and to make it easier to remember) this example sets the `Tag` property to descriptive names.



For example, the `Tag` of the axes used to display the FFT is set to `frequency_axes`. Therefore, within a callback, you access its handle with

```
handles.frequency_axes
```

Likewise, the `Tag` of the time axes is set to `time_axes`.

See "handles Structure" on page 8-15 for more information on the `handles` structure. See "Plot Push Button Callback" on page 10-6 for the details of how to use the handle to specify the target axes.

GUI Option Settings

There are two GUI option settings that are particularly important for this GUI:

- Resize behavior: **Proportional**
- Command-line accessibility: **Callback**

Proportional Resize Behavior. Selecting **Proportional** as the resize behavior enables users to change the GUI to better view the plots. The components change size in proportion to the GUI figure size. This generally produces good results except when extremes of dimensions are used.

Callback Accessibility of Object Handles. When GUIs include axes, handles should be visible from within callbacks. This enables you to use plotting commands like you would on the command line. Note that **Callback** is the default setting for command-line accessibility.

See "GUI Options" on page 5-9 for more information.

Plot Push Button Callback

This GUI uses only the **Plot** button callback; the edit text callbacks are not needed and have been deleted from the GUI M-file. When a user clicks the **Plot** button, the callback performs three basic tasks — it gets user input from the edit text components, calculates data, and creates the two plots.

Getting User Input

The three edit text boxes provide a way for the user to enter values for the two frequencies and the time vector. The first task for the callback is to read these values. This involves:

- Reading the current values in the three edit text boxes using the `handles` structure to access the edit text handles.
- Converting the two frequency values (`f1` and `f2`) from string to doubles using `str2double`.
- Evaluating the time string using `eval` to produce a vector `t`, which the callback used to evaluate the mathematical expression.

The following code shows how the callback obtains the input.

```
% Get user input from GUI
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));
```

Calculating Data

Once the input data has been converted to numeric form and assigned to local variables, the next step is to calculate the data needed for the plots. See the `fft` function for an explanation of how this is done.

Targeting Specific Axes

The final task for the callback is to actually generate the plots. This involves

- Making the appropriate axes current using the `axes` command and the handle of the axes. For example,

  ```
  axes(handles.frequency_axes)
  ```

- Issuing the `plot` command.
- Setting any properties that are automatically reset by the `plot` command.

The last step is necessary because many plotting commands (including `plot`) clear the axes before creating the graph. This means you cannot use the Property Inspector to set the `XMinorTick` and grid properties that are used in this example, since they are reset when the callback executes `plot`.

When looking at the following code listing, note how the `handles` structure is used to access the handle of the axes when needed.

Plot Button Code Listing

```
function plot_button_Callback(hObject, eventdata, handles)
% hObject    handle to plot_button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get user input from GUI
f1 = str2double(get(handles.f1_input,'String'));
f2 = str2double(get(handles.f2_input,'String'));
t = eval(get(handles.t_input,'String'));

% Calculate data
x = sin(2*pi*f1*t) + sin(2*pi*f2*t);
```

```
y = fft(x,512);
m = y.*conj(y)/512;
f = 1000*(0:256)/512;

% Create frequency plot
axes(handles.frequency_axes) % Select the proper axes
plot(f,m(1:257))
set(handles.frequency_axes,'XMinorTick','on')
grid on

% Create time plot
axes(handles.time_axes) % Select the proper axes
plot(t,x)
set(handles.time_axes,'XMinorTick','on')
grid on
```

List Box Directory Reader

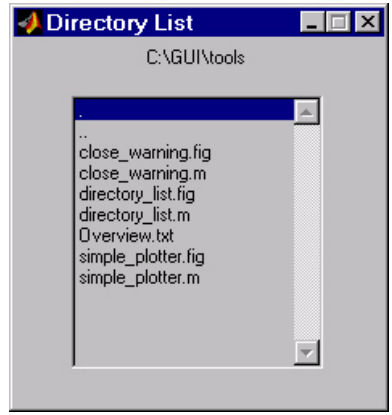
In this section...

List Box Example Outcome

This example uses a list box to display the files in a directory. When the user double clicks on a list item, one of the following happens:

- If the item is a file, the GUI opens the file appropriately for the file type.
- If the item is a directory, the GUI reads the contents of that directory into the list box.
- If the item is a single dot (**.**), the GUI updates the display of the current directory.
- If the item is two dots (**..**), the GUI changes to the directory up one level and populates the list box with the contents of that directory.

The following figure illustrates the GUI.



View Layout and GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

Implementing the GUI

The following sections describe the implementation:

- “Specifying the Directory to List” on page 10-11 — shows how to pass a directory path as input argument when the GUI is run.

- "Loading the List Box" on page 10-12 — describes the subfunction that loads the contents of the directory into the list box. This subfunction also saves information about the contents of a directory in the `handles` structure.
- "The List Box Callback" on page 10-13 — explains how the list box is programmed to respond to user double clicks on items in the list box.

Specifying the Directory to List

You can specify the directory to list when the GUI is first opened by passing the string `'create'` and a string containing the full path to the directory as arguments. The syntax for doing this is `list_box('create','dir_path')`. If you do not specify a directory (i.e., if you call the GUI M-file with no input arguments), the GUI then uses the MATLAB current directory.

The default behavior of the GUI M-file that GUIDE generates is to run the GUI when there are no input arguments or to call a subfunction when the first input argument is a character string. This example changes this behavior so that you can call the M-file with

- No input arguments — run the GUI using the MATLAB current directory.
- First input argument is `'dir'` and second input argument is a string that specifies a valid path to a directory — run the GUI, displaying the specified directory.
- First input argument is not a directory, but is a character string and there is more than one argument — execute the subfunction identified by the argument (execute callback).

The following code listing show the setup section of the GUI M-file, which does one the following:

- Sets the list box directory to the current directory, if no directory is specified.
- Changes the current directory, if a directory is specified.

```
function lbox2_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to untitled (see VARARGIN)
```

```
% Choose default command line output for lbox2
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

if nargin == 3,
    initial_dir = pwd;
elseif nargin > 4
    if strcmpi(varargin{1},'dir')
        if exist(varargin{2},'dir')
            initial_dir = varargin{2};
        else
            errordlg({'Input argument must be a valid',...
                      'directory'},'Input Argument Error!')
            return
        end
    else
        errordlg('Unrecognized input argument',...
                 'Input Argument Error!');
        return;
    end
end
% Populate the listbox
load_listbox(initial_dir,handles)
```

Loading the List Box

This example creates a subfunction to load items into the list box. This subfunction accepts the path to a directory and the `handles` structure as input arguments. It performs these steps:

- Change to the specified directory so the GUI can navigate up and down the tree as required.
- Use the `dir` command to get a list of files in the specified directory and to determine which name is a directory and which is a file. `dir` returns a structure (`dir_struct`) with two fields, `name` and `isdir`, which contain this information.

- Sort the file and directory names (`sortrows`) and save the sorted names and other information in the `handles` structure so this information can be passed to other functions.

  The `name` structure field is passed to `sortrows` as a cell array, which is transposed to get one file name per row. The `isdir` field and the sorted index values, `sorted_index`, are saved as vectors in the `handles` structure.

- Call `guidata` to save the `handles` structure.
- Set the list box `String` property to display the file and directory names and set the `Value` property to `1`. This is necessary to ensure `Value` never exceeds the number of items in `String`, since MATLAB updates the `Value` property only when a selection occurs and not when the contents of `String` changes.
- Displays the current directory in the text box by setting its `String` property to the output of the `pwd` command.

The `load_listbox` function is called by the opening function of the GUI M-file as well as by the list box callback.

```
function load_listbox(dir_path, handles)
cd (dir_path)
dir_struct = dir(dir_path);
[sorted_names,sorted_index] = sortrows({dir_struct.name}');
handles.file_names = sorted_names;
handles.is_dir = [dir_struct.isdir];
handles.sorted_index = sorted_index;
guidata(handles.figure1,handles)
set(handles.listbox1,'String',handles.file_names,...
 'Value',1)
set(handles.text1,'String',pwd)
```

The List Box Callback

The list box callback handles only one case: a double-click on an item. Double clicking is the standard way to open a file from a list box. If the selected item is a file, it is passed to the `open` command; if it is a directory, the GUI changes to that directory and lists its contents.

Defining How to Open File Types

The callback makes use of the fact that the `open` command can handle a number of different file types. However, the callback treats FIG-files differently. Instead of opening the FIG-file, it passes it to the `guide` command for editing.

Determining Which Item the User Selected

Since a single click on an item also invokes the list box callback, it is necessary to query the figure `SelectionType` property to determine when the user has performed a double click. A double-click on an item sets the `SelectionType` property to `open`.

All the items in the list box are referenced by an index from `1` to `n`, where `1` refers to the first item and `n` is the index of the nth item. MATLAB saves this index in the list box `Value` property.

The callback uses this index to get the name of the selected item from the list of items contained in the `String` property.

Determining if the Selected Item is a File or Directory

The `load_listbox` function uses the `dir` command to obtain a list of values that indicate whether an item is a file or directory. These values (1 for directory, 0 for file) are saved in the `handles` structure. The list box callback queries these values to determine if current selection is a file or directory and takes the following action:

- If the selection is a directory — change to the directory (`cd`) and call `load_listbox` again to populate the list box with the contents of the new directory.
- If the selection is a file — get the file extension (`fileparts`) to determine if it is a FIG-file, which is opened with `guide`. All other file types are passed to `open`.

The `open` statement is called within a `try/catch` block to capture errors in an error dialog (`errordlg`), instead of returning to the command line.

```
get(handles.figure1,'SelectionType');
% If double click
```

```
if strcmp(get(handles.figure1,'SelectionType'),'open')
    index_selected = get(handles.listbox1,'Value');
    file_list = get(handles.listbox1,'String');
    % Item selected in list box
    filename = file_list{index_selected};
    % If directory
    if  handles.is_dir(handles.sorted_index(index_selected))
        cd (filename)
        % Load list box with new directory.
        load_listbox(pwd,handles)
    else
        [path,name,ext,ver] = fileparts(filename);
        switch ext
            case '.fig'
                % Open FIG-file with guide command.
                guide (filename)
            otherwise
                try
                    % Use open for other file types.
                    open(filename)
                catch
                    errordlg(lasterr,'File Type Error','modal')
                end
        end
    end
end
```

Opening Unknown File Types

You can extend the file types that the `open` command recognizes to include any file having a three-character extension. You do this by creating an M-file with the name `openxyz`, where `xyz` is the extension. Note that the list box callback does not take this approach for FIG-files since `openfig.m` is required by the GUI M-file. See `open` for more information.

Accessing Workspace Variables from a List Box

In this section...

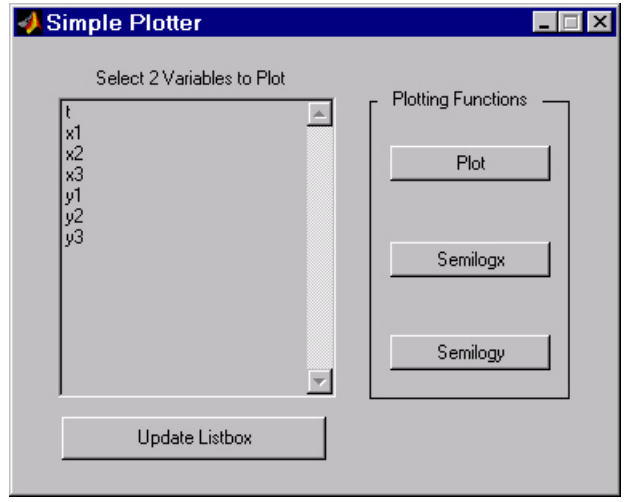
Workspace Variable Example Outcome

This GUI uses a list box to display workspace variables, which the user can then plot.

Techniques Used in This Example

- Populate the list box with the variable names that exist in the base workspace.
- Display the list box with no items initially selected.
- Enable multiple item selection in the list box.
- Update the list items when the user press a button.
- Evaluate the plotting commands in the base workspace.

The following figure illustrates the layout.



Note that the list box callback is not used in this program because the plotting actions are initiated by push buttons. In this situation you must do one of the following:

- Leave the empty list box callback in the GUI M-file.
- Delete the string assigned to the list box `Callback` property.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

Reading Workspace Variables

When the GUI initializes, it needs to query the workspace variables and set the list box `String` property to display these variable names. Adding the following subfunction to the GUI M-file accomplishes this using `evalin` to execute the `who` command in the base workspace. The `who` command returns a cell array of strings, which are used to populate the list box.

```
function update_listbox(handles)
vars = evalin('base','who');
set(handles.listbox1,'String',vars)
```

The function's input argument is the handles structure generated by the GUI M-file. This structure contains the handle of the list box, as well as the handles all other components in the GUI.

The callback for the **Update Listbox** push button also calls `update_listbox`.

Reading the Selections from the List Box

This GUI requires the user to select two variables from the workspace and then choose one of three plot commands to create the graph: `plot`, `semilogx`, or `semilogy`.

Enabling Multiple Selection

To enable multiple selection in a list box, you must set the `Min` and `Max` properties so that `Max - Min > 1`. This requires you to change the default `Min` and `Max` values of `0` and `1` to meet these conditions. Use the Property Inspector to set these properties on the list box.

How Users Select Multiple Items

List box multiple selection follows the standard for most systems:

- **Ctrl**+click left mouse button — noncontiguous multi-item selection
- **Shift**+click left mouse button — contiguous multi-item selection

Users must use one of these techniques to select the two variables required to create the plot.

Returning Variable Names for the Plotting Functions

The `get_var_names` subroutine returns the two variable names that are selected when the user clicks on one of the three plotting buttons. The function

- Gets the list of all items in the list box from the `String` property.
- Gets the indices of the selected items from the `Value` property.
- Returns two string variables, if there are two items selected. Otherwise `get_var_names` displays an error dialog explaining that the user must select two variables.

Here is the code for `get_var_names`:

```
function [var1,var2] = get_var_names(handles)
list_entries = get(handles.listbox1,'String');
index_selected = get(handles.listbox1,'Value');
if length(index_selected) ~= 2
 errordlg('You must select two variables',...
   'Incorrect Selection','modal')
else
 var1 = list_entries{index_selected(1)};
 var2 = list_entries{index_selected(2)};
end
```

Callbacks for the Plotting Buttons

The callbacks for the plotting buttons call `get_var_names` to get the names of the variables to plot and then call `evalin` to execute the plot commands in the base workspace.

For example, here is the callback for the `plot` function:

```
function plot_button_Callback(hObject, eventdata, handles)
[x,y] = get_var_names(handles);
evalin('base',['plot(' x ',' y ')'])
```

The command to evaluate is created by concatenating the strings and variables that result in the command:

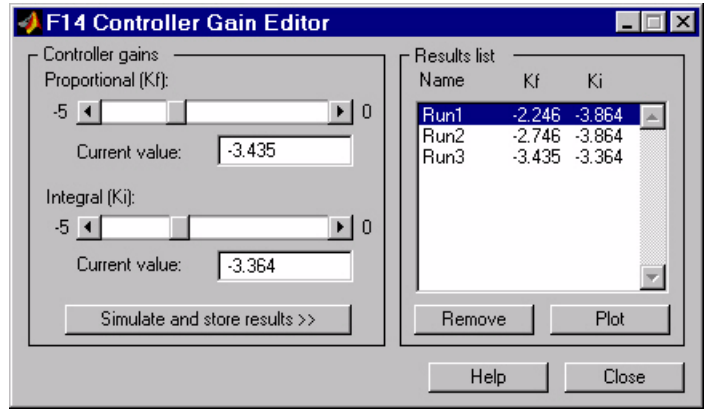
```
plot(x,y)
```

A GUI to Set Simulink Model Parameters

In this section...

Set Simulink Model Parameters Example Outcome

This example illustrates how to create a GUI that sets the parameters of a Simulink® model. In addition, the GUI can run the simulation and plot the results. The following picture shows the GUI after running three simulations with different values for controller gains.



Techniques Used in This Example

This example illustrates a number of GUI building techniques:

- Opening and setting parameters on a Simulink model from a GUI.
- Implementing sliders that operate in conjunction with text boxes, which display the current value as well as accepting user input.
- Enabling and disabling controls, depending on the state of the GUI.
- Managing a variety of shared data using the `handles` structure.
- Directing graphics output to figures with hidden handles.
- Adding a help button that displays `.html` files in the MATLAB Help browser.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the editor.

How to Use the GUI (Text of GUI Help)

You can use the F14 Controller Gain Editor to analyze how changing the gains used in the Proportional-Integral Controller affect the aircraft's angle of attack and the amount of G force the pilot feels.

Note that the Simulink diagram `f14.mdl` must be open to run this GUI. If you close the F14 Simulink model, the GUI reopens it whenever it requires the model to execute.

Changing the Controller Gains

You can change gains in two blocks:

- The Proportional gain (Kf) in the Gain block
- The Integral gain (Ki) in the Transfer Function block

You can change either of the gains in one of the two ways:

- Move the slider associated with that gain.
- Type a new value into the **Current value** edit field associated with that gain.

The block's values are updated as soon as you enter the new value in the GUI.

Running the Simulation

Once you have set the gain values, you can run the simulation by clicking the **Simulate and store results** button. The simulation time and output vectors are stored in the **Results list**.

Plotting the Results

You can generate a plot of one or more simulation results by selecting the row of results (`Run1`, `Run2`, etc.) in the **Results list** that you want to plot and clicking the **Plot** button. If you select multiple rows, the graph contains a plot of each result.

The graph is displayed in a figure, which is cleared each time you click the **Plot** button. The figure's handle is hidden so that only the GUI can display graphs in this window.

Removing Results

To remove a result from the **Results list**, select the row or rows you want to remove and click the **Remove** button.

Running the GUI

The GUI is nonblocking and nonmodal since it is designed to be used as an analysis tool.

GUI Options Settings

This GUI uses the following GUI option settings:

- Resize behavior: **Non-resizable**
- Command-line accessibility: **Off**
- M-file options selected:
  - Generate callback function prototypes
  - GUI allows only one instance to run

Opening the Simulink Block Diagrams

This example is designed to work with the F14 Simulink model. Since the GUI sets parameters and runs the simulation, the F14 model must be open when the GUI is displayed. When the GUI M-file runs the GUI, it executes the `model_open` subfunction. The purpose of the subfunction is to

- Determine if the model is open (`find_system`).
- Open the block diagram for the model and the subsystem where the parameters are being set, if not open already (`open_system`).
- Change the size of the controller Gain block so it can display the gain value (`set_param`).
- Bring the GUI forward so it is displayed on top of the Simulink diagrams (`figure`).
- Set the block parameters to match the current settings in the GUI.

Here is the code for the `model_open` subfunction.

```
function model_open(handles)
if  isempty(find_system('Name','f14')),
 open_system('f14'); open_system('f14/Controller')
 set_param('f14/Controller/Gain','Position',[275 14 340 56])
 figure(handles.F14ControllerEditor)
 set_param('f14/Controller Gain','Gain',...
  get(handles.KfCurrentValue,'String'))
 set_param(...
      'f14/Controller/Proportional plus integral compensator',...
      'Numerator',...
      get(handles.KiCurrentValue,'String'))
end
```

Programming the Slider and Edit Text Components

This GUI employs a useful combination of components in its design. Each slider is coupled to an edit text component so that:

- The edit text displays the current value of the slider.
- The user can enter a value into the edit text box and cause the slider to update to that value.

- Both components update the appropriate model parameters when activated by the user.

Slider Callback

The GUI uses two sliders to specify block gains since these components enable the selection of continuous values within a specified range. When a user changes the slider value, the callback executes the following steps:

- Calls model_open to ensure that the Simulink model is open so that simulation parameters can be set.
- Gets the new slider value.
- Sets the value of the **Current value** edit text component to match the slider.
- Sets the appropriate block parameter to the new value (`set_param`).

Here is the callback for the **Proportional (Kf)** slider.

```
function KfValueSlider_Callback(hObject, eventdata, handles)
% Ensure model is open.
model_open(handles)
% Get the new value for the Kf Gain from the slider.
NewVal = get(hObject, 'Value');
% Set the value of the KfCurrentValue to the new value
% set by slider.
set(handles.KfCurrentValue,'String',NewVal)
% Set the Gain parameter of the Kf Gain Block to the new value.
set_param('f14/Controller/Gain','Gain',num2str(NewVal))
```

Note that, while a slider returns a number and the edit text requires a string, uicontrols automatically convert the values to the correct type.

The callback for the **Integral (Ki)** slider follows a similar approach.

Current Value Edit Text Callback

The edit text box enables users to type in a value for the respective parameter. When the user clicks on another component in the GUI after typing into the text box, the edit text callback executes the following steps:

- Calls model_open to ensure that the Simulink model is open so that it can set simulation parameters.
- Converts the string returned by the edit box `String` property to a double (`str2double`).
- Checks whether the value entered by the user is within the range of the slider:

  If the value is out of range, the edit text `String` property is set to the value of the slider (rejecting the number typed in by the user).

  If the value is in range, the slider `Value` property is updated to the new value.
- Sets the appropriate block parameter to the new value (`set_param`).

Here is the callback for the Kf **Current value** text box.

```
function KfCurrentValue_Callback(hObject, eventdata, handles)
% Ensure model is open.
model_open(handles)
% Get the new value for the Kf Gain.
NewStrVal = get(hObject, 'String');
NewVal = str2double(NewStrVal);
% Check that the entered value falls within the allowable range.
if  isempty(NewVal) || (NewVal< -5) || (NewVal>0),
   % Revert to last value, as indicated by KfValueSlider.
   OldVal = get(handles.KfValueSlider,'Value');
   set(hObject, 'String',OldVal)
else % Use new Kf value
   % Set the value of the KfValueSlider to the new value.
   set(handles.KfValueSlider,'Value',NewVal)
   % Set the Gain parameter of the Kf Gain Block
   % to the new value.
   set_param('f14/Controller/Gain','Gain',NewStrVal)
end
```

The callback for the Ki **Current value** follows a similar approach.

Running the Simulation from the GUI

The GUI **Simulate and store results** button callback runs the model simulation and stores the results in the `handles` structure. Storing data in the `handles` structure simplifies the process of passing data to other subfunction since this structure can be passed as an argument.

When a user clicks on the **Simulate and store results** button, the callback executes the following steps:

- Calls `sim`, which runs the simulation and returns the data that is used for plotting.
- Creates a structure to save the results of the simulation, the current values of the simulation parameters set by the GUI, and the run name and number.
- Stores the structure in the `handles` structure.
- Updates the list box `String` to list the most recent run.

Here is the **Simulate and store results** button callback.

```
function SimulateButton_Callback(hObject, eventdata, handles)
[timeVector,stateVector,outputVector] = sim('f14');
% Retrieve old results data structure
if isfield(handles,'ResultsData') &
~isempty(handles.ResultsData)
 ResultsData = handles.ResultsData;
 % Determine the maximum run number currently used.
 maxNum = ResultsData(length(ResultsData)).RunNumber;
 ResultNum = maxNum+1;
else % Set up the results data structure
 ResultsData = struct('RunName',[],'RunNumber',[],...
               'KiValue',[],'KfValue',[],'timeVector',[],...
               'outputVector',[]);
 ResultNum = 1;
end
if isequal(ResultNum,1),
 % Enable the Plot and Remove buttons
 set([handles.RemoveButton,handles.PlotButton],'Enable','on')
end
```

```
% Get Ki and Kf values to store with the data and put in the
results list.
Ki = get(handles.KiValueSlider,'Value');
Kf = get(handles.KfValueSlider,'Value');
ResultsData(ResultNum).RunName = ['Run',num2str(ResultNum)];
ResultsData(ResultNum).RunNumber = ResultNum;
ResultsData(ResultNum).KiValue = Ki;
ResultsData(ResultNum).KfValue = Kf;
ResultsData(ResultNum).timeVector = timeVector;
ResultsData(ResultNum).outputVector = outputVector;
% Build the new results list string for the listbox
ResultsStr = get(handles.ResultsList,'String');
if isequal(ResultNum,1)
 ResultsStr = {['Run1',num2str(Kf),' ',num2str(Ki)]};
else
 ResultsStr = [ResultsStr;...
 {['Run',num2str(ResultNum),' ',num2str(Kf),' ', ...
 num2str(Ki)]}];
end
set(handles.ResultsList,'String',ResultsStr);
% Store the new ResultsData
handles.ResultsData = ResultsData;
guidata(hObject, handles)
```

Removing Results from the List Box

The GUI **Remove** button callback deletes any selected item from the **Results list** list box. It also deletes the corresponding run data from the `handles` structure. When a user clicks on the **Remove** button, the callback executes the following steps:

- Determines which list box items are selected when a user clicks on the **Remove** button and removes these items from the list box `String` property by setting each item to the empty matrix `[]`.
- Removes the deleted data from the `handles` structure.
- Displays the string `<empty>` and disables the **Remove** and **Plot** buttons (using the `Enable` property), if all the items in the list box are removed.
- Save the changes to the `handles` structure (`guidata`).

Here is the **Remove** button callback.

```
function RemoveButton_Callback(hObject, eventdata, handles)
currentVal = get(handles.ResultsList,'Value');
resultsStr = get(handles.ResultsList,'String');
numResults = size(resultsStr,1);
% Remove the data and list entry for the selected value
resultsStr(currentVal) =[];
handles.ResultsData(currentVal)=[];
% If there are no other entries, disable the Remove and Plot
button
% and change the list string to <empty>
if isequal(numResults,length(currentVal)),
 resultsStr = {'<empty>'};
 currentVal = 1;

set([handles.RemoveButton,handles.PlotButton],'Enable','off')
end
% Ensure that list box Value is valid, then reset Value and String
currentVal = min(currentVal,size(resultsStr,1));
set(handles.ResultsList,'Value',currentVal,'String',resultsStr)
% Store the new ResultsData
guidata(hObject, handles)
```

Plotting the Results Data

The GUI **Plot** button callback creates a plot of the run data and adds a legend. The data to plot is passed to the callback in the `handles` structure, which also contains the gain settings used when the simulation ran. When a user clicks on the **Plot** button, the callback executes the following steps:

- Collects the data for each run selected in the **Results list**, including two variables (time vector and output vector) and a color for each result run to plot.
- Generates a string for the legend from the stored data.
- Creates the figure and axes for plotting and saves the handles for use by the **Close** button callback.
- Plots the data, adds a legend, and makes the figure visible.

Plotting Into the Hidden Figure

The figure that contains the plot is created invisible and then made visible after adding the plot and legend. To prevent this figure from becoming the target for plotting commands issued at the command line or by other GUIs, its `HandleVisibility` and `IntegerHandle` properties are set to `off`. However, this means the figure is also hidden from the `plot` and `legend` commands.

Use the following steps to plot into a hidden figure:

- Save the handle of the figure when you create it.
- Create an axes, set its `Parent` property to the figure handle, and save the axes handle.
- Create the plot (which is one or more line objects), save these line handles, and set their `Parent` properties to the handle of the axes.
- Make the figure visible.

Plot Button Callback Listing

Here is the **Plot** button callback.

```
function PlotButton_Callback(hObject, eventdata, handles)
currentVal = get(handles.ResultsList,'Value');
% Get data to plot and generate command string with color
% specified
legendStr = cell(length(currentVal),1);
plotColor = {'b','g','r','c','m','y','k'};
for ctVal = 1:length(currentVal);
 PlotData{(ctVal*3)-2} =
handles.ResultsData(currentVal(ctVal)).timeVector;
 PlotData{(ctVal*3)-1} =
handles.ResultsData(currentVal(ctVal)).outputVector;
 numColor = ctVal - 7*( floor((ctVal-1)/7) );
 PlotData{ctVal*3} = plotColor{numColor};
 legendStr{ctVal} = ...
      [handles.ResultsData(currentVal(ctVal)).RunName,'; Kf=',...
      num2str(handles.ResultsData(currentVal(ctVal)).KfValue),...
      ';  Ki=', ...
      num2str(handles.ResultsData(currentVal(ctVal)).KiValue)];
end
```

```
% If necessary, create the plot figure and store in handles
% structure
if ~isfield(handles,'PlotFigure') ||...
   ~ishandle(handles.PlotFigure),
 handles.PlotFigure = ...
        figure('Name','F14 Simulation Output',...
        'Visible','off','NumberTitle','off',...
        'HandleVisibility','off','IntegerHandle','off');
 handles.PlotAxes = axes('Parent',handles.PlotFigure);
 guidata(hObject, handles)
end
% Plot data
pHandles = plot(PlotData{:},'Parent',handles.PlotAxes);
% Add a legend, and bring figure to the front
legend(pHandles(1:2:end),legendStr{:})
% Make the figure visible and bring it forward
figure(handles.PlotFigure)
```

The GUI Help Button

The GUI **Help** button callback displays an HTML file in the MATLAB Help browser. It uses two commands:

- The `which` command returns the full path to the file when it is on the MATLAB path
- The `web` command displays the file in the Help browser.

This is the **Help** button callback.

```
function HelpButton_Callback(hObject, eventdata, handles)
HelpPath = which('f14ex_help.html');
web(HelpPath);
```

You can also display the help document in a Web browser or load an external URL. See the Web documentation for a description of these options.

Closing the GUI

The GUI **Close** button callback closes the plot figure, if one exists and then closes the GUI. The handle of the plot figure and the GUI figure are available from the `handles` structure. The callback executes two steps:

- Checks to see if there is a `PlotFigure` field in the `handles` structure and if it contains a valid figure handle (the user could have closed the figure manually).
- Closes the GUI figure

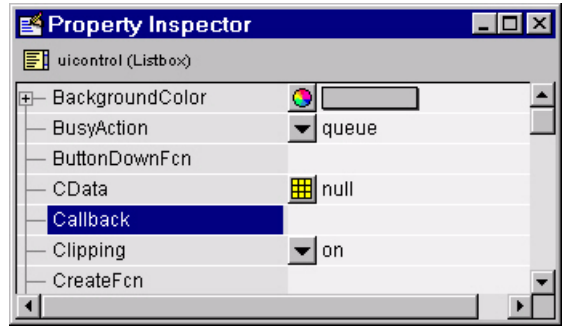
This is the **Close** button callback.

```
function CloseButton_Callback(hObject, eventdata, handles)
% Close the GUI and any plot window that is open
if isfield(handles,'PlotFigure') && ...
      ishandle(handles.PlotFigure),
   close(handles.PlotFigure);
end
close(handles.F14ControllerEditor);
```

The List Box Callback and Create Function

This GUI does not use the list box callback since the actions performed on list box items are carried out by push buttons (**Simulate and store results**, **Remove**, and **Plot**). However, GUIDE automatically inserts a callback stub when you add the list box and automatically sets the `Callback` property to execute this subfunction whenever the callback is triggered (which happens when users select an item in the list box).

In this case, there is no need for the list box callback to execute, so you should delete it from the GUI M-file. It is important to remember to also delete the `Callback` property string so MATLAB does not attempt to execute the callback. You can do this using the property inspector:



See the description of list box for more information on how to trigger the list box callback.

Setting the Background to White

The list box create function enables you to determine the background color of the list box. The following code shows the create function for the list box that is tagged `ResultsList`.

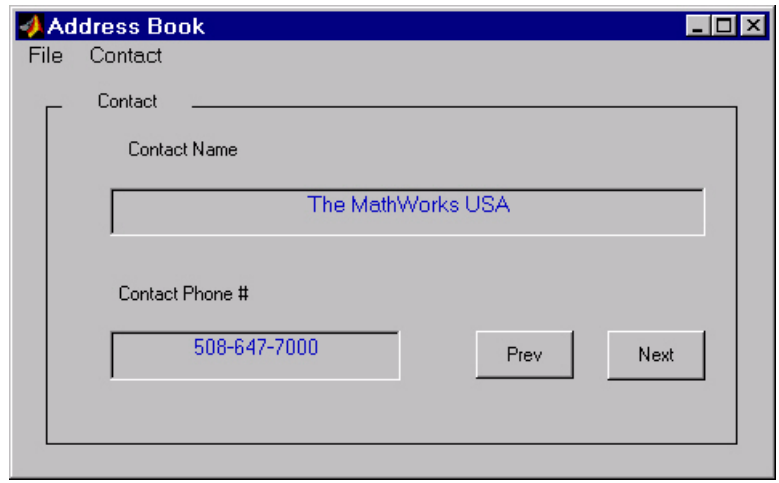
```
function ResultsList_CreateFcn(hObject, eventdata, handles)
% Hint: listbox controls usually have a white background, change
%       'usewhitebg' to 0 to use default. See ISPC and COMPUTER.
usewhitebg = 1;
if usewhitebg
    set(hObject,'BackgroundColor','white');
else
set(hObject,'BackgroundColor',...
    get(0,'defaultUicontrolBackgroundColor'));
end
```

An Address Book Reader

In this section...

Address Book Reader Example Outcome

This example shows how to implement a GUI that displays names and phone numbers, which it reads from a MAT-file.



Techniques Used in This Example

This example demonstrates the following GUI programming techniques:

- Uses open and save dialogs to provide a means for users to locate and open the address book MAT-files and to save revised or new address book MAT-files.
- Defines callbacks written for GUI menus.
- Uses the GUI's `handles` structure to save and recall shared data.
- Uses a GUI figure resize function.

Managing Shared Data

One of the key techniques illustrated in this example is how to keep track of information and make it available to the various subfunctions. This information includes

- The name of the current MAT-file
- The names and phone numbers stored in the MAT-file
- An index pointer that indicates the current name and phone number, which must be updated as the user pages through the address book

- The figure position and size
- The handles of all GUI components

The descriptions of the subfunctions that follow illustrate how to save and retrieve information from the `handles` structure. See “handles Structure” on page 8-15 for background information on this structure.

View Completed Layout and Its GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display this GUI in the Layout Editor.
- Click here to display the GUI M-file in the MATLAB Editor.

Running the GUI

The GUI is nonblocking and nonmodal since it is designed to be displayed while you perform other MATLAB tasks.

GUI Option Settings

This GUI uses the following GUI option settings:

- Resize behavior: **User-specified**
- Command-line accessibility: **Off**
- GUI M-file options selected:

- Generate callback function prototypes
- Application allows only one instance to run

Calling the GUI

You can call the GUI M-file with no arguments, in which case the GUI uses the default address book MAT-file, or you can specify an alternate MAT-file from which the GUI reads information. In this example, the user calls the GUI with a pair of arguments, `address_book('book', 'my_list.mat')`. The first argument, `'book'`, is a key word that the M-file looks for in the opening function. If the M-file finds the key word, it knows to use the second argument as the MAT-file for the address book. Calling the GUI with this syntax is analogous to calling it with a valid property-value pair, such as `('color', 'red')`. However, since `'book'` is not a valid figure property, in this example the opening function in the M-file includes code to recognize the pair `('book', 'my_list.mat')`.

Note that it is not necessary to use the key word `'book'`. You could program the M-file to accept just the MAT-file as an argument, using the syntax `address_book('my_list.mat')`. The advantage of calling the GUI with the pair `('book', 'my_list.mat')` is that you can program the GUI to accept other user arguments, as well as valid figure properties, using the property-value pair syntax. The GUI can then identify which property the user wants to specify from the property name.

The following code shows how to program the opening function to look for the key word `'book'`, and if it finds the key word, to use the MAT-file specified by the second argument as the list of contacts.

```
function address_book_OpeningFcn(hObject, eventdata,...
    handles, varargin)
% Choose default command line output for address_book
handles.output = hObject;
% Update handles structure
guidata(hObject, handles);
% User added code follows
if nargin < 4
    % Load the default address book
    Check_And_Load([],handles);
    % If the first element in varargin is 'book' and
```

```
    & the second element is a MATLAB file, then load that file
elseif (length(varargin) == 2 && ...
       strcmpi(varargin{1},'book') && ...
       (2 == exist(varargin{2},'file')))
    Check_And_Load(varargin{2},handles);
else
    errordlg('File Not Found','File Load Error')
    set(handles.Contact_Name,'String','')
    set(handles.Contact_Phone,'String','')
end
```

Loading an Address Book Into the Reader

There are two ways in which an address book (i.e., a MAT-file) is loaded into the GUI:

- When running the GUI, you can specify a MAT-file as an argument. If you do not specify an argument, the GUI loads the default address book (`addrbook.mat`).
- The user can select **Open** under the **File** menu to browse for other MAT-files.

Validating the MAT-file

To be a valid address book, the MAT-file must contain a structure called `Addresses` that has two fields called `Name` and `Phone`. The `Check_And_Load` subfunction validates and loads the data with the following steps:

- Loads (`load`) the specified file or the default if none is specified.
- Determines if the MAT-file is a valid address book.
- Displays the data if it is valid. If it is not valid, displays an error dialog (`errordlg`).
- Returns `1` for valid MAT-files and `0` if invalid (used by the **Open** menu callback)
- Saves the following items in the `handles` structure:
  - The name of the MAT-file
  - The `Addresses` structure

- An index pointer indicating which name and phone number are currently displayed

Check_And_Load Code Listing

This is the `Check_And_Load` function.

```
function pass = Check_And_Load(file,handles)
% Initialize the variable "pass" to determine if this is
% a valid file.
pass = 0;
% If called without any file then set file to the default
% filename.
% Otherwise, if the file exists then load it.
if isempty(file)
 file = 'addrbook.mat';
 handles.LastFile = file;
 guidata(handles.Address_Book,handles)
end
if exist(file) == 2
 data = load(file);
end
% Validate the MAT-file
% The file is valid if the variable is called "Addresses"
% and it has fields called "Name" and "Phone"
flds = fieldnames(data);
if (length(flds) == 1) && (strcmp(flds{1},'Addresses'))
 fields = fieldnames(data.Addresses);
 if (length(fields) == 2) && ...
         (strcmp(fields{1},'Name')) && ...
         (strcmp(fields{2},'Phone'))
       pass = 1;
 end
end
% If the file is valid, display it
if pass
 % Add Addresses to the handles structure
 handles.Addresses = data.Addresses;
 guidata(handles.Address_Book,handles)
 % Display the first entry
```

```
 set(handles.Contact_Name,'String',data.Addresses(1).Name)
 set(handles.Contact_Phone,'String',data.Addresses(1).Phone)
 % Set the index pointer to 1 and save handles
 handles.Index = 1;
 guidata(handles.Address_Book,handles)
else
 errordlg('Not a valid Address Book','Address Book Error')
end
```

The Open Menu Callback

The address book GUI contains a **File** menu that has an **Open** submenu for loading address book MAT-files. When selected, **Open** displays a dialog (`uigetfile`) that enables the user to browse for files. The dialog displays only MAT-files, but users can change the filter to display all files.

The dialog returns both the filename and the path to the file, which is then passed to `fullfile` to ensure the path is properly constructed for any platform. `Check_And_Load` validates and load the new address book.

Open_Callback Code Listing

```
function Open_Callback(hObject, eventdata, handles)
[filename, pathname] = uigetfile( ...
 {'*.mat', 'All MAT-Files (*.mat)'; ...
  '*.*','All Files (*.*)'}, ...
 'Select Address Book');
% If "Cancel" is selected then return
if isequal([filename,pathname],[0,0])
 return
% Otherwise construct the fullfilename and Check and load
% the file
else
 File = fullfile(pathname,filename);
 % if the MAT-file is not valid, do not save the name
 if Check_And_Load(File,handles)
  handles.LastFIle = File;
  guidata(hObject, handles)
 end
end
```

See the “Creating Menus” on page 6-70 section for information on creating the menu.

The Contact Name Callback

The **Contact Name** text box displays the name of the address book entry. If you type in a new name and press enter, the callback performs these steps:

- If the name exists in the current address book, the corresponding phone number is displayed.
- If the name does not exist, a question dialog (`questdlg`) asks you if you want to create a new entry or cancel and return to the name previously displayed.
- If you create a new entry, you must save the MAT-file with the **File > Save** menu.

Storing and Retrieving Data

This callback makes use of the `handles` structure to access the contents of the address book and to maintain an index pointer (`handles.Index`) that enables the callback to determine what name was displayed before it was changed by the user. The index pointer indicates what name is currently displayed. The address book and index pointer fields are added by the `Check_And_Load` function when the GUI is run.

If the user adds a new entry, the callback adds the new name to the address book and updates the index pointer to reflect the new value displayed. The updated address book and index pointer are again saved (`guidata`) in the `handles` structure.

Contact Name Callback

```
function Contact_Name_Callback(hObject, eventdata, handles)
% Get the strings in the Contact Name and Phone text box
Current_Name = get(handles.Contact_Name,'string');
Current_Phone = get(handles.Contact_Phone,'string');
% If empty then return
if isempty(Current_Name)
 return
```

```
end
% Get the current list of addresses from the handles structure
Addresses = handles.Addresses;
% Go through the list of contacts
% Determine if the current name matches an existing name
for i = 1:length(Addresses)
 if strcmp(Addresses(i).Name,Current_Name)
  set(handles.Contact_Name,'string',Addresses(i).Name)
  set(handles.Contact_Phone,'string',Addresses(i).Phone)
  handles.Index = i;
  guidata(hObject, handles)
  return
 end
end
% If it's a new name, ask to create a new entry
Answer=questdlg('Do you want to create a new entry?', ...
 'Create New Entry', ...
 'Yes','Cancel','Yes');
switch Answer
case 'Yes'
 Addresses(end+1).Name = Current_Name; % Grow array by 1
 Addresses(end).Phone = Current_Phone;
 index = length(Addresses);
 handles.Addresses = Addresses;
 handles.Index = index;
 guidata(hObject, handles)
 return
case 'Cancel'
 % Revert back to the original number

set(handles.Contact_Name,'String',Addresses(handles.Index).Name
)

set(handles.Contact_Phone,'String',Addresses(handles.Index).Pho
ne)
 return
end
```

The Contact Phone Number Callback

The **Contact Phone #** text box displays the phone number of the entry listed in the **Contact Name** text box. If you type in a new number click one of the push buttons, the callback opens a question dialog that asks you if you want to change the existing number or cancel your change.

Like the **Contact Name** text box, this callback uses the index pointer (`handles.Index`) to update the new number in the address book and to revert to the previously displayed number if the user selects **Cancel** from the question dialog. Both the current address book and the index pointer are saved in the `handles` structure so that this data is available to other callbacks.

If you create a new entry, you must save the MAT-file with the **File > Save** menu.

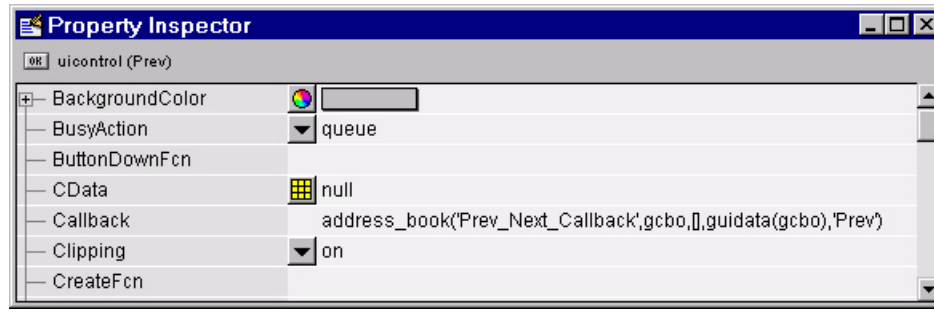
Code Listing

```
function Contact_Phone_Callback(hObject, eventdata, handles)
Current_Phone = get(handles.Contact_Phone,'string');
% If either one is empty then return
if isempty(Current_Phone)
 return
end
% Get the current list of addresses from the handles structure
Addresses = handles.Addresses;
Answer=questdlg('Do you want to change the phone number?', ...
 'Change Phone Number', ...
 'Yes','Cancel','Yes');
switch Answer
case 'Yes'
 % If no name match was found create a new contact
 Addresses(handles.Index).Phone = Current_Phone;
 handles.Addresses = Addresses;
 guidata(hObject, handles)
 return
case 'Cancel'
 % Revert back to the original number
 set(handles.Contact_Phone,...
       'String',Addresses(handles.Index).Phone)
```

```
    return
end
```

Paging Through the Address Book — Prev/Next

The **Prev** and **Next** buttons page back and forth through the entries in the address book. Both push buttons use the same callback, `Prev_Next_Callback`. You must set the `Callback` property of both push buttons to call this subfunction, as the following illustration of the **Prev** push button `Callback` property setting shows.



Determining Which Button Is Clicked

The callback defines an additional argument, `str`, that indicates which button, **Prev** or **Next**, was clicked. For the **Prev** button `Callback` property (illustrated above), the `Callback` string includes `'Prev'` as the last argument. The **Next** button `Callback` string includes `'Next'` as the last argument. The value of `str` is used in `case` statements to implement each button's functionality (see the code listing below).

Paging Forward or Backward

`Prev_Next_Callback` gets the current index pointer and the addresses from the `handles` structure and, depending on which button the user presses, the index pointer is decremented or incremented and the corresponding address and phone number are displayed. The final step stores the new value for the index pointer in the `handles` structure and saves the updated structure using `guidata`.

Code Listing

```
function Prev_Next_Callback(hObject, eventdata,handles,str)
% Get the index pointer and the addresses
index = handles.Index;
Addresses = handles.Addresses;
% Depending on whether Prev or Next was clicked,
% change the display
switch str
case 'Prev'
 % Decrease the index by one
 i = index - 1;
 % If the index is less than one then set it equal to the index
   %  of the last element in the Addresses array
 if i < 1
  i = length(Addresses);
 end
case 'Next'
 % Increase the index by one
 i = index + 1;
 % If the index is greater than the size of the array then
 % point to the first item in the Addresses array
 if i > length(Addresses)
  i = 1;
 end
end
% Get the appropriate data for the index in selected
Current_Name = Addresses(i).Name;
Current_Phone = Addresses(i).Phone;
set(handles.Contact_Name,'string',Current_Name)
set(handles.Contact_Phone,'string',Current_Phone)
% Update the index pointer to reflect the new index
handles.Index = i;
guidata(hObject, handles)
```

Saving Changes to the Address Book from the Menu

When you make changes to an address book, you need to save the current MAT-file, or save it as a new MAT-file. The **File** submenus **Save** and **Save As** enable you to do this. These menus, created with the Menu Editor, use the same callback, `Save_Callback`.

The callback uses the menu `Tag` property to identify whether **Save** or **Save As** is the callback object (i.e., the object whose handle is passed in as the first argument to the callback function). You specify the menu's `Tag` property with the Menu Editor.

Saving the Addresses Structure

The `handles` structure contains the `Addresses` structure, which you must save (`handles.Addresses`) as well as the name of the currently loaded MAT-file (`handles.LastFile`). When the user makes changes to the name or number, the `Contact_Name_Callback` or the `Contact_Phone_Callback` updates `handles.Addresses`.

Saving the MAT-File

If the user selects **Save**, the `save` command is called to save the current MAT-file with the new names and phone numbers.

If the user selects **Save As**, a dialog is displayed (`uiputfile`) that enables the user to select the name of an existing MAT-file or specify a new file. The dialog returns the selected filename and path. The final steps include

- Using `fullfile` to create a platform-independent pathname.
- Calling `save` to save the new data in the MAT-file.
- Updating the `handles` structure to contain the new MAT-file name.
- Calling `guidata` to save the `handles` structure.

Save_Callback Code Listing

```
function Save_Callback(hObject, eventdata, handles)
% Get the Tag of the menu selected
Tag = get(hObject, 'Tag');
% Get the address array
Addresses = handles.Addresses;
% Based on the item selected, take the appropriate action
switch Tag
case 'Save'
 % Save to the default addrbook file
 File = handles.LastFile;
```

```
 save(File,'Addresses')
case 'Save_As'
 % Allow the user to select the file name to save to
 [filename, pathname] = uiputfile( ...
  {'*.mat';'*.*'}, ...
  'Save as');
 % If 'Cancel' was selected then return
 if isequal([filename,pathname],[0,0])
  return
 else
  % Construct the full path and save
  File = fullfile(pathname,filename);
  save(File,'Addresses')
  handles.LastFile = File;
  guidata(hObject, handles)
 end
end
```

The Create New Menu

The **Create New** menu simply clears the **Contact Name** and **Contact Phone #** text fields to facilitate adding a new name and number. After making the new entries, the user must then save the address book with the **Save** or **Save As** menus. This callback sets the text `String` properties to empty strings:

```
function New_Callback(hObject, eventdata, handles)
set(handles.Contact_Name,'String','')
set(handles.Contact_Phone,'String','')
```

The Address Book Resize Function

The address book defines its own resize function. To use this resize function, you must set the Application Options dialog **Resize behavior** to `User-specified`, which in turn sets the figure's `ResizeFcn` property to:

```
address_book('ResizeFcn',gcbo,[],guidata(gcbo))
```

Whenever the user resizes the figure, MATLAB calls the `ResizeFcn` subfunction in the address book M-file (`address_book.m`)

Behavior of the Resize Function

The `resize` function allows users to make the figure wider, to accommodate long names and numbers, but does not allow the figure to be made narrower than its original width. Also, users cannot change the height. These restrictions do not limit the usefulness of the GUI and simplify the `resize` function, which must maintain the proper proportions between the figure size and the components in the GUI.

When the user resizes the figure and releases the mouse, the resize function executes. At that point, the resized figure's dimensions are saved. The following sections describe how the `resize` function handles the various possibilities.

Changing the Width

If the new width is greater than the original width, set the figure to the new width.

The size of the **Contact Name** text box changes in proportion to the new figure width. This is accomplished by:

- Changing the `Units` of the text box to `normalized`.
- Resetting the width of the text box to be 78.9% of the figure's width.
- Returning the `Units` to `characters`.

If the new width is less than the original width, use the original width.

Changing the Height

If the user attempts to change the height, use the original height. However, because the resize function is triggered when the user releases the mouse button after changing the size, the resize function cannot always determine the original position of the GUI on screen. Therefore, the resize function applies a compensation to the vertical position (second element in the figure `Position` vector) by adding the vertical position when the mouse is released to the height when mouse is released and subtracting the original height.

When the figure is resized from the bottom, it stays in the same position. When resized from the top, the figure moves to the location where the mouse button is released.

Ensuring the Resized Figure Is On Screen

The `resize` function calls `movegui` to ensure that the resized figure is on screen regardless of where the user releases the mouse.

When the GUI is first run, it is displayed at the size and location specified by the figure `Position` property. You can set this property with the Property Inspector when you create the GUI.

Code Listing

```
function ResizeFcn(hObject, eventdata, handles)
% Get the figure size and position
Figure_Size = get(hObject, 'Position');
% Set the figure's original size in character units
Original_Size = [ 0 0 94 19.230769230769234];
% If the resized figure is smaller than the
% original figure size then compensate.
if (Figure_Size(3)<Original_Size(3)) | ...
   (Figure_Size(4) ~= Original_Size(4))
   if Figure_Size(3) < Original_Size(3)
       % If the width is too small then reset to origianl width.
       set(hObject, 'Position',...
           [Figure_Size(1), Figure_Size(2), ...
            Original_Size(3), Original_Size(4)])
       Figure_Size = get(hObject, 'Position');
   end
   if Figure_Size(4) ~= Original_Size(4)
       % Do not allow the height to change.
       set(hObject, 'Position',...
           [Figure_Size(1),...
            Figure_Size(2)+Figure_Size(4)-Original_Size(4),...
            Figure_Size(3), Original_Size(4)])
   end
end
% Adjust the size of the Contact Name text box.
```

```
% Set the units of the Contact Name field to 'Normalized'.
set(handles.Contact_Name,'units','normalized')
% Get its Position.
C_N_pos = get(handles.Contact_Name,'Position');
% Reset it so that it's width remains normalized.
% relative to figure.
set(handles.Contact_Name,'Position',...
 [C_N_pos(1) C_N_pos(2)  0.789 C_N_pos(4)])
% Return the units to 'Characters'.
set(handles.Contact_Name,'units','characters')
% Reposition GUI on screen.
movegui(hObject, 'onscreen')
```

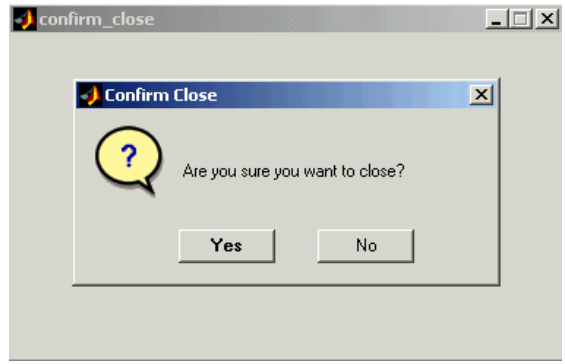
Using a Modal Dialog to Confirm an Operation

In this section...

Modal Dialog Example Outcome

This example illustrates how to use the modal dialog GUI together with another GUI that has a **Close** button. Clicking the **Close** button displays the modal dialog, which asks users to confirm that they really want to proceed with the close operation.

The following figure illustrates the dialog positioned over the GUI application, awaiting the user's response.



View Completed Layouts and Their GUI M-Files

If you are reading this in the MATLAB Help Browser, you can click the following links to display the GUIDE Layout Editor and the MATLAB Editor

with a completed version of this example. This enables you to see the values of all component properties and to understand how the components are assembled to create the GUI. You can also see a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

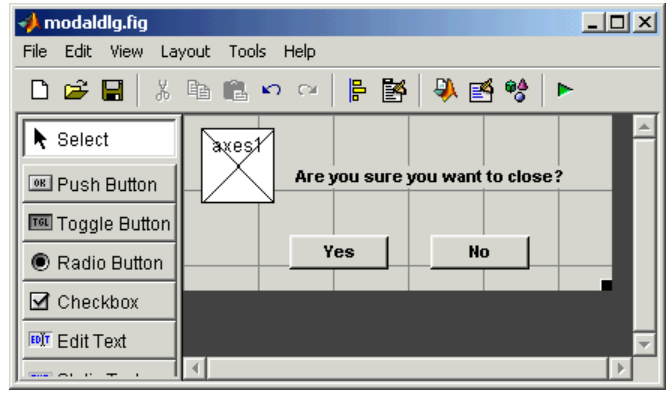
- Click here to display the GUIs in the Layout Editor.
- Click here to display the GUI M-files in the editor.

Setting Up the Close Confirmation Dialog

To set up the dialog, do the following:

1. Select **New** from the **File** menu in the GUIDE Layout Editor.
2. In the **GUIDE Quick Start** dialog, select the **Modal Question Dialog** template and click **OK**.
3. Right-click the static text, `Do you want to create a question dialog?`, in the Layout Editor and select **Property Inspector** from the pop-up menu.
4. Scroll down to `String` in the Property Inspector and change the `String` property to `Are you sure you want to close?`
5. Select **Save** from the **File** menu and type `modaldlg.fig` in the **File name** field.

The GUI should now appear as in the following figure.



Setting Up the GUI with the Close Button

To set up the second GUI with a **Close** button, do the following:

1. Select **New** from the **File** menu in the GUIDE Layout Editor.
2. In the **GUIDE Quick Start** dialog, select **Blank GUI (Default)** and click **OK**. This opens the blank GUI in a new Layout Editor window.
3. Drag a push button from the Component palette of the Layout Editor into the layout area.
4. Right-click the push button and select **Property Inspector** from the pop-up menu.
5. Change the `String` property to `Close`.
6. Change the `Tag` property to `close_pushbutton`.
7. Click the M-file Editor icon on the toolbar of the Layout Editor.
8. Click the Show functions icon on the toolbar of the M-file editor and select `close_pushbutton_Callback` from the menu.

   The following generated code for the **Close** button callback should appear in the M-file editor:

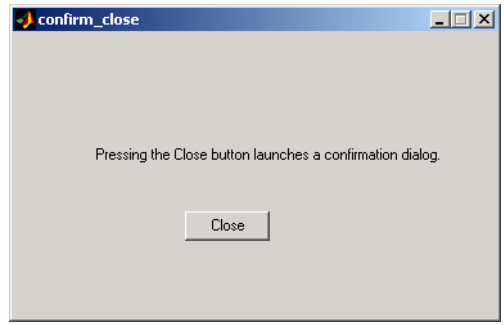
```
% --- Executes on button press in close_pushbutton.
function close_pushbutton_Callback(hObject, eventdata, handles)
% hObject    handle to close_pushbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

9 After these comments, add the following code:

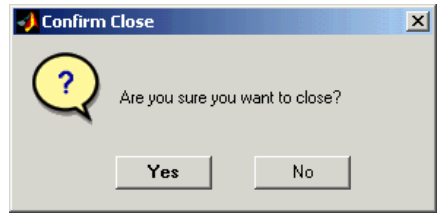
```
% Get the current position of the GUI from the handles structure
% to pass to the modal dialog.
pos_size = get(handles.figure1,'Position');
% Call modaldlg with the argument 'Position'.
user_response = modaldlg('Title','Confirm Close');
switch user_response
case {'No'}
 % take no action
case 'Yes'
 % Prepare to close GUI application window
 %                  .
 %                  .
 %                  .
 delete(handles.figure1)
end
```

Running the GUI with the Close Button

Run the GUI with the **Close** button by clicking the **Run** button on the Layout Editor toolbar. The GUI appears as in the following figure:



When you click the **Close** button on the GUI, the modal dialog appears as shown in the following figure:



Clicking the **Yes** button closes both the close dialog and the GUI that calls it. Clicking the **No** button closes just the dialog.

How the GUI and Dialog Work

This section describes what occurs when you click the **Close** button on the GUI:

1. User clicks the **Close** button. Its callback then
   - Gets the current position of the GUI from the `handles` structure with the command

     ```
     pos_size = get(handles.figure1,'Position')
     ```

   - Calls the modal dialog with the command

     ```
     user_response = modaldlg('Title','Confirm Close');
     ```

     This is an example of calling a GUI with a property value pair. In this case, the figure property is `'Title'`, and its value is the string `'Confirm Close'`. Opening `modaldlg` with this syntax displays the text "Confirm Close" at the top of the dialog.
2. The modal dialog opens with the `'Position'` obtained from the GUI that calls it.
3. The opening function in the modal dialog M-file:
   - Makes the dialog modal.

- Executes the `uiwait` command, which causes the dialog to wait for the user to click the **Yes** button or the **No** button, or click the close box (`X`) on the window border.

4 When a user clicks one of the two push buttons, the callback for the push button

- Updates the output field in the `handles` structure
- Executes `uiresume` to return control to the opening function where `uiwait` is called.

5 The output function is called, which returns the string `Yes` or `No` as an output argument, and deletes the dialog with the command

```
delete(handles.figure1)
```

6 When the GUI with the **Close** button regains control, it receives the string `Yes` or `No`. If the answer is `'No'`, it does nothing. If the answer is `'Yes'`, the **Close** button callback closes the GUI with the command

```
delete(handles.figure1)
```

Creating GUIs Programmatically

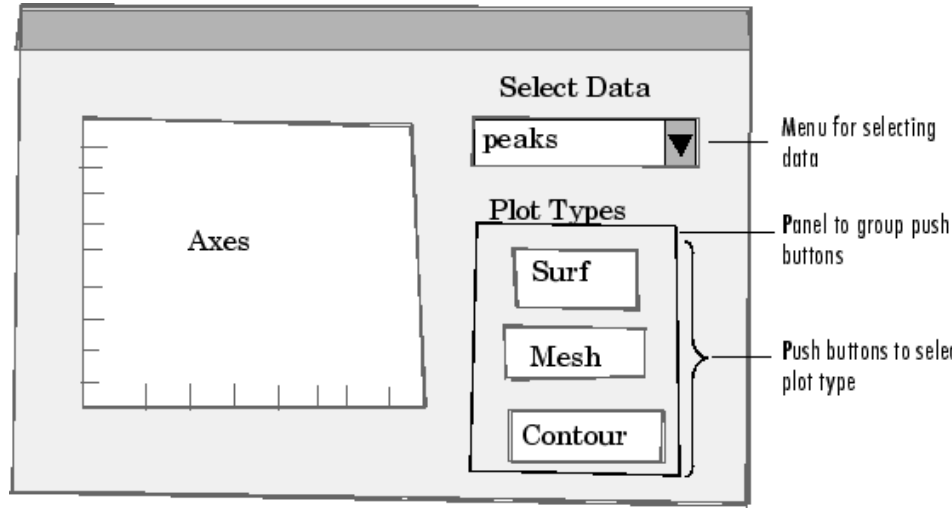
Laying Out a GUI

Designing a GUI

Before creating the actual GUI, it is important to decide what it is you want your GUI to do and how you want it to work. It is helpful to draw your GUI on paper and envision what the user sees and what actions the user takes.

Note MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see "Predefined Dialog Boxes" in the MATLAB Function Reference documentation.

The GUI used in this example contains an axes component that displays either a surface, mesh, or contour plot of data selected from the pop-up menu. The following picture shows a sketch that you might use as a starting point for the design.



A panel contains three push buttons that enable you to choose the type of plot you want. The pop-up menu contains three strings—`peaks`, `membrane`, and `sinc`, which correspond to MATLAB functions and generate data to plot. You can select the data to plot from this menu.

Many Web sites and commercial publications such as the following provide guidelines for designing GUIs:

- AskTog — Essays on good design and a list of First Principles for good user interface design. The author, Tognazzini, is a well-respected user interface designer. `http://www.asktog.com/basics/firstPrinciples.html`.
- Galitz, Wilbert, O., *Essential Guide to User Interface Design*. Wiley, New York, NY, 2002.
- GUI Design Handbook — A detailed guide to the use of GUI controls. `http://www.fast-consulting.com/GUI%20Design%20Handbook/GDH_FRNTMTR.htm`.
- Johnson, J., *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann, San Francisco, CA, 2000.
- Usability Glossary — An extensive glossary of terms related to GUI design, usability, and related topics. `http://www.usabilityfirst.com/glossary/main.cgi`.
- UsabilityNet — Covers design principles, user-centered design, and other usability and design-related topics. `http://www.usabilitynet.org/management/b_design.htm`.

Creating and Running the GUI M-File

In this section...

Note For an example of creating an M-file, see Chapter 3, "Creating a Simple GUI Programmatically" in the "Getting Started" part of this document.

File Organization

Typically, a GUI M-file has the following ordered sections. You can help to maintain the organization by adding comments that name the sections when you first create them.

1. Comments displayed in response to the MATLAB `help` command.
2. Initialization tasks such as data creation and any processing that is needed to construct the components. See "Initializing the GUI" on page 12-4 for more information.
3. Construction of figure and components. For more information, see "Creating the GUI Figure" on page 11-7 and "Adding Components to the GUI" on page 11-10.
4. Initialization tasks that require the components to exist, and output return. See "Initializing the GUI" on page 12-4 for more information.
5. Callbacks for the components. Callbacks are the routines that execute in response to user-generated events such as mouse clicks and key strokes. See Chapter 12, "Programming the GUI" for more information.
6. Utility functions.

File Template

This is a template for a GUI M-file:

```
function varargout = mygui(varargin)
% MYGUI Brief description of GUI.
%       Comments displayed at the command line in response
%       to the help command.

% (Leave a blank line following the help.)

%  Initialization tasks

%  Construct the components

%  Initialization tasks

%  Callbacks for MYGUI

%  Utility functions for MYGUI

end
```

The `end` statement that matches the `function` statement is necessary because this document treats GUI creation using nested functions. Chapter 12, “Programming the GUI” addresses this topic.

Save the file in your current directory or at a location that is on your MATLAB path.

Running the GUI

You can display your GUI at any time by executing its M-file. For example, if your GUI M-file is `mygui.m`, type

```
mygui
```

at the command line. Provide run-time arguments as appropriate. The files must reside on your path or in your current directory.

When you execute the GUI M-file, a fully functional copy of the GUI displays on the screen. You can manipulate components that it contains, but nothing happens unless the M-file includes code to initialize the GUI and callbacks

to service the components. Chapter 12, “Programming the GUI” tells you how to do this.

Creating the GUI Figure

In MATLAB, a GUI is a figure. Before you add components to it, create the figure explicitly and obtain a handle for it. In the initialization section of your file, use a statement such as the following to create the figure:

```
fh = figure;
```

where `fh` is the figure handle.

Note If you create a component when there is no figure, MATLAB creates a figure automatically but you do not know the figure handle.

When you create the figure, you can also specify properties for the figure. The most commonly used figure properties are shown in the following table:

Property	Values	Description
`MenuBar`	`figure`, `none`. Default is `figure`.	Display or hide the MATLAB standard menu bar menus. If `none` and there are no user-created menus, the menu bar itself is removed.
`Name`	String	Title displayed in the figure window. If `NumberTitle` is `on`, this string is appended to the figure number.
`NumberTitle`	`on`, `off`. Default is `on`.	Determines whether the string 'Figure `n`' (where `n` is the figure number) is prefixed to the figure window title specified by `Name`.
`Position`	4-element vector: [distance from left, distance from bottom, width, height].	Size of the GUI figure and its location relative to the lower-left corner of the screen.

Property	Values	Description
`Resize`	`on`, `off`. Default is `on`.	Determines if the user can resize the figure window with the mouse.
`Toolbar`	`auto`, `none`, `figure`. Default is `auto`.	Display or hide the default figure toolbar. • `none` — do not display the figure toolbar. • `auto` — display the figure toolbar, but remove it if a user interface control (`uicontrol`) is added to the figure. • `figure` — display the figure toolbar.
`Units`	`pixels`, `centimeters`, `characters`, `inches`, `normalized`, `points`, Default is `pixels`.	Units of measurement used to interpret position vector
`Visible`	`on`, `off`. Default is `on`.	Determines whether a figure is displayed on the screen.

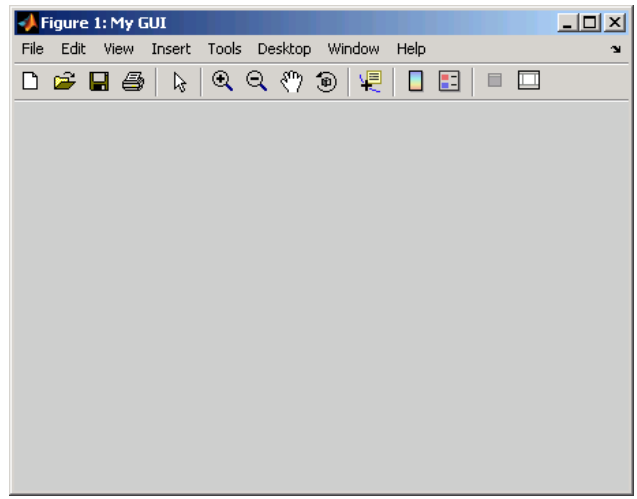
For a complete list of properties and for more information about the properties listed in the table, see the Figure Properties reference page in the MATLAB reference documentation.

The following statement names the figure My GUI, positions the figure on the screen, and makes the GUI invisible so that the user cannot see the components as they are added or initialized. All other properties assume their defaults.

```
f = figure('Visible','off','Name','My GUI',...
           'Position',[360,500,450,285]);
```

The `Position` property is a four-element vector that specifies the location of the GUI on the screen and its size: [distance from left, distance from bottom, width, height]. Default units are pixels.

If the figure were visible, it would look like this:



The next topic, “Adding Components to the GUI” on page 11-10, shows you how to add push buttons, axes, and other components to the GUI. “Creating Menus” on page 11-45 shows you how to create toolbar and context menus. “Creating Toolbars” on page 11-56 shows you how to add your own toolbar to a GUI.

Adding Components to the GUI

In this section...

Available Components

Components include user interface controls such as push buttons and sliders, containers such as panels and button groups, axes, and ActiveX controls. This topic tells you how to populate your GUI with these components.

Note MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see "Predefined Dialog Boxes" in the MATLAB Function Reference documentation.

The following table describes the available components and the function used to create each. Subsequent topics provide specific information about adding the components.

Component	Function	Description
ActiveX	`actxcontrol`	ActiveX components enable you to display ActiveX controls in your GUI. They are available only on the Microsoft Windows platform.
"Axes" on page 11-35	`axes`	Axes enable your GUI to display graphics such as graphs and images.

Component	Function	Description
"Button Group" on page 11-32	`uibuttongroup`	Button groups are like panels, but are used to manage exclusive selection behavior for radio buttons and toggle buttons.
"Check Box" on page 11-16	`uicontrol`	Check boxes can generate an action when checked and indicate their state as checked or not checked. Check boxes are useful when providing the user with a number of independent choices, for example, displaying a toolbar.
"Edit Text" on page 11-17	`uicontrol`	Edit text components are fields that enable users to enter or modify text strings. Use an edit text when you want text as input. Users can enter numbers, but you must convert them to their numeric equivalents.
"List Box" on page 11-18	`uicontrol`	List boxes display a list of items and enable users to select one or more items.
"Panel" on page 11-30	`uipanel`	Panels arrange GUI components into groups. By visually grouping related controls, panels can make the user interface easier to understand. A panel can have a title and various borders. Panel children can be user interface controls and axes, as well as button groups and other panels. The position of each component within a panel is interpreted relative to the panel. If you move the panel, its children move with it and maintain their positions on the panel.

Component	Function	Description
"Pop-Up Menu" on page 11-20	`uicontrol`	Pop-up menus open to display a list of choices when users click the arrow.
"Push Button" on page 11-21	`uicontrol`	Push buttons generate an action when clicked. For example, an *OK* button might apply settings and close a dialog box. When you click a push button, it appears depressed; when you release the mouse button, the push button appears raised.
"Radio Button" on page 11-23	`uicontrol`	Radio buttons are similar to check boxes, but radio buttons are typically mutually exclusive within a group of related radio buttons. That is, when you select one button the previously selected button is deselected. To activate a radio button, click the mouse button on the object. The display indicates the state of the button. Use a button group to manage mutually exclusive radio buttons.
"Slider" on page 11-24	`uicontrol`	Sliders accept numeric input within a specified range by enabling the user to move a sliding bar, which is called a slider or thumb. Users move the slider by clicking the slider and dragging it, by clicking in the trough, or by clicking an arrow. The location of the slider indicates the relative location within the specified range.

Component	Function	Description
"Static Text" on page 11-26	`uicontrol`	Static text controls display lines of text. Static text is typically used to label other controls, provide directions to the user, or indicate values associated with a slider. Users cannot change static text interactively.
"Toggle Button" on page 11-27	`uicontrol`	Toggle buttons generate an action and indicate whether they are turned on or off. When you click a toggle button, it appears depressed, showing that it is on. When you release the mouse button, the toggle button remains depressed until you click it a second time. When you do so, the button returns to the raised state, showing that it is off. Use a button group to manage mutually exclusive radio buttons.

Components are sometimes referred to by the name of the function used to create them. For example, a push button is created using the `uicontrol` function, and it is sometimes referred to as a uicontrol. A panel is created using the `uipanel` function and may be referred to as a uipanel.

Adding User Interface Controls

Use the `uicontrol` function to create user interface controls. These include push buttons, toggle buttons, sliders, radio buttons, edit text controls, static text controls, pop-up menus, check boxes, and list boxes.

Note See "Available Components" on page 11-10 for descriptions of these components. See "Programming User Interface Controls" on page 12-15 for basic examples of programming these components.

A syntax for the `uicontrol` function is

```
uich = uicontrol(parent,'PropertyName',PropertyValue,...)
```

where `uich` is the handle of the resulting user interface control. If you do not specify `parent`, the component parent is the current figure as specified by the root `CurrentFigure` property. See the `uicontrol` reference page for other valid syntaxes.

Subsequent topics describe commonly used properties of user interface controls and offer a simple example for each kind of control:

Commonly Used Properties

The most commonly used properties needed to describe a user interface control are shown in the following table:

Property	Values	Description
`Max`	Scalar. Default is 1.	Maximum value. Interpretation depends on the `Style` property.
`Min`	Scalar. Default is 0.	Minimum value. Interpretation depends on the `Style` property.

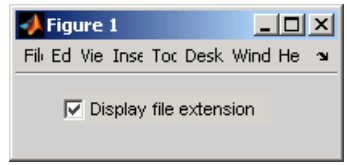
Property	Values	Description
`Position`	4-element vector: [distance from left, distance from bottom, width, height]. Default is [20, 20, 60, 20].	Size of the component and its location relative to its parent.
`String`	String. Can be a cell array or character array or strings.	Component label. For list boxes and pop-up menus it is a list of the items. To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove**.
`Style`	`pushbutton`, `togglebutton`, `radiobutton`, `checkbox`, `edit`, `text`, `slider`, `listbox`, `popupmenu`. Default is `pushbutton`.	Type of user interface control object.
`Units`	`pixels`, `centimeters`, `characters`, `inches`, `normalized`, `points`, Default is `pixels`.	Units of measurement used to interpret position vector
`Value`	Scalar or vector	Value of the component. Interpretation depends on the `Style` property.

For a complete list of properties and for more information about the properties listed in the table, see Uicontrol Properties in the MATLAB Function Reference documentation. Properties needed to control GUI behavior are discussed in Chapter 12, “Programming the GUI” .

Check Box

The following statement creates a check box with handle `cbh`.

```
cbh = uicontrol(fh,'Style','checkbox',...
                'String','Display file extension',...
                'Value',1,'Position',[30 20 130 20]);
```



The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. See “Panel” on page 11-30 and “Button Group” on page 11-32 for more information.

The `Style` property, `checkbox`, specifies the user interface control as a check box.

The `String` property labels the check box as **Display file extension**. The check box accommodates only a single line of text. If you specify a component width that is too small to accommodate the specified `String`, MATLAB truncates the string with an ellipsis.

The `Value` property specifies whether the box is checked. Set `Value` to the value of the `Max` property (default is `1`) to create the component with the box checked. Set `Value` to `Min` (default is `0`) to leave the box unchecked. Correspondingly, when the user clicks the check box, MATLAB sets `Value` to `Max` when the user checks the box and to `Min` when the user unchecks it.

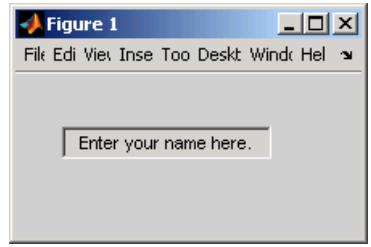
The `Position` property specifies the location and size of the list box. In this example, the list box is 130 pixels wide and 20 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom. The statement assumes the default value of the `Units` property, which is `pixels`.

Note You can also use an image as a label. See “Adding an Image to a Push Button” on page 11-22 for more information.

Edit Text

The following statement creates an edit text component with handle `eth`:

```
eth = uicontrol(fh,'Style','edit',...
                'String','Enter your name here.',...
                'Position',[30 50 130 20]);
```



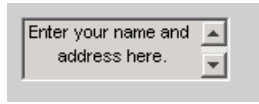
The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. See “Panel” on page 11-30 and “Button Group” on page 11-32 for more information.

The `Style` property, `edit`, specifies the user interface control as an edit text component.

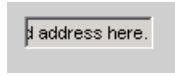
The `String` property defines the text that appears in the component.

To enable multiple-line input, `Max - Min` must be greater than 1, as in the following statement. MATLAB wraps the string if necessary.

```
eth = uicontrol(fh,'Style','edit',...
                'String','Enter your name and address here.',...
                'Max',2,'Min',0,...
                'Position',[30 20 130 80]);
```



If `Max-Min` is less than or equal to 1, the edit text component admits only a single line of input. If you specify a component width that is too small to accommodate the specified string, MATLAB displays only part of the string. The user can use the arrow keys to move the cursor over the entire string.

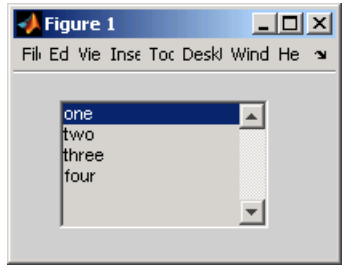


The `Position` property specifies the location and size of the edit text component. In this example, the edit text is 130 pixels wide and 20 high. It is positioned 30 pixels from the left of the figure and 50 pixels from the bottom. The statement assumes the default value of the `Units` property, which is `pixels`.

List Box

The following statement creates a list box with handle `lbh`:

```
lbh = uicontrol(fh,'Style','listbox',...
                'String',{'one','two','three','four'},...
                'Value',1,'Position',[30 80 130 20]);
```



The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. See “Panel” on page 11-30 and “Button Group” on page 11-32 for more information.

The `Style` property, `listbox`, specifies the user interface control as a list box.

The `String` property defines the list items. You can specify the items in any of the formats shown in the following table.

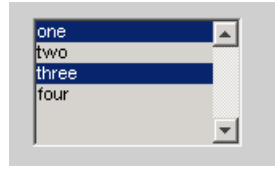
String Property Format	Example
Cell array of strings	`{'one' 'two' 'three'}`
Padded string matrix	`['one ';'two ';'three']`
String vector separated by vertical slash (`|`) characters	`['one|two|three']`

If you specify a component width that is too small to accommodate one or more of the specified strings, MATLAB truncates those strings with an ellipsis.

The `Value` property specifies the item or items that are selected when the component is created. To select a single item, set `Value` to a scalar that indicates the index of the selected list item, where 1 corresponds to the first item in the list.

To select more than one item, set `Value` to a vector of indices of the selected items. To enable selection of more than one item, `Max - Min` must be greater than 1, as in the following statement:

```
lbh = uicontrol(fh,'Style','listbox',...
                'String',{'one','two','three','four'},...
                'Max',2,'Min',0,'Value',[1 3],,...
                'Position',[30 20 130 80]);
```



If you want no initial selection:

1. Set the `Max` and `Min` properties to enable multiple selection
2. Set the `Value` property to an empty matrix `[]`.

If the list box is not large enough to display all list entries, you can set the `ListBoxTop` property to the index of the item you want to appear at the top when the component is created.

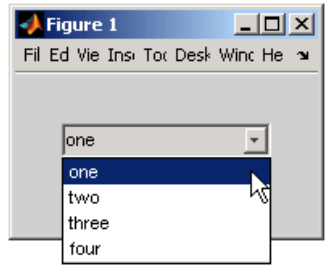
The `Position` property specifies the location and size of the list box. In this example, the list box is 130 pixels wide and 80 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom. The statement assumes the default value of the `Units` property, which is `pixels`.

The list box does not provide for a label. Use a static text component to label the list box.

Pop-Up Menu

The following statement creates a pop-up menu (also known as a drop-down menu or combo box) with handle `pmh`:

```
pmh = uicontrol(fh,'Style','popupmenu',...
                'String',{'one','two','three','four'},...
                'Value',1,'Position',[30 80 130 20]);
```



The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. See “Panel” on page 11-30 and “Button Group” on page 11-32 for more information.

The `Style` property, `popupmenu`, specifies the user interface control as a pop-up menu.

The `String` property defines the menu items. You can specify the items in any of the formats shown in the following table.

String Property Format	Example
Cell array of strings	`{'one' 'two' 'three'}`
Padded string matrix	`['one ';'two ';'three']`
String vector separated by vertical slash (`|`) characters	`['one|two|three']`

If you specify a component width that is too small to accommodate one or more of the specified strings, MATLAB truncates those strings with an ellipsis.

The `Value` property specifies the index of the item that is selected when the component is created. Set `Value` to a scalar that indicates the index of the selected menu item, where 1 corresponds to the first item in the list. In the statement, if `Value` is 2, the menu looks like this when it is created:

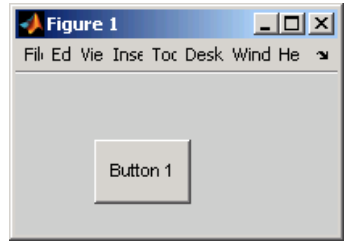
The `Position` property specifies the location and size of the pop-up menu. In this example, the pop-up menu is 130 pixels wide. It is positioned 30 pixels from the left of the figure and 80 pixels from the bottom. The height of a pop-up menu is determined by the font size; the height you set in the position vector is ignored. The statement assumes the default value of the `Units` property, which is `pixels`.

The pop up menu does not provide for a label. Use a static text component to label the pop-up menu.

Push Button

The following statement creates a push button with handle `pbh`:

```
pbh = uicontrol(fh,'Style','pushbutton','String','Button 1',...
                'Position',[50 20 60 40]);
```



The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. See “Panel” on page 11-30 and “Button Group” on page 11-32 for more information.

The `Style` property, `pushbutton`, specifies the user interface control as a push button. Because `pushbutton` is the default style, you can omit the `'Style'` property from the statement.

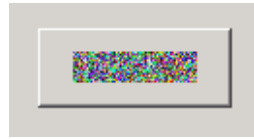
The `String` property labels the push button as **Button 1**. The push button allows only a single line of text. If you specify more than one line, only the first line is shown. If you specify a component width that is too small to accommodate the specified `String`, MATLAB truncates the string with an ellipsis.

The `Position` property specifies the location and size of the push button. In this example, the push button is 60 pixels wide and 40 high. It is positioned 50 pixels from the left of the figure and 20 pixels from the bottom. This statement assumes the default value of the `Units` property, which is `pixels`.

Adding an Image to a Push Button. To add an image to a push button, assign the button's `CData` property an m-by-n-by-3 array of RGB values that defines a truecolor image. For example, the array `img` defines 16-by-64 truecolor image using random values between 0 and 1 (generated by `rand`).

```
img(:,:,1) = rand(16,64);
img(:,:,2) = rand(16,64);
img(:,:,3) = rand(16,64);
```

```
pbh = uicontrol(fh,'Style','pushbutton',...
                'Position',[50 20 100 45],...
                'CData',img);
```

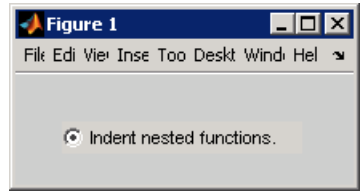


Note Create your own icon with the icon editor described in "Icon Editor" on page 15-29. See `ind2rgb` for information on converting a matrix `X` and corresponding colormap, i.e., an `(X, MAP)` image, to RGB (truecolor) format.

Radio Button

The following statement creates a radio button with handle `rbh`:

```
rbh = uicontrol(fh,'Style','radiobutton',...
                'String','Indent nested functions.',...
                'Value',1,'Position',[30 20 150 20]);
```



The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. Use a button group to manage exclusive selection of radio buttons and toggle buttons. See "Panel" on page 11-30 and "Button Group" on page 11-32 for more information.

The `Style` property, `radiobutton`, specifies the user interface control as a radio button.

The `String` property labels the radio button as **Indent nested functions**. The radio button allows only a single line of text. If you specify more than

one line, only the first line is shown. If you specify a component width that is too small to accommodate the specified `String`, MATLAB truncates the string with an ellipsis.

The `Value` property specifies whether the radio button is selected when the component is created. Set `Value` to the value of the `Max` property (default is `1`) to create the component with the radio button selected. Set `Value` to `Min` (default is `0`) to leave the radio button unselected.

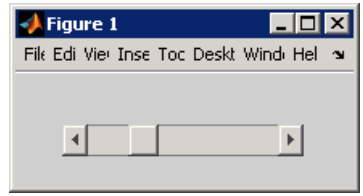
The `Position` property specifies the location and size of the radio button. In this example, the radio button is 150 pixels wide and 20 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom. The statement assumes the default value of the `Units` property, which is `pixels`.

Note You can also use an image as a label. See "Adding an Image to a Push Button" on page 11-22 for more information.

Slider

The following statement creates a slider with handle `sh`:

```
sh = uicontrol(fh,'Style','slider',...
                  'Max',100,'Min',0,'Value',25,...
                  'SliderStep',[0.05 0.2],...
                  'Position',[30 20 150 30]);
```



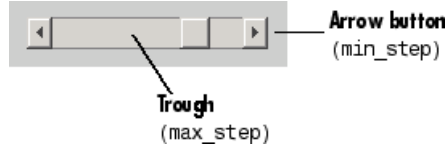
The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. See “Panel” on page 11-30 and “Button Group” on page 11-32 for more information.

The `Style` property, `slider`, specifies the user interface control as a slider.

The `Max` property is the maximum value of the slider. The `Min` property is the minimum value of the slider and must be less than `Max`.

The `Value` property specifies the value indicated by the slider when it is created. Set `Value` to a number that is less than or equal to `Max` and greater than or equal to `Min`. If you specify `Value` outside the specified range, the slider is not rendered.

The `SliderStep` property controls the amount the slider `Value` changes when a user clicks the arrow button to produce a minimum step or the slider trough to produce a maximum step. Specify `SliderStep` as a two-element vector, `[min_step,max_step]`, where each value is in the range [0, 1].



The example provides a 5 percent minimum step and a 20 percent maximum step. The default, `[0.01 0.10]`, provides a 1 percent minimum step and a 10 percent maximum step.

The `Position` property specifies the location and size of the slider. In this example, the slider is 150 pixels wide and 30 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom. The statement assumes the default value of the `Units` property, which is `pixels`.

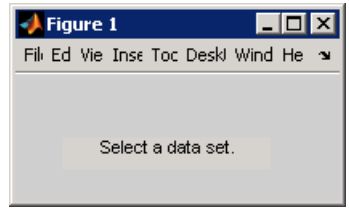
Note On Mac platforms, the height of a horizontal slider is constrained. If the height you set in the position vector exceeds this constraint, the displayed height of the slider is the maximum allowed. The height element of the position vector is not changed.

The slider component provides no text description. Use static text components to label the slider.

Static Text

The following statement creates a static text component with handle `sth`:

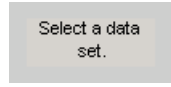
```
sth = uicontrol(fh,'Style','text',...
                'String','Select a data set.',...
                'Position',[30 50 130 20]);
```



The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. See “Panel” on page 11-30 and “Button Group” on page 11-32 for more information.

The `Style` property, `text`, specifies the user interface control as a static text component.

The `String` property defines the text that appears in the component. If you specify a component width that is too small to accommodate the specified `String`, MATLAB wraps the string.

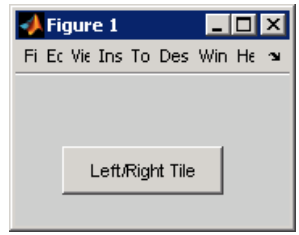


The `Position` property specifies the location and size of the static text component. In this example, the static text is 130 pixels wide and 20 high. It is positioned 30 pixels from the left of the figure and 50 pixels from the bottom. The statement assumes the default value of the `Units` property, which is `pixels`.

Toggle Button

The following statement creates a toggle button with handle `tbh`:

```
tbh = uicontrol(fh,'Style','togglebutton',...
                'String','Left/Right Tile',...
                'Value',0,'Position',[30 20 100 30]);
```



The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. Use a button group to manage exclusive selection of radio buttons and toggle buttons. See “Panel” on page 11-30 and “Button Group” on page 11-32 for more information.

The `Style` property, `togglebutton`, specifies the user interface control as a toggle button.

The `String` property labels the toggle button as **Left/Right Tile**. The toggle button allows only a single line of text. If you specify more than one line, only the first line is shown. If you specify a component width that is too small to accommodate the specified `String`, MATLAB truncates the string with an ellipsis.

The `Value` property specifies whether the toggle button is selected when the component is created. Set `Value` to the value of the `Max` property (default is `1`) to create the component with the toggle button selected (depressed). Set `Value` to `Min` (default is `0`) to leave the toggle button unselected (raised). The following figure shows the toggle button in the depressed position.

The `Position` property specifies the location and size of the toggle button. In this example, the toggle button is 100 pixels wide and 30 high. It is positioned 30 pixels from the left of the figure and 20 pixels from the bottom. The statement assumes the default value of the `Units` property, which is `pixels`.

Note You can also use an image as a label. See "Adding an Image to a Push Button" on page 11-22 for more information.

Adding Panels and Button Groups

Panels and button groups are containers that arrange GUI components into groups. If you move the panel or button group, its children move with it and maintain their positions relative to the panel or button group.

Note See "Available Components" on page 11-10 for descriptions of these components.

Use the `uipanel` and `uibuttongroup` functions to create these components.

A syntax for panels is

```
ph = uipanel(fh,'PropertyName',PropertyValue,...)
```

where `ph` is the handle of the resulting panel. The first argument, `fh`, specifies the handle of the parent figure. You can also specify the parent as a panel or button group. See the `uipanel` reference page for other valid syntaxes.

A syntax for button groups is

```
bgh = uibuttongroup('PropertyName',PropertyValue,...)
```

where `bgh` is the handle of the resulting button group. For button groups, you must use the `Parent` property to specify the component parent. See the `uibuttongroup` reference page for other valid syntaxes.

For both panels and button groups, if you do not specify a parent, the component parent is the current figure as specified by the root `CurrentFigure` property.

Subsequent topics describe commonly used properties of panels and button groups and offer a simple example for each component.

- "Commonly Used Properties" on page 11-29
- "Panel" on page 11-30
- "Button Group" on page 11-32

Commonly Used Properties

The most commonly used properties needed to describe a panel or button group are shown in the following table:

Property	Values	Description
`Parent`	Handle	Handle of the component's parent figure, panel, or button group.
`Position`	4-element vector: [distance from left, distance from bottom, width, height]. Default is [0, 0, 1, 1].	Size of the component and its location relative to its parent.

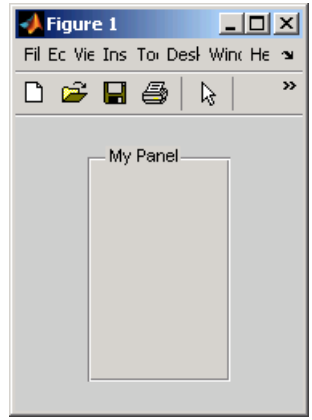
Property	Values	Description
`Title`	String	Component label. To display the & character in a label, use two & characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove**.
`TitlePosition`	`lefttop`, `centertop`, `righttop`, `leftbottom`, `centerbottom`, `rightbottom`. Default is `lefttop`.	Location of title string in relation to the panel or button group.
`Units`	`normalized`, `centimeters`, `characters`, `inches`, `pixels`, `points`. Default is `normalized`.	Units of measurement used to interpret position vector

For a complete list of properties and for more information about the properties listed in the table, see Uipanel Properties and Uibuttongroup Properties in the MATLAB Function Reference documentation. Properties needed to control GUI behavior are discussed in Chapter 12, “Programming the GUI”.

Panel

The following statement creates a panel with handle `ph`. Use a panel to group components in the GUI.

```
ph = uipanel('Parent',fh,'Title','My Panel',...
             'Position',[.25 .1 .5 .8]);
```



The `Parent` property specifies the handle `fh` of the parent figure. You can also specify the parent as a panel or button group.

The `Title` property labels the panel as **My Panel**.

The statement assumes the default `TitlePosition` property, which is `lefttop`.

The `Units` property is used to interpret the `Position` property. This panel assumes the default `Units` property, `normalized`. This enables the panel to resize automatically if the figure is resized.

The `Position` property specifies the location and size of the panel. In this example, the panel is 50 percent of the width of the figure and 80 percent of its height. It is positioned 25 percent of the figure width from the left of the figure and 10 percent of the figure height from the bottom. As the figure is resized the panel retains these proportions.

The following statements add two push buttons to the panel with handle `ph`. The `Position` property of each component within a panel is interpreted relative to the panel.

```
pbh1 = uicontrol(ph,'Style','pushbutton','String','Button 1',...
                 'Units','normalized',...
                 'Position',[.1 .55 .8 .3]);
pbh2 = uicontrol(ph,'Style','pushbutton','String','Button 2',...
```

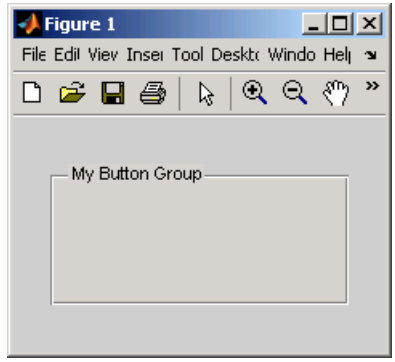
```
                'Units','normalized',...
                'Position',[.1 .15 .8 .3]);
```

See “Push Button” on page 11-21 for more information about adding push buttons.

Button Group

The following statement creates a button group with handle `bgh`. Use a button group to exclusively manage radio buttons and toggle buttons.

```
bgh = uibuttongroup('Parent',fh,'Title','My Button Group',...
            'Position',[.1 .2 .8 .6]);
```



The `Parent` property specifies the handle `fh` of the parent figure. You can also specify the parent as a panel or button group.

The `Title` property labels the button group as **My Button Group**.

The statement assumes the default `TitlePosition` property, which is `lefttop`.

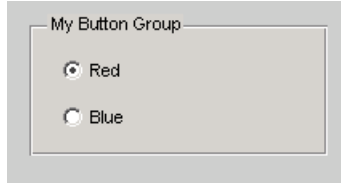
The `Units` property is used to interpret the `Position` property. This button group assumes the default `Units` property, `normalized`. This enables the button group to resize automatically if the figure is resized.

The `Position` property specifies the location and size of the button group. In this example, the button group is 80 percent of the width of the figure and 60 percent of its height. It is positioned 10 percent of the figure width from the left of the figure and 20 percent of the figure height from the bottom. As the figure is resized the button group retains these proportions.

The following statements add two radio buttons to the button group with handle `bgh`.

```
rbh1 = uicontrol(bgh,'Style','radiobutton','String','Red',...
                 'Units','normalized',...
                 'Position',[.1 .6 .3 .2]);
rbh2 = uicontrol(bgh,'Style','radiobutton','String','Blue',...
                 'Units','normalized',...
                 'Position',[.1 .2 .3 .2]);
```

By default, MATLAB automatically selects the first radio button added to a button group. You can use the radio button `Value` property to explicitly specify the initial selection. See “Radio Button” on page 11-23 for information.



Adding Axes

Axes enable your GUI to display graphics such as graphs and images using commands such as: `plot`, `surf`, `line`, `bar`, `polar`, `pie`, `contour`, and `mesh`.

Note See "Available Components" on page 11-10 for a description of this component.

Use the `axes` function to create an axes. A syntax for this function is

```
ah = axes('PropertyName',PropertyValue,...)
```

where `ah` is the handle of the resulting axes. You must use the `Parent` property to specify the axes parent. If you do not specify `Parent`, the parent is the current figure as specified by the root `CurrentFigure` property. See the `axes` reference page for other valid syntaxes.

Subsequent topics describe commonly used properties of axes and offer a simple example.

- "Commonly Used Properties" on page 11-34
- "Axes" on page 11-35

Commonly Used Properties

The most commonly used properties needed to describe an axes are shown in the following table:

Property	Values	Description
`HandleVisibility`	`on`, `callback`, `off`. Default is `on`.	Determines if an object's handle is visible in its parent's list of children. For axes, set `HandleVisibility` to `callback` to protect them from command line operations.
`Parent`	Handle	Handle of the component's parent figure, panel, or button group.

Property	Values	Description
`Position`	4-element vector: [distance from left, distance from bottom, width, height].	Size of the component and its location relative to its parent.
`Units`	`normalized`, `centimeters`, `characters`, `inches`, `pixels`, `points`. Default is `normalized`.	Units of measurement used to interpret position vector

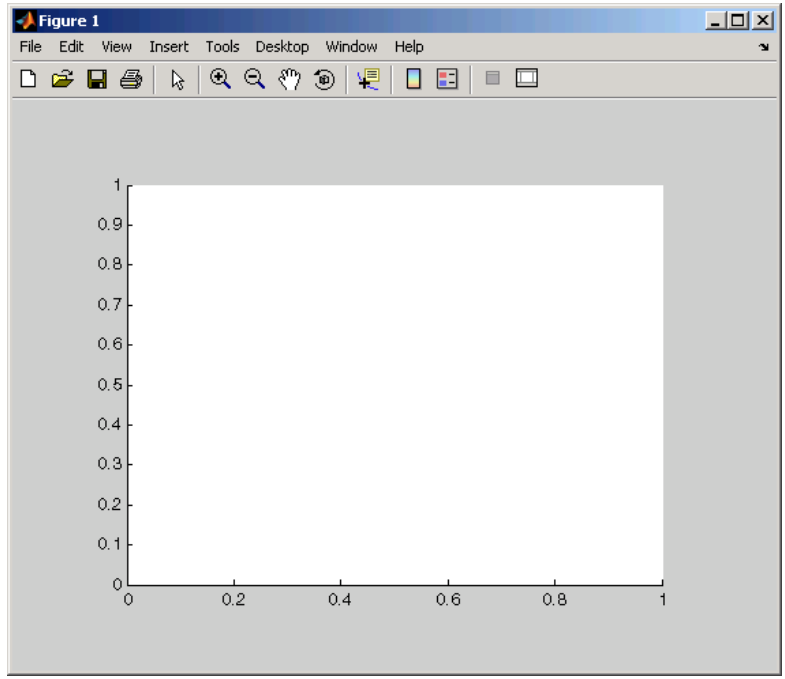
For a complete list of properties and for more information about the properties listed in the table, see Axes Properties in the MATLAB Function Reference documentation. Properties needed to control GUI behavior are discussed in Chapter 12, "Programming the GUI".

See commands such as the following for more information on axes objects: `plot`, `surf`, `line`, `bar`, `polar`, `pie`, `contour` and `mesh`. See "Functions — By Category" in the MATLAB Function Reference documentation for a complete list.

Axes

The following statement creates an axes with handle `ah`:

```
ah = axes('Parent',fh,'Position',[.15 .15 .7 .7]);
```



The `Parent` property specifies the handle `fh` of the parent figure. You can also specify the parent as a panel or button group.

The `Units` property is used to interpret the `Position` property. This axes assumes the default `Units` property, `normalized`. This enables the axes to resize automatically if the figure is resized.

The `Position` property specifies the location and size of the axes. In this example, the axes is 70 percent of the width of the figure and 70 percent of its height. It is positioned 15 percent of the figure width from the left of the figure and 15 percent of the figure height from the bottom. As the figure is resized the axes retains these proportions.

MATLAB automatically adds the tick marks. Most functions that draw in the axes update the tick marks appropriately.

Adding ActiveX Controls

ActiveX components enable you to display ActiveX controls in your GUI. They are available only on the Microsoft Windows platform.

An ActiveX control can be the child only of a figure, i.e., of the GUI itself. It cannot be the child of a panel or button group.

See “Creating an ActiveX Control” in the MATLAB External Interfaces documentation for information about adding an ActiveX control to a figure. See “MATLAB COM Client Support” in the MATLAB External Interfaces documentation for general information about ActiveX controls.

Aligning Components

In this section...

Using the `Align` Function

Use the `align` function to align user interface controls and axes. This function enables you to align the components vertically and horizontally. You can also distribute the components evenly, or specify a fixed distance between them.

A syntax for the `align` function is

```
align(HandleList,'HorizontalAlignment',...
                 'VerticalAlignment')
```

where *`HorizontalAlignment`* can be `None`, `Left`, `Center`, `Right`, `Distribute`, or `Fixed` and *`VerticalAlignment`* can be `None`, `Top`, `Middle`, `Bottom`, `Distribute`, or `Fixed`. All handles in `HandleList` must have the same parent. See the `align` reference page for information about other syntaxes.

The following code creates three push buttons that are somewhat randomly placed. Each subsequent example starts with these same three push buttons and aligns them in different ways. Components are aligned with reference to their bounding box, shown as a blue dashed line in the figures.

```
b1 = uicontrol(fh,'Posit',[30 10 60 30],'String','Button 1');
b2 = uicontrol(fh,'Posit',[50 50 60 30],'String','Button 2');
b3 = uicontrol(fh,'Posit',[10 80 60 30],'String','Button 3');
```

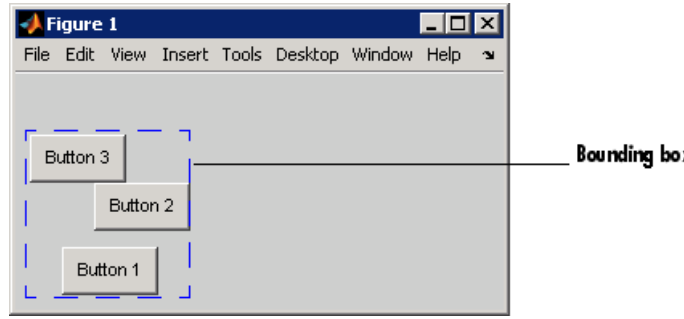


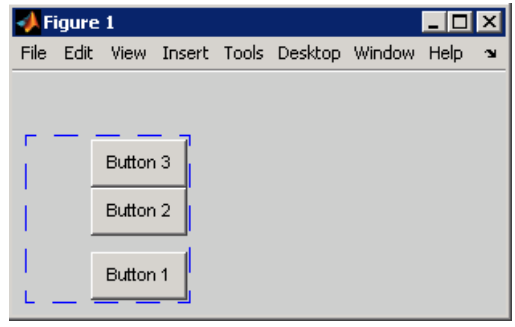
Figure 1
File Edit View Insert Tools Desktop Window Help
Button 3
Button 2
Button 1
Bounding box

Examples

Aligning Components Horizontally

The following statement moves the push buttons horizontally to the right of their bounding box. It does not alter their vertical positions. The figure shows the original bounding box.

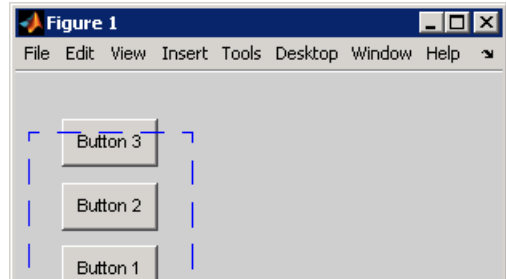
```
align([b1 b2 b3],'Right','None');
```



Aligning Components Horizontally While Distributing Them Vertically

The following statement moves the push buttons horizontally to the center of their bounding box and adjusts their vertical placement to create a fixed distance of 7 points between the boxes. The push buttons appear in the center of the original bounding box. The bottom push button remains at the bottom of the original bounding box.

```
align([b1 b2 b3],'Center','Fixed',7);
```



Setting Tab Order

In this section...

How Tabbing Works

A GUI's tab order is the order in which components of the GUI acquire focus when a user presses the keyboard **Tab** key. Focus is generally denoted by a border or a dotted border.

Tab order is determined separately for the children of each parent. For example, child components of the GUI figure have their own tab order. Child components of each panel or button group also have their own tab order.

If, in tabbing through the components at one level, a user tabs to a panel or button group, then the tabbing sequences through the components of the panel or button group before returning to the level from which the panel or button group was reached. For example, if a GUI figure contains a panel that contains three push buttons and the user tabs to the panel, then the tabbing sequences through the three push buttons before returning to the figure.

Note You cannot tab to axes and static text components. You cannot determine programmatically which component has focus.

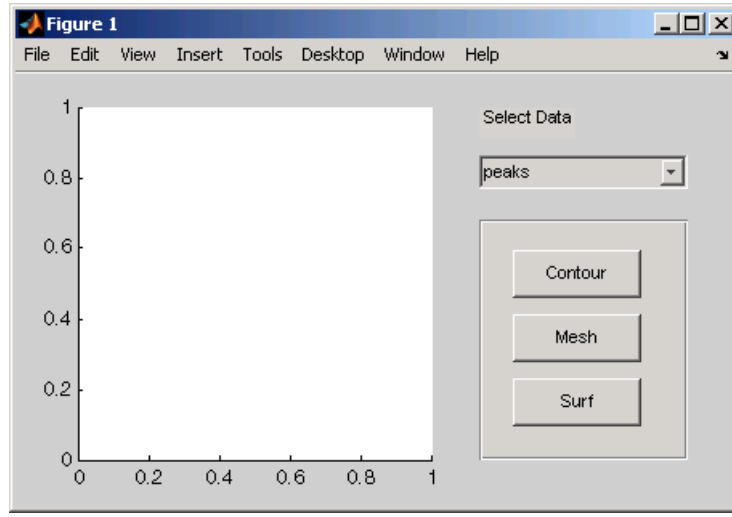
Default Tab Order

The default tab order for each level is the order in which you create the components at that level.

The following code creates a GUI that contains a pop-up menu with a static text label, a panel with three push buttons, and an axes.

```
fh = figure('Position',[200 200 450 270]);
pmh = uicontrol(fh,'Style','popupmenu',...
```

```
            'String',{'peaks','membrane','sinc'},...
            'Position',[290 200 130 20]);
sth = uicontrol(fh,'Style','text','String','Select Data',...
            'Position',[290 230 60 20]);
ph = uipanel('Parent',fh,'Units','pixels',...
            'Position',[290 30 130 150]);
ah = axes('Parent',fh,'Units','pixels',...
            'Position',[40 30 220 220]);
bh1 = uicontrol(ph,'Style','pushbutton',...
            'String','Contour','Position',[20 20 80 30]);
bh2 = uicontrol(ph,'Style','pushbutton',...
            'String','Mesh','Position',[20 60 80 30]);
bh3 = uicontrol(ph,'Style','pushbutton',...
            'String','Surf','Position',[20 100 80 30]);
```



You can obtain the default tab order for a figure, panel, or button group by retrieving its `Children` property. For the example, the statement is

```
ch = get(ph,'Children')
```

where `ph` is the handle of the panel. This statement returns a vector containing the handles of the children, the three push buttons.

```
ch =
```

```
4.0076
3.0076
2.0076
```

These handles correspond to the push buttons as shown in the following table:

Handle	Handle Variable	Push Button
4.0076	bh3	**Surf**
3.0076	bh2	**Mesh**
2.0076	bh1	**Contour**

The default tab order of the push buttons is the reverse of the order of the child vector: **Contour > Mesh > Surf**.

Note The `get` function returns only those children whose handles are visible, i.e., those with their `HandleVisibility` property set to `on`. Use `allchild` to retrieve children regardless of their handle visibility.

In the example GUI figure, the default order is pop-up menu followed by the panel's **Contour**, **Mesh**, and **Surf** push buttons (in that order), and then back to the pop-up menu. You cannot tab to the axes component or the static text component.

Try modifying the code to create the pop-up menu following the creation of the **Contour** push button and before the **Mesh** push button. Now execute the code to create the GUI and tab through the components. This code change does not alter the default tab order. This is because the pop-up menu does not have the same parent as the push buttons. The figure is the parent of the panel and the pop-up menu.

Changing the Tab Order

Use the `uistack` function to change the tab order of components that have the same parent. A convenient syntax for `uistack` is

```
uistack(h,stackopt,step)
```

where `h` is a vector of handles of the components whose tab order is to be changed.

`stackopt` represents the direction of the move. It must be one of the strings: `up`, `down`, `top`, or `bottom`, and is interpreted relative to the column vector returned by the statement:

```
ch = get(ph,'Children')

ch =
   4.0076
   3.0076
   2.0076
```

If the tab order is currently **Contour > Mesh > Surf**, the statement

```
uistack(bh2,up,1)
```

moves `bh2` (**Surf**) up one place in the vector of children and changes the tab order to **Contour > Surf > Mesh**.

```
ch = get(ph,'Children')
```

now returns

```
ch =
   3.0076
   4.0076
   2.0076
```

`step` is the number of levels changed. The default is 1.

Note Tab order also affects the stacking order of components. If components overlap, those that appear lower in the child order, are drawn on top of those that appear higher in the order. If the push buttons in the example overlapped, the **Contour** push button would be on top.

Creating Menus

In this section...

Adding Menu Bar Menus

Use the `uimenu` function to add a menu bar menu to your GUI. A syntax for `uimenu` is

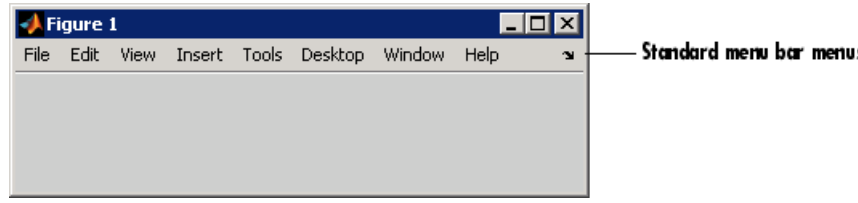
```
mh = uimenu(parent,'PropertyName',PropertyValue,...)
```

Where `mh` is the handle of the resulting menu or menu item. See the `uimenu` reference page for other valid syntaxes.

These topics discuss use of the MATLAB standard menu bar menus and describe commonly used menu properties and offer some simple examples.

Displaying Standard Menu Bar Menus

Displaying the standard menu bar menus is optional.



If you use the standard menu bar menus, any menus you create are added to it. If you choose not to display the standard menu bar menus, the menu bar contains only the menus that you create. If you display no standard menus and you create no menus, the menu bar itself is not displayed.

Use the figure `MenuBar` property to display or hide the MATLAB standard menus shown in the preceding figure. Set `MenuBar` to `figure` (the default) to display the standard menus. Set `MenuBar` to `none` to hide them.

```
set(fh,'MenuBar','figure');  % Display standard menu bar menus.
set(fh,'MenuBar','none');    % Hide standard menu bar menus.
```

In these statements, `fh` is the handle of the figure.

Commonly Used Properties

The most commonly used properties needed to describe a menu bar menu are shown in the following table.

Property	Values	Description
`Accelerator`	Alphabetic character	Keyboard equivalent. Available for menu items that do not have submenus.
`Checked`	`off`, `on`. Default is `off`.	Menu check indicator
`Enable`	`on`, `off`. Default is `on`.	Controls whether a menu item can be selected. When set to `off`, the menu label appears dimmed.
`HandleVisibility`	`on`, `off`. Default is `on`.	Determines if an object's handle is visible in its parent's list of children. For menus, set `HandleVisibility` to `off` to protect menus from operations not intended for them.

Property	Values	Description
`Label`	String	Menu label. To display the `&` character in a label, use two `&` characters in the string. The words `remove`, `default`, and `factory` (case sensitive) are reserved. To use one of these as a label, prepend a backslash (`\`) to the string. For example, `\remove` yields **remove**.
`Position`	Scalar. Default is `1`.	Position of a menu item in the menu.
`Separator`	`off`, `on`. Default is `off`.	Separator line mode

For a complete list of properties and for more information about the properties listed in the table, see Uimenu Properties in the MATLAB Function Reference documentation. Properties needed to control GUI behavior are discussed in Chapter 12, “Programming the GUI”.

Menu Bar Menu

The following statements create a menu bar menu with two menu items.

```
mh = uimenu(fh,'Label','My menu');
eh1 = uimenu(mh,'Label','Item 1');
eh2 = uimenu(mh,'Label','Item 2','Checked','on');
```

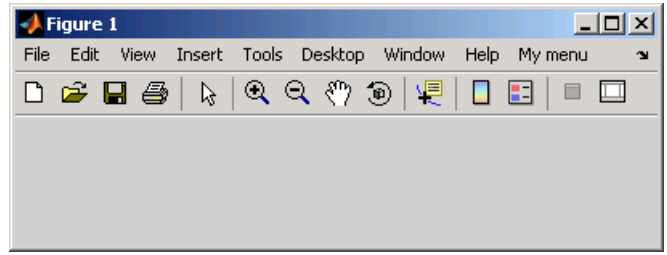
`fh` is the handle of the parent figure.

`mh` is the handle of the parent menu.

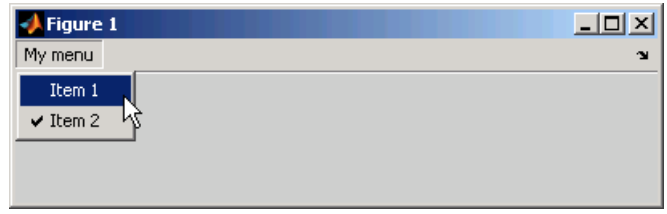
The `Label` property specifies the text that appears in the menu.

The `Checked` property specifies that this item is displayed with a check next to it when the menu is created.

If your GUI displays the standard menu bar, the new menu is added to it.

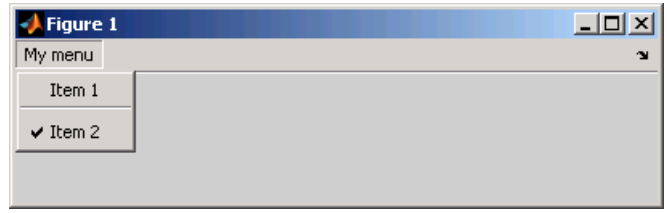


If your GUI does not display the standard menu bar, MATLAB creates a menu bar if none exists and then adds the menu to it.



The following statement adds a separator line preceding the second menu item.

```
set(eh2,'Separator','on');
```



The following statements add two menu subitems to **Item 1**, assign each subitem a keyboard accelerator, and disable the first subitem.

```
seh1 = uimenu(eh1,'Label','Choice 1','Accelerator','C',...
              'Enable','off');
seh2 = uimenu(eh1,'Label','Choice 2','Accelerator','H');
```

The `Accelerator` property adds keyboard accelerators to the menu items. Some accelerators may be used for other purposes on your system and other actions may result.

The `Enable` property disables the first subitem **Choice 1** so a user cannot select it when the menu is first created. The item appears dimmed.

Note After you have created all menu items, set their `HandleVisibility` properties `off` by executing the following statements:

```
menuhandles = findall(figurehandle,'type','uimenu');
set(menuhandles,'HandleVisibility','off')
```

See “Programming Menu Items” on page 12-28 for information about programming menu items.

Adding Context Menus

Context menus appear when the user right-clicks on a figure or GUI component. Follow these steps to add a context menu to your GUI:

1. Create the context menu object using the `uicontextmenu` function.
2. Add menu items to the context menu using the `uimenu` function.
3. Associate the context menu with a graphics object using the object’s `UIContextMenu` property.

Subsequent topics describe commonly used context menu properties and explain each of these steps:

Commonly Used Properties

The most commonly used properties needed to describe a context menu object are shown in the following table. These properties apply only to the menu object and not to the individual menu items.

Property	Values	Description
`HandleVisibility`	`on`, `off`. Default is `on`.	Determines if an object's handle is visible in its parent's list of children. For menus, set `HandleVisibility` to `off` to protect menus from operations not intended for them.
`Parent`	Figure handle	Handle of the context menu's parent figure.
`Position`	2-element vector: [distance from left, distance from bottom]. Default is `[0 0]`.	Distances from the bottom left corner of the parent figure to the top left corner of the context menu. This property is used only when you programmatically set the context menu `Visible` property to `on`.
`Visible`	`off`, `on`. Default is `off`	• Indicates whether the context menu is currently displayed. While the context menu is displayed, the property value is `on`; when the context menu is not displayed, its value is `off`. • Setting the value to `on` forces the posting of the context menu. Setting to `off` forces the context menu to be removed. The `Position` property determines the location where the context menu is displayed.

For a complete list of properties and for more information about the properties listed in the table, see the Uicontextmenu Properties reference page in the MATLAB Function Reference documentation. Properties needed to control GUI behavior are discussed in Chapter 12, “Programming the GUI”.

Creating the Context Menu Object

Use the `uicontextmenu` function to create a context menu object. The syntax is

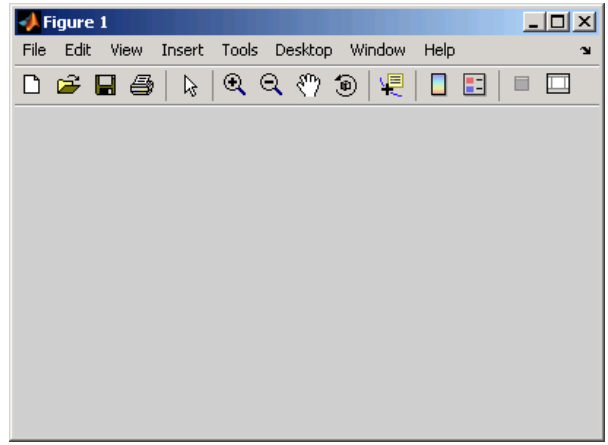
```
handle = uicontextmenu('PropertyName',PropertyValue,...)
```

The parent of a context menu must always be a figure. Use the context menu `Parent` property to specify its parent. If you do not specify `Parent`, the parent is the current figure as specified by the root `CurrentFigure` property.

The following code creates a figure and a context menu whose parent is the figure.

```
fh = figure('Position',[300 300 400 225]);
cmenu = uicontextmenu('Parent',fh,'Position',[10 215]);
```

At this point, the figure is visible, but not the menu.



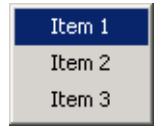
Note "Forcing Display of the Context Menu" on page 11-54 explains the use of the `Position` property.

Adding Menu Items to the Context Menu

Use the `uimenu` function to add items to the context menu. The items appear on the menu in the order in which you add them. The following code adds three items to the context menu created above.

```
mh1 = uimenu(cmenu,'Label','Item 1');
mh2 = uimenu(cmenu,'Label','Item 2');
mh3 = uimenu(cmenu,'Label','Item 3');
```

If you could see the context menu, it would look like this:



You can use any applicable Uimenu Properties such as `Checked` or `Separator` when you define context menu items. See the `uimenu` reference page and "Adding Menu Bar Menus" on page 11-45 for information about using `uimenu` to create menu items. Note that context menus do not have an `Accelerator` property.

Note After you have created the context menu and all its items, set their `HandleVisibility` properties to `off` by executing the following statements:

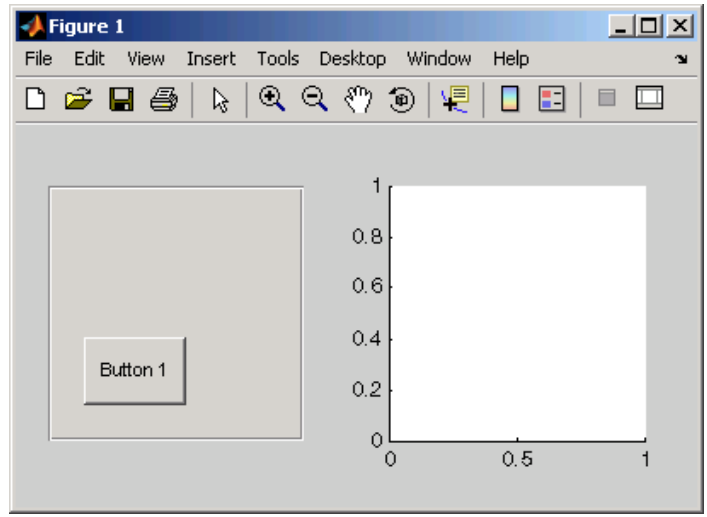
```
cmenuhandles = findall(figurehandle,'type','uicontextmenu');
set(cmenuhandles,'HandleVisibility','off')
menuitemhandles = findall(cmenuhandles,'type','uimenu');
set(menuitemhandles,'HandleVisibility','off')
```

Associating the Context Menu with Graphics Objects

You can associate a context menu with the figure itself and with all components that have a `UIContextMenu` property. This includes axes, panel, button group, all user interface controls (uicontrols).

The following code adds a panel and an axes to the figure. The panel contains a single push button.

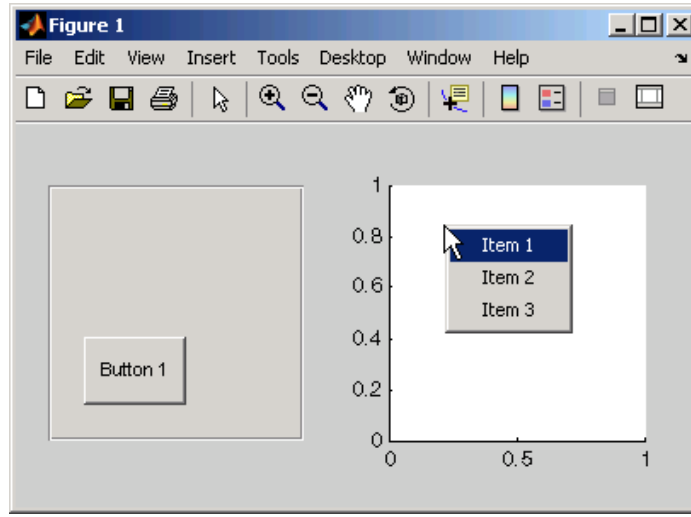
```
ph = uipanel('Parent',fh,'Units','pixels',...
             'Position',[20 40 150 150]);
bh1 = uicontrol(ph,'String','Button 1',...
             'Position',[20 20 60 40]);
ah = axes('Parent',fh,'Units','pixels',...
             'Position',[220 40 150 150]);
```



This code associates the context menu with the figure and with the axes by setting the `UIContextMenu` property of the figure and the axes to the handle `cmenu` of the context menu.

```
set(fh,'UIContextMenu',cmenu);   % Figure
set(ah,'UIContextMenu',cmenu);   % Axes
```

Right-click on the figure or on the axes. The context menu appears with its upper-left corner at the location you clicked. Right-click on the panel or its push button. The context menu does not appear.

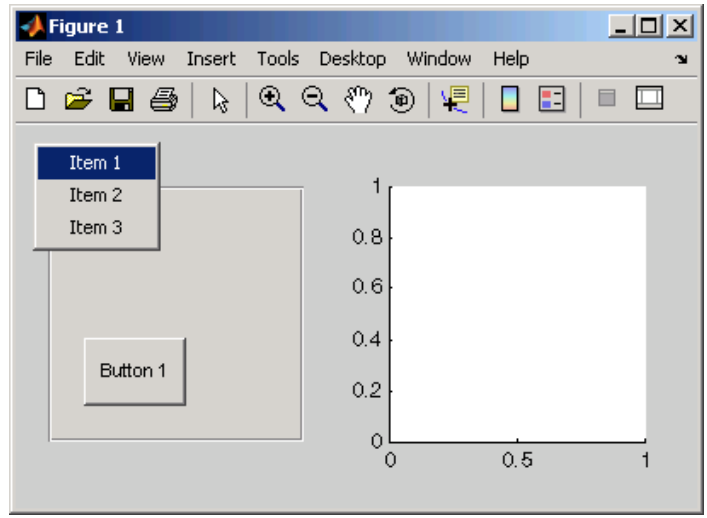


Forcing Display of the Context Menu

If you set the context menu `Visible` property `on`, the context menu is displayed at the location specified by the `Position` property, without the user taking any action. In this example, the context menu `Position` property is [10 215].

```
set(cmenu,'Visible','on');
```

The context menu is displayed 10 pixels from the left of the figure and 215 pixels from the bottom.



If you set the context menu `Visible` property to `off`, or if the user clicks the GUI outside the context menu, the context menu disappears.

Creating Toolbars

In this section...

Using the `uitoolbar` Function

Use the `uitoolbar` function to add a custom toolbar to your GUI. Use the `uipushtool` and `uitoggletool` functions to add push tools and toggle tools to a toolbar. A push tool functions as a push button. A toggle tool functions as a toggle button. You can add push tools and toggle tools to the standard toolbar or to a custom toolbar.

Syntaxes for the `uitoolbar`, `uipushtool`, and `uitoggletool` functions include

```
tbh = uitoolbar(h,'PropertyName',PropertyValue,...)
pth = uipushtool(h,'PropertyName',PropertyValue,...)
tth = uitoggletool(h,'PropertyName',PropertyValue,...)
```

where `tbh`, `pth`, and `tth` are the handles, respectively, of the resulting toolbar, push tool, and toggle tool. See the `uitoolbar`, `uipushtool`, and `uitoggletool` reference pages for other valid syntaxes.

Subsequent topics describe commonly used properties of toolbars and toolbar tools, offer a simple example, and discuss use of the MATLAB standard toolbar:

Commonly Used Properties

The most commonly used properties needed to describe a toolbar and its tools are shown in the following table.

Property	Values	Description
`CData`	3-D array of values between 0.0 and 1.0	n-by-m-by-3 array of RGB values that defines a truecolor image displayed on either a push button or toggle button.
`HandleVisibility`	`on`, `off`. Default is `on`.	Determines if an object's handle is visible in its parent's list of children. For toolbars and their tools, set `HandleVisibility` to `off` to protect them from operations not intended for them.
`Separator`	`off`, `on`. Default is `off`.	Draws a dividing line to left of the push tool or toggle tool
`State`	`off`, `on`. Default is `off`.	Toggle tool state. `on` is the down, or depressed, position. `off` is the up, or raised, position.
`TooltipString`	String	Text of the tooltip associated with the push tool or toggle tool.

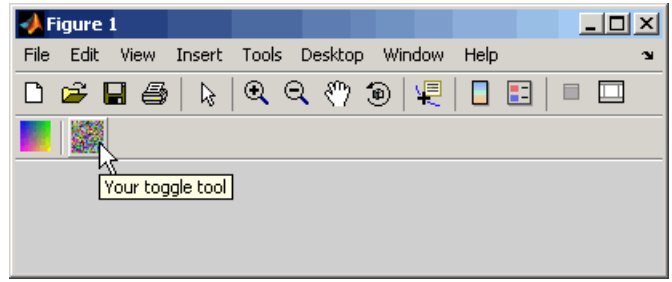
For a complete list of properties and for more information about the properties listed in the table, see the Uitoolbar Properties, Uipushtool Properties, and Uitoggletool Properties reference pages in the MATLAB Function Reference documentation. Properties needed to control GUI behavior are discussed in Chapter 12, "Programming the GUI".

Toolbars

The following statements add a toolbar to a figure, and then add a push tool and a toggle tool to the toolbar. By default, the tools are added to the toolbar, from left to right, in the order they are created.

```
% Create the toolbar
th = uitoolbar(fh);
```

```
% Add a push tool to the toolbar
a = [.20:.05:0.95]
img1(:,:,1) = repmat(a,16,1)'
img1(:,:,2) = repmat(a,16,1);
img1(:,:,3) = repmat(flipdim(a,2),16,1);
pth = uipushtool(th,'CData',img1,...
           'TooltipString','My push tool',...
           'HandleVisibility','off')
% Add a toggle tool to the toolbar
img2 = rand(16,16,3);
tth = uitoggletool(th,'CData',img2,'Separator','on',...
           'TooltipString','Your toggle tool',...
           'HandleVisibility','off')
```



`fh` is the handle of the parent figure.

`th` is the handle of the parent toolbar.

`CData` is a 16-by-16-by-3 array of values between 0 and 1. It defines the truecolor image that is displayed on the tool. If your image is larger than 16 pixels in either dimension, it may be clipped or cause other undesirable effects. If the array is clipped, only the center 16-by-16 part of the array is used.

Note Create your own icon with the icon editor described in "Icon Editor" on page 15-29. See the `ind2rgb` reference page for information on converting a matrix `X` and corresponding colormap, i.e., an `(X, MAP)` image, to RGB (truecolor) format.

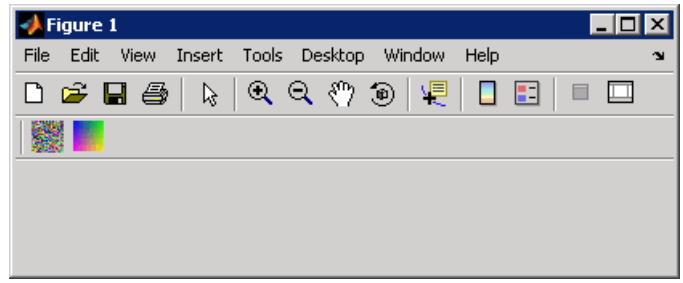
`TooltipString` specifies the tooltips for the push tool and the toggle tool as `My push tool` and `Your toggle tool`, respectively.

In this example, setting the toggle tool `Separator` property to `on` creates a dividing line to the left of the toggle tool.

You can change the order of the tools by modifying the child vector of the parent toolbar. For this example, execute the following code to reverse the order of the tools.

```
oldOrder = allchild(th);
newOrder = flipud(oldOrder);
set(th,'Children',newOrder);
```

This code uses `flipud` because the `Children` property is a column vector.



Use the `delete` function to remove a tool from the toolbar. The following statement removes the  toggle tool from the toolbar. The toggle tool handle is `tth`.

```
delete(tth)
```

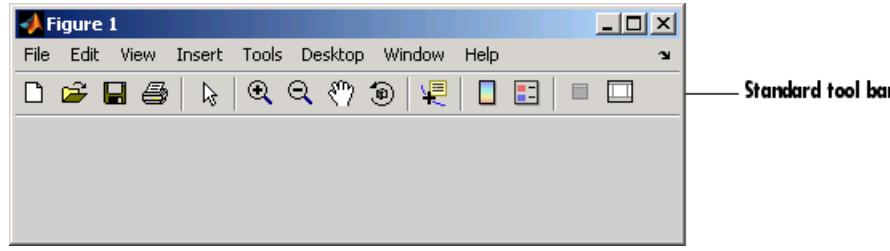
If necessary, you can use the `findall` function to determine the handles of the tools on a particular toolbar.

Note After you have created a toolbar and its tools, set their `HandleVisibility` properties `off` by executing statements similar to the following:

```
set(toolbarhandle,'HandleVisibility','off')
toolhandles = get(toolbarhandle,'Children');
set(toolhandles,'HandleVisibility','off')
```

Displaying and Modifying the Standard Toolbar

You can choose whether or not to display the MATLAB standard toolbar on your GUI. You can also add or delete tools from the standard toolbar.



Displaying the Standard Toolbar

Use the figure `Toolbar` property to display or hide the MATLAB standard toolbar. Set `Toolbar` to `figure` to display the standard toolbar. Set `Toolbar` to `none` to hide it.

```
set(fh,'Toolbar','figure');  % Display the standard toolbar
set(fh,'Toolbar','none');    % Hide the standard toolbar
```

In these statements, `fh` is the handle of the figure.

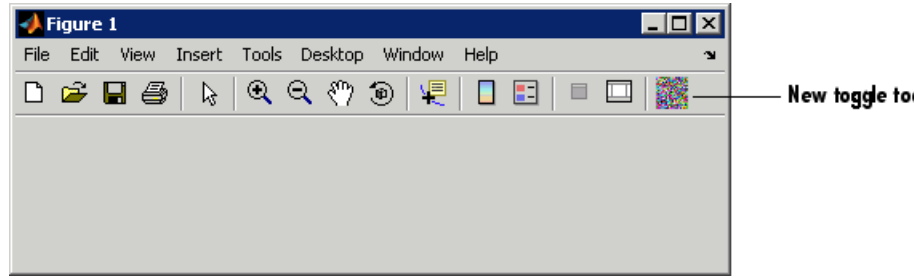
The default figure `Toolbar` setting is `auto`. This setting displays the figure toolbar, but removes it if you add a user interface control (uicontrol) to the figure.

Modifying the Standard Toolbar

Once you have the handle of the standard toolbar, you can add tools, delete tools, and change the order of the tools.

Add a tool the same way you would add it to a custom toolbar. The following code retrieves the handle of the MATLAB standard toolbar and adds to the toolbar a toggle tool similar to the one defined in “Toolbars” on page 11-57. `fh` is the handle of the figure.

```
tbh = findall(fh,'Type','uitoolbar');
tth = uitoggletool(tbh,'CData',rand(20,20,3),...
            'Separator','on',...
            'HandleVisibility','off');
```



To remove a tool from the standard toolbar, determine the handle of the tool to be removed, and then use the `delete` function to remove it. The following code deletes the toggle tool that was added to the standard toolbar above.

```
delete(tth)
```

If necessary, you can use the `findall` function to determine the handles of the tools on the standard toolbar.

Designing for Cross-Platform Compatibility

In this section...

Default System Font

By default, user interface controls (uicontrols) use the default font for the platform on which they are running. For example, when displaying your GUI on PCs, user interface controls use MS San Serif. When your GUI runs on a different platform, they use that computer's default font. This provides a consistent look with respect to your GUI and other application GUIs on the same platform.

If you have set the `FontName` property to a named font and want to return to the default value, you can set the property to the string `default`. This ensures that MATLAB uses the system default at run-time.

You can use the `set` command to set this property. For example, if there is a push button with handle `pbh1` in your GUI, then the statement

```
set(pbh1,'FontName','default')
```

sets the `FontName` property to use the system default.

Specifying a Fixed-Width Font

If you want to use a fixed-width font for a user interface control, set its `FontName` property to the string `fixedwidth`. This special identifier ensures that your GUI uses the standard fixed-width font for the target platform.

You can find the name of the fixed-width font that is used on a given platform by querying the root `FixedWidthFontName` property.

```
get(0,'FixedWidthFontName')
```

Using a Specific Font Name

You can specify an actual font name (such as Times or Courier) for the `FontName` property. However, doing so may cause your GUI to appear differently than you intended when run on a different computer. If the target computer does not have the specified font, it substitutes another font that may not look good in your GUI or may not be the standard font used for GUIs on that system. Also, different versions of the same named font may have different size requirements for a given set of characters.

Standard Background Color

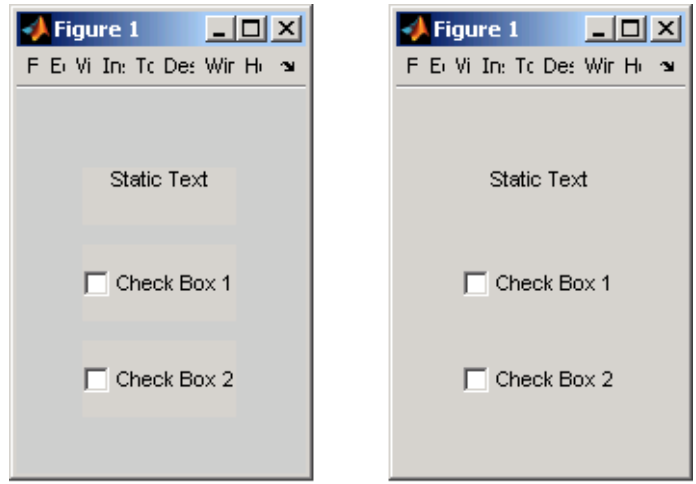
MATLAB uses the standard system background color of the system on which the GUI is running as the default component background color. This color varies on different computer systems, e.g., the standard shade of gray on the PC differs from that on UNIX, and may not match the default GUI background color.

You can make the GUI background color match the default component background color. The following statements retrieve the default component background color and assign it to the figure.

```
defaultBackground = get(0,'defaultUicontrolBackgroundColor');
set(figurehandle,'Color',defaultBackground)
```

The figure `Color` property specifies the figure's background color.

The following figures illustrate the results with and without system color matching.



Without system color matching

With system color matching

Cross-Platform Compatible Units

Cross-platform compatible GUIs should look correct on computers having different screen sizes and resolutions. Since the size of a pixel can vary on different computer displays, using the default figure `Units` of `pixels` does not produce a GUI that looks the same on all platforms. Setting the figure and components `Units` properties appropriately can help to determine how well the GUI transports to different platforms.

Units and Resize Behavior

The choice of units is also tied to the GUI's resize behavior. The figure `Resize` and `ResizeFcn` properties control the resize behavior of your GUI.

`Resize` determines if you can resize the figure window with the mouse. The `on` setting means you can resize the window, `off` means you cannot. When you set `Resize` to `off`, the figure window does not display any resizing controls to indicate that it cannot be resized.

`ResizeFcn` enables you to customize the GUI's resize behavior and is valid only if you set `Resize` to `on`. `ResizeFcn` is the handle of a user-written callback that is executed when a user resizes the GUI. It controls the resizing of all components in the GUI.

The following table shows appropriate `Units` settings based on the resize behavior of your GUI. These settings enable your GUI to automatically adjust the size and relative spacing of components as the GUI displays on different computers and when the GUI is resized.

Component	Default Units	Resize = on ResizeFcn = []	Resize = off
Figure	`pixels`	`characters`	`characters`
User interface controls (uicontrol) such as push buttons, sliders, and edit text components	`pixels`	`normalized`	characters
Axes	`normalized`	`normalized`	`characters`
Panel	`normalized`	`normalized`	`characters`
Button group	`normalized`	`normalized`	`characters`

Note The default settings shown in the table above are not the same as the GUIDE default settings. GUIDE default settings depend on the GUIDE **Resize behavior** option and are the same as those shown in the last two columns of the table.

About Some Units Settings

Characters. Character units are defined by characters from the default system font. The width of a character unit equals the width of the letter `x` in the system font. The height of a character unit is the distance between the baselines of two lines of text. Note that character units are not square.

Normalized. Normalized units represent a percentage of the size of the parent. The value of normalized units lies between 0 and 1. For example, if a panel contains a push button and the button units setting is `normalized`, then the push button `Position` setting [.2 .2 .6 .25] means that the left side of the push button is 20 percent of the panel width from the left side of the panel; the bottom of the button is 20 percent of the panel height from the bottom of the panel; the button itself is 60 percent of the width of the panel and 25 percent of its height.

Using Familiar Units of Measure. At times, it may be convenient to use a more familiar unit of measure, e.g., inches or centimeters, when you are laying out the GUI. However, to preserve the look of your GUI on different computers, remember to change the figure `Units` property back to `characters`, and the components' `Units` properties to `characters` (nonresizable GUIs) or `normalized` (resizable GUIs) before you save the M-file.

12

Programming the GUI

Introduction

After you have laid out your GUI, you need to program its behavior. This chapter addresses the programming of GUIs created programmatically. Specifically, it discusses data creation, GUI initialization, and the use of callbacks to control GUI behavior.

The following ordered list shows these topics within the organization of the typical GUI M-file.

1. Comments displayed in response to the MATLAB `help` command.
2. Initialization tasks such as data creation and any processing that is needed to construct the components. See "Initializing the GUI" on page 12-4 for information.
3. Construction of figure and components. See Chapter 11, "Laying Out a GUI" for information.
4. Initialization tasks that require the components to exist, and output return. See "Initializing the GUI" on page 12-4 for information.
5. Callbacks for the components. Callbacks are the routines that execute in response to user-generated events such as mouse clicks and key strokes. See "Callbacks: An Overview" on page 12-9 and "Examples: Programming GUI Components" on page 12-15 for information.
6. Utility functions.

Discussions in this chapter assume the use of nested functions. For information about using nested functions, see "Nested Functions" in the MATLAB Programming documentation.

See "Functions — By Category" in the MATLAB Function Reference documentation for a list of functions that are provided for GUI creation.

Note MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see “Predefined Dialog Boxes” in the MATLAB Function Reference documentation.

Initializing the GUI

Many kinds of tasks can be thought of as initialization tasks. This is a sampling of some of them:

- Define variables for supporting input and output arguments. See "Declaring Variables for Input and Output Arguments" on page 12-5.
- Define default values for input and output arguments.
- Define custom property values used for constructing the components. See "Defining Custom Property/Value Pairs" on page 12-5.
- Process command line input arguments.
- Create variables and data to be used by functions that are nested below the initialization section of the M-file. See "Nested Functions" in the MATLAB Programming documentation.
- Define variables for data to be shared between GUIs.
- Return user output if it is requested.
- Update or initialize components.
- Make changes needed to refine the look and feel of the GUI.
- Make changes needed for cross-platform compatibility. See "Designing for Cross-Platform Compatibility" on page 11-62.
- Make the GUI invisible while the components are being created and initialized. See "Making the Figure Invisible" on page 12-6.
- Make the GUI visible when you are ready for the user to see it.

Group these tasks together rather than scattering them throughout the code. If an initialization task is long or complex, consider creating a utility function to do the work.

Typically, some initialization tasks appear in the M-file before the components are constructed. Others appear after the components are constructed. Initialization tasks that require the components must appear following their construction.

Examples

These are some initialization examples taken from the examples discussed in Chapter 15, “Examples of GUIs Created Programmatically”. If MATLAB is running on your system, you can use these links to see the complete M-files:

- Color Palette
- Icon Editor

Declaring Variables for Input and Output Arguments

These are typical declarations for input and output arguments. They are taken from example “Icon Editor” on page 15-29.

```
mInputArgs = varargin; % Command line arguments when invoking
                       % the GUI
mOutputArgs = {};      % Variable for storing output when GUI
                       % returns
```

See the `varargin` reference page and the Icon Editor M-file for more information.

Defining Custom Property/Value Pairs

The example “Icon Editor” on page 15-29 defines property value pairs to be used as input arguments.

The example defines the properties in a cell array, `mPropertyDefs`, and then initializes the properties.

```
mPropertyDefs = {...
                'iconwidth',  @localValidateInput, 'mIconWidth';
                'iconheight', @localValidateInput, 'mIconHeight';
                'iconfile',   @localValidateInput, 'mIconFile'};
mIconWidth = 16;   % Use input property 'iconwidth' to initialize
mIconHeight = 16;  % Use input property 'iconheight' to initialize
mIconFile = fullfile(matlabroot,'toolbox/matlab/icons/');
                  % Use input property 'iconfile' to initialize
```

Each row of the cell array defines one property. It specifies, in order, the name of the property, the routine that is called to validate the input, and the name of the variable that holds the property value.

The `fullfile` function builds a full filename from parts.

The following statements each start the Icon Editor. The first one could be used to create a new icon. The second one could be used to edit an existing icon file.

```
cdata = iconEditor('iconwidth',16,'iconheight',25)
cdata = iconEditor('iconfile','eraser.gif');
```

`iconEditor` calls a routine, `processUserIputs`, during the initialization to

- Identify each property by matching it to the first column of the cell array
- Call the routine named in the second column to validate the input
- Assign the value to the variable named in the third column

See the complete Icon Editor M-file for more information.

Making the Figure Invisible

When you create the GUI figure, make it invisible so that you can display it for the user only when it is complete. Making it invisible during creation also enhances performance.

To make the GUI invisible, set the figure `Visible` property to `off`. This makes the entire figure window invisible. The statement that creates the figure might look like this:

```
hMainFigure = figure(...
              'Units','characters',...
              'MenuBar','none',...
              'Toolbar','none',...
              'Position',[71.8 34.7 106 36.15],...
              'Visible','off');
```

Just before returning to the caller, you can make the figure visible with a statement like the following:

```
set(hMainFigure,'Visible','on')
```

Most components have `Visible` properties. You can also use these properties to make individual components invisible.

Returning Output to the User

If your GUI function provides for an argument to the left of the equal sign, and the user specifies such an argument, then you want to return the expected output. The code that provides this output usually appears just before the GUI returns.

In the example shown here, taken from the Icon Editor example M-file,

1. A call to `uiwait` blocks execution until `uiresume` is called or the current figure is deleted.
2. While execution is blocked, the GUI user creates the desired icon.
3. When the user signals completion of the icon by clicking **OK**, the routine that services the **OK** push button calls `uiresume` and control returns to the statement following the call to `uiwait`.
4. The GUI then returns the completed icon to the user as output of the GUI.

```
% Make the GUI blocking.
uiwait(hMainFigure);

% Return the edited icon CData if it is requested.
mOutputArgs{1} = mIconCData;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

`mIconData` contains the icon that the user created or edited. `mOutputArgs` is a cell array defined to hold the output arguments. `nargout` indicates how many output arguments the user has supplied. `varargout` contains the optional

output arguments returned by the GUI. See the complete Icon Editor M-file for more information.

Callbacks: An Overview

In this section...

What Is a Callback?

The callback functions you provide control how the GUI responds to events such as button clicks, slider movement, menu item selection, or the creation and deletion of components. There is a set of callbacks for each component and for the GUI figure itself.

The callback routines usually appear in the M-file following the initialization code and the creation of the components. See "File Organization" on page 11-4 for more information.

A callback is a function that you write and associate with a specific component in the GUI or with the GUI figure itself. The callbacks control GUI or component behavior by performing some action in response to an event for its component. The event can be a mouse click on a push button, menu selection, key press, etc. This kind of programming is often called event-driven programming.

When an event occurs for a component, MATLAB invokes the component callback that is associated with that event. As an example, suppose a GUI has a push button that triggers the plotting of some data. When the user clicks the button, MATLAB calls the callback you associated with clicking that button, and then the callback, which you have programmed, gets the data and plots it.

A component can be any control device such as an axes, push button, list box, or slider. For purposes of programming, it can also be a menu, toolbar tool, or a container such as a panel or button group. See "Available Components" on page 11-10 for a list and descriptions of components.

Kinds of Callbacks

The GUI figure and each type of component has specific kinds of callbacks with which you can associate it. The callbacks that are available for each component are defined as properties of that component. For example, a push button has five callback properties: `ButtonDownFcn`, `Callback`, `CreateFcn`, `DeleteFcn`, and `KeyPressFcn`. A panel has four callback properties: `ButtonDownFcn`, `CreateFcn`, `DeleteFcn`, and `ResizeFcn`. You can, but are not required to, create a callback function for each of these properties. The GUI itself, which is a figure, also has certain kinds of callbacks with which it can be associated.

Each kind of callback has a triggering mechanism or event that causes it to be called. The following table lists the callback properties that are available, their triggering events, and the components to which they apply.

Callback Property	Triggering Event	Components
`ButtonDownFcn`	Executes when the user presses a mouse button while the pointer is on or within five pixels of a component or figure.	Axes, figure, button group, panel, user interface controls
`Callback`	Control action. Executes, for example, when a user clicks a push button or selects a menu item.	Context menu, menu user interface controls
`ClickedCallback`	Control action. Executes when the push tool or toggle tool is clicked. For the toggle tool, this is independent of its state.	Push tool, toggle tool
`CloseRequestFcn`	Executes when the figure closes.	Figure

Callback Property	Triggering Event	Components
`CreateFcn`	Initializes the component when it is created. It executes after the component or figure is created, but before it is displayed.	Axes, button group, context menu, figure, menu, panel, push tool, toggle tool, toolbar, user interface controls
`DeleteFcn`	Performs cleanup operations just before the component or figure is destroyed.	Axes, button group, context menu, figure, menu, panel, push tool, toggle tool, toolbar, user interface controls
`KeyPressFcn`	Executes when the user presses a keyboard key and the callback's component or figure has focus.	Figure, user interface controls
`KeyReleaseFcn`	Executes when the user releases a keyboard key and the figure has focus.	Figure
`OffCallback`	Control action. Executes when the state of a toggle tool is changed to off.	Toggle tool
`OnCallback`	Control action. Executes when the state of a toggle tool is changed to on.	Toggle tool
`ResizeFcn`	Executes when a user resizes a panel, button group, or figure whose figure `Resize` property is set to `On`.	Figure, button group, panel

Callback Property	Triggering Event	Components
`SelectionChangeFcn`	Executes when a user selects a different radio button or toggle button in a button group component.	Button group
`WindowButtonDownFcn`	Executes when you press a mouse button while the pointer is in the figure window.	Figure
`WindowButtonMotionFcn`	Executes when you move the pointer within the figure window.	Figure
`WindowButtonUpFcn`	Executes when you release a mouse button.	Figure
`WindowScrollWheelFcn`	Executes when the mouse wheel is scrolled while the figure has focus.	Figure

Note User interface controls include push buttons, sliders, radio buttons, check boxes, editable text boxes, static text boxes, list boxes, and toggle buttons. They are sometimes referred to as uicontrols.

Check the properties reference page for your component, e.g., Uicontrol Properties, to get specific information for a given callback property.

Associating Callbacks with Components

A GUI can have many components and each component's properties provide a way of specifying which callback should run in response to a particular event for that component. The callback that runs when the user clicks a **Yes** button is not the one that runs for the **No** button. Each menu item also performs a different function and needs its own callback.

You associate a callback with a specific component by setting the value of the appropriate component callback property to the callback. This is usually done in the component definition.

You can specify a component callback property value as any of the following:

- **String** that is a valid MATLAB expression or the name of an M-file.
- **Cell array of strings**. This example uses a cell array of strings to specify `pushbutton_callback` as the callback routine to be executed when a user clicks **Button 1**.

  ```
  pbh = uicontrol(fh,'Style','pushbutton','String','Button 1',...
                  'Position',[50 20 60 40],...
                  'Callback',{'pushbutton_callback',width,...});
  ```

  `Callback` is the name of the callback property. The first element of the cell array is the name of the callback routine, subsequent elements are input arguments to the callback.

  The corresponding function definition would look like this:

  ```
  function pushbutton_callback(width,...)
  ```

  See “Defining Callbacks as a Cell Array of Strings — Special Case” in the MATLAB Graphics documentation for more information.

- **Function handle or cell array** containing a function handle and additional arguments. This example uses a function handle to specify `pushbutton_callback` as the callback routine to be executed when a user clicks **Button 1**.

  ```
  pbh = uicontrol(fh,'Style','pushbutton','String','Button 1',...
                  'Position',[50 20 60 40],...
                  'Callback',{@pushbutton_callback,width,...});
  ```

  `Callback` is the name of the callback property. The first element of the cell array is the handle of the callback routine, subsequent elements are input arguments to the callback.

  Because the callback is specified as a handle, MATLAB automatically passes two additional arguments, the handle of the component for which the event was triggered and `eventdata`, as the first two arguments of the

callback. The second element of the cell array, `width` in the example above, becomes the third argument of the callback.

The corresponding function definition would contain these two additional arguments:

```
function pushbutton_callback(hObject,eventdata,width,...)
```

See “Introduction” in the MATLAB Graphics documentation for more information.

When an appropriate event occurs, it triggers execution of the MATLAB expression, the script or function contained in the M-file, the specified function, or the function associated with the function handle. The same is true for menus, toolbar tools, and for the figure itself.

See “Kinds of Callbacks” on page 12-10 for a list of the available callbacks for each component. See the component property pages for information about specific callback properties.

Examples: Programming GUI Components

In this section...

Programming User Interface Controls

The examples assume that callback properties are specified using function handles, enabling MATLAB to pass arguments `hObject`, which is the handle of the component for which the event was triggered, and `eventdata`. See "Associating Callbacks with Components" on page 12-12 for more information.

- "Check Box" on page 12-16
- "Edit Text" on page 12-16
- "List Box" on page 12-18
- "Pop-Up Menu" on page 12-19
- "Push Button" on page 12-20
- "Radio Button" on page 12-21
- "Slider" on page 12-21
- "Toggle Button" on page 12-22

Note See "Available Components" on page 11-10 for descriptions of these components. See "Adding User Interface Controls" on page 11-13 for information about adding these components to your GUI.

Check Box

You can determine the current state of a check box from within any of its callbacks by querying the state of its `Value` property, as illustrated in the following example:

```
function checkbox1_Callback(hObject,eventdata)
if (get(hObject,'Value') == get(hObject,'Max'))
   % Checkbox is checked-take approriate action
else
   % Checkbox is not checked-take approriate action
end
```

`hObject` is the handle of the component for which the event was triggered.

You can also change the state of a check box by programmatically by setting the check box `Value` property to the value of the `Max` or `Min` property. For example,

```
set(cbh,'Value','Max')
```

puts the check box with handle `cbh` in the checked state.

Edit Text

To obtain the string a user types in an edit box, use any of its callbacks to get the value of the `String` property. This example uses the `Callback` callback.

```
function edittext1_Callback(hObject,eventdata)
user_string = get(hObject,'String');

% Proceed with callback
```

If the edit text `Max` and `Min` properties are set such that `Max - Min > 1`, the user can enter multiple lines. For example, setting `Max` to `2`, with the default value of `0` for `Min`, enables users to enter multiple lines. If you originally specify `String` as a character string, multiline user input is returned as a 2-D character array with each row containing a line. If you originally specify `String` as a cell array, multiline user input is returned as a 2-D cell array of strings.

`hObject` is the handle of the component for which the event was triggered.

Retrieving Numeric Data from an Edit Text Component. MATLAB returns the value of the edit text `String` property as a character string. If you want users to enter numeric values, you must convert the characters to numbers. You can do this using the `str2double` command, which converts strings to doubles. If the user enters nonnumeric characters, `str2double` returns `NaN`.

You can use code similar to the following in an edit text callback. It gets the value of the `String` property and converts it to a double. It then checks whether the converted value is `NaN` (`isnan`), indicating the user entered a nonnumeric character and displays an error dialog box (`errordlg`).

```
function edittext1_Callback(hObject,eventdata)
user_entry = str2double(get(hObject,'string'));
if isnan(user_entry)
 errordlg('You must enter a numeric value','Bad Input','modal')
 return
end

% Proceed with callback...
```

Triggering Callback Execution. If the contents of the edit text component have been changed, clicking inside the GUI, but outside the edit text, causes the edit text callback to execute. The user can also press **Enter** for an edit text that allows only a single line of text, or **Ctrl+Enter** for an edit text that allows multiple lines.

Available Keyboard Accelerators. GUI users can use the following keyboard accelerators to modify the content of an edit text. These accelerators are not modifiable.

- **Ctrl+X** – Cut
- **Ctrl+C** – Copy
- **Ctrl+V** – Paste
- **Ctrl+H** – Delete last character
- **Ctrl+A** – Select all

List Box

When the list box `Callback` callback is triggered, the list box `Value` property contains the index of the selected item, where `1` corresponds to the first item in the list. The `String` property contains the list as a cell array of strings.

This example retrieves the selected string. Note that it is necessary to convert the value of the `String` property from a cell array to a string.

```
function listbox1_Callback(hObject,eventdata)
index_selected = get(hObject,'Value');
list = get(hObject,'String');
item_selected = list{index_selected}; % Convert from cell array
                                      % to string
```

`hObject` is the handle of the component for which the event was triggered.

You can also select a list item programmatically by setting the list box `Value` property to the index of the desired item. For example,

```
set(lbh,'Value',2)
```

selects the second item in the list box with handle `lbh`.

Triggering Callback Execution. MATLAB executes the list box `Callback` callback after the mouse button is released or after certain key press events:

- The arrow keys change the `Value` property, trigger callback execution, and set the figure `SelectionType` property to `normal`.
- The **Enter** key and space bar do not change the `Value` property, but trigger callback execution and set the figure `SelectionType` property to `open`.

If the user double-clicks, the callback executes after each click. MATLAB sets the figure `SelectionType` property to `normal` on the first click and to `open` on the second click. The callback can query the figure `SelectionType` property to determine if it was a single or double click.

List Box Examples. See the following examples for more information on using list boxes:

- "List Box Directory Reader" on page 10-9 — Shows how to creates a GUI that displays the contents of directories in a list box and enables users to open a variety of file types by double-clicking the filename.
- "Accessing Workspace Variables from a List Box" on page 10-16 — Shows how to access variables in the MATLAB base workspace from a list box GUI.

Pop-Up Menu

When the pop-up menu `Callback` callback is triggered, the pop-up menu `Value` property contains the index of the selected item, where `1` corresponds to the first item on the menu. The `String` property contains the menu items as a cell array of strings.

Note A pop-up menu is sometimes referred to as a drop-down menu or combo box.

Using Only the Index of the Selected Menu Item. This example retrieves only the index of the item selected. It uses a switch statement to take action based on the value. If the contents of the pop-up menu are fixed, then you can use this approach. Else, you can use the index to retrieve the actual string for the selected item.

```
function popupmenu1_Callback(hObject,eventdata)
val = get(hObject,'Value');
switch val
case 1   % User selected the first item
case 2   % User selected the second item

% Proceed with callback...
```

`hObject` is the handle of the component for which the event was triggered.

You can also select a menu item programmatically by setting the pop-up menu `Value` property to the index of the desired item. For example,

```
set(pmh,'Value',2)
```

selects the second item in the pop-up menu with handle `pmh`.

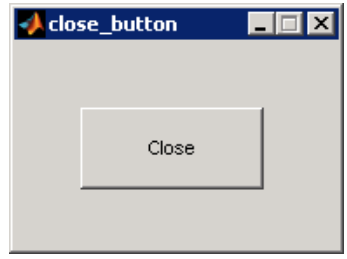
Using the Index to Determine the Selected String. This example retrieves the actual string selected in the pop-up menu. It uses the pop-up menu `Value` property to index into the list of strings. This approach may be useful if your program dynamically loads the contents of the pop-up menu based on user action and you need to obtain the selected string. Note that it is necessary to convert the value returned by the `String` property from a cell array to a string.

```
function popupmenu1_Callback(hObject,eventdata)
val = get(hObject,'Value');
string_list = get(hObject,'String');
selected_string = string_list{val}; % Convert from cell array
                                    % to string
% Proceed with callback...
```

`hObject` is the handle of the component for which the event was triggered.

Push Button

This example contains only a push button. Clicking the button, closes the GUI.



This is the push button's `Callback` callback. It displays the string `Goodbye` at the command line and then closes the GUI.

```
function pushbutton1_Callback(hObject,eventdata)
display Goodbye
close(gcbf)
```

`gcbf` returns the handle of the figure containing the object whose callback is executing.

Radio Button

You can determine the current state of a radio button from within its `Callback` callback by querying the state of its `Value` property, as illustrated in the following example:

```
function radiobutton_Callback(hObject,eventdata)
if (get(hObject,'Value') == get(hObject,'Max'))
 % Radio button is selected-take approriate action
else
 % Radio button is not selected-take approriate action
end
```

Radio buttons set `Value` to `Max` when they are on (when selected) and `Min` when off (not selected). `hObject` is the handle of the component for which the event was triggered.

You can also change the state of a radio button programmatically by setting the radio button `Value` property to the value of the `Max` or `Min` property. For example,

```
set(rbh,'Value','Max')
```

puts the radio button with handle `rbh` in the selected state.

Note You can use a button group to manage exclusive selection behavior for radio buttons. See “Button Group” on page 12-23 for more information.

Slider

You can determine the current value of a slider from within its `Callback` callback by querying its `Value` property, as illustrated in the following example:

```
function slider1_Callback(hObject,eventdata)
slider_value = get(hObject,'Value');

% Proceed with callback...
```

The Max and Min properties specify the slider's maximum and minimum values. The slider's range is Max - Min. hObject is the handle of the component for which the event was triggered.

Toggle Button

The callback for a toggle button needs to query the toggle button to determine what state it is in. MATLAB sets the Value property equal to the Max property when the toggle button is pressed (Max is 1 by default). It sets the Value property equal to the Min property when the toggle button is not pressed (Min is 0 by default).

The following code illustrates how to program the callback in the GUI M-file.

```
function togglebutton1_Callback(hObject,eventdata)
button_state = get(hObject,'Value');
if button_state == get(hObject,'Max')
 % Toggle button is pressed-take approperiate action
   ...
elseif button_state == get(hObject,'Min')
 % Toggle button is not pressed-take appropriate action
    ...
end
```

hObject is the handle of the component for which the event was triggered.

You can also change the state of a toggle button programmatically by setting the toggle button Value property to the value of the Max or Min property. For example,

```
set(tbh,'Value','Max')
```

puts the toggle button with handle tbh in the pressed state.

Note You can use a button group to manage exclusive selection behavior for toggle buttons. See "Button Group" on page 12-23 for more information.

Programming Panels and Button Groups

These topics provide basic code examples for panels and button group callbacks.

The examples assume that callback properties are specified using function handles, enabling MATLAB to pass arguments `hObject`, which is the handle of the component for which the event was triggered, and `eventdata`. See "Associating Callbacks with Components" on page 12-12 for more information.

Panel

Panels group GUI components and can make a GUI easier to understand by visually grouping related controls. A panel can contain panels and button groups, as well as axes and user interface controls such as push buttons, sliders, pop-up menus, etc. The position of each component within a panel is interpreted relative to the lower-left corner of the panel.

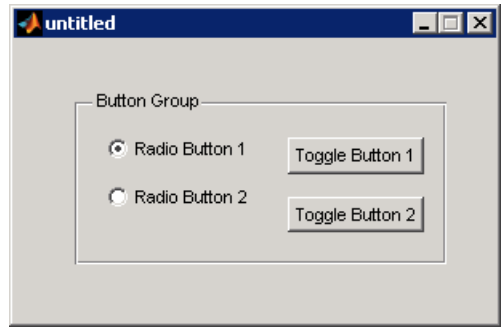
Generally, if the GUI is resized, the panel and its components are also resized. However, you can control the size and position of the panel and its components. You can do this by setting the GUI `Resize` property to `on` and providing a `ResizeFcn` callback for the panel.

Note See "Cross-Platform Compatible Units" on page 11-64 for information about the effect of units on resize behavior.

Button Group

Button groups are like panels except that they manage exclusive selection behavior for radio buttons and toggle buttons. If a button group contains a set of radio buttons, toggle buttons, or both, the button group allows only one of them to be selected. When a user clicks a button, that button is selected and all other buttons are deselected.

The following figure shows a button group with two radio buttons and two toggle buttons. **Radio Button 1** is selected.



If a user clicks the other radio button or one of the toggle buttons, it becomes selected and **Radio Button 1** is deselected. The following figure shows the result of clicking **Toggle Button 2**.

The button group `SelectionChangeFcn` callback is called whenever a selection is made. If you have a button group that contains a set of radio buttons and toggle buttons and you want:

- An immediate action to occur when a radio button or toggle button is selected, you must include the code to control the radio and toggle buttons in the button group's `SelectionChangeFcn` callback function, not in the individual toggle button `Callback` functions. "Color Palette" on page 15-17 provides a practical example of a `SelectionChangeFcn` callback.

- Another component such as a push button to base its action on the selection, then that component's `Callback` callback can get the handle of the selected radio button or toggle button from the button group's `SelectedObject` property.

This example of a `SelectionChangeFcn` callback uses the `Tag` property of the selected object to choose the appropriate code to execute. The `Tag` property of each component is a string that identifies that component and must be unique in the GUI.

```
function uibuttongroup1_SelectionChangeFcn(hObject,eventdata)
switch get(eventdata.NewValue,'Tag') % Get Tag of selected object.
    case 'radiobutton1'
        % Code for when radiobutton1 is selected.
    case 'radiobutton2'
        % Code for when radiobutton2 is selected.
    case 'togglebutton1'
        % Code for when togglebutton1 is selected.
    case 'togglebutton2'
        % Code for when togglebutton2 is selected.
    % Continue with more cases as necessary.
    otherwise
        % Code for when there is no match.
end
```

The `hObject` and `eventdata` arguments are available to the callback only if the value of the callback property is specified as a function handle. See the`SelectionChangeFcn` property on the Uibuttongroup Properties reference page for information about `eventdata`. See the `uibuttongroup` reference page and "Color Palette" on page 15-17 for other examples.

Programming Axes

Axes components enable your GUI to display graphics, such as graphs and images. This topic briefly tells you how to plot to an axes in your GUI.

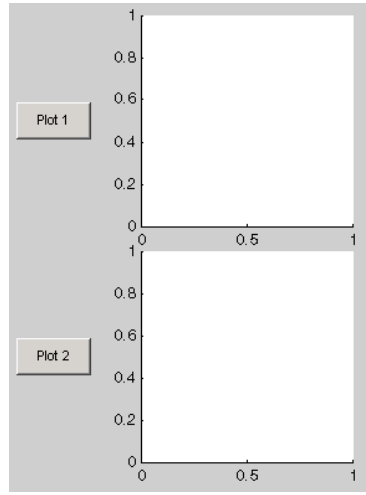
In most cases, you create a plot in an axes from a callback that belongs to some other component in the GUI. For example, pressing a button might trigger the plotting of a graph to an axes. In this case, the button's `Callback` callback contains the code that generates the plot.

The following example contains two axes and two push buttons. Clicking the first button generates a `contour` plot in one axes and clicking the other button generates a `surf` plot in the other axes. The example generates data for the plots using the `peaks` function, which returns a square matrix obtained by translating and scaling Gaussian distributions.

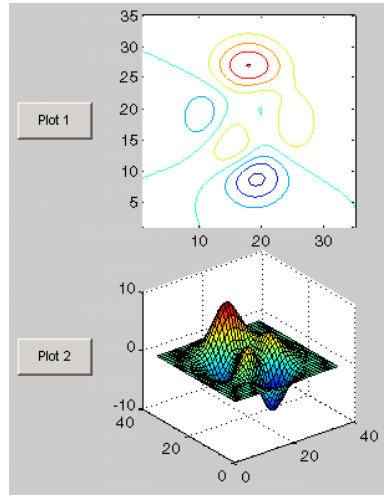
1 Save this code in an M-file named `two_axes.m`.

```
function two_axes
fh = figure;
bh1 = uicontrol(fh,'Position',[20 290 60 30],...
                'String','Plot 1',...
                'Callback',@button1_plot);
bh2 = uicontrol(fh,'Position',[20 100 60 30],...
                'String','Plot 2',...
                'Callback',@button2_plot);
ah1 = axes('Parent',fh,'units','pixels',...
           'Position',[120 220 170 170]);
ah2 = axes('Parent',fh,'units','pixels',...
           'Position',[120 30 170 170]);
   %-----------------------------------------------
   function button1_plot(hObject,eventdata)
      contour(ah1,peaks(35));
   end
   %-----------------------------------------------
   function button2_plot(hObject,eventdata)
      surf(ah2,peaks(35));
   end
end
```

2 Run the GUI by typing `two_axes` at the command line. This is what the example looks like before you click the push buttons.



3 Click the **Plot 1** button to display the contour plot in the first axes. Click the **Plot 2** button to display the surf plot in the second axes.



See “GUI with Multiple Axes” on page 10-2 for a more complex example that uses two axes.

If your GUI contains axes, you should ensure that their `HandleVisibility` properties are set to `callback`. This allows callbacks to change the contents of the axes and prevents command line operations from doing so. The default is `on`.

For more information about:

- Properties that you can set to control many aspects of axes behavior and appearance, see “Axes Properties” in the MATLAB Graphics documentation.
- Creating axes in a tiled pattern, see the `subplot` function reference page.
- Plotting in general, see “Plots and Plotting Tools” in the MATLAB Graphics documentation.

Programming ActiveX Controls

For information about programming ActiveX controls, see the following topics in the MATLAB External Interfaces documentation.

- “Control and Server Events”
- “Writing Event Handlers”

See “MATLAB COM Client Support” in the MATLAB External Interfaces documentation for general information.

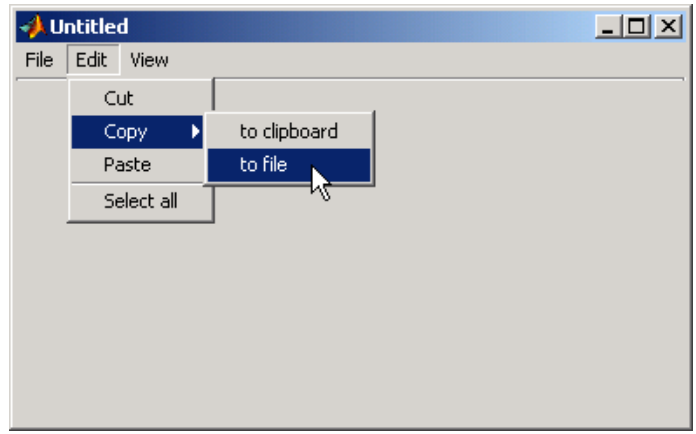
Programming Menu Items

Programming a Menu Title

Because clicking a menu title automatically displays the menu below it, you may not need to program callbacks at the title level. However, the callback

associated with a menu title can be a good place to enable or disable menu items below it.

Consider the example illustrated in the following picture.



When a user selects **Edit > Copy > to file**, no **Copy** callback is needed to perform the action. Only the `Callback` callback associated with the **to file** item is required.

Suppose, however, that only certain objects can be copied to a file. You can use the **Copy** item `Callback` callback to enable or disable the **to file** item, depending on the type of object selected.

The following code disables the **to file** item by setting its `Enable` property `off`. The menu item would then appear dimmed.

```
set(tofilehandle,'Enable','off')
```

Setting `Enable` to `on`, would then enable the menu item.

Opening a Dialog Box from a Menu Callback

The `Callback` callback for the **to file** menu item could contain code such as the following to display the standard dialog box for saving files.

```
[file,path] = uiputfile('animinit.m','Save file name');
```

'Save file name' is the dialog box title. In the dialog box, the filename field is set to animinit.m, and the filter set to M-files (*.m). For more information, see the uiputfile reference page.

Note MATLAB provides a selection of standard dialog boxes that you can create with a single function call. For information about these dialog boxes and the functions used to create them, see "Predefined Dialog Boxes" in the MATLAB Function Reference documentation.

Updating a Menu Item Check

A check is useful to indicate the current state of some menu items. If you set the Checked property to on when you create the menu item, the item initially appears checked. Each time the user selects the menu item, the callback for that item must turn the check on or off. The following example shows you how to do this by changing the value of the menu item's Checked property.

```
function menu_copyfile(hObject,eventdata)
if strcmp(get(hObject,'Checked'),'on')
    set(hObject,'Checked','off');
else
    set(hObject,'Checked','on');
end
```

hObject is the handle of the component for which the event was triggered. Its use here assumes the menu item's Callback property specifies the callback as a function handle. See "Associating Callbacks with Components" on page 12-12 for more information.

The strcmp function compares two strings and returns logical 1 (true) if the two are identical, and logical 0 (false) otherwise.

Use of checks when the GUI is first displayed should be consistent with the display. For example, if your GUI has an axes that is visible when a user first opens it and the GUI has a **Show axes** menu item, be sure to set the menu item's Checked property on when you create it so that a check appears next to the **Show axes** menu item initially.

Programming Toolbar Tools

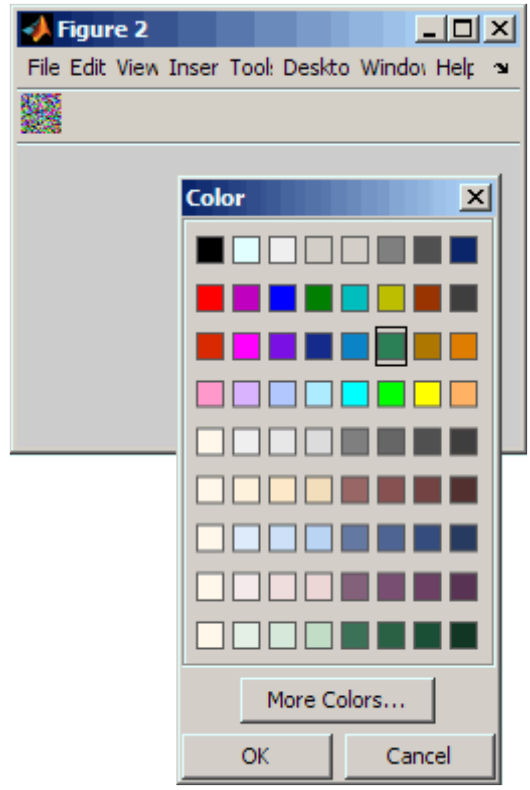
Push Tool

The push tool `ClickedCallback` property specifies the push tool control action. The following example creates a push tool and programs it to open a standard color selection dialog box. You can use the dialog box to set the background color of the GUI.

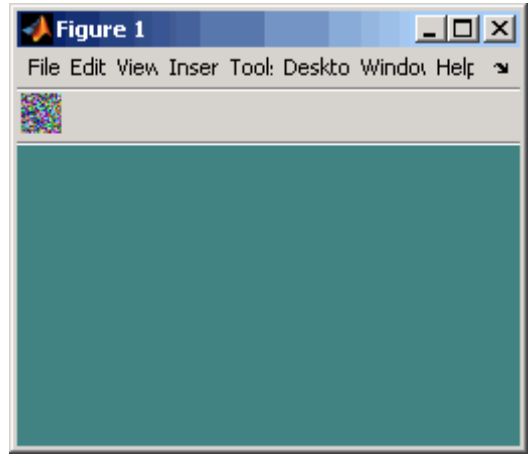
1 Copy the following code into an M-file and save it in your current directory or on your path as `color_gui.m`. Run the script by typing `color_gui` at the command line.

```
function color_gui
fh = figure('Position',[250 250 250 150],'Toolbar','none');
th = uitoolbar('Parent',fh);
pth = uipushtool('Parent',th,'Cdata',rand(20,20,3),...
                 'ClickedCallback',@color_callback);
%-------------------------------------------------------
    function color_callback(hObject,eventdata)
    color = uisetcolor(fh,'Pick a color');
    end
end
```

2 Click the push tool to display the color selection dialog box and click a color to select it.



3 Click **OK** on the color selection dialog box. The GUI background color changes to the color you selected—in this case, green.



Note Create your own icon with the icon editor described in "Icon Editor" on page 15-29. See the `ind2rgb` reference page for information on converting a matrix `X` and corresponding colormap, i.e., an `(X, MAP)` image, to RGB (truecolor) format.

Toggle Tool

The toggle tool `OnCallback` and `OffCallback` properties specify the toggle tool control actions that occur when the toggle tool is clicked and its `State` property changes to `on` or `off`. The toggle tool `ClickedCallback` property specifies a control action that takes place whenever the toggle tool is clicked, regardless of state.

The following example uses a toggle tool to toggle a plot between surface and mesh views of the `peaks` data. The example also counts the number of times you have clicked the toggle tool.

The `surf` function produces a 3-D shaded surface plot. The `mesh` function creates a wireframe parametric surface. `peaks` returns a square matrix obtained by translating and scaling Gaussian distributions

1 Copy the following code into an M-file and save it in your current directory or on your path as `toggle_plots.m`. Run the script by typing `toggle_plots` at the command line.

```
function toggle_plots
counter = 0;
fh = figure('Position',[250 250 300 340],'Toolbar','none');
ah = axes('Parent',fh,'Units','pixels',...
          'Position',[35 85 230 230]);
th = uitoolbar('Parent',fh);
tth = uitoggletool('Parent',th,'Cdata',rand(20,20,3),...
                 'OnCallback',@surf_callback,...
                 'OffCallback',@mesh_callback,...
                 'ClickedCallback',@counter_callback);
sth = uicontrol('Style','text','String','Counter: ',...
                'Position',[35 20 45 20]);
cth = uicontrol('Style','text','String',num2str(counter),...
                'Position',[85 20 30 20]);
%-------------------------------------------------------
    function counter_callback(hObject,eventdata)
    counter = counter + 1;
    set(cth,'String',num2str(counter))
    end
%-------------------------------------------------------
    function surf_callback(hObject,eventdata)
    surf(ah,peaks(35));
    end
%-------------------------------------------------------
    function mesh_callback(hObject,eventdata)
    mesh(ah,peaks(35));
    end
end
```

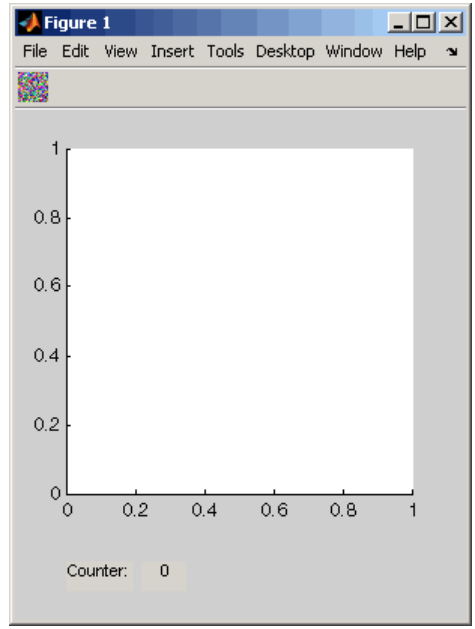
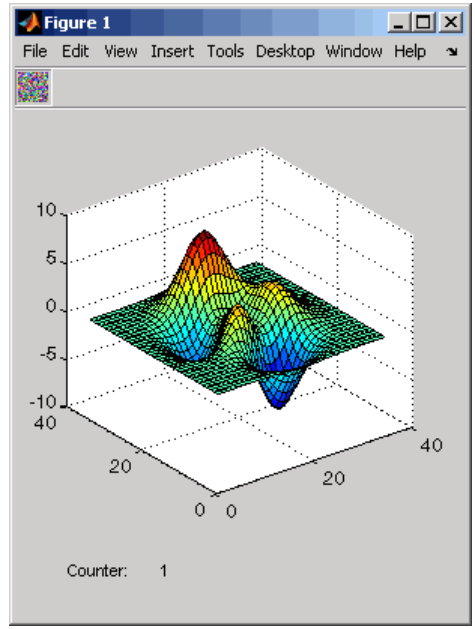


Figure 1
File Edit View Insert Tools Desktop Window Help
1
0.8
0.6
0.4
0.2
0
0 0.2 0.4 0.6 0.8 1
Counter: 0

2 Click the toggle tool to display the initial plot. The counter increments to 1.



3 Continue clicking the toggle tool to toggle between surf and mesh plots of the `peaks` data.

13

Managing Application-Defined Data

Mechanisms for Managing Data

In this section...

Nested Functions

Use nested function to create your GUI M-files. They enable callback functions to share data freely without it having to be passed as arguments.

1. Construct components, define variables, and generate data in the initialization segment of your code.
2. Nest the GUI callbacks and utility functions at a level below the initialization.

The callbacks and utility functions automatically have access to the data and the component handles because they are defined at a higher level.

Note For information about using nested functions, see "Nested Functions" in the MATLAB Programming documentation.

GUI Data

Most GUIs generate or use data that is specific to the application. These mechanisms provide a way for applications to save and retrieve data stored with the GUI.

The GUI data and application data mechanisms are similar, but GUI data can be simpler to use. The figure and component `UserData` properties can also hold application-defined data.

GUI data is managed using the `guidata` function. This function can store a single variable as GUI data. It is also used to retrieve the value of that variable.

Note If your M-file was originally created by GUIDE, see "Changing GUI Data in an M-File Generated by GUIDE" on page 9-4.

About GUI Data

GUI data is always associated with the GUI figure. It is available to all callbacks of all components of the GUI. If you specify a component handle when you save or retrieve GUI data, MATLAB automatically associates the data with the component's parent figure.

GUI data can contain only one variable at any time. Writing GUI data with a different variable overwrites the existing GUI data. For this reason, GUI data is usually defined to be a structure to which you can add fields as you need them.

You can access the data from within a callback routine using the component's handle, without having to find the figure handle. If you specify a component's callback properties as function handles, the component handle is automatically passed to each callback as `hObject`. See "Associating Callbacks with Components" on page 12-12 for more information.

Because there can be only one GUI data variable and it is associated with the figure, you do not need to create and maintain a hard-coded name for the data throughout your source code.

Creating and Updating GUI Data

1 Create a structure and add to it the fields you want. For example,

```
mydata.iteration_counter = 0;
mydata.number_errors = 0;
```

2 Save the structure as GUI data. MATLAB associates GUI data with the figure, but you can use the handle of any component in the figure to retrieve or save it.

```
guidata(figurehandle,mydata);
```

3 To change GUI data from a callback, get a copy of the structure, update the desired field, and then save the GUI data.

```
mydata = guidata(hObject);      % Get the GUI data.
mydata.iteration_counter = mydata.iteration_counter +1;
guidata(hObject,mydata);        % Save the GUI data.
```

Note To use `hObject`, you must specify a component's callback properties as function handles. When you do, the component handle is automatically passed to each callback as `hObject`. See "Associating Callbacks with Components" on page 12-12 for more information.

Adding Fields to a GUI Data Structure

To add a field to a GUI data structure:

1 Get a copy of the structure with a command similar to the following where `hObject` is the handle of the component for which the callback was triggered.

```
mydata = guidata(hObject)
```

2 Assign a value to the new field. This adds the field to the structure. For example,

```
mydata.iteration_state = 0;
```

adds the field `iteration_state` to the structure `mydata` and sets it to `0`.

3 Use the following command to save the data.

```
guidata(hObject,mydata)
```

where `hObject` is the handle of the component for which the callback was triggered. MATLAB associates a new copy of the `mydata` structure with the component's parent figure.

Application Data

Application data provides a way for applications to save and retrieve data associated with a specified object. For a GUI, this is usually the GUI figure but can also be any component. The data is stored as name/value pairs. Application data enables you to create what are essentially user-defined properties for an object.

The following table summarizes the functions that provide access to application data. For more detailed information, see the individual function reference pages.

Functions for Managing Application Data

Function	Purpose
`setappdata`	Specify named application data for an object. The object does not have to be a figure. You can specify more than one named application data for an object. However, each name must be unique for that object and can be associated with only one value, usually a structure.
`getappdata`	Retrieve named application data. To retrieve named application data, you must know the name associated with the application data and the handle of the object with which it is associated.

Functions for Managing Application Data (Continued)

Function	Purpose
`isappdata`	True if the named application data exists on the specified object.
`rmappdata`	Remove named application data from the specified object.

Creating Application Data

Use the `setappdata` function to create application data. This example generates a 35-by-35 matrix of normally distributed random numbers and creates application data `mydata`, associated with the figure, to manage it.

```
matrices.rand_35 = randn(35);
setappdata(figurehandle,'mydata',matrices);
```

By using nested functions and creating the figure at the top level, the figure handle is accessible to all callbacks and utility functions nested at lower levels. For information about using nested functions, see “Nested Functions” in the MATLAB Programming documentation.

Adding Fields to an Application Data Structure

Application data is usually defined as a structure to enable you to add fields as necessary. This example adds a field to the application data structure `mydata` created in the previous topic.

1. Use `getappdata` to retrieve the structure.

   From the example in the previous topic, the name of the application data structure is `mydata`. It is associated with the figure.

   ```
   matrices = getappdata(figurehandle,'mydata');
   ```

2 Create a new field and assign it a value. For example

```
matrices.randn_50 = randn(50);
```

adds the field `randn_50` to the `matrices` structure and sets it to a 50-by-50 matrix of normally distributed random numbers.

3 Use `setappdata` to save the data. This example uses `setappdata` to save the `matrices` structure as the application data structure `mydata`.

```
setappdata(figurehandle,'mydata',matrices);
```

UserData Property

Each GUI component and the figure itself has a `UserData` property. You can assign any valid MATLAB value to a `UserData` property. To retrieve the data, a callback must know the handle of the component with which the data is associated.

1 In this example, an edit text component stores the user-entered string in its `UserData` property.

```
function edittext1_callback(hObject,eventdata)
mystring = get(hObject,'String');
set(hObject,'UserData',mystring);
```

2 A push button retrieves the string from the edit text component `UserData` property.

```
function pushbutton1_callback(hObject,eventdata)
string = get(edittexthandle,'UserData');
```

Specify `UserData` as a structure if you want to store multiple fields.

Note By using nested functions and creating the figure and the components at the top level, their handles are accessible to all callbacks and utility functions nested at lower levels. For information about using nested functions, see "Nested Functions" in the MATLAB Programming documentation. To use `hObject`, you must specify a component's callback properties as function handles. When you do, the component handle is automatically passed to each callback as `hObject`. See "Associating Callbacks with Components" on page 12-12 for more information.

Sharing Data Among a GUI's Callbacks

In this section...

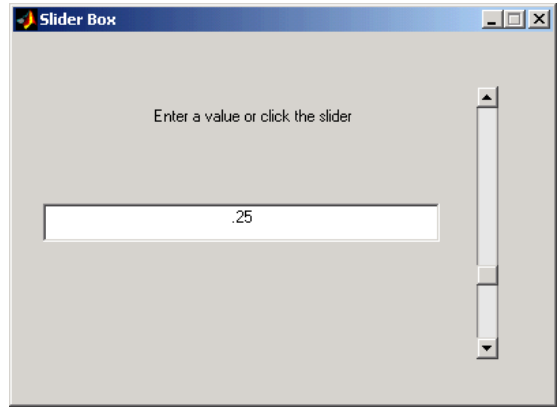
See "Mechanisms for Managing Data" on page 13-2 for general information about these methods.

Nested Functions

You can use GUI data, application data, and the `UserData` property to share data among a GUI's callbacks. In many cases nested functions enables you to share data among callbacks without using the other data forms.

Nested Functions Example: Passing Data Between Components

This example uses a GUI that contains a slider and an edit text component as shown in the following figure. A static text component instructs the user to enter a value in the edit text or click the slider. The example initializes and maintains an error counter as well as the old and new values of the slider in a nested functions environment.

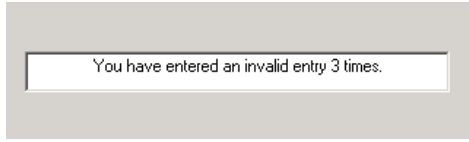


The GUI behavior is as follows:

- When a user moves the slider, the edit text component displays the slider's current value and prints a message to the command line, similar to the following, indicating how many units the slider moved.

  ```
  You moved the slider 25 units.
  ```

- When a user types a value into the edit text component and then presses **Enter** or clicks outside the component, the slider updates to this value an d the edit text component prints a message to the command line indicating how many units the slider moved.
- If a user enters a value in the edit text component that is out of range for the slider—that is, a value that is not between the slider's `Min` and `Max` properties—the application returns a message in the edit text indicating how many times the user has entered an erroneous value.



The following code constructs the components, initializes the error counter and the previous and new slider values in the initialization section of the function, and uses two callbacks to implement the interchange between the slider and the edit text component. Copy this code into an M-file and save it in your current directory or on your path as `slider_gui.m`. Run the script by typing `slider_gui` at the command line.

```
function slider_gui
fh = figure('Position',[250 250 350 350]);
sh = uicontrol(fh,'Style','slider',...
               'Max',100,'Min',0,'Value',25,...
               'SliderStep',[0.05 0.2],...
               'Position',[300 25 20 300],...
               'Callback',@slider_callback);
eth = uicontrol(fh,'Style','edit',...
               'String',num2str(get(sh,'Value')),...
               'Position',[30 175 240 20],...
```

```
               'Callback',@edittext_callback);
sth = uicontrol(fh,'Style','text',...
               'String','Enter a value or click the slider.',...
               'Position',[30 215 240 20]);
number_errors = 0;
previous_val = 0;
val = 0;
% --------------------------------------------------
% Set the value of the edit text component String property
% to the value of the slider.
   function slider_callback(hObject,eventdata)
      previous_val = val;
      val = get(hObject,'Value');
      set(eth,'String',num2str(val));
      sprintf('You moved the slider %d units.',abs(val - previous_val))
   end
% --------------------------------------------------
% Set the slider value to the number the user types in
% the edit text or display an error message.
   function edittext_callback(hObject,eventdata)
      previous_val = val;
      val = str2double(get(hObject,'String'));
      % Determine whether val is a number between the
      % slider's Min and Max. If it is, set the slider Value.
      if isnumeric(val) && length(val) == 1 && ...
         val >= get(sh,'Min') && ...
         val <= get(sh,'Max')
         set(sh,'Value',val);
         sprintf('You moved the slider %d units.',abs(val - previous_val)
      else
      % Increment the error count, and display it.
         number_errors = number_errors+1;
         set(hObject,'String',...
             ['You have entered an invalid entry ',...
             num2str(number_errors),' times.']);
         val = previous_val;
      end
   end
end
```

Because the components are constructed at the top level, their handles are immediately available to the callbacks that are nested at a lower level of the routine. The same is true of the error counter `number_errors`, the previous slider value `previous_val`, and the new slider value `val`. There is no need to pass these variables as arguments.

Both callbacks use the input argument `hObject` to get and set properties of the component that triggered execution of the callback. This argument is available to the callbacks because the components' `Callback` properties are specified as function handles. See "Associating Callbacks with Components" on page 12-12 for more information.

Slider Callback. The slider callback, `slider_callback`, uses the edit text component handle, `eth`, to set the edit text `'String'` property to the value the user typed.

The slider `Callback` saves the previous value, `val`, of the slider in `previous_val` before assigning the new value to `val`. These variables are known to both callbacks because they are initialized at a higher level. They can be retrieved and set by either callback.

```
previous_val = val;
val = get(hObject,'Value');
```

The following statements in the slider `Callback` update the value displayed in the edit text component when a user moves the slider and releases the mouse button.

```
val = get(hObject,'Value');
set(eth,'String',num2str(val));
```

The code combines three commands:

- The `get` command obtains the current value of the slider.
- The `num2str` command converts the value to a string.
- The `set` command sets the `String` property of the edit text component to the updated value.

Edit Text Callback. The edit text `Callback`, `edittext_callback`, uses the slider handle, `sh`, to determine the slider's `Max` and `Min` properties and to set the slider `Value` property, which determine's the position of the slider thumb.

The edit text `Callback` uses the following code to set the slider's value to the number the user types in, after checking to see if it is a single numeric value within the allowed range.

```
if isnumeric(val) && length(val) == 1 && ...
   val >= get(sh,'Min') && ...
   val <= get(sh,'Max')
   set(sh,'Value',val);
```

If the value is out of range, the `if` statement continues by incrementing the error counter, `number_errors`, and displaying a message telling the user how many times they have entered an invalid number.

```
else
   number_errors = number_errors+1;
   set(hObject,'String',...
   ['You have entered an invalid entry ',...
   num2str(number_errors),' times.']);
end
```

GUI Data

GUI data, which you manage with the `guidata` function, is accessible to all callbacks of the GUI. A callback for one component can set a value in GUI data, which can then be read by a callback for another component. See "GUI Data" on page 13-2 for more information.

GUI Data Example: Passing Data Between Components

The previous topic, "Nested Functions Example: Passing Data Between Components" on page 13-9, uses nested function capabilities to initialize and maintain an error counter as well as the old and new values of the slider. This example shows you how to initialize and maintain the old and new values of the slider using GUI data and make them available to the both callbacks. Refer to the previous topic for details of the example.

The following code is similar to the previous topic but uses GUI data to initialize and maintain the old and new slider values in the edit text and slider `Callbacks`. Copy this code into an M-file and save it in your current directory or on your path as `slider_gui.m`. Run the script by typing `slider_gui` at the command line.

```
function slider_gui
fh = figure('Position',[250 250 350 350]);
sh = uicontrol(fh,'Style','slider',...
               'Max',100,'Min',0,'Value',25,...
               'SliderStep',[0.05 0.2],...
               'Position',[300 25 20 300],...
               'Callback',@slider_callback);
eth = uicontrol(fh,'Style','edit',...
               'String',num2str(get(sh,'Value')),...
               'Position',[30 175 240 20],...
               'Callback',@edittext_callback);
sth = uicontrol(fh,'Style','text',...
               'String','Enter a value or click the slider.',...
               'Position',[30 215 240 20]);
number_errors = 0;
slider.val = 25;
guidata(fh,slider);
% --------------------------------------------------
% Set the value of the edit text component String property
% to the value of the slider.
   function slider_callback(hObject,eventdata)
        slider = guidata(fh);  % Get GUI data.
        slider.previous_val = slider.val;
        slider.val = get(hObject,'Value');
        set(eth,'String',num2str(slider.val));
        sprintf('You moved the slider %d units.',...
                 abs(slider.val - slider.previous_val))
        guidata(fh,slider) % Save GUI data before returning.
   end
% --------------------------------------------------
% Set the slider value to the number the user types in
% the edit text or display an error message.
   function edittext_callback(hObject,eventdata)
        slider = guidata(fh);   % Get GUI data.
```

```
            slider.previous_val = slider.val;
            slider.val = str2double(get(hObject,'String'));
         % Determine whether slider.val is a number between the
         % slider's Min and Max. If it is, set the slider Value.
         if isnumeric(slider.val) && length(slider.val) == 1 && ...
            slider.val >= get(sh,'Min') && ...
            slider.val <= get(sh,'Max')
            set(sh,'Value',slider.val);
            sprintf('You moved the slider %d units.',...
                    abs(slider.val - slider.previous_val))
         else
         % Increment the error count, and display it.
            number_errors = number_errors+1;
            set(hObject,'String',...
                ['You have entered an invalid entry ',...
                num2str(number_errors),' times.']);
            slider.val = slider.previous_val;
         end
         guidata(fh,slider); % Save the changes as GUI data.
      end
   end
```

Slider Values. In this example, both the slider callback`slider_callback` and the edit text callback `edittext_callback` retrieve the GUI data structure `slider` which hold previous and current values of the slider. They then save the value, `slider.val` to `slider.previous_val` before retrieving the new value and assigning it to `slider.val`. Before returning, each callback saves the `slider` structure to GUI data.

```
slider = guidata(fh);   % Get GUI data.
slider.previous_val = slider.val;
slider.val = ...;
...

guidata(fh,slider)  % Save GUI data before returning.
```

Both callbacks use the `guidata` function to retrieve and save the `slider` structure as GUI data.

Application Data

Application data can be associated with any object—a component, menu, or the figure itself. To access application data, a callback must know the name of the data and the handle of the component with which it is associated. Use the functions `setappdata`, `getappdata`, `isappdata`, and `rmappdata` to manage application data.

See "Application Data" on page 13-5 for more information about application data.

Application Data Example: Passing Data Between Components

The earlier topic, "Nested Functions Example: Passing Data Between Components" on page 13-9, uses nested function capabilities to initialize and maintain an error counter as well as the old and new values of the slider. This example shows you how to initialize and maintain the old and new values of the slider using application data (appdata) and make them available to the both callbacks. Refer to the earlier topic for details of the example.

The following code is similar to the earlier topic but uses application data to initialize and maintain the old and new slider values in the edit text and slider `Callbacks`. Copy this code into an M-file and save it in your current directory or on your path as `slider_gui.m`. Run the script by typing `slider_gui` at the command line.

```
function slider_gui
fh = figure('Position',[250 250 350 350]);
sh = uicontrol(fh,'Style','slider',...
               'Max',100,'Min',0,'Value',25,...
               'SliderStep',[0.05 0.2],...
               'Position',[300 25 20 300],...
               'Callback',@slider_callback);
eth = uicontrol(fh,'Style','edit',...
               'String',num2str(get(sh,'Value')),...
               'Position',[30 175 240 20],...
               'Callback',@edittext_callback);
sth = uicontrol(fh,'Style','text',...
               'String','Enter a value or click the slider.',...
               'Position',[30 215 240 20]);
number_errors = 0;
```

```
slider_data.val = 25;
% Create appdata with name 'slider'.
setappdata(fh,'slider',slider_data);
% --------------------------------------------------
% Set the value of the edit text component String property
% to the value of the slider.
   function slider_callback(hObject,eventdata)
      % Get 'slider' appdata.
      slider_data = getappdata(fh,'slider');
      slider_data.previous_val = slider_data.val;
      slider_data.val = get(hObject,'Value');
      set(eth,'String',num2str(get(slider_data.val)));
      sprintf('You moved the slider %d units.',...
              abs(slider_data.val - slider_data.previous_val))
      % Save 'slider' appdata before returning.
      setappdata(fh,'slider',slider_data)
end
% --------------------------------------------------
% Set the slider value to the number the user types in
% the edit text or display an error message.
   function edittext_callback(hObject,eventdata)
      % Get 'slider' appdata.
      slider_data = getappdata(fh,'slider');
      slider_data.previous_val = slider_data.val;
      slider_data.val = str2double(get(hObject,'String'));
      % Determine whether val is a number between the
      % slider's Min and Max. If it is, set the slider Value.
      if isnumeric(slider_data.val) && ...
            length(slider_data.val) == 1 && ...
         slider_data.val >= get(sh,'Min') && ...
         slider_data.val <= get(sh,'Max')
         set(sh,'Value',slider_data.val);
      else
      % Increment the error count, and display it.
         number_errors = number_errors+1;
         set(hObject,'String',...
             ['You have entered an invalid entry ',...
             num2str(number_errors),' times.']);
         slider_data.val = slider_data.previous_val;
      end
```

```
        % Save appdata before returning.
        setappdata(fh,'slider',slider_data);
    end
end
```

Slider Values. In this example, both the slider callbacks`slider_callback` and the edit text callback `edittext_callback` retrieve the application data structure `slider_data` which holds previous and current values of the slider. They then save the value, `slider_data.val` to `slider_data.previous_val` before retrieving the new value and assigning it to `slider_data.val`. Before returning, each callback saves the `slider_data` structure in the `slider` application data.

```
% Get 'slider' appdata.
slider_data = getappdata(fh,'slider');
slider_data.previous_val = slider_data.val;
slider_data.val = ...;
...
% Save 'slider' appdata before returning.
setappdata(fh,'slider',slider_data)
```

Both callbacks use the `getappdata` and `setappdata` functions to retrieve and save the `slider_data` structure as `slider` application data.

UserData Property

Every GUI component, and the figure itself, has a `UserData` property that you can use to store application-defined data. To access `UserData`, a callback must know the handle of the component with which a specific `UserData` property is associated.

Use the `get` function to retrieve `UserData`, and the `set` function to set it.

UserData Property Example: Passing Data Between Components

The previous topic, “Nested Functions Example: Passing Data Between Components” on page 13-9, uses nested function capabilities to initialize and maintain an error counter. This example shows you how to do the same thing using the edit text component’s `UserData` property to store the error count. Refer to the earlier example for example details.

The following code is the same as in the earlier topic but uses the `UserData` property to initialize and increment the error counter.

```
function slider_gui
fh = figure('Position',[250 250 350 350]);
sh = uicontrol(fh,'Style','slider',...
               'Max',100,'Min',0,'Value',25,...
               'SliderStep',[0.05 0.2],...
               'Position',[300 25 20 300],...
               'Callback',@slider_callback);
eth = uicontrol(fh,'Style','edit',...
               'String',num2str(get(sh,'Value')),...
               'Position',[30 175 240 20],...
               'Callback',@edittext_callback);
sth = uicontrol(fh,'Style','text',...
               'String','Enter a value or click the slider.',...
               'Position',[30 215 240 20]);
number_errors = 0;
slider.val = 25;
% Set edit text UserData property to slider structure.
set(eth,'UserData',slider)
% ------------------------------------------------------
% Set the value of the edit text component String property
% to the value of the slider.
   function slider_callback(hObject,eventdata)
      % Get slider from edit text UserData.
      slider = get(eth,'UserData');
      slider.previous_val = slider.val;
      slider.val = get(hObject,'Value');
      set(eth,'String',num2str(slider.val));
      sprintf('You moved the slider %d units.',...
              abs(slider.val - slider.previous_val))
      % Save slider in UserData before returning.
      set(eth,'UserData',slider)
   end
% ------------------------------------------------------
% Set the slider value to the number the user types in
% the edit text or display an error message.
   function edittext_callback(hObject,eventdata)
      % Get slider from edit text UserData.
```

```
        slider = get(eth,'UserData');
        slider.previous_val = slider.val;
        slider.val = str2double(get(hObject,'String'));
        % Determine whether slider.val is a number between the
        % slider's Min and Max. If it is, set the slider Value.
        if isnumeric(slider.val) && ...
              length(slider.val) == 1 && ...
           slider.val >= get(sh,'Min') && ...
           slider.val <= get(sh,'Max')
           set(sh,'Value',slider.val);
           sprintf('You moved the slider %d units.',...
                   abs(slider.val - slider.previous_val))
        else
        % Increment the error count, and display it.
           data = get(hObject,'UserData');
           data.number_errors = data.number_errors+1;
           set(hObject,'UserData',data);     % Save the changes.
           set(hObject,'String',...
              ['You have entered an invalid entry ',...
              num2str(number_errors),' times.']);
           slider.val = slider.previous_val;
        end
        % Save slider structure in UserData before returning.
        set(eth,'UserData',slider)
     end
  end
```

Slider Values. In this example, both the slider callbacks`slider_callback` and the edit text callback `edittext_callback` retrieve the structure `slider` from the edit text `UserData` property. The `slider` structure holds previous and current values of the slider. The callbacks then save the value `slider.val` to `slider.previous_val` before retrieving the new value and assigning it to `slider.val`. Before returning, each callback saves the `slider` structure in the edit text`UserData` property.

```
% Get slider structure from edit text UserData.
slider = get(eth,'UserData',slider);
slider.previous_val = slider.val;
slider.val = ...;
...
```

```
% Save slider structure in UserData before returning.
set(eth,'UserData',slider)
```

Both callbacks use the `get` and `set` functions to retrieve and save the `slider` structure in the edit text `UserData` property.

14

Managing Callback Execution

Callback Interruption

In this section...

Callback Execution

Callback execution is event driven and callbacks from different GUIs share the same event queue. In general, callbacks are triggered by user events such as a mouse click or key press. Because of this, you cannot predict, when a callback is requested, whether or not another callback is executing or, if one is, which callback it is.

If a callback is executing and the user triggers an event for which a callback is defined, that callback attempts to interrupt the callback that is already executing. When this occurs, MATLAB processes the callbacks according to these factors:

- The `Interruptible` property of the object whose callback is already executing. The `Interruptible` property specifies whether the executing callback can be interrupted.
- The `BusyAction` property of the object whose callback has just been triggered and wants to execute. The `BusyAction` property specifies whether a callback should be queued to await execution or be canceled.

How the Interruptible Property Works

An object's `Interruptible` property can be either `on` (the default) or `off`.

If the`Interruptible` property of the object whose callback is executing is `on`, the callback can be interrupted. However, it is interrupted only when it, or a function it triggers, calls `drawnow`, `figure`, `getframe`, `pause`, or `waitfor`. Before performing their defined tasks, these functions process any events in the event queue, including any waiting callbacks. If the executing callback, or

a function it triggers, calls none of these functions, it cannot be interrupted regardless of the value of its object's `Interruptible` property.

If the `Interruptible` property of the object whose callback is executing is `off`, the callback cannot be interrupted with the following exceptions. If the interrupting callback is a `DeleteFcn` or `CreateFcn` callback or a figure's `CloseRequest` or `ResizeFcn` callback, it interrupts an executing callback regardless of the value of the executing callback object's `Interruptible` property. These callbacks too can interrupt only when a `drawnow`, `figure`, `getframe`, `pause`, or `waitfor` function executes.

The callback properties to which `Interruptible` can apply depend on the objects for which the callback properties are defined:

- For figures, only callback routines defined for the `ButtonDownFcn`, `KeyPressFcn`, `KeyReleaseFcn`, `WindowButtonDownFcn`, `WindowButtonMotionFcn`, `WindowButtonUpFcn`, and `WindowScrollWheelFcn` are affected by the `Interruptible` property.
- For GUI components, `Interruptible` is applies to the `ButtonDownFcn`, `Callback`, `KeyPressFcn`, `SelectionChangeFcn`, `ClickedCallback`, `OffCallback`, and `OnCallback` properties, for the components for which these properties are defined.

How the Busy Action Property Works

An object's `BusyAction` property can be either `queue` (the default) or `cancel`. The `BusyAction` property of the interrupting callback's object is taken into account only if the `Interruptible` property of the executing callback's object is `off`, i.e., the executing callback is not interruptible.

If a noninterruptible callback is executing and an event (such as a mouse click) triggers a new callback, MATLAB uses the value of the new callback object's `BusyAction` property to decide whether to queue the requested callback or cancel it.

- If the `BusyAction` value is `queue`, the requested callback is added to the event queue and executes in its turn when the executing callback finishes execution.

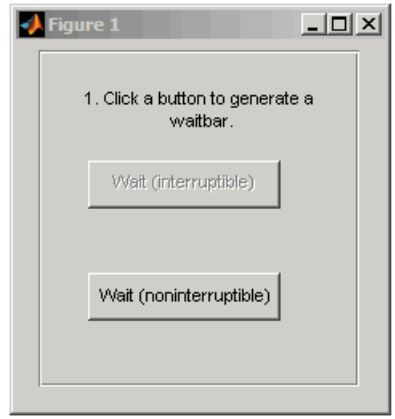
- If the value is `cancel`, the event is discarded and the requested callback does not execute.

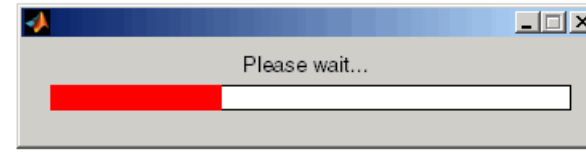
If an interruptible callback is executing, the requested callback runs when the executing callback terminates or calls `drawnow`, `figure`, `getframe`, `pause`, or `waitfor`. The `BusyAction` property of the requested callback's object has no effect.

Example

This example demonstrates control of callback interruption using the `Interruptible` and `BusyAction` properties. It creates two GUIs:

- The first GUI contains two push buttons, **Wait (interruptible)** whose `Interruptible` property is set to `on`, and **Wait (noninterruptible)**whose `Interruptible` property is set to `off`. Clicking either button triggers the button's `Callback` callback, which creates and updates a waitbar.



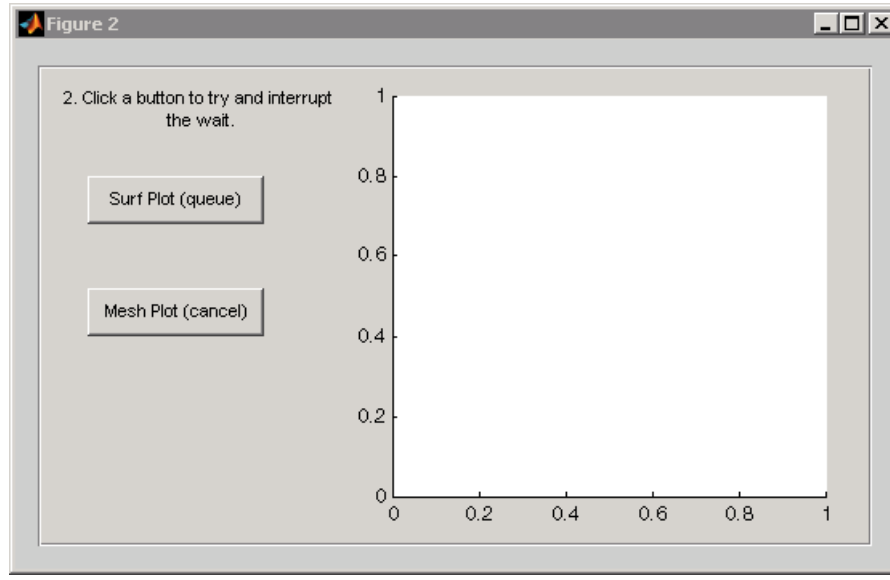


This code creates the two **Wait** buttons and specifies the callbacks that service them.

```
h_interrupt = uicontrol(h_panel1,'Style','pushbutton',...
                        'Position',[30,110,120,30],...
                        'String','Wait (interruptible)',...
                        'Interruptible','on',...
                        'Callback',@wait_interruptible);
```

```
h_noninterrupt = uicontrol(h_panel1,'Style','pushbutton',...
                           'Position',[30,40,120,30],...
                           'String','Wait (noninterruptible)',...
                           'Interruptible','off',...
                           'Callback',@wait_noninterruptible);
```

- The second GUI contains two push buttons, **Surf Plot (queue)** whose `BusyAction` property is set to `queue`, and **Mesh Plot (cancel)**whose `BusyAction` property is set to `cancel`. Clicking either button triggers the button's `Callback` callback to generate a plot in the axes.



This code creates the two plot buttons and specifies the callbacks that service them.

```
hsurf_queue = uicontrol(h_panel2,'Style','pushbutton',...
                           'Position',[30,200,110,30],...
                           'String','Surf Plot (queue)',...
                           'TooltipString','BusyAction = queue',...
                           'BusyAction','queue',...
                           'Callback',@surf_queue);
hmesh_cancel = uicontrol(h_panel2,'Style','pushbutton',...
                           'Position',[30,130,110,30],...
```

```
'String','Mesh Plot (cancel)',...
'BusyAction','cancel',...
'TooltipString','BusyAction = cancel',...
'Callback',@mesh_cancel);
```

Using the Example GUIs

Click here to run the example GUIs.

Note This link executes MATLAB commands and is designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the link.

To see the interplay of the `Interruptible` and `BusyAction` properties:

1. Click one of the **Wait** buttons in the first GUI. Both buttons create and update a waitbar.
2. While the waitbar is active, click either the **Surf Plot** or the **Mesh Plot** button in the second GUI. The **Surf Plot** button creates a `surf` plot using `peaks` data. The **Mesh Plot** button creates a `mesh` plot using the same data.

The following topics describe what happens when you click specific combinations of buttons:

- "Clicking a Wait Button" on page 14-6
- "Clicking a Plot Button" on page 14-7

Clicking a Wait Button.

The **Wait** buttons are the same except for their `Interruptible` properties. Their `Callback` callbacks, which are essentially the same, call the utility function `create_update_waitbar` which calls `waitbar` to create and update a waitbar. The **Wait (Interruptible)** button `Callback` callback,`wait_interruptible`, can be interrupted each time `waitbar` calls `drawnow`. The **Wait (Noninterruptible)** button `Callback`

callback,`wait_noninterruptible`, cannot be interrupted (except by specific callbacks listed in "How the Interruptible Property Works" on page 14-2).

This is the **Wait (Interruptible)** button `Callback` callback,`wait_interruptible`:

```
function wait_interruptible(hObject,eventdata)
    % Disable the other push button.
    set(h_noninterrupt,'Enable','off')
    % Clear the axes in the other GUI.
    cla(h_axes2,'reset')
    % Create and update the waitbar.
    create_update_waitbar
    % Enable the other push button
    set(h_noninterrupt,'Enable','on')
end
```

The callback first disables the other push button and clears the axes in the second GUI. It then calls the utility function `create_update_waitbar` to create and update a waitbar. When `create_update_waitbar` returns, it enables the other button.

Clicking a Plot Button. What happens when you click a **Plot** button depends on which **Wait** button you clicked first and the `BusyAction` property of the **Plot** button.

- If you click **Surf Plot**, whose `BusyAction` property is `queue`, MATLAB queues the **Surf Plot** callback `surf_queue`.

  If you clicked the **Wait (interruptible)** button first, `surf_queue` runs and displays the surf plot when the waitbar issues a call to `drawnow`, terminates, or is destroyed.

  If you clicked the **Wait (noninterruptible)** button first, `surf_queue` runs only when the waitbar terminates or is destroyed.

  This is the `surf_queue` callback:

  ```
  function surf_queue(hObject,eventdata)
          h_plot = surf(h_axes2,peaks_data);
     end
  ```

- If you click **Mesh Plot** , whose `BusyAction` property is `cancel`, after having clicked **Wait (noninterruptible)**, MATLAB discards the button click event and does not queue the `mesh_cancel` callback.

  If you click **Mesh Plot** after having clicked **Wait (interruptible)**, the **Mesh Plot** `BusyAction` property has no effect. MATLAB queues the **Mesh Plot** callback, `mesh_cancel`. It runs and displays the mesh plot when the waitbar issues a call to `drawnow`, terminates, or is destroyed.

  This is the `mesh_plot` callback:

```
function mesh_cancel(hObject,eventdata)
        h_plot = surf(h_axes2,peaks_data);
    end
```

View the Complete GUI M-File

If you are reading this in the MATLAB Help browser, you can click here to display a complete listing of the code used in this example in the MATLAB Editor.

Note This link executes MATLAB commands and is designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

15

Examples of GUIs Created Programmatically

Introduction

This chapter provides three examples that illustrate the application of certain techniques in programmatically created GUIs.

Each example lists the techniques it illustrates. These techniques include:

- Creation of a dialog that does not return until the user makes a choice
- Passing input arguments to the GUI when it is opened
- Obtaining output from the GUI when it returns
- Shielding the GUI from accidental changes
- Running the GUI across multiple platforms
- Making a GUI modal
- Sharing data among multiple GUIs
- Creating menus and context menus
- Creating toolbars
- Using an external utility function
- Achieving proper resize behavior

The examples all use nested functions. For information about using nested functions, see "Nested Functions" in the MATLAB Programming documentation.

GUI with Axes, Menu, and Toolbar

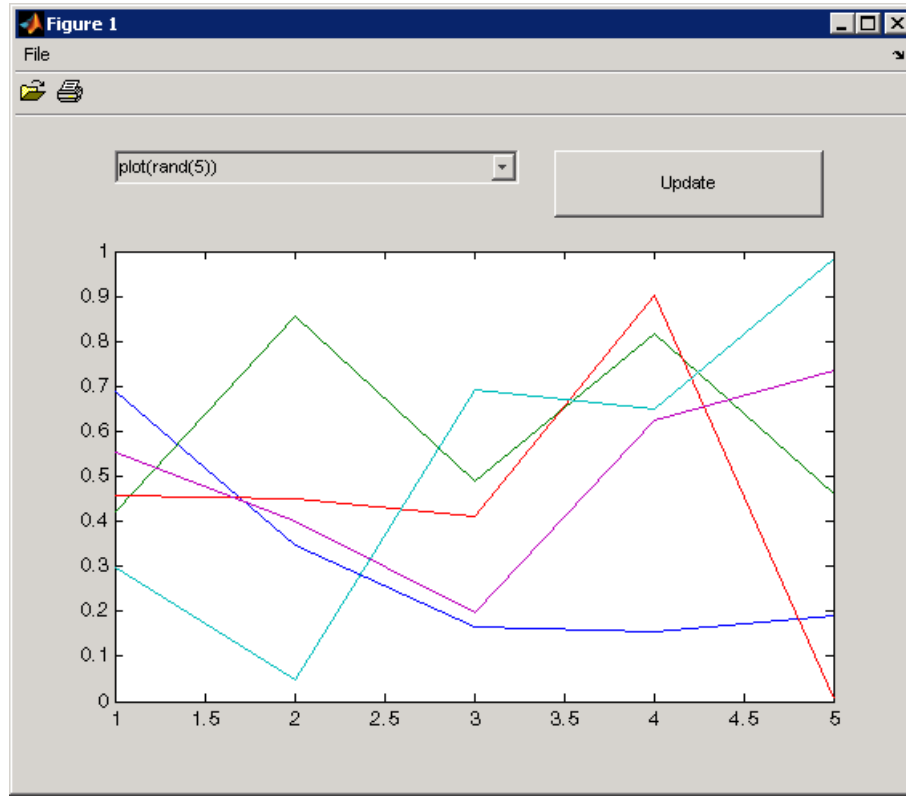
In this section...

The Example

This example creates a GUI that displays a user-selected plot in an axes. The GUI contains the following components:

- Axes
- Pop-up menu with a list of five plots
- Push button for updating the contents of the axes
- Menu bar File menu with three items: Open, Print, and Close
- Toolbar with two buttons that enable a user to open files and print the plot.

When you run the GUI, it initially displays a plot of five random numbers generated by the MATLAB `rand(5)` command, as shown in the following figure.



You can select other plots in the pop-up menu. Clicking the **Update** button displays the currently selected plot on the axes.

The GUI **File** menu has three items:

- **Open** displays a dialog from which you can open files on your computer.
- **Print** opens the Print dialog. Clicking **Yes** in the Print dialog prints the plot.
- **Close** closes the GUI.

The GUI toolbar has two buttons:

- The Open button  performs the same function as the **Open** menu item. It displays a dialog from which you can open files on your computer.
- The Print button  performs the same function as the **Print** menu item. It opens the Print dialog. Clicking **Yes** in the Print dialog prints the plot.

Techniques Used in the Example

This example illustrates the following techniques:

- Passing input arguments to the GUI when it is opened
- Obtaining output from the GUI when it returns
- Shielding the GUI from accidental changes
- Running the GUI across multiple platforms
- Creating menus
- Creating toolbars
- Achieving proper resize behavior

Note This example uses nested functions. For information about using nested functions, see “Nested Functions” in the MATLAB Programming documentation.

View and Run the Completed GUI M-Files

If you are reading this in the MATLAB Help browser, you can click the following links to display the MATLAB Editor with complete listings of the code used in this example.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display the main GUI M-file in the MATLAB Editor.
- Click here to display the utility `iconRead` M-file in the MATLAB Editor.
- Click here to run the GUI with axes, menu, and toolbar.

Creating the Data

The example defines two variables `mOutputArgs` and `mPlotTypes`.

`mOutputArgs` is a cell array that holds output values should the user request them. The example later assigns a default value to this argument.

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

`mPlotTypes` is a 5-by-2 cell array that holds the data to be plotted in the axes. The first column contains the strings that are used to populate the pop-up menu. The second column contains the functions, as anonymous function handles, that create the plots.

```
mPlotTypes = {...      % Example plot types shown by this GUI
              'plot(rand(5))',        @(a)plot(a,rand(5));
              'plot(sin(1:0.01:25))', @(a)plot(a,sin(1:0.01:25));
              'bar(1:.5:10)',         @(a)bar(a,1:.5:10);
              'plot(membrane)',       @(a)plot(a,membrane);
              'surf(peaks)',          @(a)surf(a,peaks)};
```

Because the data is created at the top level of the GUI function, it is available to all callbacks and other functions in the GUI.

See “Anonymous Functions” in the MATLAB Programming documentation for information about using anonymous functions.

Creating the GUI and Its Components

Like the data, the components are created at the top level so that their handles are available to all callbacks and other functions in the GUI.

- “The Main Figure” on page 15-7
- “The Axes” on page 15-7
- “The Pop-Up Menu” on page 15-8

The Main Figure

The following statement creates the figure for GUI.

```
hMainFigure = figure(...       % The main GUI figure
                     'MenuBar','none', ...
                     'Toolbar','none', ...
                     'HandleVisibility','callback', ...
                     'Color', get(0,...
                              'defaultuicontrolbackgroundcolor'));
```

- The `figure` function creates the GUI figure.
- Setting the `MenuBar` and `Toolbar` properties to `none`, prevents the standard menu bar and toolbar from displaying.
- Setting the `HandleVisibility` property to `callback` ensures that the figure can be accessed only from within a GUI callback, and cannot be drawn into or deleted from the command line.
- The `Color` property defines the background color of the figure. In this case, it is set to be the same as the default background color of `uicontrol` objects, such as the **Update** push button. The factory default background color of `uicontrol` objects is the system default and can vary from system to system. This statement ensures that the figure's background color matches the background color of the components.

See the Figure Properties reference page for information about figure properties and their default values.

The Axes

The following statement creates the axes.

```
hPlotAxes = axes(...    % Axes for plotting the selected plot
                 'Parent', hMainFigure, ...
                 'Units', 'normalized', ...
                 'HandleVisibility','callback', ...
```

```
                    'Position',[0.11 0.13 0.80 0.67]);
```

- The `axes` function creates the axes. Setting the axes `Parent` property to `hMainFigure` makes it a child of the main figure.
- Setting the `Units` property to `normalized` ensures that the axes resizes proportionately when the GUI is resized.
- The `Position` property is a 4-element vector that specifies the location of the axes within the figure and its size: [distance from left, distance from bottom, width, height]. Because the units are normalized, all values are between 0 and 1.

Note If you specify the `Units` property, then the `Position` property, and any other properties that depend on the value of the `Units` property, should follow the `Units` property specification.

See the Axes Properties reference page for information about axes properties and their default values.

The Pop-Up Menu

The following statement creates the pop-up menu.

```
hPlotsPopupmenu = uicontrol(... % List of available types of plot
                      'Parent', hMainFigure, ...
                      'Units','normalized',...
                      'Position',[0.11 0.85 0.45 0.1],...
                      'HandleVisibility','callback', ...
                      'String',mPlotTypes(:,1),...
                      'Style','popupmenu');
```

- The `uicontrol` function creates various user interface controls based on the value of the `Style` property. Here the `Style` property is set to `popupmenu`.
- For a pop-up menu, the `String` property defines the list of items in the menu. Here it is defined as a 5-by-1 cell array of strings derived from the cell array `mPlotTypes`.

See the Uicontrol Properties reference page for information about properties of `uicontrol` objects and their default values.

The Update Push Button

This statement creates the **Update** push button as a uicontrol object.

```
hUpdateButton = uicontrol(... % Button for updating selected plot
                  'Parent', hMainFigure, ...
                  'Units','normalized',...
                  'HandleVisibility','callback', ...
                  'Position',[0.6 0.85 0.3 0.1],...
                  'String','Update',...
                  'Callback', @hUpdateButtonCallback);
```

- The `uicontrol` function creates various user interface controls based on the value of the `Style` property. This statement does not set the `Style` property because its default is `pushbutton`.
- For a push button, the `String` property defines the label on the button. Here it is defined as the string `Update`.
- Setting the `Callback` property to `@hUpdateButtonCallback` defines the name of the callback function that services the push button. That is, clicking the push button triggers the execution of the named callback. This callback function is defined later in the script.

See the Uicontrol Properties reference page for information about properties of `uicontrol` objects and their default values.

The File Menu and Its Menu Items

These statements define the **File** menu and the three items it contains.

```
hFileMenu      =   uimenu(...       % File menu
                       'Parent',hMainFigure,...
                       'HandleVisibility','callback', ...
                       'Label','File');
hOpenMenuitem  =   uimenu(...       % Open menu item
                       'Parent',hFileMenu,...
                       'Label','Open',...
                       'HandleVisibility','callback', ...
```

```
                        'Callback', @hOpenMenuitemCallback);
hPrintMenuitem  =  uimenu(...       % Print menu item
                        'Parent',hFileMenu,...
                        'Label','Print',...
                        'HandleVisibility','callback', ...
                        'Callback', @hPrintMenuitemCallback);
hCloseMenuitem  =  uimenu(...       % Close menu item
                        'Parent',hFileMenu,...
                        'Label','Close',...
                        'Separator','on',...
                        'HandleVisibility','callback', ...
                        'Callback', @hCloseMenuitemCallback');
```

- The `uimenu` function creates both the main menu, **File**, and the items it contains. For the main menu and each of its items, set the `Parent` property to the handle of the desired parent to create the menu hierarchy you want. Here, setting the `Parent` property of the **File** menu to `hMainFigure` makes it the child of the main figure. This statement creates a menu bar in the figure and puts the **File** menu on it.

  For each of the menu items, setting its `Parent` property to the handle of the parent menu, `hFileMenu`, causes it to appear on the **File** menu.

- For the main menu and each item on it, the `Label` property defines the strings that appear in the menu.
- Setting the `Separator` property to `on` for the **Close** menu item causes a separator line to be drawn above this item.
- For each of the menu items, the `Callback` property specifies the callback that services that item. In this example, no callback services the **File** menu itself. These callbacks are defined later in the script.

See the Uicontrol Properties reference page for information about properties of `uicontrol` objects and their default values.

The Toolbar and Its Tools

These statements define the toolbar and the two buttons it contains.

```
hToolbar = uitoolbar(...    % Toolbar for Open and Print buttons
                  'Parent',hMainFigure, ...
                  'HandleVisibility','callback');
```

```
hOpenPushtool  =  uipushtool(...   % Open toolbar button
                   'Parent',hToolbar,...
                   'TooltipString','Open File',...
                   'CData',iconRead(fullfile(matlabroot,...
                       'toolbox\matlab\icons\opendoc.mat')),...
                   'HandleVisibility','callback', ...
                   'ClickedCallback', @hOpenMenuitemCallback);
hPrintPushtool = uipushtool(...    % Print toolbar button
                    'Parent',hToolbar,...
                    'TooltipString','Print Figure',...
                    'CData',iconRead(fullfile(matlabroot,...
                        'toolbox\matlab\icons\printdoc.mat')),...
                    'HandleVisibility','callback', ...
                    'ClickedCallback', @hPrintMenuitemCallback);
```

- The `uitoolbar` function creates the toolbar on the main figure.
- The `uipushtool` function creates the two push buttons on the toolbar.
- The uipushtool `TooltipString` property assigns a tool tip that displays when the GUI user moves the mouse pointer over the button and leaves it there.
- The `CData` property specifies a truecolor image that displays on the button. For these two buttons, the utility `iconRead` function supplies the image. If you are reading this in the MATLAB Help browser, click here to display this utility M-file in the MATLAB Editor.
- For each of the uipushtools, the `ClickedCallback` property specifies the callback that executes when the GUI user clicks the button. Note that the Open push button and the Print push button use the same callbacks as their counterpart menu items.

See “Creating Toolbars” on page 11-56 for more information.

Initializing the GUI

These statements create the plot that appears in the GUI when it first displays, and, if the user provides an output argument when running the GUI, define the output that is returned to the user .

```
% Update the plot with the initial plot type
```

```
localUpdatePlot();

% Define default output and return it if it is requested by users
mOutputArgs{1} = hMainFigure;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

- The `localUpdatePlot` function plots the selected plot type in the axes. For a pop-up menu, the uicontrol `Value` property specifies the index of the selected menu item in the `String` property. Since the default value is 1, the initial selection is `'plot(rand(5))'`. The `localUpdatePlot` function is a helper function that is defined later in the script, at the same level as the callbacks.
- The default output is the handle of the main figure.

Defining the Callbacks

This topic defines the callbacks that service the components of the GUI. Because the callback definitions are at a lower level than the component definitions and the data created for the GUI, they have access to all data and component handles.

Although the GUI has six components that are serviced by callbacks, there are only four callback functions. This is because the **Open** menu item and the Open toolbar button share the same callbacks. Similarly, the **Print** menu item and the Print toolbar button share the same callbacks.

- "Update Button Callback" on page 15-13
- "Open Menu Item Callback" on page 15-13
- "Print Menu Item Callback" on page 15-14
- "Close Menu Item Callback" on page 15-15

Note These are the callbacks that were specified in the component definitions, "Creating the GUI and Its Components" on page 15-6.

Update Button Callback

The `hUpdateButtonCallback` function services the **Update** push button. Clicking the **Update** button triggers the execution of this callback function.

```
function hUpdateButtonCallback(hObject, eventdata)
  % Callback function run when the Update button is pressed
      localUpdatePlot();
  end
```

The `localUpdatePlot` function is a helper function that plots the selected plot type in the axes. It is defined later in the script, "Helper Function: Plotting the Plot Types" on page 15-16.

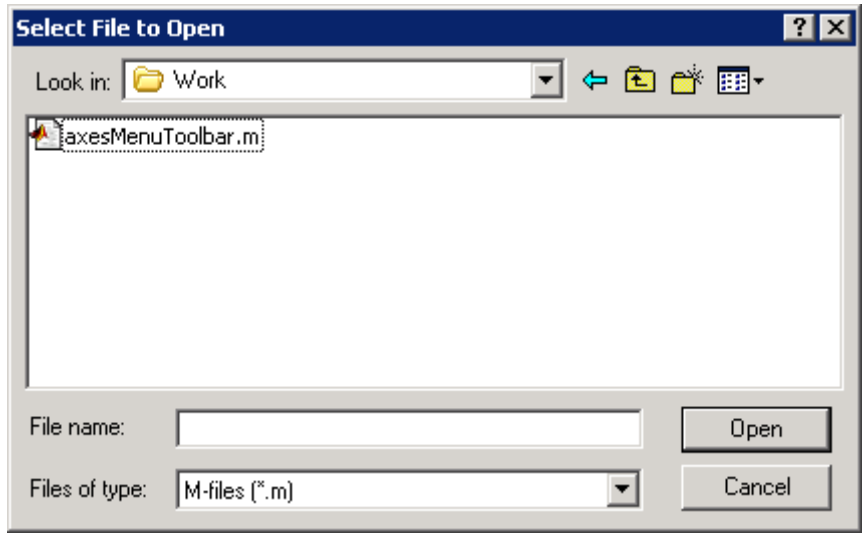
Note MATLAB automatically passes `hUpdateButtonCallback` two arguments, `hObject` and `eventdata`, because the **Update** push button component `Callback` property, `@hUpdateButtonCallback`, is defined as a function handle. `hObject` contains the handle of the component that triggered execution of the callback. `eventdata` is reserved for future use. The function definition line for your callback must account for these two arguments.

Open Menu Item Callback

The `hOpenMenuitemCallback` function services the **Open** menu item and the Open toolbar button  . Selecting the menu item or clicking the toolbar button triggers the execution of this callback function.

```
function hOpenMenuitemCallback(hObject, eventdata)
% Callback function run when the Open menu item is selected
    file = uigetfile('*.m');
    if ~isequal(file, 0)
        open(file);
    end
end
```

The `hOpenMenuitemCallback` function first calls the `uigetfile` function to open the standard dialog box for retrieving files. This dialog box lists all M-files. If `uigetfile` returns a filename, the function then calls the `open` function to open it.

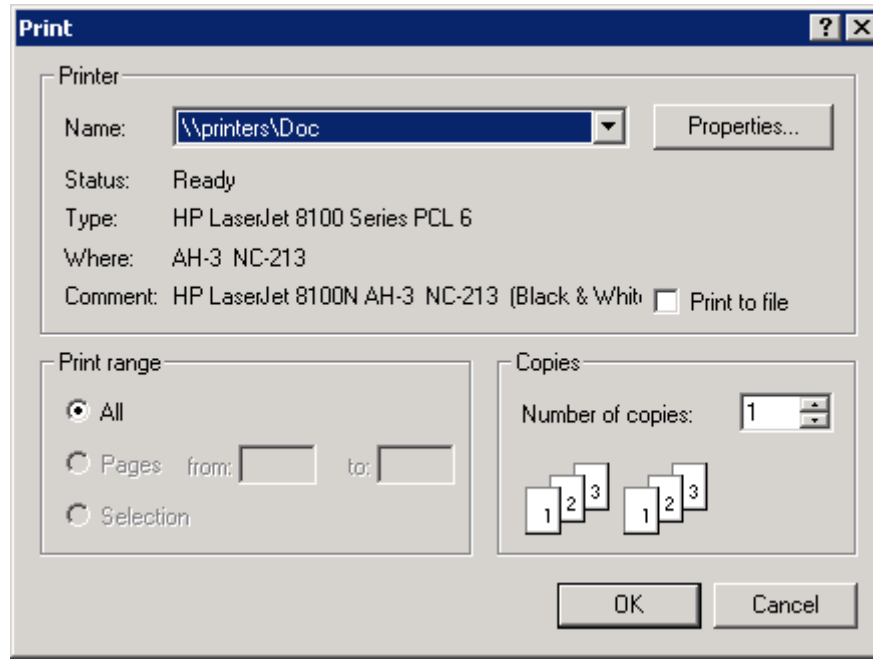


Print Menu Item Callback

The `hPrintMenuitemCallback` function services the **Print** menu item and the Print toolbar button. Selecting the menu item or clicking the toolbar button triggers the execution of this callback function.

```
function hPrintMenuitemCallback(hObject, eventdata)
% Callback function run when the Print menu item is selected
    printdlg(hMainFigure);
end
```

The `hPrintMenuitemCallback` function calls the `printdlg` function. This function opens the standard dialog box for printing the current figure.



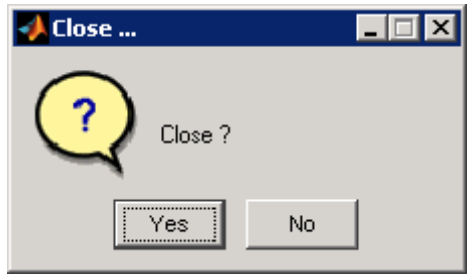
Close Menu Item Callback

The `hCloseMenuitemCallback` function services the **Close** menu item. It executes when the GUI user selects **Close** from the **File** menu.

```
function hCloseMenuitemCallback(hObject, eventdata)
% Callback function run when the Close menu item is selected
    selection = ...
       questdlg(['Close ' get(hMainFigure,'Name') '?'],...
                ['Close ' get(hMainFigure,'Name') '...'],...
                'Yes','No','Yes');
    if strcmp(selection,'No')
        return;
    end

    delete(hMainFigure);
```

```
end
```

The `hCloseMenuitemCallback` function calls the `questdlg` function to create and open the question dialog box shown in the following figure.



If the user clicks the **No** button, the callback returns. If the user clicks the **Yes** button, the callback deletes the GUI.

See “Helper Function: Plotting the Plot Types” on page 15-16 for a description of the `localUpdatePlot` function.

Helper Function: Plotting the Plot Types

The example defines the `localUpdatePlot` function at the same level as the callback functions. Because of this, `localUpdatePlot` has access to the same data and component handles.

```
function localUpdatePlot
% Helper function for plotting the selected plot type
    mPlotTypes{get(hPlotsPopupmenu, 'Value'), 2}(hPlotAxes);
end
```

The `localUpdatePlot` function uses the pop-up menu `Value` property to identify the selected menu item from the first column of the `mPlotTypes` 5-by-2 cell array, then calls the corresponding anonymous function from column two of the cell array to create the plot in the axes.

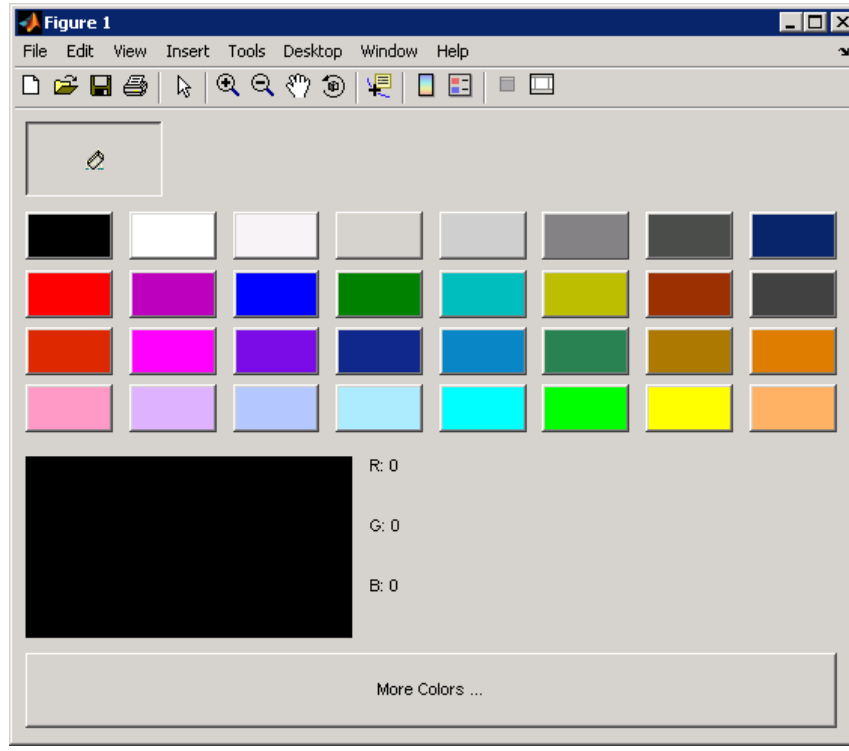
Color Palette

In this section...

The Example

This example creates a GUI, colorPalette, that enables a user to select a color from a color palette or display the standard color selection dialog box. Another example, "Icon Editor" on page 15-29, embeds the colorPalette, as the child of a panel, in a GUI you can use to design an icon.

The `colorPalette` function populates a GUI figure or panel with a color palette. The figure below shows the palette as the child of a figure.



The Components

The colorPalette includes the following components:

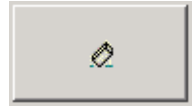
- An array of color cells defined as toggle buttons
- An Eraser toggle button with the  icon
- A button group that contains the array of color cells and the eraser button. The button group provides exclusive management of these toggle buttons.
- A **More Colors** push button
- A preview of the selected color, below the color cells, defined as a text component

- Text components to specify the red, blue, and green color values

Using the Color Palette

These are the basic steps for using the color palette.

1. Clicking a color cell toggle button:
   - Displays the selected color in the preview area.
   - The red, green, and blue values for the newly selected color are displayed in the **R**, **G**, and **B** fields to the right of the preview area.
   - Causes `colorPalette` to return a function handle that the host GUI can use to get the currently selected color.
2. Clicking the Eraser toggle button, causes `colorPalette` to return a value, `NaN`, that the host GUI can use to remove color from a data point.



3. Clicking the **More Colors** button displays the standard dialog box for setting a color.

Calling the colorPalette Function

You can call the `colorPalette` function with a statement such as

```
mGetColorFcn = colorPalette('Parent',hPaletteContainer)
```

The `colorPalette` function accepts property value pairs as input arguments. Only the custom property `Parent` is supported. This property specifies the handle of the parent figure or panel that contains the color palette. If the call to `colorPalette` does not specify a parent, it uses the current figure, `gcf`. Unrecognized property names or invalid values are ignored.

`colorPalette` returns a function handle that the host GUI can call to get the currently selected color. The host GUI can use the returned function handle at any time before the color palette is destroyed. For more information, see “Sharing Data Between Two GUIs” on page 15-26 for implementation details. “Icon Editor” on page 15-29 is an example of a host GUI that uses the colorPalette.

Techniques Used in the Example

This example illustrates the following techniques:

- Retrieving output from the GUI when it returns.
- Supporting custom input property/value pairs with data validation.
- Sharing data between two GUIs

See “Icon Editor” on page 15-29 for examples of these and other programming techniques.

Note This example uses nested functions. For information about using nested functions, see “Nested Functions” in the MATLAB Programming documentation.

View and Run the Completed GUI M-File

If you are reading this in the MATLAB Help browser, you can click the following link to display the MATLAB Editor with a complete listing of the code that is discussed in the following sections.

Note The following link executes MATLAB commands and is designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the link.

- Click here to display the main GUI M-file in the MATLAB Editor.
- Click here to run the colorPalette GUI.

Subfunction Summary

The color palette example includes the callbacks listed in the following table.

Function	Description
`colorCellCallback`	Called by `hPalettePanelSelectionChanged` when any color cell is clicked.
`eraserToolCallback`	Called by `hPalettePanelSelectionChanged` when the Eraser button is clicked.
`hMoreColorButtonCallback`	Executes when the **More Colors** button is clicked. It calls `uisetcolor` to open the standard color-selection dialog box, and calls `localUpdateColor` to update the preview.
`hPalettePanelSelectionChanged`	Executes when the GUI user clicks on a new color. This is the `SectionChangeFcn` callback of the `uibuttongroup` that exclusively manages the tools and color cells that it contains. It calls the appropriate callback to service each of the tools and color cells.

Note Three `eventdata` fields are defined for use with button groups (`uibuttongroup`). These fields enable you to determine the previous and current radio or toggle button selections maintained by the button group. See `SelectionChangeFcn` in the Uibuttongroup Properties reference page for more information.

The example also includes the helper functions listed in the following table.

Function	Description
`layoutComponent`	Dynamically creates the Eraser tool and the color cells in the palette. It calls `localDefineLayout`.
`localUpdateColor`	Updates the preview of the selected color.
`getSelectedColor`	Returns the currently selected color which is then returned to the `colorPalette` caller.
`localDefineLayout`	Calculates the preferred color cell and tool sizes for the GUI. It calls `localDefineColors` and `localDefineTools`
`localDefineTools`	Defines the tools shown in the palette. In this example, the only tool is the **Eraser** button.
`localDefineColors`	Defines the colors that are shown in the array of color cells.
`processUserInputs`	Determines if the property in a property/value pair is supported. It calls `localValidateInput`.
`localValidateInput`	Validates the value in a property/value pair.

M-File Structure

The colorPalette is programmed using nested functions. Its M-file is organized in the following sequence:

1. Comments displayed in response to the `help` command.
2. Data creation. Because the example uses nested functions, defining this data at the top level makes the data accessible to all functions without having to pass them as arguments.
3. Command line input processing.
4. GUI figure and component creation.
5. GUI initialization.
6. Return output if it is requested.

7 Callback definitions. These callbacks, which service the GUI components, are subfunctions of the `colorPalette` function and so have access to the data and component handles created at the top level, without their having to be passed as arguments.

8 Helper function definitions. These helper functions are subfunctions of the `colorPalette` function and so have access to the data and component handles created at the top level, without their having to be passed as arguments.

Note For information about using nested functions, see "Nested Functions" in the MATLAB Programming documentation.

GUI Programming Techniques

This topic explains the following GUI programming techniques as they are used in the creation of the colorPalette.

- "Passing Input Arguments to a GUI" on page 15-24
- "Passing Output to a Caller on Returning" on page 15-26
- "Sharing Data Between Two GUIs" on page 15-26

See "Icon Editor" on page 15-29 for additional examples of these and other programming techniques.

Passing Input Arguments to a GUI

Inputs to the GUI are custom property/value pairs. `colorPalette` allows one such property: `Parent`. The names are case insensitive. The `colorPalette` syntax is

```
mGetColorFcn = colorPalette('Parent',hPaletteContainer)
```

Definition and Initialization of the Properties. The `colorPalette` function first defines a variable `mInputArgs` as `varargin` to accept the user input arguments.

```
mInputArgs = varargin;  % Command line arguments when invoking
                        % the GUI
```

The `colorPalette` function then defines the valid custom properties in a 3-by-3 cell array.

```
mPropertyDefs = {...    % The supported custom property/value
                        % pairs of this GUI
              'parent',   @localValidateInput, 'mPaletteParent';
```

- The first column contains the property name.
- The second column contains a function handle for the function, `localValidateInput`, that validates the input property values.
- The third column is the local variable that holds the value of the property.

`colorPalette` then initializes the properties with default values.

```
mPaletteParent = [];  % Use input property 'parent' to initialize
```

Processing the Input Arguments. The `processUserInputs` helper function processes the input property/value pairs. `colorPalette` calls `processUserInputs` before it creates the components, to determine the parent of the components.

```
% Process the command line input arguments supplied when
% the GUI is invoked
processUserInputs();
```

1. `processUserInputs` sequences through the inputs, if any, and tries to match each property name to a string in the first column of the `mPropertyDefs` cell array.
2. If it finds a match, `processUserInputs` assigns the value that was input for the property to its variable in the third column of the `mPropertyDefs` cell array.
3. `processUserInputs` then calls the helper function specified in the second column of the `mPropertyDefs` cell array to validate the value that was passed in for the property.

Passing Output to a Caller on Returning

If a host GUI calls the `colorPalette` function with an output argument, it returns a function handle that the host GUI can call to get the currently selected color.

The host GUI calls `colorPalette` only once. The call creates the color palette in the specified parent and then returns the function handle. The host GUI can call the returned function at any time before the color palette is destroyed.

The data definition section of the `colorPalette` M-file creates a cell array to hold the output:

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

Just before returning, `colorPalette` assigns the function handle, `mgetSelectedColor`, to the cell array `mOutputArgs` and then assigns `mOutputArgs` to `varargout` to return the arguments.

```
mOutputArgs{} = @getSelectedColor;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

Sharing Data Between Two GUIs

The iconEditor embeds a GUI, the colorPalette, to enable the user to select colors for the icon cells. The colorPalette returns a function handle the iconEditor. The iconEditor can then call the returned function at any time to get the selected color.

The colorPalette GUI. The `colorPalette` function defines a cell array, `mOutputArgs`, to hold its output arguments.

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

Just before returning, `colorPalette` assigns `mOutputArgs` the function handle for its `getSelectedColor` helper function and then assigns `mOutputArgs` to `varargout` to return the arguments.

```
% Return user defined output if it is requested
mOutputArgs{1} =@getSelectedColor;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

The iconEditor executes the colorPalette's `getSeclectedColor` function whenever it invokes the function that colorPalette returns to it.

```
function color = getSelectedColor
% function returns the currently selected color in this
% colorPlatte
    color = mSelectedColor;
```

The iconEditor GUI. The `iconEditor` function calls `colorPalette` only once and specifies its parent to be a panel in the iconEditor.

```
% Host the ColorPalette in the PaletteContainer and keep the
% function handle for getting its selected color for editing
% icon.
mGetColorFcn = colorPalette('parent', hPaletteContainer);
```

This call creates the colorPalette as a component of the iconEditor and then returns a function handle that `iconEditor` can call to get the currently selected color.

The iconEditor's `localEditColor` helper function calls `mGetColorFcn`, the function returned by `colorPalette`, to execute the colorPalette's `getSelectedColor` function.

```
function localEditColor
% helper function that changes the color of an icon data
% point to that of the currently selected color in
% colorPalette
    if mIsEditingIcon
        pt = get(hIconEditAxes,'currentpoint');
        x = ceil(pt(1,1));
```

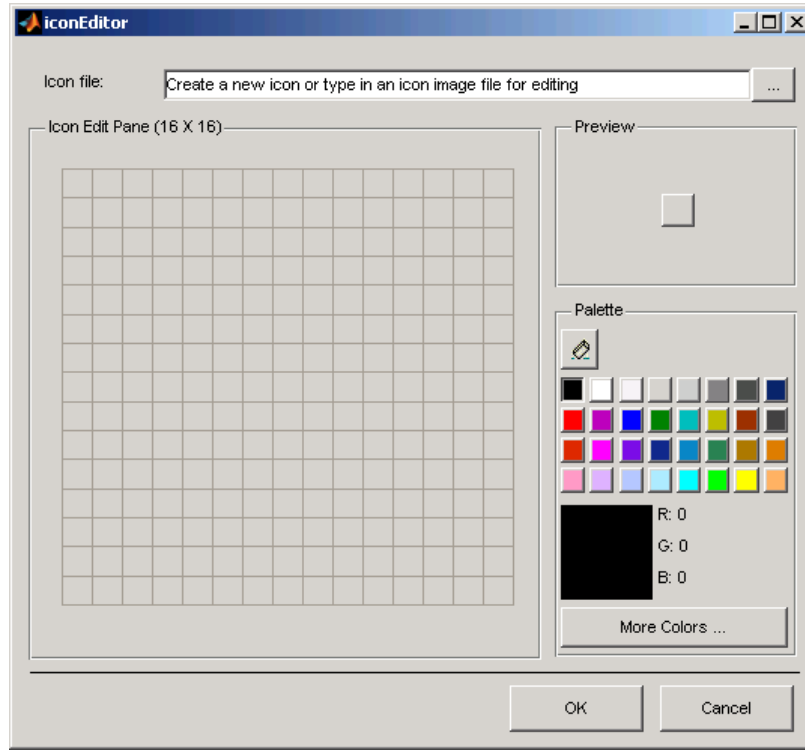
```
            y = ceil(pt(1,2));
            color = mGetColorFcn();

            % update color of the selected block
            mIconCData(y, x,:) = color;

            localUpdateIconPlot();
        end
    end
```

Icon Editor

In this section...

The Example

This example creates a GUI that enables the user to create or edit an icon. The figure below shows the editor.



The Components

The GUI includes the following components:

- A edit text that instructs the user or contains the name of the file to be edited. The edit text is labeled using a static text.
- A push button to the right of the edit text enables the user to select an existing icon file for editing.
- A panel containing an axes. The axes displays a 16-by-16 grid for drawing an icon.
- A panel containing a button that shows a preview of the icon as it is being created.
- A color palette that is created in a separate script and embedded in this GUI. See “Color Palette” on page 15-17.

- A panel, configured as a line, that separates the icon editor from the **OK** and **Cancel** buttons.
- An **OK** push button that causes the GUI to return the icon as an m-by-n-by-3 array and closes the GUI.
- A **Cancel** push button that closes the GUI without returning the icon.

Using the Icon Editor

These are the basic steps to create an icon:

1. Start the icon editor with a command such as

   ```
   myicon = iconEditor('iconwidth',32,'iconheight',56);
   ```

   where the `iconwidth` and `iconheight` properties specify the icon size in pixels.

2. Color the squares in the grid.
   - Click a color cell in the palette. That color is then displayed in the palette preview.
   - Click in specific squares of the grid to transfer the selected color to those squares.
   - Hold down the left mouse button and drag the mouse over the grid to transfer the selected color to the squares that you touch.
   - Change a color by writing over it with another color.

3. Erase the color in some squares.
   - Click the **Eraser** button on the palette.
   - Click in specific squares to erase those squares.
   - Click and drag the mouse to erase the squares that you touch.
   - Click a color cell to disable the Eraser.

4. Click **OK** to close the GUI and return, in `myicon`, the icon you created – as a 32-by-65-by-3 array. Click **Cancel** to close the GUI and return an empty array `[]` in `myicon`.

Techniques Used in the Example

This example illustrates the following GUI programming techniques:

- Creating a GUI that does not return a value until the user makes a choice.
- Retrieving output from the GUI when it returns.
- Supporting custom input property/value pairs with data validation.
- Protecting a GUI from being changed from the command line.
- Creating a GUI that runs on multiple platforms
- Sharing data between two GUIs
- Achieving the proper resize behavior

Note This example uses nested functions. For information about using nested functions, see "Nested Functions" in the MATLAB Programming documentation.

View and Run the Completed GUI M-Files

If you are reading this in the MATLAB Help browser, you can click the following links to display the MATLAB Editor with a complete listing of the code that is discussed in the following sections.

Note The following links execute MATLAB commands and are designed to work within the MATLAB Help browser. If you are reading this online or in PDF, you should go to the corresponding section in the MATLAB Help Browser to use the links.

- Click here to display the main GUI M-file in the MATLAB Editor.
- Click here to display the utility iconRead M-file in the MATLAB Editor.
- Click here to run the iconEditor GUI.

Subfunction Summary

The icon editor example includes the callbacks listed in the following table.

Function	Description
`hMainFigureWindowButtonDownFcn`	Executes when the user clicks a mouse button anywhere in the GUI figure. It calls `localEditColor`.
`hMainFigureWindowButtonUpFcn`	Executes when the user releases the mouse button.
`hMainFigureWindowButtonMotionFcn`	Executes when the user drags the mouse anywhere in the figure with a button pressed. It calls `localEditColor`.
`hIconFileEditCallback`	Executes after the user manually changes the filename of the icon to be edited. It calls `localUpdateIconPlot`.
`hIconFileEditButtondownFcn`	Executes the first time the user clicks the **Icon file** edit box.
`hOKButtonCallback`	Executes when the user clicks the **OK** push button.
`hCancelButtonCallback`	Executes when the user clicks the **Cancel** push button.
`hIconFileButtonCallback`	Executes when the user clicks the **Icon file** push button  . It calls `localUpdateIconPlot`.

The example also includes the helper functions listed in the following table.

Function	Description
`localEditColor`	Changes the color of an icon data point to the currently selected color. Call the function `mGetColorFcn` returned by the `colorPalette` function. It also calls `localUpdateIconPlot`.
`localUpdateIconPlot`	Updates the icon preview. It also updates the axes when an icon is read from a file.
`processUserInputs`	Determines if the property in a property/value pair is supported. It calls `localValidateInput`.
`localValidateInput`	Validates the value in a property/value pair.
`prepareLayout`	Makes changes needed for look and feel and for running on multiple platforms.

M-File Structure

The iconEditor is programmed using nested functions. Its M-file is organized in the following sequence:

1. Comments displayed in response to the `help` command.
2. Data creation. Because the example uses nested functions, defining this data at the top level makes the data accessible to all functions without having to pass them as arguments.
3. GUI figure and component creation.
4. Command line input processing.
5. GUI initialization.
6. Block execution of the program until the GUI user clicks **OK** or **Cancel**.
7. Return output if requested.
8. Callback definitions. These callbacks, which service the GUI components, are subfunctions of the `iconEditor` function and so have access to the data and component handles created at the top level, without their having to be passed as arguments.
9. Helper function definitions. These helper functions are subfunctions of the `iconEditor` function and so have access to the data and component handles created at the top level, without their having to be passed as arguments.

Note For information about using nested functions, see "Nested Functions" in the MATLAB Programming documentation.

GUI Programming Techniques

This topic explains the following GUI programming techniques as they are used in the creation of the iconEditor.

- "Returning Only After the User Makes a Choice" on page 15-36
- "Passing Input Arguments to a GUI" on page 15-37

Returning Only After the User Makes a Choice

At the end of the initialization code, and just before returning, iconEditor calls `uiwait` with the handle of the main figure to make the GUI blocking.

```
% Make the GUI blocking
uiwait(hMainFigure);

% Return the edited icon CData if it is requested
mOutputArgs{1} =hMainFigure;
mOutputArgs{2} =mIconCData;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

Placement of the call to `uiwait` is important. Calling `uiwait` stops the sequential execution of the `iconEdit` M-file after the GUI is initialized and just before the file would return the edited icon data.

When the user clicks the **OK** button, its callback, `hOKButtonCallback`, calls `uiresume` which enables the M-file to resume execution where it stopped and return the edited icon data.

```
function hOKButtonCallback(hObject, eventdata)
% Callback called when the OK button is pressed
    uiresume;
    delete(hMainFigure);
end
```

When the user clicks the **Cancel** button, its callback, `hOCancelButtonCallback`, effectively deletes the icon data then calls `uiresume`. This enables the M-file to resume execution where it stopped but it returns a null matrix.

```
function hCancelButtonCallback(hObject, eventdata)
% Callback called when the Cancel button is pressed
    mIconCData =[];
    uiresume;
    delete(hMainFigure);
end
```

Passing Input Arguments to a GUI

Inputs to the GUI are custom property/value pairs. `iconEdit` allows three such properties: `IconWidth`, `IconHeight`, and `IconFile`. The names are caseinsensitive.

Definition and Initialization of the Properties. The `iconEdit` first defines a variable `mInputArgs` as `varargin` to accept the user input arguments.

```
mInputArgs = varargin;  % Command line arguments when invoking
                        % the GUI
```

The `iconEdit` function then defines the valid custom properties in a 3-by-3 cell array.

```
mPropertyDefs = {...    % Supported custom property/value
                        % pairs of this GUI
              'iconwidth',   @localValidateInput, 'mIconWidth';
              'iconheight',  @localValidateInput, 'mIconHeight';
              'iconfile',    @localValidateInput, 'mIconFile'};
```

- The first column contains the property name.
- The second column contains a function handle for the function, `localValidateInput`, that validates the input property values.
- The third column is the local variable that holds the value of the property.

`iconEdit` then initializes the properties with default values.

```
mIconWidth = 16;   % Use input property 'iconwidth' to initialize
mIconHeight = 16;  % Use input property 'iconheight' to initialize
mIconFile = fullfile(matlabroot,'/toolbox/matlab/icons/');
```

The values of `mIconWidth` and `mIconHeight` are interpreted as pixels. The `fullfile` function builds a full filename from parts.

Processing the Input Arguments. The `processUserInputs` helper function processes the input property/value pairs. `iconEdit` calls `processUserInputs` after the layout is complete and just before it needs the inputs to initialize the GUI.

```
% Process the command line input arguments supplied when
% the GUI is invoked
processUserInputs();
```

1. `processUserInputs` sequences through the inputs, if any, and tries to match each property name to a string in the first column of the `mPropertyDefs` cell array.
2. If it finds a match, `processUserInputs` assigns the value that was input for the property to its variable in the third column of the `mPropertyDefs` cell array.
3. `processUserInputs` then calls the helper function specified in the second column of the `mPropertyDefs` cell array to validate the value that was passed in for the property.

Retrieving Output on Return from a GUI

If you call `iconEditor` with an output argument, it returns a truecolor image as an n-by-m-by-3 array.

The data definition section of the M-file creates a cell array to hold the output:

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

Following the call to `uiwait`, which stops the sequential execution of the M-file, iconEdit assigns the constructed icon array, `mIconEdit`, to the cell array `mOutputArgs` and then assigns `mOutputArgs` to `varargout` to return the arguments.

```
mOutputArgs{} =mIconCData;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

This code is the last that iconEditor executes before returning. It executes only after clicking the **OK** or **Cancel** button triggers execution of `hOKButtonCallback` or `hCancelButtonCallback`, which call `uiresume` to resume execution.

Protecting a GUI from Inadvertent Access

The `prepareLayout` utility function protects the iconEditor from inadvertently being altered from the command line by setting the `HandleVisibility` properties of all the components. The iconEditor calls `prepareLayout` with the handle of the main figure, in the initialization section of the M-file.

```
% Make changes needed for proper look and feel and running on
% different platforms
prepareLayout(hMainFigure);
```

`prepareLayout` first uses `findall` to retrieve the handles of all objects contained in the figure. The list of retrieved handles includes the `colorPalette`, which is embedded in the `iconEditor`, and its children. The figure's handle is passed to `prepareLayout` as the input argument `topContainer`.

```
allObjects = findall(topContainer);
```

`prepareLayout` then sets the `HandleVisibility` properties of all those objects that have one to `Callback`.

```
% Make GUI objects available to callbacks so that they cannot
% be changed accidentally by other MATLAB commands
set(allObjects(isprop(allObjects,'HandleVisibility')),...
    'HandleVisibility','Callback');
```

Setting `HandleVisibility` to `Callback` causes the GUI handles to be visible from within callback routines or functions invoked by callback routines, but not from within functions invoked from the command line. This ensures that command-line users cannot inadvertently alter the GUI when it is the current figure.

Running a GUI on Multiple Platforms

The `prepareLayout` utility function sets various properties of all the GUI components to enable the GUI to retain the correct look and feel on multiple platforms. The iconEditor calls `prepareLayout` with the handle of the main figure, in the initialization section of the M-file.

```
% Make changes needed for proper look and feel and running on
% different platforms
prepareLayout(hMainFigure);
```

First, `prepareLayout` uses `findall` to retrieve the handles of all objects contained in the figure. The list of retrieved handles also includes the `colorPalette`, which is embedded in the `iconEditor`, and its children. The figure's handle is passed to `findall` as the input argument `topContainer`.

```
function prepareLayout(topContainer)
   ...
   allObjects = findall(topContainer);
```

Background Color. The default component background color is the standard system background color on which the GUI is running. This color varies on different computer systems, e.g., the standard shade of gray on the PC differs from that on UNIX, and may not match the default GUI background color.

The `prepareLayout` function sets the background color of the GUI to be the same as the default component background color. This provides a consistent look within the GUI, as well as with other application GUIs.

It first retrieves the default component background color from the root object. Then sets the GUI background color using the figure's `Color` property.

```
defaultColor = get(0,'defaultuicontrolbackgroundcolor');
if isa(handle(topContainer),'figure')

    ...

    % Make figure color match that of GUI objects
    set(topContainer, 'Color',defaultColor);
end
```

Selecting Units. The `prepareLayout` function decides what units to use based on the GUI's resizability. It uses `strcmpi` to determine the value of the GUI's Resize property. Depending on the outcome, it sets the `Units` properties of all the objects to either `Normalized` or `Characters`.

```
% Make the GUI run properly across multiple platforms by using
% the proper units
if strcmpi(get(topContainer, 'Resize'),'on')
    set(allObjects(isprop(allObjects,'Units')),...
        'Units','Normalized');
else
    set(allObjects(isprop(allObjects,'Units')),...
        'Units','Characters');
end
```

For a resizable figure, normalized units map the lower-left corner of the figure and of each component to (0,0) and the upper-right corner to (1.0,1.0). Because of this, component size is automatically adjusted to its parent's size when the GUI is displayed.

For a nonresizable figure, character units automatically adjusts the size and relative spacing of components as the GUI displays on different computers.

Character units are defined by characters from the default system font. The width of a character unit equals the width of the letter `x` in the system font. The height of a character unit is the distance between the baselines of two lines of text. Note that character units are not square.

Making a GUI Modal

iconEditor is a modal figure. Modal figures remain stacked above all normal figures and the MATLAB command window. This forces the user to respond without being able to interact with other windows. iconEditor makes the main figure modal by setting its `WindowStyle` property to `modal`.

```
hMainFigure = figure(...
 ...
              'WindowStyle','modal',...
```

See the Figure Properties in the MATLAB Function Reference documentation for more information about using the `WindowStyle` property.

Sharing Data Between Two GUIs

The iconEditor embeds a GUI, the colorPalette, to enable the user to select colors for the icon cells. The colorPalette returns the selected color to the iconEditor via a function handle.

The colorPalette GUI. Like the iconEditor, the colorPalette defines a cell array, `mOutputArgs`, to hold its output arguments.

```
mOutputArgs = {};  % Variable for storing output when GUI returns
```

Just before returning, `colorPalette` assigns `mOutputArgs` the function handle for its `getSelectedColor` helper function and then assigns `mOutputArgs` to `varargout` to return the arguments.

```
% Return user defined output if it is requested
mOutputArgs{1} =@getSelectedColor;
if nargout>0
    [varargout{1:nargout}] = mOutputArgs{:};
end
```

The iconEditor executes the colorPalette's `getSeclectedColor` function whenever it invokes the function that colorPalette returns to it.

```
function color = getSelectedColor
% function returns the currently selected color in this
% colorPlatte
    color = mSelectedColor;
```

The iconEditor GUI. The `iconEditor` function calls `colorPalette` only once and specifies its parent to be a panel in the iconEditor.

```
% Host the ColorPalette in the PaletteContainer and keep the
% function handle for getting its selected color for editing
% icon.
mGetColorFcn = colorPalette('parent', hPaletteContainer);
```

This call creates the colorPalette as a component of the iconEditor and then returns a function handle that `iconEditor` can call to get the currently selected color.

The iconEditor's `localEditColor` helper function calls `mGetColorFcn`, the function returned by `colorPalette`, to execute the colorPalette's `getSelectedColor` function.

```
function localEditColor
% helper function that changes the color of an icon data
% point to that of the currently selected color in
% colorPalette
    if mIsEditingIcon
        pt = get(hIconEditAxes,'currentpoint');
        x = ceil(pt(1,1));
        y = ceil(pt(1,2));
        color = mGetColorFcn();
        % update color of the selected block
        mIconCData(y, x,:) = color;
        localUpdateIconPlot();
    end
end
```

Achieving Proper Resize Behavior

The `prepareLayout` utility function sets the `Units` properties of all the GUI components to enable the GUI to resize correctly on multiple platforms. The iconEditor calls `prepareLayout` with the handle of the main figure, in the initialization section of the M-file.

```
prepareLayout(hMainFigure);
```

First, `prepareLayout` uses `findall` to retrieve the handles of all objects contained in the figure. The list of retrieved handles includes the `colorPalette`, which is embedded in the `iconEditor`, and its children. The figure's handle is passed to `findall` as the input argument `topContainer`.

```
function prepareLayout(topContainer)
   ...
   allObjects = findall(topContainer);
```

Then, `prepareLayout` uses `strcmpi` to determine if the GUI is resizable. Depending on the outcome, it sets the `Units` properties of all the objects to either `Normalized` or `Characters`.

```
if strcmpi(get(topContainer, 'Resize'),'on')
    set(allObjects(isprop(allObjects,'Units')),...
        'Units','Normalized');
else
    set(allObjects(isprop(allObjects,'Units')),...
        'Units','Characters');
end
```

Note The iconEditor is resizable because it accepts the default value, `on`, of the figure `Resize` property.

Resizable Figure. Normalized units map the lower-left corner of the figure and of each component to (0,0) and the upper-right corner to (1.0,1.0). Because of this, when the GUI is resized, component size is automatically changed relative its parent's size.

Nonresizable Figure. Character units automatically adjusts the size and relative spacing of components as the GUI displays on different computers.

Character units are defined by characters from the default system font. The width of a character unit equals the width of the letter `x` in the system font. The height of a character unit is the distance between the baselines of two lines of text. Note that character units are not square.

A

Examples

Use this list to find examples in the documentation.

Simple Examples (GUIDE)

Simple Examples (Programmatic)

Programming GUI Components (GUIDE)

Application-Defined Data (GUIDE)

Application Examples (GUIDE)

GUI Layout (Programmatic)

Programming GUI Components (Programmatic)

Application-Defined Data (Programmatic)

Application Examples (Programmatic)

Index

E

F

G

H

I

L

M

N

O

P

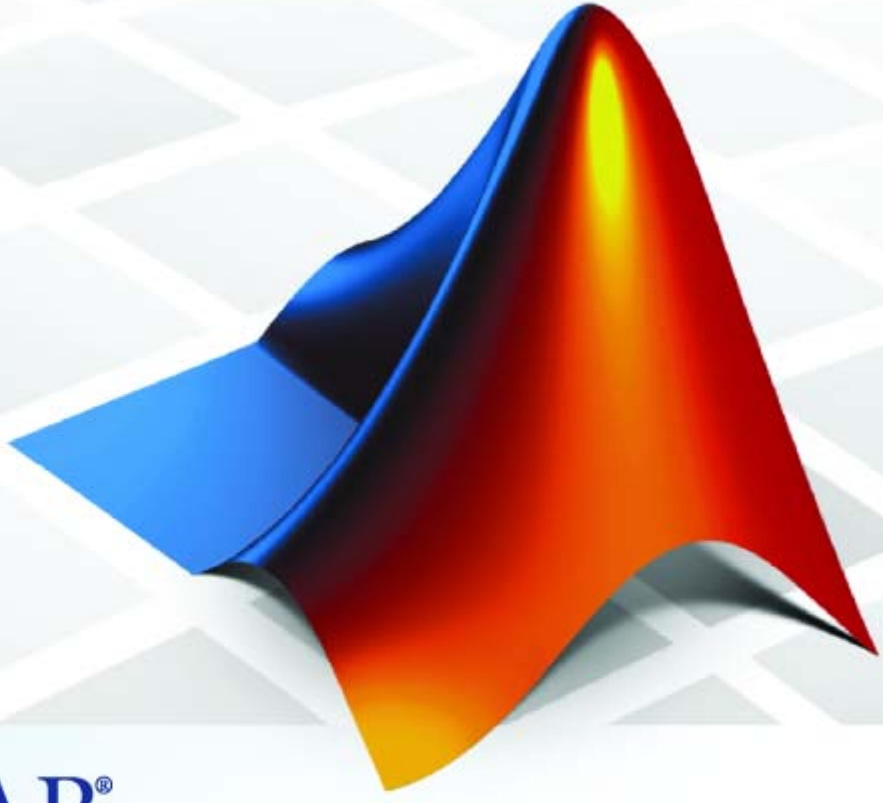
R

S

T

U

MATLAB® 7
Data Analysis



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Data Analysis

Revision History

September 2005	Online only	New for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for Version 7.2 (Release 2006a)
September 2006	Online only	Revised for Version 7.3 (Release 2006b)
March 2007	Online only	Revised for Version 7.4 (Release 2007a)
September 2007	Online only	Revised for Version 7.5 (Release 2007b)

Contents

1 Preparing Data for Analysis

2 Linear Regression Analysis

3 Fourier Analysis

4 Time Series Objects and Methods

5 Time Series Tools

1

Preparing Data for Analysis

The following sections summarize MATLAB® data-analysis capabilities, and provide information about preparing your data for analysis.

MATLAB for Data Analysis (p. 1-3)	Provides an overview of data analysis using MATLAB
Importing and Exporting Data (p. 1-7)	Explains where to get information about importing and exporting data
Plotting Data (p. 1-8)	Provides information about MATLAB plots, and includes an example of loading data from a text file and creating a time plot
Removing and Interpolating Missing Values (p. 1-11)	Describes using `NaN`s to represent missing data, as well as removing or interpolating these values
Removing Outliers (p. 1-14)	Describes how to identify and remove values that seem inconsistent with the majority of the data
Filtering Data (p. 1-16)	Describes how to smooth and shape data using filters
Detrending Data (p. 1-21)	Describes how to remove the mean or a best-fit line from the data

MATLAB for Data Analysis

In this section...

Introduction

MATLAB provides functions and GUIs to perform a variety of common data-analysis tasks, such as plotting data, computing descriptive statistics, and performing linear correlation analysis, data fitting, and Fourier analysis.

Typically, the first step to any data analysis is to plot the data. After examining the plot, you can determine which portions of the data to include in the analysis. You can also use the plot to evaluate if your data contains any features that might distort or confuse the analysis results, and then process your data to work only with the regions of interest.

This chapter describes the common techniques you can use to ready your data for analysis. When you work with empirical data, it is often necessary to treat it by doing the following:

- Removing or interpolating missing values. For more information, see "Removing and Interpolating Missing Values" on page 1-11.
- Removing outliers. For more information, see "Removing Outliers" on page 1-14.
- Smoothing the data using a first-order filter, a transfer function, or an ideal filter. For more information, see "Filtering Data" on page 1-16.
- Removing the mean or a linear trend (*detrending*). For more information, see "Detrending Data" on page 1-21.
- Differencing the data. For more information, see "Finite Differences" on page 1-25.

After isolating the data of interest, you can proceed with the core data-analysis tasks, which might include basic data fitting (see Chapter 2, "Linear Regression Analysis") and Fourier analysis (see Chapter 3, "Fourier Analysis"). If your data analysis requires more advanced or specialized functionality, see "Related Toolboxes" on page 1-5 to learn about the toolboxes available from The MathWorks.

If you are working with time series data, MATLAB provides the `timeseries` and `tscollection` objects and methods that enable you to efficiently represent and manipulate time series data. For more information about creating and working with these objects, see Chapter 4, "Time Series Objects and Methods". Alternatively, you can use the MATLAB Time Series Tools graphical user interface (GUI) to import, plot, and analyze time series. For more information, see Chapter 5, "Time Series Tools".

Calculations on Vectors and Matrices

Whereas some MATLAB functions support only vector inputs, others accept matrices.

When your data is a vector, the result is the same whether the vector has a rowwise or columnwise orientation.

However, when your data is a matrix, MATLAB performs calculations independently for each column. This means that when you pass a matrix as an argument to the function `max`, for example, the result is a row vector containing maximum data values for each column in the matrix.

Note When your data is a matrix where each row contains a data set, you must transpose the matrix before proceeding with the data-analysis tasks to make the data sets have a columnwise orientation. For example, to transpose a real matrix `A`, use the syntax `A'`.

MATLAB GUIs for Data Analysis

In addition to the various MATLAB functions for performing data analysis, MATLAB provides four graphical user interfaces (GUIs) that facilitate

common data-analysis tasks. The following table lists these GUIs and tells you how to get more information about each one.

MATLAB GUIs for Data Analysis

GUI	Description	More Information
MATLAB Figure window	For plotting variables in the MATLAB workspace and editing plot properties	MATLAB Graphics documentation
Data Statistics dialog box	For calculating and plotting descriptive statistics	"Example — Using MATLAB Data Statistics" on page 1-29
Basic Fitting dialog box	For basic data fitting using polynomial and spline models, as well as plotting fitted data and residuals	"Interactive Fitting" on page 2-9
Time Series Tools	For plotting and manipulating time series data	Chapter 5, "Time Series Tools"

Related Toolboxes

The following table summarizes the toolboxes that extend MATLAB data-analysis capabilities. For the latest information about these and other MathWorks products, point your Web browser to

www.mathworks.com

Toolboxes That Extend MATLAB Data Analysis

Toolbox	Description
Bioinformatics Toolbox	Import, analyze, and visualize genomic, proteomic, and microarray data.
Curve Fitting Toolbox	Interactively model one-dimensional data.
Financial Toolbox	Analyze financial data and develop financial algorithms.

Toolboxes That Extend MATLAB Data Analysis (Continued)

Toolbox	Description
Image Processing Toolbox	Perform image processing, analysis, and algorithm development.
Model-Based Calibration Toolbox	Calibrate complex powertrain systems.
Neural Network Toolbox	Design and simulate neural networks.
Optimization Toolbox	Fit nonlinear models to data.
Signal Processing Toolbox	Perform signal processing, analysis, and algorithm development.
Spline Toolbox	Create and manipulate spline approximation models of data.
Statistics Toolbox	Analyze and model data, simulate systems, and develop statistical algorithms.
System Identification Toolbox	Create linear dynamic models from measured input-output data.
Wavelet Toolbox	Analyze and synthesize signals and images using wavelet techniques.

Importing and Exporting Data

The first step in analyzing data is to import it into MATLAB. The MATLAB Programming documentation provides detailed information about supported data formats and the functions for bringing data into MATLAB.

The easiest way to import data into MATLAB is to use the MATLAB Import Wizard, as described in the MATLAB Programming documentation. With the Import Wizard, you can import the following types of data sources:

- Text files, such as `.txt` and `.dat`
- MAT-files
- Spreadsheet files, such as `.xls`
- Graphics files, such as `.gif` and `.jpg`
- Audio and video files, such as `.avi` and `.wav`

The MATLAB Import Wizard processes the data source and recognizes data delimiters, as well as row or column headers, to facilitate the process of data selection.

After you finish analyzing your data, you might have created new variables. You can export these variables to a variety of file formats. For more information about exporting data from the MATLAB workspace, see the MATLAB Programming documentation.

When working with time series data, it is easiest to use the Time Series Tools GUI to import the data and create `timeseries` objects. The Import Wizard in Time Series Tools also makes it easy to import or define a time vector for your data. For more information, see “Importing and Exporting Data” on page 5-7.

Plotting Data

In this section...

Introduction

After you import data into MATLAB, it is a good idea to plot the data so that you can explore its features. An exploratory plot of your data enables you to identify discontinuities and potential outliers, as well as the regions of interest.

The MATLAB Graphics documentation fully describes the MATLAB figure window, which displays the plot. It also discusses the various plot tools that are available in MATLAB to help you annotate and edit plot properties.

If you are working with time series data, see Chapter 5, "Time Series Tools", for detailed information about working with time series plots.

Example — Loading and Plotting Data

In this example, you perform the following tasks on the data in a space-delimited text file:

This example uses sample data in `count.dat` that consists of three sets of hourly traffic counts, recorded at three different town intersections over a 24-hour period. Each data column in the file represents data for one intersection.

Loading the Data

Import data into MATLAB using the `load` function:

```
load count.dat
```

Loading this data creates a 24-by-3 matrix called `count` in the MATLAB workspace.

You can get the size of the data matrix by

```
[n,p] = size(count)
n =
    24
p =
     3
```

where `n` represents the number of rows, and `p` represents the number of columns.

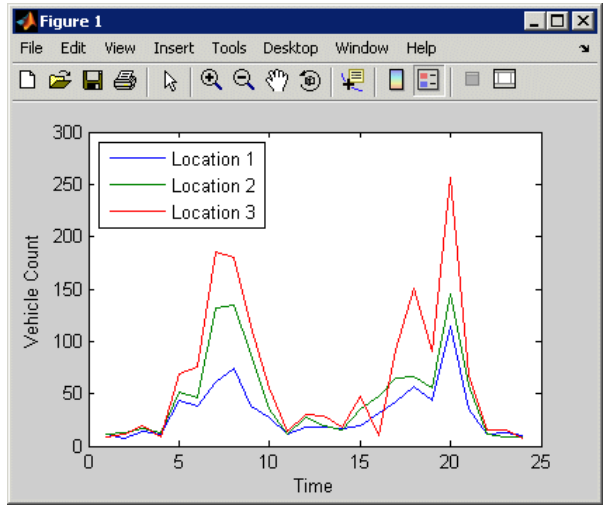
Plotting the Data

Create a time vector, `t`, containing integers from `1` to `n`:

```
t = 1:n;
```

Use the following commands to plot the data as a function of time, and to annotate the plot:

```
plot(t,count),
legend('Location 1','Location 2','Location 3',2)
xlabel('Time'), ylabel('Vehicle Count')
```



Traffic Counts at Three Intersections

Removing and Interpolating Missing Values

In this section...

Representing Missing Data Values

In MATLAB, missing or unavailable data values are represented by the special value `NaN`, which stands for *Not-a-Number*.

The IEEE floating-point arithmetic convention defines `NaN` as the result of an undefined operation, such as 0/0.

Calculating with NaNs

When you perform calculations on a MATLAB variable that contains `NaN`s, the `NaN` values are propagated to the final result. This might render the result useless.

For example, consider a matrix containing the 3-by-3 magic square with its center element replaced with `NaN`:

```
a = magic(3); a(2,2) = NaN

a =
     8     1     6
     3   NaN     7
     4     9     2
```

Compute the sum for each column in the matrix:

```
sum(a)

ans =
    15   NaN    15
```

Notice that the sum of the elements in the middle column is a `NaN` value because that column contains a `NaN`.

If you do not want to have `NaN`s in your final results, you must remove these values from your data. For more information, see "Removing NaNs from the Data" on page 1-12.

Removing NaNs from the Data

You can use the MATLAB function `isnan` to identify `NaN`s in the data, and then remove them using the techniques in the following table.

Note You must use the function `isnan` to identify `NaN`s because, by IEEE arithmetic convention, the logical comparison `NaN == NaN` always produces `0` (i.e., it never evaluates to `true`). Therefore, you *cannot* use `x(x==NaN) = []` to remove `NaN`s from your data.

Code	Description
`i = find(~isnan(x));` `x = x(i)`	Find the indices of elements in a vector `x` that are not `NaN`s. Keep only the non-`NaN` elements.
`x = x(~isnan(x));`	Remove `NaN`s from a vector `x`.
`x(isnan(x)) = [];`	Remove `NaN`s from a vector `x` (alternative method).
`X(any(isnan(X),2),:)  = [];`	Remove any rows containing `NaN`s from a matrix `X`.

If you frequently need to remove NaNs, you might want to write a short M-file function that you can call:

```
function X = exciseRows(X)
X(any(isnan(X),2),:) = [];
```

The following command computes the correlation coefficients of X after all rows containing NaNs are removed:

```
C = corrcoef(excise(X));
```

For more information about correlation coefficients, see “Correlation Analysis” on page 2-5.

Interpolating Missing Data

You can use interpolation to find intermediate points in your data. The simplest function for performing interpolation is `interp1`, which is a 1-D interpolation function.

By default, the interpolation method is `'linear'`, which fits a straight line between a pair of existing data points to calculate the intermediate value. The complete set of available methods, which you can specify as arguments in the `interp1` function, includes the following:

- `'nearest'` — Nearest neighbor interpolation
- `'linear'` — Linear interpolation
- `'spline'` — Piecewise cubic spline interpolation
- `'pchip'` or `'cubic'` — Shape-preserving piecewise cubic interpolation
- `'v5cubic'` — Cubic interpolation from MATLAB 5, which does not use `'extrapolate'` and uses `'spline'` when `X` is not equally spaced

For more information about `interp1`, see the MATLAB documentation or type at the MATLAB prompt

```
help interp1
```

Removing Outliers

When you examine a data plot, you might find that some points appear to dramatically differ from the rest of the data. In some cases, it is reasonable to consider such points *outliers*, or data values that do not appear to be consistent with the rest of the data.

The following example illustrates how to remove outliers from three data sets in the 24-by-3 matrix `count`. In this case, an outlier is defined as a value that is more than three standard deviations away from the mean.

Caution Be cautious about changing data unless you are confident that you understand the source of the problem you want to correct. Removing an outlier has a greater effect on the standard deviation than on the mean of the data. Deleting one such point leads to a smaller new standard deviation, which might result in making some remaining points appear to be outliers!

```
% Import the sample data
load count.dat;
% Calculate the mean and the standard deviation
% of each data column in the matrix
mu = mean(count)
sigma = std(count)
```

MATLAB displays

```
mu =
       32.0000   46.5417   65.5833

sigma =
       25.3703   41.4057   68.0281
```

When an *outlier* is considered to be more than three standard deviations away from the mean, you can use the following syntax to determine the number of outliers in each column of the `count` matrix:

```
[n,p] = size(count);
% Create a matrix of mean values by
% replicating the mu vector for n rows
MeanMat = repmat(mu,n,1);
% Create a matrix of standard deviation values by
% replicating the sigma vector for n rows
SigmaMat = repmat(sigma,n,1);
% Create a matrix of zeros and ones, where ones indicate
% the location of outliers
outliers = abs(count - MeanMat) > 3*SigmaMat;
% Calculate the number of outliers in each column
nout = sum(outliers)
```

MATLAB returns the following number of outliers in each column:

```
nout =
       1    0    0
```

There is one outlier in the first data column of `count` and none in the other two columns.

To remove an entire row of data containing the outlier, type

```
count(any(outliers,2),:) = [];
```

Here, `any(outliers,2)` returns a `1` when any of the elements in the `outliers` vector is a nonzero number, and the argument `2` specifies that `any` works down the second dimension of the count matrix—its columns.

Filtering Data

In this section...

Introduction

MATLAB provides functions for working with difference equations and filters to shape the variations in the raw data. These functions operate on both vectors and matrices. You can filter data to smooth out high-frequency fluctuations or remove periodic trends of a specific frequency.

A vector input represents a single, sampled data signal (or *sequence*). For a matrix input, each signal corresponds to a column in the matrix and each data sample is a row.

Filter Function

The function

```
y = filter(b,a,x)
```

creates filtered data `y` by processing the data in vector `x` with the filter described by vectors `a` and `b`.

The `filter` function is a general tapped delay-line filter, described by the difference equation

$$\begin{aligned} a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(N_b)x(n-N_b+1) \\ - a(2)y(n-1) - \ldots - a(N_a)y(n-N_a+1) \end{aligned}$$

Here, n is the index of the current sample, N_a is the order of the polynomial described by vector a, and N_b is the order of the polynomial described by

vector b. The output $y(n)$ is a linear combination of current and previous inputs, $x(n)$ $x(n-1)$..., and previous outputs, $y(n-1)$ $y(n-2)$... .

Example 1 — Moving Average Filter

You can smooth the data in `count.dat` using a moving-average filter to see the average traffic flow over a 4-hour window (covering the current hour and the previous 3 hours). This is represented by the following difference equation:

$$y(n) = \frac{1}{4}x(n) + \frac{1}{4}x(n-1) + \frac{1}{4}x(n-2) + \frac{1}{4}x(n-3)$$

The corresponding vectors are

```
a = 1;
b = [1/4 1/4 1/4 1/4];
```

Enter the following syntax to load the sample data:

```
load count.dat
```

This adds the matrix `count` to the workspace.

Extract the first column of `count` and assign it to the vector `x`:

```
x = count(:,1);
```

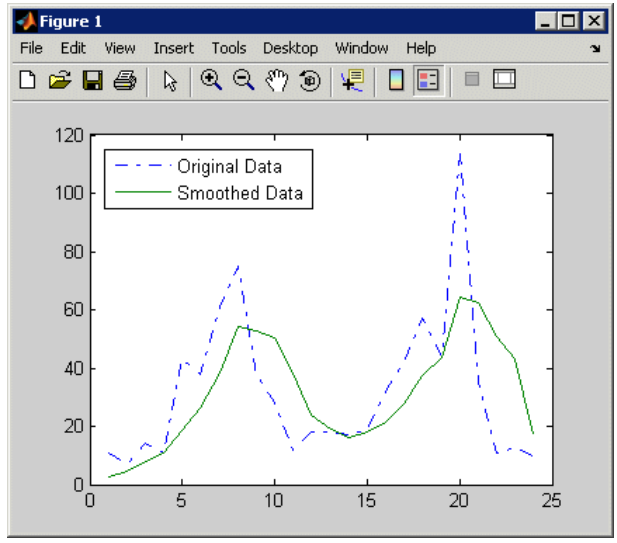
The 4-hour moving average of the data is calculated by

```
y = filter(b,a,x);
```

Compare the original data and the smoothed data with an overlaid plot of the two curves:

```
t = 1:length(x);
plot(t,x,'-.',t,y,'-'), grid on
legend('Original Data','Smoothed Data',2)
```

The filtered data, represented by the solid line in the plot, is the 4-hour moving average of the `count` data. The original data is represented by the dashed line.



Plot of Original and Smoothed Data

Example 2 — Discrete Filter

You use the discrete filter to shape the data by applying a transfer function to the input signal.

Depending on your objectives, the transfer function you choose might alter both the amplitude and the phase of the variations in the data at different frequencies to produce either a smoother or a rougher output.

Taking the z-transform of the following difference equation

$$a(1)y(n) = b(1)x(n) + b(2)x(n-1) + \ldots + b(N_b)x(n-N_b+1) - a(2)y(n-1) - \ldots - a(N_a)x(n-N_a+1)$$

results in the following transfer function:

$$Y(z) = H(z^{-1})X(z) = \frac{b(1) + b(2)z^{-1} + \ldots b(N_b)z^{-N_b+1}}{a(1) + a(2)z^{-1} + \ldots a(N_a)z^{-N_a+1}}X(z)$$

Here $Y(z)$ is the z-transform of the filtered output $y(n)$. The coefficients b and a are unchanged by the z-transform.

In digital signal processing (DSP), it is customary to write transfer functions as rational expressions in z^{-1} and to order the numerator and denominator terms in ascending powers of z^{-1}.

Consider the following transfer function:

$$H(z^{-1}) = \frac{b(z^{-1})}{a(z^{-1})} = \frac{2 + 3z^{-1}}{1 + 0.2z^{-1}}$$

To apply this transfer function to the data in `count.dat`:

1 Load the matrix count into the workspace:

```
load count.dat;
```

2 Extract the first column and assign it to `x`:

```
x = count(:,1);
```

3 Enter the coefficients of the denominator ordered in ascending powers of z^{-1} to represent $1 + 0.2z^{-1}$:

```
a = [1 0.2];
```

4 Enter the coefficients of the numerator to represent $2+2z^{-1}$:

```
b = [2 3];
```

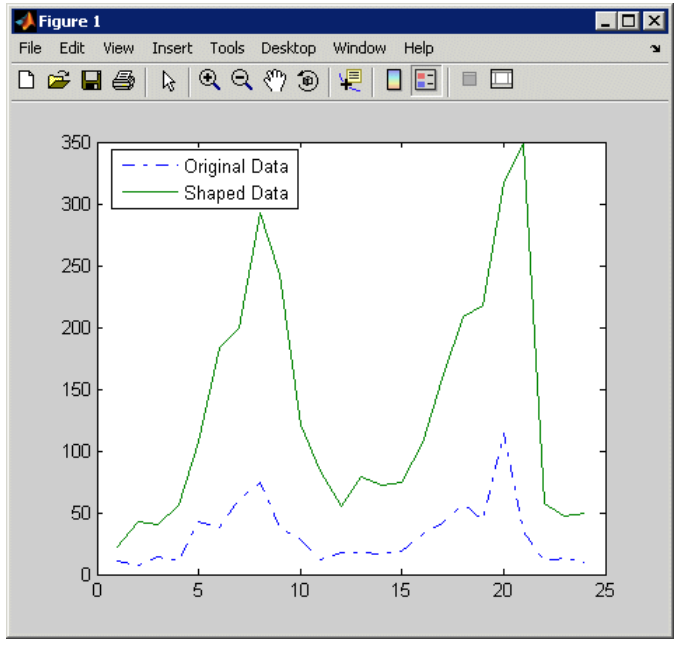
5 Call the filter function:

```
y = filter(b,a,x);
```

6 Compare the original data and the shaped data with an overlaid plot of the two curves:

```
t = 1:length(x);
plot(t,x,'-.',t,y,'-'), grid on
legend('Original Data','Shaped Data',2)
```

As you can see from the plot, this filter primarily modifies the amplitude of the original data.



Plot of Original and Shaped Data

Detrending Data

In this section...

Introduction

The MATLAB function `detrend` subtracts the mean or a best-fit line (in the least-squares sense) from your data. If your data contains several data columns, MATLAB detrends each data column separately.

Removing a trend from the data enables you to focus your analysis on the fluctuations in the data about the trend. A linear trend typically indicates a systematic increase or decrease in the data. This might be caused by sensor drift, for example.

You must decide whether it makes sense to remove trend effects in the data based on the objectives of your analysis.

Example — Removing Linear Trends from Data

This example shows how to remove a linear trend from daily closing stock prices to emphasize the price fluctuations about the overall increase. This data is available in the `predict_ret_data.mat` file.

You can follow along with the steps in this example to perform the following tasks:

- "Loading and Plotting Data" on page 1-22
- "Detrending Data and Plotting Results" on page 1-23

Loading and Plotting Data

1. Load the sample data:

   ```
   load predict_ret_data.mat
   ```

   This adds the variable `sdata` to the workspace, which contains the daily stock prices.

2. View the contents of the column vector `sdata`:

   ```
   sdata
   ```

   The last data value is a `NaN`, which must be removed before detrending the data.

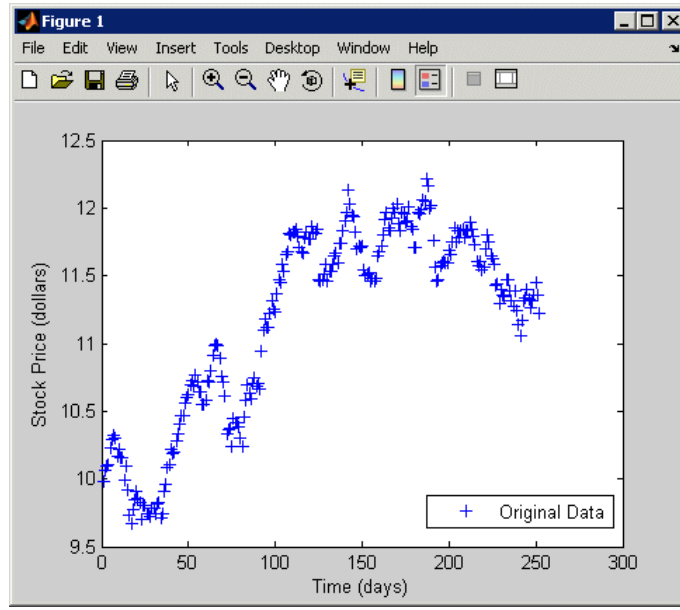
3. Identify and remove the `NaN` value from `sdata`:

   ```
   sdata(any(isnan(sdata),2),:) = []
   ```

   For more information about removing `NaN`s, see “Removing NaNs from the Data” on page 1-12.

4. Plot the data:

   ```
   plot(t, sdata,'+')
   legend('Original Data',1);
   xlabel('Time (days)');
   ylabel('Stock Price (dollars)');
   ```



Daily Closing Stock Prices

Notice the systematic increase in the stock prices when this data was collected.

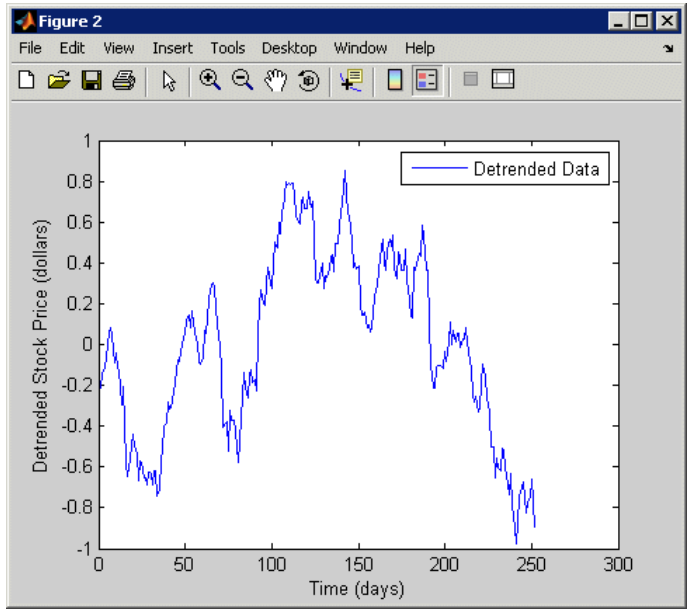
Detrending Data and Plotting Results

1. Remove a best-fit line (in the least-squares sense) from `sdata` and save the results to a new variable, `detrend_sdata`:

   ```
   detrend_sdata=detrend(sdata);
   ```

2. Plot the detrended data in a new MATLAB Figure window:

   ```
   figure
   plot(detrend_sdata,'-')
   legend('Detrended Data',2)
   xlabel('Time (days)');
   ylabel('Detrended Stock Price (dollars)');
   ```



Stock Prices with the Removed Linear Trend

Notice that the data is now centered about 0 and the linear drift is removed from the data.

Finite Differences

MATLAB provides three functions for finite difference calculations.

Function	Description
`del2`	Discrete Laplacian of a matrix
`diff`	Differences between successive elements of a vector; numerical partial derivatives of a vector
`gradient`	Numerical partial derivatives of a matrix

The `diff` function computes the difference between successive elements in a numeric vector. That is, `diff(X)` is `[X(2)-X(1) X(3)-X(2)...X(n)-X(n-1)]`. You might want to perform this operation on your data if you are more interested in analyzing the changes in the values, rather than the absolute values.

For a vector `A`,

```
A = [9 -2 3 0 1 5 4];
diff(A)

ans =
   -11     5    -3     1     4    -1
```

Besides computing the first difference, you can use `diff` to determine certain characteristics of vectors. For example, you can use `diff` to determine whether the vector values are monotonically increasing or decreasing, or whether a vector has equally spaced elements.

The following table provides examples for using `diff` with a vector `x`.

Test	Description
`any(diff(x)==0)`	Tests whether there are any repeated elements in X
`all(diff(x)>0)`	Tests whether the values are monotonically increasing
`all(diff(diff(x))==0)`	Tests for equally spaced vector elements

Descriptive Statistics

In this section...

If you need more advanced statistics functionality, you might want to use the Statistics Toolbox. For more information see the Statistics Toolbox documentation.

Functions for Calculating Descriptive Statistics

You can use the following MATLAB functions to calculate the descriptive statistics for your data.

Note For matrix data, MATLAB calculates descriptive statistics for each column independently.

Statistics Function Summary

Function	Description
`max`	Maximum value
`mean`	Average or mean value
`median`	Median value
`min`	Smallest value
`mode`	Most frequent value

Statistics Function Summary (Continued)

Function	Description
`std`	Standard deviation
`var`	Variance, which measures the spread or dispersion of the values

The following examples apply MATLAB functions to calculate descriptive statistics:

- "Example 1 — Calculating Maximum, Mean, and Standard Deviation" on page 1-27
- "Example 2 — Subtracting the Mean" on page 1-29

Example 1 — Calculating Maximum, Mean, and Standard Deviation

This example shows how to use MATLAB functions to calculate the maximum, mean, and standard deviation values for a 24-by-3 matrix called `count`. MATLAB computes these statistics independently for each column in the matrix.

```
% Load the sample data
load count.dat
% Find the maximum value in each column
mx = max(count)
% Calculate the mean of each column
mu = mean(count)
% Calculate the standard deviation of each column
sigma = std(count)
```

MATLAB responds with

```
mx =
          114           145           257

mu =
      32.0000       46.5417       65.5833

sigma =
      25.3703       41.4057       68.0281
```

To get the row numbers where the maximum data values occur in each data column, you can specify a second output parameter `indx` to return the row index. For example:

```
[mx,indx] = max(count)
```

MATLAB responds with this result:

```
mx =
      114     145     257

indx =
      20    20    20
```

Here, the variable `mx` is a row vector that contains the maximum value in each of the three data columns. The variable `indx` contains the row indices in each column that correspond to the maximum values.

To find the minimum value in the entire `count` matrix, you can reshape this 24-by-3 matrix into a 72-by-1 column vector by using the syntax `count(:)`. Then, to find the minimum value in the single column, you can use the following syntax:

```
min(count(:))

ans =
      7
```

Example 2 — Subtracting the Mean

You can subtract the mean from each column of the matrix by using the following syntax:

```
% Get the size of the count matrix
[n,p] = size(count)
% Compute the mean of each column
mu = mean(count)
% Create a matrix of mean values by
% replicating the mu vector for n rows
MeanMat = repmat(mu,n,1)
% Subtract the column mean from each element
% in that column
x = count - MeanMat
```

Note Subtracting the mean from the data is also called *detrending*. For more information about removing the mean or the best-fit line from the data, see "Detrending Data" on page 1-21.

Example — Using MATLAB Data Statistics

MATLAB provides the Data Statistics dialog box to help you calculate and plot descriptive statistics with the data. This example shows how to use MATLAB Data Statistics to calculate and plot statistics for a 24-by-3 matrix, called `count`.

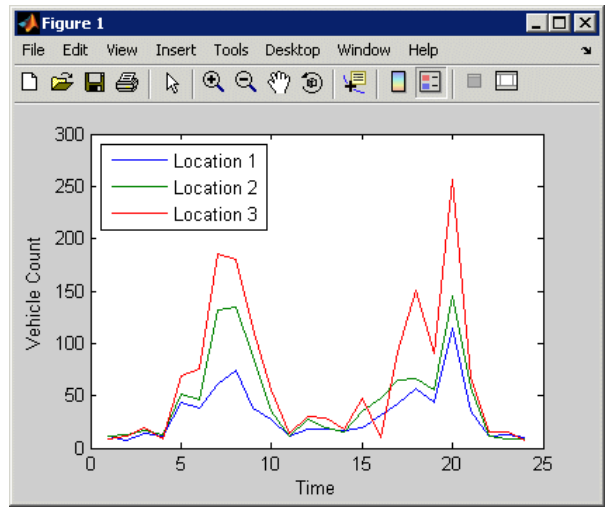
This section contains the following topics:

Note MATLAB Data Statistics is available only for 2-D plots.

Calculating and Plotting Descriptive Statistics

1 Load and plot the data:

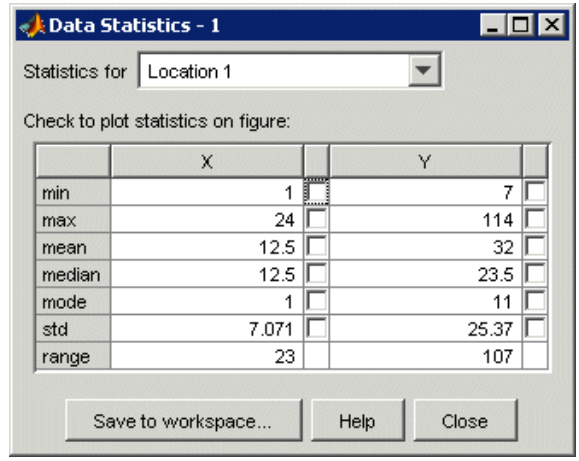
```
load count.dat
[n,p] = size(count);
% Define the x-values
t = 1:n;
% Plot the data and annotate the graph
plot(t,count)
legend('Location 1','Location 2','Location 3',2)
xlabel('Time'), ylabel('Vehicle Count')
```



Note The legend contains the name of each data set, as specified by the `legend` function: `Location 1`, `Location 2`, and `Location 3`. A *data set* refers to each column of data in the array you plotted. If you do not name the data sets, MATLAB assigns them default names: `data 1`, `data 2`, and so on.

2 In the Figure window, select **Tools > Data Statistics** .

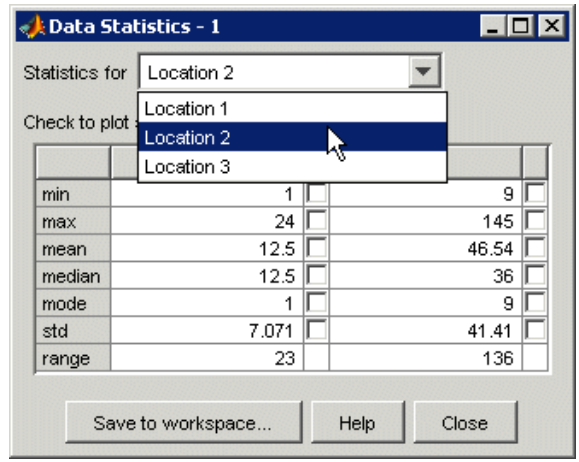
This opens the Data Statistics dialog box, which displays descriptive statistics for the `X`- and `Y`-data of the `Location 1` data set.



Note The Data Statistics GUI calculates the *range*, which is the difference between the minimum and maximum values in the selected data set. The Data Statistics GUI does not display the range on the plot.

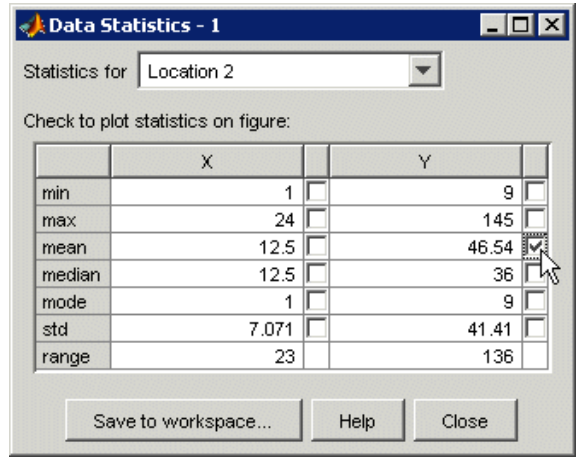
3 Select a different data set in the **Statistics for** list: `Location 2`.

This displays the statistics for the `X` and `Y` data of the `Location 2` data set.

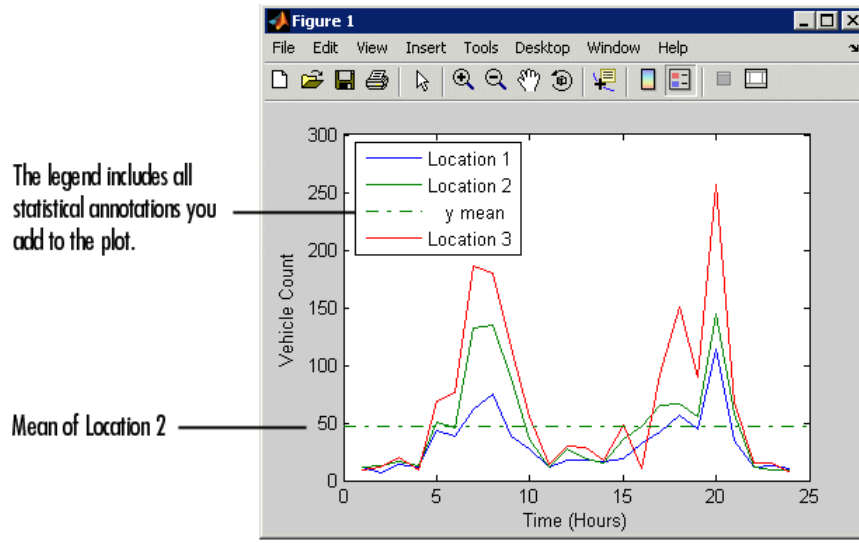


4 Select the check box for each statistic you want to display on the plot.

For example, to plot the mean of `Location 2`, select the **mean** check box in the **Y** column.



This plots a horizontal line to represent the mean of `Location 2` and updates the plot legend to include this statistic.



Formatting Data Statistics on Plots

The Data Statistics GUI uses colors and line styles to distinguish statistics from the data on the plot. This portion of the example shows how to customize the display of descriptive statistics on a plot, such as the color, line width, line style, or marker.

Note Do not edit display properties of statistics until you finish plotting all the statistics with the data. If you add or remove statistics after editing plot properties, the changes to plot properties are lost.

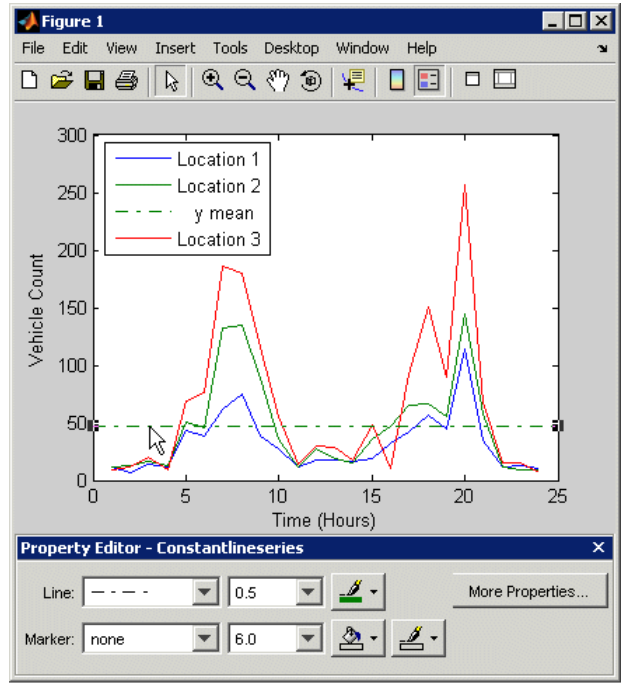
To modify the display of data statistics on a plot:

1. In the MATLAB Figure window, click the  (**Edit Plot**) button in the toolbar.

   This enables plot editing.

2 Double-click the statistic on the plot for which you want to edit display properties. For example, double-click the horizontal line representing the mean of `Location 2`.

This opens the Property Editor below the MATLAB Figure window, where you can modify the appearance of the line used to represent this statistic.



3 In the Property Editor, specify the **Line** and **Marker** styles, sizes, and colors.

Tip Alternatively, right-click the statistic on the plot, and select an option from the shortcut menu.

Saving Statistics to the MATLAB Workspace

This portion of the example shows how to save statistics in the Data Statistics GUI to the MATLAB workspace.

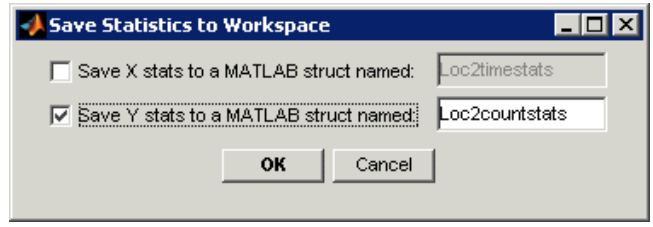
Note When your plot contains multiple data sets, you must save statistics for each data set individually. To display statistics for a different data set, select it from the **Statistics for** list in the Data Statistics GUI.

---

1. In the Data Statistics dialog box, click the **Save to workspace** button.

2. In the Save Statistics to Workspace dialog box, specify to save statistics for either `X` data, `Y` data, or both. Then, enter the corresponding variable names.

   In this example, save only the `Y` data. Enter the variable name as `Loc2countstats`.



3. Click **OK**.

   This saves the descriptive statistics to a structure. The new variable is added to the MATLAB workspace.

To view the new structure variable, type the variable name at the MATLAB prompt:

```
Loc2countstats

Loc2countstats =

       min: 9
       max: 145
      mean: 46.5417
    median: 36
      mode: 9
       std: 41.4057
     range: 136
```

Generating an M-file

This portion of the example shows how to generate an M-file that reproduces the format of the plot and the plotted statistics with new data.

1. In the Figure window, select **File > Generate M-File**.

   This creates a function M-file and displays it in the MATLAB Editor. The code in the M-file shows you how to programmatically reproduce what you did interactively with the Data Statistics GUI and the Property Editor.

2. Change the name of the function on the first line of the M-file from `createfigure` to something more specific, like `countplot`. Save the file to your current directory with the file name `countplot.m`.

3. Generate some new, random count data:

   ```
   randcount = 300*rand(24,3);
   ```

4. Reproduce the plot with the new data and the recomputed statistics:

   ```
   countplot(t,randcount)
   ```

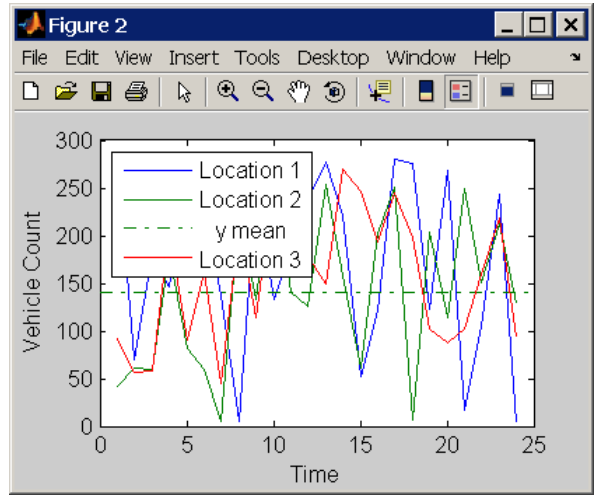


Figure 2
File Edit View Insert Tools Desktop Window Help
Location 1
Location 2
y mean
Location 3
Vehicle Count
Time
300
250
200
150
100
50
0
0
5
10
15
20
25

2

Linear Regression Analysis

Linear Regression

In this section...

Introduction

MATLAB allows you to model your data using linear regression. A *model* is a relationship between independent and dependent variables. Linear regression produces a model that is linear in the model coefficients. The most common type of linear regression is a *least-squares fit*, which can fit both lines and polynomials.

Before you model the relationship between pairs of quantities, it is a good idea to perform correlation analysis to establish if a relationship exists between these quantities. For more information, see "Correlation Analysis" on page 2-5.

MATLAB provides the Basic Fitting GUI for fitting your data, which enables you to calculate model coefficients and plot the model on top of the data. For an example of using this GUI, see "Example — Using Basic Fitting GUI" on page 2-11. You can also use the MATLAB functions `polyfit` and `polyval` to fit your data to a model that is linear in the coefficients. For an example of using these functions, see "Example — Data Fitting Using MATLAB Functions" on page 2-29.

If you need to fit nonlinear data using MATLAB, you can try transforming the variables in your model to make the model linear, use the nonlinear algorithm `fminsearch`, or use the Curve Fitting Toolbox (see the Curve Fitting Toolbox documentation).

In this chapter, you learn how to do the following:

- Use correlation analysis to determine whether two quantities are related to justify fitting the data.

- Fit a linear model to the data.
- Plot the model and the data on the same plot.
- Evaluate the goodness of fit using a plot of the residuals.

Residuals and Goodness of Fit

Residuals are defined as the difference between the *observed* values of the dependent variable and the values that are *predicted* by the model. When you fit a model that is appropriate for your data, the residuals approximate independent random errors.

To calculate fit parameters for a linear model, MATLAB minimizes the sum of the squares of the residuals to produce a good fit. This is called a least-squares fit.

You can gain insight into the "goodness" of a fit by visually examining a plot of the residuals: if the residual plot has a pattern, this indicates that the model does not properly fit the data.

Notice that the "goodness" of a fit must be determined in the context of your data. For example, if your goal of fitting the data is to extract coefficients that have physical meaning, then it is important that your model reflect the physics of the data. In this case, understanding what your data represents and how it was measured is just as important as evaluating the goodness of fit.

When to Use the Curve Fitting Toolbox

The Curve Fitting Toolbox extends core MATLAB functionality by enabling the following data-fitting capabilities:

- Linear and nonlinear parametric fitting, including standard linear least squares, nonlinear least squares, weighted least squares, constrained least squares, and robust fitting procedures
- Nonparametric fitting
- Statistics for determining the goodness of fit
- Extrapolation, differentiation, and integration
- GUI that facilitates data sectioning and smoothing

- Saving fit results in various formats, including M-files, MAT-files, and workspace variables

For more information, see the Curve Fitting Toolbox documentation.

Correlation Analysis

In this section...

Introduction

Before you fit a function to model the relationship between two measured quantities, it is a good idea to determine if a relationship exists between these quantities.

Correlation is a method for establishing the degree of probability that a linear relationship exists between two measured quantities. When there is no correlation between the two quantities, then there is no tendency for the values of one quantity to increase or decrease with the values of the second quantity.

MATLAB provides the following three functions for computing correlation coefficients and covariance. In typical data analysis applications, where you are mostly interested in the degree of relationship between variables, you need only to calculate correlation coefficients. That is, it is not necessary to calculate the covariance independently.

Function	Description
`corrcoef`	Correlation coefficient matrix
`cov`	Covariance matrix
`xcorr` (in Signal Processing Toolbox)	Cross-correlation sequence of a random process (includes autocorrelation)

Covariance

Use the MATLAB `cov` function to explicitly calculate the covariance matrix for a data matrix (where each column represents a separate quantity).

In typical data analysis applications, where you are mostly interested in the degree of relationship between variables, you can calculate the correlation coefficients directly without calculating the covariance first.

The covariance matrix has the following properties:

- `cov(X)` is symmetrical.
- `diag(cov(X))` is a vector of variances for each data column, which represent a measure of the spread or dispersion of data in the corresponding column.
- `sqrt(diag(cov(X)))` is a vector of standard deviations.
- The off-diagonal elements of the covariance matrix represent the covariance between the individual data columns.

Here, `X` can be a vector or a matrix. For an *m*-by-*n* matrix, the covariance matrix is *n*-by-*n*.

For an example of calculating the covariance, load the sample data in `count.dat` that contains a 24-by-3 matrix:

```
load count.dat
```

Calculate the covariance matrix for this data:

```
cov(count)
```

MATLAB responds with the following result:

```
ans =
    1.0e+003 *
       0.6437  0.9802  1.6567
       0.9802  1.7144  2.6908
       1.6567  2.6908  4.6278
```

The covariance matrix for this data has the following form:

$$\begin{bmatrix} \sigma^2_{11} & \sigma^2_{12} & \sigma^2_{13} \\ \sigma^2_{21} & \sigma^2_{22} & \sigma^2_{23} \\ \sigma^2_{31} & \sigma^2_{32} & \sigma^2_{33} \end{bmatrix}$$

$$\sigma^2 ij = \sigma^2_{ji}$$

Here, σ^2_{ij} is the covariance between column i and column j of the data. Because the `count` matrix contains three columns, the covariance matrix is 3-by-3.

Note In the special case when a vector is the argument of `cov`, the function returns the variance.

Correlation Coefficients

The correlation coefficient matrix represents the normalized measure of the strength of linear relationship between variables.

Correlation coefficients r_k are given by

$$r_k = \frac{\sum_{t=1}^{N} (x_t - \bar{x})(x_{t+k} - \bar{x})}{\sum_{t=1}^{N} (x_t - \bar{x})^2}$$

where x_t is a data value at time step t, k is the lag, and the overall mean is given by

$$\bar{x} = \sum_{t=1}^{N} \frac{x_t}{N}$$

The MATLAB function `corrcoef` produces a matrix of correlation coefficients for a data matrix (where each column represents a separate quantity). The correlation coefficients range from -1 to 1, where

- Values close to 1 suggest that there is a positive linear relationship between the data columns.
- Values close to -1 suggest that one column of data has a negative linear relationship to another column of data (*anticorrelation*).
- Values close to or equal to 0 suggest there is no linear relationship between the data columns.

For an *m*-by-*n* matrix, the correlation-coefficient matrix is *n*-by-*n*. The arrangement of the elements in the correlation coefficient matrix corresponds to the location of the elements in the covariance matrix, as described in "Covariance" on page 2-6.

For an example of calculating correlation coefficients, load the sample data in `count.dat` that contains a 24-by-3 matrix:

```
load count.dat
```

Type the following syntax to calculate the correlation coefficients:

```
corrcoef(count)
```

This results in the following 3-by-3 matrix of correlation coefficients:

```
ans =
    1.0000    0.9331    0.9599
    0.9331    1.0000    0.9553
    0.9599    0.9553    1.0000
```

Because all correlation coefficients are close to 1, there is a strong correlation between each pair of data columns in the `count` matrix.

Interactive Fitting

In this section...

The Basic Fitting GUI

The MATLAB Basic Fitting GUI allows you to interactively:

- Model data using a spline interpolant, a shape-preserving interpolant, or a polynomial up to the tenth degree
- Plot one or more fits together with data
- Plot the residuals of the fits
- Compute model coefficients
- Compute the norm of the residuals (a measure of the goodness of fit)
- Use the model to interpolate or extrapolate outside of the data
- Save coefficients and computed values to the MATLAB workspace for use outside of the GUI
- Generate an M-file to recompute fits and reproduce plots with new data

---

Note The Basic Fitting GUI is only available for 2-D plots. For more advanced fitting and regression analysis, see the Curve Fitting Toolbox and the Statistics Toolbox.

Preparing for Basic Fitting

The Basic Fitting GUI sorts your data in ascending order before fitting. If your data set is large and the values are not sorted in ascending order, it will take longer for the Basic Fitting GUI to preprocess your data before fitting.

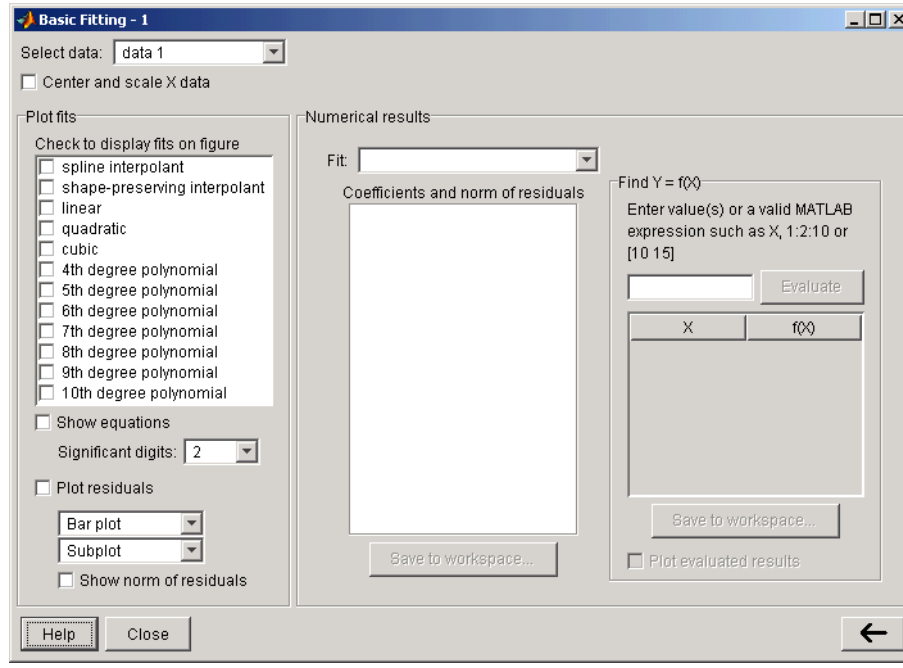
You can speed up the Basic Fitting GUI by first sorting your data in MATLAB. To create sorted vectors `x_sorted` and `y_sorted` from data vectors `x` and `y`, use the MATLAB `sort`function:

```
[x_sorted, i] = sort(x);
y_sorted = y(i);
```

Opening the Basic Fitting GUI

To use the Basic Fitting GUI, you must first plot your data in a figure window, using any MATLAB plotting command that produces (only) *x* and *y* data.

To open the Basic Fitting GUI, select **Tools > Basic Fitting** from the menus at the top of the figure window.



The GUI consists of three panels:

- For selecting a model and plotting options

- For examining and exporting model coefficients and norms of residuals
- For examining and exporting interpolated and extrapolated values

To expand or collapse the panels, use the arrow button in the lower right corner of the interface.

Example — Using Basic Fitting GUI

The example in this section shows you how to use the Basic Fitting GUI.

Loading and Plotting Data

The file `census.mat` contains U.S. population data for the years 1790 through 1990.

To load and plot the data, type the following commands at the MATLAB prompt:

```
load census
plot(cdate,pop,'ro')
```

The `load` command adds the following two variables to the MATLAB workspace:

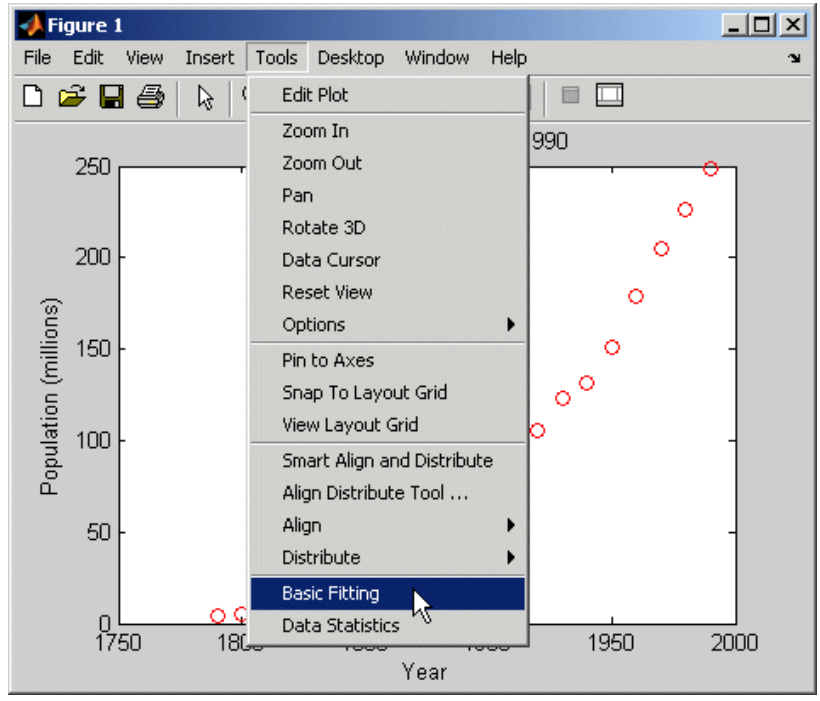
- `cdate` is a column vector containing the years from 1790 to 1990 in increments of 10. This is the predictor variable.
- `pop` is a column vector with U.S. population for each year in `cdate`. This is the response variable.

The data vectors are sorted in ascending order, by year. The plot shows the population as a function of year.

Now you are ready to fit the data.

Fitting Data

1 Open the Basic Fitting dialog box by selecting **Tools > Basic Fitting** in the Figure window.



2 In the **Plot fits** area of the Basic Fitting dialog box, select the **cubic** check box to fit a cubic polynomial to the data.

MATLAB displays the following warning:

```
Polynomial is badly conditioned. Removing
repeated data points or centering and scaling
may improve results.
```

The warning indicates that the computed coefficients for the model will be highly sensitive to random errors in the response (in this case, the measured population). To improve model accuracy, it is helpful to transform the predictors (in this case, the dates) by normalizing their center and scale. This is done by computing the *z-scores*:

$$z = \frac{x - \mu}{\sigma}$$

where x is the predictor data, μ is the mean of x, and σ is the standard deviation of x. This centers the data at 0, with a standard deviation of 1.

To perform this transformation on the predictor data, select the **Center and scale x data** check box.

After centering and scaling, model coefficients are computed for the y data as a function of z. These are different (and more robust) than the coefficients computed for y as a function of x. The form of the model, and the norm of the residuals, is unchanged. The Basic Fitting GUI automatically rescales the z-scores so that the fit is displayed on the same scale as the original x data.

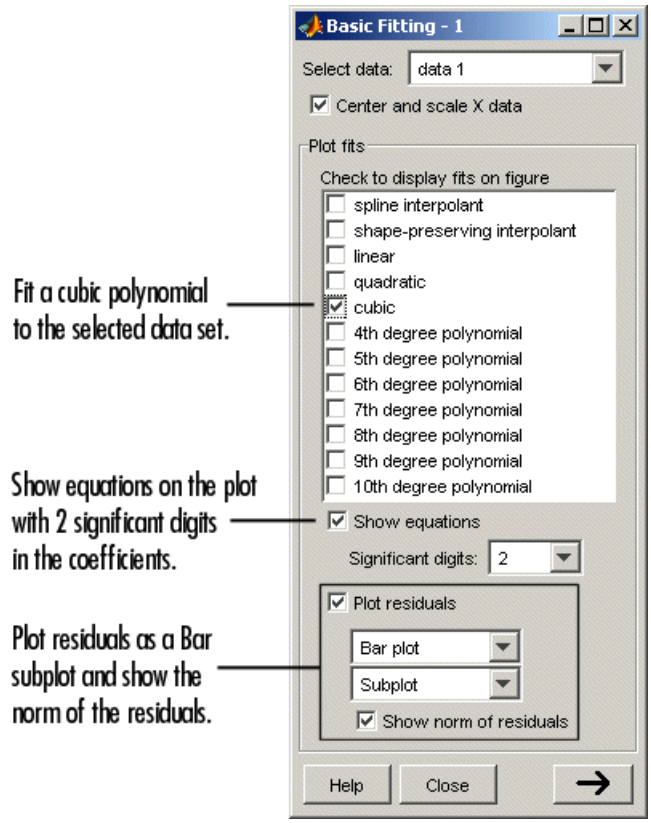
The Basic Fitting GUI calls the MATLAB functions `polyfit` and `polyval` to compute and display the fit. To understand the way in which the centered and scaled data is used as an intermediary to create the final plot, type the following at the MATLAB command prompt:

```
load census
x = cdate;
y = pop;
z = (x-mean(x))/std(x); % Compute z-scores of x data

plot(x,y,'ro') % Plot data
hold on

zfit = linspace(z(1),z(end),100);
pz = polyfit(z,y,3); % Compute conditioned fit
yfit = polyval(pz,zfit);

xfit = linspace(x(1),x(end),100);
```

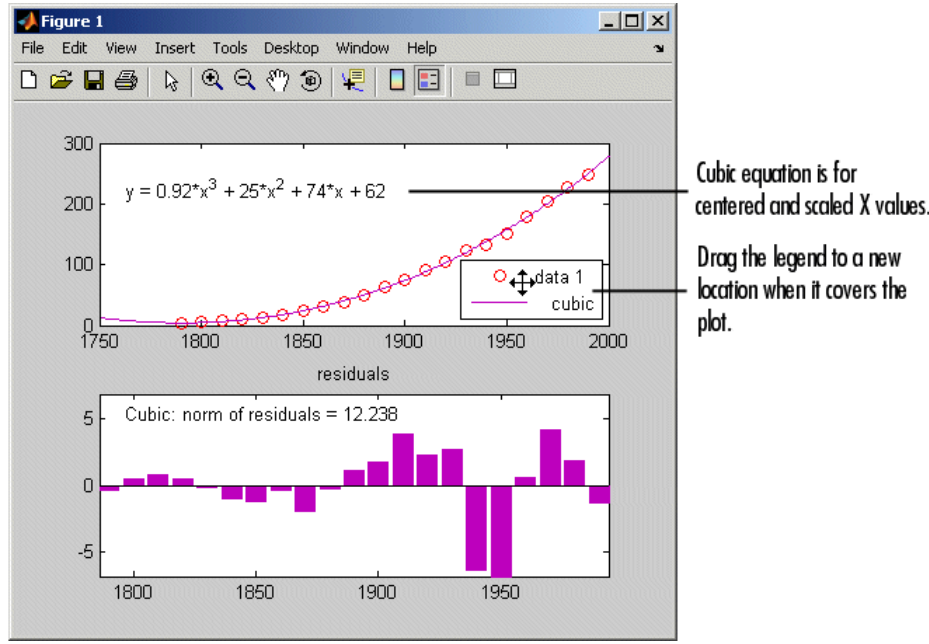
```
plot(xfit,yfit,'b-') % Plot conditioned fit vs. x data
```

3 Select the following options:

- Display the model equation in the plot
- Display the residuals as a subplot
- Display the norm of the residuals in the plot



The resulting display is shown in the following figure:



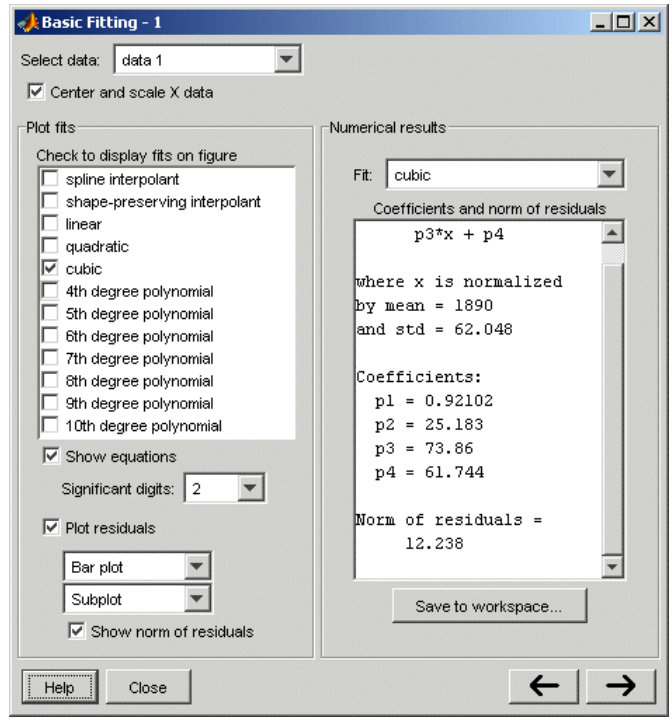
The cubic fit is a poor predictor before the year 1790, where it indicates a decreasing population. The model seems to approximate the data reasonably well after 1790, but a pattern in the residuals shows that the model does not meet the assumption of normal error, which is a basis for the least-squares fitting carried out by the Basic Fitting GUI.

For comparison, try fitting another equation to the census data by selecting it in the **Plot fits** area.

Tip You can change the default plot settings or rename data sets with the Property Editor.

Viewing and Saving Fit Parameters

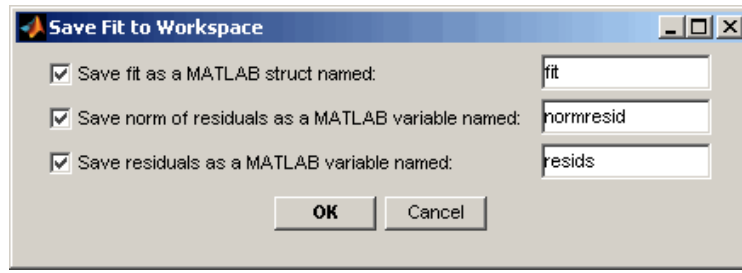
In the Basic Fitting dialog box, click the arrow button → to display the estimated coefficients and the norm of the residuals in the **Numerical results** panel.



To view a specific fit, select it from the **Fit** list. This displays the coefficients in the Basic Fitting dialog box, but does not plot the fit in the figure window.

Note If you also want to display a fit on the plot, you must select the corresponding **Plot fits** check box.

Save the fit data to the MATLAB workspace by clicking the **Save to workspace** button on the **Numerical results** panel. This opens the following dialog box:



Click **OK** to save the fit parameters as a MATLAB structure:

```
fit
fit =
     type: 'polynomial degree 3'
    coeff: [0.9210 25.1834 73.8598 61.7444]
```

You can now use the fit results in MATLAB programming, outside of the Basic Fitting GUI.

Interpolating and Extrapolating Values

Suppose you wish to use the cubic model to interpolate the U.S. population in 1965 (not in the original data).

In the Basic Fitting dialog box, click the → button to specify a vector of x values at which to evaluate the current fit.

1 In the **Enter value(s)...** field, type the following value:

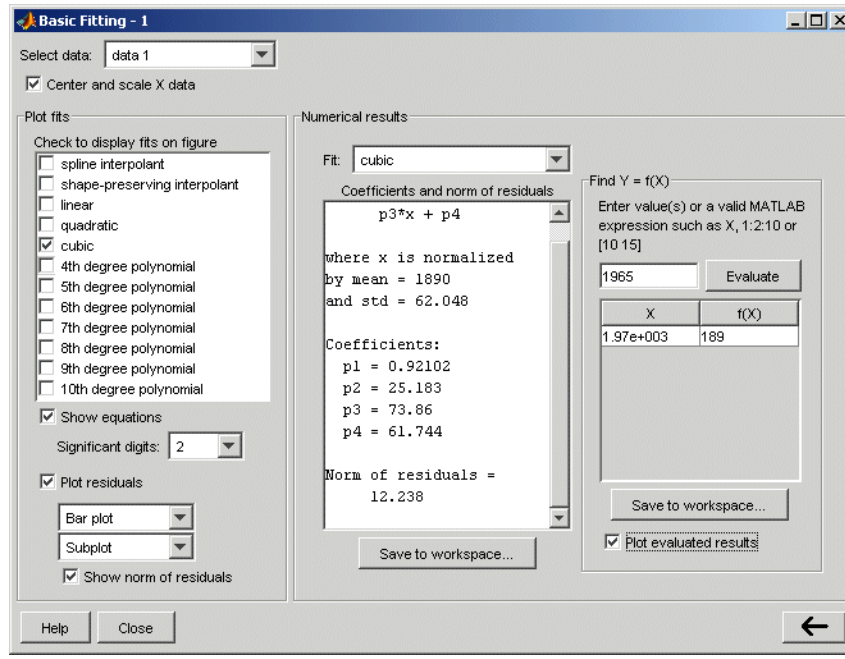
```
1965
```

Note Use unscaled and uncentered `x` values. You do not need to center and scale first, even though you selected to scale `x` values to obtain the coefficients in "Fitting Data" on page 2-12. Basic Fitting makes the necessary adjustments behind the scenes.

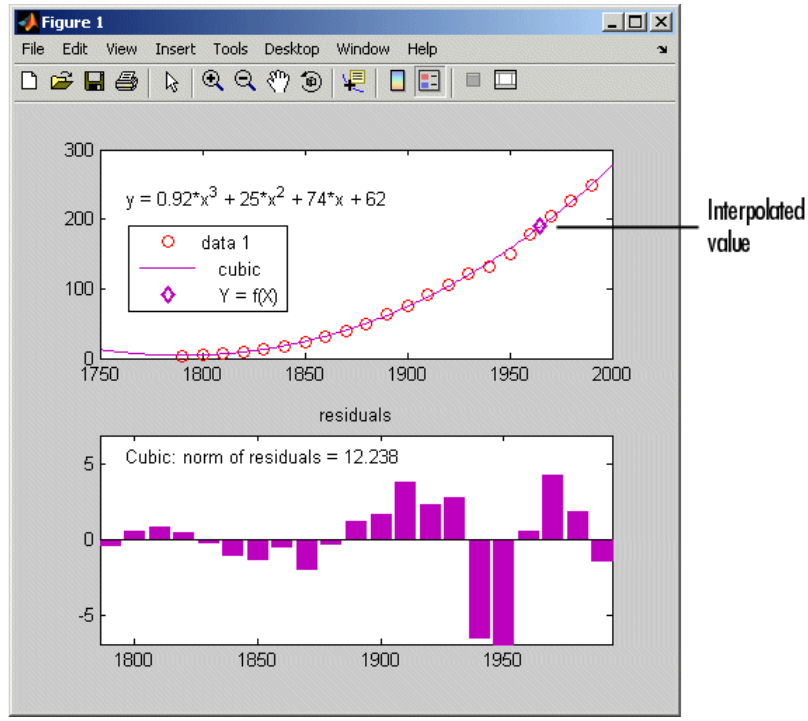
---

2 Click **Evaluate**.

The `x` values and the corresponding values for `f(x)` computed from the fit and displayed in a table, as shown below:

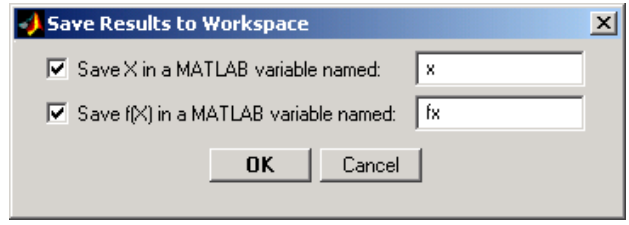


3 Select the **Plot evaluated results** check box to display the interpolated value:



4 Save the interpolated population in 1965 to the MATLAB workspace by clicking **Save to workspace**.

This opens the following dialog box, where you specify the variable names:



Generating an M-file

After completing a Basic Fitting session, you can generate an M-file that recomputes fits and reproduces plots with new data.

1. In the Figure window, select **File > Generate M-File**.

   This creates a function M-file and displays it in the MATLAB Editor. The code in the M-file shows you how to programmatically reproduce what you did interactively with the Basic Fitting dialog box.

2. Change the name of the function on the first line of the M-file from `createfigure` to something more specific, like `censusplot`. Save the file to your current directory with the file name `censusplot.m`

3. Generate some new, randomly perturbed census data:

   ```
   randpop = pop + 10*randn(size(pop));
   ```

4. Reproduce the plot with the new data and recompute the fit:

   ```
   censusplot(cdate,randpop,1965)
   ```

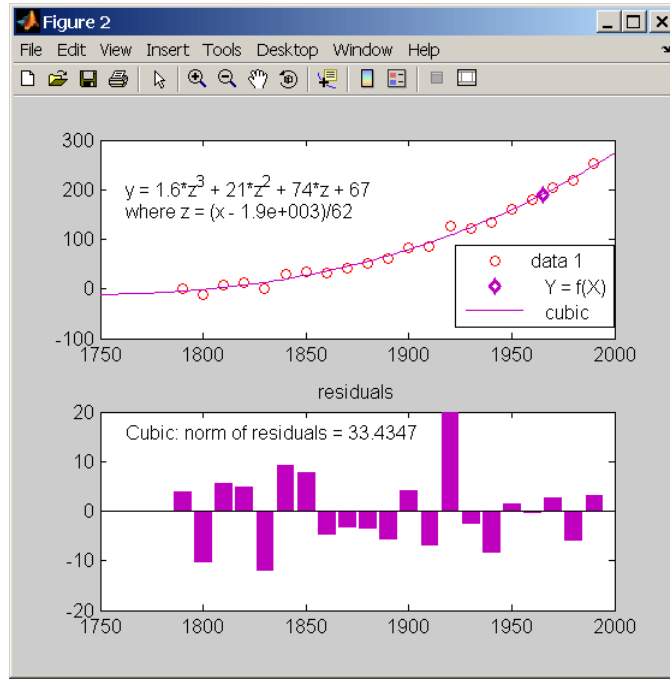


Figure 2
File Edit View Insert Tools Desktop Window Help
y = 1.6*z^3 + 21*z^2 + 74*z + 67
where z = (x - 1.9e+003)/62
data 1
Y = f(X)
cubic
300
200
100
0
-100
1750
1800
1850
1900
1950
2000
residuals
Cubic: norm of residuals = 33.4347
20
10
0
-10
-20
1750
1800
1850
1900
1950
2000

Programmatic Fitting

In this section...

MATLAB Functions for Polynomial Models

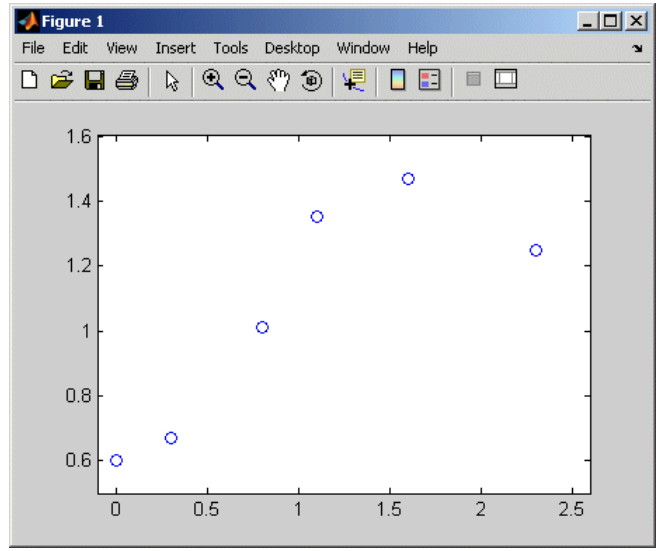
MATLAB provides two functions for modeling your data with a polynomial.

Polynomial Fit Functions

Function	Description
`polyfit`	`polyfit(x,y,n)` finds the coefficients of a polynomial `p(x)` of degree `n` that fits the `y` data by minimizing the sum of the squares of the deviations of the data from the model (least-squares fit).
`polyval`	`polyval(p,x)` returns the value of a polynomial of degree `n` that was determined by `polyfit`, evaluated at `x`.

For example, suppose you measure a quantity `y` at several values of time `t`:

```
t = [0 0.3 0.8 1.1 1.6 2.3];
y = [0.6 0.67 1.01 1.35 1.47 1.25];
plot(t,y,'o')
```



Plot of y Versus t

You can try modeling this data using a second-degree polynomial function:

$$y = a_2 t^2 + a_1 t + a_0$$

The unknown coefficients a_0, a_1, and a_2 are computed by minimizing the sum of the squares of the deviations of the data from the model (least-squares fit).

To find the polynomial coefficients, type the following at the MATLAB prompt:

```
p=polyfit(t,y,2)
```

MATLAB calculates the polynomial coefficients in descending powers:

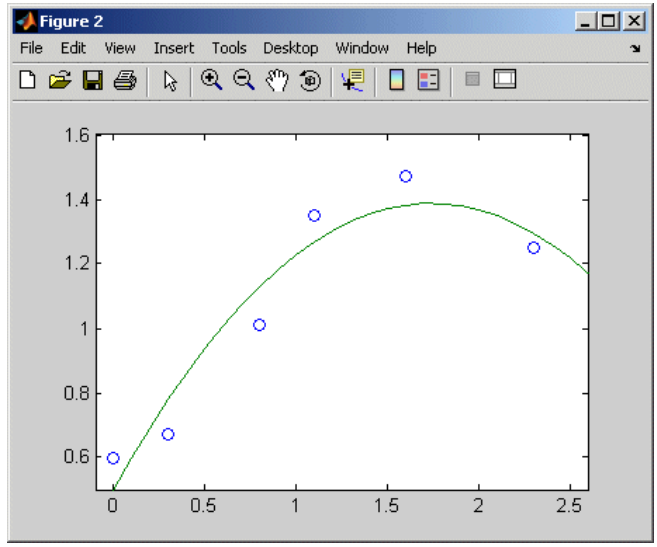
```
p =
   -0.2942    1.0231    0.4981
```

The second-degree polynomial model of the data is given by the following equation:

$$y = -0.2942t^2 + 1.0231t + 0.4981$$

To plot the model with the data, evaluate the polynomial at uniformly spaced times `t2` and overlay the original data on a plot:

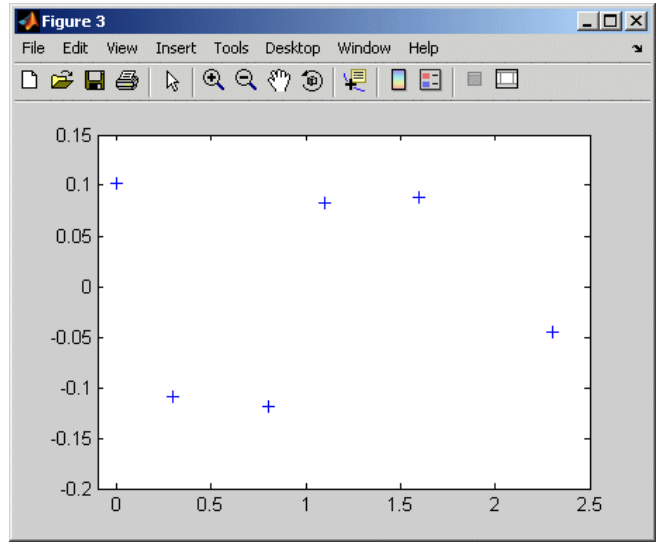
```
t2 = 0:0.1:2.8;     % Define a uniformly spaced time vector
y2=polyval(p,t2);   % Evaluate the polynomial at t2
figure
plot(t,y,'o',t2,y2) % Plot the fit on top of the data
                    % in a new Figure window
```



Plot of Data (Points) and Model (Line)

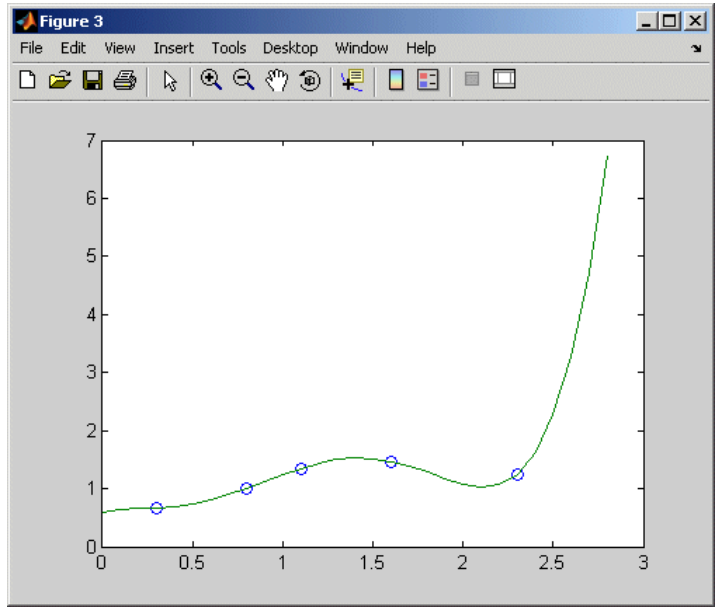
Use the following syntax to calculate the residuals:

```
y2=polyval(p,t); % Evaluate model at the data time vector
res=y-y2; % Calculate the residuals by subtracting
figure, plot(t,res,'+') % Plot the residuals
```



Plot of the Residuals

Notice that the second-degree fit roughly follows the basic shape of the data, but does not capture the smooth curve on which the data seems to lie. There appears to be a pattern in the residuals, which indicates that a different model might be necessary. A fifth-degree polynomial (shown next) does a better job of following the fluctuations in the data.



Fifth-Degree Polynomial Fit

Note If you are trying to model a physical situation, it is always important to consider whether a model of a specific order is meaningful in your situation.

Linear Model with Nonpolynomial Terms

When a polynomial function does not produce a satisfactory model of your data, you can try using a linear model with nonpolynomial terms. For example, consider the following function that is linear in the parameters a_0, a_1, and a_2, but nonlinear in the t data:

$$y = a_0 + a_1 e^{-t} + a_2 t e^{-t}$$

You can compute the unknown coefficients a_0, a_1, and a_2 by constructing and solving a set of simultaneous equations and solving for the parameters. The following syntax accomplishes this by forming a *design matrix*, where each column represents a variable used to predict the response (a term in the model) and each row corresponds to one observation of those variables:

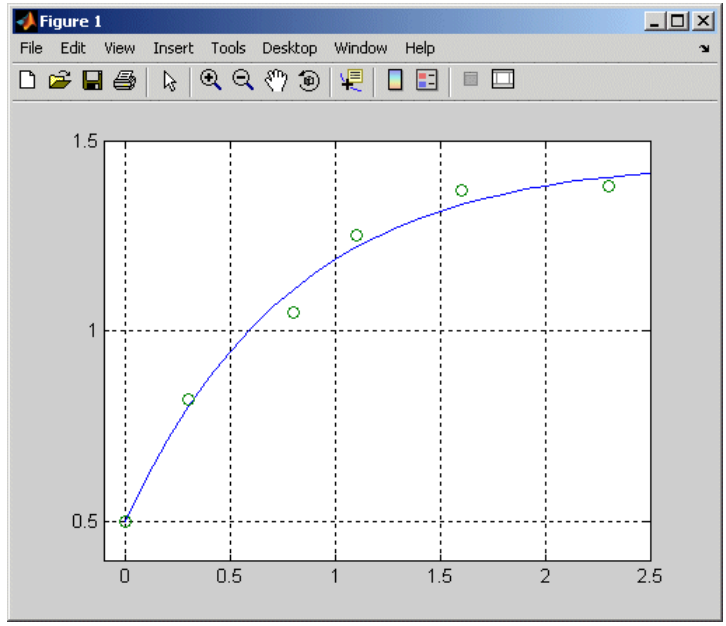
```
% Enter t and y as columnwise vectors
t = [0 0.3 0.8 1.1 1.6 2.3]';
y = [0.6 0.67 1.01 1.35 1.47 1.25]';

% Form the design matrix
X = [ones(size(t))  exp(-t)  t.*exp(-t)];

% Calculate model coefficients
a = X\y

a =
    1.3983
  - 0.8860
    0.3085
```

Therefore, the model of the data is given by

$$y = 1.3983 - 0.8860e^{-t} + 0.3085te^{-t}$$

Now evaluate the model at regularly spaced points and plot the model with the original data, as follows:

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



Linear Fit with Nonpolynomial Terms

Multiple Regression

When y is a function of more than one independent variable, the matrix equations that express the relationships among the variables must be expanded to accommodate the additional data. This is called *multiple regression*.

Suppose you measure a quantity y for several values of x_1 and x_2. Enter these variables in the MATLAB Command Window, as follows:

```
x1 = [.2 .5 .6 .8 1.0 1.1]';
x2 = [.1 .3 .4 .9 1.1 1.4]';
y  = [.17 .26 .28 .23 .27 .24]';
```

A model of this data is of the form

$$y = a_0 + a_1 x_1 + a_2 x_2$$

Multiple regression solves for unknown coefficients a_0, a_1, and a_2 by minimizing the sum of the squares of the deviations of the data from the model (least-squares fit).

Construct and solve the set of simultaneous equations by forming a design matrix, `X`, and solving for the parameters by using the backslash operator:

```
X = [ones(size(x1))  x1  x2];
a = X\y

a =
    0.1018
    0.4844
   -0.2847
```

The least-squares fit model of the data is

$$y = 0.1018 + 0.4844x_1 - 0.2847x_2$$

To validate the model, find the maximum of the absolute value of the deviation of the data from the model:

```
Y = X*a;
MaxErr = max(abs(Y - y))

MaxErr =
     0.0038
```

This value is much smaller than any of the data values, indicating that this model accurately follows the data.

Example — Data Fitting Using MATLAB Functions

In this example, you use MATLAB functions to accomplish the following:

This example uses the data in `census.mat`, which contains U.S. population data for the years 1790 to 1990.

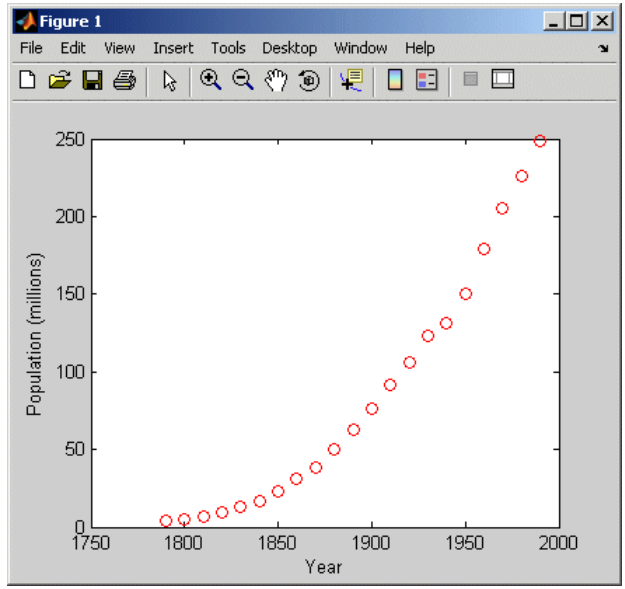
To load and plot the data, type the following commands at the MATLAB prompt:

```
load census
plot(cdate,pop,'ro')
```

This adds the following two variables to the MATLAB workspace:

- `cdate` is a column vector containing the years 1790 to 1990 in increments of 10.
- `pop` is a column vector with the U.S. population numbers corresponding to each year in `cdate`.

The following plot of the data shows a strong pattern, which indicates a high correlation between the variables.



U.S. Population from 1790 to 1990

Calculating Correlation Coefficients

In this portion of the example, you determine the statistical correlation between the variables `cdate` and `pop` to justify modeling the data. For more information about correlation coefficients, see “Correlation Analysis” on page 2-5.

Type the following syntax at the MATLAB prompt:

```
corrcoef(cdate,pop)
```

MATLAB calculates the following correlation-coefficient matrix:

```
ans =

    1.0000    0.9597
    0.9597    1.0000
```

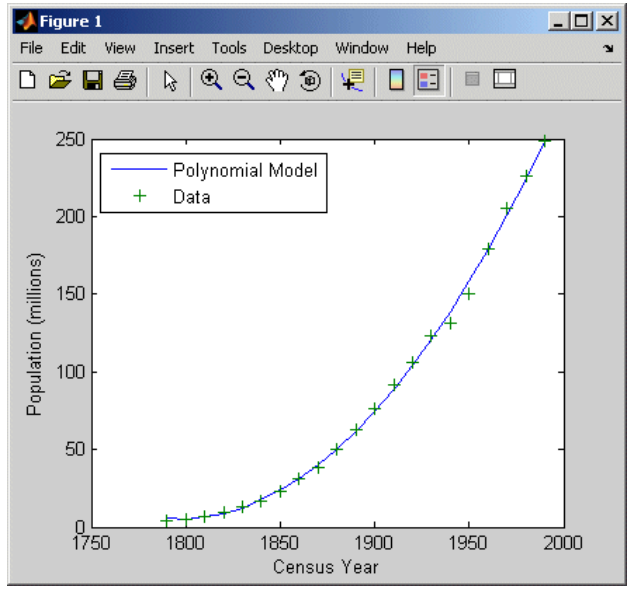
The diagonal matrix elements represent the perfect correlation of each variable with itself and are equal to 1. The off-diagonal elements are very close to 1, indicating that there is a strong statistical correlation between the variables `cdate` and `pop`.

Fitting a Polynomial to the Data

This portion of the example applies the `polyfit` and `polyval` MATLAB functions to model the data:

```
% Calculate fit parameters
[p,ErrorEst] = polyfit(cdate,pop,2);
% Evaluate the fit
pop_fit = polyval(p,cdate,ErrorEst);
% Plot the data and the fit
plot(cdate,pop_fit,'-',cdate,pop,'+');
% Annotate the plot
legend('Polynomial Model','Data');
xlabel('Census Year');
ylabel('Population (millions)');
```

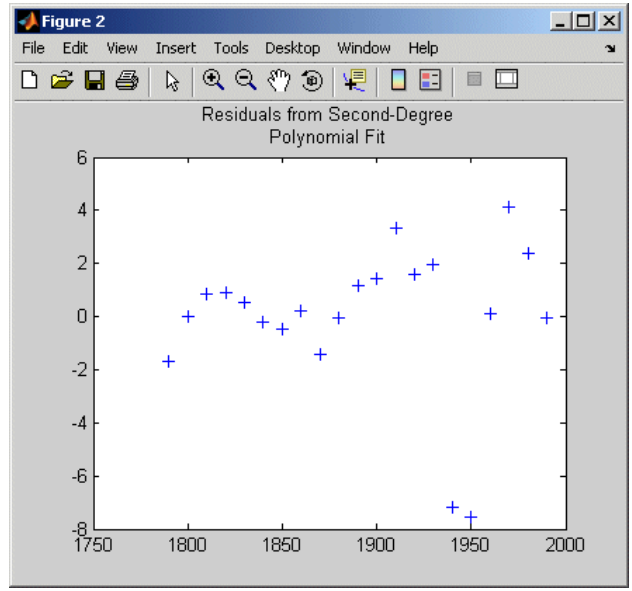
The following figure shows that the quadratic-polynomial fit provides a good approximation to the data:



Quadratic Polynomial Fit to the Census Data

To calculate the residuals for this fit, type the following syntax at the MATLAB prompt:

```
res = pop - pop_fit;
figure, plot(cdate,res,'+')
```



Residuals for the Quadratic Polynomial Model

Notice that the plot of the residuals exhibits a pattern, which indicates that a second-degree polynomial might not be appropriate for modeling this data.

Plot and Calculate Confidence Bounds

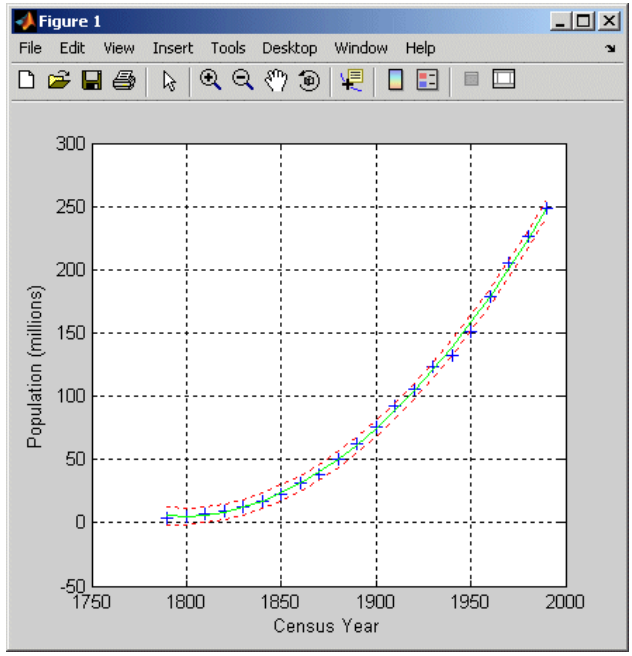
Confidence bounds are confidence intervals for a predicted response. The width of the interval indicates the degree of certainty of the fit.

This example applies `polyfit` and `polyval` to the `census` sample data to produce confidence bounds for a second-order polynomial model.

The following syntax uses an interval of $\pm 2\Delta$, which corresponds to a 95% confidence interval for large samples:

```
% Evaluate the fit and the prediction error estimate (delta)
[pop_fit,delta] = polyval(p,cdate,ErrorEst);
% Plot the data, the fit, and the confidence bounds
plot(cdate,pop,'+',...
     cdate,pop_fit,'g-',...
     cdate,pop_fit+2*delta,'r:',...
     cdate,pop_fit-2*delta,'r:');
% Annotate the plot
xlabel('Census Year');
ylabel('Population (millions)');
grid on
```

The 95% interval indicates that you have a 95% chance that a new observation will fall within the bounds.



Quadratic Polynomial Fit with Confidence Bounds

3

Fourier Analysis

The following sections describe how to perform Fourier analysis in MATLAB for gaining insight into periodic signals.

Introduction

Fourier analysis is particularly useful in areas such as signal and image processing, filtering, convolution, frequency analysis, and power spectrum estimation.

Fourier analysis provides insight into the periodicities in data by representing the data using a linear combination of sinusoidal components with different frequencies. The amplitude and phase of each sinusoidal component in the sum determines the relative contribution of that frequency component to the entire signal.

For discretely-sampled data, Fourier analysis is performed using the discrete Fourier transform (DFT). MATLAB calculates the DFT of a data sequence by applying the fast Fourier transform (FFT) algorithms; the FFT is an efficient computational method and not a different kind of transform.

To learn about more advanced power-spectrum methods, see the Signal Processing Toolbox documentation.

Function Summary

MATLAB provides the following functions for computing and working with Fourier transforms.

FFT Function Summary

Function	Description
`abs`	Absolute value and complex magnitude
`angle`	Phase angle
`cplxpair`	Sort numbers into complex conjugate pairs
`fft`	One-dimensional discrete Fourier transform, computed with a fast Fourier transform (FFT) algorithm
`fft2`	Two-dimensional discrete Fourier transform
`fftn`	N-dimensional discrete Fourier transform
`fftshift`	Shift DC component of the discrete Fourier transform to the center of spectrum
`ifft`	Inverse one-dimensional discrete Fourier transform
`ifft2`	Inverse two-dimensional discrete Fourier transform
`ifftn`	Inverse N-dimensional discrete Fourier transform
`ifftshift`	Inverse FFT shift
`nextpow2`	Next higher power of 2
`unwrap`	Unwrap phase angle in radians

Calculating Fourier Transforms

In this section...

Introduction

MATLAB performs Fourier analysis by computing the discrete Fourier transform (DFT) using the fast Fourier transform (FFT) algorithms, which improve computational performance.

Consider an input sequence $x(n)$ of length N. The DFT of this sequence is given by the vector $X(k)$, as follows:

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

You use the `fft` function in MATLAB to compute the spectrum. The length of $X(k)$ is the same as the length of $x(n)$. Notice that the value of $X(1)$ equals the sum of the data values in $x(n)$.

Note Traditional Fourier equations have summations from 0 to $N - 1$. However, because the first element of a MATLAB vector has the index of 1, the summations in the above equations are from 1 to N and are equivalent to traditional equations.

For a discrete input sequence, there is an upper limit on the frequency at which you can get meaningful information about the periodicities in the data. The highest frequency that can be uniquely fit to the data, called the *Nyquist frequency*, equals one cycle every two successive measurements. This makes sense because you cannot get information about the variations in the data at frequencies higher than the sampling rate—the rate at which you measured successive data values. For example, suppose that your data consists of daily temperature measurements in your town; here, 1 cycle (the time between

two successive measurements) equals 1 day and the Nyquist frequency is 0.5 cycle/day. In this situation, you can only determine variations in the temperature from one day to the next. If you want to study temperature fluctuations during the day, you must collect the data at more frequent intervals.

The lowest frequency that can be uniquely fit to the data, called the *fundamental frequency*, is one cycle for the entire length of the data vector.

The inverse DFT of a transformed sequence is given by:

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

You use the `ifft` function in MATLAB to synthesize the signal from its spectrum.

When $x(n)$ is real, you can rewrite the synthesis equation as a sum 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

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

Example — FFT of a Column Vector

Consider the following column vector:

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

In this example, the length of the input sequence $N = 8$. The Nyquist frequency is 1 cycle every 2 observations, or 0.5. The index of the component

k at the Nyquist frequency is determined by setting the frequency to the Nyquist frequency value:

$$f = \frac{\omega}{2\pi} = \frac{k-1}{N} = 0.5$$

Compute the FFT of `x` as follows:

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

MATLAB responds with the following FFT vector:

```
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
```

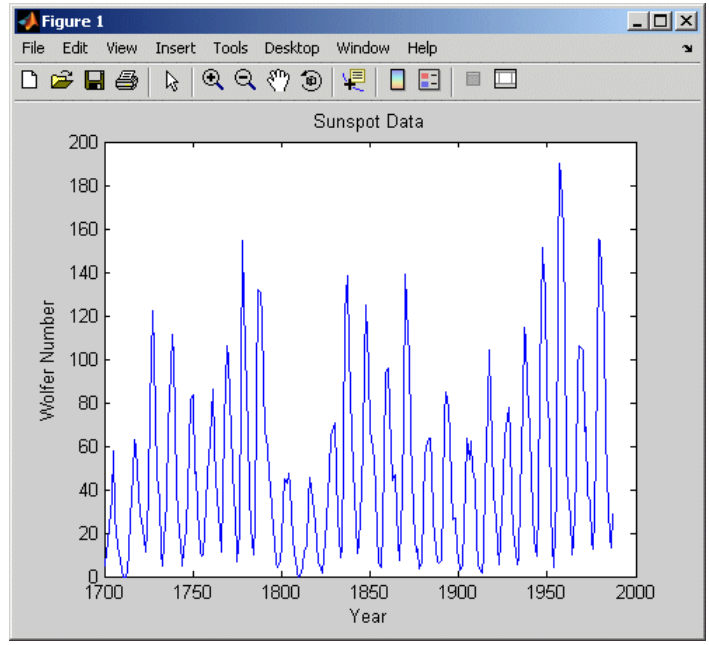
Notice that although the input sequence `x` is real, `y` is complex. The first element of `y` is the sum of the data values. The fifth element corresponds to the contribution at the Nyquist frequency. The last three values of `y` correspond to negative frequencies and, for the real sequence `x`, they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

Example — Sunspot Periodicity

In this example, you use the MATLAB `fft` function to analyze the variations in sunspot activity. You will use data collected by astronomers for almost 300 years of a quantity called the Wolfer number, which measures both the number and the size of sunspots.

Load and plot the sunspot data:

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



Sunspot Data

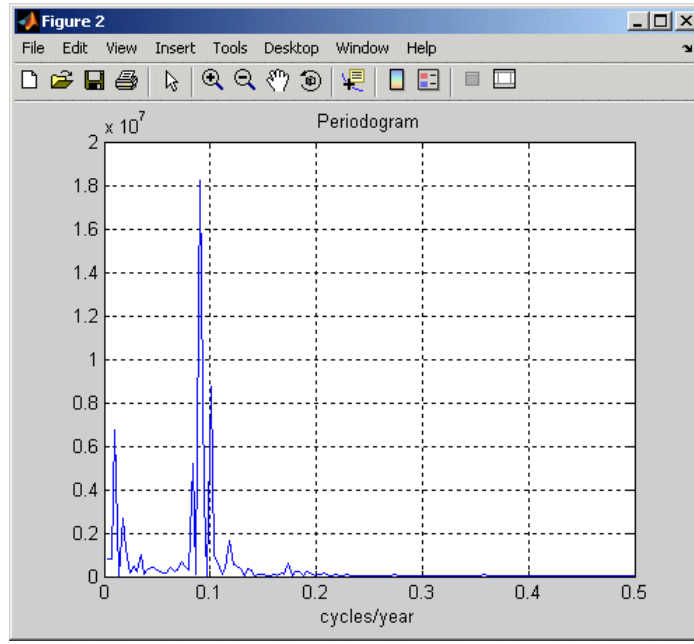
Take the FFT of the sunspot data:

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

The result of this transform is the complex vector `Y`. The magnitude of `Y` squared is called the estimated power spectrum. A plot of the estimated power spectrum versus frequency is called a *periodogram*.

Because the first component of `Y`, which is simply the sum of the data, has a large magnitude, the following syntax removes it before generating the periodogram:

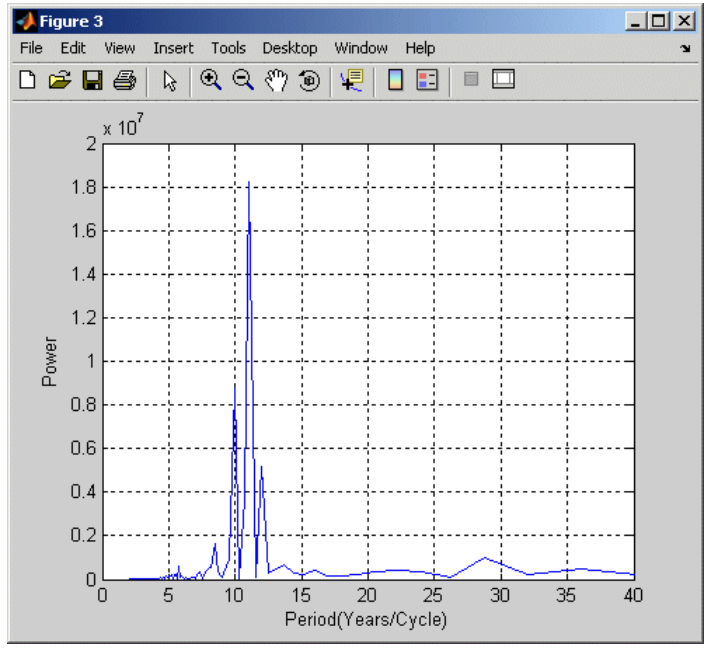
```
N = length(Y);
Y(1) = [];
power = abs(Y(1:N/2)).^2;
nyquist = 1/2;
freq = (1:N/2)/(N/2)*nyquist;
plot(freq,power), grid on
xlabel('cycles/year')
title('Periodogram')
```



Periodogram of Sunspot Data

The frequency scale is in cycles/year, which is inconvenient because for estimating the period of one cycle in years. Therefore, 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)')
```



Power Spectrum Versus Period of Sunspot Data

In order to determine the cycle more precisely, use the following syntax:

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

ans =
      11.0769
```

This plot confirms the cyclical nature of sunspot activity, which reaches a maximum about every 11 years.

Magnitude and Phase of Transformed Data

Important information about a transformed data 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);
```

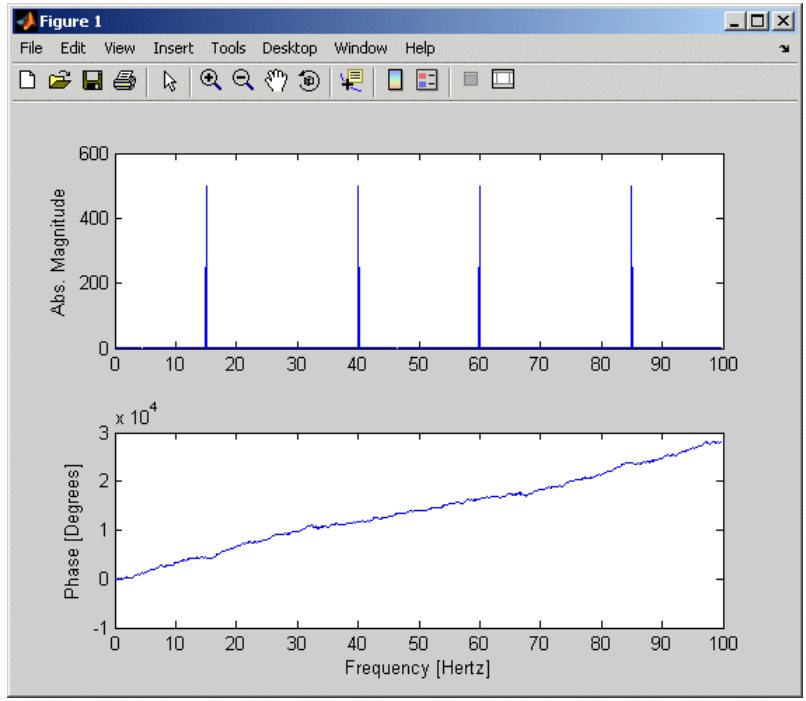
Now use the `fft` function to compute the DFT of the sequence. The following code calculates the magnitude and phase of the transformed sequence. It also uses the `abs` function to obtain the magnitude of the data, the `angle` function to obtain the phase information, and the `unwrap` function to remove phase jumps greater than π 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 Hz. The useful information in the signal is found in the range 0 to 50 Hz. For more information about the Nyquist frequency, see “Calculating Fourier Transforms” on page 3-4.



Magnitude and Phase Information in Transformed Data

FFT Length Versus Performance

The execution time for the `fft` depends on the length of the transform.

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

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

With this syntax, `fft` pads `x` with `0`s 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. `fft` is fastest for powers of 2. 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 have large prime factors.

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

4

Time Series Objects and Methods

The following sections describe how to analyze time series data using MATLAB objects and methods.

Introduction

MATLAB provides methods for analyzing time series data. These methods operate on the following MATLAB objects:

- `timeseries` — Stores data and time values, as well as the metadata information that includes units, events, data quality, and interpolation method
- `tscollection` — Stores a collection of `timeseries` objects that share a common time vector, convenient for performing operations on synchronized time series with different units

In this chapter, you learn how to

- Use time series constructors to instantiate time series classes
- Modify object properties using `set` methods or dot notation
- Call time series functions and methods

To get a quick overview of programming with `timeseries` and `tscollection` objects, follow the steps in "Example — Time Series Objects and Methods" on page 4-6.

If you prefer to work with a graphical user interface (GUI), use MATLAB Time Series Tools to work with time series data. For more information about Time Series Tools, see Chapter 5, "Time Series Tools".

Note If you are new to programming with `timeseries` and `tscollection` objects, you might want to start by working with Time Series Tools and enabling the **Record M-Code** feature. This generates reusable M-code based on the operations you perform in the GUI. For more information, see "Generating Reusable M-Code" on page 5-6.

Time Series Data Sample

To properly understand the description of `timeseries` object properties and methods in this documentation, it is important to clarify some terms related to storing data in a `timeseries` object—the difference between a *data value* and a *data sample*.

A *data value* is a single, scalar value recorded at a specific time. A *data sample* consists of one or more values associated with a specific time in the `timeseries` object. The number of data samples in a time series is the same as the length of the time vector.

For example, consider data that consists of three sensor signals: two signals represent the position of an object in meters, and the third represents its velocity in meters/second.

To enter the data matrix, type the following at the MATLAB prompt:

```
x = [-0.2 -0.3 13;
     -0.1 -0.4 15;
      NaN  2.8 17;
      0.5 0.3 NaN;
     -0.3 -0.1 15]
```

The `NaN` value represents a missing data value. MATLAB displays the following 5-by-3 matrix:

```
x=
   -0.2000   -0.3000   13.0000
   -0.1000   -0.4000   15.0000
    NaN       2.8000   17.0000
    0.5000    0.3000       NaN
   -0.3000   -0.1000   15.0000
```

The first two columns of `x` contain quantities with the same units and you can create a multivariate `timeseries` object to store these two time series. For more information about creating `timeseries` objects, see “Time Series Constructor Syntax” on page 4-22. The following command creates a `timeseries` object `ts_pos` to store the position values:

```
ts_pos = timeseries(x(:,1:2), 1:5, 'name', 'Position')
```

MATLAB responds by displaying the following properties of ts_pos:

```
Time Series Object: Position

Time vector characteristics

      Length                5
      Start time            1 seconds
      End time              5 seconds

Data characteristics

      Interpolation method  linear
      Size                  [5  2]
      Data type             double
```

The Length of the time vector, which is 5 in this example, equals the number of data samples in the timeseries object. Find the size of the data sample in ts_pos by typing the following at the MATLAB prompt:

```
getdatasamplesize(ts_pos)

ans =

     1     2
```

Similarly, you can create a second timeseries object to store the velocity data:

```
ts_vel = timeseries(x(:,3), 1:5, 'name', 'Velocity');
```

Find the size of each data sample in ts_vel by typing the following:

```
getdatasamplesize(ts_vel)

ans =

     1     1
```

Notice that ts_vel has one data value in each data sample and ts_pos has two data values in each data sample.

Note In general, when the time series data is an *M*-by-*N*-by-*P*-by-... multidimensional array with *M* samples, the size of each data sample is *N*-by-*P*-by-... .

If you want to perform operations on the `ts_pos` and `ts_vel` `timeseries` objects while keeping them synchronized, group them in a time series collection. For more information, see "Time Series Collection Constructor Syntax" on page 4-36.

Example — Time Series Objects and Methods

In this section...

Creating Time Series Objects

This portion of the example illustrates how to create several `timeseries` objects from an array. For more information about the `timeseries` object, see "Time Series Constructor" on page 4-21.

The sample data provided with this example consists of a 24-by-3 matrix of double values, where each column represents the hourly traffic counts at three town intersections.

This adds the variable `count` to the MATLAB workspace:

```
%% Import the sample data
load count.dat
```

To view the `count` matrix, type

```
count
```

MATLAB responds by displaying the following 24-by-3 matrix:

```
    11    11     9
     7    13    11
    14    17    20
    11    13     9
    43    51    69
    38    46    76
    61   132   186
    75   135   180
    38    88   115
    28    36    55
    12    12    14
    18    27    30
    18    19    29
    17    15    18
    19    36    48
    32    47    10
    42    65    92
    57    66   151
    44    55    90
   114   145   257
    35    58    68
    11    12    15
    13     9    15
    10     9     7
```

Create three `timeseries` objects to store the data collected at each intersection:

```
count1 = timeseries(count(:,1), 1:24,'name', 'intersection1');
count2 = timeseries(count(:,2), 1:24,'name', 'intersection2');
count3 = timeseries(count(:,3), 1:24,'name', 'intersection3');
```

Note In the above construction, `timeseries` objects have both a variable name (e.g., `count1`) and an internal object name (e.g., `intersection1`). The variable name is used with MATLAB functions. The object name is a property of the object, accessed with object methods. For more information on `timeseries` object properties and methods, see "Time Series Properties" on page 4-24 and "Time Series Methods" on page 4-31.

Each time series has a time vector in units of seconds, starting at 1 second and increasing up to 24 seconds in 1-second increments. The software assumes this increment when you do not explicitly specify one. You will change the time units to hours in "Modifying Time Series Units and Interpolation Method" on page 4-11.

Note If you want to create a `timeseries` object that groups the three data columns in `count`, use the following syntax:

```
count_ts = timeseries(count, 1:24,'name','traffic_counts')
```

This is useful when all time series have the same units and you want to keep them synchronized during calculations.

Viewing Time Series Objects

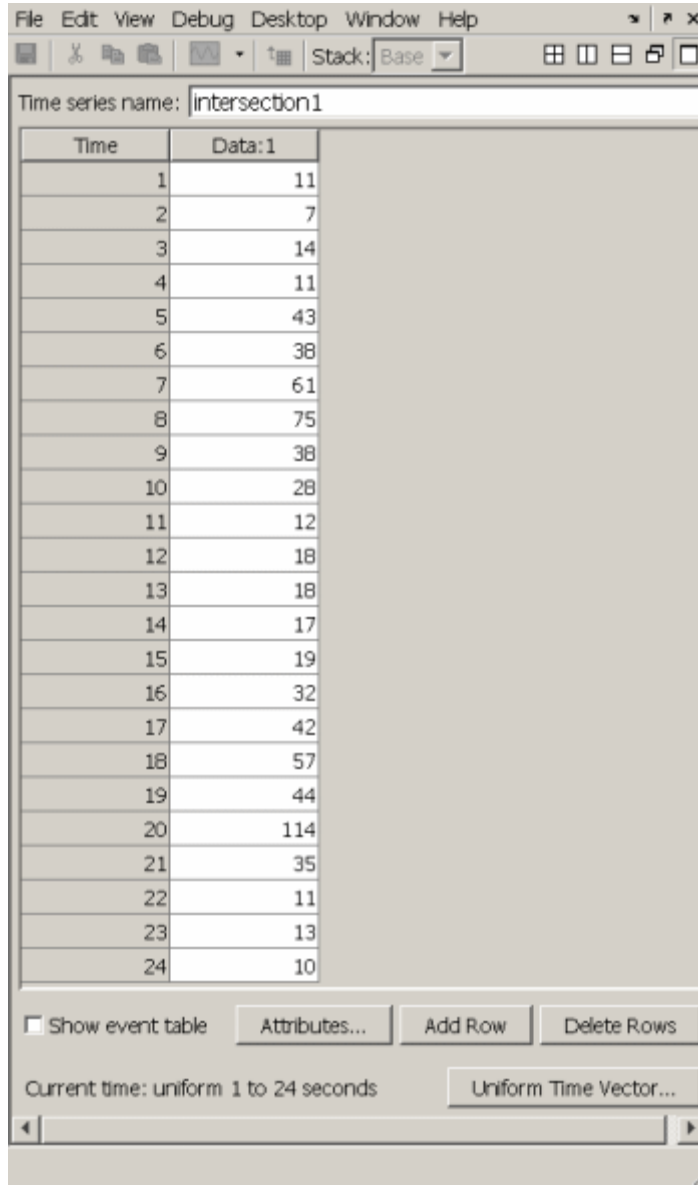
After creating a `timeseries` object, as described in "Creating Time Series Objects" on page 4-6, you can view it in either the Array Editor or Time Series Tools.

To view a `timeseries` object like `count1` in the Array Editor, use any one of several methods:

- Type `open('count1')` at the command prompt.
- Select `count1` in the Workspace Browser and click the **Open selection** button  .
- Double-click `count1` in the Workspace Browser.
- Right-click `count1` in the Workspace Browser and select **Open selection** from the context menu.

To view `count1` in Time Series Tools, right-click `count1` in the Workspace Browser and choose **Open in Time Series Tools** from the context menu.

When a `timeseries` object is opened in either the Array Editor or Time Series Tools, it is displayed with the Time Series Editor:



For information on using the Time Series Editor, see “Editing Data, Time, Attributes, and Events” on page 5-31.

Modifying Time Series Units and Interpolation Method

After creating a `timeseries` object, as described in "Creating Time Series Objects" on page 4-6, you can modify its units and interpolation method using dot notation.

To view the current properties of `count1`, type

```
get(count1)
```

MATLAB responds by displaying the current property values of the `count1` `timeseries` object:

```
Events: []
Name: 'intersection1'
Data: [24x1 double]
DataInfo: [1x1 tsdata.datametadata]
Time: [24x1 double]
TimeInfo: [1x1 tsdata.timemetadata]
Quality: []
QualityInfo: [1x1 tsdata.qualmetadata]
IsTimeFirst: true
TreatNaNasMissing: true
```

To view the current `DataInfo` properties, use dot notation:

```
count1.DataInfo
```

Change the data units and the default interpolation method for `count1`, as follows:

```
count1.DataInfo.Units = 'cars';
        % Specify new data units
count1.DataInfo.Interpolation = tsdata.interpolation('zoh');
        % Set the interpolation method to zero-order hold
```

To verify that the `DataInfo` properties have been modified, type

```
count1.DataInfo
```

MATLAB confirms the change by displaying

```
Time Series Data Meta Data Object
      Unit                 cars
      Interpolation Method  zoh
```

Modify the time units to be `'hours'` for the three time series:

```
count1.TimeInfo.Units = 'hours';
count2.TimeInfo.Units = 'hours';
count3.TimeInfo.Units = 'hours';
```

Defining Events

This portion of the example illustrates how to define events for a `timeseries` object by using the `tsdata.event` auxiliary object. Events mark the data at specific times. When you plot the data, event markers are displayed on the plot. Events also provide a convenient way to synchronize multiple time series.

Use the following syntax to add two events to the data that mark the times of the AM commute and PM commute:

```
%% Construct and add the first event to all time series
e1 = tsdata.event('AMCommute',8);
                          % Construct the first event at 8 AM
e1.Units = 'hours';       % Specify the time units of the time
count1 = addevent(count1,e1);  % Add the event to count1
count2 = addevent(count2,e1);  % Add the event to count2
count3 = addevent(count3,e1);  % Add the event to count3
%% Construct and add the second event to all time series
e2 = tsdata.event('PMCommute',18);
                          % Construct the first event at 6 PM
e2.Units = 'hours';       % Specify the time units of the time
count1 = addevent(count1,e2);  % Add the event to count1
count2 = addevent(count2,e2);  % Add the event to count2
count3 = addevent(count3,e2);  % Add the event to count3
```

Creating Time Series Collection Objects

This portion of the example illustrates how to create a `tscollection` object. Each individual time series in a collection is called a *member*. For more

information about the tscollection object, see “Time Series Collection Constructor” on page 4-36.

Note Typically, you use the tscollection object to group synchronized time series that have different units. In this simple example, all time series have the same units and the tscollection object does not provide an advantage over grouping the three time series in a single timeseries object. For an example of how to group several time series in one timeseries object, see “Creating Time Series Objects” on page 4-6.

Use the following syntax to create a tscollection object named count_coll and use the constructor syntax to immediately add two of the three time series currently in the MATLAB workspace (you will add the third time series later):

```
tsc = tscollection({count1 count2},'name', 'count_coll')
```

MATLAB responds with

```
Time Series Collection Object: count_coll
Time vector characteristics
Start time            1 hours
End time              24 hours
Member Time Series Objects:
      intersection1
      intersection2
```

Note The time vectors of the timeseries objects you are adding to the tscollection must match.

Notice that the Name property of the timeseries objects is used to name the collection members as intersection1 and intersection2.

Add the third timeseries object in the workspace to the tscollection by using the following syntax:

```
tsc = addts(tsc, count3)
```

MATLAB now lists all three members in the collection:

```
Time Series Collection Object: count_coll
Time vector characteristics
Start time            1 hours
End time              24 hours
Member Time Series Objects:
      intersection1
      intersection2
      intersection3
```

Resampling a Time Series Collection Object

This portion of the example illustrates how to resample each member in a `tscollection` using a new time vector. The resampling operation is used to either select existing data at specific time values, or to interpolate data at finer intervals. If the new time vector contains time values that did not exist in the previous time vector, the new data values are calculated using the default interpolation method you associated with the time series.

To resample the time series to include data values every 2 hours instead of every hour and save it as a new `tscollection` object, enter the following syntax:

```
tsc1 = resample(tsc,1:2:24)
```

In some cases you might need a finer sampling of information than you currently have and it is reasonable to obtain it by interpolating data values. For example, the following syntax interpolates values at each half-hour mark:

```
tsc1 = resample(tsc,1:0.5:24)
```

To add values at each half-hour mark, the default interpolation method of a time series is used. For example, the new data points in `intersection1` are calculated by using the zero-order hold interpolation method, which holds the value of the previous sample constant. You set the interpolation method for `intersection1` as described in “Modifying Time Series Units and Interpolation Method” on page 4-11.

The new data points in `intersection2` and `intersection3` are calculated using linear interpolation, which is the default method.

Adding a Data Sample to a Time Series Collection Object

This portion of the example illustrates how to add a data sample to a `tscollection`.

You can use the following syntax to add a data sample to the `intersection1` collection member at 3.25 hours (i.e., 15 minutes after the hour):

```
tsc1 = addsampletocollection(tsc1,'time',3.25,...
       'intersection1',5)
```

There are three members in the `tsc1` collection, and adding a data sample to one member adds a data sample to the other two members at 3.25 hours. However, because you did not specify the data values for `intersection2` and `intersection3` in the new sample, the missing values are represented by `NaN`s for these members. To learn how to remove or interpolate missing data values, see "Removing and Interpolating Missing Data" on page 4-16.

`tsc1` Data from 2.0 to 3.5 Hours

Hours	Intersection 1	Intersection 2	Intersection 3
2.0	7	13	11
2.5	7	15	15.5
3.0	14	17	20
3.25	5	`NaN`	`NaN`
3.5	14	15	14.5

To view all `intersection1` data (including the new sample at 3.25 hours), type

```
tsc1.intersection1
```

Similarly, to view all `intersection2` data (including the new sample at 3.25 hours containing a `NaN` value), type

```
tsc1.intersection2
```

Removing and Interpolating Missing Data

MATLAB uses NaNs to represent missing data in a time series. This portion of the example illustrates how to either remove the missing data or interpolate it by using the interpolation method you specified for that time series. In "Adding a Data Sample to a Time Series Collection Object" on page 4-15, you added a new data sample to the tsc1 collection at 3.25 hours.

There are three members in the tsc1 collection, and adding a data sample to one member adds a data sample to the other two members at 3.25 hours. However, because you did not specify the data values for the intersection2 and intersection3 members at 3.25 hours, they currently contain missing values that are represented by NaNs.

Removing Missing Data

You can use the following syntax to find and remove the data samples containing NaN values in the tsc1 collection:

```
tsc1 = delsamplefromcollection(tsc1,'index',...
       find(isnan(tsc1.intersection2.Data)));
```

This command searches one tscollection member at a time—in this case, intersection2. When a missing value is located in intersection2, the data at that time is removed from *all* members of the tscollection.

Note You can use the following dot-notation syntax to access the Data property of the intersection2 member in the tsc1 collection:

```
tsc1.intersection2.Data
```

For a complete list of timeseries properties, see "Time Series Properties" on page 4-24.

Interpolating Missing Data

For the sake of this example, you must reintroduce `NaN` values in `intersection2` and `intersection3` (which you removed):

```
tsc1 = addsampletocollection(tsc1,'time',3.25,...
       'intersection1',5);
```

To interpolate the missing values in `tsc1` using the current time vector (`tsc1.Time`), type the following syntax:

```
tsc1 = resample(tsc1,tsc1.Time)
```

This replaces the `NaN` values in `intersection2` and `intersection3` by using linear interpolation—the default interpolation method for these time series.

Note Dot notation `tsc1.Time` is used to access the `Time` property of the `tsc1` collection. For a complete list of `tscollection` properties, see "Time Series Collection Properties" on page 4-38.

To view `intersection2` data after interpolation, for example, type

```
tsc1.intersection2
```

New tsc1 Data from 2.0 to 3.5 Hours

Hours	Intersection 1	Intersection 2	Intersection 3
2.0	7	13	11
2.5	7	15	15.5
3.0	14	17	20

New tsc1 Data from 2.0 to 3.5 Hours (Continued)

Hours	Intersection 1	Intersection 2	Intersection 3
3.25	5	16	17.3
3.5	14	15	14.5

Removing a Time Series from a Time Series Collection

To remove the `intersection3` time series from the `tscollection` object `tsc1`, type:

```
tsc1 = removets(tsc1,'intersection3')
```

MATLAB now lists two time series as members in the collection:

```
Time Series Collection Object: count_coll
Time vector characteristics
Start time            1 hours
End time              24 hours
Member Time Series Objects:
      intersection1
      intersection2
```

Changing a Numerical Time Vector to Date Strings

This portion of the example illustrates how to convert the display format of a numerical time vector to MATLAB date strings. For a complete list of the MATLAB date-string formats supported for `timeseries` and `tscollection` objects, see “Time Vector Format” on page 4-21.

To convert a numerical time vector to date strings, you must set the `StartDate` field of the `TimeInfo` property. All values in the time vector are converted to date strings using `StartDate` as a reference date.

For example, suppose the reference date occurs on December 25, 2004:

```
tsc1.TimeInfo.StartDate = 'DEC-25-2004 00:00:00';
```

To verify that the time vector now uses date strings, type the following command to look at the sixth element of the `intersection2` member:

```
tsc1.intersection2(6)
```

MATLAB responds with

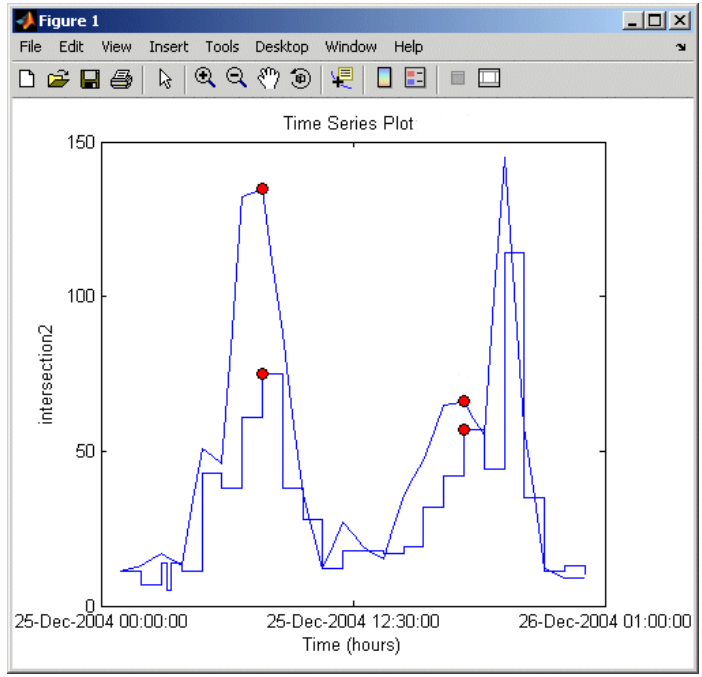
```
Time Series Object: unnamed
Time vector characteristics
      Length                1
      Start date            25-Dec-2004 03:15:00
      End date              25-Dec-2004 03:15:00
Data characteristics
      Interpolation method  linear
      Size                  [1  1]
      Data type             double
Time                              Data                  Quality
------------------------------------------------------------
25-Dec-2004 03:15:00              16
```

This result shows that the sixth element of `intersection2` has an interpolated data value of 16 cars at 3.25 hours (or 3:15:00).

Plotting Time Series Collection Members

You can plot the two remaining members in the `tsc1` collection by using the following command sequence:

```
plot(tsc1.intersection1); hold on;
plot(tsc1.intersection2)
```



Time Plot of Two Time Series in a Collection

This plot shows the two time series in the collection: `intersection1` and `intesection2`. `intersection1` uses the zero-order hold interpolation method and therefore has a jagged curve. In contrast, `intersection2` uses a linear interpolation method. The vertical axis is labeled as `intersection2` because this was the last time series plotted.

The filled circles on the plot indicate events, as specified in “Defining Events” on page 4-12.

Time Series Constructor

In this section...

Time Vector Format

You can specify the time vector of the `timeseries` object either as numerical (double) values or as valid MATLAB date strings.

When the `timeseries` `TimeInfo.StartDate` property is empty, the numerical `Time` values are measured relative to 0 (or another numerical value) in specified units. In this case, the time vector is described as *relative* (that is, it contains time values that are not associated with a specific start date).

When `TimeInfo.StartDate` is nonempty, the time values are date strings measured relative to `StartDate` in specified units. In this case, the time vector is described as *absolute* (that is, it contains time values that are associated with a specific calendar date). For more information, see "Time Series Properties" on page 4-24.

MATLAB supports the following date-string formats for time series applications.

Date-String Format	Usage Example
`dd-mmm-yyyy HH:MM:SS`	`01-Mar-2000 15:45:17`
`dd-mmm-yyyy`	`01-Mar-2000`
`mm/dd/yy`	`03/01/00`
`mm/dd`	`03/01`
`HH:MM:SS`	`15:45:17`
`HH:MM:SS PM`	`3:45:17 PM`
`HH:MM`	`15:45`

Date-String Format	Usage Example
HH:MM PM	3:45 PM
mmm.dd,yyyy HH:MM:SS	Mar.01,2000 15:45:17
mmm.dd,yyyy	Mar.01,2000
mm/dd/yyyy	03/01/2000

For an example of how to represent a numerical time vector relative to calendar dates, see "Changing a Numerical Time Vector to Date Strings" on page 4-18.

Time Series Constructor Syntax

Before implementing the various MATLAB functions and methods specifically designed to handle time series data, you must create a `timeseries` object to store the data.

The following table summarizes the syntax when using the `timeseries` constructor. For an example of using the constructor, see "Creating Time Series Objects" on page 4-6.

Time Series Syntax Descriptions

Syntax	Description
`ts = timeseries`	Creates an empty `timeseries` object. The size of this object is 0-by-1.
`ts = timeseries(Data)`	Creates a `timeseries` object with the specified `Data`. `ts` has a default time vector ranging from `0` to `N-1` with 1-second increments, where `N` is the number of samples. The default name of the `timeseries` object is `'unnamed'`.

Time Series Syntax Descriptions (Continued)

Syntax	Description
`ts = timeseries('Name')`	Creates an empty `timeseries` object with the name specified by a string `Name`. This name can differ from the `timeseries` variable name.
`ts = timeseries(Data,Time)`	Creates a `timeseries` object with the specified `Data` array and `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.

Time Series Syntax Descriptions (Continued)

Syntax	Description
`ts = timeseries(Data,Time,Quality)`	The `Quality` attribute is an integer vector containing values `-128` to `127` that specifies the quality in terms of codes defined by `QualityInfo.Code`. For more information about `QualityInfo`, see "Time Series Properties" on page 4-24.
`ts = timeseries(Data,..., 'Parameter',Value,...)`	Optionally enter the following parameter-value pairs after the `Data`, `Time`, and `Quality` arguments. You can specify the following parameters: • `Name` • `IsTimeFirst` • `IsDatenum` `Name` and `IsTimeFirst` are described in "Time Series Properties" on page 4-24. When set to `true`, `IsDatenum` specifies that `Time` values are dates in the format of MATLAB serial dates.

Time Series Properties

The following table lists the properties of the `timeseries` object. You can specify the `Data`, `IsTimeFirst`, `Name`, `Quality`, and `Time` properties as input arguments in the constructor. To assign other properties, use the `set` function or dot notation.

Note To get property information from the command line, type `help timeseries/tsprops` at the MATLAB prompt.

For an example of editing `timeseries` object properties, see "Modifying Time Series Units and Interpolation Method" on page 4-11.

Time Series Property Descriptions

Property	Description
`Data`	Time series data, where each data sample corresponds to a specific time. The data can be a scalar, a vector, or a multidimensional array. Either the first or last dimension of the data must align with `Time`. By default, `NaN`s represent missing or unspecified data. Set the `TreatNaNasMissing` property to determine how missing data is treated in calculations.

Time Series Property Descriptions (Continued)

Property	Description
`DataInfo`	Contains fields for storing contextual information about `Data`: • `Unit` — String that specifies data units. • `Interpolation` — A `tsdata.interpolation` object that specifies the interpolation method for this time series. Fields in the `tsdata.interpolation` object include: - `Fhandle`: Function handle to a user-defined interpolation function. - `Name`: String that specifies the name of the interpolation method. Predefined interpolation methods include `'linear'` and `'zoh'` (zero-order hold). `'linear'` is the default. • `UserData` — Any user-defined information entered as a string.

Time Series Property Descriptions (Continued)

Property	Description
`Events`	An array of `tsdata.event` objects that stores event information for this `timeseries` object. You add events using the `addevent` method. Fields in the `tsdata.event` object include the following: • `EventData` — Any user-defined information about the event • `Name` — String that specifies the name of the event • `Time` — Time value when this event occurs, specified as a real number or a date string relative to `StartDate` • `Units` — Time units • `StartDate` — A reference date specified in MATLAB date string format. `StartDate` is empty when you have a numerical time vector.

Time Series Property Descriptions (Continued)

Property	Description
`IsTimeFirst`	Logical value (`true` or `false`) that specifies whether the first or last dimension of the `Data` array aligns with the time vector. You can set this property when the `Data` array is square and it is ambiguous which dimension aligns with time. By default, the first `Data` dimension that matches the length of the time vector is aligned with `Time`. When you set this property to • `true`, the first dimension of the data array is aligned with the time vector. • `false`, the last dimension of the data array is aligned with the time vector. After a time series is created, this property is read-only.
`Name`	`timeseries` object name entered as a string. This name can differ from the name of the `timeseries` variable in the MATLAB workspace.
`Quality`	An integer vector or array containing values `-128` to `127` that specifies the quality in terms of codes defined by the `QualityInfo.Code` field. When `Quality` is a vector, it must have the same length as the time vector. In this case, each `Quality` value applies to the corresponding data sample. When `Quality` is an array, it must have the same size as the data array. In this case, each `Quality` value applies to the corresponding value of the data array.

Time Series Property Descriptions (Continued)

Property	Description
`QualityInfo`	Provides a lookup table that converts numerical `Quality` codes to readable descriptions. `QualityInfo` fields include the following: • `Code` — Integer vector containing values `-128` to `127` that defines the "dictionary" of quality codes, which you can assign to each `Data` value by using the `Quality` property • `Description` — Cell vector of strings, where each element provides a readable description of the associated quality `Code` • `UserData` — Stores any additional user-defined information The length of `Code` and `Description` must match.
`Time`	Vector of time values. When `TimeInfo.StartDate` is empty, the numerical `Time` values are measured relative to `0` in specified units. When `TimeInfo.StartDate` is defined, the time values are date strings measured relative to `StartDate` in specified units. The length of `Time` must match either the first or the last dimension of `Data`.

Time Series Property Descriptions (Continued)

Property	Description
`TimeInfo`	Uses the following fields to store contextual information about `Time`: • `Units` — Time units with the following values: `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, and `'nanoseconds'` • `Start` — Start time • `End` — End time (read-only) • `Increment` — Interval between two subsequent time values • `Length` — Length of the time vector (read-only) • `Format` — String defining the date string display format. See the MATLAB `datestr` function reference page for more information. • `StartDate` — Date string defining the reference date. See the MATLAB `setabstime (timeseries)` function reference page for more information. • `UserData` — Stores any additional user-defined information
`TreatNaNasMissing`	Logical value that specifies how to treat `NaN` values in `Data`: • `true` — (Default) Treat all `NaN` values as missing data except during statistical calculations. • `false` — Include `NaN` values in statistical calculations, in which case `NaN` values are propagated to the result.

Time Series Methods

In this section...

General Methods

Use the following methods to query and set object properties, and plot the data.

Methods for Querying Properties

Method	Description
`get (timeseries)`	Query `timeseries` object property values.
`getdatasamplesize`	Return the size of each data sample in a `timeseries` object.
`getqualitydesc`	Return data quality descriptions based on the `Quality` property values assigned to a `timeseries` object.
`isempty (timeseries)`	Evaluate to `true` for an empty `timeseries` object.
`length (timeseries)`	Return the length of the time vector.
`plot (timeseries)`	Plot the `timeseries` object.
`set (timeseries)`	Set `timeseries` property values.

Methods for Querying Properties (Continued)

Method	Description
`size (timeseries)`	Return the `size` property of a `timeseries` object.
`tstool`	Open the Time Series Tools GUI.

Data and Time Manipulation Methods

Use the following methods to add or delete data samples, and manipulate the `timeseries` object.

Methods for Manipulating Data and Time

Method	Description
`addsample`	Add a data sample to a `timeseries` object.
`ctranspose (timeseries)`	Transpose a `timeseries` object.
`delsample`	Delete a sample from a `timeseries` object.
`detrend (timeseries)`	Subtract the mean or best-fit line and remove all `NaN`s from time series data.
`filter (timeseries)`	Shape frequency content of time series data using a 1-D digital filter.
`getabstime (timeseries)`	Extract a date-string time vector from a `timeseries` object into a cell array.
`getinterpmethod`	Get the interpolation method for a `timeseries` object.
`getsampleusingtime (timeseries)`	Extract specified data samples from an existing `timeseries` object into a new `timeseries` object.
`idealfilter (timeseries)`	Apply an ideal pass or notch (noncausal) filter to a `timeseries` object.

Methods for Manipulating Data and Time (Continued)

Method	Description
`resample (timeseries)`	Select or interpolate data in a `timeseries` object using a new time vector.
`setabstime (timeseries)`	Set the time values in the time vector as date strings.
`setinterpmethod`	Set interpolation method for a `timeseries` object.
`synchronize`	Synchronize and resample two `timeseries` objects using a common time vector.
`transpose (timeseries)`	Transpose a `timeseries` object.
`vertcat (timeseries)`	Vertical concatenation for `timeseries` objects.

Event Methods

To construct an event object, use the constructor `tsdata.event`. For an example of defining events for a time series, see “Defining Events” on page 4-12.

Methods That Define and Use Events

Method	Description
`addevent`	Add one or more events to a `timeseries` object.
`delevent`	Delete one or more events from a `timeseries` object.
`gettsafteratevent`	Create a new `timeseries` object by extracting the samples from an existing time series that occur after or at a specified event.
`gettsafterevent`	Create a new `timeseries` object by extracting the samples that occur after a specified event from an existing time series.

Methods That Define and Use Events (Continued)

Method	Description
`gettsatevent`	Create a new `timeseries` object by extracting the samples that occur at the same time as a specified event from an existing time series.
`gettsbeforeatevent`	Create a new `timeseries` object by extracting the samples that occur before or at a specified event from an existing time series.
`gettsbeforeevent`	Create a new `timeseries` object by extracting the samples that occur before a specified event from an existing time series.
`gettsbetweenevents`	Create a new `timeseries` object by extracting the samples that occur between two specified events from an existing time series.

Arithmetic Operation Methods

Use the following operators to arithmetically combine `timeseries` objects.

Methods to Arithmetically Combine Time Series

Operation	Description
`+`	Add the corresponding data values of `timeseries` objects.
`-`	Subtract the corresponding data values of `timeseries` objects.
`.*`	Element-by-element multiplication of `timeseries` data.
`*`	Matrix-multiply `timeseries` data.
`./`	Right element-by-element division of `timeseries` data.
`/`	Right matrix division of `timeseries` data.

Methods to Arithmetically Combine Time Series (Continued)

Operation	Description
`.\`	Element-by-element left-array divide of `timeseries` data.
`\`	Left matrix division of `timeseries` data.

Statistical Methods

Use the following methods to calculate descriptive statistics for a `timeseries` object.

Methods for Calculating Descriptive Statistics

Method	Description
`iqr (timeseries)`	Return the interquartile range of `timeseries` data.
`max (timeseries)`	Return the maximum value of `timeseries` data.
`mean (timeseries)`	Return the mean of `timeseries` data.
`median (timeseries)`	Return the median of `timeseries` data.
`min (timeseries)`	Return the minimum of `timeseries` data.
`std (timeseries)`	Return the standard deviation of `timeseries` data.
`sum (timeseries)`	Return the sum of `timeseries` data.
`var (timeseries)`	Return the variance of `timeseries` data.

Time Series Collection Constructor

In this section...

Introduction

The MATLAB object, called `tscollection`, is a MATLAB variable that groups several time series with a common time vector. The `timeseries` objects that you include in the `tscollection` object are called *members* of this collection.

MATLAB provides several methods for convenient analysis and manipulation of `timeseries` in a `tscollection` object.

Time Series Collection Constructor Syntax

Before you implement the MATLAB methods specifically designed to operate on a collection of `timeseries` objects, you must create a `tscollection` object to store the data.

The following table summarizes the syntax for using the `tscollection` constructor. For an example of using this constructor, see "Creating Time Series Collection Objects" on page 4-12.

Time Series Collection Syntax Descriptions

Syntax	Description
`tsc = tscollection(ts)`	Creates a `tscollection` object `tsc` that includes one or more `timeseries` objects. The `ts` argument can be one of the following: • Single `timeseries` object in the MATLAB workspace • Cell array of `timeseries` objects in the MATLAB workspace The `timeseries` objects share the same time vector in the `tscollection`.
`tsc = tscollection(Time)`	Creates an empty `tscollection` object with the time vector `Time`. When time values are date strings, you must specify `Time` as a cell array of date strings.
`tsc = tscollection(Time, TimeSeries, 'Parameter', Value, ...)`	Optionally enter the following parameter-value pairs after the `Time` and `TimeSeries` arguments: • `Name` (see "Time Series Collection Properties" on page 4-38) • `IsDatenum` When set to `true`, `IsDatenum` specifies that `Time` values are dates in the format of MATLAB serial dates.

Time Series Collection Properties

This table lists the properties of the `tscollection` object. You can specify the `Name`, `Time`, and `TimeInfo` properties as input arguments in the `tscollection` constructor.

Time Series Collection Property Descriptions

Property	Description
`Name`	`tscollection` object name entered as a string. This name can differ from the name of the `tscollection` variable in the MATLAB workspace.

Time Series Collection Property Descriptions (Continued)

Property	Description
`Time`	A vector of time values. When `TimeInfo.StartDate` is empty, the numerical `Time` values are measured relative to 0 in specified units. When `TimeInfo.StartDate` is defined, the time values represent date strings measured relative to `StartDate` in specified units. The length of `Time` must match either the first or the last dimension of the `Data` property of each `tscollection` member.
`TimeInfo`	Uses the following fields to store contextual information about `Time`: • `Units` — Time units with the following values: `'weeks'`, `'days'`, `'hours'`, `'minutes'`, `'seconds'`, `'milliseconds'`, `'microseconds'`, and `'nanoseconds'` • `Start` — Start time • `End` — End time (read-only) • `Increment` — Interval between two subsequent time values. The increment is `NaN` when times are not uniformly sampled. • `Length` — Length of the time vector (read-only) • `Format` — String defining the date string display format. See the MATLAB `datestr` function reference page for more information. • `StartDate` — Date string defining the reference date. See the MATLAB `setabstime (timeseries)` function reference page for more information. • `UserData` — Stores any additional user-defined information

Time Series Collection Methods

In this section...

General Time Series Collection Methods

Use the following methods to query and set object properties, and plot the data.

Methods for Querying Properties

Method	Description
`get (tscollection)`	Query `tscollection` object property values.
`isempty (tscollection)`	Evaluate to `true` for an empty `tscollection` object.
`length (tscollection)`	Return the length of the time vector.
`plot (timeseries)`	Plot the time series in a collection.
`set (tscollection)`	Set `tscollection` property values.
`size (tscollection)`	Return the size of a `tscollection` object.
`tstool`	Open the Time Series Tools GUI.

Data and Time Manipulation Methods

Use the following methods to add or delete data samples, and manipulate the `tscollection` object.

Methods for Manipulating Data and Time

Method	Description
`addts`	Add a `timeseries` object to a `tscollection` object.
`addsampletocollection`	Add data samples to a `tscollection` object.
`delsamplefromcollection`	Delete one or more data samples from a `tscollection` object.
`getabstime (tscollection)`	Extract a date-string time vector from a `tscollection` object into a cell array.
`getsampleusingtime (tscollection)`	Extract data samples from an existing `tscollection`object into a new `tscollection` object.
`gettimeseriesnames`	Return a cell array of time series names in a `tscollection` object.
`horzcat (tscollection)`	Horizontal concatenation of `tscollection` objects. Combines several `timeseries` objects with the same time vector into one time series collection.
`removets`	Remove one or more `timeseries` objects from a `tscollection` object.
`resample (tscollection)`	Select or interpolate data in a `tscollection` object using a new time vector.
`setabstime (tscollection)`	Set the time values in the time vector of a `tscollection` object as date strings.
`settimeseriesnames`	Change the name of the selected `timeseries` object in a `tscollection` object.
`vertcat (tscollection)`	Vertical concatenation of `tscollection` objects. Joins several `tscollection` objects along the time dimension.

5

Time Series Tools

The following sections describe how to use the MATLAB Time Series Tools graphical user interface (GUI) for analyzing time series data.

Introduction (p. 5-2)	Summarizes the Time Series Tools window and workflow
Importing and Exporting Data (p. 5-7)	Describes supported data sources and instructions for importing and exporting data in Time Series Tools
Plotting Time Series (p. 5-13)	Describes how to work with time plots, histograms, spectral plots, correlation plots, and *XY* plots
Selecting Data for Analysis (p. 5-28)	Provides instructions for selecting data on which to focus your analysis
Editing Data, Time, Attributes, and Events (p. 5-31)	Describes how to edit data, time, units, interpolation method, quality codes, and events for time series
Processing and Manipulating Time Series (p. 5-41)	Provides instructions for processing time series, including filtering, interpolating, resampling, and algebraically manipulating data
Example — Time Series Tools (p. 5-42)	Provides an example of importing, plotting, and analyzing time series

Introduction

In this section...

Opening Time Series Tools

To open Time Series Tools, type the following at the MATLAB prompt:

```
tstool
```

You can also open Time Series Tools using the MATLAB **Start** button by selecting **Start > MATLAB > Time Series Tools**.

For a description of the Time Series Tools GUI, see "Time Series Tools Window" on page 5-3.

To learn how to import data into Time Series Tools, see "Importing and Exporting Data" on page 5-7.

You can also start Time Series Tools and simultaneously import the following kinds of objects from the MATLAB workspace:

- `timeseries`
- `tscollection`
- Simulink® logged signals

Note You cannot import Simulink logged signals that contain a `/` in their `Name` property at any point in the signal hierarchy.

Syntax for Loading Data from the MATLAB Workspace

MATLAB Object	Syntax	Description
`timeseries`	`tstool(tsname)`	`tsname` is the name of a `timeseries` object.
`tscollection`	`tstool(tscname)`	`tscname` is the name of a `tscollection` object.
Simulink logged-signal data	`tstool(sldata)`	`sldata` is the name of a signal logged in a Simulink model.

Getting Help

Time Series Tools provides extensive context-sensitive help directly from the GUI.

In the Time Series Tools window, the context-sensitive help pane is available on the right to assist you with the primary tasks. To toggle between displaying or hiding the help pane, click the  (**Help**) button in the toolbar. You can resize the help pane by dragging the vertical divider to the left or to the right.

Context-sensitive help is also available via the **Help** button in Time Series Tools dialog boxes.

Time Series Tools Window

The Time Series Tools window consists of the following three areas:

- Time Series Session tree

  Organizes time series data and plots (or **Views**).

  The **Simulink Time Series** node is shown only when you have installed Simulink.

- Options and Settings pane

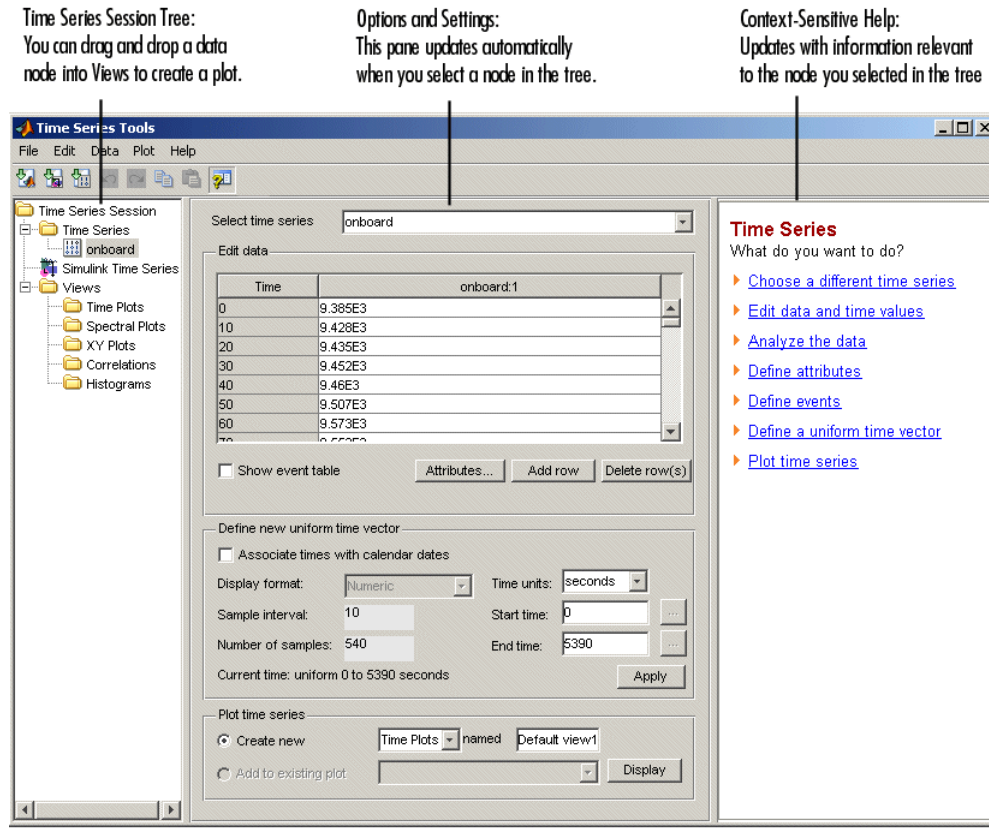
  After you select a node in the tree, this pane displays options and settings pertaining to the node you selected in the tree.

- Context-Sensitive Help pane

Provides information and instructions about entering the options and settings currently shown in Time Series Tools. You can toggle between displaying or hiding this help by clicking the button in the toolbar. You can change the width of the help pane by dragging the vertical divider to the left or to the right.

To learn about other help available in Time Series Tools, see “Getting Help” on page 5-3.

The following figure shows the three main areas of the Time Series Tools GUI:



Time Series Tools Workflow

When you analyze data using Time Series Tools, your workflow might include the following tasks:

1. Import data from an Excel workbook, MAT-file, or MATLAB workspace.

   For more information, see “Importing and Exporting Data” on page 5-7.

2. Create a time plot to gain insight into the data features.

   For more information, see “Creating a Plot” on page 5-14.

3. Select data subset for analysis.

   For more information, see “Selecting Data for Analysis” on page 5-28.

4. Edit the data by

   - Identifying and removing outliers or “dead time” (see “Selecting Data Using Rules” on page 5-28).
   - Manually correcting errors (see “Editing Data, Time, Attributes, and Events” on page 5-31).

5. Process the data by

   - Interpolating or removing missing values.
   - Detrending data by subtracting a mean value or a linear trend.
   - Filtering to smooth and shape the data.
   - Algebraically manipulating existing time series to create a new time series.
   - Resampling data using a specified time vector by selecting or interpolating values.

   For more information, see “Processing and Manipulating Time Series” on page 5-41.

6. Generating correlation plots, spectral plots, histograms, and *XY* plots.

   For more information, see “Plotting Time Series” on page 5-13.

7 Exporting data from Time Series Tools to the MATLAB workspace or to a file.

For more information, see “Exporting Data from Time Series Tools” on page 5-12.

Generating Reusable M-Code

You can enable automatic generation of reusable M-code while you perform operations that modify data in Time Series Tools. To do this, select **File > Record M-Code** in the Time Series Tools window.

If you are new to programming with MATLAB `timeseries` methods, you can use the generated M-code to get syntax examples. For more information about programming with MATLAB `timeseries` objects, see Chapter 4, “Time Series Objects and Methods”.

For an example of automatically generating and viewing M-code, see “Example — Time Series Tools” on page 5-42.

Note The scope of the **Record M-Code** feature is restricted to recording actions on the time series data itself. It does not generate code to import data or reproduce time series plots.

Importing and Exporting Data

In this section...

Types of Data You Can Import

You can import data into Time Series Tools from

- A Microsoft Excel workbook, a text file, or a MAT-file.
- An array in the MATLAB workspace.
- A `timeseries` or `tscollection` object in the MATLAB workspace.

  For more information about creating these objects, see Chapter 4, "Time Series Objects and Methods".
- Simulink logged-signal data from a Simulink model.

Note You cannot import a `timeseries` or `tscollection` object from a MAT-file.

How to Import Data

This section includes the following topics:

- "Importing Time Series and Time Series Collection Objects" on page 5-8
- "Importing Data from External Files" on page 5-8
- "Using the Import Wizard" on page 5-8

Importing Time Series and Time Series Collection Objects

If you have already encapsulated time series data in a `timeseries` or `tscollection` object in the MATLAB workspace, you can open Time Series Tools and import the data in a single operation. Simply right-click the object name in the Workspace Browser and choose **Open in Time Series Tools** from the context menu.

Importing Data from External Files

Once you have opened Time Series Tools, use the following commands to import data from external files. Each command opens a dialog box. You can get detailed information about options by clicking **Help**.

Data Source	Import Command
Microsoft Excel worksheet (`.xls`)	Select **File > Create Time Series from File** to open the Import Wizard.
Text file (`.csv`, `.txt`, `.dat`)	Select **File > Create Time Series from File** to open the Import Wizard.
MAT-file array (`.mat`)	Select **File > Create Time Series from File** to open the Import Wizard.
MATLAB workspace array	Select **File > Import from Workspace > Array Data** to open the Import Wizard.
`timeseries` or `tscollection` object in the MATLAB workspace	Select **File > Import from Workspace > Time Series Objects or Collections**.
Simulink logged signal	Select **File > Import from Workspace > Simulink Data Logs**.

Using the Import Wizard

When in Time Series Tools, you import data from the MATLAB workspace or an external file using the Import Wizard. The Import Wizard lets you select the data to import when analyzing a portion of an Excel worksheet or specific columns or rows in a MATLAB array.

After you select the data, you can specify to import time values from a file or define a uniformly spaced time vector in the Import Wizard. For an example

of importing data from an Excel worksheet, see “Importing Data into Time Series Tools” on page 5-43.

Each time series you import is added as a data node to the **Time Series Session** tree.

Note The Import Wizard in Time Series Tools imports data as `timeseries` objects. This is different from the Import Wizard you access from the MATLAB Command Window, which imports data as MATLAB vectors and matrices.

For instructions about working with the Import Wizard, click **Help** in the Import Wizard window. You can also get help on specific fields in the Wizard as follows:

1. Right-click the text label of a field for which you want to get help.
2. Select **What's This** from the shortcut menu.

Changes to Data Representation During Import

When you import data into Time Series Tools, a copy of the data is imported without affecting the original data source.

The data copy is changed during import, as follows:

- Rowwise data is transposed to become columnwise with the time vector in the first column.
- Data with more than two dimensions is reshaped to two dimensions such that dimensions three and higher become additional columns. For example, a 2-by-3-by-5 data array becomes a 2-by-15 data array.
- Non-`double` data, such as `int`, `logical`, and `fixed-point`, is converted to `double`.
- Missing data values are replaced by `NaN`s.
- A sparse matrix is converted to a full matrix.

Caution When you export data from Time Series Tools to a file or to the MATLAB workspace, please note that its representation might differ from what you imported into Time Series Tools. For more information about exporting data, see "Exporting Data from Time Series Tools" on page 5-12.

Importing Multivariate Data

When your data consists of several related variables measured at the same time, you might want to group this data so that you can plot variables together or perform calculations on all variables simultaneously.

There are two ways to represent multivariate data in Time Series Tools:

- Create a time series collection with a common time vector, where each time series is a member of the collection.
- Import a data array into a single `timeseries` object, where each time series is stored as a column.

Choosing How to Represent Multivariate Data

How you choose to represent your data depends on whether the variables have the same or different units.

When your data contains different measurements of the same quantity (same units), you can store all measurements as separate columns in a single time series. Plotting such a time series displays all columns on the same axes and distinguishes the data sets by line and marker styles. For more information, see "Customizing Line and Marker Styles" on page 5-15.

When your data contains different quantities, measured in different units, you might want to distinguish these quantities on plots and during analysis. In this case, we recommend that you store each quantity as a separate time series and then group them into a time series collection. For example, if you are working with stock-price data in a portfolio, you might represent each stock as a separate time series and group them in a collection. When you plot this collection, each member is plotted on separate axes. However, when you perform data-analysis operations on the collection, such as filtering or

interpolation, these operations are applied to all time series in the collection simultaneously.

Creating a Time Series Collection

You can create a time series collection in the MATLAB Command Window, as described in Chapter 4, "Time Series Objects and Methods", and then import the collection into Time Series Tools. Alternatively, you can use the Import Wizard to facilitate creating the `timeseries` objects and then group them into a collection in the MATLAB Command Window.

The following procedure describes one way to create a time series collection using data from a file.

Note At each step, you can click the **Help** button in the GUI to access context-sensitive help.

---

1. To import each variable in the Microsoft Excel worksheet or MATLAB array as a separate time series in Time Series Tools, select **File > Import from Workspace > Array Data**. This opens the Import Wizard.
2. After importing the data, select the **Time Series** node in the tree and export these time series to the MATLAB workspace.
3. In the MATLAB Command Window, combine individual time series into a time series collection object. For an example of creating a time series collection, see "Creating Time Series Collection Objects" on page 4-12.
4. In Time Series Tools, select **File > Import from Workspace > Time Series Objects or Collections** and import the collection from the MATLAB workspace.

Importing Data with Missing Values

When you import data from a Microsoft Excel worksheet into Time Series Tools that contains missing values, the missing data is automatically replaced with `NaN`s. `NaN`s are ignored in Time Series Tools calculations.

To remove or interpolate missing values:

1. Select a time series or a collection in the **Time Series Session** tree containing missing values.
2. Select **Data > Interpolate** or **Data > Remove Missing Data**, depending on the operation you want to perform. This opens the Process Data dialog box.
3. Click **Help** to access context-sensitive help on specific options in the dialog box.

Exporting Data from Time Series Tools

Importing data into Time Series Tools creates a copy of the original data. After you finish analyzing the data in Time Series Tools, you must export it to a file or to the MATLAB workspace to make it available for other processing in MATLAB.

To export a time series or a collection, select the desired node in the **Time Series Session** tree. Then, do one of the following:

- Export to a file (Microsoft Excel worksheet or MAT-file):

  Select **File > Export > To File**.

  When you export a time series collection, the individual time series are extracted into separate Microsoft Excel worksheets.

- Export to the MATLAB workspace:

  Select **File > Export > To Workspace**.

Plotting Time Series

In this section...

Types of Plots in Time Series Tools

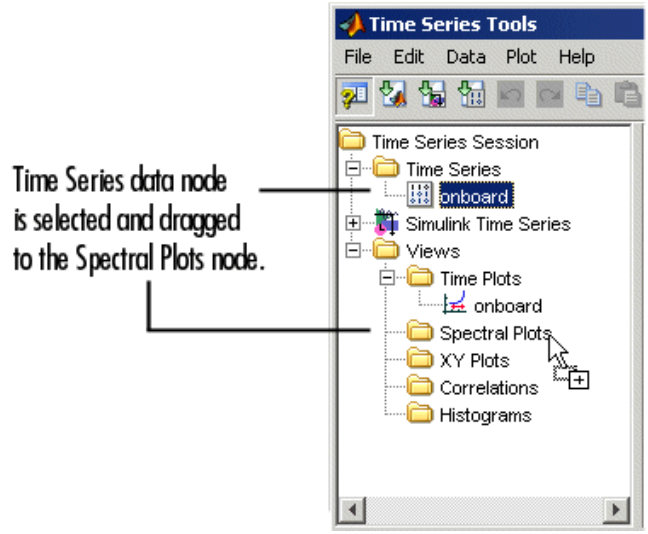
You can generate the following types of plots in Time Series Tools.

Plot Type	Description
Time Plot	Plots data as a function of time to help you see important features, such as outliers, discontinuities, trends, and periodicities.
Histogram	Plots the number of data values that occur in specified data ranges, called *bins*.
Spectral Plot	Shows data periodicities by plotting the estimated power spectral density as a function of frequency.
Correlation Plot	Shows the autocorrelation of a time series or cross-correlation between two time series.
XY Plot	Shows the relationship between two time series by plotting the data values of one on the x-axis and the data values of the other on the y-axis.

Creating a Plot

You can create a plot in Time Series Tools is by dragging and dropping a **Time Series** data node in the **Time Series Session** tree onto a **Views** node.

The following figure shows an example of how to create a spectral plot by dragging the `onboard` time series onto the **Spectral Plots** node:



This opens the spectral plot in the Time Series Plots window and adds a tree node under **Spectral Plots**. The Time Series Plots window is similar to the MATLAB Figure window but includes additional commands in the toolbar and the **Tools** menu.

Tip To change the default plot name, right-click the plot node and select **Rename** and enter the new name.

Subplots. To create subplots in a single figure window, drag several time series onto the same plot node. If a time series contains several columns of data, all data columns are plotted on the same axes. See “Editing Plot Appearance” on page 5-15 for information on interactively modifying the appearance of subplots.

***XY* and cross-correlation plots.** These plots require two time series. To create these plots, drag one time series onto a plot node and then drag a second time series onto the same plot node.

Customizing Line and Marker Styles

When you plot several time series on the same axes, or a single `timeseries` object that contains multiple columns of data, you can specify how to visually distinguish between the different sets of data in the plot.

To distinguish data by color, type of marker, or line style, select **Plot > Set Line Properties** in the Time Series Tools window. This opens the Line Styles dialog box. Click **Help** to learn how to work with this dialog box.

Note Your changes are applied to all open plots.

For an example of setting line styles, see "Creating a Time Plot" on page 5-46.

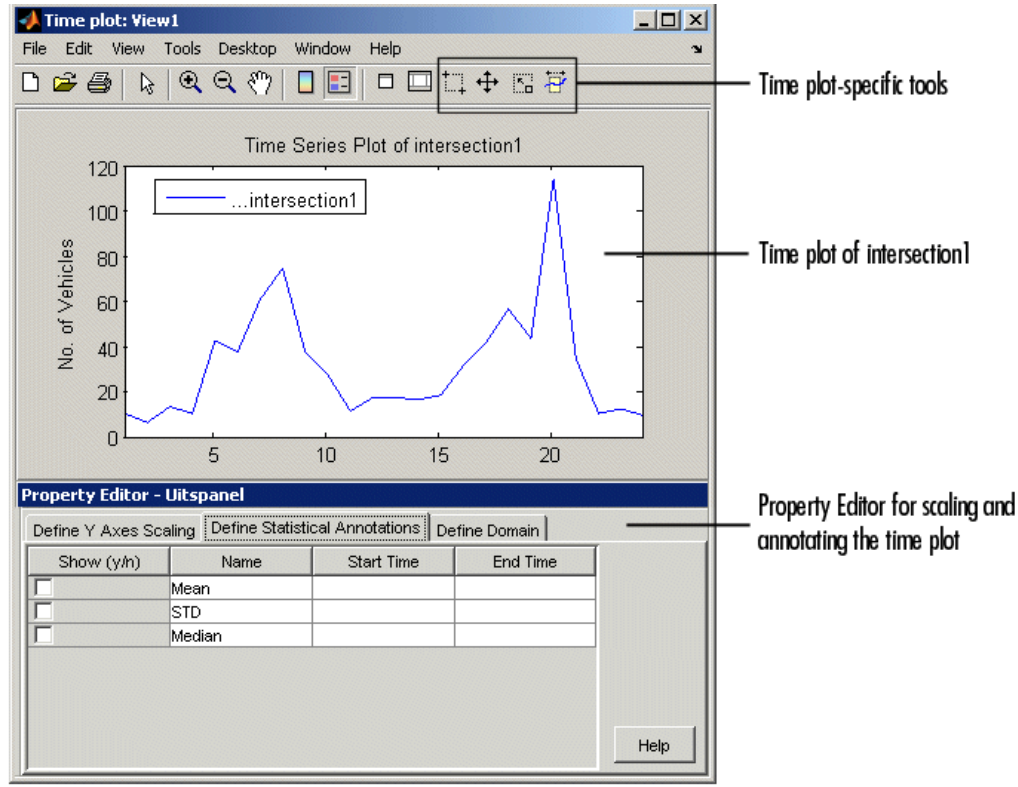
Editing Plot Appearance

After you create a plot, you can modify the plot appearance using the Property Editor as follows:

- Change the range of the horizontal and vertical axes.
- Show statistical annotations on the plot, such as the mean and standard deviation.

  The kinds of statistical quantities you can display vary depending on the type of plot.

The following figure shows the location of the Property Editor relative to the plot window:



To display the Property Editor for any Time Series Tools plot:

1. Select the plot in the **Time Series Session** tree.
2. In Time Series Tools, click the **Edit Plot** button. This displays the plot window on top with the Property Editor below the plot.
3. In the Property Editor, click **Help** to get information about options and settings.

Note The Property Editor options change depending on the type of plot and the plot item you select, such as lines or plot legends.

Subplots. You can change subplot indices interactively. To do so, click on a plotted line in a time series view and drag and drop it from one subplot to another. To create a new subplot, drag and drop the plotted line below the bottom axes.

Time Plots

By plotting data as a function of time, you can quickly gain insight into the following data features:

- Outliers, or values that do not appear to be consistent with the rest of the data
- Discontinuities
- Trends
- Periodicities
- Time intervals containing the data of interest

These features, when considered in the context of the data, enable you to plan your analysis strategy. For more information about creating a time plot, see “Creating a Plot” on page 5-14.

After you create the plot, you can use the Property Editor to

- Define Y-axis scale.
- Display statistical annotations on the plot, such as mean, standard deviation, and median.
- Define X-axis scale (or domain).

In the Property Editor, click **Help** to get information about options and settings.

The Time Plot window contains the following toolbar commands specific to working with time series data.

Time Plot Commands

Button	Description
	Select Data — Enables you to click and drag a rectangular region on the time plot to select the data inside the region.
	Move Time Series — Enables you to click and drag a time series to translate a time series on the plot and recalculate the data and time values. When you translate a time series in time, its time vector is shifted by a constant offset. If you had associated any events with this time series, the events are not shifted with the time series. For more information about editing event times, see “Defining Events” on page 5-37.
	Rescale Time Series — Rescales both axes of the time plot to the original view.
	Select Interval — Enables you to click and drag to select data corresponding to one or more time intervals. You can select multiple disconnected intervals.

Spectral Plots

You use a spectral plot (or periodogram) to determine the frequencies of the periodic variations in the data and to filter the data. For more information about creating a periodogram, see “Creating a Plot” on page 5-14.

The periodogram is the unbiased estimate of the power spectral density of a time series, calculated as the scaled absolute value of the $(FFT)^2$ of the time series. The corresponding frequency vector is computed in cycles per unit time and has the same length as the power vector. The periodogram is scaled so that the variance equals the mean of the periodogram.

The periodogram is useful for picking out periodic components in the presence of noise; a peak in the periodogram indicates an important contribution to variance frequencies near the value that corresponds to the peak.

After you create the plot, you can use the Property Editor to

- Define Y-axis scale.
- Display the variance for a selected frequency range on the plot.

  The periodogram is scaled so that the variance equals the mean of the periodogram.
- Define frequency scale.

In the Property Editor, click **Help** to get information about options and settings.

Filtering the Data

You can use the spectral plot to apply an ideal pass or stop filter to the data.

You use the *ideal notch (stop) filter* when you want to attenuate the variations in the data for a specific frequency range. Alternatively, you use the *ideal pass filter* to allow only the variations in a specific frequency range. These filters are "ideal" in the sense that they are not realizable; an ideal filter is noncausal and the ends of the filter amplitude are perfectly flat in the frequency domain.

To apply an ideal filter:

1. In the Spectral Plot window, click the **Select Frequency Interval(s)**  button in the toolbar.
2. Click and drag on the plot to select a frequency interval. The selected interval appears in a different color.
3. Decide if you want to select another frequency interval.
   - If yes, repeat step 2. The previously selected remains selected.
   - If no, go to step 4.
4. Right-click a selected region on the plot and select one of the following from the shortcut menu:
   - To allow only the variations in the selected frequency range, select **Pass**.
   - To remove the variations in the selected frequency range, select **Notch**.

Histograms

The histogram plot shows the distribution of data by counting the number of data values within a specific range of values and displaying each range as a rectangular bin. The heights of the bins represent the numbers of values that fall within each range. For more information about creating a histogram, see “Creating a Plot” on page 5-14.

You can use a histogram plot to select data values that fall in a specific range to exclude or include them in your analysis. If you want to interpolate specific data values, you can select them in a histogram plot first, and then replace them with `NaN`s. For more information, see “Removing or Replacing Data with NaNs” on page 5-21. Then, you can interpolate all values tagged as `NaN`s using the selected interpolation method. For more information about specifying an interpolation method, see “Defining Data Attributes” on page 5-34.

Note Time Series Tools generates a histogram plot of a time series by applying the MATLAB `hist` function.

After you create the plot, you can use the Property Editor to

- Define Y-axis scale.
- Display statistical annotations on the plot, including the mean and the median.
- Define data bins.

In the Property Editor, click **Help** to get information about options and settings.

Selecting Data

1. In the Histogram window, click the **Select Y Range Interval**  button in the toolbar.
2. Click and drag a rectangular region on the plot to select a data interval. The selected interval appears in a different color.

3 Decide if you want to select another data range.

- If yes, repeat step 2. The previously selected remains selected.
- If no, you are done.

Removing or Replacing Data with NaNs

After you select the data, as described in "Selecting Data" on page 5-20, you can delete it or replace it with `NaN`s. If you want to interpolate specific data values, you must replace the selected data with `NaN`s first.

To delete data, right-click the selected region and select **Remove Selection** from the shortcut menu.

To replace data with `NaN`s, right-click the selected region and select **Replace with NaNs** from the shortcut menu.

Correlation Plots

You can create autocorrelation plots (*correlograms*) and cross-correlation plots in Time Series Tools. A correlation plot shows correlation coefficients on the vertical axis, and lag values on the horizontal axis.

A *lag* is defined as the number of time steps by which a time series is shifted relative to itself (when autocorrelated), or relative to the corresponding time values of another time series (when crosscorrelated). Notice that a lag is not a time shift (in specified time units). However, you can interpret a lag as a time shift when the time series is uniformly sampled (autocorrelation), or when both time series are uniformly sampled with the same time interval (cross-correlation).

This section includes the following topics:

Note If your data is sampled at irregular time intervals, resample it on a uniform time vector before creating correlation plots. This is because correlation analysis only considers the number of time steps between data values, and not the actual time elapsed between successive measurements. For more information about resampling time series, see "Processing and Manipulating Time Series" on page 5-41.

Autocorrelation of a Time Series

The *autocorrelation* function is an important diagnostic tool for analyzing time series in the time domain. You use the autocorrelation plot, or *correlogram*, to better understand the evolution of a process through time by the probability of relationship between data values separated by a specific number of time steps.

The correlogram plots correlation coefficients on the vertical axis, and lag values on the horizontal axis. To learn more about correlation coefficients, see "Correlation Coefficients" on page 2-7.

To create a correlogram, drag and drop a time series into a **Correlations** node. Then explore the plot by editing the lag range in the Property Editor.

If a time series contains multiple data columns, your plot contains cross-correlations of the various data columns. For more information, see "Cross-Correlation of Time Series" on page 5-23.

Note A correlogram is not useful when the data contains a trend; data at all lags will appear to be correlated because a data value on one side of the mean tends to be followed by a large number of values on the same side of the mean. You must remove any trend in the data before you create a correlogram. For more information about accessing detrending functionality, see "Processing and Manipulating Time Series" on page 5-41.

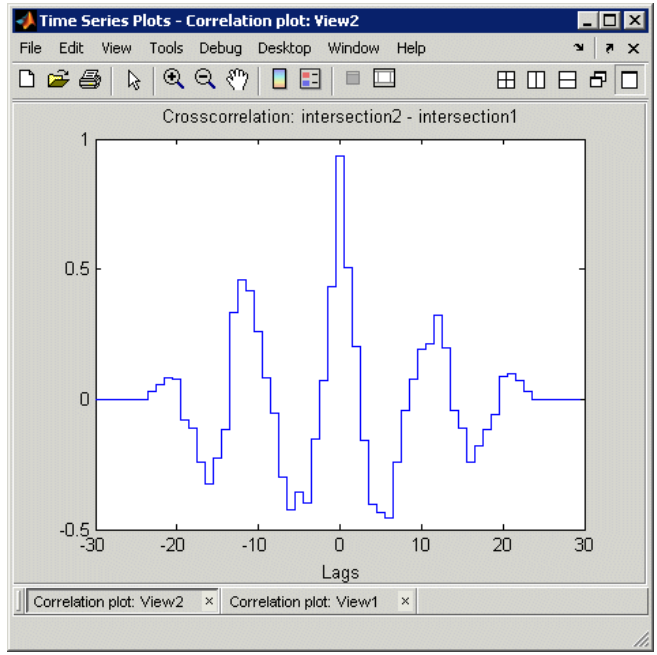
Cross-Correlation of Time Series

Cross-correlation is a measure of the degree of the linear relationship between two time series. A high correlation between time series at a specific lag might indicate a time delay in the system.

Note Before creating a cross-correlation plot, make sure that both time series have the same uniform time vector.

To create a cross-correlation plot, successively drag and drop the first time series and the second time series into the same **Correlations** node in the **Time Series Session** tree. Then explore the plot by varying the lag range in the Property Editor.

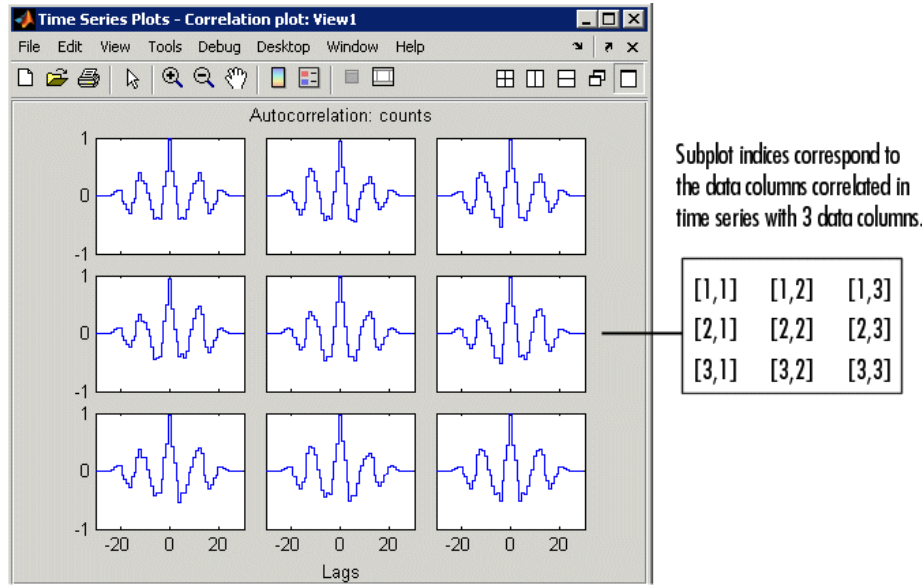
A cross-correlation plot of two time series, where each contains a single column of data, shows the degree of linear relationship between the data values in the two time series at various lags. For example, the following figure shows a cross-correlation plot of two time series, `intersection1` and `intersection2`. There is a high correlation when there is no lag in the data, as well as for lags of about `-11` and `11`.



Cross-Correlation of Two Time Series

A cross-correlation plot of two time series, where each contains multiple data columns, is displayed as a grid of subplots. The number of subplots equals the number of columns of data in the first time series multiplied by the number of columns of data in the second time series.

When you autocorrelate a time series with multiple data columns, the resulting plot also contains subplots. The diagonal of the subplot is the autocorrelation of a specific data column. The off-diagonal subplots are cross-correlation plots of the various columns. The subplot indices correspond to the indices of the data columns being correlated. For example, the figure below shows a correlation plot of the time series `counts` with three data columns.



Cross-Correlation of Multiple Data Columns in a Time Series

Interpreting Correlation Plots

The following table describes the degree of relationship between the data values at a given lag for various correlation values.

Correlation Value	Meaning
Close to 1	There is a relationship between data values at a specific lag: an increase in one corresponds to an increase in the other.
0	The variations in the data show no relationships at this lag.
Close to -1	There is an anticorrelation between the data values at a specific lag: a decrease in one data value corresponds to an increase in the other data value.

Cross-Correlation Algorithm

When computing the cross-correlation of two vector-valued time series `x` and `y`, Time Series Tools uses an algorithm that is functionally equivalent to calling `xcorr` from Signal Processing Toolbox with the `'biased'` option, after the time series means have been removed. Unlike `xcorr`, however, the cross-correlation estimate in Time Series Tools also works for matrix-valued time series `X` and `Y`, where it computes the cross-correlation of `X(:,i)` against `Y(:,j)` for all combinations of columns `i` and `j`. Note that Time Series Tools do not actually use the `xcorr` code, but rather a simplified version which works under these restricted assumptions.

XY Plots

An *XY* plot plots the data values of one time series against the data values of another time series at corresponding times. Any relationship between the two time series is evident from a pattern on the plot. For example, when the points on the *XY* plot form a straight line, there is a linear relationship between the data values of the two time series plotted. The *XY* plot does not show any time information.

Note To generate an *XY* plot, both time series must have the same time vectors.

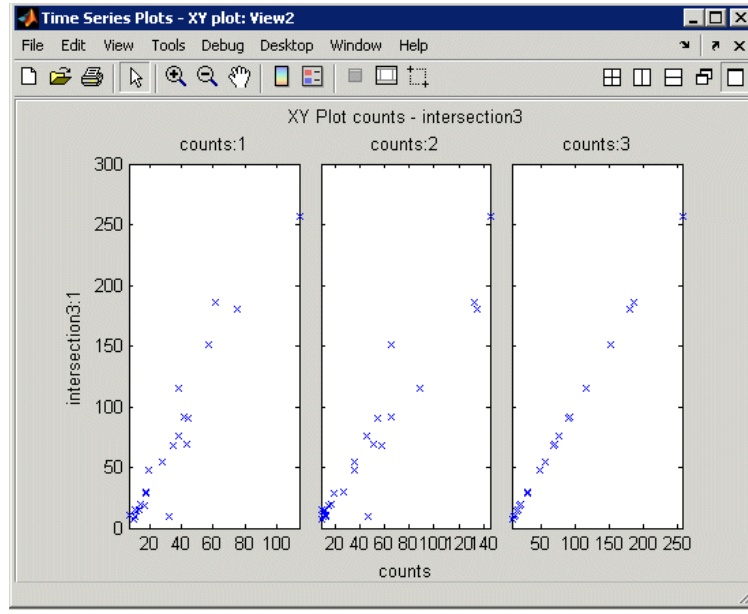
To create an *XY* plot, successively drag and drop the first time series and the second time series into the same **XY Plots** node in the **Time Series Session** tree.

When you are plotting two time series where each contains a single column of data, the *XY* plot includes a single set of axes. The pairs of data values from the same position in the column of data; that is, the third data point from one column is plotted against the third data point from the other column. For an example of generating such an *XY* plot, see “Comparing Data on an *XY* Plot” on page 5-53.

An *XY* plot of two time series, where each contains one or multiple data columns, is displayed as a grid of subplots. The number of subplots equals the number of columns of data in the first time series multiplied by the number

of columns of data in the second time series. The subplot indices correspond to the indices of the data columns.

The following figure shows an *XY* plot, where the data values in time series `count` are plotted on the *X*-axis against the corresponding data values of `intersection1` on the *Y*-axis. Because `count` contains three data columns and `intersection1` contains one data column, the *XY* plot window shows three subplots.



***XY* Plot Where One Time Series Contains Three Data Columns**

Selecting Data for Analysis

In this section...

Selecting Data Using Rules

You can select data using logical expressions in the Select Data Using Rules dialog box, which you access from a time plot. For more information about creating a time plot, see “Creating a Plot” on page 5-14.

To open the Select Data Using Rules dialog box, right-click inside the time plot and choose **Select Data** from the shortcut menu. Click **Help** in the dialog box to get information about specific options.

You can define up to four kinds of data-selection conditions:

- Bounds — Upper and lower bounds for time and data values
- Outliers — Condition for detecting outliers, or data values that are outside a specified confidence level
- MATLAB expression — A logical MATLAB expression that selects specific data values
- Flatlines – Condition for detecting a specified number of successive data points with a constant value

---

Tip To learn how to exclude data from analysis based on your selection, see “Excluding Data from Analysis” on page 5-30.

Selecting Data Graphically

This section describes how to select data in a time plot by using the mouse. For more information on creating a time plot, see “Creating a Plot” on page 5-14.

You can select data using two modes:

- Data mode — Enables you to select data values in a rectangular region on the time plot.

  For more information, see “Selecting Data in a Rectangular Region” on page 5-29.

- Time mode — Enables you to select data values in one or more time intervals on the time plot.

  For more information, see “Selecting Data in a Time Interval” on page 5-29.

Tip To learn how you can select specific data values in a histogram plot, see “Selecting Data” on page 5-20.

Selecting Data in a Rectangular Region

1. In the Time Plot window, click the **Select Data** button in the toolbar.
2. Click and drag a rectangular region on the plot that encloses the data you want to select.

   The data values are selected when you release the mouse button.
3. Decide if you want to select another region.

   - If yes, repeat step 2. This does not clear the previous selection.
   - If no, you can continue by excluding data from analysis (see “Excluding Data from Analysis” on page 5-30).

Selecting Data in a Time Interval

1. In the Time Plot window, click the **Select Time Interval(s)** button in the toolbar.

2 Click the start of a region that encloses the time interval where you want to select data, and then drag it. The selected time interval appears in a different color.

3 Decide if you want to select another time interval.

- If yes, repeat step 2. This does not clear the previous selection.
- If no, you can continue by excluding data from analysis (see "Excluding Data from Analysis" on page 5-30).

Excluding Data from Analysis

After you select the data, you can either exclude or keep the selected values. The following table summarizes how to do this.

Task	Operation
Exclude selected data from analysis	Right-click the selected data in the time plot and select **Remove Observations** from the shortcut menu. When there are multiple data columns in a single time series, this removes the entire data sample at that time.
Exclude unselected data from analysis	Right-click the selected data in the time plot and select **Keep Observations** from the shortcut menu.

Editing Data, Time, Attributes, and Events

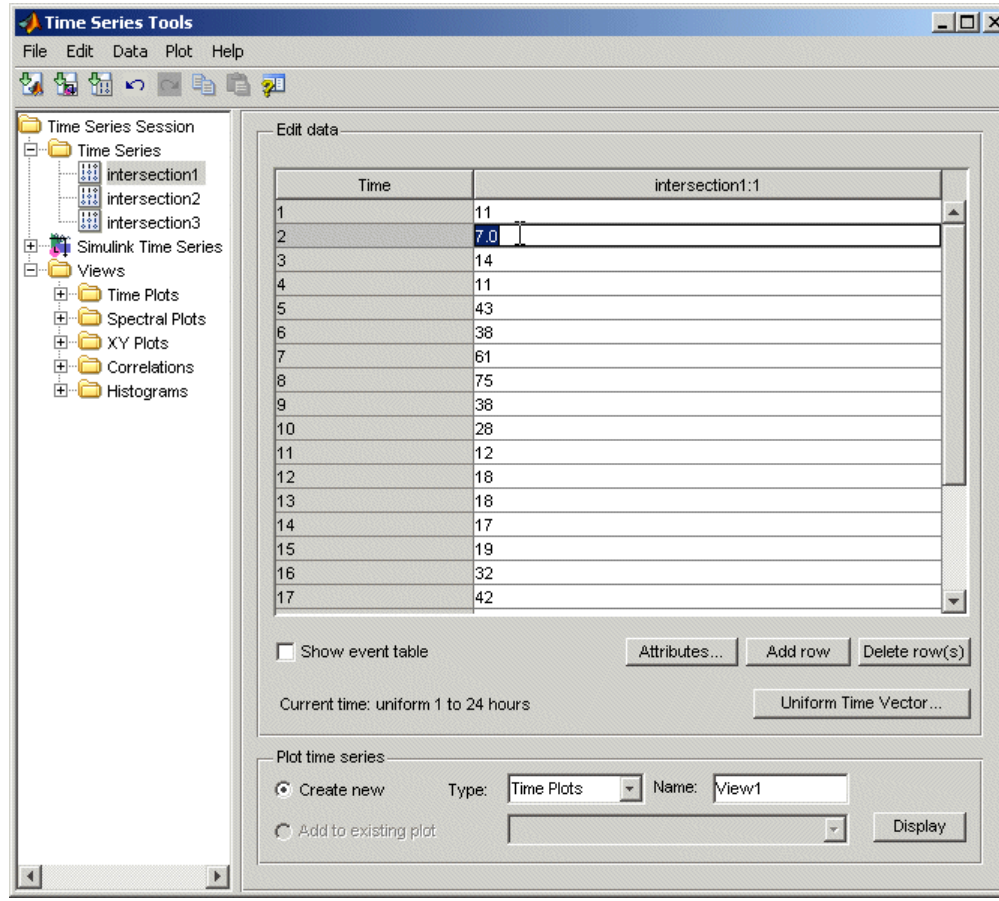
In this section...

Displaying the Data Table

To display the time series in an editable table, select the time series node in the Time Series Session tree.

In the following figure, the time series `intersection1` is selected in the tree and its data table is shown on the right. The **Time** column contains time values and the **intersection1:1** column contains the corresponding data values in the first column and only data column of `intersection1`.

If `intersection1` had multiple data columns, they would appear in the table and numbered as **intersection1:2**, **intersection1:3**, and so on. The data column headers are also used as plot labels to distinguish time series in plots. For more information about creating plots, see “Plotting Time Series” on page 5-13.



Note To toggle between displaying and hiding the help pane in Time Series Tools, click the button in the toolbar.

Editing Data and Time

After you display the time series data, as described in “Displaying the Data Table” on page 5-31, you can edit specific data and time values, define a uniform time vector, and add or remove data samples.

Edit Time or Data Values

To edit a specific time or data value, double-click that cell in the table and enter the new value. Press **Enter**.

Note When entering time values, you must use the current display format of your time vector. For more information, see "Time Vector Format" on page 4-21.

Define a Uniform Time Vector

To define a uniformly-increasing time vector, click **Uniform Time Vector** below the data table. This opens the Define Uniform Time Vector dialog box.

Here, you specify the start and end time of the time vector, the time units, and the display format. The time interval is calculated automatically by dividing the total time range by the number of data samples. You can get more instructions by clicking **Help** in the Define Uniform Time Vector dialog box.

When you are done specifying the time vector, the new time values replace the previous time values in the data table.

Add Data Samples

To insert a row in the data table, click any cell in a row and click the **Add Row** button. Enter the time and the corresponding data values.

Delete Data Samples

To delete a row in the data table, select one or more rows with the mouse and click the **Delete Row(s)** button.

Defining Data Attributes

The following attributes are defined for time series:

- Units — Stored as metadata for each time series.
- Interpolation method — Default method used to fill in missing data or to resample data on a new time vector.
- Quality codes — Used to annotate the quality of each value in the data table.

Click the **Attributes** button below the data table to open the Define Data Attributes dialog box. For information about displaying the data table, see "Displaying the Data Table" on page 5-31.

Units and Interpolation Method

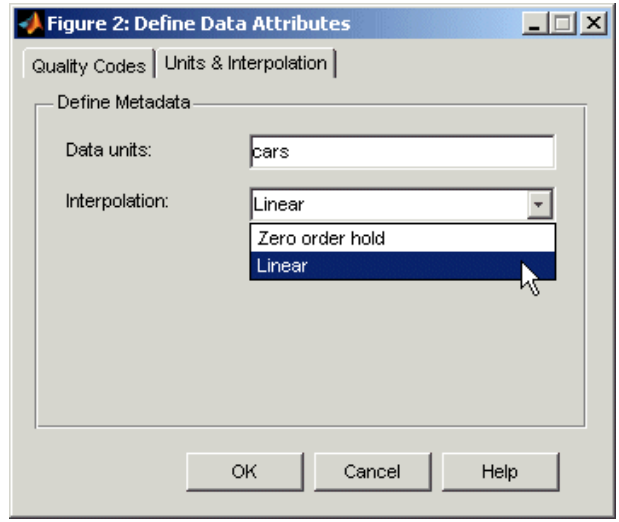
Data units are stored as metadata for the currently selected time series. If this time series contains multiple data columns, all data is assigned the same units.

In the **Units & Interpolation** tab, enter a string in the **Data units** field. For example, enter `N/m^2`.

The interpolation method you select here is used by default for this time series to fill in missing data or to resample the data on a new time vector.

In the **Units & Interpolation** tab, select one of the following **Interpolation** methods:

- `Linear` — A 1-D interpolation method that implements the MATLAB function `interp1` to fit a straight line between a pair of existing data points to calculate the missing value.
- `Zero-order hold` — Calculates the missing value by setting it equal to the last available data value. In other words, this methods "holds" the last value constant until the next available measurement.



Quality Codes

You can define quality codes to annotate the quality of each value in the data table. Each quality attribute consists of a numerical code and a brief description. For information about assigning quality codes to specific data values, see “Assigning Quality Codes to Data” on page 5-36.

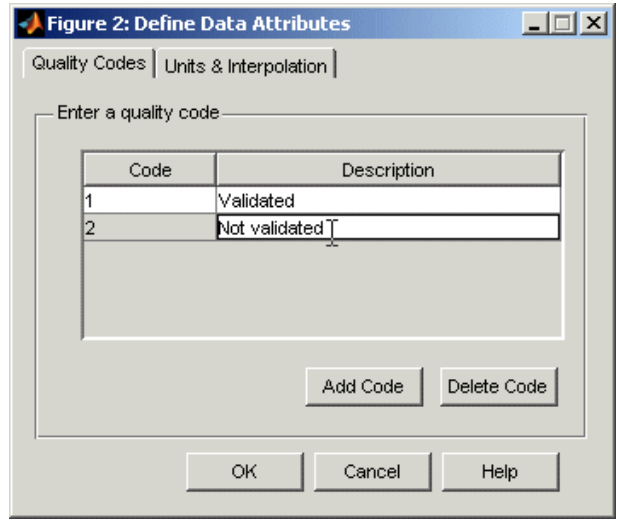
Tip To save time, first define the quality attribute that applies to most of your data values. It is automatically assigned to all data values. Then, define the attributes that occur less frequently and set them manually in the **Quality** column of the data table.

---

1. In the Define Data Attributes dialog box, click the **Quality Codes** tab.
2. Click the **Add Code** button. This adds an empty row in the Quality Codes table.
3. Click the empty cell in the **Code** column and type an integer from 0 to 127.
4. Press the **Tab** key. This highlights the cell in the **Description**. Type one or two words that briefly describe the numerical code, such as `Validated`.

5 To add another quality code, repeat steps 2 to 4. Or click **OK** to close the dialog box. This also assigns the first quality code you defined to all data values in the table.

The following figure shows two quality codes: `Validated` and `Not validated`.

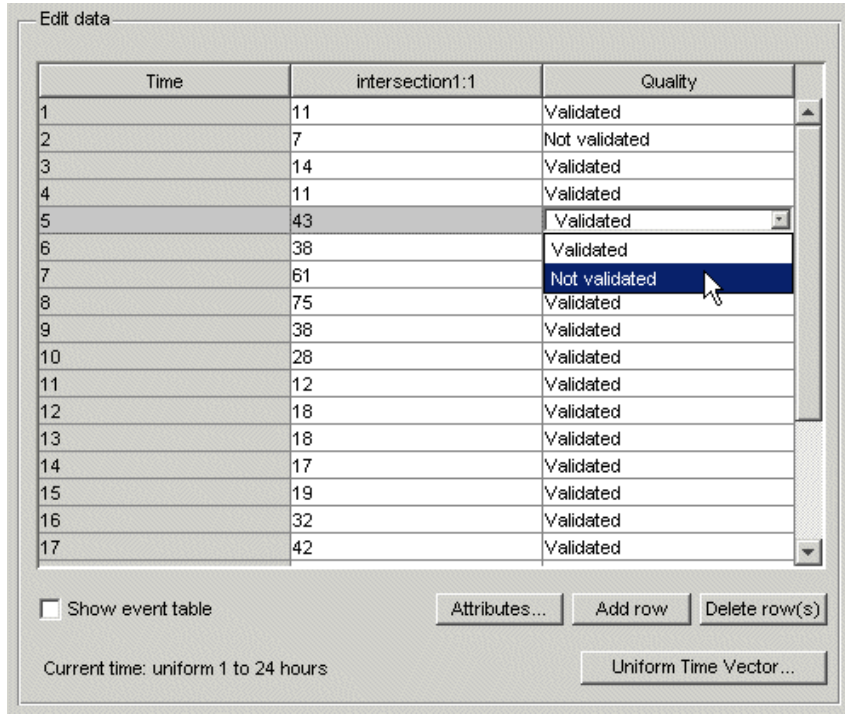


Note To delete a quality attribute, select it and click **Delete Code**.

Assigning Quality Codes to Data

After you define quality codes, as described in "Quality Codes" on page 5-35, the quality code you defined first is automatically assigned to all data values in the data table. For information about displaying the data table, see "Displaying the Data Table" on page 5-31.

To assign a different quality code to a specific data value, click the corresponding cell in the **Quality** column and select a different value from the drop-down list.



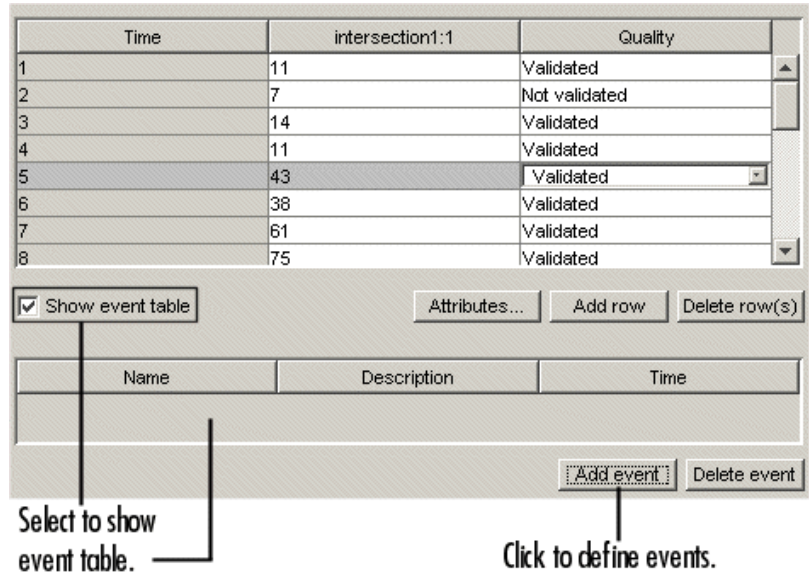
Defining Events

Events are stored as metadata for each time series. Time series events mark the data at a specific time in the data table and on a plot. For information about displaying the data table, see “Displaying the Data Table” on page 5-31.

You can also use events as reference points when shifting time series in time. For more information about synchronizing time series, see “Processing and Manipulating Time Series” on page 5-41.

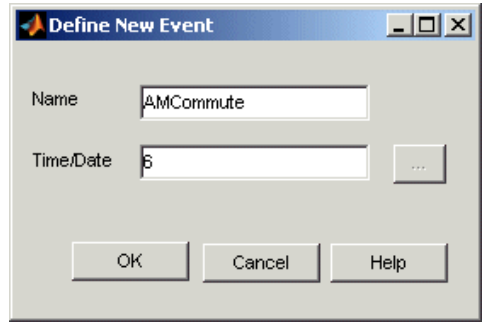
To define events for the selected time series:

1 Make sure that the **Show event table** check box is selected. This check box is located below the data table:



2 Click the **Add event** button below the event table. This opens the Define New Event dialog box.

3 In the **Name** field, enter the name of the event, such as `AMCommute`.

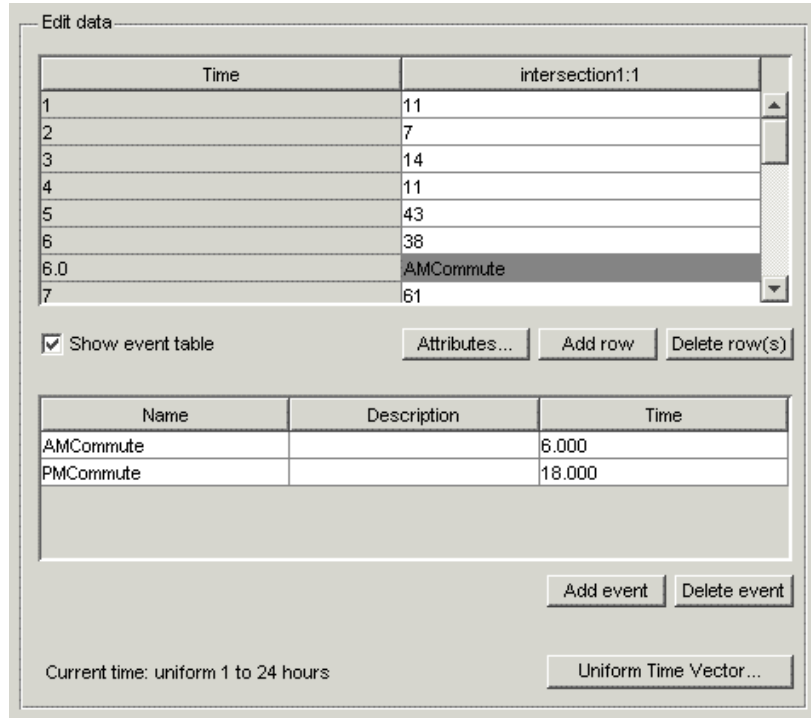


4 In the **Time/Date** field, enter or edit the time of the event in the appropriate display format. For information about time-vector formats, see “Time Vector Format” on page 4-21.

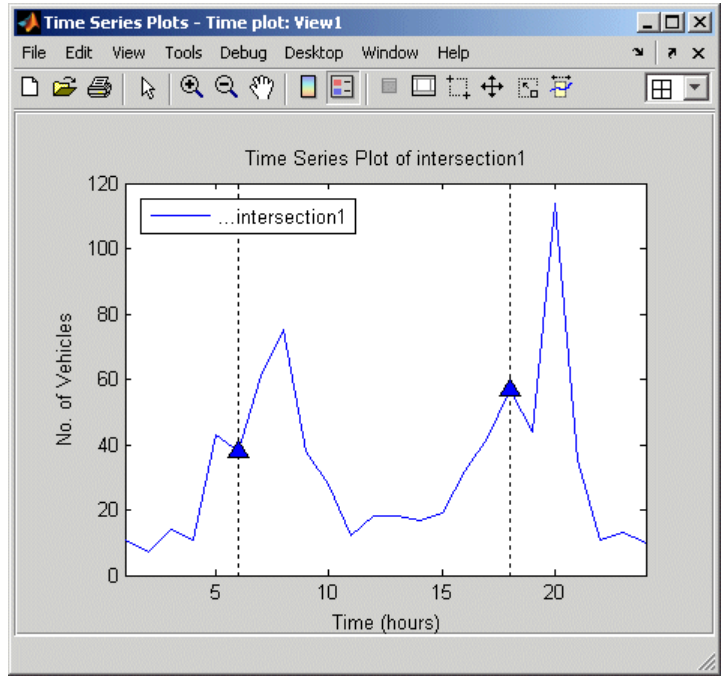
Tip To facilitate entering a date string, click the ... (**Browse**) button to open the Specify Date/Time dialog box. Select the month, year, and day. Then enter the **Time** in HH:MM:SS format.

5 Click **OK**.

The following figure shows two events in the event table: `AMCommute` and `PMCommute`. The data table also contains both events and `AMCommute` is shown at 6.0 hours.



Events are displayed as markers on time series plots. The following figure shows the `AMCommute` marker (at 6.0 hours) and `PMCommute` marker (at 18.0 hours) on a time plot.



Time Plot with Event Markers

Processing and Manipulating Time Series

The following table summarizes the operations you can perform on individual time series or time series collection. These commands are available from the **Data** menu in Time Series Tools after you select a time series or collection node in the Time Series Session tree.

Note If you are viewing a time plot, these operations are available by right-clicking inside the time plot and selecting a command from the shortcut menu. For more information about plotting data, see "Plotting Time Series" on page 5-13.

Each command opens a dialog box where you can get detailed instructions by clicking the **Help** button.

Data Analysis Commands

Command	Description
Data > Remove Missing Data	Delete the times that contain missing data.
Data > Detrend	Subtract a constant or a linear trend from the data.
Data > Filter	Smooth and shape the time series data.
Data > Interpolate	Interpolate missing values.
Data > Resample	Select or interpolate data values using a specified time vector.
Data > Transform Algebraically	Create a new time series by algebraically manipulating existing time series. This command is available only when you select an individual time series in the tree.
Data > Descriptive Statistics	Get summary statistics for each time series.

Example — Time Series Tools

In this section...

Loading Data into the MATLAB Workspace

Type the following command at the MATLAB prompt to load the hourly traffic counts at three road intersections, collected over a 24-hour period:

```
load count.dat
```

This adds the variable `count` to the MATLAB workspace.

Starting Time Series Tools

To start Time Series Tools, type

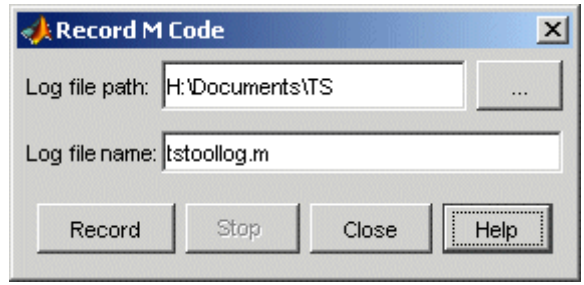
```
tstool
```

This opens the Time Series Tools window. For more information about this GUI, see "Time Series Tools Window" on page 5-3.

Enabling M-Code Generation

In this portion of the example, you will enable automatic M-code generation in Time Series Tools to capture reusable M-code as a MATLAB function.

1 In the Time Series Tools window, select **File > Record M-Code**. This opens the Record M-Code dialog box.



2 Click the ... button and select the folder where you want to store the M-file.

3 In the **Log file name** field, either select the name of a recently used file, or type a new name. The file name creates the function name you call in your M-code to reuse this function.

4 To begin capturing M-code, click **Record**. The M-code is recorded until you stop recording, as described in "Viewing Generated M-Code" on page 5-55.

Tip You can close this dialog box without interrupting the recording operation by clicking **Close**. To reopen the dialog box, select **File > Record M-Code** in the Time Series Tools window.

Note The scope of the **Record M-Code** feature is restricted to recording actions on the time series data itself. It does not generate code to import data or reproduce time series plots.

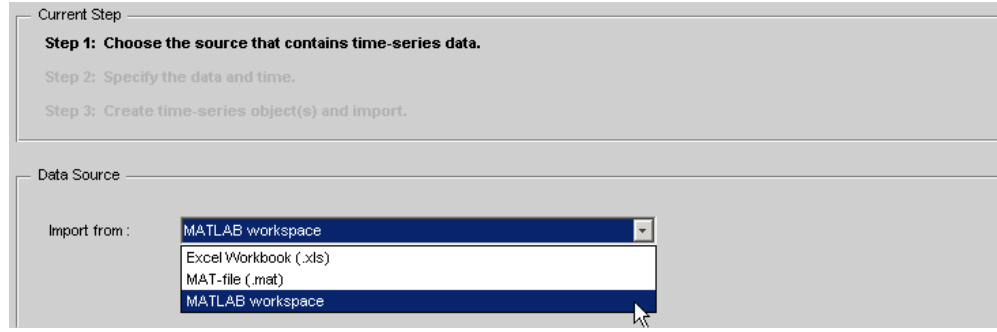
Importing Data into Time Series Tools

This portion of the example shows how to create three time series from the 24-by-3 count array you loaded into the MATLAB workspace.

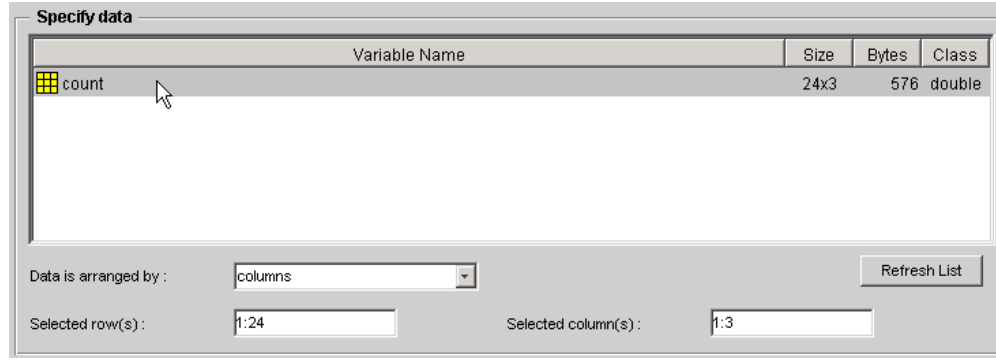
Note To get help on a specific field in the Import Wizard, right-click the field label and select **What's This** from the shortcut menu.

1 In the Time Series Tools window, select **File > Import from Workspace > Array Data**. This opens the Import Wizard.

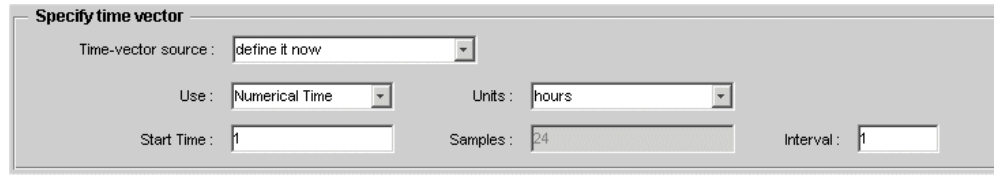
2 In the **Import from** list, select **MATLAB workspace** and click **Next**.



3 In **Step 2** of the Import Wizard, select the `count` variable. The Import Wizard infers from the data that it is arranged in columns.

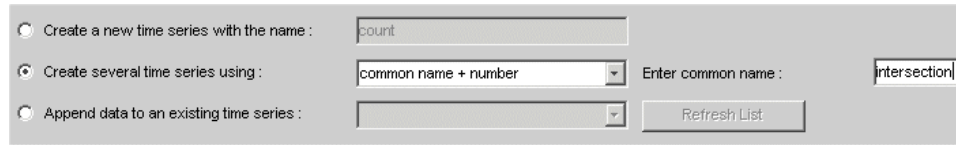


4 In the **Specify Time Vector** area, select `hours` from the **Units** list. In the **Start Time** field, type `1` to start the time vector at 1 hour. The Import Wizard has already filled in the remaining options to define a uniformly spaced time vector with a length of `24` and an interval of `1`.

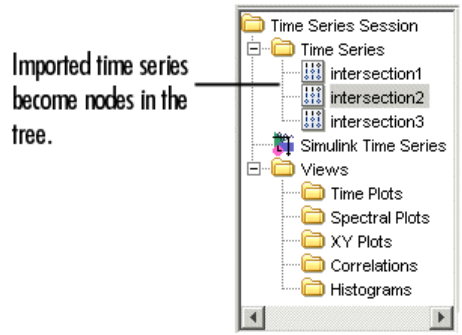


5 Click **Next**.

6 In **Step 3** of the Import Wizard, select **Create several time series using:** `common name+number`. In the **Enter common name** field, type `intersection`.



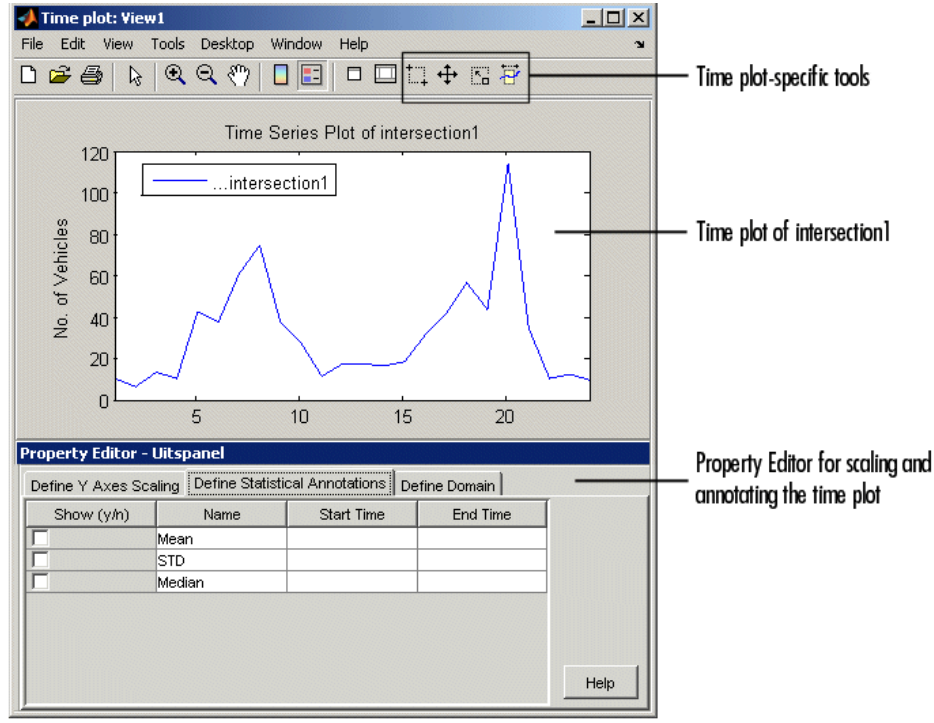
7 Click **Finish**. This adds three time series to the Time Series Session tree: `intersection1`, `intersection2`, and `intersection3` (as shown below).



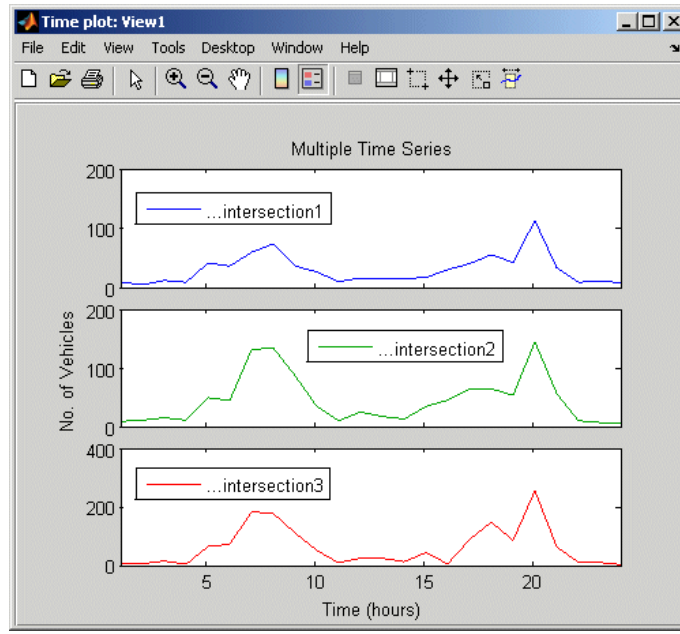
Creating a Time Plot

To explore the data, you can create a time plot of the three time series in the Time Series Tools window.

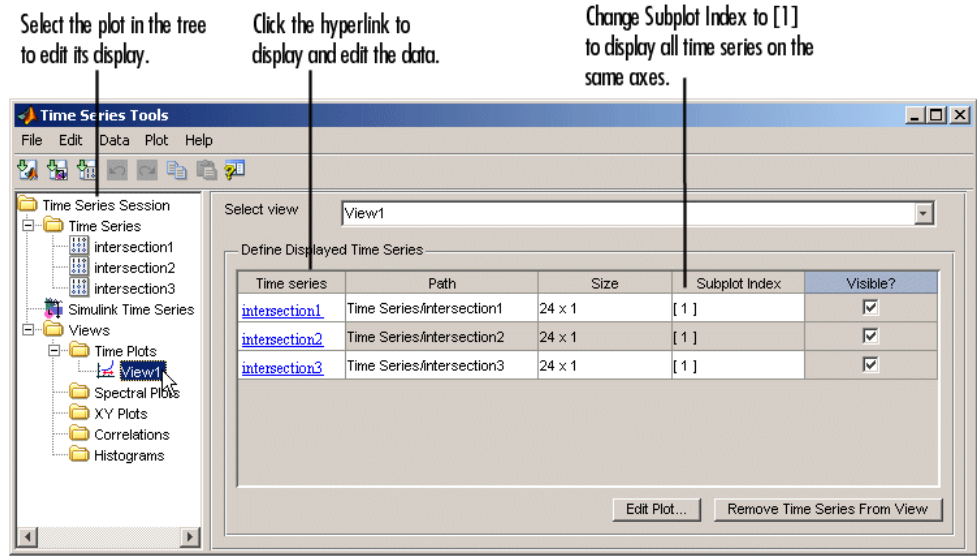
1 In the **Time Series Session** tree, drag and drop the `intersection1` time series into the **Time Plots** node. This creates a time plot in a new window with the default name **View1**.



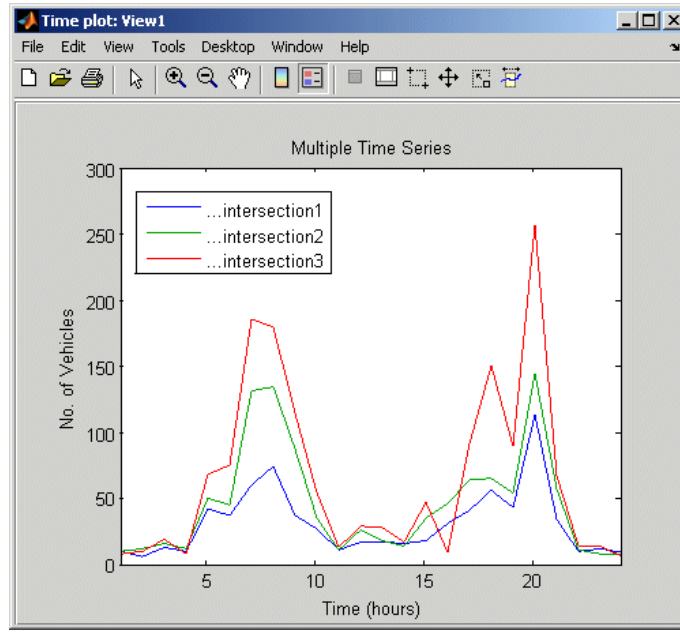
2 In the **Time Series Session** tree, drag and drop the `intersection2` and `intersection3` time series into **View1** to add them to the plot.



3 To display all three time series on the same axes, click the **View1** node in the Time Series Tools window. Change the subplot indices for `intersection2` and `intersection3` to `[1]` and press **Enter**.

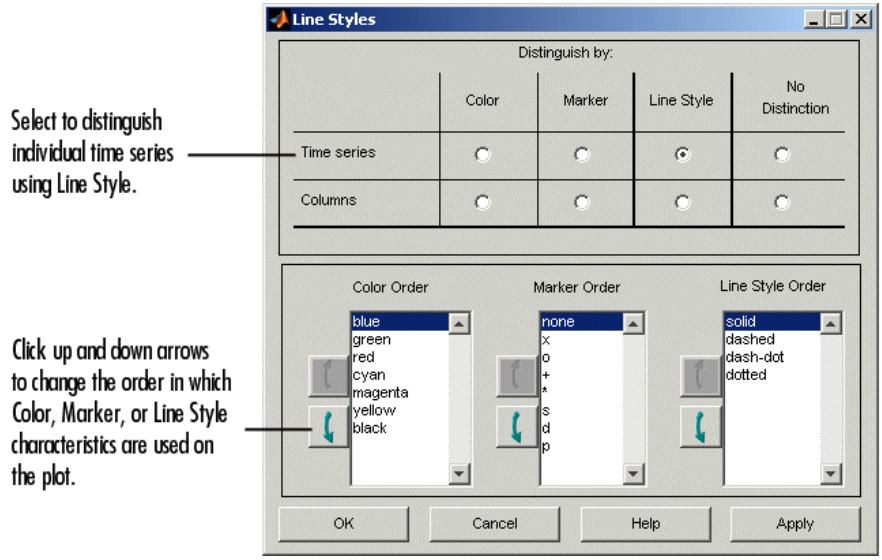


This displays all time series on the same axes, as follows:

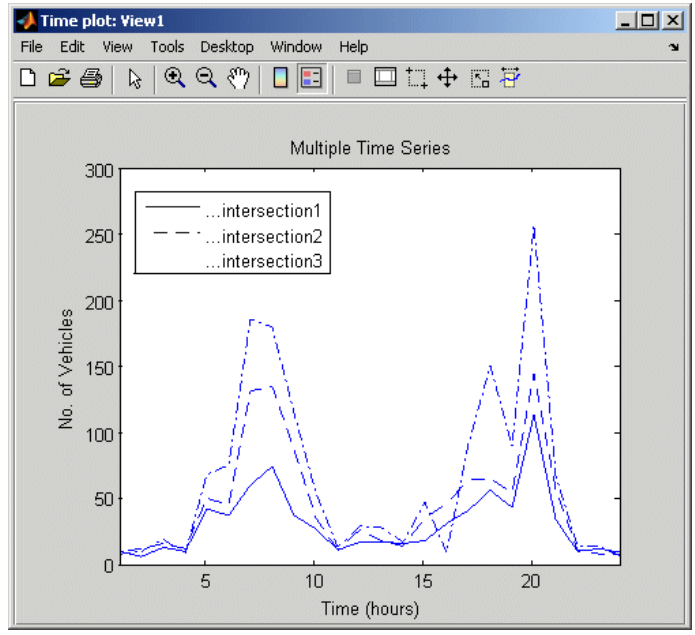


4 To change the appearance of the time series in the plot, go to the main Time Series Tools window and select **Plot > Set Line Properties**. This opens the Line Styles dialog box.

5 In the Line Styles dialog box, click **Line Style** to distinguish the time series, shown as follows.



The plot now looks like this.



Resampling Time Series

You can select or interpolate time series data using a specified time vector. When the new time vector contains time values that are not present in the original time vector, the intermediate data values are calculated using the interpolation method you associated with this time series. Linear interpolation is used by default. For more information about specifying the interpolation method, see “Defining Data Attributes” on page 5-34.

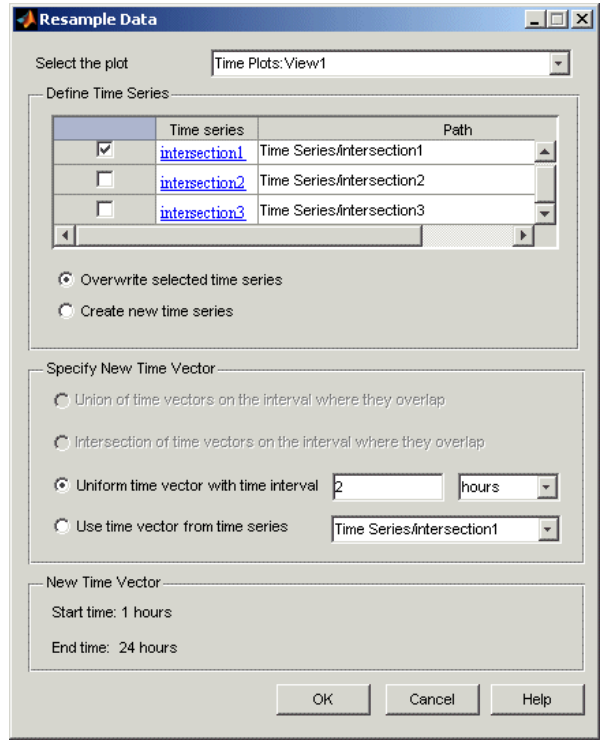
This portion of the example shows

Note You can only resample one time series at a time.

Resampling on a Uniform Time Vector

First, you resample the time series `intersection1` to include values every 2 hours.

1 Right-click inside the time plot you created in “Creating a Time Plot” on page 5-46 and select **Resample Data** from the shortcut menu. This opens the Resample Data dialog box.



2 In the **Define Time Series** area, select only `intersection1` and clear the rest.

3 In the **Specify New Time Vector** area, click **Uniform time vector with time interval** and specify the time interval as 2 hours. Click **OK**.

Tip To verify that `intersection1` is resampled, select it in the **Time Series Session** tree and examine the data table. It should have a time vector that starts at 1 hour and increases in increments of 2 hours.

Resampling by Finding a Common Time Vector

In some cases, you might want one time series to have the same time vector as another time series on the overlapping region of time values. This is especially useful when you want a specific time series to inherit a nonuniformly spaced time vector.

In this example, you resample `intersection2` on the same time vector as `intersection1`.

1. Right-click inside the time plot you created in "Creating a Time Plot" on page 5-46 and select **Resample Data** from the shortcut menu. This opens the Resample Data dialog box.

2. In the **Define Time Series** area, select only `intersection2` and clear the rest.

3. In the **Specify New Time Vector** area, click **Use time vector from time series** and select `intersection1` from the list. Click **OK**.

To verify that `intersection2` is resampled, select it in the **Time Series Session** tree and examine the data table. It should have a time vector that starts at 1 hour and increases in increments of 2 hours.

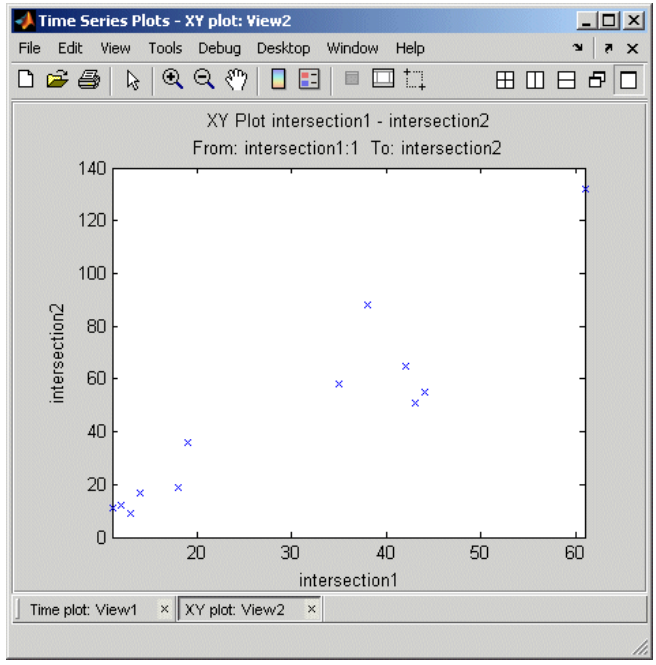
Comparing Data on an *XY* Plot

The *XY* plot is useful for visually determining a relationship between the data values of time series at corresponding times. For example, when the points on an *XY* plot form a straight line, there is a linear relationship between the two time series.

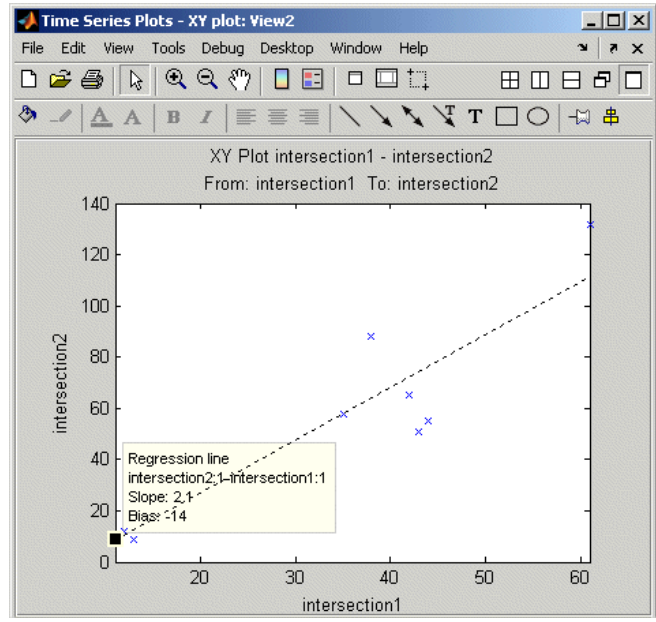
In this portion of the example, you examine the relationship between the corresponding data values of `intersection1` and `intersection2` by using an *XY* plot.

1. In the Time Series Session tree, drag and drop the **intersection1** time series into the **XY Plots** node. This creates a new plot node with the default name **View2**.

2 Drag and drop the **intersection2** time series into the **View2** node. This creates the following *XY* plot.



3 To show the best-fit line on the *XY* plot, click the **Define Statistical Annotations** tab in the Property Editor and select the **Best fit line** check box. Then, click the line to display the line equation on the plot.



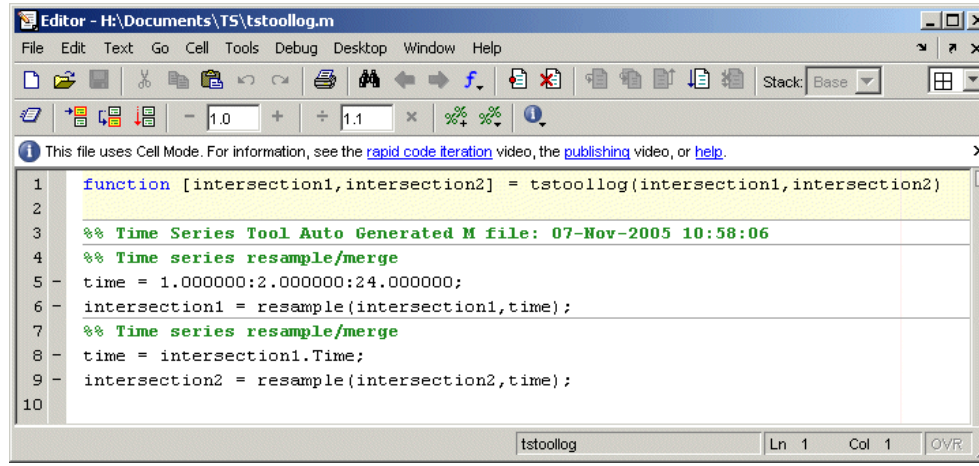
Viewing Generated M-Code

You can now view the M-code that Time Series Tools generated while you performed the previous steps in this example.

To view the M-file:

1 In the Time Series Tools window, select **File > Record M-Code** to open the Record M-Code dialog box.

2 Click **Stop** to open the M-file with the generated M-code in the MATLAB Editor.



Automatically Generated M-Code

You can reuse this M-code by calling the `tstoollog` function, which has the same name as this M-file. You specified the file name when you enabled M-code generation in this example, as described in “Enabling M-Code Generation” on page 5-42.

Examine the code of the `tstoollog` function to confirm that it takes two time series as input arguments and resamples them using a uniform time vector with the range 1 to 24 and intervals of 2.

Note The scope of the **Record M-Code** feature is restricted to recording actions on the time series data itself. It does not generate code to import data or reproduce time series plots.

Exporting Time Series to the Workspace

You can export individual time series, as well as time series collections, from Time Series Tools to the MATLAB workspace. You can also export time series to a Microsoft Excel worksheet or a MAT-file.

In this portion of the example, you will export the time series `intersection1` as a variable to the MATLAB workspace. This time series differs from the original data you imported into Time Series Tools because it has been resampled, as described in "Resampling Time Series" on page 5-51.

1. Click the **interesection1** node in the Time Series Session tree to select it.
2. Select **File > Export > To Workspace**. The variable `intersection1` is now listed in the MATLAB workspace.

Note If the MATLAB workspace is hidden, select **Desktop > Workspace** from the MATLAB window to display it.

Index

Q

R

S

T

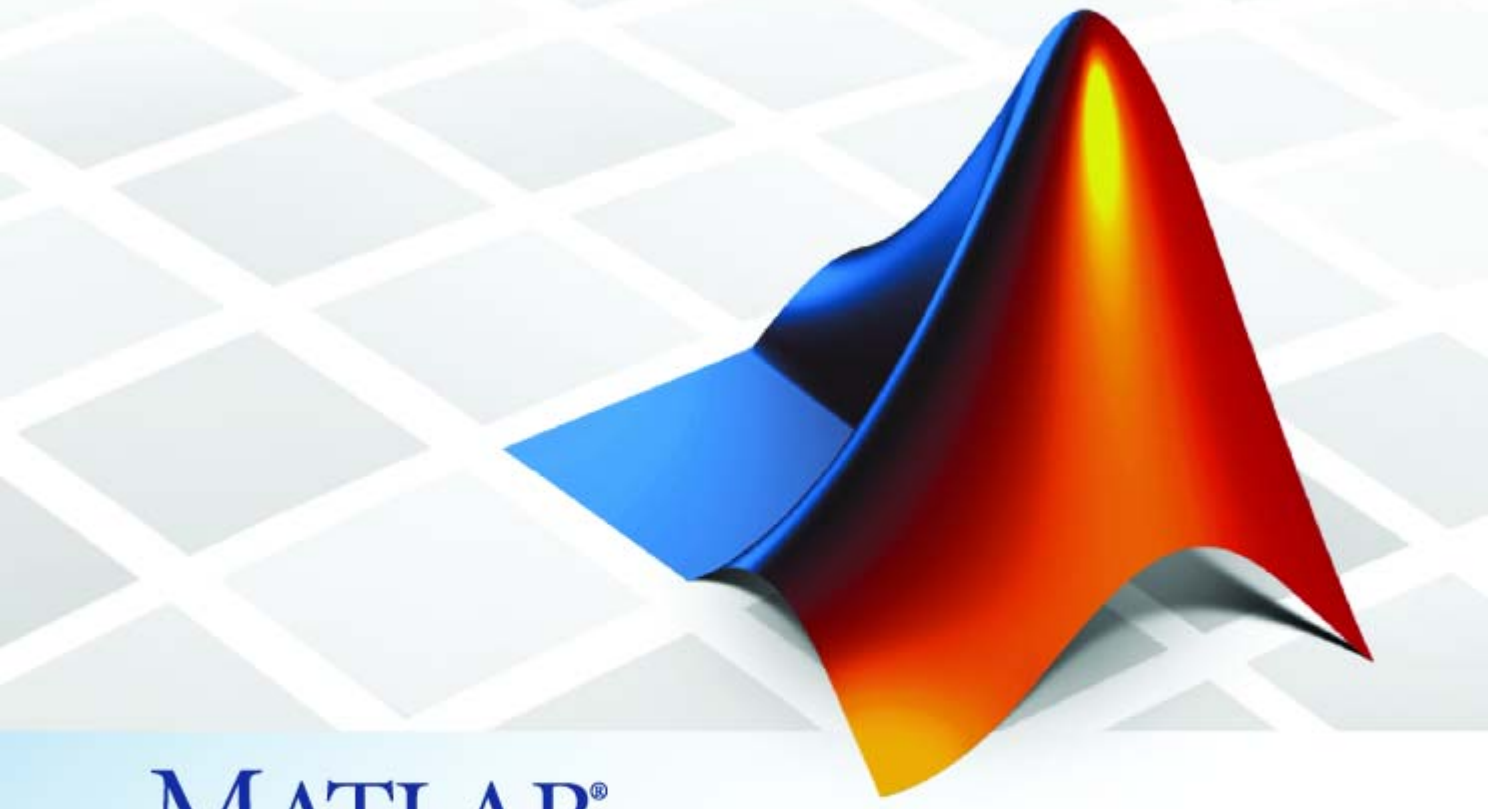
U

V

W

X

MATLAB® 7
Desktop Tools and Development Environment



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Desktop Tools and Development Environment

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are 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)
March 2005	Second printing	Revised for MATLAB 7.0.4 (Release 14SP2)
June 2005	Third printing	Minor revision for MATLAB 7.0.4 (Release 14SP2)
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b)

Contents

1 Startup and Shutdown

2 Desktop

3 Running Functions — Command Window and History

Help for Using MATLAB

4

5 Workspace, Search Path, and File Operations

6 Editing and Debugging M-Files

7 Tuning and Managing M-Files

8 Publishing Results

9 Source Control Interface

Index

1

Startup and Shutdown

This set of topics includes options for customizing the startup and shutdown.

Starting MATLAB on Windows Platforms (p. 1-2)	Ways to start MATLAB®, including from a desktop icon, or from a file in Windows Explorer. Associating file types with MATLAB.
Starting MATLAB on UNIX Platforms (p. 1-6)	MATLAB startup on UNIX.
Starting MATLAB on Macintosh Platforms (p. 1-7)	Ways to start MATLAB on a Macintosh, including from a MATLAB file type in the Finder.
Startup Directory for MATLAB (p. 1-8)	Ways to change the directory in which MATLAB starts.
Startup Options (p. 1-12)	Instruct MATLAB to perform operations upon startup via a `startup` file or the `matlab` function.
Toolbox Path Caching in MATLAB (p. 1-17)	About the cache file and updating the cache file.
Other Startup Topics (p. 1-21)	Error Log Reporter, passing Perl variables, and calling Java from MATLAB.
Quitting MATLAB (p. 1-23)	End a MATLAB session. Instruct MATLAB to perform specified operations upon shutdown.

Starting MATLAB on Windows Platforms

In this section...

Starting MATLAB from the Windows Desktop or a DOS Window

To start MATLAB on a Microsoft Windows platform, select the **Start > Programs > MATLAB > R2007b > MATLAB R2007b**, or double-click the MATLAB R2007b shortcut icon on your Windows desktop. The shortcut was automatically created when you installed MATLAB. If you have trouble starting MATLAB, see troubleshooting information in the Installation Guide for Windows.

To start MATLAB from a DOS window, `cd` to the directory in which you want to start MATLAB and type `matlab` at the DOS prompt.

After starting MATLAB, the desktop opens. Desktop components that were open when you last shut down MATLAB will be opened on startup. For more information, see Chapter 2, "Desktop".

Starting MATLAB from an M-File or Other File Type in Windows

On Windows platforms, you can start MATLAB from a Windows Explorer window by double-clicking a file with one of these extensions: `.fig`, `.m`, `.mat`, and `.mdl`. MATLAB starts and opens in an appropriate tool. If MATLAB is already running, the file opens in an appropriate tool in the existing session.

This startup feature is based on your Windows file type associations. When you installed MATLAB for Windows, you specified the file types to

associate with MATLAB. For example, if you accepted the default options, double-clicking an M-file in Windows Explorer opens the file in the MATLAB Editor/Debugger.

Other default options associate MEX-files and P-files with MATLAB in Windows Explorer, which assigns the file types a MATLAB icon. However, double-clicking a file with a `.mex` (`.mexw32` or `.mexw64`), or `.p` extension does not run or open the file in MATLAB.

File Type and Resulting Action

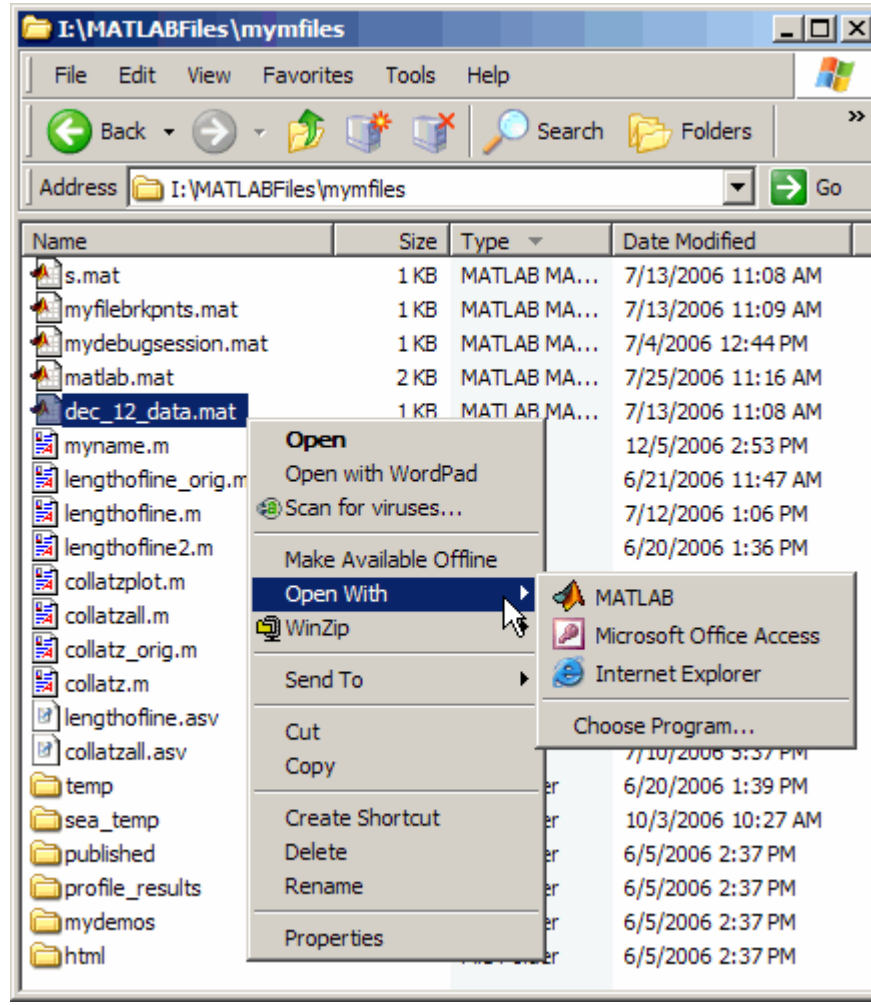
File Type	Result
FIG-file	Opens file in figure window
M-file	Opens file in Editor/Debugger
MAT-file	Opens Import Wizard to load the data into the MATLAB workspace
MDL-file	Opens file in Simulink® model window
MEX-file	Displays MATLAB icon in Windows Explorer
P-file	Displays MATLAB icon in Windows Explorer

Other applications you use can change Windows file associations. For example, Microsoft Access might associate files having a `.mat` extension. Then, double-clicking a MAT-file opens Access rather than MATLAB.

If you double-click a FIG-file, M-file, MAT-file, or MDL-file and it does not open in MATLAB, try this instead:

1. In Windows Explorer, right-click a file with one of the extensions listed in the preceding table, for example, `myfile.mat`.
2. From the context menu, select **Open With**. If MATLAB is one of the choices, select it to open `myfile.mat` in MATLAB. If MATLAB is not one of the choices, you will need to associate the file type with MATLAB using one of these techniques:
   - “Utility to Change Windows File Associations” on page 1-5
   - “Changing File Associations for MATLAB from Windows” on page 1-5

After associating a file type with MATLAB, you can open other applications using that file type via the context menu. For example, right-click `myfile.mat`, and from the context menu, select **Open With**. Microsoft Access will be one of the options.



File associations for Windows Explorer do not affect what happens when you open one of these file types from *within* MATLAB. MATLAB acts on the file

using the MATLAB tool associated with that file type. For example, even if you associate `.mat` files with Microsoft Access, when you open a MAT-file from within MATLAB, it opens the Import Wizard to load the data.

Utility to Change Windows File Associations

If you are viewing this topic in the MATLAB Help browser, you can run one of the utilities provided here to create Windows associations for common MATLAB file types. This requires you to have permission to write to the `HKEY_CLASSES_ROOT` registry key, which typically requires power user or administrator privileges.

- Run utility to associate MATLAB with FIG-files
- Run utility to associate MATLAB with M-files
- Run utility to associate MATLAB with MAT-files
- Run utility to associate MATLAB with MDL-files
- Run utility to associate MATLAB with MEX-files
- Run utility to associate MATLAB with P-files
- Run utility to associate MATLAB with all of these file types: FIG, M, MAT, MDL. MEX, and P

The file type icon in Windows Explorer might not reflect the change immediately.

Changing File Associations for MATLAB from Windows

You can associate file types with MATLAB from Windows Explorer. This is useful if you want associate file types other than those offered by the above utilities. For example, you can double-click `.xml` files to open them in the MATLAB Editor/Debugger.

The following examples show one way to change file associations in Windows. Note that these instructions might not exactly apply to your version of Windows. If you encounter differences or problems, try to delete the association before using these instructions, or see your Windows documentation.

Assume that when you double-click a `.mat` file in Windows Explorer, it opens in Microsoft Access, but you want the file to open in MATLAB.

1. In Windows Explorer, select **Tools > Folder Options**.
2. In the resulting Folder Options dialog box, select the **File Types** tab. From the **Registered file types** list, select the `MAT` extension. (If you do not see `MAT` in the list, click **New** to add it.)

   Under **Details for 'MAT' extension**, click **Change**.
3. In the resulting Open With dialog box, select `MATLAB` from the list.

   If MATLAB is not listed, click **Browse**. Then look for and select `matlab.exe`, and click **Open**. The file is located in the folder in which you installed MATLAB. An example of the default location is `C:\Program Files\MATLAB\R2007b\bin`.
4. In the Open With dialog box, click **OK**. In the Folder Options dialog box, click **Close**.

Starting MATLAB on UNIX Platforms

To start MATLAB on a UNIX platform, type `MATLAB` at the operating system prompt.

If you did not set up symbolic links in the installation procedure, you must enter the full pathname to start MATLAB, *`matlabroot`*`/bin/matlab`, where *`matlabroot`* is the name of your MATLAB installation directory. If you have trouble starting MATLAB, see troubleshooting information in the Installation Guide for UNIX.

After starting MATLAB, the desktop opens. Desktop components that were open when you last shut down MATLAB will be opened on startup. For more information, see Chapter 2, "Desktop". If the `DISPLAY` environment variable is not set or is invalid, the desktop will not display.

Starting MATLAB on Macintosh Platforms

In this section...

Starting MATLAB from the Macintosh Desktop

To start MATLAB on Macintosh platforms, double-click the MATLAB icon on the desktop. If there isn't a MATLAB icon on the desktop, you can find it in the MATLAB folder within the Applications folder. MATLAB starts, assuming you accepted the defaults for the license manager to automatically start at system startup.

Note On Macintosh systems, you cannot perform a remote login, that is, you cannot run MATLAB remotely. For example, you cannot `rlogin`.

Starting MATLAB from an M-File or Other File Type on Macintosh Platforms

On Macintosh platforms, you can start MATLAB from the Finder by double-clicking a file with one of these extensions: `.fig`, `.m`, `.mat`, and `.mdl`. MATLAB starts and opens in an appropriate tool. If MATLAB is already running, the file opens in an appropriate tool in the existing session—see File Type and Resulting Action on page 1-3.

Startup Directory for MATLAB

In this section...

What Is the Startup Directory?

The *startup directory* is the current directory in MATLAB when it starts. The default startup directory depends on your platform and installation. You can specify a different startup directory.

Startup Directory (Folder) on Windows Platforms

When you start MATLAB from a Windows shortcut icon, MATLAB sets the default startup directory to `My Documents\MATLAB` or `Documents\MATLAB` on Vista. This is a location where you can conveniently work with your personal MATLAB files. It offers these features:

- Because the `MATLAB` subfolder is the default startup directory, the current directory is automatically the one containing the files your work with.
- So that the files you work with are on the search path, MATLAB automatically adds the `MATLAB` subfolder to the top of the search path upon startup. If the `MATLAB` subfolder does not exist, MATLAB creates it.
- This location also utilizes the benefits provided by the Windows (or Vista) standard location for storing personal files, `My Documents` (or `Documents`). Files in the `MATLAB` subfolder are available to you when you use other machines, if your profile is set to roam. Because each user has their own `My Documents` or `Documents` folder, other users, even those using your machine, cannot access files in your `MATLAB` subfolder.
- When you upgrade to a newer version of MATLAB, MATLAB automatically continues to use the same `MATLAB` subfolder and your existing files, with all of the above benefits.

Startup Directory on UNIX Platforms

On UNIX platforms, the default startup directory is the directory you are in on your UNIX file system when you start MATLAB, for example, `/home/$user/matlab`. MATLAB automatically adds this `matlab` directory to the top of the MATLAB search path upon startup.

Changing the Startup Directory

You can start MATLAB in a directory other than the default.

For Windows Platforms Only

To change the startup directory on Windows platforms,

1. Right-click the MATLAB shortcut icon and select **Properties** from the context menu.

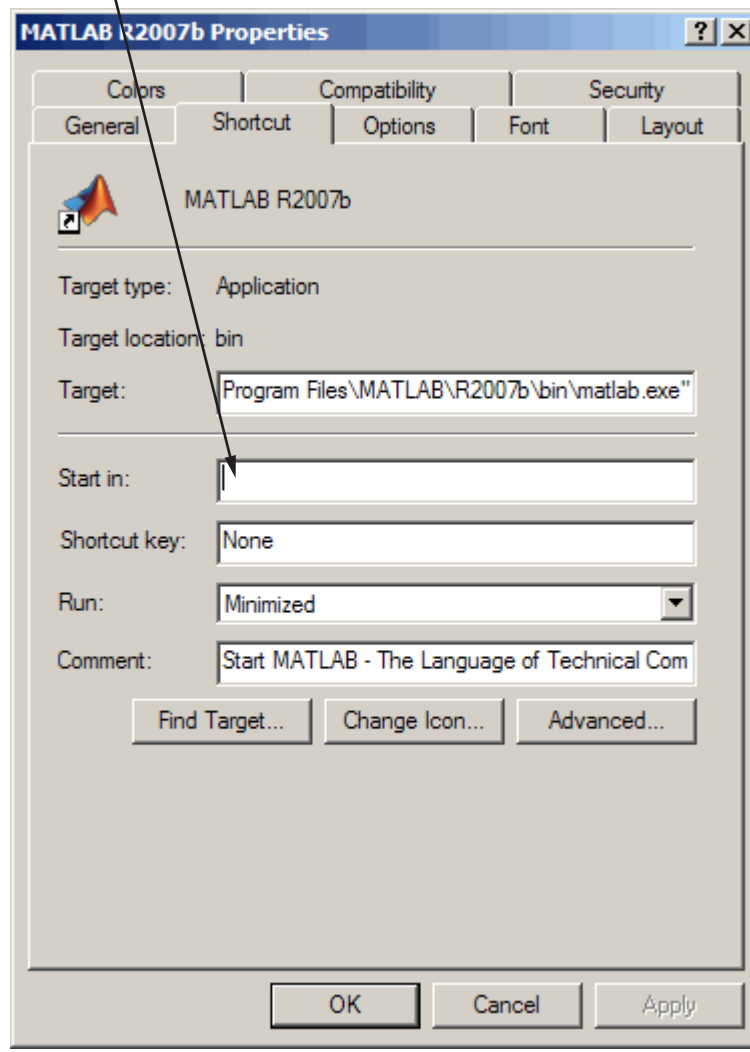
   The Properties dialog box for MATLAB opens to the **Shortcut** pane.

2. The **Target** field contains the full path to start MATLAB.

   By default, the startup directory is `My Documents\MATLAB` or `Documents\MATLAB` on Vista; for more information, see “Startup Directory (Folder) on Windows Platforms” on page 1-8.

   In the **Start in** field, specify the full path to the directory in which you want MATLAB to start, and click **OK**.

In the **Start in** field, enter the full path to the directory in which you want MATLAB to the start. For example, `I:\my_matlab_files`.



The next time you start MATLAB using that shortcut icon, the current directory will be the one you specified in step 2.

You can make multiple shortcuts to start MATLAB, each with its own startup directory, and with each startup directory having different startup options.

For All Platforms

To change the startup directory,

1. Create a `startup.m` file — see "Using the Startup File for MATLAB, startup.m" on page 1-12.
2. In the `startup.m` file, include the `cd` function to change to the new directory.
3. Put the `startup.m` file in the default startup directory for your platform, as noted in "Startup Directory (Folder) on Windows Platforms" on page 1-8 or "Startup Directory on UNIX Platforms" on page 1-9. Alternatively, put `startup.m` in the current startup directory.

Startup Options

In this section...

About Startup Options

You can define startup options that instruct MATLAB to perform certain operations when you start it. You specify these options using a startup file (`startup.m`) or as options to the `matlab` startup command.

Using the Startup File for MATLAB, startup.m

At startup, MATLAB automatically executes the master M-file `matlabrc.m` and, if it exists, `startup.m`. The file `matlabrc.m`, which is in the *`matlabroot`*`/toolbox/local` directory, is reserved for use by The MathWorks and by the system manager on multiuser systems.

The file `startup.m` is for you to specify startup options. For example, you can modify the default search path, predefine variables in your workspace, or define Handle Graphics® defaults. Creating a `startup.m` file with the lines

```
addpath /home/$user/mytools
cd /home/$user/mytools
```

adds `/home/`*`$user`*`/mytools` to your default search path and makes `mytools` the current directory upon startup.

Location of startup.m

Place the `startup.m` file in the default or current startup directory, which is where MATLAB first looks for it. For more information, see "Startup Directory for MATLAB" on page 1-8.

Adding Startup Options for Windows Platforms

You can add selected startup options (also called command flags or switches for the command line) to the target path for your Windows shortcut for MATLAB. Or you can add them to the command line when you start MATLAB in a DOS window. On Windows systems a startup option is preceded by either a hyphen (-) or a slash (/). For example, `-nosplash` and `/nosplash` are equivalent ways of specifying the `nosplash` option for Windows users.

Startup Options in Windows Shortcut

To use startup options in the Windows shortcut for MATLAB, follow these steps:

1. Right-click the MATLAB shortcut icon and select **Properties** from the context menu. The Properties dialog box for MATLAB opens to the **Shortcut** pane.
2. In the **Target** field, after the target path for `matlab.exe`, add the startup option, and click **OK**. For example, adding `-r "filename"` runs the M-file `filename` after startup.

This example instructs MATLAB to automatically run the file `results` after startup, where `results.m` is in the startup directory or on the MATLAB search path. The statement in the **Target** field might appear as

```
C:\Program Files\MATLAB\bin\matlab.exe -r "results"
```

Include the statement in double quotation marks (`"`*`statement`*`"`). Use only the filename, not the file extension or pathname. For example, MATLAB produces an error when you run

```
... matlab.exe -r "D:\results.m"
```

Use semicolons or commas to separate multiple statements. This example changes the format to `short`, and then runs the M-file `results`:

```
... matlab.exe -r "format('short');results"
```

Separate multiple options with spaces. This example starts MATLAB without displaying the splash screen, and then runs the M-file `results`:

```
... matlab.exe -nosplash -r "results"
```

Startup Options in DOS Window

When you start MATLAB in a DOS window, include startup options after the `matlab` command.

This example uses the `nosplash` startup option to start MATLAB without the splash screen, and adds the `-r` option to run the `results` function located in the startup directory, after starting MATLAB in a DOS window:

```
matlab -nosplash -r "results"
```

Adding Startup Options for UNIX Platforms

Include startup options (also called command flags or command line switches) after the `matlab` command. On UNIX systems a startup option is preceded by a hyphen (-). For example, to start MATLAB without the splash screen, type

```
matlab -nosplash
```

Commonly Used Startup Options

The following table provides a list of some commonly used startup options for both Windows and UNIX platforms. For more information, including a complete list of startup options, see the `matlab (Windows)` reference page or the `matlab (UNIX)` reference page.

Platform	Option	Description
All	`-c licensefile`	Set `LM_LICENSE_FILE` to `licensefile`. It can have the form `port@host`.
All	`-h or -help`	Display startup options (without starting MATLAB).
All	`-logfile "logfilename"`	Automatically write output from MATLAB to the specified log file.

Platform	Option	Description
UNIX	`-nodesktop`	Start MATLAB without bringing up the MATLAB desktop. Use this option to run without an X-window, for example, in VT100 mode, or in batch processing mode. Note that if you pipe to MATLAB using the > constructor, the `nodesktop` option is used automatically. With `nodesktop`, you can still use most development environment tools by starting them via a function. For example, use `preferences` to open the Preferences dialog box and `helpbrowser` to open the Help browser. Do not use `nodesktop` to provide a command-line interface. If you prefer a command-line interface, select **Desktop > Desktop Layout > Command Window > Only**.
Windows	`-minimize`	Start MATLAB with the desktop minimized. Any desktop tools or documents that were undocked when MATLAB was last closed will not be minimized upon startup.
UNIX	`-nojvm`	Start MATLAB without loading the Java VM. This minimizes memory usage and improves initial startup speed, but restricts functionality. With `nojvm`, you cannot use the desktop, or any tools that require Java. For example, you cannot set preferences if you start MATLAB with the `-nojvm` option. However, you can start MATLAB once *without* the `-nojvm` option, set the preference, and quit MATLAB. MATLAB remembers that preference when you start it again, even if you use the `-nojvm` option.

Platform	Option	Description
All	`-nosplash`	Start MATLAB without displaying the MATLAB splash screen.
All	`-r "statement"`	Automatically run the specified MATLAB statement immediately after MATLAB starts. This is sometimes referred to as calling MATLAB in batch mode. Files you run must be in the MATLAB startup directory or on the MATLAB search path. Do not include pathnames or file extensions. Enclose the statement in double quotation marks (`"statement"`).

Toolbox Path Caching in MATLAB

In this section...

About Toolbox Path Caching

For performance reasons, MATLAB caches toolbox directory information across sessions. The caching features are mostly transparent to you. However, if MATLAB does not see the latest versions of your M-files or if you receive warnings about the toolbox path cache, you might need to update the cache.

Using the Cache File Upon Startup

Upon startup, MATLAB gets information from a cache file to build the toolbox directory cache. Because of the cache file, startup is faster, especially if you run MATLAB from a network server or if you have many toolbox directories. When you end a session, MATLAB updates the cache file.

MATLAB does not use the cache file at startup if you clear the **Enable toolbox path cache** check box in **File > Preferences > General**. Instead, it creates the cache by reading from the operating system directories, which is slower than using the cache file.

Updating the Cache and Cache File

How the Toolbox Path Cache Works

MATLAB caches (essentially, stores in a known files list) the names and locations of files in *`matlabroot`*`/toolbox` directories. These directories are for MathWorks supplied files that should not change except for product installations and updates. Caching those directories provides better performance during a session because MATLAB does not actively monitor those directories.

We strongly recommend that you save any M-files you create and any MathWorks supplied M-files that you edit in a directory that is *not* in the *matlabroot*/toolbox directory tree. If you keep your files in *matlabroot*/toolbox directories, they may be overwritten when you install a new version of MATLAB.

When to Update the Cache

When you add files to *matlabroot*/toolbox directories, the cache and the cache file need to be updated. MATLAB updates the cache and cache file automatically when you install toolboxes or toolbox updates using the MATLAB installer. MATLAB also updates the cache and cache file automatically when you use MATLAB tools, such as when you save files from the MATLAB Editor/Debugger to *matlabroot*/toolbox directories.

When you add or remove files in *matlabroot*/toolbox directories by some other means, MATLAB might not recognize those changes. For example, when you

- Save new files in *matlabroot*/toolbox directories using an external editor
- Use operating system features and commands to add or remove files in *matlabroot*/toolbox directories

MATLAB displays this message:

```
Undefined function or variable
```

You need to update the cache so MATLAB will recognize the changes you made in *matlabroot*/toolbox directories.

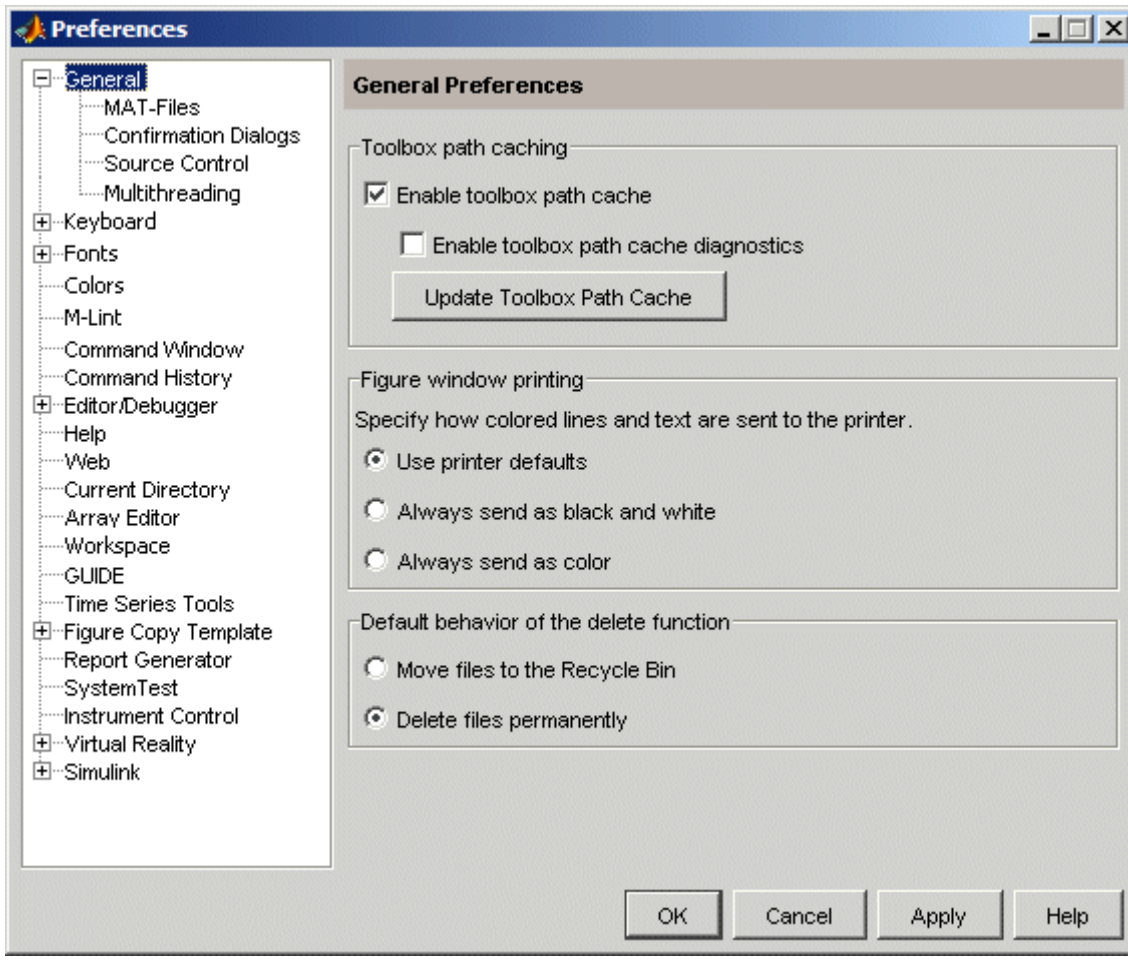
Steps to Update the Cache

To update the cache and the cache file,

1. Select **File > Preferences > General**.

   The **General Preferences** pane is displayed.

2. Click **Update Toolbox Path Cache** and click **OK**.



Function Alternative

To update the cache, use `rehash toolbox`. To also update the cache file, use `rehash toolboxcache`. For more information, see `rehash`.

Additional Diagnostics with Toolbox Path Caching

To display information about startup time when you start MATLAB, select the **Enable toolbox path cache diagnostics** check box in **General Preferences**.

Other Startup Topics

In this section...

Error Log Reporter

Upon startup, if MATLAB detects an error log generated by a serious problem encountered during the *previous* session, an Error Log Reporter prompts you to e-mail the log to The MathWorks for analysis. Click **Send Report** to e-mail the log, or click **Help** for more information. After sending the log, a confirmation message appears in the Command Window. For more information, see “Abnormal Termination” on page 1-24.

Passing Perl Variables on Startup

You can pass Perl variables to MATLAB on startup by using the `-r` option of the `matlab` function. For example, assume a MATLAB function `test` that takes one input variable:

```
function test(x)
```

To start MATLAB with the function `test`, use the command

```
matlab -r "test(10)"
```

On some platforms, you might need to use double quotation marks:

```
matlab -r "test(10)"
```

This command starts MATLAB and runs `test` with the input argument `10`.

To pass a Perl variable instead of a constant as the input parameter, follow these steps.

1 Create a Perl script such as

```
#!/usr/local/bin/perl
$val = 10;
system('matlab -r "test(' . ${val} . ')"');
```

2 Invoke the Perl script at the prompt using a Perl interpreter.

For more information, see the `matlab (Windows)` or `matlab (UNIX)` reference page.

Startup and Calling Java from MATLAB

When MATLAB starts, it constructs the Java class path using `librarypath.txt` as well as `classpath.txt`. If you call Java from MATLAB, see more about this in “The Java Class Path” and “Locating Native Method Libraries” in the MATLAB External Interfaces documentation.

Quitting MATLAB

In this section...

Ways to Quit MATLAB

To quit MATLAB at any time, do one of the following:

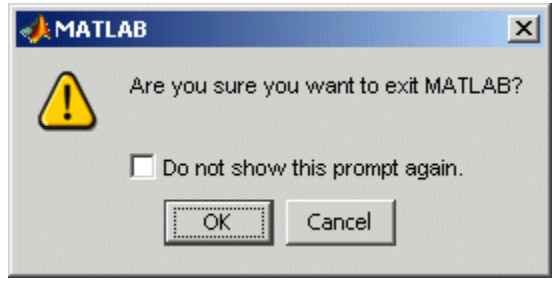
- Click the Close box in the MATLAB desktop.
- Select **Exit MATLAB** from the desktop **File** menu.
- Type `quit` at the Command Window prompt.

MATLAB closes after

- Prompting you to confirm quitting, if that preference is specified (see "Confirm Quitting MATLAB" on page 1-23)
- Prompting you to save any unsaved files
- Running the `finish.m` script, if it exists in the current directory or on the MATLAB path (see "Running a Script When Quitting MATLAB" on page 1-24)

Confirm Quitting MATLAB

To set a preference that displays a confirmation dialog box when you quit MATLAB, select **File > Preferences > General > Confirmation Dialogs**, select the **Confirm before quitting** check box, and click **OK**. MATLAB then displays the following dialog box when you quit.



For more information, see “Confirmation Dialogs Preferences” on page 2-81.

You can also display your own quitting confirmation dialog box using a `finish.m` script, as described in the following section.

Running a Script When Quitting MATLAB

When MATLAB quits, it runs the script `finish.m`, if `finish.m` exists in the current directory or anywhere on the MATLAB search path. You create the file `finish.m`. It contains statements to run when MATLAB terminates, such as saving the workspace or displaying a confirmation dialog box. There are two sample files in *`matlabroot`*`/toolbox/local` that you can use as the basis for your own `finish.m` file:

- `finishsav.m` — Includes a `save` function so the workspace is saved to a MAT-file when MATLAB quits.
- `finishdlg.m` — Displays a confirmation dialog box that allows you to cancel quitting.

For more information, see the `finish` reference page.

Abnormal Termination

In the event MATLAB experiences a segmentation violation (segv) or another serious problem, save your files and workspace if possible, exit MATLAB, and restart.

Upon startup, if MATLAB detects an error log generated by a serious problem during the *previous* session, an Error Log Reporter prompts you to e-mail the log to The MathWorks for analysis. If the problem occurs repeatedly,

make note of what seems to cause it and look for information about it in the MathWorks Bug Reports database.

There are some situations where the Error Log Reporter will not open, for example, when you start MATLAB with a `-r` option or run in deployed mode. If you experience segmentation violations but do not see the Error Log Reporter on subsequent startups, you can instead e-mail the reports by following the instructions at the end of the segmentation violation message in the Command Window.

Crash Recovery and Multithreading

When multithreaded computation is enabled, if MATLAB experiences a segmentation violation or other serious problem, it cannot try to return control to the Command Window. You do not have an opportunity to view a segmentation violation message in the Command Window as you might when multithreaded computation is *not* enabled. Instead, your platform's vendor, for example, Microsoft or Apple, provides an error dialog box. MATLAB then terminates.

Upon the next MATLAB startup after a fatal problem, the "Error Log Reporter" on page 1-21 prompts you to e-mail the log to The MathWorks.

2

Desktop

If you have an active Internet connection, you can watch the Working in the Development Environment video demo for an overview of the major functionality. The easiest way to learn to use the desktop is just by working with it. If you have problems or questions, refer to the following sections.

Overview of the Desktop (p. 2-3)	Basic summary of the desktop and its tools.
Arranging the Desktop (p. 2-6)	Open and arrange desktop tools and documents to suit your needs.
Examples of Desktop Arrangements (p. 2-15)	Scan the examples to see various ways to arrange the desktop.
Shortcuts for MATLAB — Easily Run a Group of Statements (p. 2-32)	Use MATLAB shortcuts, an easy way to run a group of MATLAB functions from the desktop.
Keyboard Shortcuts (p. 2-40)	Use the keyboard as an alternative to a mouse or other pointing device to access desktop features.
Other Desktop Features (p. 2-44)	Use the **Start** button, toolbars, menus and context menus, and status bar. Select multiple items, cut, copy, and paste, set up pages for printing, use a MATLAB Web browser, and access the MathWorks Web site from MATLAB.
Preferences (p. 2-59)	Specify options for tools such as fonts, colors, and more.

Fonts Preferences for Desktop Tools (p. 2-62)	Use desktop font preferences to specify the font characteristics for MATLAB desktop tools.
Colors Preferences for Desktop Tools (p. 2-70)	Set desktop color preferences for desktop tools, including syntax highlighting.
General Preferences for MATLAB (p. 2-76)	Set options for toolbox path caching, figure window printing, delete function behavior, MAT-file save format, confirmation dialogs, source control system, and multithreaded computation.
Accessibility (p. 2-85)	Use assistive technologies and accessibility features when working with MathWorks software.

Overview of the Desktop

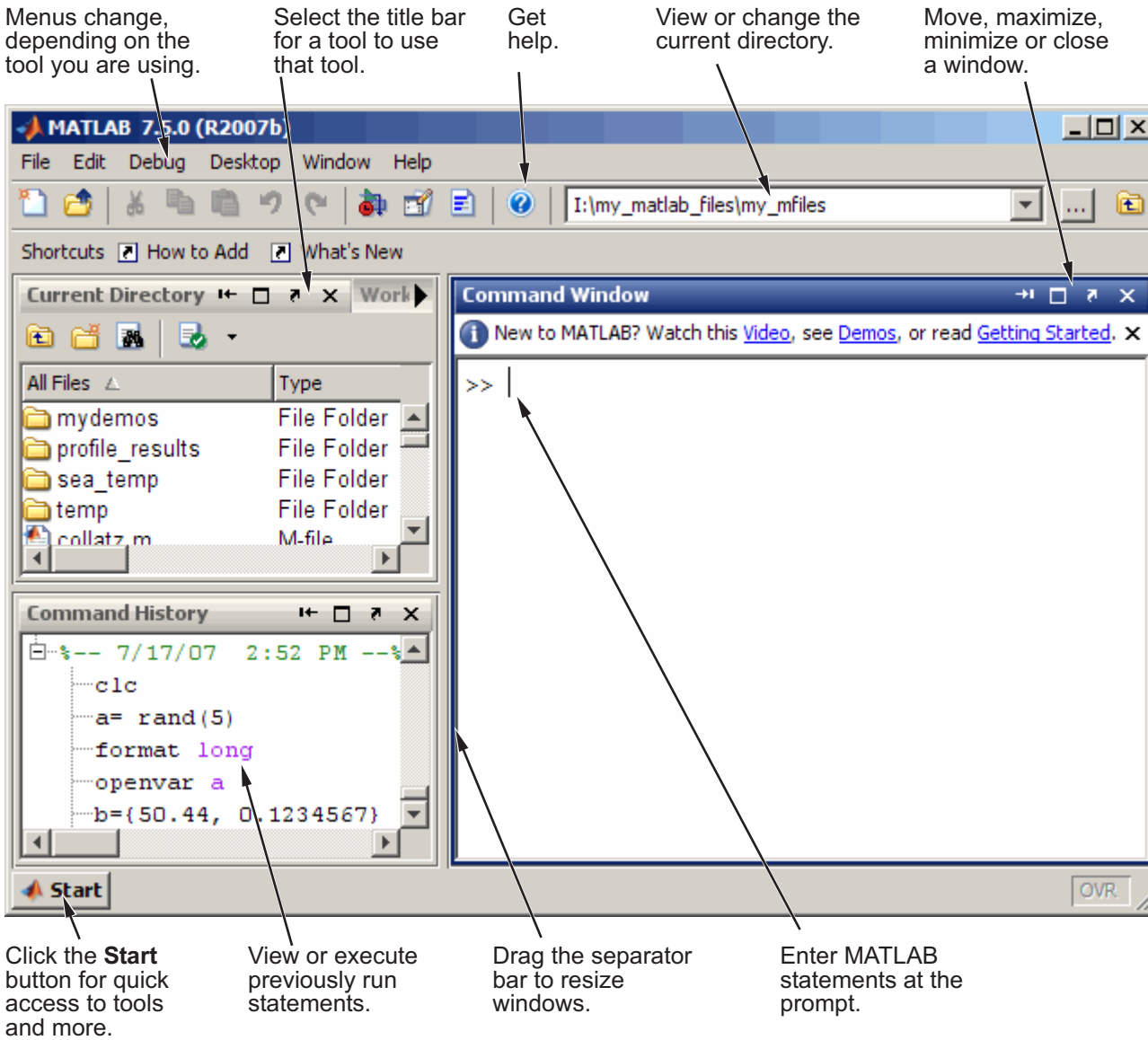
In this section...

About the Desktop

In general, when you start MATLAB, it displays the MATLAB desktop, a set of tools (graphical user interfaces or GUIs) for managing files, variables, and applications associated with MATLAB.

The first time you start MATLAB, the desktop appears with the default layout, as shown in the following illustration. You can change the desktop arrangement to meet your needs, including resizing, moving, and closing tools. For details, see "Arranging the Desktop" on page 2-6.

Some tools, such as the Editor/Debugger and Array Editor, support multiple document windows within them. Similarly, you can group multiple figure windows together. For information about working with documents in the desktop, see "Opening and Arranging Documents" on page 2-8.


Menus change, depending on the tool you are using.
Select the title bar for a tool to use that tool.
Get help.
View or change the current directory.
Move, maximize, minimize or close a window.
MATLAB 7.5.0 (R2007b)
File Edit Debug Desktop Window Help
I:\my_matlab_files\my_mfiles
Shortcuts How to Add What's New
Current Directory
All Files Type
mydemos File Folder
profile_results File Folder
sea_temp File Folder
temp File Folder
collatz.m M-file
Command History
%-- 7/17/07 2:52 PM --%
clc
a= rand(5)
format long
openvar a
b={50.44, 0.1234567}
Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
>>
Start
OVR
Click the Start button for quick access to tools and more.
View or execute previously run statements.
Drag the separator bar to resize windows.
Enter MATLAB statements at the prompt.

Summary of Desktop Tools

The following tools are managed by the MATLAB desktop, although not all of them appear by default when you first start. If you prefer a command-line interface, you can often use equivalent functions to accomplish the same result. To perform the equivalent of the GUI tasks in M-files, you must use the equivalent function. Instructions for using equivalent functions to perform the task are provided with the documentation for each tool and are typically labeled as Function Alternatives.

Desktop Tool	Description
Array Editor	View array contents in a table format and edit the values.
Command History	View a log of or search for the statements you entered in the Command Window, copy them, execute them, and more.
Command Window	Run MATLAB statements.
Current Directory Browser	View files, perform file operations such as open, find files and file content, and manage and tune your files.
Editor/Debugger	Create, edit, debug, and analyze M-files (files containing MATLAB statements).
Figures	Create, modify, view, and print MATLAB figures.
File Comparisons	View line-by-line differences between two files.
Help Browser	View and search the documentation and demos for all your MathWorks products.
Profiler	Improve the performance of your M-files.
Start Button	Run tools and access documentation for all your MathWorks products, and create and use MATLAB shortcuts.
“Web Browser” on page 2-55	View HTML and related files produced by MATLAB.
Workspace Browser	View and make changes to the contents of the workspace.

Arranging the Desktop

In this section...

See also "Examples of Desktop Arrangements" on page 2-15.

Modifying the Desktop Configuration

You can modify the desktop configuration to best meet your needs. Because the desktop uses many standard graphical user interface (GUI) conventions, it is easy to learn about arranging the desktop just by using it.

The desktop manages tools differently from documents. The Command History and Editor/Debugger are examples of tools, and an M-file is an example of a document, which appears in the Editor/Debugger tool.

Opening and Arranging Tools

This table summarizes actions for arranging desktop tools. For further information, click the "see more details online" links.

Tool Action	Steps to Perform
Opening desktop tools	To maximize your work area, keep open only those tools you use. To open a tool, select the tool name from the **Desktop** menu. Opened tools have a check mark before them in the menu. The tool appears in the location it occupied the last time it was open. The sizes of other tools adjust to accommodate the newly opened tool. See more details online.

Tool Action	Steps to Perform
Navigating among desktop tools	The **Window** menu displays all open desktop tools and documents, as well as opened tools for other MathWorks products. Select an entry in the **Window** menu to go directly to that tool or document. Another way to access an undocked desktop tool is by selecting its entry in the Windows task bar, or the equivalent for your platform. See also "Keyboard Shortcuts" on page 2-40 and more details online.
Closing desktop tools	To close a desktop tool, select the item in the **Desktop** menu, which clears the check mark in the menu and closes the tool. Or click the Close box (X) in the title bar for the tool, or select **File > Close** for the tool. See more details online.
Resizing tools	To resize tools in the MATLAB desktop, drag the separator bar, which is the bar between tools. You can hide the title bars for tools in the desktop so the tools use less space—select **Desktop > Titles**, and then hover over a title bar to see a tooltip containing the name of the tool. See more details online.
Moving tools within the desktop	To move a tool in the MATLAB desktop, drag the title bar of the tool toward where you want the tool to be located. As you drag the tool, an outline of it appears. When the outline nears a position where you can keep it, the outline snaps to that location. Release the mouse button. The tool stays at the new location. Other tools in the desktop resize to accommodate the new configuration. The inside edges of the desktop container and tools all act as if they are "sticky," so you can position a tool along any inside edge. See more details online.
Moving tools out of the desktop (undocking)	Move a tool out of the desktop to make it larger or easier to work with. To move a tool outside the MATLAB desktop (called *undocking*), select the tool to make it active, and then select **Desktop > Undock > Toolname**. The tool appears outside the MATLAB desktop and an entry for it appears in the Windows task bar or the equivalent for your platform. Tools within the desktop resize accordingly. Another way to undock is by using the Undock button  in the tool's title bar. See more details online.
Moving tools into the desktop (docking)	To move a tool that is outside the MATLAB desktop into the desktop, click the Dock button  in the tool's menu bar, or select **Desktop > Dock Toolname**. See more details online.

Tool Action	Steps to Perform
Grouping tools together	You can group tools so that they overlay each other in the MATLAB desktop. To group tools together, drag the title bar of one tool in the desktop on top of the title bar of another tool in the desktop. To make a tool active, click its name in the title bar. See more details online.
Maximizing tools in the desktop	To resize the active tool so it occupies the entire MATLAB desktop, double-click the tool's title bar; to return to the layout prior to maximizing, double-click the title bar of the maximized tool. Alternatively, use the menus: select **Desktop > Maximize Toolname**. To return to the layout prior to maximizing, select **Desktop > Restore Toolname**. You can also use the Maximize button and Restore button in the tool's title bar. This feature is not supported on Macintosh platforms.
Minimizing tools in the desktop	You can minimize any tool in the desktop, which creates a button along an edge of the desktop that represents the tool. Select **Desktop > Minimize Toolname**. You can also use the Minimize button in the tool's title bar. The tool's button appears along the edge indicated by the minimize arrow in the menu item or on the button. To move the tool's button to a different edge, right-click the button, and from the context menu, select an edge. To view or use a minimized tool, hover over or click the button—this temporarily opens the tool in the desktop. Once you are finished using the tool, click the button or another tool and the tool is again shown only as a button along the edge. To return the tool to the desktop layout position it occupied before being minimized, double-click the button. Alternatively, restore it by right-clicking the button and selecting **Restore > Toolname**, or use the Restore button in the tool's title bar. This feature is not supported on Macintosh platforms.

Opening and Arranging Documents

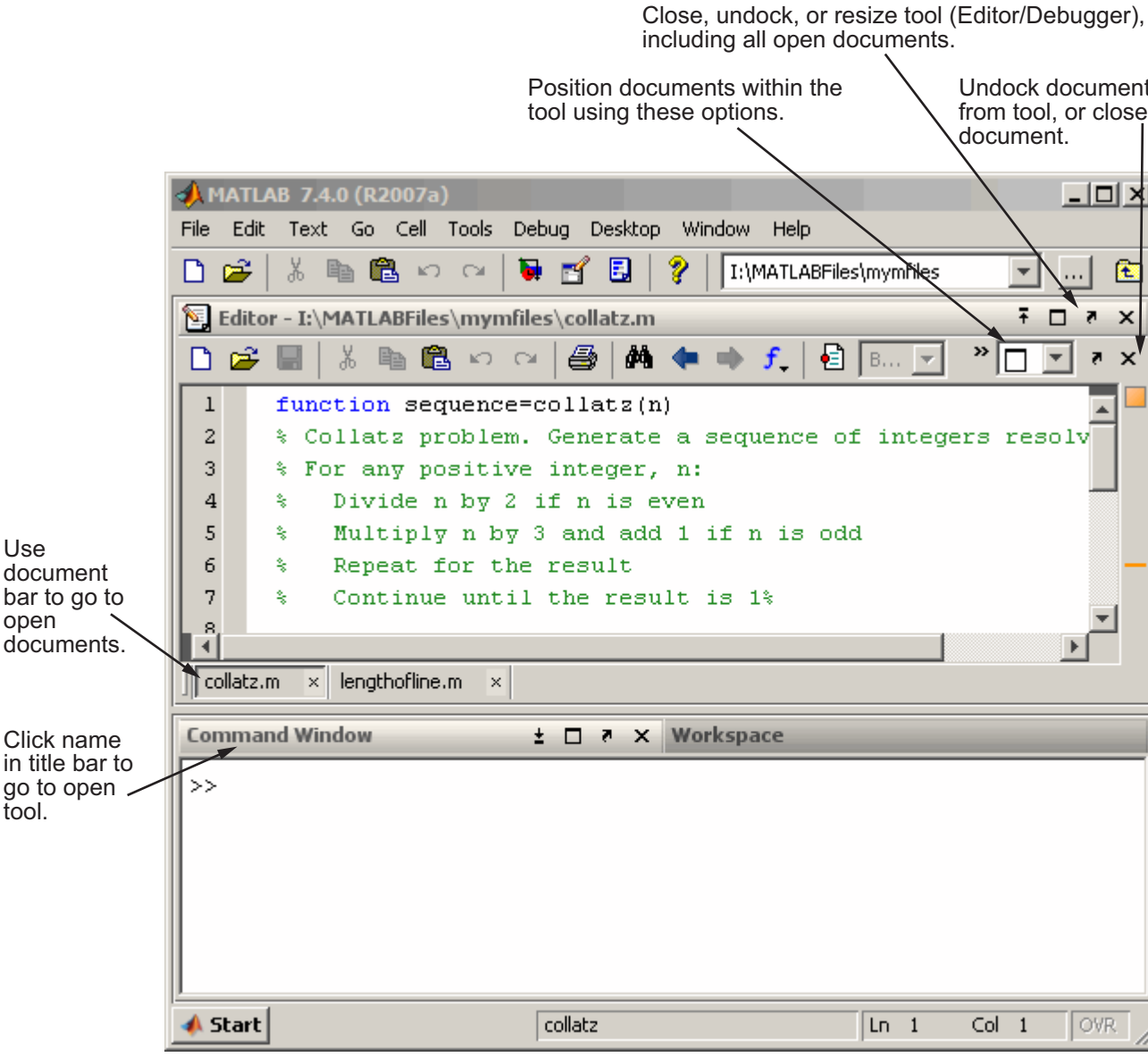
Open a document, such as an M-file or a variable, and it opens in its tool, for example, the Editor/Debugger or Array Editor. The following example illustration shows a desktop arrangement that includes Editor/Debugger and Array Editor documents. See instructions in "Summary of Actions for Arranging Documents" on page 2-11.

Example of Documents in the Desktop

Some common actions for working with documents in the desktop are

- Use the document bar to go to open documents.
- Use the **Window** menu or equivalent toolbar buttons to position documents.
- Close or undock a tool, including all documents in the tool.
- Undock a document from its tool.
- Use the document Close box with the **Ctrl** key to close the document without saving it and without displaying the unsaved document dialog box.

See also “Examples of Desktop Arrangements” on page 2-15.


Close, undock, or resize tool (Editor/Debugger), including all open documents.
Position documents within the tool using these options.
Undock document from tool, or close document.
Use document bar to go to open documents.
Click name in title bar to go to open tool.
MATLAB 7.4.0 (R2007a)
File Edit Text Go Cell Tools Debug Desktop Window Help
I:\MATLABFiles\mymfiles
Editor - I:\MATLABFiles\mymfiles\collatz.m
function sequence=collatz(n)
% Collatz problem. Generate a sequence of integers resolv
% For any positive integer, n:
% Divide n by 2 if n is even
% Multiply n by 3 and add 1 if n is odd
% Repeat for the result
% Continue until the result is 1%
collatz.m
lengthofline.m
Command Window
Workspace
>>
Start
collatz
Ln 1 Col 1
OVR

Summary of Actions for Arranging Documents

This table summarizes actions for arranging documents in their tool. For further information, click the click the “see more details online” links.

Document Action	Overview
Opening documents	When you open a MATLAB document, it opens in the associated tool. If the tool is not already open, it opens when you open the document and appears in the position it occupied when last used. Figures open undocked, regardless of the last position occupied. How to open a document depends on the document type: • M-file: Select **File > Open** and select the M-file. It opens in the Editor/Debugger. • Workspace variable: In the Workspace browser, double-click the variable. It opens in the Array Editor. • HTML file: In the Current Directory browser, double-click the file. It opens in the MATLAB Web Browser. • Figure: Type `plot` or use another graphics function. The plot appears in a figure window. There are many additional ways to open documents. See more details online.

Document Action	Overview
Navigating among documents — the document bar	When more than one document is open within a tool, each document is either maximized (the default), or arranged so that multiple documents are visible at once. Click a document that is in view to make it the active document. See also "Keyboard Shortcuts" on page 2-40. Use the document bar to go to a document that is open but not in view. The names of all open documents appear in the document bar. Select a document name in the document bar to make that document active. To show the document bar if it is not open, select **Desktop > Document Bar > Bar Position** and select the position for it, for example, **Right**. See more details online. Entries for undocked documents appear in the Windows task bar, or the equivalent for your platform. Click the task bar entry for a document to make that document active.
Positioning, moving, and resizing documents	To position open documents within their tool, select an arrangement from the **Window** menu when the tool is active, or by using the equivalent toolbar button for **Maximize**, **Float**, **Left/Right Tile**, **Top/Bottom Tile**, and **Tile**. You can also define a specific grid arrangement using **Window > Tile...**. On the Macintosh platform, the tile option might not be available in the **Window** menu so use the Tile button ⊞ instead. With the tile arrangements, you refine the document position by moving the pointer over the handle (▪) on the separator bar. A Close box then appears. When you click the Close box between two open documents, both documents stay open, but one moves on top of the other. When you click the Close box between a document and an empty tile, the empty tile closes. To move a document in a tiled arrangement, drag the title bar of a document to another tile. To resize tiled documents, drag the separator bar between the documents. See also the Editor/Debugger's "Split Screen Display" on page 6-38, which allows you to view two different parts of the same file simultaneously. To move or resize maximized documents, you move or resize their tool. See more details online.

Document Action	Overview
Closing documents	To close a document, click the Close box in the document's title bar. After closing all the documents in a tool, the tool remains open with no documents in it. If you select the Close box for the tool, all documents in that tool close. In the Editor/Debugger, when you close a file that has unsaved changes, a prompt appears asking if you want to save the document. To close a file without saving changes and without seeing the save prompt, use **Ctrl** when you click the document's Close box. See more details online.
Moving documents and tools out of the desktop (undocking)	To undock all documents in a tool from the desktop, click the Undock button in the tool's title bar. The tool and its documents move outside of the desktop. See more details online. To undock a document from its tool, click the Undock button for the document. The Undock button is either in the document's title bar, menu bar, or toolbar, depending on the document type and whether or not the document is within the desktop or is in its tool outside of the desktop. Undocked documents have entries in the Windows task bar (or the equivalent for your platform).
Docking documents and tools	When you dock a document, it moves to the position in the tool that it occupied before you undocked the document. To dock a document, click the Dock button in the document's menu bar. See more details online.
Grouping documents in a tool outside the desktop	To group all of the documents for a tool together outside of the desktop, undock the tool from the desktop, not just the documents. If you have already undocked all of the documents and closed the empty tool that had contained them, select **Desktop > Dock All in Editor**, for example. This moves all the documents into the tool in the desktop. Then undock the tool.

Saving Desktop Layouts

When you end a session, MATLAB saves the desktop layout. The next time you start MATLAB, the desktop is restored to the way you last had it.

To use a predefined layout, select **Desktop > Desktop Layout**, and choose a configuration. See more details in the online documentation.

To save your own layouts for later reuse, select **Desktop > Save Layout** and provide a name. To reuse a saved layout, select the name from **Desktop > Desktop Layout**. See more details in the online documentation.

Examples of Desktop Arrangements

In this section...

About These Examples

Scan the illustrations in the following examples for a desktop arrangement similar to what you want, and then follow the brief instructions to achieve the arrangement. There are many different ways to accomplish the result; these instructions present just one way. The instructions might not apply exactly, depending on how your desktop looks before you start.

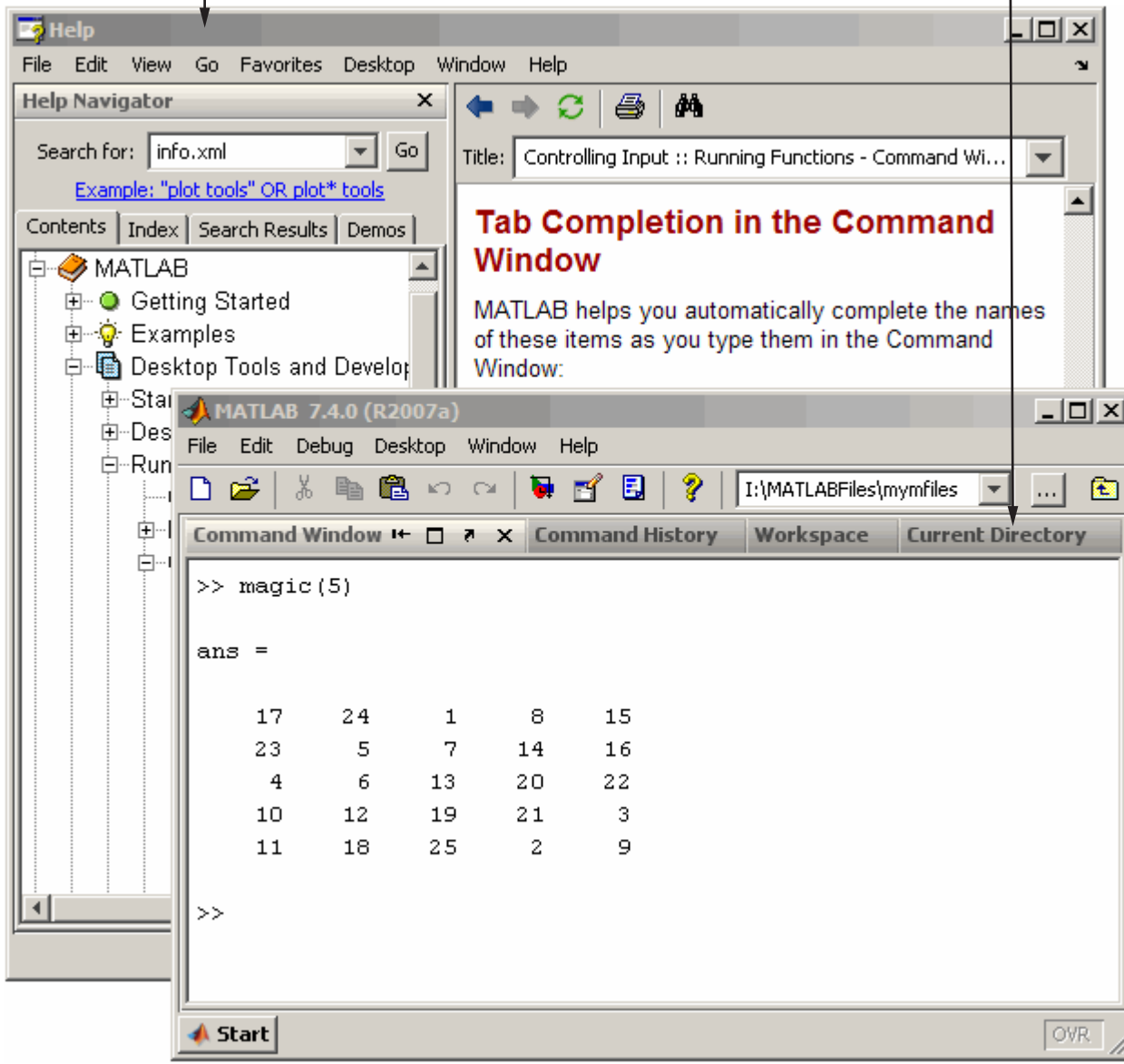
Tool Outside of Desktop and Other Tools Grouped Inside Desktop Example

This example shows two ways you can increase the size of a tool. One way is to move a tool outside of the desktop to increase its size. Here, the Help browser was moved outside of the desktop and made larger. To move a tool outside of the desktop, click the Undock button in the tool's title bar when the tool is in the desktop.

Another way to increase the size of a tool is by grouping tools together inside the desktop, and then accessing a tool via the tool's name in the title bar. Here the Command Window, Command History, Workspace browser, and Current Directory browser are grouped together. To achieve this, drag the title bar of one tool on top of the title bar of the tool(s) you want to group it with.

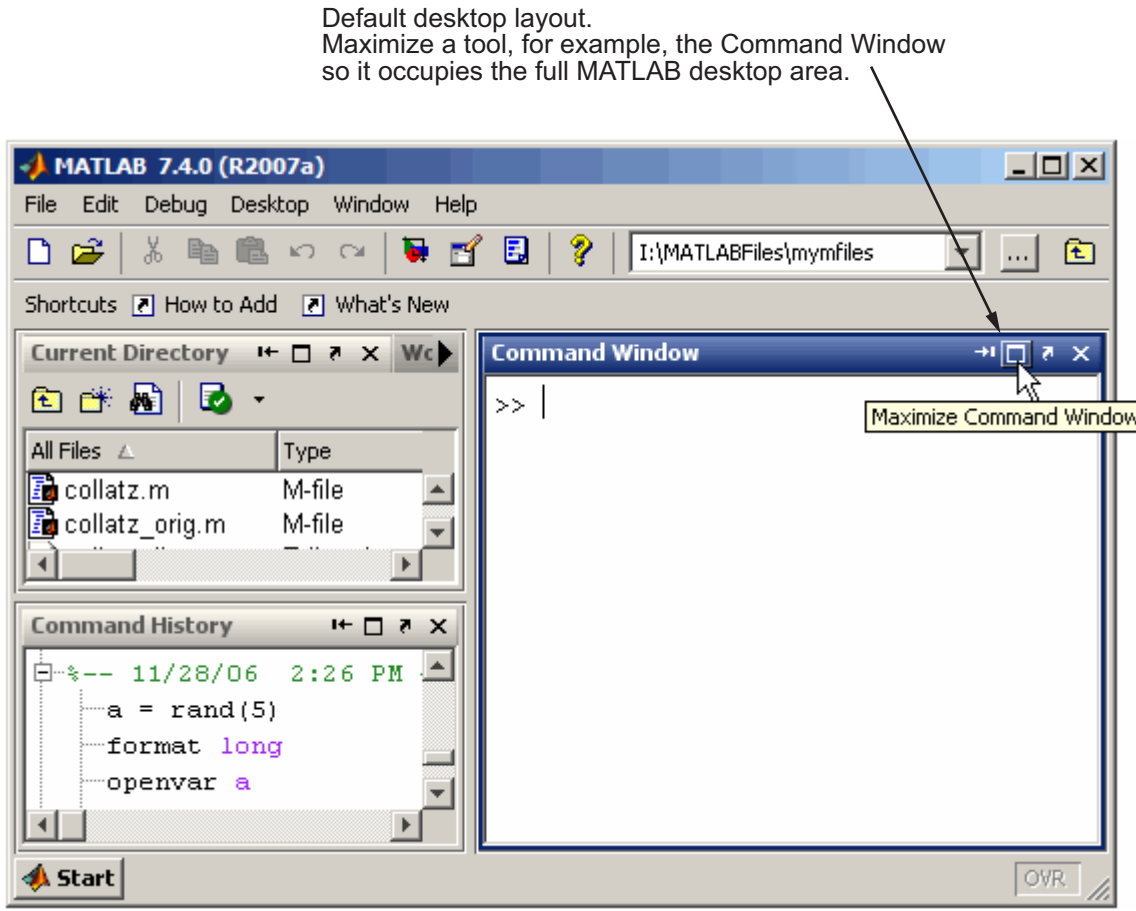
Help browser is undocked from desktop to provide a large area for viewing documentation when needed.

Four tools in the desktop are grouped together, providing a large area for working with a given tool. Click a tool's name in the title bar to make that tool active.



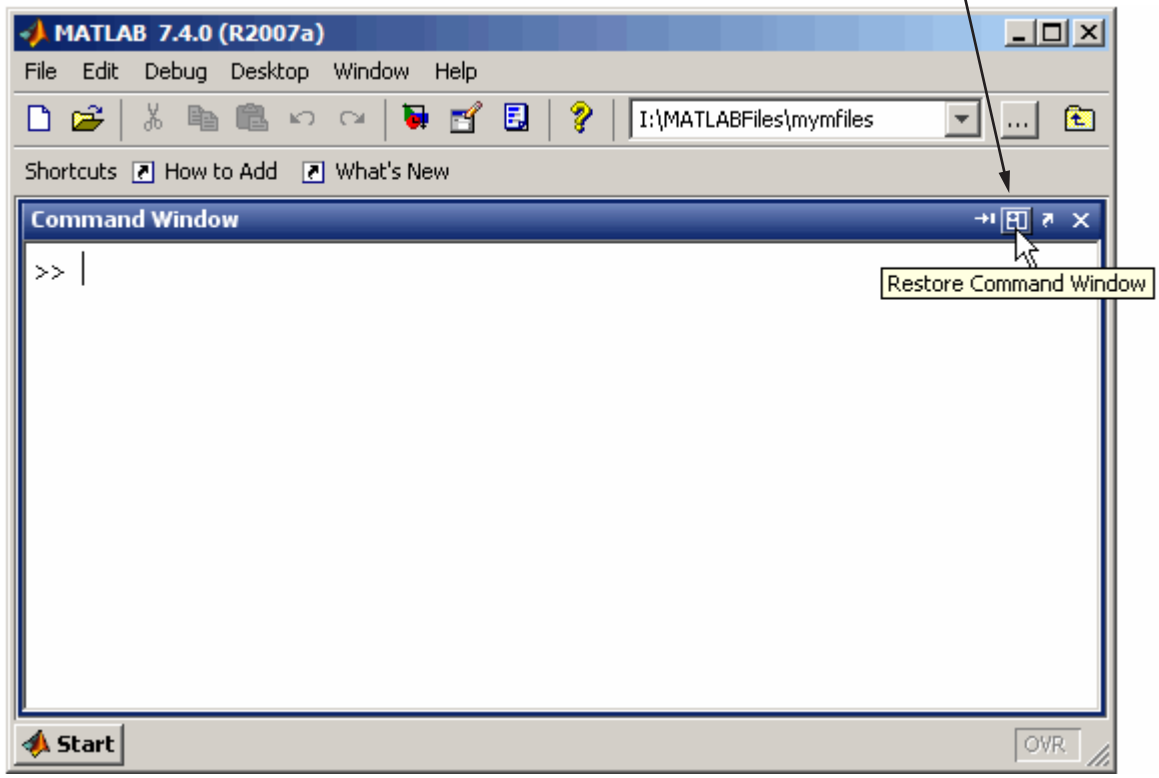
Maximized Tool in Desktop Example

This example shows a way you can temporarily increase the size of a tool so that it occupies the entire area of the desktop. In this example, the Command Window in the default desktop layout is temporarily maximized by clicking the Maximize button in the tool's title bar.



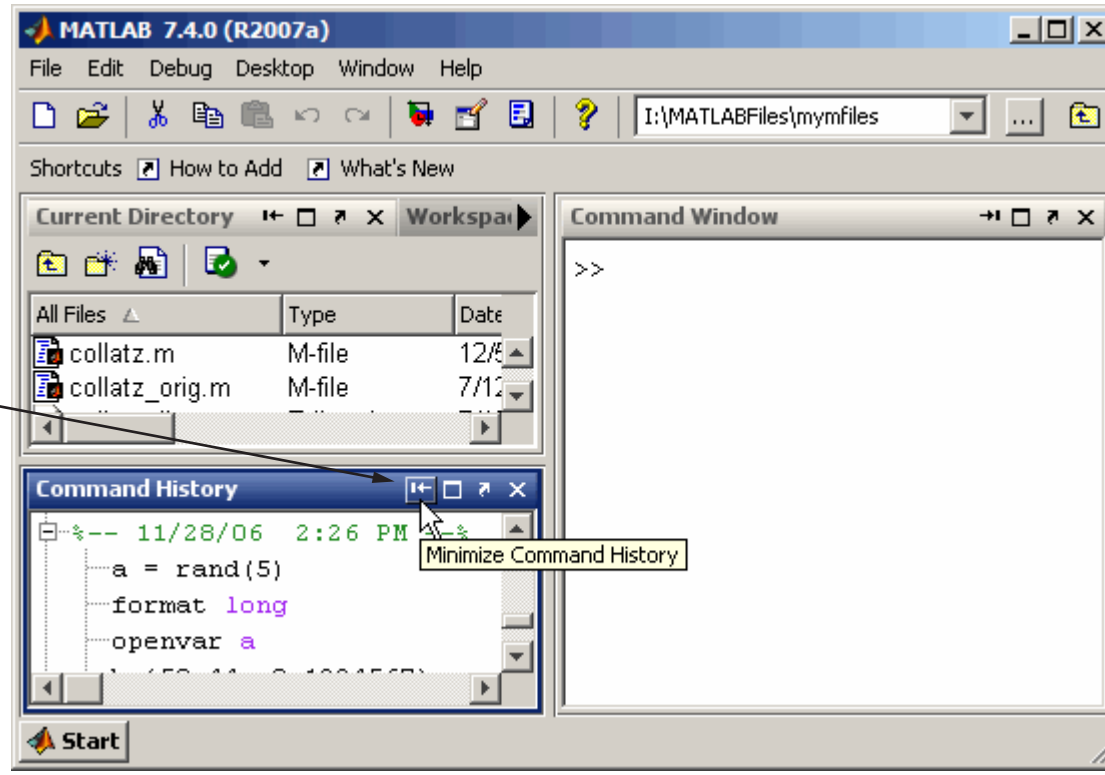
In this example, the maximized Command Window is being returned to its size and position in the default desktop layout by clicking the Restore button in the title bar.

Maximized, the Command Window now occupies the full desktop area. Restoring the Command Window returns it to its original size and location in the desktop.



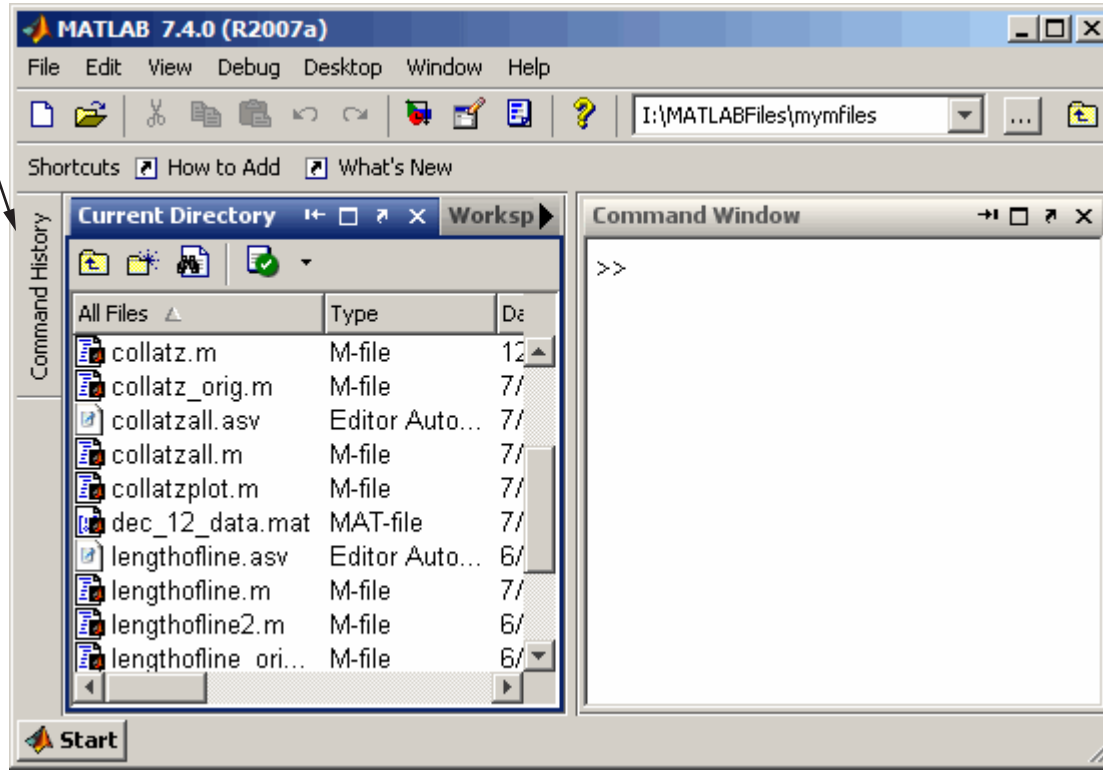
Minimized Tools in Desktop Example

Minimize a tool in the desktop to give the remaining desktop tools more space in the desktop. Minimizing is available on Windows and UNIX platforms. In this example, the Command History in the default layout is being minimized to the left edge of the desktop.



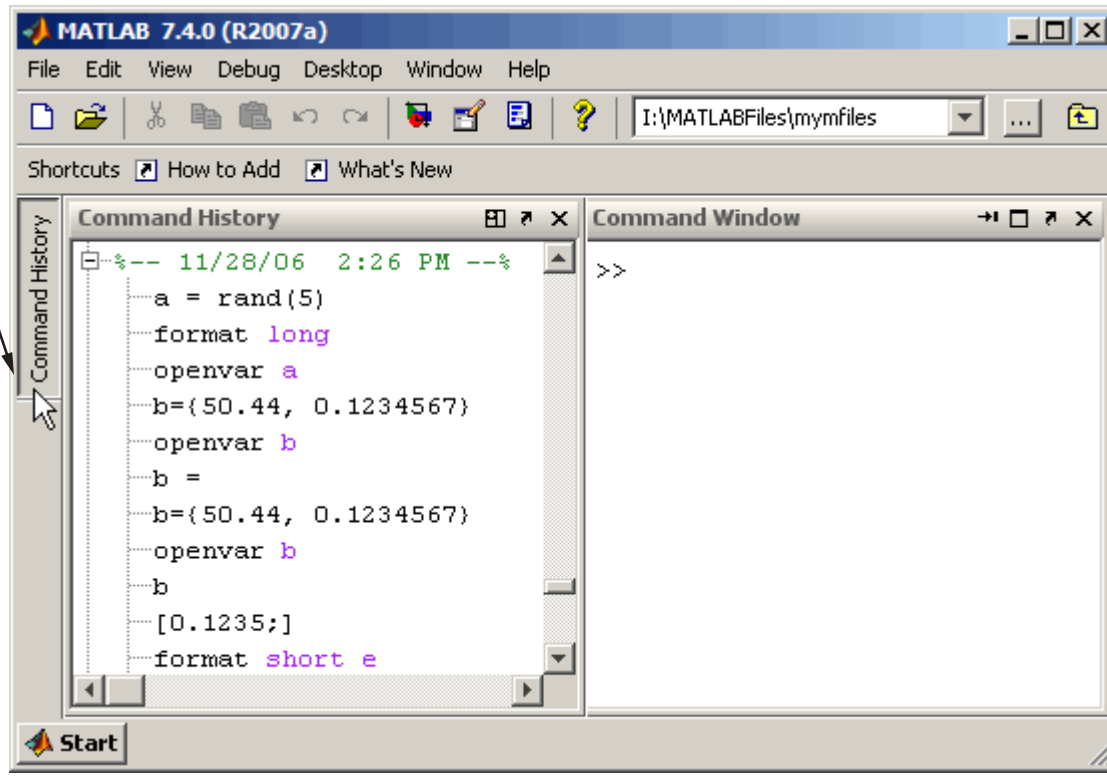
In this illustration, the Command History has been minimized and appears as a button along the left edge.

When minimized, a tool, such as the Command Window in this example, is represented by a button on the desktop border.



This illustration shows the minimized Command History being temporarily opened, as a result of clicking or hovering over the button.

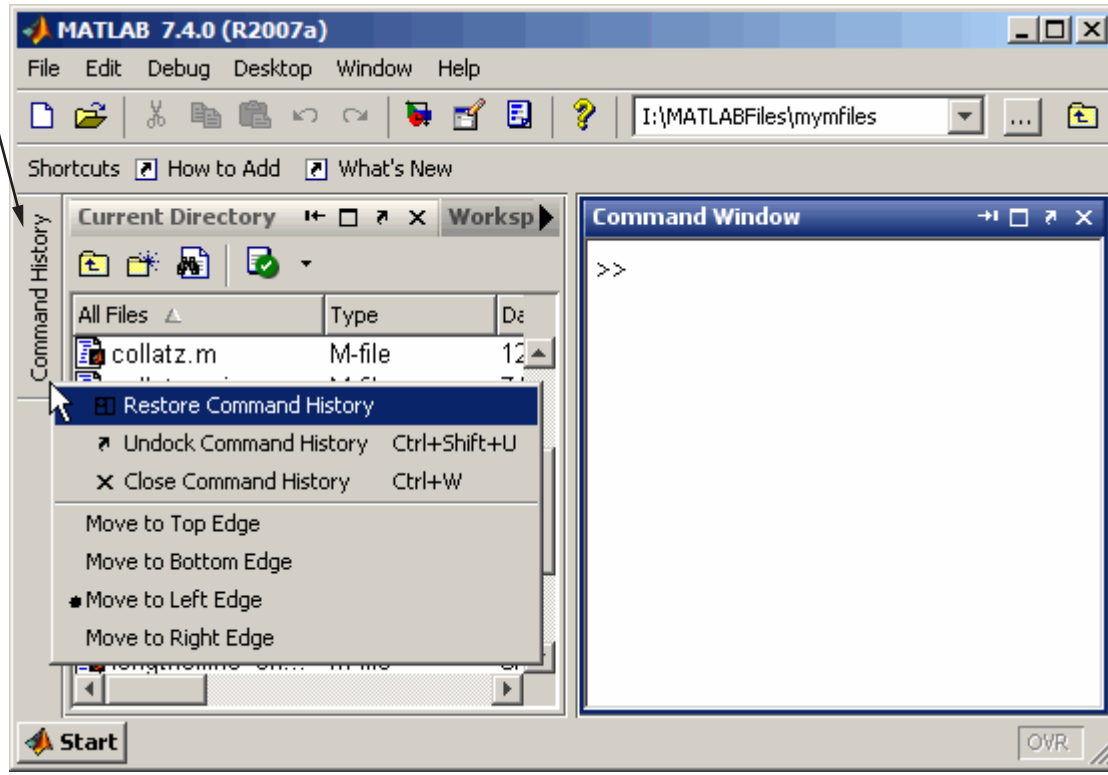
Hover over or click the button for a minimized tool to temporarily view or use the tool.
The tool is temporarily displayed until you select another tool.
Then the tool becomes minimized again.



After using the Command History and clicking the button, or moving on to another tool, the Command History again becomes minimized as a button along the left edge.

This illustration shows the Command History being returned to the position and size it occupied in the desktop prior to being minimized by clicking the Restore button .

On the button for a minimized tool, right-click, and from the context menu,select **Restore**. The tool resumes the size and position it had in the desktop before it was minimized.



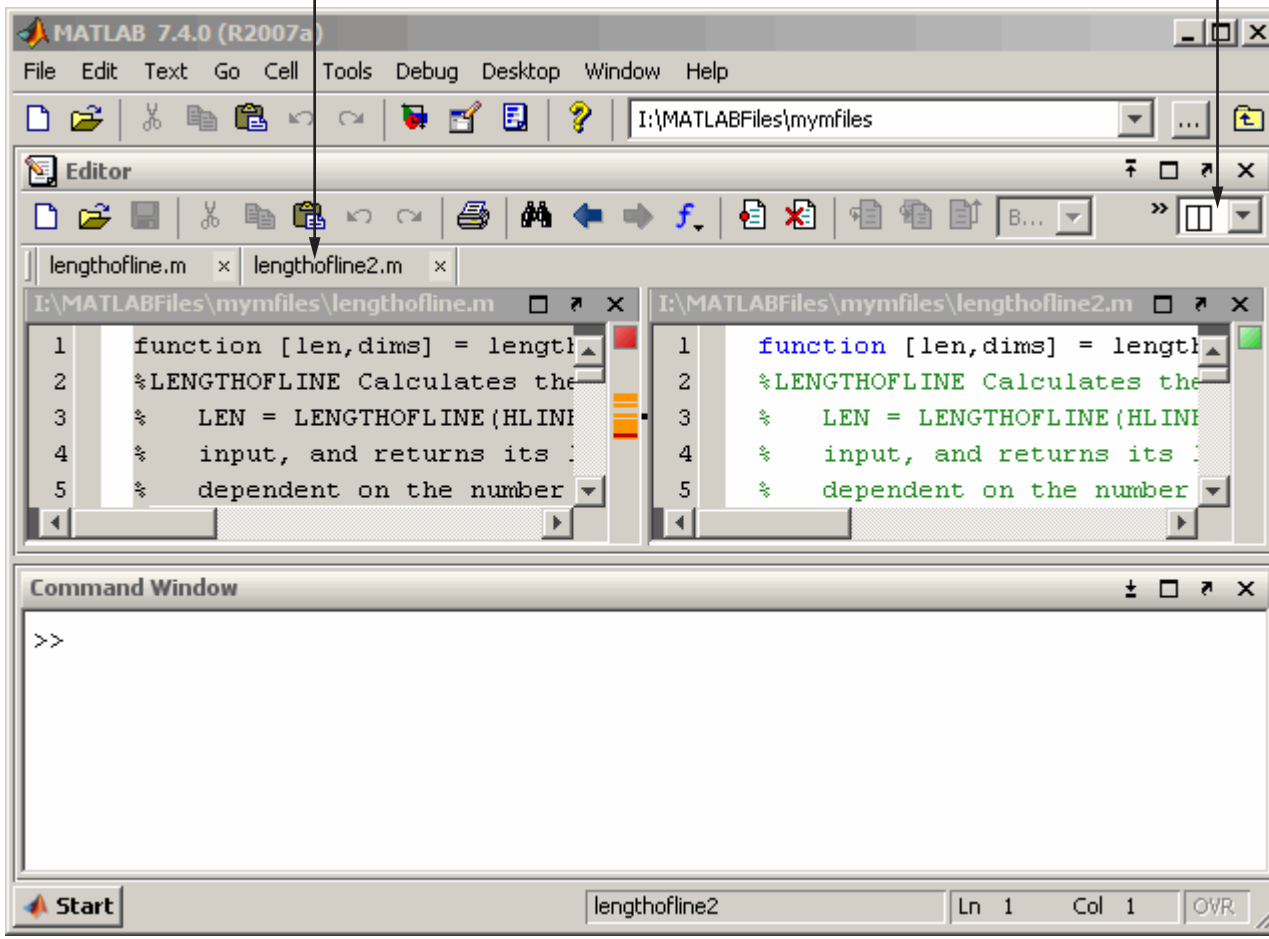
Tiled Documents in Desktop Example

When you open a document (for example, an M-file), it also opens the tool (for example, the Editor/Debugger) if the tool is not already opened. Subsequent documents of the same type open in the tool and you can then arrange the documents within the tool. You can move a document on top of another document, so that the one on top hides the one(s) beneath it, or you can show multiple documents at once. This example shows two M-files side-by-side, as a result of selecting **Window > Left/Right Tile** (or the ◫ toolbar button).

When tools and documents are docked, you might want to save space by hiding toolbars and document bars. In this illustration, the desktop shortcuts toolbar is hidden. Select **Desktop > Toolbar name** to hide (or show) a toolbar. To see or move the document bar, select **Desktop > Document Bar > Bar Position**, and choose its location, for example, **Top**.

The shortcuts toolbar is hidden. The document bar is at the top edge of the Editor/Debugger.

Select a button from the list to arrange the documents, such as Left/Right Tile.



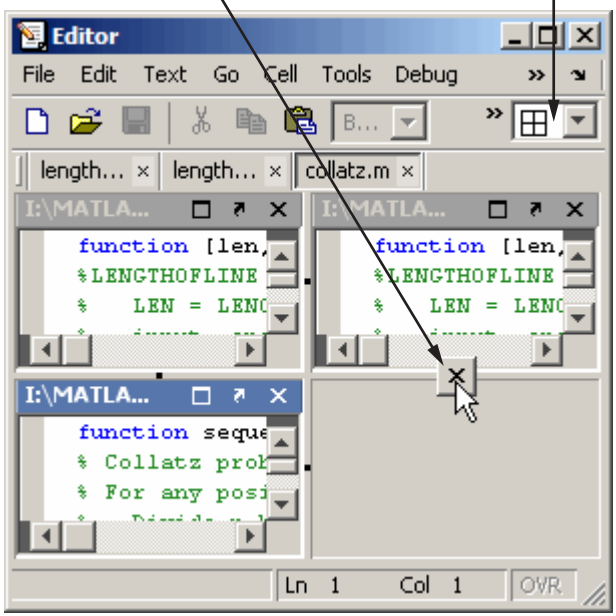
No Empty Document Tiles Example

To see more than two documents at once, select the Tile button and move the pointer across the gird that appears to select the number of tiles you want. The following “Before” illustration has four tiles, but only three documents are open. (The empty tile is gray.) You can move a document to any empty tile by dragging its title bar to the new location. To close an empty tile, position the pointer over the handle on the separator bar. It becomes a Close box, as shown here, which you click to close the empty tile. After clicking the Close box, the empty tile closes and the neighboring document expands as shown in the following “After” illustration. Similarly, click the Close box between two tiles containing documents, and one document becomes hidden. Note that preferences to show line numbers and M-Lint indicators have been cleared to provide more horizontal space.

Before

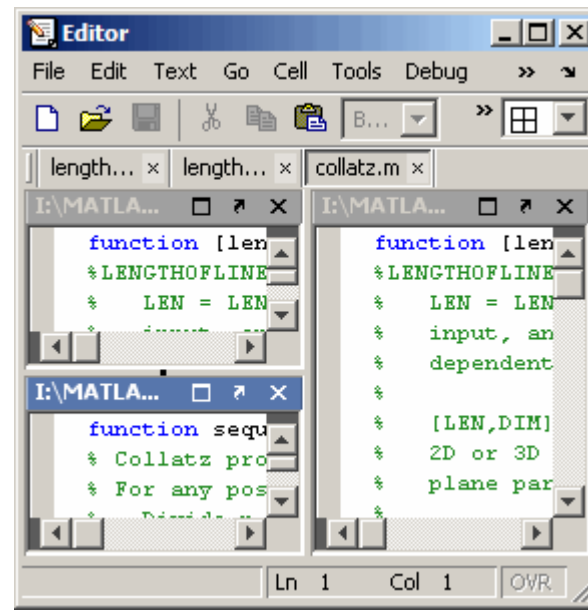
Close the empty tile using the handle on the separator bar.

Tile more than two documents with the Tile button.



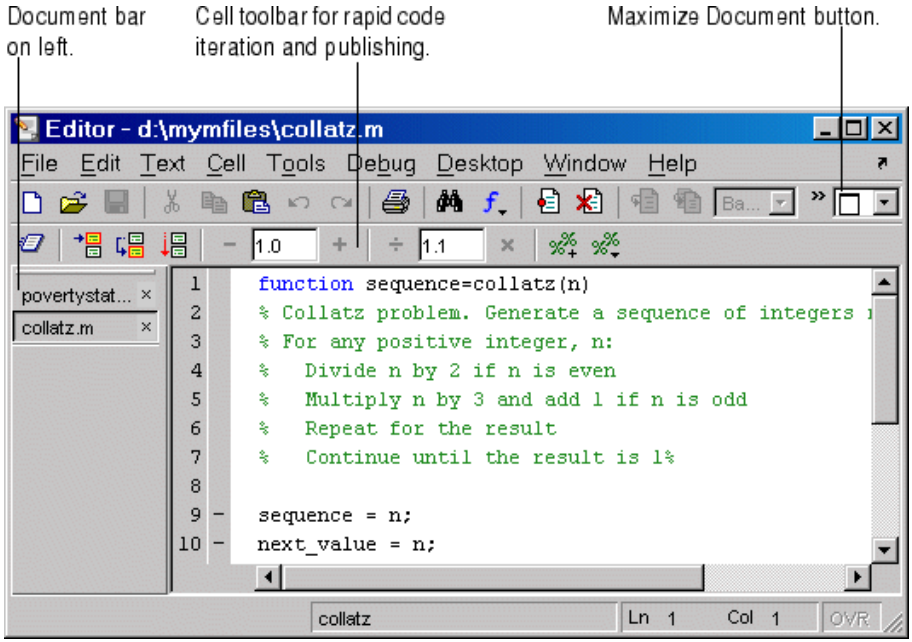
After

There are only three tiles.



Maximized Documents Outside of the Desktop Example

This example illustrates a way to provide a large area for multiple documents, in this case, M-files maximized in the undocked Editor/Debugger.



Some common actions for working with documents outside of the desktop are

- Group all Editor/Debugger documents together — select **Desktop > Dock All in Editor** from any Editor/Debugger document.
- Move all Editor/Debugger documents outside of the desktop — select **Desktop > Undock Editor** when the Editor/Debugger is the active window.
- Make a document occupy the full area in the Editor/Debugger — click the Maximize button in the Editor/Debugger toolbar, or select **Window > Maximize**.
- Display the cell toolbar — select **Desktop > Cell Toolbar**. This menu item is available only when the current document is an M-file.

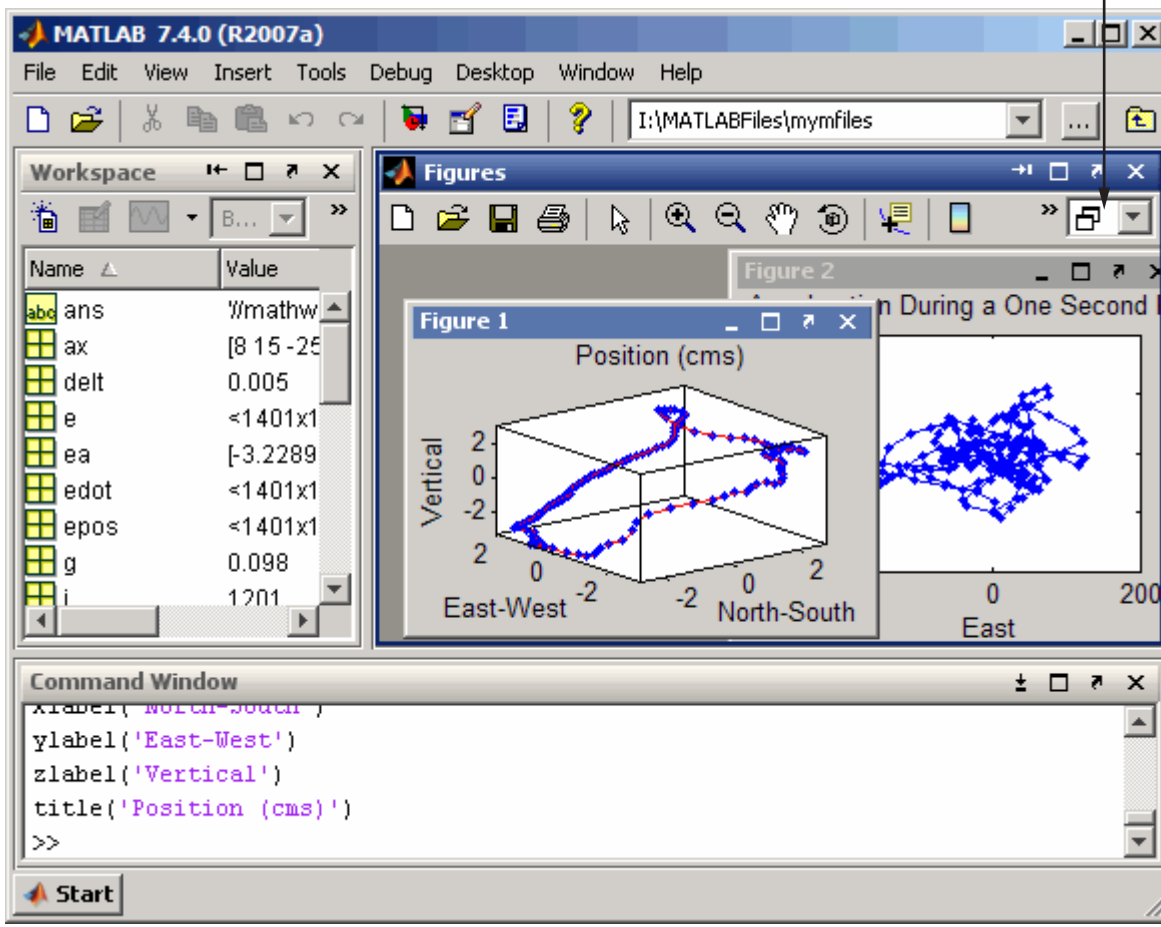
- Access any document in the Editor/Debugger using the document bar. To show the document bar on the left side of the Editor/Debugger, select **Desktop > Bar Position > Document Bar > Left** from the Editor/Debugger.

Floating (Cascaded) Figures in Desktop Example

This example illustrates multiple figures in the desktop. By default, figures open outside the desktop. Click the Dock button in each figure's menu bar to move the figures into the desktop.

You can float (also called cascade) the figures by selecting **Window > Float**, or clicking the Float button. To get even more screen area for the figures, hide the document bar as shown in this example — select **Desktop > Document Bar > Bar Position > Hide**.

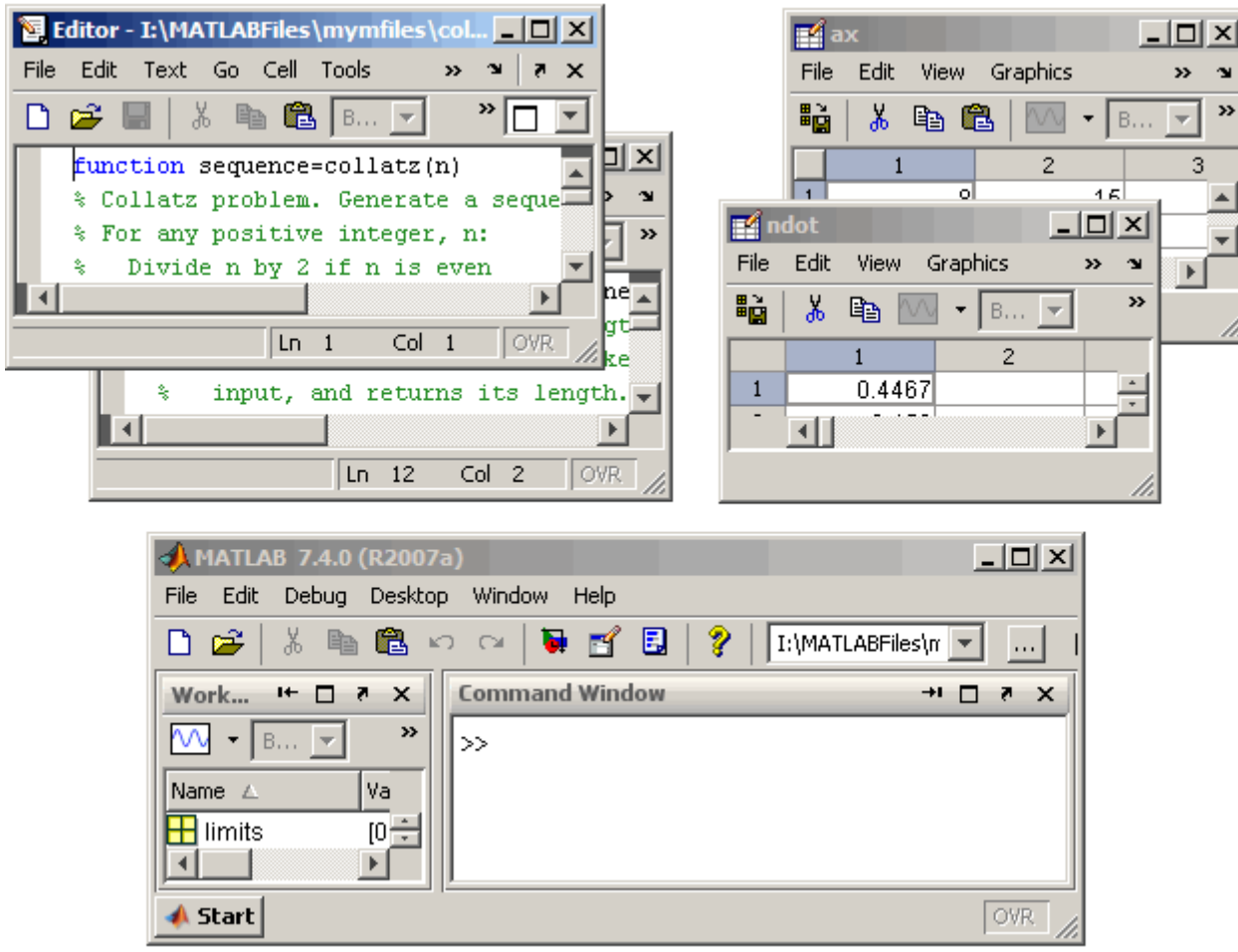
Dock figures in the desktop and use the float option to arrange them within a Figures group. The document bar is hidden.



Undocked Tools and Documents Example

You can use tools and documents outside of the desktop. One way to achieve this is to first undock the tool from the desktop by selecting **Desktop > Undock Toolname**. Then undock documents from the undocked tool by selecting **Desktop > Undock Documentname** from the tool. If you undock all documents from a tool, an "empty" tool window remains.

In this example, one of the Editor/Debugger documents, `collatz.m`, includes the name of the tool with it; the other Editor/Debugger document, `lengthofline.m`, does not. Contrast this with the Array Editor documents, where neither document window includes the name of the tool. This is because the Array Editor was undocked from the desktop, the variables were undocked from the Array Editor, and the "empty" Array Editor window was closed. The tool's undocked documents remain open. If you closed the Editor/Debugger, the `lengthofline.m` document would remain open. To close all undocked documents and their tools at once, select **Window > Close All Documents** from an undocked document window.


Editor - I:\MATLABFiles\mymfiles\col...
File Edit Text Go Cell Tools
function sequence=collatz(n)
% Collatz problem. Generate a seque
% For any positive integer, n:
% Divide n by 2 if n is even
Ln 1 Col 1 OVR
% input, and returns its length.
Ln 12 Col 2 OVR
ax
File Edit View Graphics
ndot
File Edit View Graphics
1 2
1 0.4467
MATLAB 7.4.0 (R2007a)
File Edit Debug Desktop Window Help
I:\MATLABFiles\m
Work...
Name Va
limits [0
Command Window
>>
Start
OVR

Shortcuts for MATLAB — Easily Run a Group of Statements

In this section...

What Is a Shortcut?

A MATLAB shortcut is an easy way to run a group of MATLAB statements that you use regularly. First you create a shortcut that contains all the statements. Then you select and run the shortcut to execute all the statements it contains. Create, run, and organize shortcuts from the **Start > Shortcuts** menu or the desktop **Shortcuts** toolbar.

Differences Between Shortcuts and M-Files

A shortcut is like an M-file script, but unlike an M-file, a shortcut does not have to be on the MATLAB search path or in the current directory when you run it. In addition, you can run the shortcut by selecting it from the **Start** button or desktop **Shortcuts** toolbar, which are readily accessible.

Although shortcuts run MATLAB statements, they are not M-files and are not stored as M-files.

Examples of Useful Shortcuts

These are some examples of useful types of shortcuts:

- If you frequently run the same group of functions, consider creating a shortcut for them. An example of this is setting up your environment when you start working if you do not use a `startup` file, or if there are statements you do not want to include in the `startup` file. Some users

create a shortcut for even a single function they use frequently, such as `clc` to clear the Command Window.

- Create a shortcut to set the same properties for figures you create, such as adding a legend and setting the background color.
- Create a shortcut for a long statement, such as changing the current directory (`cd`) when the pathnames are long.
- Create a shortcut for a statement you do not easily remember but need to use.

Creating Shortcuts

This is an example of a shortcut you might create for a project you work on, the Sea Temperature project. When you work on that project, you might want to set up your environment in a certain way by running a series of statements. You create a shortcut called `sea_temp_env`, which contains the statements. Then when you work on the project, you run the shortcut to execute all of the statements with a single click. The statements are

```
more on
format long e
cd d:/mymfiles/sea_temp_project
clear
workspace
filebrowser
clc
```

To create a shortcut, perform the following steps:

1. From the **Start** button, select **Shortcuts > New Shortcut**.

   The Shortcut Editor dialog box appears.

2. Create the shortcut by completing the dialog box.

   a. Provide a shortcut name in the **Label** field, for example, `sea_temp_environment`.

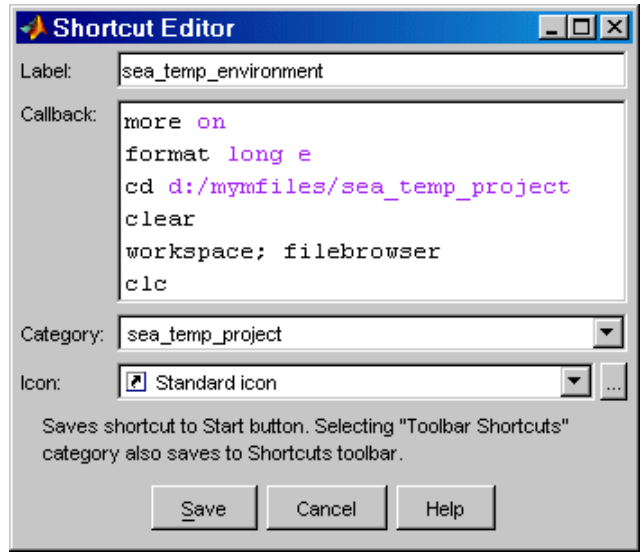
   b. Put the MATLAB statements in the **Callback** field as shown in the following illustration. Either type them in, or copy and paste or drag them from a desktop tool. Edit the statements as needed. The field uses

the Editor/Debugger preferences for key bindings, colors, and fonts. Note that if you copy the statements from the Command Window, the prompt appears in the shortcut, but MATLAB removes the prompt when you save the shortcut.

c Assign a category, which is like a directory for organizing shortcuts. Specify `sea_temp_project`. To add the shortcut to the shortcuts toolbar, select the `Toolbar Shortcuts` category.

d Use the default shortcuts icon, or select your own.

e Click **Save**. MATLAB automatically removes any Command Window prompts (>>) in the **Callback** field upon saving the shortcuts.



3 MATLAB adds the shortcut to the **Shortcuts** entry in the **Start** button, and to the **Shortcuts** toolbar, if you selected that **Category**.

After creating a shortcut, run it by selecting it from its category in the **Start** button. You can also run it from the **Shortcuts** toolbar if you selected the `Toolbar Shortcuts` category.

MATLAB maintains shortcut information in the file `shortcuts.xml`. Type `prefdir`, and MATLAB displays the location of the file. Most likely, you will not need to access this file, as MATLAB updates the file automatically.

For more information on the options in the Shortcut Editor dialog box, click the **Help** button.

Additional Ways to Create Shortcuts

You can also use these methods to create shortcuts:

- Add shortcuts to and run them from the desktop **Shortcuts** toolbar. See "Shortcuts Toolbar" on page 2-35.
- From the Command History window, create a shortcut by selecting statements, right-clicking, and selecting **Create Shortcut** from the context menu. By default, shortcuts created from the Command History window are assigned to the `Toolbar Shortcuts` category, meaning they will appear on the **Shortcuts** toolbar.
- From the Help browser, select **Favorites > Add to Favorites**, complete the Favorites Editor dialog box, and the shortcut appears in the shortcuts `Help Browser Favorites` category. You can also access `Help Browser Favorites` shortcuts from the Help browser **Favorites** menu.
- Drag statements from a desktop tool, such as the Command History, onto the **Start** button.

Running Shortcuts

To run a shortcut, select the shortcut name, for example, **sea_temp_environment**, from the **Start > Shortcuts** menu or from one of its category submenus. All of the statements in the shortcut **Callback** field execute. It is as if you ran those statements from the Command Window, although they are not reflected in the Command History window.

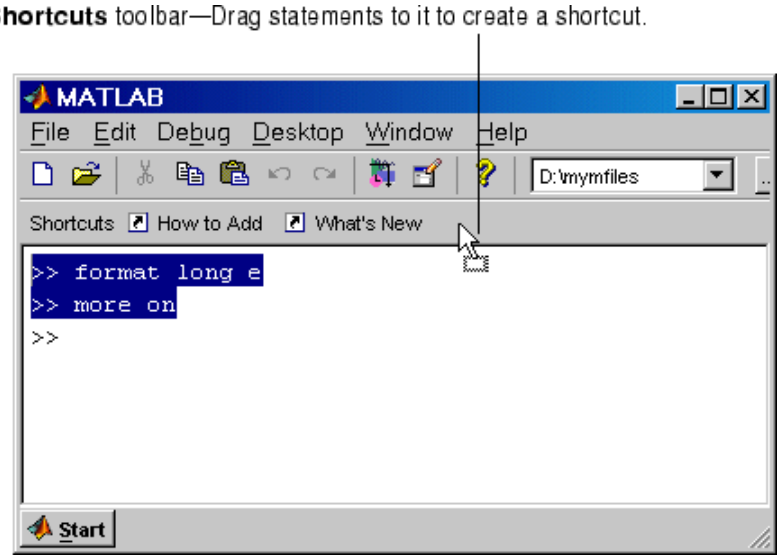
If you added a shortcut to the **Shortcuts** toolbar, you can run it by clicking its icon on the shortcuts toolbar.

Shortcuts Toolbar

The **Shortcuts** toolbar is an alternative to creating and running shortcuts via the **Start** button. To show or hide the shortcuts toolbar, use **Desktop > Shortcuts Toolbar**. To create and run shortcuts via the desktop **Shortcuts** toolbar, perform these steps:

1 Select statements from the Command History window, the Command Window, or an M-file.

2 Drag the selection to the desktop **Shortcuts** toolbar. The following illustration shows two statements being dragged from the Command Window.

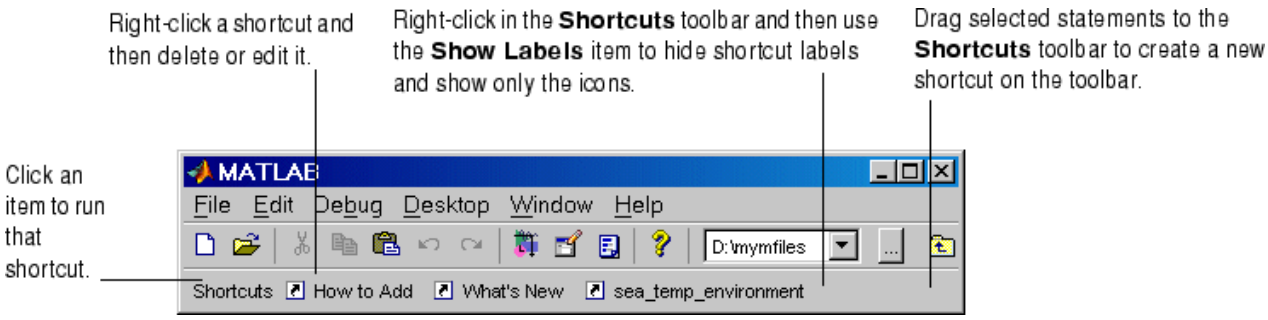


3 The Shortcut Editor dialog box appears. The **Callback** field contains the selected statements, which you can edit as needed. If prompts (>>) from the Command Window appear, note that MATLAB automatically removes them when you save the shortcut. The **Category** field is `Toolbar Shortcuts`, which you must retain in order for the shortcut to appear on the toolbar.

Provide the **Label**, select an **Icon**, and click **Save**.

The shortcut icon and label appear on the toolbar. If you have more shortcuts on the toolbar than can be displayed at once, use the drop-down list to access all of them. For more information, click the **Help** button in the Shortcut Editor dialog box.

4 Click the icon on the **Shortcuts** toolbar to run the shortcut. You can also run the shortcut from the **Start** button by selecting it in the `Toolbar Shortcuts` category.



You can also add a shortcut to the desktop **Shortcuts** toolbar by right-clicking the toolbar and selecting **New Shortcut**. Complete the resulting Shortcut Editor dialog box. Assuming you maintain the `Toolbar Shortcuts` category, the shortcut appears on the toolbar. To change the order of the shortcuts on the toolbar, select **Start > Shortcuts > Organize Shortcuts** and move the shortcuts within the `Toolbar Shortcuts` category.

How to Add and What's New Shortcuts

The **Shortcuts** toolbar includes two shortcuts provided with MATLAB. The **How to Add** shortcut provides help about shortcuts and adding them to the **Shortcuts** toolbar. **What's New** displays the Release Notes documentation.

To remove the **How to Add** or **What's New** shortcut from the **Shortcuts** toolbar, choose a different category. For instructions, see "Organizing and Editing Shortcuts" on page 2-38.

If you do not want to keep these shortcuts, remove each one by right-clicking its toolbar shortcut button and selecting **Delete** from the context menu. Click **OK** in the confirmation dialog box to remove the shortcut.

Shortcut Labels on Toolbar

You can hide the shortcut labels on the toolbar. Right-click in the **Shortcuts** toolbar. From the context menu, select **Show Labels**, which clears the check mark next to the item. The shortcut icons appear on the toolbar without labels.

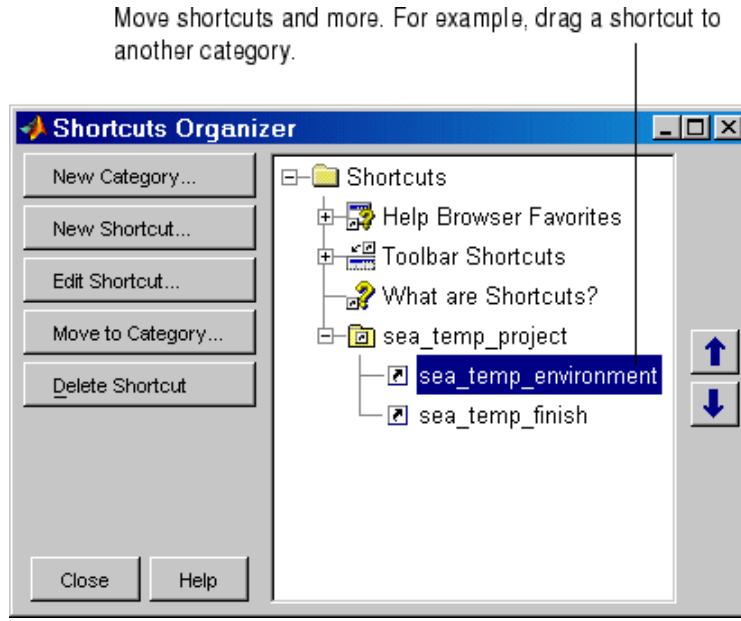
When you move the mouse over a shortcut icon, its label appears as a tooltip. To make labels display in the toolbar, right-click the toolbar and select **Show Labels**, which adds a check mark next to the item and displays the labels.

Organizing and Editing Shortcuts

To create categories for shortcuts, and to move, edit, and delete shortcuts, perform these steps:

1 Select **Shortcuts > Organize Shortcuts** from the **Start** button. Alternatively, access it via the shortcuts toolbar context menu.

The Shortcuts Organizer dialog box appears. When a shortcut category is selected in the dialog box, the **Edit Shortcut** button is replaced by the **Rename Category** button.



2 Use the buttons in the dialog box to edit and organize shortcuts and categories. You can also right-click an item and select an action from the context menu.

Changes take effect immediately.

3 Click **Close**.

For more information about using the Shortcuts Organizer dialog box, click the **Help** button.

Keyboard Shortcuts

In this section...

Keyboard Shortcuts (Accelerators or Hot Keys) and Mnemonics

You can access many of the menu items using shortcut keys (sometimes called accelerators or hot keys) for your platform. For example, use the **Ctrl+X** shortcut to perform a cut on Windows platforms. Many of the menu items show the shortcuts. Additional standard shortcuts for your platform usually work but only one is listed with each menu item.

See additional shortcuts for the Command Window at "Keyboard Shortcuts in the Command Window" on page 3-26, and for the Editor/Debugger at "Keyboard Shortcuts in the Editor/Debugger" on page 6-59.

Instructions in the documentation specify shortcuts using the Windows **Ctrl+** key convention, but with Macintosh key bindings selected, you can use the Command key instead. On the Macintosh, to make full use of all keyboard shortcuts, you need to select the **Full Access** system preference for **Keyboard Shortcuts**.

You can also use mnemonics to access menu items and buttons, such as **Alt+F** to open the **File** menu. This is not supported on the Macintosh platform. Mnemonics are listed with the menu item or button. For example, on the **File** menu, the **F** in **File** is underlined, which indicates that **Alt+F** opens the menu. In the Profiler, the **R** in the **Run this code** toolbar field is underlined, indicating that **Alt+R** moves the cursor to this field.

Note that some versions of Windows do not automatically show the mnemonics on the menu. For example, you might need to hold down the **Alt** key while the tool is selected in order to see the mnemonics on the menus and buttons. In

Windows 2000, go to **Display Control Panel**, select **Effects**, and clear the item **Hide keyboard navigation indicators until I use the Alt key**. See your Windows documentation for details.

Following are some general shortcuts that are not listed on menu items.

Key	Result
Enter	The equivalent of double-clicking, **Enter** performs the default action for a selection. For example, press **Enter** while a statement in the Command History window is selected to run that statement in the Command Window. For buttons in tools and dialog boxes, **Enter** executes the default button (the button with a border around it). If there is no default button, press the **space bar** to execute the active button (the button with a dotted outline inside it). See "Default Button and Active Button (Button with Focus)" on page 2-42 for an illustration.
Esc (escape)	Cancels the current action. For example, if you select the **Edit** menu, the menu items display. Pressing **Esc** retracts the menu items. Pressing **Esc** in a dialog box is the same as selecting the **Cancel** button.
Tab	Advances to the next button or field in a tool or dialog box. In the Command Window, completes a statement if the tab completion preference is selected.
Space bar	For buttons in tools and dialog boxes, activates the active button. See "Default Button and Active Button (Button with Focus)" on page 2-42 for an illustration of selecting default and active buttons using keys.
+ or **-** or * on numeric keypad	Use these keys on the numeric keypad to expand and collapse items in tree views. The Help browser **Help Navigator** pane and the Command History window use tree views. Use **+** to expand the selected item, use **-** to collapse the selected item, and use * to recursively expand it, meaning open all items contained in the selected item.
Alt+S	Displays the **Start** button menu (except on Macintosh platforms).
Alt+Y	Provides access to the current directory field in the toolbar (except on Macintosh platforms).
Ctrl+Tab	Moves to the next open tool in the desktop, or to the next open group of tools tabbed together.

Key	Result
Ctrl+Shift+Tab	Moves to the previous open tool or group of tabbed tools in the desktop.
Ctrl+Page Down	Moves to the next tool within a group of tools tabbed together. In a group of documents, moves to next document.
Ctrl+Page Up	Moves to the previous tool within a group of tools tabbed together. In a group of documents, moves to previous document.
Ctrl+F6	Moves to the next tool or document (only for Windows and Solaris platforms).
Ctrl+Shift+F6	Moves to the previous tool or document (only for Windows and Solaris platforms).
Alt+F4	Closes the desktop, thereby quitting MATLAB. Or outside the desktop, closes the active window (except on Macintosh platforms).

For additional shortcuts available in the various desktop tools, see the documentation for each tool. For example, see "Keyboard Shortcuts in the Command Window" on page 3-26 and "Keyboard Shortcuts in the Editor/Debugger" on page 6-59.

Go To First Letter (Type Ahead) Feature in Desktop Tool Lists

In the Current Directory browser and Command History window, you can type a letter to move directly to the next item in the list that starts with the letter you typed. This is sometimes referred to as *type ahead*.

Default Button and Active Button (Button with Focus)

These illustrations demonstrate the default versus active button in a dialog box.

The default button has a border around it. Here, **Save** is the default button. Press the **Enter** key to execute the default button.

The active button (the button with focus) has a dotted outline inside it. Here, **Cancel** is the active button. Press the space bar to execute the active button.

Here, the **Help** button is both the default button and the active button. In some cases, the default always changes to match the active button. You can press either **Enter** or the space bar to execute the **Help** button

Other Desktop Features

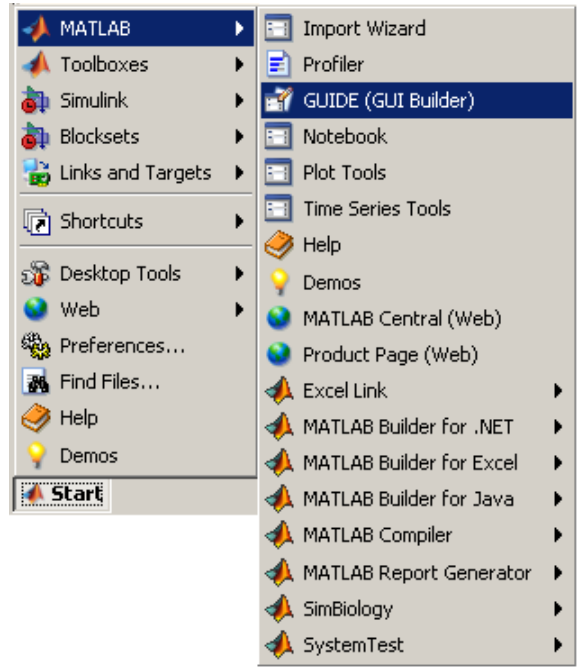
In this section...

Start Button for Accessing Tools

The MATLAB **Start** button provides easy access to tools, demos, and documentation for all your MathWorks products. From it, you can also create and run MATLAB shortcuts, which are groups of MATLAB statements.

Using the Start Button

1 Click the **Start** button to view a menu of product categories and desktop tools installed on your system. As an alternative, press **Alt+S** to view the **Start** button contents (except on Macintosh platforms). In the following illustration, MATLAB is selected.



2 From the menu and submenu items, select an item to open it. Use the icons to quickly locate a type of product or tool — see the following description of icons.

For example, select **Start > MATLAB > GUIDE (GUI Builder)** to open that tool.

Icons in the Start Button. Icons help you quickly locate a particular type of product or tool. This table describes the action performed when you select an entry with one of these icons in the **Start** button.

Icon	Description of Action When Opened
	Documentation for that product opens in the Help browser.
	Demos for the product are listed in the Help browser **Demos** pane.
	Selected tool opens.

Icon	Description of Action When Opened
	Block library opens.
	Document opens in your system Web browser.

Customizing the Start Button

You can add your own toolboxes to the **Start** button. Select **Start > Desktop Tools > View Source Files** to open the Start Button Configuration Files dialog box. For more information, click the **Help** button in the dialog box.

Menus and Context Menus

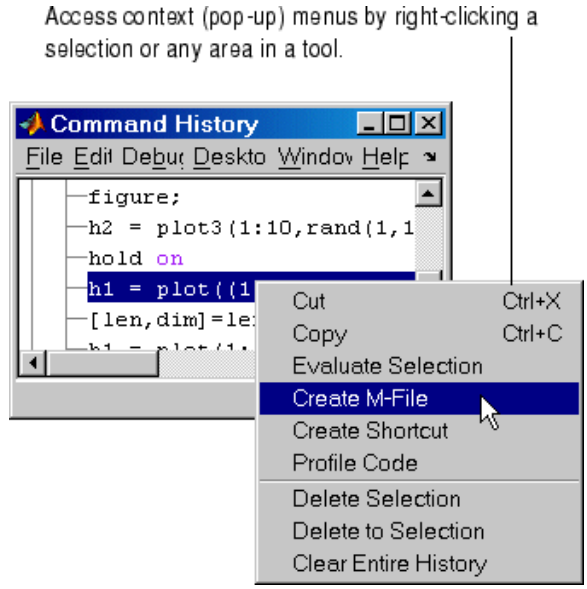
Merged Menus

When you use a tool in the desktop, its menu appears at the top of the desktop. When you work in a different tool in the desktop, you still use the menu at the top of the desktop, but the menu content changes to support that tool. When you undock a tool from the desktop, access its menu at the top of the undocked tool.

Context Menus

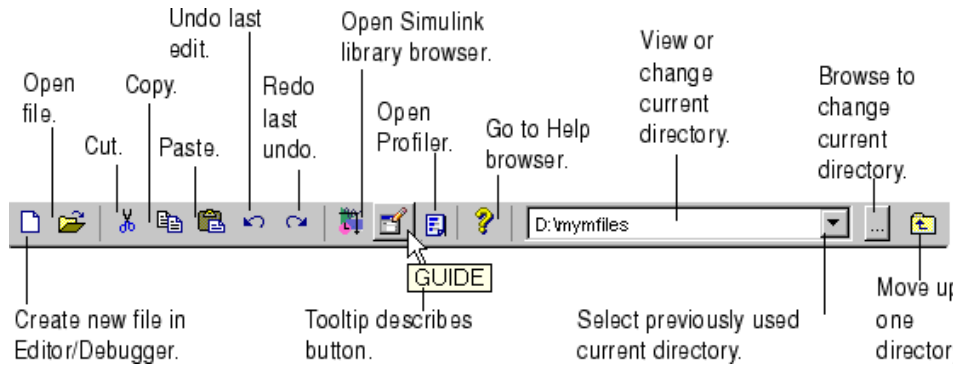
Many of the features in MATLAB desktop tools are available from context menus, also known as pop-up or right-click menus. To access a context menu, right-click a selection or an area, or press **Ctrl+Shift+F10**. The context menu for the selection or tool appears, presenting the available actions. For example, following is the context menu for a selection in the Command History window.

If a context menu does not appear, try right-clicking in a different part of the tool. When a context menu item is gray, the item does not apply to the current selection or area.



Toolbars

The toolbar in the desktop provides easy access to frequently used operations. Position the pointer over a button for a second or two and a tooltip appears that describes the item.



Some tools also have their own toolbars, which are located within the tool's own window. For example, the Current Directory browser has its own toolbar. When you undock one of these tools, the undocked tool includes the toolbar.

To hide a toolbar, or to show it again after hiding it, use the appropriate toolbar item in the **Desktop** menu. As an alternative, right-click a toolbar or menu bar and select a toolbar from the context menu to hide or show it.

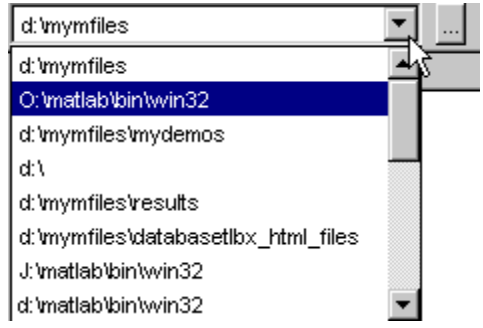
For figure windows, use the toolbar item in its **View** menu.

Current Directory Field

The current directory field in the desktop toolbar shows the MATLAB current working directory. You can change the current directory using this field and any of these methods:

- Type the new current directory directly in the field.
- Use the drop-down list to change to a previously used current directory. To specify the number of entries maintained each session, use the **History** preference you access via **File > Preferences > Current Directory**.
- Use the Browse for folder button **...** to select a new current directory.
- Use the Go Up One Level button to move the current directory up one level.

The same current directory field also appears in the Current Directory browser when the Current Directory browser is undocked from the desktop. Use the Current Directory browser to perform many additional file operations. For more information, see "File Management Operations" on page 5-35.



Status Bar

Along the bottom of the desktop is the status bar. It displays messages, such as when MATLAB is busy executing statements or when the Profiler is on. Some tools, such as the Editor/Debugger, display additional status information, such as the current line number. Not all status information appears on the status bar — many MATLAB functions and tools provide status information that is not reported to the status bar.

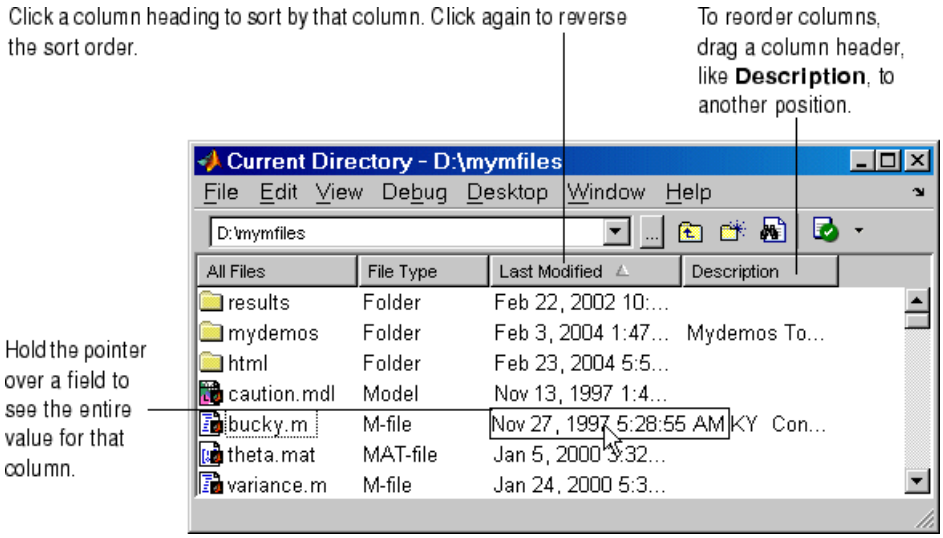
You can construct your own functions to provide status information. See the `timer` function, and search for other specific terms describing the status of interest.

Sizing, Arranging, and Sorting Columns in Tools

Some desktop tools present information in columns, such as the Current Directory browser.

To change the column width, drag the separator bar between two column headings in a tool. When a column is too narrow to show all the information in it, position the pointer over an item and the full value for that item displays like a tooltip.

To rearrange the columns in a tool, drag the column header to a different position. To sort the information by a particular column, click the column header. For example, in the Current Directory browser, click the **Last Modified** date to sort the items in date order. Some columns also allow you to reverse the sort order by clicking the column header again. A small gray arrow in the header indicates the current sort order — for example, an up arrow in the **Last Modified Date** column header indicates an ascending sort order, meaning the oldest files are at the top of the list.



Selecting Multiple Items

In many desktop tools, you can select multiple items and then select an action to perform on all the selected items. Select multiple items using the standard practices for your platform.

For example, if your platform is Windows, do the following to select multiple items:

1. Click the first item you want to select.
2. Hold the **Ctrl** key and then click the next item you want to select. Repeat this step until you have selected all the items you want. To select contiguous items, select the first item, hold the **Shift** key, and then select the last item.

Now you can perform an action on the selected items, such as delete.

To clear one of multiple selected items, use **Ctrl**+click. To clear all selected items, click outside of the selection.

Cut, Copy, Paste, and Move

You can cut and copy a selection from a desktop tool to the clipboard and then paste it from the clipboard into another tool or application. Use the **Edit** menu, toolbar, context menus, or standard keyboard shortcuts. For example, you can copy a selection of statements from the Command History window and paste them into some MATLAB desktop tools.

Use **Paste** to move items copied to the clipboard from other applications. The **Paste to Workspace** item in the **Edit** menu opens the selection on the clipboard in the Import Wizard. You can use this to copy data from another application, such as Excel, into MATLAB. For details, see the "Using the Import Wizard".

When editing in the Command Window and Editor/Debugger, to move text to a new location, select the text and drag it. To copy text, press **Ctrl** and drag the selected text to the new location.

To undo the most recent cut, copy, or paste command, select **Undo** from the **Edit** menu. Use **Redo** to reverse the **Undo**. For some tools, you can undo multiple times in succession.

See also the `clipboard` function.

Drag and Drop

You can also move or copy a selection from one tool to another by dragging the selection. For example, make a selection in the Command History window and drag it to the Command Window, which pastes it there. Edit the lines in the Command Window, if needed, and then press the **Enter** key to run the lines from the Command Window.

Another example is to drag a filename from the Current Directory browser to the Editor/Debugger to open that file in the Editor/Debugger. If you drag editable text, for example, text in the Editor/Debugger, the text is cut rather than copied. Use **Ctrl** and drag to copy rather than cut editable text.

On Windows platforms, you can drag items from external applications into MATLAB. For example, dragging text from a Microsoft Word document into the Editor/Debugger cuts and pastes it into the open file. Dragging an M-file from Windows Explorer to the Command Window runs the file. Similarly, you

can drag selections from desktop tools to other applications. For example, you can drag text from the Editor/Debugger to Microsoft Word.

Macintosh Differences in the Desktop

MATLAB on the Macintosh platform sometimes uses Macintosh GUI conventions, which might be different from what is stated in the MATLAB documentation, but the intended action should be clear. For example, if you select **File > Save** on the Macintosh, the Save dialog box that appears presents the options **Don't Save** and **Save**. On Windows and UNIX platforms, the Save dialog box presents the options **Yes**, **No**, and **Cancel**.

The standard Macintosh mouse is a single-button device. Other platforms use a mouse with more than one button. MATLAB takes advantage of these buttons. The documentation does not usually present the equivalent Macintosh instruction. When the documentation instruction is right-click, use **Ctrl**+click on the Macintosh. When the documentation instruction is middle-click, use **Command**+click on the Macintosh.

Printing and Page Setup Options for Desktop Tools

You can print from all desktop tools except the Current Directory browser, but there are some differences in usage.

To print, select **File > Print** from the tool. A Print dialog box opens. The **Properties** button in the Print dialog box is enabled for the Web and Help browsers and the Profiler, but is not enabled for the other desktop tools.

To specify standard page setup options for your platform when you print from the Command History, Workspace browser, and Array Editor, select **File > Page Setup**. A standard page setup dialog box for your platform opens.

MATLAB provides special page setup options for printing from the Command Window and Editor/Debugger. The setup options are essentially the same for both tools, with minor variations. This section covers their use:

- "Specifying Page Setup Options" on page 2-53
- "Layout Options for Page Setup" on page 2-54
- "Header Options for Page Setup" on page 2-54

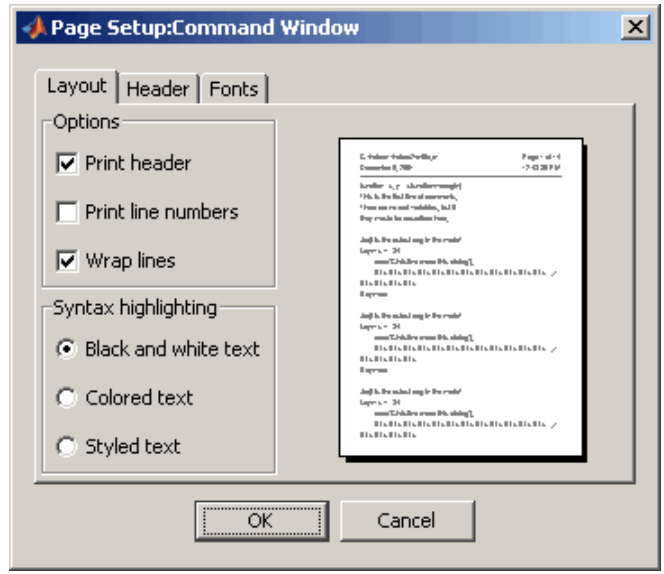
- "Fonts Options for Page Setup" on page 2-54

Specifying Page Setup Options

To specify page setup options, perform these steps:

1. In the tool you want to print from, for example, the Command Window, select **File > Page Setup**.

   The Page Setup dialog box opens for that tool.



2. Click the **Layout**, **Header**, or **Fonts** tab in the dialog box and set those options for that tool, as detailed in subsequent sections.

3. Click **OK**.

4. After specifying the options, select **File > Print** in the tool you want to print from, for example, the Command Window.

   The contents from the tool are printed, using the options you specified in Page Setup.

Layout Options for Page Setup

You can specify the following layout options. A preview area shows you the effects of your selections:

- **Print header** — Print the header specified in the **Header** pane.
- **Print line numbers** — Print line numbers.
- **Wrap lines** — Wrap any lines that are longer than the printed page width.
- **Syntax highlighting** — For keywords and comments that are highlighted in the Command Window, specify how they are to appear in print. Options are black and white text (that is, no highlighting), colored text (for use with a color printer), or styled text. For styled text, keywords appear in bold, comments appear in italics, and all other text appears in the normal style. Only keywords and comments you input in the Command Window are highlighted; output is not highlighted.

Header Options for Page Setup

If you want to print a header, select the **Layout** tab and then select **Print header**. Then select the **Header** tab and specify how the elements of the header are to appear. A preview area shows you the effects of your selections:

- **Page number** — Format for the page number, for example `# of n`
- **Border** — Border style for the header, for example, `Shaded box`
- **Layout** — Layout style for the header. For example, `Standard one line` includes the date, time, and page number all on one line

Fonts Options for Page Setup

Specify the font to be used for the printed contents:

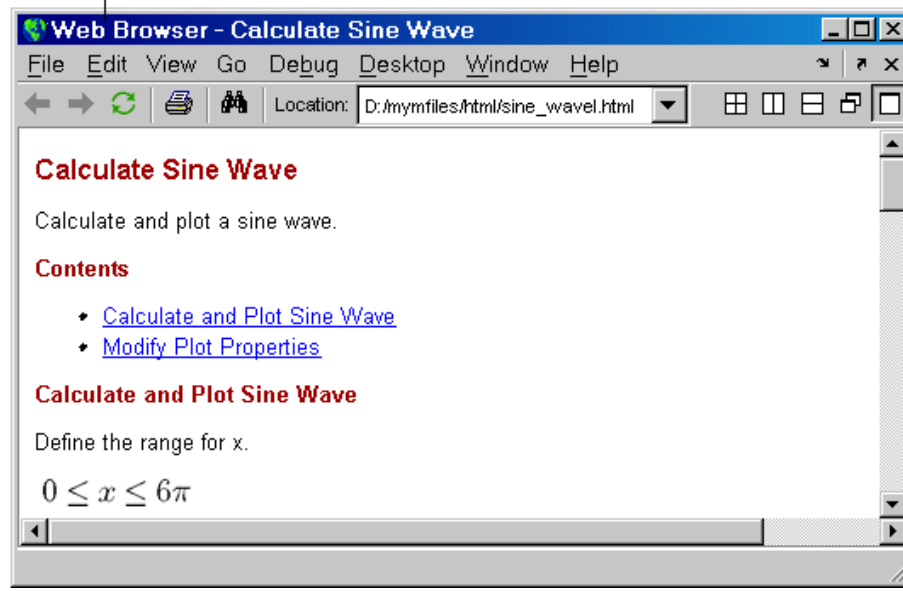
1. From **Choose font**, select the element, either `Body` or `Header`, where `Body` text is everything except the `Header`.
2. Select the font to use for that element. For example, select **Use Command Window font** for `Body` text if you want the printed text to be the same as the font that appears in the Command Window. This is the font specified in **File > Preferences > Fonts > Custom** for the `Command Window`.

3 Repeat for the other element. If you did not select **Print header** on the **Layout** pane, you do not need to specify the `Header` font. As an example, for `Header` text, select **Use custom font** and then specify the font characteristics — type, style, and size. After you specify a custom font, the **Sample** area shows how the font will look.

Web Browser

Some tools in MATLAB and related products display HTML documents in the MATLAB Web Browser. For example, after using the Editor/Debugger's cell features to publish an M-file to HTML, you view the HTML file in the MATLAB Web Browser. Because the MATLAB Web Browser is a desktop tool, you can dock it and perform other desktop operations on it.

Example of Web browser displaying results of M-file published to HTML format.



To display an HTML document in the Web Browser, double-click the document name in the Current Directory browser or use the `web` function. The `web` function supports arguments that display documents in your system browser, for example, Netscape, or in the Help browser.

The toolbar buttons and menu items in the Web Browser are similar to those found in the Help browser display pane. For more information, see “Viewing Documentation in the Help Browser” on page 4-26.

One feature of the Web Browser not found in the Help browser is the **Location** field. In the Web Browser, type a URL in the field to display that Web page.

Like any Web browser, the MATLAB Web Browser might not support all of the HTML or related features used in a particular Web site or HTML page. For example, the MATLAB Web Browser does not support the display of `.bmp` (bitmap) image files. Instead use `.gif` or `.jpeg` formats for image files in HTML pages. As another example, it does not support HTML pages you generate directly from Microsoft Word and PowerPoint.

Internet Connection and Fonts for Web Browser — Web Preferences

To specify a proxy server to connect from the MATLAB Web Browser to the Internet, use Web preferences. You might need to specify this preference if you have a firewall, for example. If you have a firewall and do not specify the proxy settings, links from the Web Browser to URLs will not work.

Select **File > Preferences > Web**. By default, the check box **Use a proxy server to connect to the Internet** is not selected. This is for when you have a direct connection to the Internet.

To specify a proxy server, select the check box and specify the **Proxy host** and **Proxy port**. See your system administrator for the information you need to specify the proxy settings. As an example, `172.16.10.8` illustrates the format for host, and `3128` is the type of value you enter for port.

Fonts for Web Browser. To modify the font used in the Web Browser, select **File > Preferences > Fonts**. The Web Browser uses the font settings you specify for HTML Proportional Text tool. For more information about setting fonts, click the **Help** button in the preference pane for **Fonts**.

Accessing The MathWorks on the Web

You can access popular MathWorks Web pages from the MATLAB desktop. Select one of the following items from the **Help > Web Resources** menu.

For most items, the selected Web page then opens in your default system Web browser, for example, Netscape:

- **The MathWorks Web Site** — Home page of the MathWorks Web site (`http://www.mathworks.com`).
- **Products & Services** — MathWorks Products and Services page (`http://www.mathworks.com/products/`) with information about the full family of products.
- **Support** — MathWorks Support page (`http://www.mathworks.com/support`) where you can look for solutions to problems you are having, or report new problems.
- **MathWorks Account**
  - **Login in or Create Account** — Login page for MathWorks Account (`http://www.mathworks.com/accesslogin/`). If you are registered, your main account page displays. Otherwise, you are directed to a page where you register online. Registration allows you to view your product registration and license information and helps you stay up to date on the latest MATLAB developments.
  - **Get Passcodes and Manage Licenses** — If you have a MathWorks Account, displays your Licenses page.
  - **Get Product Trials** — If you have a MathWorks Account, provides access to trial versions of products.
- **MATLAB Central** — MATLAB Central Web site (`http://www.mathworks.com/matlabcentral/`) for the MATLAB user community. It includes MATLAB contest entries and results, a MATLAB screen saver, and these technical resources:
  - **MATLAB File Exchange** — Code library of files contributed by MathWorks customers and employees, available for free download and use with MathWorks products.
  - **MATLAB Newsgroup Access** — Provides access to the Usenet newsgroup for MATLAB and related products, `comp.soft-sys.matlab`, where you can post and answer questions, as well as view the archives.
- **MATLAB Newsletters** — Access to online versions of News and Notes and MATLAB Digest. News and Notes is published twice a year and contains feature articles, technical notes, and product information for MATLAB

users. MATLAB Digest, an electronic bulletin consisting of technical notes, solutions, and timely announcements to the user community, is issued more frequently. See `http://www.mathworks.com/company/newsletters`.

Check for Updates

This features allows you to easily determine if more recent versions of your MathWorks products are available. Select **Help > Check for Updates**. A dialog box appears, listing the version numbers of all MathWorks products installed on your system. Click **Check for Updates** in the dialog box, which accesses the MathWorks Web site and reports back for each product if a newer version is available or if your version is the latest.

Terms of Use and Patents

Access the terms of use and patent information for MathWorks products.

Preferences

In this section...

Setting Preferences

Use preferences to specify options for MATLAB tools, as follows:

1. Select **File > Preferences**.
2. In the left pane of the Preferences dialog box, preferences appear for MATLAB tools as well as for any other MathWorks products installed on your system.

   Choose a tool and click the **+** to display more preferences for that tool. From the expanded list, select the entry you want. The right pane shows the preferences for that item.
3. Change settings. Click **Apply** or **OK** to set the preferences. Preferences take effect immediately. They remain persistent across MATLAB sessions.

Note that some tools allow you to control these settings from within the tool without setting a preference. Use that method if you only want the change to apply to the current session.

Function Alternative

Open the Preferences dialog box using the `preferences` function.

Summary of Preferences

Preference	What You Can Specify
General Preferences	Toolbox path caching, figure window printing, `delete` function behavior, MAT-file save formats, confirmation dialogs, source control, and multithreaded computation.
Keyboard	Key bindings, tab completion, and delimiter matching for the Command Window and the Editor/Debugger.
Fonts	Font type, style, and size for desktop tools. Customize for any tool.
Colors	Colors for text, background, syntax highlighting, and hyperlinks in desktop tools.
M-Lint	Show or hide M-Lint messages in the Editor/Debugger M-Lint automatic code analyzer and in the M-Lint Code Check Report.
Command Window	Numeric format and display, accessibility, and tab size.
Command History	Display, filtering, and saving.
Editor/Debugger	Editor type, startup options, display, tab size and indenting, language, including M-Lint messages, publishing, and autosave.
Help	Product filter and synchronization.
Web	Internet proxy server settings.
Current Directory	Number of entries in history and display options.
Array Editor	Numeric format, use of **Enter** key, and decimal separator.
Workspace	Statistical calculation options.
GUIDE	Display options for GUI-building tool.
Time Series Tools	Property Editor dialog and x-axes warning dialog.
Figure Copy Template	Application, text, line, uicontrols, axis, format, background color, and size.
Other products	Preferences for other installed MathWorks products.

Preferences File — matlab.prf

Preferences are stored in a preferences file, `matlab.prf`. Type `prefdir` in the Command Window to see the full pathname for the preferences directory that contains `matlab.prf`. The preference directory also contains related files.

On Macintosh platforms, the directory might be in a hidden folder, for example, `myname/.matlab/R2007b`. To access the directory, select **Go > Go to Folder** in the Mac OS Finder. In the resulting dialog box, type the path returned by `prefdir` and press **Enter**.

The `matlab.prf` file is loaded when MATLAB starts and is overwritten when you close MATLAB.

The exact name of the preferences directory that MATLAB uses depends on the release. When you install a new version of MATLAB, MATLAB tries to use your existing preferences from the previous version, where possible. For more information on the preference directory filename and the preference migration process, see the reference page for `prefdir`.

Fonts Preferences for Desktop Tools

In this section...

Setting Desktop Fonts

Use desktop font preferences to specify the font characteristics for MATLAB desktop tools. The font characteristics are

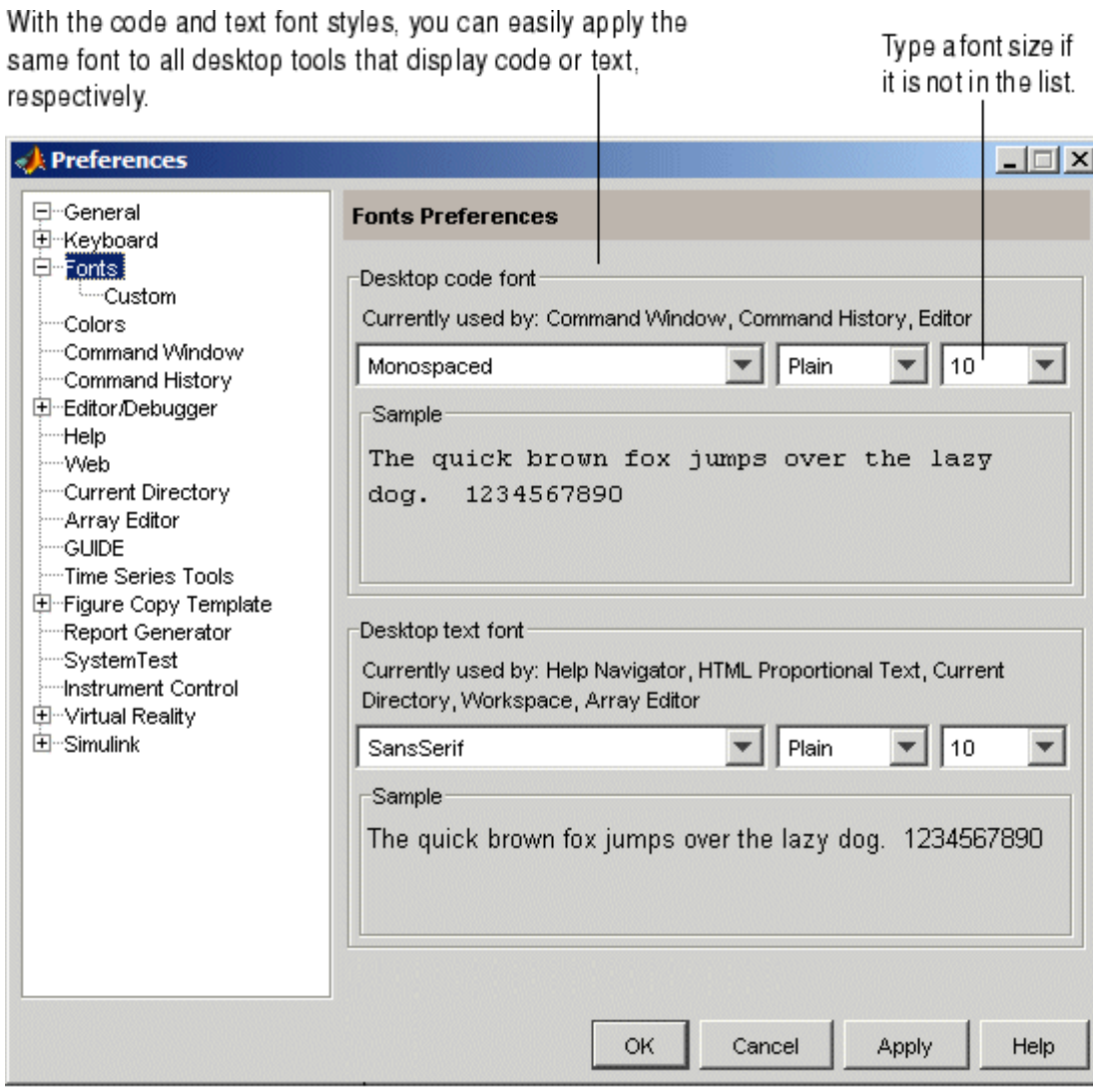
- Name (also called family or type), for example, select SansSerif
- Style, for example, select bold
- Size in points, for example, type 11 points

Select **File > Preferences > Fonts** to set fonts for desktop tools. You can specify the font to be used by all tools that primarily display code such as the Command Window, and specify the font to be used by all other desktop tools. Or you can separately specify the font for any desktop tool.

Select the font characteristics from the lists shown. For font size, not all entries are shown. You can type in a size, including one not shown.

You can set some font options differently for printing — see "Printing and Page Setup Options for Desktop Tools" on page 2-52.

For information about making additional fonts available to MATLAB, see "Making Fonts Available to MATLAB" on page 2-69.



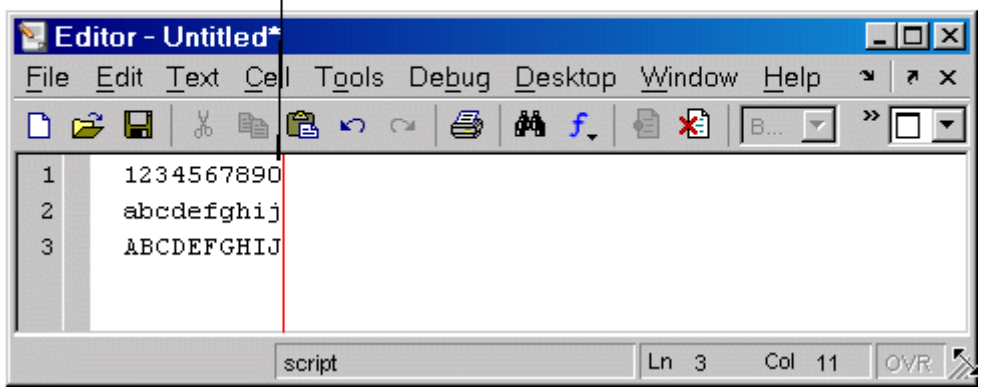
Desktop Code Font and Desktop Text Font

You specify separate font characteristics for tools that primarily display code (**Desktop code font**), such as the Command Window, and tools that primarily display text (**Desktop text font**), such as the Current Directory

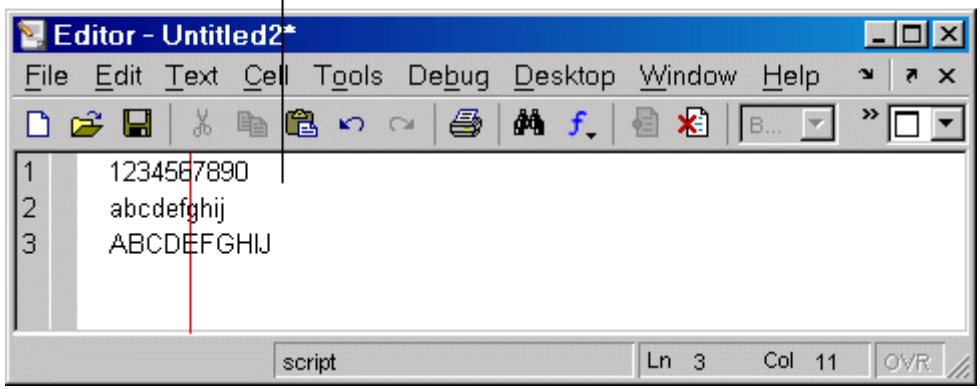
browser. Many users prefer that code display in a monospace font to provide better alignment, and prefer a more narrow font style for text information. With the desktop code font preference, you set just one preference to apply a monospace style to all tools that display code (except the Help and Web Browsers). Similarly, you can set just one preference to apply a text font to all desktop tools that display text.

The following illustrations show how the Editor/Debugger would look using a monospace font and a proportional font. Note that a monospace font is useful when you care about alignment, but a proportional font uses less space.

With a monospaced font, all characters are the same width. Here, the font is 10 pt. Monospace. Note the 10th character in each line aligns with the Editor/Debugger's right-hand text limit, which is set to column 10.



With a proportional font, characters are different widths. Here, the font is 10 pt. SansSerif. Each line contains 10 characters but the length of each line differs. The Editor/Debugger's right-hand text limit is not relevant.



Default Font Settings

Default settings are listed in the following table. Note that Lucida Console approximates the `fixedsys` font available in earlier versions of MATLAB.

Font Type	Default Characteristics and Sample	Tools Using Font Type by Default
Desktop code font	Monospaced, Plain, 10 point `Sample code font`	• Command History • Command Window • Editor/Debugger (which also applies to the Shortcuts Editor)
Desktop text font	SansSerif, Plain, 10 point Sample text font	• Array Editor • Current Directory browser (which also applies to the Path browser) • Help Navigator • HTML Proportional Text. This is the font used for noncode text in the Web browser (including, for example, HTML reports generated from cell publishing), Profiler, and Help browser display pane. While you can select the font name, you cannot change the font style (for example, to bold or italic) for HTML Proportional Text. Changes to size affect noncode and code text. • Workspace browser

When you change a font characteristic for **Desktop code font**, the characteristic takes effect for all tools that use the desktop code font. The same is true when you change a font characteristic for **Desktop text font**.

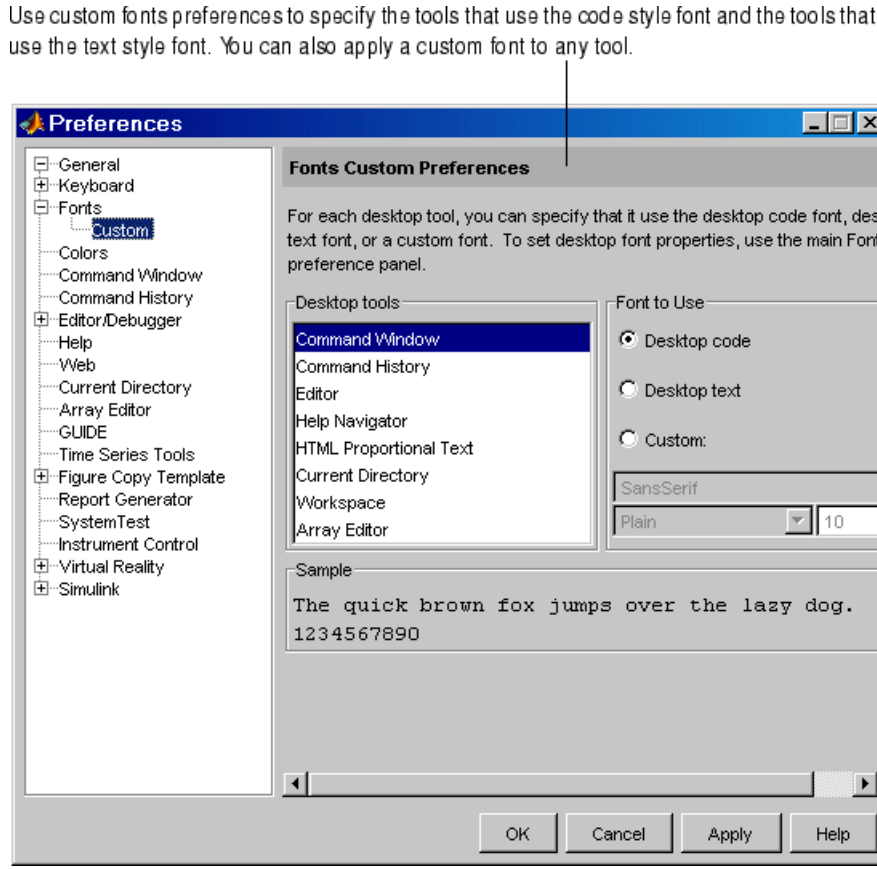
After changing a characteristic, a sample in the dialog box shows how it will look. Click **Apply** or **OK** to make the change take effect in the desktop tools.

See Also

"Preferences" on page 2-59

Custom Fonts Preferences

If you do not want to use the current settings for "Desktop Code Font and Desktop Text Font" on page 2-63, you can specify that a tool use the code font, the text font, or a different font. Select **File > Preferences > Fonts**. Click **+** and select **Custom**. The **Fonts Custom Preferences** pane appears.



Select a tool from the **Desktop tools** list. The type of font it uses, code or text, appears under **Font to Use**. In the illustration shown, the Command Window uses the **Desktop code font**, which is defined in the **Fonts** pane as described in the previous section.

To change the font characteristics the selected tool uses, select a different radio button. For **Custom**, you then specify the font characteristics for that tool.

Changing the Font — Example

This example changes the default settings (see "Default Font Settings" on page 2-65) for the desktop code font, changes the Command History font preference so that it uses the desktop text font instead of the code font, and specifies a custom font for the Current Directory browser:

1. Change the characteristics for the desktop code font. On the **Fonts** pane, set the **Desktop code font** to `Times New Roman`, `Plain`, `14` point. Use the default for the **Desktop text font**, `SansSerif`, `Plain`, `10` point. Click **Apply**.

2. Make the Command History window use the desktop text font. Select **Fonts**, click **+**, select **Custom**, and then select `Command History` from **Desktop tools**. Select the **Desktop text** radio button.

3. Apply a custom font to the Current Directory browser. Select `Current Directory` from **Desktop tools.** Select the **Custom** radio button. Select `Arial Narrow` and `Plain`, and type `11` in the size field. Click **OK**.

The following table details the results of the changes.

Tool	Font Type	Font Characteristics
Command Window	Desktop code	Times New Roman, Plain, 14 point
Command History	Desktop text	SansSerif, Plain, 10 point
Editor/Debugger	Desktop code	Times New Roman, Plain, 14 point
Help Navigator	Desktop text	SansSerif, Plain, 10 point
HTML Proportional Text	Desktop text	SansSerif, Plain, 10 point
Current Directory	Custom	Arial Narrow, Plain, 11 point

Tool	Font Type	Font Characteristics
Workspace	Desktop text	SansSerif, Plain, 10 point
Array Editor	Desktop text	SansSerif, Plain, 10 point

See Also

For help about how MATLAB stores preferences and help for other preferences, see “Preferences” on page 2-59.

Antialiasing for Desktop Fonts on Linux/UNIX

To give the desktop a smoother appearance in Linux/UNIX, select the antialiasing preference on the **Preference > Fonts** pane. The preference apply to all fonts.

Note The antialiasing option is not necessary on Windows or Mac, because MATLAB follows the operating system’s font settings on these platforms.

Making Fonts Available to MATLAB

On Windows platforms, desktop components (such as the Command Window and Workspace browser), figure windows, and uicontrols support only TrueType and OpenType fonts. Some graphics objects can render bitmapped fonts as well, such as `xlabel`, `ylabel`, `title`, and `text`.

To make a new compatible font available to MATLAB, install the font by selecting **Start > Control Panel > Fonts** in the Windows desktop, and then selecting **File > Install New Font**. Restart MATLAB so that it can use the font.

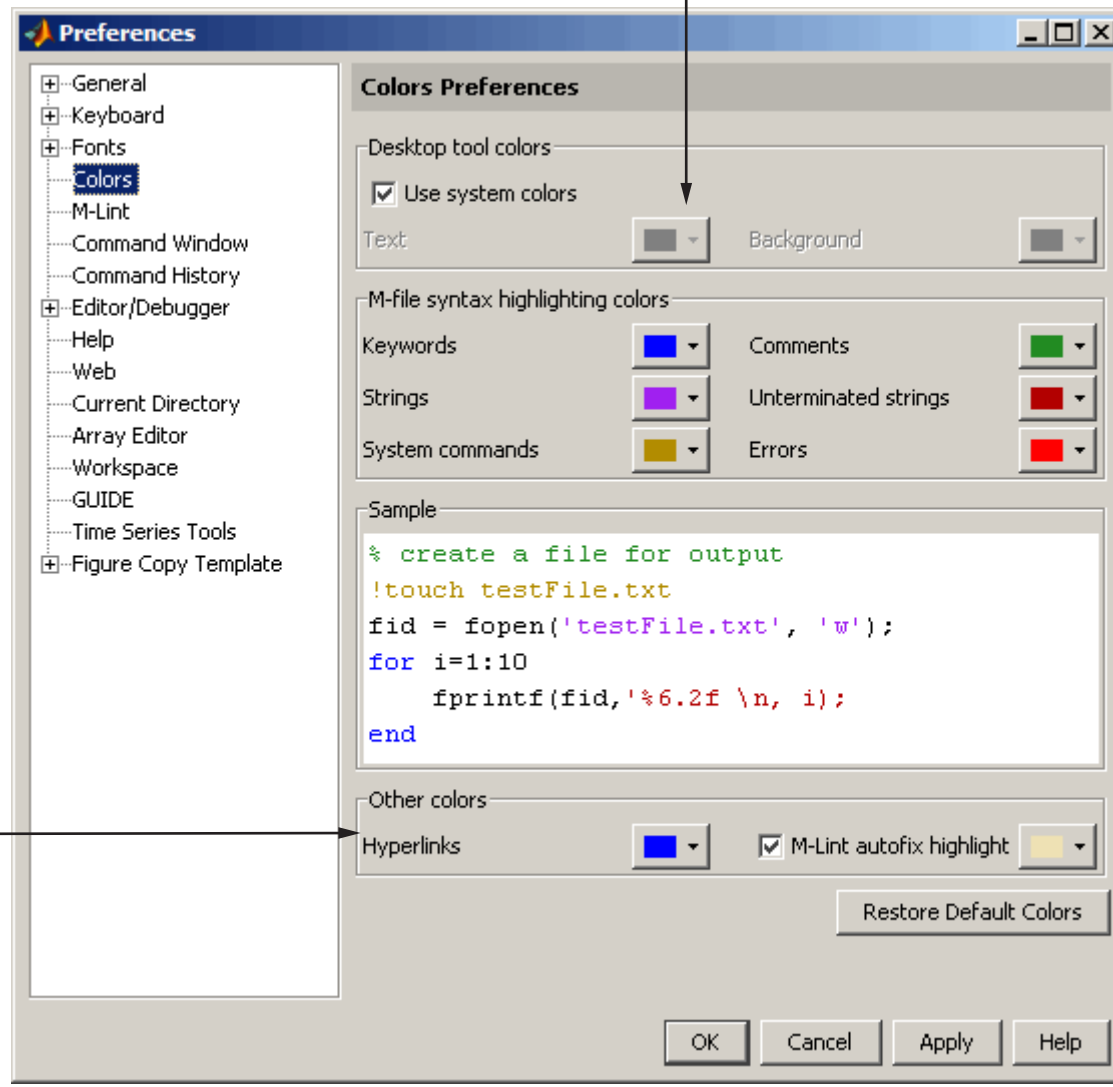
Colors Preferences for Desktop Tools

In this section...

Setting Colors Used in Desktop Tools

Desktop color preferences specify the colors used in MATLAB desktop tools and the colors that convey syntax highlighting. Select **File > Preferences > Colors** to set color preferences for desktop tools. You can set some color options differently for printing — see "Printing and Page Setup Options for Desktop Tools" on page 2-52.

To set colors for text and the background, clear the **Use system colors** check box and then select colors from the palettes.



Specify the color of hyperlinks in the Command Window and Help browser Index pane.

Desktop Tool Colors

Use **Desktop tool colors** to change the color of the text and background in the desktop tools. The colors also apply to the Import Wizard. The colors do not apply to the HTML display pane nor to the Web Browser.

Select the check box **Use system colors** if you want the desktop to use the same text and background colors that your platform (for example, Windows) uses for other applications.

To specify different text and background colors, follow these steps:

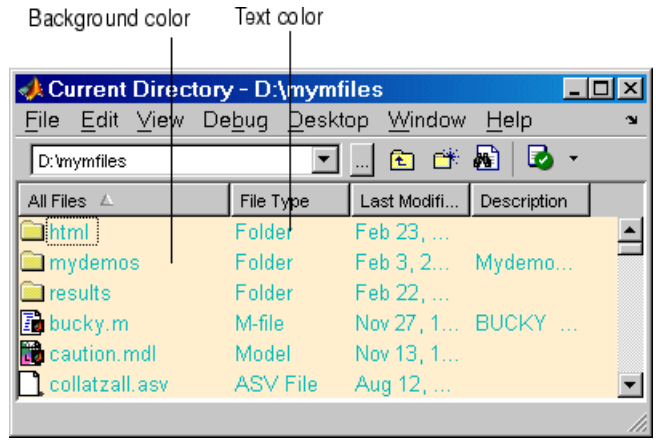
1. Clear the **Use system colors** check box.
2. Click the arrow next to the **Text** color and choose a new color from the palette shown.

   When you choose a color, the **Sample** area in the dialog box updates to show you how it will look.
3. Click the arrow next to the **Background** color and choose a new color.

   If you use a gray background color, a selection in an inactive window will not be visible.
4. Click **Apply** or **OK** to see the changes in the desktop tools.

Click **Restore Default Colors** to return to the default settings for desktop tool colors, as well as for syntax highlighting colors.

The following illustration shows how the Current Directory browser looks with blue-green text and a beige background. These colors are only discernible in the online version of this documentation.



Gray Background Color

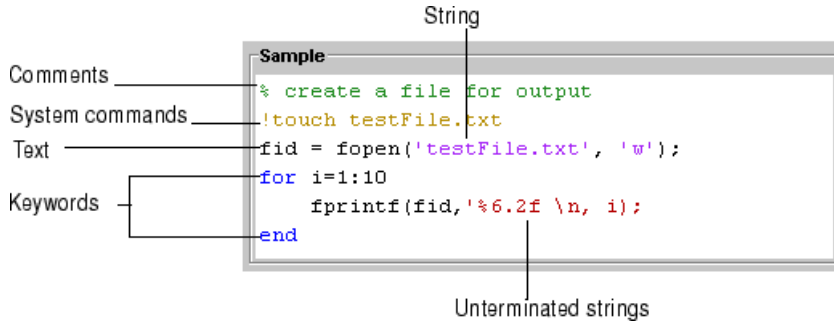
For some UNIX platforms, there is a gray background color for desktop tools, such as the Editor/Debugger. This occurs when the preference for **Desktop tool colors** is set to **Use system colors**, and the system's window manager uses gray as the background color default. To change the color, clear the check box for **Use system colors** and then select a new **Background** color from the palette.

Syntax Highlighting Colors

In the Command Window, Command History, Editor/Debugger, and Shortcuts callback area, MATLAB conveys syntax information via different colors to help you easily identify elements, such as `if/else` statements. This is known as syntax highlighting.

In the Command Window, only the input you type is highlighted; output from running MATLAB functions is not highlighted. In the Editor/Debugger, you can specify syntax highlighting preferences for use with files in M, C/C++, Java, and HTML. For details, click the **Help** button in the Preferences dialog box for the Editor/Debugger to see Language Preferences in the online documentation.

Use preferences to specify the syntax highlighting colors. When you choose a color, the **Sample** area in the dialog box updates to show you how it will look.



The default colors are listed here:

- **Keywords** — Flow control functions, such as `for` and `if`, as well as the continuation ellipsis (`...`), are colored blue.
- **Comments** — All lines beginning with a `%`, designating the lines as comments in MATLAB, are colored green. Similarly, the block comment symbols, `%{` and `%}`, as well as the code in between, appear in green. Text following the continuation ellipsis on a line is also green because it is a comment.
- **Strings** — Type a string and it is colored maroon. When you complete the string with the closing quotation mark (`'`), it becomes purple. Note that for functions you enter using command syntax instead of function syntax, the arguments are highlighted as strings. This is to alert you that in command notation, variables are passed as literal strings rather than as their values. For more information, see “MATLAB Command Syntax” in the MATLAB Programming documentation.
- **Unterminated strings** — A single quote without a matching single quote, and whatever follows the quote, are colored maroon. This might alert you to a possible error.
- **System commands** — Commands such as `!` (shell escape) are colored gold.
- **Errors** — Error text that appears after you run code, including any hyperlinks, is colored red.

Click **Restore Default Colors** to return to the default settings for syntax highlighting colors and desktop tool colors.

Other Colors

Specify the color for **Hyperlinks**, which applies to links in the Command Window and Help browser **Index** pane. If you use a dark background color for those tools, be sure to use a light or other contrasting color for hyperlinks so that you can see them.

With the **M-Lint autofix highlight** preference selected, code that M-Lint can automatically correct is highlighted in the Editor/Debugger. Use the palette to change the highlight color. For more information, see “M-Lint Automatic Code Analyzer in the Editor/Debugger” on page 6-88.

See Also

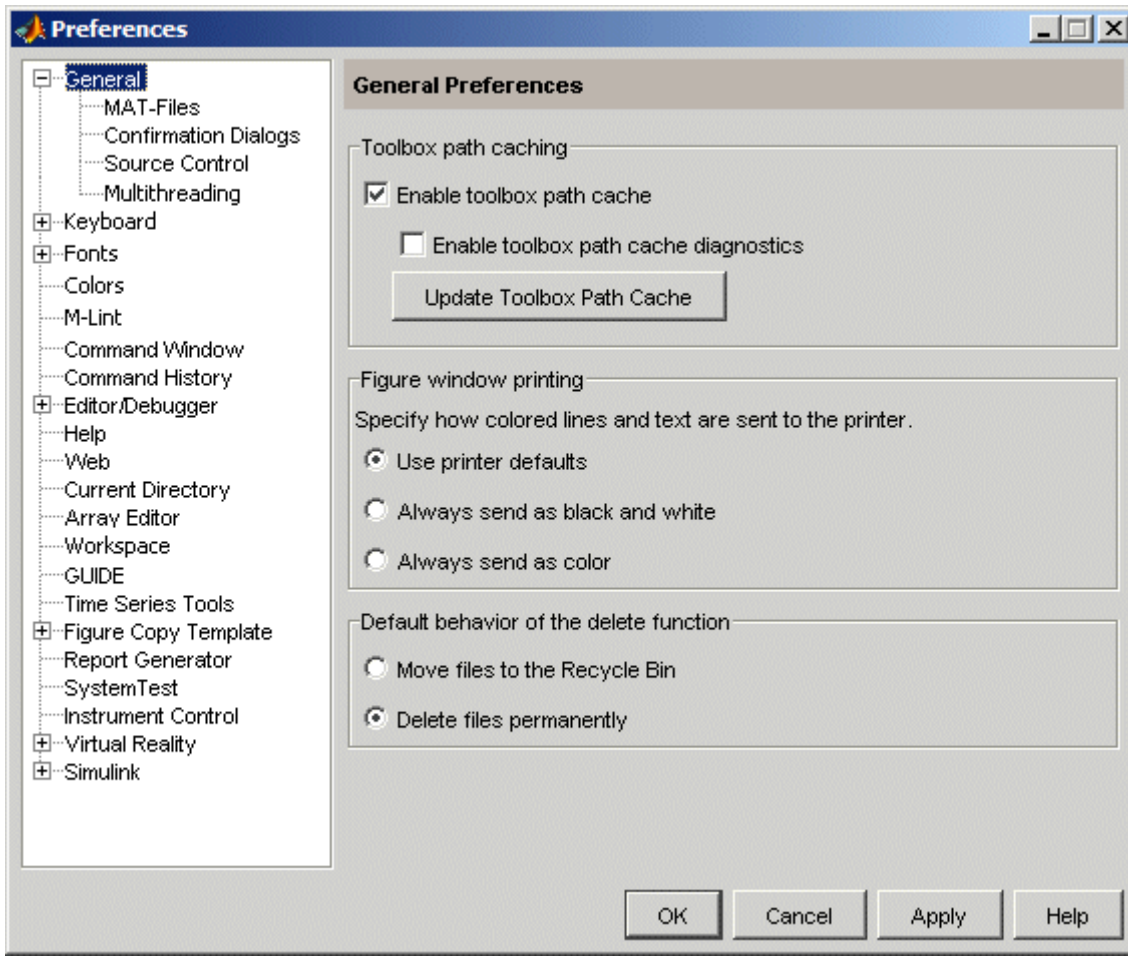
For information about other preferences and how MATLAB stores preferences, see “Preferences” on page 2-59.

General Preferences for MATLAB

In this section...

Setting General Preferences for MATLAB

Select **File > Preferences > General** from any desktop tool to access **General Preferences**.



These preferences apply to all relevant tools in MATLAB.

- Toolbox path caching preference — see “Toolbox Path Caching in MATLAB” on page 1-17
- Figure window printing — see “Printing and Exporting” in MATLAB Graphics documentation
- “Default Behavior of the Delete Function” on page 2-78

- "MAT-Files Preferences" on page 2-79
- "Confirmation Dialogs Preferences" on page 2-81
- Chapter 9, "Source Control Interface"
- "Multithreading Preferences" on page 2-84

Default Behavior of the Delete Function

Files you delete using the `delete` function are permanently removed by default. There is no opportunity to retrieve them.

You can use this preference to instead move deleted files to the Recycle Bin on Windows, to the Trash Can on Macintosh, or to a `tmp` directory on UNIX platforms. Then, you can recover any accidentally deleted files from these locations. Deleted files in these locations are not automatically removed; you must remove them using operating system features, such as **Empty Recycle Bin** on Windows. When you select this preference, `delete` might run slower.

Function Alternative

The MATLAB delete preference actually sets the state of the `recycle` function upon startup and when you change the preference. You can override the behavior of the preference by setting the `recycle` function state. For example, regardless of the preference setting, when you run

```
recycle('off')
delete('thisfile.m')
```

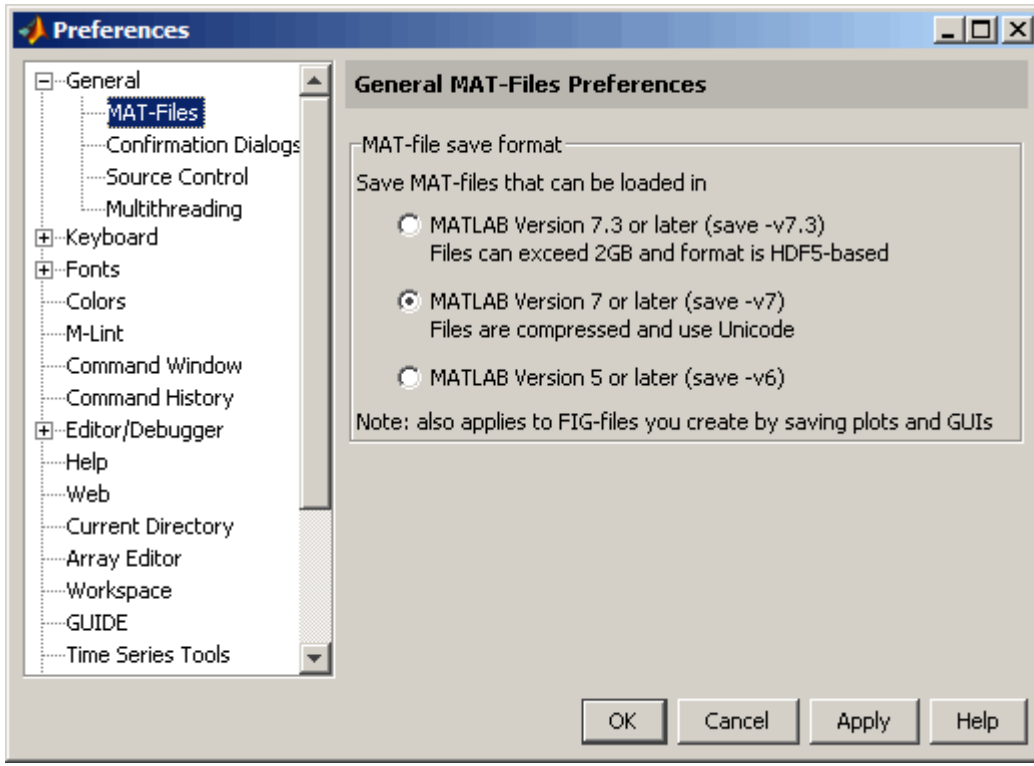
MATLAB permanently removes `thisfile.m` from the current directory. Files you subsequently remove using `delete` are also permanently removed, unless you reapply the preference to recycle or run `recycle('on')`. Regardless of the state of the `recycle` function when you end a session, the next time you start MATLAB, the setting for the preference is honored. For more information, see the `recycle` and `delete` reference pages.

Note that this preference and the `recycle` function do not apply to files you delete using the Current Directory browser. For more information, see "Cutting or Deleting Files and Directories" on page 5-45.

MAT-Files Preferences

The **MAT-file save format** sets the default version compatibility option MATLAB uses when saving MAT-files. Use these options if you use multiple versions of MATLAB or share MAT-files with others who run a different version of MATLAB. The setting applies when you use the `save` function as well as when you use**Save** menu items for MAT-files, such as **File > Save Workspace As** from any desktop tool.

The MAT-file preference also applies to saving FIG-files, which include plots, as well as GUIs you create with GUIDE.



Options are

- **MATLAB Version 7.3 or later (save -v7.3)**—Starting in MATLAB Version 7.3, you can save data that is larger than 2 GB on platforms that

allow it, which is the primary purpose of this option. Using this option is equivalent to running `save -v7.3`. This format of the resulting MAT-file is HDF5-based. You cannot load these MAT-files into any versions prior to MATLAB Version 7.3; in those cases, use one of the other two options.

- **MATLAB Version 7 or later (save -v7)**—Starting in MATLAB Version 7, MATLAB compresses the data when saving a MAT-file, thereby reducing the storage space required. When you load the MAT-file, MATLAB automatically uncompresses the data. In addition, MATLAB uses Unicode character encoding for strings when you save a MAT-file, making the data accessible to other MATLAB users, regardless of the default character encoding scheme used by their systems. MAT-files saved with this option work in all MATLAB 7 versions. Using this option is equivalent to running `save -v7`.
- **MATLAB Version 5 or later (save -v6)**—Releases of MATLAB prior to Version 7 did not save compressed MAT-files. They also did not use Unicode character encoding, which sometimes prevented the exchange of MAT-files among users, particularly when they used localized systems. Specify this option to save MAT-files for use with versions prior to MATLAB Version 7. Using this option is equivalent to running `save -v6`.

Like other preferences, the **MAT-file save format** preference gets its initial value from the preference file for the previous installed version. For example, if the setting in your MATLAB 7.4 preference is `-v6`, when you upgrade from MATLAB Version 7.4 (R2007a) to MATLAB Version 7.5 (R2007b), the initial value in Version 7.5 is `-v6`.

If you upgrade from a version prior to MATLAB Version 7.3, or if you do not have a previous MATLAB version installed, the initial value is `-v7`.

Note For more information about MAT-file save formats, including restrictions, see Version Compatibility Options and Remarks in the `save` reference page.

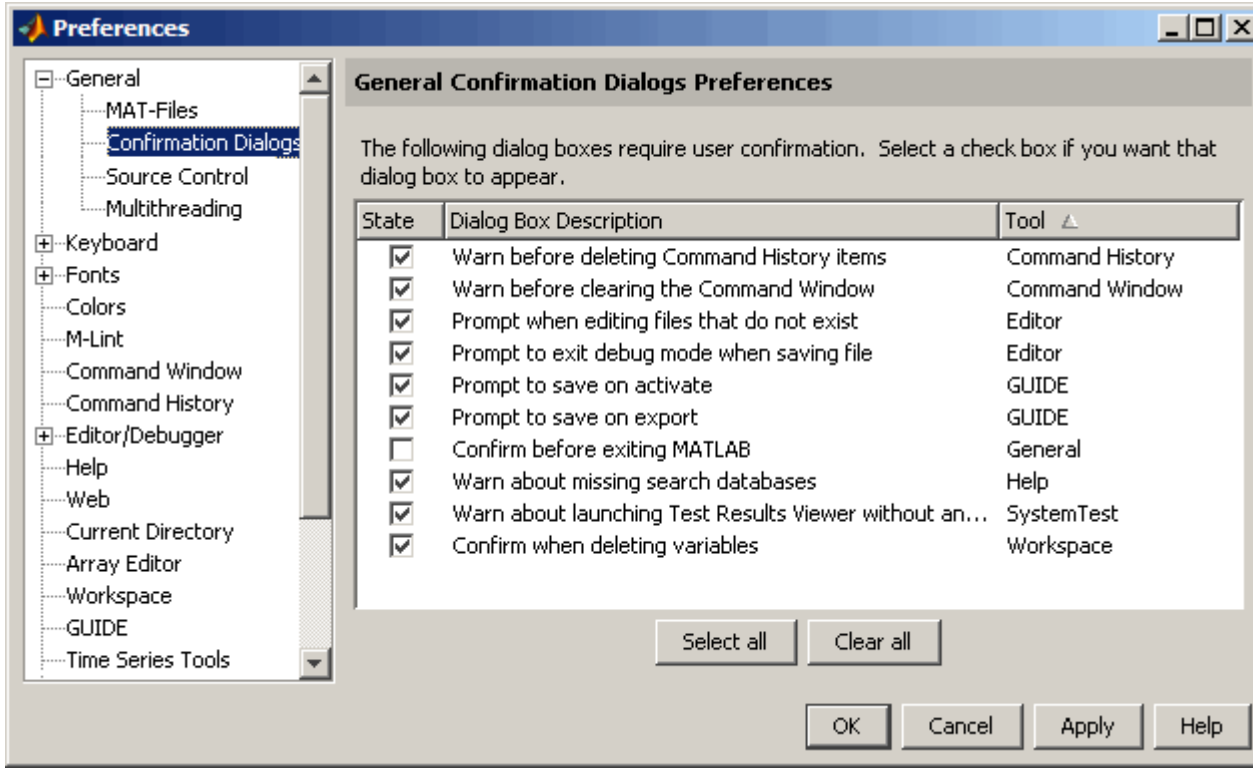
Function Alternative

You can override the **MAT-file save format** preference by using the `save` function with a specified version compatibility option. For occasional use, this

might be more convenient than changing the preference. For example, use `save` with the `-v6` option to ensure compatibility with MATLAB versions prior to Version 7. For more information, see the `save` reference page.

Confirmation Dialogs Preferences

These preferences instruct MATLAB to display or not display specific confirmation dialog boxes.



When the check box for a confirmation dialog is selected and you perform the action it refers to, the confirmation dialog box appears. If you clear that check box, the dialog box does not appear when you perform the action.

When the confirmation dialog box does appear, it includes a **Do not show this prompt again** check box. If you select the check box in the dialog box, it automatically clears the check box for the confirmation preference.

For example, select the check box **Warn before deleting Command History items**. Then select **Edit > Delete Selection** in the Command History, MATLAB displays the following confirmation dialog box.

If you select the **Do not show this prompt again** check box and click **OK**, the confirmation dialog box will not appear the next time you delete items from the Command History window. In addition, the **Warn before deleting Command History items** check box in the **Confirmations Dialogs** preferences pane is cleared.

The following table summarizes the confirmation dialog boxes.

Confirmation Dialogs Check Box Item	About the Confirmation Dialog Box	For More Information
Warn before deleting Command History items	Appears when you delete entries from the Command History window.	"Deleting Entries from the Command History Window" on page 3-57
Warn before clearing the Command Window	Appears when you clear the Command Window content using menu items. Does not appear when you use the `clc` function.	"Clearing the Command Window" on page 3-32

Confirmation Dialogs Check Box Item	About the Confirmation Dialog Box	For More Information
Prompt when editing files that do not exist	Appears when you type `edit filename`, if `filename` does not exist in the current directory or on the MATLAB path.	"Function Alternative" on page 6-9
Prompt to exit debug mode when saving file	Appears when you try to save a modified file while in debug mode.	"Ending Debugging" on page 6-119
Prompt to save on activate	Appears when you have unsaved changes to a figure and M-file, and then activate the GUI, by clicking the **Run** button, for example.	"GUIDE Preferences" in the GUIDE documentation
Prompt to save on export	Appears when you have unsaved changes to a figure and M-file, and then select **File > Export**.	"GUIDE Preferences" in the GUIDE documentation
Confirm before exiting MATLAB	Appears when you quit MATLAB.	Quitting MATLAB
Warn about missing search databases	Appears if you have help files in the Help browser for non-MathWorks products and the search database for those files has not been updated for the version of MATLAB you are running.	Contact the provider of the help files to obtain the correct version of the search database. Without the most current version, you can use the help files in the Help browser, but the Help browser search will not include those files in search results.
Confirm when deleting variables	Appears when you delete variables from the workspace using menu items. Does not appear with the `clear` function.	"Deleting Workspace Variables" on page 5-8

Multithreading Preferences

If you run MATLAB on a multiple-CPU system (multiprocessor or multicore), you can use multithreaded computation, which can improve performance for some operations. For more information, see .

Accessibility

In this section...

Software Accessibility Support

MathWorks products includes a number of modifications to make them more accessible to all users. Software accessibility support for blind and visually impaired users includes

- Support for screen readers and screen magnifiers, as described in "Assistive Technologies" on page 2-87
- Command-line alternatives for most graphical user interface (GUI) options
- Keyboard access to GUI components
- A clear indication of the current cursor focus
- Information available to assistive technologies about user interface elements, including the identity, operation, and state of the element
- Nonreliance on color coding as the sole means of conveying information about working with a GUI
- Noninterference with user-selected contrast and color selections and other individual display attributes, as well as noninterference for other operating system-level accessibility features
- Consistent meaning for bitmapped images used in GUIs
- HTML documentation that is accessible to screen readers

Keyboard access to the user interface includes support for "sticky keys," which allow you to press key combinations (such as **Ctrl+C**) sequentially rather than simultaneously.

Except for scopes and real-time data acquisition, the MathWorks software does not use flashing or blinking text, objects, or other elements having a flash or blink frequency greater than 2 Hz and lower than 55 Hz.

The MathWorks believes that its products do not rely on auditory cues as the sole means of conveying information about working with a GUI. However, if you do encounter any issues in this regard, please report them to the MathWorks Technical Support group.

```
http://www.mathworks.com/contact_TS.html
```

Documentation Accessibility Support

Documentation is available in HTML format for all MathWorks products in this release.

Accessing the Documentation

To access the documentation with a screen reader, go to the documentation area on the MathWorks Web site at

```
http://www.mathworks.com/access/helpdesk/help/helpdesk.html
```

Navigating the Documentation

Note that the first page that opens lists the products. To get the documentation for a specific product, click the link for that product.

The table of contents is in a separate frame. You can use a document's table of contents to navigate through the sections of that document.

Because you will be using a general Web browser, you will not be able to use the search feature included in the MATLAB Help browser. You will have access to an index for the specific document you are using. The cross-product index of the MATLAB Help browser is not available when you are using a general Web browser.

Products

The documentation for all products is in HTML and can be read with a screen reader. However, for most products, most equations and most graphics are not accessible.

The following product documentation has been modified (as described below) to enhance its accessibility for people using a screen reader such as JAWS:

- MATLAB (many sections, but not the function reference pages (however, M-file help is accessible))
- Excel Link
- Optimization Toolbox
- Signal Processing Toolbox
- Statistics Toolbox

Documentation Modifications

Modifications to the documentation include the following:

- Describing illustrations in text (either directly or via links)
- Providing text to describe the content of tables (as necessary)
- Restructuring information in tables to be easily understood when a screen reader is used
- Providing text links in addition to any image mapped links

Equations

Equations that are integrated in paragraphs are generally explained in words. However, most complex equations that are represented as graphics are not currently explained with alternative text.

Assistive Technologies

Note To take advantage of accessibility support features, you must use MathWorks products on a Microsoft Windows platform.

Tested Assistive Technologies

The MathWorks has tested the following assistive technologies:

- JAWS 5.0, 6.0, and 7.0 for Windows (screen reader) from Freedom Scientific
- Built-in accessibility aids from Microsoft, including the Magnifier and "sticky keys"

Use of Other Assistive Technologies

Although The MathWorks has not tested other assistive technologies, such as other screen readers or ZoomText Xtra (screen magnifier) from Ai Squared, The MathWorks believes that most of the accessibility support built into its products should work with most assistive technologies that are generally similar to the ones tested.

If you use other assistive technologies than the ones tested, The MathWorks is very interested in hearing from you about your experiences.

Installation Notes for Accessibility Support

Note If you are not using a screen reader such as JAWS, you can skip this section.

This section describes the installation process for setting up your MATLAB environment to work effectively with JAWS.

Use the regular MATLAB installation script to install the products for which you are licensed. The installation script has been modified to improve its accessibility for all users.

Note Java Access Bridge 2.0 is installed automatically when you install MATLAB.

After you complete the product installation, there are some additional steps you need to perform to ensure JAWS works effectively with MathWorks products.

Setting Up JAWS

Make sure that JAWS is installed on your machine. If it is, there is probably a shortcut to it on the Windows desktop.

Setting up JAWS involves these tasks:

1. Add the Access Bridge to your Windows path (for networked installations only).
2. Create the `accessibility.properties` file.

These tasks are described in more detail below.

(For Networked Installations Only) Add Access Bridge to Your Path.

If you are running MATLAB in a networked installation environment (that is, if the MATLAB Installer was not run on your machine), you need to take the following steps to add Access Bridge to your Windows path.

Note This procedure assumes your Windows **Start** button is set to Classic mode. To set Classic mode, from the **Start** button, select **Settings**. Next select **Task Bar and Menu**. Then select the **Start Menu** tab and make sure the `Classic Start Menu` option is enabled. Click **OK** and you are done.

---

1. From the **Start** button, select **Settings**, next select **Control Panel**. Scroll down and click the **System** icon to display the System Properties dialog box**.**
2. In the **System Properties** dialog box, select the **Advanced** tab.
3. Click **Environment Variables**.
4. Under **System variables**, select the `Path` option.
5. Click the **Edit** button.
6. To the start of the `Path` environment variable, add the directory that contains `matlab.exe`; for example:

   ```
   C:\matlab\bin\win32;
   ```

Be sure to include that semicolon between the end of this directory name and the text that was already there.

7 Click **OK** three times.

8 If JAWS is already running, exit and restart.

Note JAWS must be started with these path changes in effect to work properly with MATLAB.

Create the accessibility.properties File.

1 Create a text file that contains the following two lines:

```
screen_magnifier_present=true
assistive_technologies=com.sun.java.accessibility.AccessBridge
```

2 Use the filename `accessibility.properties`.

3 Move the `accessibility.properties` file into

matlabroot`\sys\java\jre\win32\jre1.5.0_07\lib\`

JAWS Pronunciation Dictionary. As a convenience, The MathWorks provides a pronunciation dictionary for JAWS. This dictionary is in a file called `MATLAB.jdf`.

During installation, the file is copied to your system under the MATLAB root directory at `sys\Jaws\matlab.jdf`.

To use the dictionary, you must copy it to the `\SETTINGS\ENU` folder located beneath the JAWS root installation directory.

You need to restart JAWS and MATLAB for the settings to take effect.

Testing

After you install JAWS and set up your environment as described above, you should test to ensure JAWS is working properly:

1. Start JAWS.
2. Start MATLAB.

JAWS should start talking to you as you select menu items and work with the MATLAB user interface in other ways.

Troubleshooting

This section identifies workarounds for some possible issues you may encounter related to accessibility support in MathWorks products.

JAWS Does Not Detect When the MATLAB Installation Has Started

When you select `setup.exe`, the Windows copying dialog box opens and you are informed. After the files have been copied, the installation splash screen opens, and then the installer starts. However, JAWS does not inform you that the installer has begun: the installer either starts up below other windows or applications or it is minimized. Since the installer is not an active item, nothing is read.

Therefore, check the Windows applications bar for the installer. After you go to the installer, you can use JAWS to perform the installation.

JAWS Stops Speaking

When many desktop components are open, JAWS with MATLAB sometimes stops speaking.

If this happens, close most of the desktop components, exit MATLAB, and restart.

Command Output Not Read

In the MATLAB Command Window, JAWS does not automatically read the results of commands.

To read command output, first select **File** > **Preferences** > **Command Window**, select the option **Use arrow keys for navigation instead of command history recall**, and click **OK**. Then, in the Command Window,

press the arrow keys to move to the command output and use JAWS keystrokes to read the output.

With this preference set, you cannot use arrow keys to recall previous commands. Instead use the following key bindings:

- Windows key bindings:
  - Previous history: **Ctrl+up arrow**
  - Next history: **Ctrl+down arrow**
- Emacs key bindings:
  - Previous history: **Ctrl+p**
  - Next history: **Ctrl+n**

To return to using the up and down arrow keys to recall previous commands, clear the preference.

Some GUI Menus Are Treated as Check Boxes

For some GUIs (for example, the figure window), menus are treated by JAWS as though they are check boxes, whether or not they actually are.

You can choose a menu item for such GUIs by using accelerator keys (e.g., **Ctrl+N** to select **New Figure**), if one is associated with a menu item. You can also use mnemonics for menu navigation (e.g., **Alt+E**).

Note that check boxes that you encounter by tabbing through the elements of a GUI are handled properly.

Text Ignored in Some GUIs

For some dialog boxes, JAWS reads the dialog box title and any buttons, but ignores any text in the dialog box.

Also, in parts of some GUIs, such as some text-entry fields, JAWS ignores the label of the field. However, JAWS will read any text in the text box.

3

Running Functions — Command Window and History

If you have an active Internet connection, you can watch the Working in the Development Environment video demo and the Command History video demo for an overview of the major functionality. The Command Window is where you run (execute) MATLAB statements, while the Command History is a log of the statements you have run.

The Command Window (p. 3-3)	Access the Command Window.
Running Functions and Programs, and Entering Variables (p. 3-7)	Enter statements at the prompt. Run M-files, interrupt programs, run external programs, and examine errors. Evaluate and open selections.
Controlling Input (p. 3-15)	Consider case sensitivity, enter long statements, edit statements, and use syntax highlighting and keyboard shortcuts.
Controlling Output (p. 3-30)	Suppress, page and format output, clear and print contents, and save a session.
Searching in the Command Window (p. 3-34)	Use the Find dialog or incremental search features to find content in the Command Window.

Preferences for the Command Window (p. 3-40)	Specify options for text, display, tab size, accessibility, and indenting for the Command Window and the Editor/Debugger.
Command History Window (p. 3-49)	View session histories. Run statements, copy entries, search, and print the history. Set preferences.
Preferences for Command History (p. 3-59)	Specify how often to automatically save the history file and the types of statements to exclude.

The Command Window

In this section...

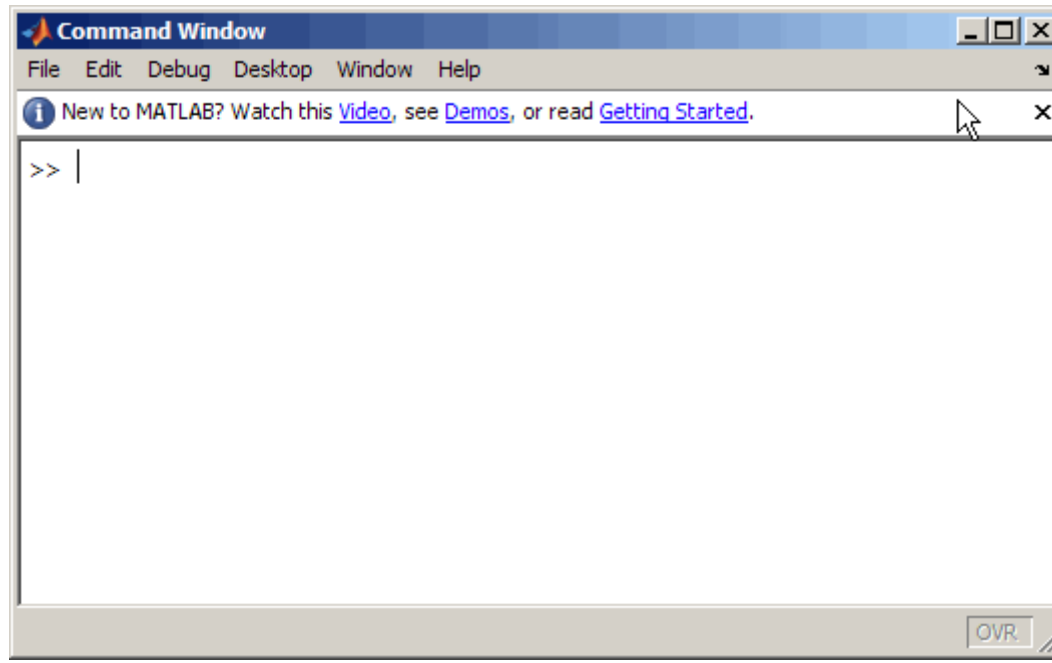
About the Command Window

The Command Window is one of the main tools you use to enter data, run MATLAB functions and other M-files, and display results. If you have an active Internet connection, you can Working in the Development Environment video demo for an overview of the major functionality.

Opening the Command Window

When the Command Window is not open, access it by selecting **Command Window** from the **Desktop** menu. Alternatively, open the Command Window with the `commandwindow` function.

If you prefer a simple command line interface without the other MATLAB desktop tools, select **Desktop > Desktop Layout > Command Window Only**. For more information, see "Arranging the Desktop" on page 2-6.



Command Window Prompt

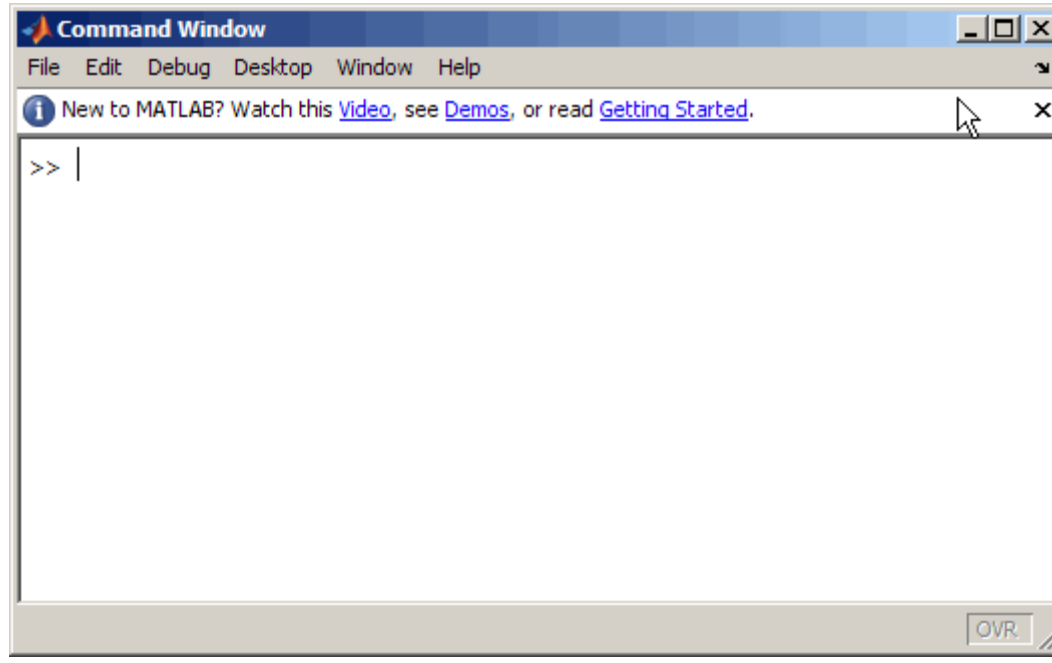
The Command Window prompt, >>, is where you enter statements. For example, you can enter a MATLAB function with arguments, or assign values to variables. The prompt indicates that MATLAB is ready to accept input from you. When you see the prompt, you can enter a variable or run a statement. This prompt is also known as the command line.

When MATLAB displays the `K>>` prompt in the Command Window, MATLAB is in debug mode. Type `dbquit` to return to normal mode. For more information, see Chapter 6, “Editing and Debugging M-Files”

MATLAB displays the `EDU>>` prompt for the MATLAB Student Version.

Getting Started Message Bar in the Command Window

Just below the Command Window menu bar is a message bar that includes links to a video, demos, and information on getting started with MATLAB. If you want to remove the message bar in the Command Window, click the Close box in the right corner of the bar.



If after having closed it, you want to display the information bar again, use “Preferences for the Command Window” on page 3-40.

In addition to the message bar, there are numerous other ways to access the documentation and demos, including using the **Help** menu in most tools. Furthermore, if you place your cursor in a function name, you can right-click and then choose **Help on Selection** from the context menu to see documentation for that function. The reference page for that function opens in a popup window, or if the reference page does not exist, the M-file help appears. For more information help on selection, see “Getting Pop-Up

Help for Functions” on page 4-49—for more general information on help, see Chapter 4, “Help for Using MATLAB”.

Running Functions and Programs, and Entering Variables

In this section...

Running Statements at the Command Line Prompt

Entering Variables and Running Functions

At the prompt, enter data and run functions. For example, to create `A`, a 3-by-3 matrix, type

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

When you press the **Enter** or **Return** key after typing the line, MATLAB responds with

```
A =

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

To run a function, type the function including all arguments and press **Enter** or **Return**. MATLAB displays the result. For example, type

```
magic(2)
```

and MATLAB returns

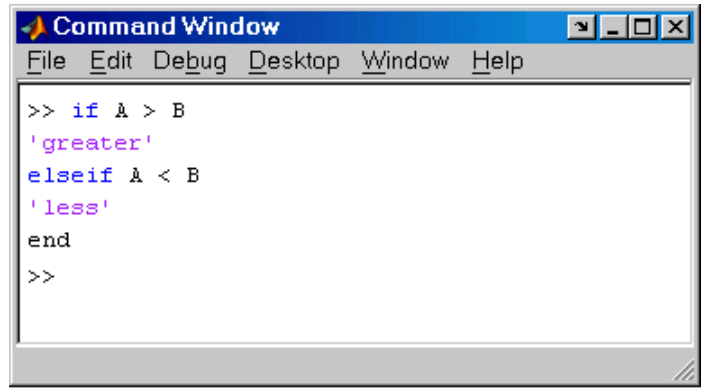
```
ans =
 1     3
 4     2
```

Definition of a Statement. All of the information you type before pressing **Enter** or **Return** is known as a statement. This can include:

- Variable assignments: For example, `a = 3`
- Commands: M-files provided with MATLAB or toolboxes that do not accept input arguments, for example, `clc`, which clears the Command Window.
- Scripts: M-files (MATLAB program files) you write that do not take input arguments or return output arguments, for example, `myfile.m`.
- Functions and their arguments: M-files that can accept input arguments and return output arguments, for example, `magic`.

Some functions support a form that does not require an input argument, thereby operating as commands. For convenience, the term function is used to refer to both functions and commands.

When you enter program control statements, such as `if ... end`, the prompt does not appear until you complete the set of functions. In the following example, you press **Enter** at the end of each line, but the prompt does not appear until you complete the set of statements with `end`.



Running M-Files

Run M-files, files that contain code in the MATLAB language, the same way that you would run any other MATLAB function. Type the name of the M-file in the Command Window and press **Enter** or **Return**. The M-file must be in

the MATLAB current directory or on the MATLAB search path — for details, see “Search Path” on page 5-23. You can also use the `run` function and specify the full pathname to an M-file script.

To determine the name of the M-file currently running, use `mfilename`.

Examining Errors

If an error message appears when you run an M-file, click the underlined portion of the error message, or position the cursor within the filename and press **Ctrl+Enter**. The offending M-file opens in the Editor/Debugger, scrolled to the line containing the error.

Processing Order

In MATLAB, you can only run one process at a time. If MATLAB is busy running one function, any further statements you issue are buffered in a queue. The next statement will run when the previous one finishes.

Interrupting a Running Program

You can stop a running program by pressing **Ctrl+C** or **Ctrl+Break** at any time. On Macintosh platforms, you can also use **Command+.** (the Command key and the period key) to stop the program. For certain operations, stopping the program might generate errors in the Command Window.

For M-files that run a long time, or that call built-ins or MEX-files that run a long time, **Ctrl+C** does not always effectively stop execution. Typically, this happens on Windows rather than UNIX platforms. If you experience this problem, you can help MATLAB break execution by including a `drawnow`, `pause`, or `getframe` function in your M-file, for example, within a large loop. Note that **Ctrl+C** might be less responsive if you started MATLAB with the `-nodesktop` option (an option only for UNIX platforms).

Running External Programs

The exclamation point character, !, sometimes called bang, is a *shell escape* and indicates that the rest of the input line is a command to the operating system. Use it to invoke utilities or call other executable programs without quitting MATLAB. On UNIX, for example,

```
!vi yearlystats.m
```

invokes the vi editor for a file named `yearlystats.m`. After the external program completes or you quit the program, the operating system returns control to MATLAB. Add `&` to the end of the line, such as

```
!dir &
```

on Windows platforms to display the output in a separate window or to run the application in background mode. For example

```
!excel.exe &
```

opens Excel and returns control to the Command Window so you can continue running MATLAB statements.

The maximum length of the argument list provided as input to the bang (!) command is determined by any restrictions maintained within the operating system. If you are running the Windows 2003 Server, for example, the length of the argument list input to the bang command cannot exceed 512 characters.

See the reference pages for the `unix`, `dos`, and `system` functions for details about running external programs that return results and status.

Note To execute operating system commands with specific environment variables, include all commands to the operating system within the `system` call. Separate the commands using `&` (ampersand) for DOS, and `;` (semicolon) for UNIX. This applies to the MATLAB `!` (bang), `dos`, `unix`, and `system` functions. Another approach is to set environment variables before starting MATLAB.

On Macintosh platforms, you cannot run AppleScript directly from MATLAB. However, you can run the Macintosh OS X `osascript` function from the MATLAB `unix` or `!` (bang) function to run AppleScript from MATLAB.

UNIX System Path and Running UNIX Programs from MATLAB

To run a UNIX program from MATLAB if its directory is not on the UNIX system path MATLAB uses, take one of the actions described here.

Change Current Directory in MATLAB. Change the current directory in MATLAB to the directory that contains the program you want to run.

Modify the UNIX System Path MATLAB Uses. Add the directories to the system path from the shell. The exact steps depend on your shell. This is an example using sh:

1. At the system command prompt, type

   ```
   export PATH="$PATH:<mydirectory>"
   ```

   where `<mydirectory>` is the directory that contains the program you want to run.

2. Start MATLAB.

3. In MATLAB, type

   ```
   !echo $PATH
   ```

The directory containing the file is added to the system path that MATLAB uses. This change applies only to the current session of the terminal window.

Automatically Modify System Path at MATLAB Startup. If you want to add a directory to the `PATH` environment variable each time you start MATLAB, perform these steps:

1. In a text editor, open the file `MATLAB/bin/matlab`. This file is used to start MATLAB.

2. Add this line to the beginning of the `matlab` file

   ```
   export PATH="$PATH:<mydirectory>"
   ```

   where `<mydirectory>` is the directory you want to add to the path.

   If you run a tsch shell instead of a bash shell, use `setenv` instead of `export`.

3 Save the file.

The `matlab` file will modify the `PATH` environment variable, and then start MATLAB.

Evaluating or Opening a Selection

Make a selection in the Command Window and press **Enter** or **Return**. The selection is appended to whatever is at the prompt, and MATLAB executes it.

Similarly, you can select a statement from any MATLAB desktop tool, right-click, and select **Evaluate Selection** from the context menu. Alternatively, after making a selection, use the shortcut key, **F9**, or for some tools, press **Enter** or **Return**. For example, you can scroll up in the Command Window, select a statement you entered previously, and then press **Enter** to run it. If you try to evaluate a selection while MATLAB is busy, for example, running an M-file, execution waits until the current operation is done.

You can open a function, file, variable, or Simulink model from the Command Window. Select the name in the Command Window, and then right-click and select **Open Selection** from the context window. This runs the `open` function for the item you selected so that it opens in the appropriate tool:

- M-files and other text files open in the Editor/Debugger.
- Figure files (`.fig`) open in a figure window.
- Variables open in the Array Editor.
- Models open in Simulink.

See the `open` reference page for details about what action occurs if there are name conflicts. If no action exists to work with the selected item, **Open selection** calls `edit`.

Function Alternative

Use `open` or `edit` to open a file in the Editor/Debugger. Use `type` to display the M-file in the Command Window.

Displaying Hyperlinks in the Command Window

You can use MATLAB commands to create hyperlinks in the Command Window. The created hyperlink can:

- Open a Web page in a MATLAB browser using an `href` string.
- Transfer files via the file transfer protocol (FTP).
- Run a MATLAB M-file using the `matlabcolon` (`matlab:`) command.

Hyperlinks to Web Pages

When creating a hyperlink to a Web page, append a full hypertext string on a single line as input to the `disp` or `fprintf` command. For example, the command

```
disp('<a href = "http://www.mathworks.com">The MathWorks Web Site</a>')
```

displays the hyperlink

```
The MathWorks Web Site
```

in the Command Window.

When you click this link, a MATLAB Web browser opens and displays the requested page.

Transferring Files via FTP

To create a link to an FTP site, enter the site address as input to the `disp` command as shown below.

```
disp('<a href = "ftp://ftp.mathworks.com">The MathWorks FTP Site</a>')
```

This command displays

```
The MathWorks FTP Site
```

as a link in the Command Window.

When you click this link, a MATLAB browser opens and displays the requested FTP site.

Running MATLAB Functions by Hyperlink

Use `matlab:` to run a specified statement when you click a hyperlink in the Command Window. For example

```
disp('<a href="matlab:magic(4)">Generate magic square</a>')
```

displays

Generate magic square

When you click the link `Generate magic square`, MATLAB runs `magic(4)`. Alternatively, you can press **Ctrl+Enter** if the cursor is positioned in the link text. You can use the `disp`, `error`, `fprintf`, or `warning` function with this feature. Change the hyperlink color using **Colors Preferences** — see “Colors Preferences for Desktop Tools” on page 2-70. For more information, including examples, see the `matlabcolon (matlab:)` reference page.

Controlling Input

In this section...

Case and Space Sensitivity

Uppercase and Lowercase for Variables

With respect to case, MATLAB requires an exact match for variable names. For example, if you have a variable `a`, you cannot refer to that variable as `A`.

Uppercase and Lowercase for Files and Functions

With respect to functions, filenames, objects, and classes on the search path or in the current directory, MATLAB prefers an exact match with regard to case. MATLAB runs a function if you do not enter the function name using the exact case, but displays a warning the first time you do this.

To avoid ambiguity and warning messages, always match the case exactly. It is a best practice to use lowercase only when running and naming functions. This is especially useful when you use both Windows and UNIX platforms because their file systems behave differently with regard to case.

Note that if you use the `help` function, function names are shown in all uppercase, for example, `PLOT`, solely to distinguish them. Some functions for interfacing to Java do use mixed case and the M-file help and documentation accurately reflect that.

Examples. The directory `first` is at the top of the search path and contains the file `A.m`. If you type `a` instead of `A`, MATLAB runs `A.m` but issues a warning. When you type `a` again during that session, MATLAB runs `A.m` but does not show the warning.

Add the directory `second` after `first` on the search path, with the file `a.m` in `second`. The directory `first` contains `A.m`, while `second` contains `a.m` Type `a`. MATLAB runs `a.m` but displays a warning the first time you do this.

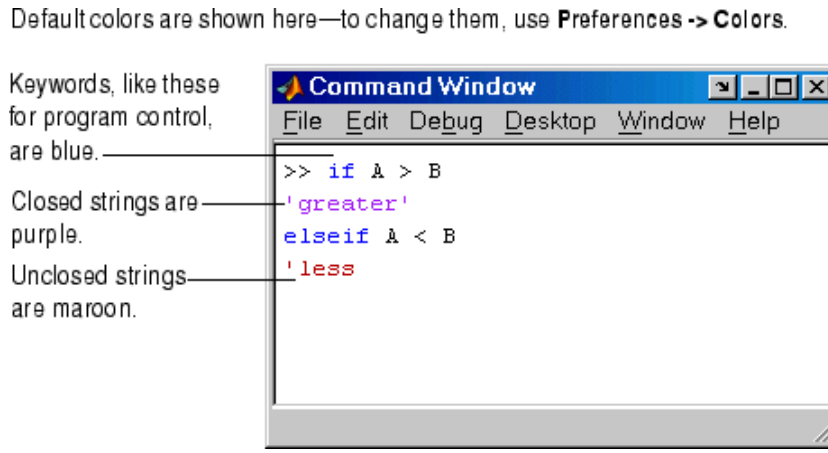
Spaces in Expressions

Blank spaces around operators such as `-`, `:`, and `( )`, are optional, but they can improve readability. For example, MATLAB interprets the following statements the same way.

```
y = sin (3 * pi) / 2
y=sin(3*pi)/2
```

Syntax Highlighting

Some entries appear in different colors to help you better find elements, such as matching `if/else` statements. This is known as syntax highlighting. You can change the colors using preferences. Note that output does not appear with syntax highlighting, except for errors. For more information, see “Colors Preferences for Desktop Tools” on page 2-70.



Matching Delimiters (Parentheses)

You can set a preference for MATLAB to notify you about matched and unmatched delimiters. For example, when you type a parenthesis, bracket, or brace, MATLAB highlights the matched delimiter in the pair. To set these preferences, select **File > Preferences > Keyboard > Delimiter Matching**. This feature is also available in the Editor/Debugger.

For more information, see “Delimiter Matching” on page 3-46.

Cut, Copy, Paste, and Undo Features

Use the **Cut**, **Copy**, **Paste**, **Undo**, and **Redo** features from the **Edit** menu when working in the Command Window. You can also access some of these features in the context menu for the Command Window.

Undo applies to some of the actions listed in **Edit** menu. You can undo multiple times in succession until there are no remaining actions to undo. Select **Edit > Redo** to reverse an undo.

If you use **Enter**, you cannot edit a line after entering it, even though you have not completed the flow. In that event, use **Ctrl+C** to end the flow, and then enter the statements again.

Enter Multiple Lines Without Running Them

To enter multiple lines before running any of them, use **Shift+Enter** or **Shift+Return** after typing a line. This is useful, for example, when entering a set of statements containing keywords, such as `if ... end`. The cursor moves down to the next line, which does not show a prompt, where you can type the next line. Continue for more lines. Then press **Enter** or **Return** to run all of the lines.

This allows you to edit any of the lines you entered before you pressing **Enter** or **Return**.

Entering Multiple Functions in a Line

To enter multiple functions on a single line, separate the functions with a comma (,) or semicolon (;). Using the semicolon instead of the comma will suppress the output for the command preceding it. For example, put three functions on one line to build a table of logarithms by typing

```
format short; x = (1:10)'; logs = [x log10(x)]
```

and then press **Enter** or **Return**. The functions run in left-to-right order.

Entering Long Statements (Line Continuation)

If a statement does not fit on one line, enter three periods (`...`) , also called dots, stops, or an ellipsis, at the end of the line to indicate it continues on the next line. Then press **Enter** or **Return**. Continue typing the statement on the next line. You can repeat the ellipsis to add a line break after each line until you complete the statement. When you finish the statement, press **Enter** or **Return**.

For items in single quotation marks, such as strings, you must complete the string in the line on which it was started. For example, completing a string as shown here

```
headers = ['Author Last Name, Author First Name, ' ...
'Author Middle Initial']
```

results in

```
headers =
Author Last Name, Author First Name, Author Middle Initial
```

MATLAB produces an error when you do not complete the string, as shown here:

```
headers = ['Author Last Name, Author First Name, ...
Author Middle Initial']

??? headers = ['Author Last Name, Author First Name, ...
Error: Missing variable or function.
```

Note that MATLAB ignores anything appearing after the `...` on a line, and continues processing on the next line. This effectively creates a comment out of the text following the `...` on a line. For more information, see "Commenting Out Part of a Statement" on page 6-19.

Recalling Previous Lines

Use the arrow, tab, and control keys on your keyboard to recall, edit, and reuse functions you typed earlier. For example, suppose you mistakenly enter

```
rho = (1+ sqt(5))/2
```

Because you misspelled `sqrt`, MATLAB responds with

```
Undefined function or variable 'sqt'.
```

Instead of retyping the entire line, press the up arrow ↑ key. The previously typed line is redisplayed. Use the left arrow key to move the cursor, add the missing `r`, and press **Enter** or **Return** to run the line. Repeated use of the up arrow key recalls earlier lines, from the current and previous sessions. Using the up arrow key, you can recall any line maintained in the Command History window.

Similarly, specify the first few characters of a line you entered previously and press the up arrow key to recall the previous line. For example, type the letters `plo` and then press the up arrow key. This displays the last line that started with `plo`, as in the most recent `plot` function. Press the up arrow key

again to display the next most recent line that began with `plo`, and so on. Then press **Enter** or **Return** to run the line. This feature is case sensitive.

If the up arrow key moves the cursor up but does not recall previous lines, clear the accessibility preference. For more information, see "Accessibility" on page 3-43.

Another way to view and access commands from the current and previous MATLAB sessions is with the Command History window — see "Command History Window" on page 3-49.

Tab Completion in the Command Window

MATLAB helps you automatically complete the names of these items as you type them in the Command Window:

- Function or model on the search path or in the current directory
- Filename or directory
- Variable, including structures, in the current workspace
- Handle Graphics property for figure in the current workspace

Type the first few characters of the item name and then press the **Tab** key. To use tab completion, you must have the tab completion preference for the Command Window selected. For details, see "Keyboard Preferences" on page 3-43.

Tab completion is also available in the Editor/Debugger, but there are some slight differences in usage. See "Tab Completion in the Editor/Debugger" on page 6-21.

These examples demonstrate how to use tab completion in the Command Window:

• "Tab Completion for Properties" on page 3-25

Basic Example — Unique Completion

This example illustrates a basic use for tab completion. After creating a variable, `costs_march`, type

```
costs
```

and press **Tab**. MATLAB automatically completes the name of the variable, displaying

```
costs_march
```

Then complete the statement, adding any arguments, operators, or options, and press **Return** or **Enter** to run it. In this example, if you just press **Enter**, MATLAB displays the contents of `costs_march`. If MATLAB does not complete the name `costs_march` but instead moves the cursor to the right, you do not have the preference set for tab completion. If MATLAB displays `No Completions Found`, `costs_march` does not exist in the current workspace.

You can use tab completion anywhere in the line, not just at the beginning. For example, if you type

```
a = cost
```

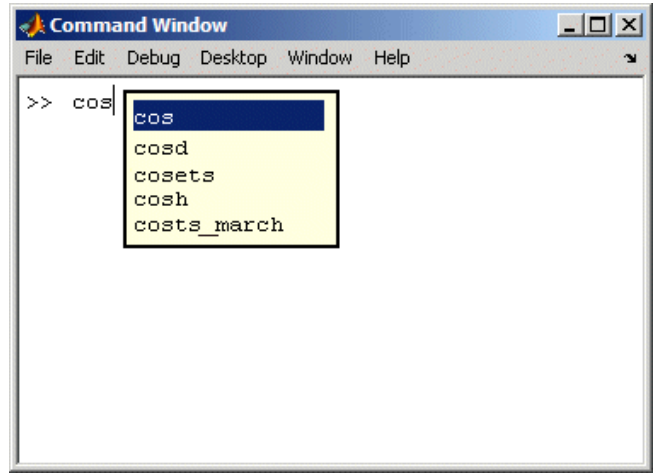
and press **Tab**, MATLAB completes `costs_march`. You can also select `co` or position the cursor after `co` and press **Tab** to complete `costs_march`.

Multiple Possible Completions

If there is more than one name that starts with the characters you typed, when you press the **Tab** key, MATLAB displays a list of all names that start with those characters. For example, type

```
cos
```

and press **Tab**. MATLAB displays

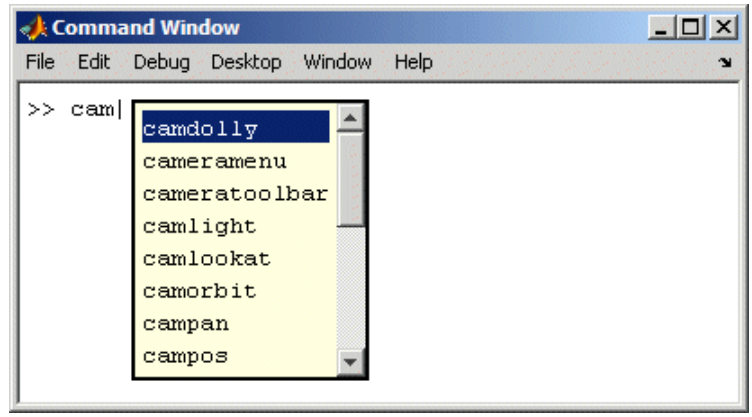


The resulting list of possible completions includes the variable name you created, `costs_march`, but also includes functions that begin with `cos`, including `cosets` from Communications Toolbox, if it is installed on the system and on the MATLAB search path. MATLAB completes variable names in the currently selected workspace, and the names of functions and models on the MATLAB search path or in the current directory.

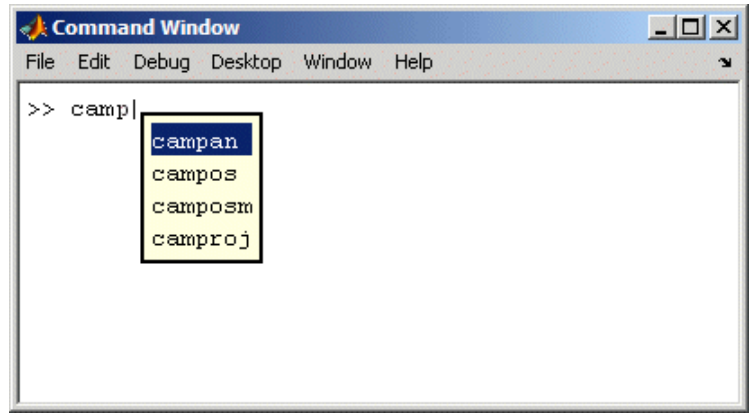
Continue typing to make your entry unique. For example, type the next character, such as `t` in the example. MATLAB selects the first item in the list that matches what you typed, in this case, `costs_march`. Press **Enter** (or **Return**) or **Tab** to select that item, which completes the name at the prompt. In the example, MATLAB displays `costs_march` at the prompt. Add any arguments, and press **Enter** again to run the statement.

You can navigate the list of possible completions using up and down arrow keys, and **Page Up** and **Page Down** keys. You can clear the list without selecting anything by pressing **Escape**. Note that the list of possible completions might include items that are not valid commands, such as private functions.

Narrowing Completions Shown. You can narrow the list of completions shown by typing a character and then pressing **Tab** if the Command Window preference **Tab key narrows completions** is selected. This is particularly useful for large lists. For example, type `cam` and press **Tab** to see the possible completions. There is a scroll bar with the list because there are too many completions to be seen at once.



Type `p` and press **Tab** again. MATLAB narrows the list, showing only all possible `camp` completions.



Continue narrowing the list in the same way. For the above example, type `o` and press **Tab** to further narrow the list. Press **Enter** or **Return** to select an item, which completes the name at the prompt.

Tab Completion for Directories and Filenames

Tab completion works for directories and filenames in MATLAB functions. For example, type

```
edit d:/
```

and press **Tab**.

MATLAB displays the list of directories and files in `d`, from which you can choose one. For example, type

```
mym
```

and press **Tab**.

MATLAB displays

```
edit d:/mymfiles/
```

where `mymfiles` is the only directory on your `d` drive whose name begins with `mym`. Continue using tab completion to display and complete directory names or filenames until you finish the `edit` statement.

Tab completion for directories and filenames is not supported for functions you write.

Tab Completion for Structures

For structures in the current workspace, after the period separator, press **Tab**. For example, type

```
mystruct.
```

and press **Tab** to display all fields of `mystruct`. If you type a structure and include the start of a unique field after the period, pressing **Tab** completes that structure and field entry.

For example, type

```
mystruct.n
```

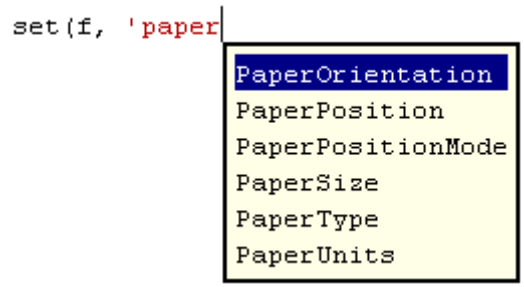
and press **Tab**, which completes the entry `mystruct.name`, where `mystruct` contains no other fields that begin with `n`.

Tab Completion for Properties

Complete property names for figures in the current workspace using tab completion, as in this graphics example. Here, `f` is a figure. Type

```
set(f, 'pap
```

and press **Tab**. MATLAB displays



Select a property from the list. For example, type

```
u
```

and press **Enter**. MATLAB completes the property, including the closing quote.

```
set(f, 'paperunits'
```

Continue adding to the statement, as in this example

```
set(f, 'paperunits', 'c
```

and press **Tab**. MATLAB automatically completes the property

```
set(f, 'paperUnits', 'centimeters'
```

because `centimeters` is the only possible completion.

Keyboard Shortcuts in the Command Window

Following is the list of arrow and control keys that serve as shortcuts for using the Command Window. In addition to these shortcut keys (sometimes called hot keys), you can use shortcuts for menu items, which you can view on the menus, as well as general desktop shortcuts described in "Keyboard Shortcuts" on page 2-40. If you select the Emacs (MATLAB standard) preference for key bindings (see "Command Window Key Bindings" on page 3-44 for an explanation), you can also use the **Ctrl+key** combinations shown in the table.

Key or Mouse Action for Windows Preference	Control Key for MATLAB standard (Emacs) Preference	Key or Mouse Action for Macintosh Preference	Operation
↑	**Ctrl+P**	↑	Recall *p*revious line — for details, see "Recalling Previous Lines" on page 3-19. See also "Command History Window" on page 3-49, which is a log of previously used functions, and "Keeping a Session Log" on page 3-33. With the **Accessibility** preference selected, moves the cursor up a line when it is above the prompt. In that event, use **Ctrl+**↑ to recall previous lines for Windows and Macintosh key bindings.

Key or Mouse Action for Windows Preference	Control Key for MATLAB standard (Emacs) Preference	Key or Mouse Action for Macintosh Preference	Operation
↓	**Ctrl+N**	↓	Recall *n*ext line — for details, see "Recalling Previous Lines" on page 3-19. Works only after using the up arrow or **Ctrl**+**P**. With the **Accessibility** preference selected, moves the cursor down a line when it is above the prompt. In that event, use **Ctrl**+↓ to recall previous lines for Windows and Macintosh key bindings.
Ctrl+Home	None	**Home**	Move to top of Command Window.
Ctrl+End	None	**End**	Move to end of Command Window.
None	None	**Cmd+Home**	Move cursor and scroll to top of Command Window.
None	None	**Cmd+End**	Move cursor and scroll to end of Command Window.
None	None	**Shift+Cmd+Home**	Select to top of Command Window.
None	None	**Shift+Cmd+End**	Select to end of Command Window.
←	**Ctrl+B**	←	Move *b*ack one character.
→	**Ctrl+F**	→	Move *f*orward one character.
Ctrl+←	None	**Option+**←	Move *l*eft one word.
Ctrl+→	None	**Option+**→	Move *r*ight one word.
Home	**Ctrl+A**	**Cmd+**←	Move to beginning of current statement. With Macintosh key bindings, move to beginning of current line.
End	**Ctrl+E**	**Cmd+**→	Move to *e*nd of current statement. With Macintosh key bindings, move to end of current line.

Key or Mouse Action for Windows Preference	Control Key for MATLAB standard (Emacs) Preference	Key or Mouse Action for Macintosh Preference	Operation
Esc	**Ctrl+U**	**Esc**	Clear the command line when cursor is at the command line. Otherwise, move cursor to command line.
Delete	**Ctrl+D**	Forward **Delete**	Delete character after cursor.
Backspace	**Ctrl+H**	**Delete**	Delete character before cursor.
None	**Ctrl+K**	None	Cut contents (*k*ill) from cursor to end of current line.
Insert	None	None	Change to overwrite mode from insert mode, or change to insert mode from overwrite mode. View current mode in the status bar: OVR is gray for insert mode. In overwrite mode, what you type replaces existing text and the cursor is a wide block. (Not supported on Macintosh platforms.)
Double-click	None	Double-click	Select current word. To select additional words, hold mouse after second click and continue dragging left or right.
None	None	**Shift+Option+←**	Select to previous word.
None	None	**Shift+Option+→**	Select to next word.
Triple-click	None	None	Select current line. To select additional lines, hold mouse after second click and continue dragging up or down.
Shift+Home	None	**Shift+Cmd+←**	Select from cursor to beginning of statement. With Macintosh key bindings, select to beginning of line.

Key or Mouse Action for Windows Preference	Control Key for MATLAB standard (Emacs) Preference	Key or Mouse Action for Macintosh Preference	Operation
Shift+End	None	**Shift+Cmd+→**	Select from cursor to end of statement. With Macintosh key bindings, select to end of line.
Enter in selection	None	None	Append selection to statement at command line and execute it.
Ctrl+Enter in hyperlink	None	**Ctrl+Enter** in hyperlink	Open hyperlink displayed in Command Window. For example, in the hyperlink of an error message, opens the file in the Editor/Debugger at that line number.

Navigating Above the Command Line

To look at or copy information in the Command Window that is above the command line (>> prompt), use the mouse and scroll bar, key combinations such as **Ctrl+Home**, and search features. By default, the up and down arrow keys recall statements so you cannot use them to move the cursor when it is above the command line.

To use the up and down arrow keys to move the cursor when it is above the command line, select **File > Preferences > Command Window**, and select the **Accessibility** preference.

Controlling Output

In this section...

Echoing Execution

To display each function within a statement as it executes, run `echo on`. For details, see the `echo` reference page.

Suppressing Output

If you end a statement with a semicolon (`;`) and then press **Enter** or **Return**, MATLAB runs the statement but does not display any output. This is particularly useful when you generate large matrices. For example, running

```
A = magic(100);
```

creates `A` but does not show the resulting matrix in the Command Window.

Paging of Output in the Command Window

If output in the Command Window is lengthy, it might not fit within the screen and display too quickly for you to see it without scrolling back to it. To avoid that problem, use the `more` function to control the paging of output in the Command Window. By default, `more` is `off`.

After you type `more on`, MATLAB displays only a page (a screen full) of output, pauses, and displays

```
--more--
```

indicating there is more output to display. Press one of the following keys.

Key	Action
Enter or **Return**	To advance to the next line
Space Bar	To advance to the next page
q	To stop displaying the output

You can scroll in the Command Window to see input and output that are no longer in view. As an alternative to scrolling, you can use the up and down arrow keys if the Command Window Accessibility preference is selected.

Formatting and Spacing Numeric Output

By default, numeric output in the Command Window is displayed as 5-digit scaled, fixed-point values, called the short format. To change the numeric format of output for the current and future sessions, set the Command Window preference for text display. The text display format affects only how numbers are shown, not how MATLAB computes or saves them.

Function Alternative

Use the `format` function to control the output format of the numeric values displayed in the Command Window. The format you specify applies until you change it or until the end of the session. More advanced alternatives are listed in the “See Also” section of the `format` reference page.

Examples of Formats

Here are a few examples of the various formats and the output produced from the following two-element vector `x`.

```
x = [4/3 1.2345e-6]

format short
   1.3333    0.0000

format short e
```

```
   1.3333e+000  1.2345e-006

format +
++
```

A complete list and description of available formats is in the reference page for `format`. For more control over the output format, use the `sprintf` and `fprintf` functions.

Controlling Spacing

To control spacing in the output, use the Command Window preference for text display or the `format` function. Use

```
format compact
```

to suppress blank lines, allowing you to view more information in the Command Window. To include the blank lines, which can help make output more readable, use

```
format loose
```

Clearing the Command Window

Select **Clear Command Window** from the **Edit** menu or context menu to clear it. This does not clear the workspace, but only clears the view. Afterwards, you still can use the up arrow key to recall previous functions. A confirmation dialog box appears if you select the preference for it; see preferences for “Confirmation Dialogs Preferences” on page 2-81 for more information.

Function Alternative

Use `clc` to clear the Command Window. Similar to `clc` is the `home` function, which moves the prompt to provide a clear screen, but does not clear the text so you can still scroll up to see it.

Printing Command Window Contents

To print the complete contents of the Command Window, select **File > Print**. To print only a selection, first make the selection in the Command Window and then select **File > Print Selection**.

Specify printing options for the Command Window by selecting **File > Page Setup**. For example, you can print with a header. For more information, see "Printing and Page Setup Options for Desktop Tools" on page 2-52.

Keeping a Session Log

The diary Function

The `diary` function creates a copy of your MATLAB session in a disk file, including keyboard input and system responses, but excluding graphics. You can view and edit the resulting text file using any text editor, such as the Editor/Debugger. To create a file on your disk called `sept23.out` that contains all the functions you enter, as well as MATLAB output, enter

```
diary('sept23.out')
```

To stop recording the session, use

```
diary('off')
```

To view the file, run

```
edit('sept23.out')
```

Other Session Logs

There are two other means of viewing session information:

- The Command History window, which contains a log of all functions executed in the current and previous sessions.
- The `logfile` startup option — see "Startup Options" on page 1-12.

Searching in the Command Window

In this section...

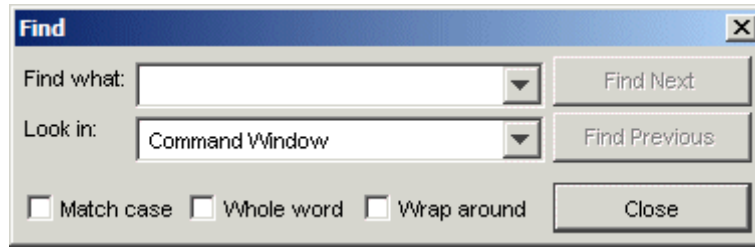
Introduction

You can search for specified text that appears in the Command Window, where the text was either part of input you supplied, or output displayed by MATLAB. After finding the desired text, you can copy and paste it to the prompt in the Command Window to run it, or into an M-file or other file.

See also "Recalling Previous Lines" on page 3-19, "Tab Completion in the Command Window" on page 3-20, and "Keyboard Shortcuts in the Command Window" on page 3-26 for techniques to reuse previous statements and navigate in the Command Window. To find files and text in files, see "Finding Files and Content Within Files" on page 5-49.

Find Dialog Box

Select **Find** from the **Edit** menu to search for specified text in the Command Window using the Find dialog box. Complete the dialog box. The search begins at the current cursor position. MATLAB finds the text you specified and highlights it. Click **Find Next** or **Find Previous** to find another occurrence, or use the keyboard shortcuts **F3** and **Shift+F3**.



MATLAB beeps when a search for **Find Next** reaches the end of the Command Window, or when a search for **Find Previous** reaches the top of the Command Window. If you have **Wrap around** selected, it continues searching after beeping.

Note that you can only search for text currently displayed in the Command Window. To increase the amount of information maintained in the Command Window, increase the setting for the command session scroll buffer size in **Command Window Preferences**, and do not clear the Command Window.

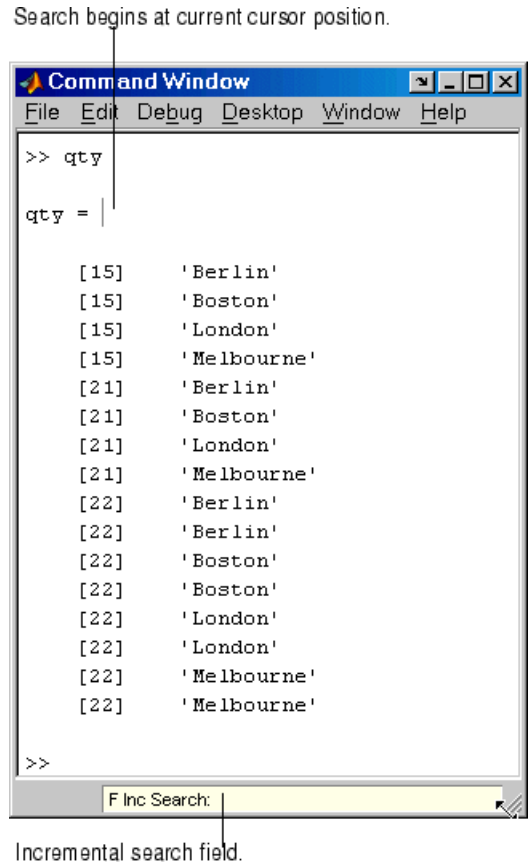
Change the selection in the **Look in** field to search for the specified text in other MATLAB desktop tools.

Incremental Search

With the incremental search feature, the cursor moves to the next or previous occurrence of the specified text in the Command Window. It is similar to the Emacs search feature. To use the incremental search feature in the Command Window,

1. Position the cursor where you want the search to begin.
2. How you begin the incremental search depends on your setting for the Command Window key bindings preference:
   - Press **Ctrl**+**S** for **Emacs**, or
   - Press **Ctrl**+**Shift**+**S** for **Windows**
3. To look for the previous occurrence, press **Ctrl**+**R** or **Ctrl**+**Shift**+**R** instead.

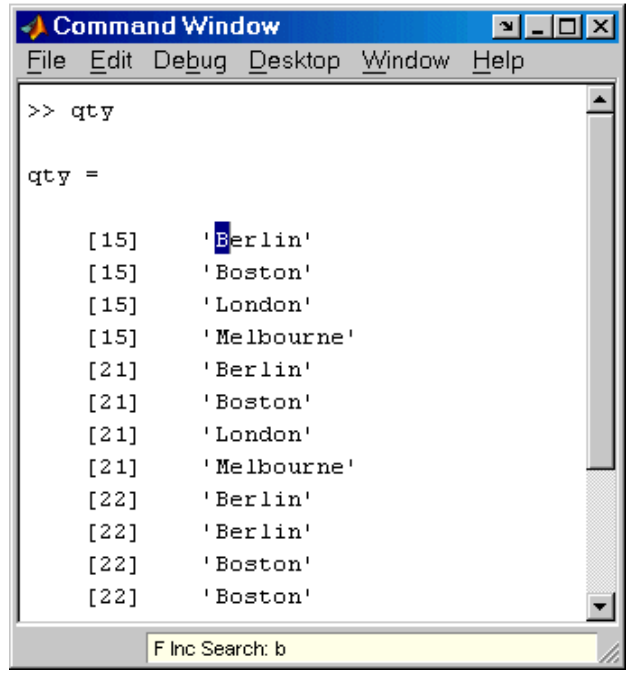
   An incremental search field, **Inc Search**, appears at the bottom of the Command Window and is preceded by **F** for a forward search, or **R** when you are looking for the previous occurrence (reverse search).



4 In the **Inc Search** field, type the text you want to find. For example, look for `Boston`.

As you type the first letter, `b`, the first occurrence of that letter in the Command Window after the current cursor position is highlighted. For the example shown, the first occurrence of `b` is highlighted, the `b` in `Berlin`. Note that incremental search allows for case sensitivity — see “Case Sensitivity in Incremental Search” on page 3-38.

MATLAB finds the next b.



When you type the next letter, the first occurrence of the text becomes highlighted. In the example, when you add the letter `o` to the `b` so that the **Inc Search** field now has `bo`, the `bo` in `Boston` becomes highlighted.

- If you mistype in the **Inc Search** field, use the **Back Space** key to remove the last letters and make corrections.
- After finding the `bo`, you can press **Ctrl+W** to complete that word. In this example, `Boston` appears in the **Inc Search** field.

5 To find the next occurrence of `Boston` in the Command Window, press **Ctrl+S**. To find the previous occurrence of the text, press **Ctrl+R**

6 If MATLAB beeps, it means either that the text was not found, or the search wrapped past the end (or beginning) of the Command Window and continued at the beginning (or end).

- When the text is not found, **Failing** appears in the incremental search field. Modify the search term in the incremental search field and try again. Use **Ctrl+G** to automatically remove characters back to the last successful search. For example, if `plode` fails, **Ctrl+G** removes the `de` from the search term because `plo` does exist in the Command Window.

7 To end the incremental search, press **Esc** or **Enter**, or any other key that is not a character or number.

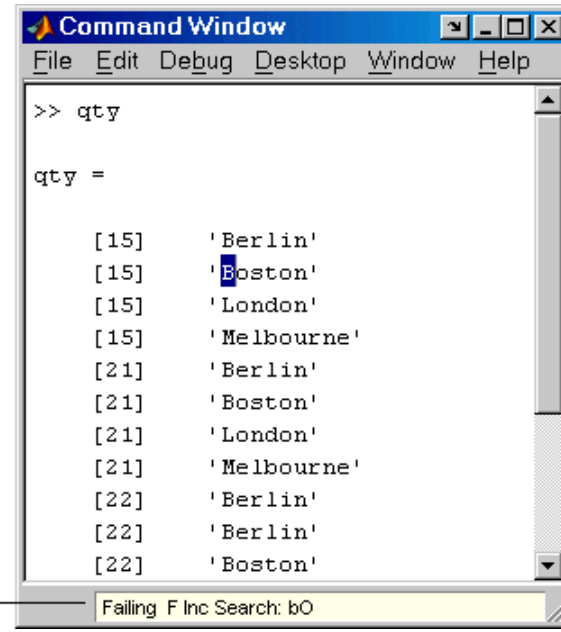
The **Inc Search** field no longer appears. The cursor is at the position where the text was last found, with the search text highlighted.

Incremental search is also available in the Editor/Debugger — see "Incremental Search" on page 6-51.

Case Sensitivity in Incremental Search

When you enter lowercase letters in the **Inc Search** field, for example, `b`, incremental search looks for both lowercase and uppercase instances of the letters, for example `b` and `B`. However, if you enter uppercase letters, for example, `B`, incremental search only looks for instances that match the case you entered.

In the example, enter `b0` in the **Inc Search** field and incremental search does not find any matching text.



Preferences for the Command Window

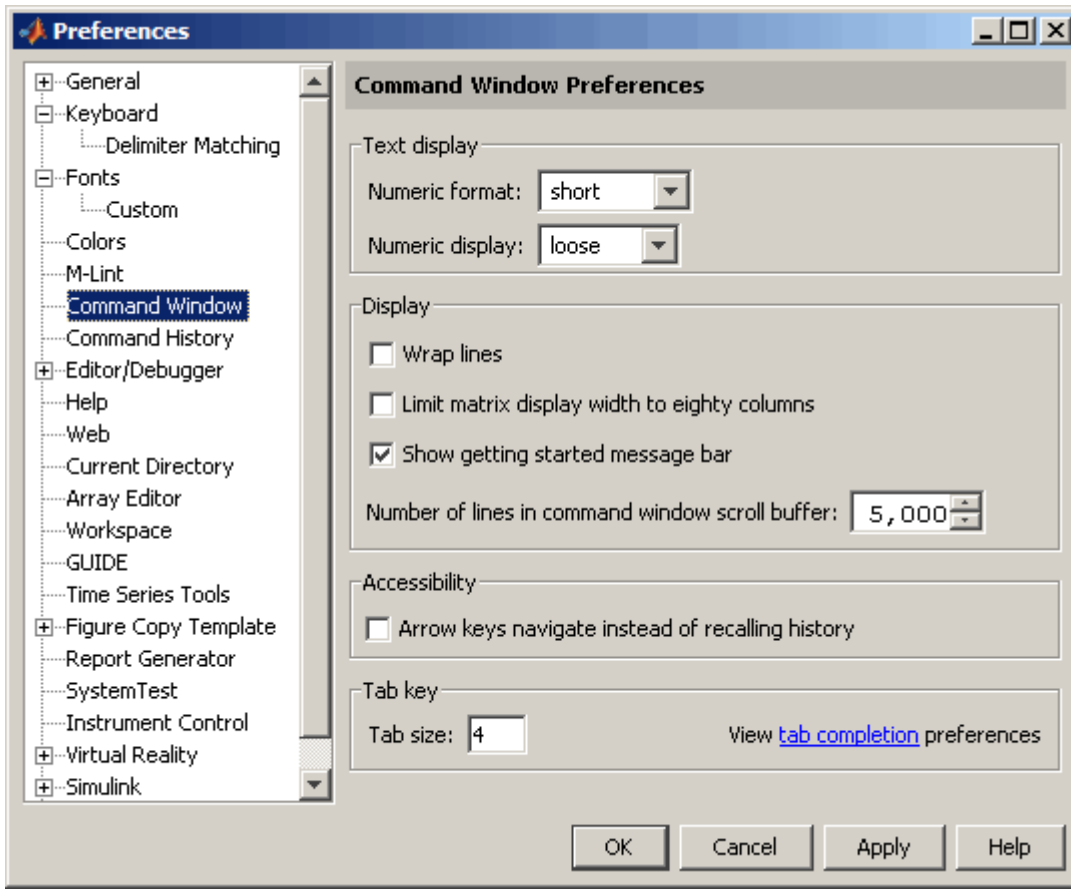
In this section...

See also:

- "Fonts Preferences for Desktop Tools" on page 2-62
- "Confirmation Dialogs Preferences" on page 2-81

Text, Display, Accessibility, and Tab Size Preferences

To set these preferences for the Command Window, select **File > Preferences** and then select **Command Window** in the left pane of the Preferences dialog box.



Text Display

Specify the format, that is, how output appears in the Command Window.

Numeric format. Specify the output format of numeric values displayed in the Command Window. This affects only how numbers are displayed, not how MATLAB computes or saves them. The `format` reference page includes the list of available formats, with examples.

Numeric display. Specify spacing of output in the Command Window. To suppress blank lines, use `compact`. To display blank lines, use `loose`. For more information, see the reference page for `format`.

Display

Wrap lines. Select to make a single line of input or output in the Command Window break into multiple lines in order to fit within the current width of the Command Window. This is useful for console mode. With this option selected, an entire line is visible without scrolling, and the horizontal scroll bar does not appear because it is not needed.

Show getting started message bar. Select to display the getting started message bar in the Command Window. It appears beneath the menu bar and contains links to a video, documentation, and demos. For more information, see “Getting Started Message Bar in the Command Window” on page 3-5

Limit matrix display width to eighty columns. When selected, MATLAB displays only 80 columns of matrix output, regardless of the width of the Command Window. Clear the check box if you make the Command Window wider than 80 columns and want matrix output to fill the width of the Command Window. See also the `display` reference page.

To determine the number of columns and rows that will display in the Command Window, given its current size, use

```
get(0,'CommandWindowSize')
```

With the matrix display width preference cleared, the number of columns is based on the width of the Command Window. With the preference set to 80 columns, the number of columns is always 80.

Number of lines in command window scroll buffer. Set the number of lines maintained in the Command Window, from `1,000` to `25,000`. This is the number of lines you can see when you scroll vertically. A larger buffer means you can view more lines and it provides a larger base for search features, but requires more memory.

This preference setting does not impact the number of lines you can recall when you use the up arrow key in the Command Window. Using the up arrow key, you can recall all lines shown in the Command History window, regardless of how many lines you can see in the Command Window.

Accessibility

Select this option to use the up and down arrow keys to move the cursor when it is above the command line. With this preference selected, use the **Ctrl+** up arrow or down arrow key to recall statements for Windows and Macintosh key bindings, or **Ctrl+P** and **Ctrl+N** for **MATLAB standard (Emacs)** key bindings.

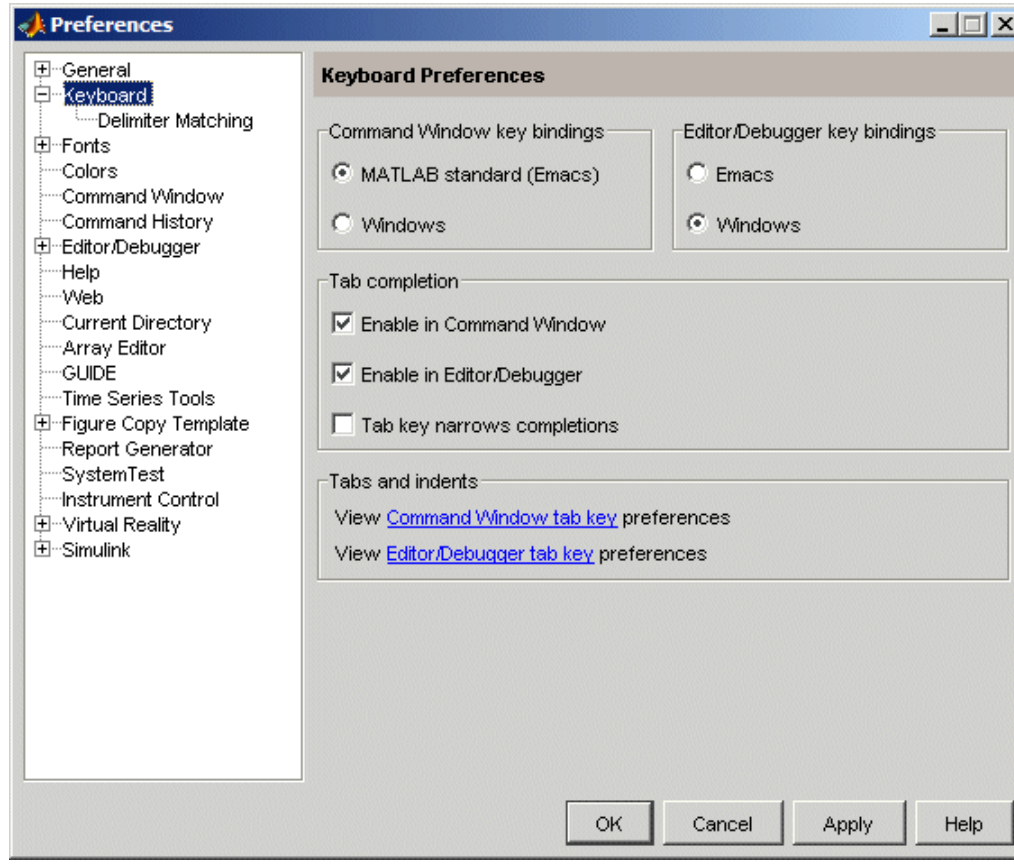
Clear this preference to use the up and down arrow keys to recall statements. Use the mouse and other features to move the cursor when above the command line.

Tab key

Tab size. Number of spaces assigned to a tab stop when displaying output. The default is four spaces, except on UNIX platforms where the default is eight spaces. This does not apply when the tab completion preference is selected.

Keyboard Preferences

To set key binding, tab completion, and delimiter matching preferences for the Command Window and the Editor/Debugger, select **File > Preferences** and then select **Keyboard** in the left pane of the Preferences dialog box.



Command Window Key Bindings

Specify the keyboard shortcuts (key bindings) to be used at the command line.

MATLAB standard (Emacs). Allows you to use the control keys listed in "Keyboard Shortcuts in the Command Window" on page 3-26, which should be familiar to existing MATLAB users and Emacs users. For example, **Ctrl+A** moves the cursor to the beginning of the line.

Windows. Allows you to use standard Windows control keys. For example, **Ctrl+A** is the shortcut for **Edit > Select All**, which selects the entire contents of the Command Window.

Macintosh. This option is available only on Macintosh platforms. It allows you to use Macintosh keys, such as the **Command** key instead of the **Ctrl** key.

Editor/Debugger Key Bindings

Specify the keyboard shortcuts (key bindings) to be used by the Editor/Debugger. The Editor/Debugger key bindings are also used by other tools, for example, the **Callback** field in the Shortcut Editor dialog box.

Select **Windows**, **Emacs**, or **Macintosh** (available only on Macintosh platforms), depending on which convention you want the Editor/Debugger to follow for accelerators and shortcuts. The accelerators on the menus change after you change this option.

For example, when you select Windows key bindings, the shortcut to paste a selection is **Ctrl+V**. When you select Emacs key bindings, the shortcut to paste a selection is **Ctrl+Y**. When you select Macintosh key bindings, the shortcut to paste a selection is **Command+V**. You can see the accelerator on the **Edit** menu for the **Paste** item.

Tab Completion

Enable in Command Window. Select the check box to use tab completion when typing functions in the Command Window. Clear the check box if you do not want to use the tab completion feature. In that event, when you press the **Tab** key, MATLAB moves the cursor to the next tab stop rather than completing a function — see also the preference for "Tab size" on page 3-43.

Enable in Editor/Debugger. Select the check box to use tab completion when typing functions in the Editor/Debugger. Clear the check box if you do not want to use the tab completion feature. In that event, when you press the **Tab** key, MATLAB moves the cursor to the next tab stop rather than completing a function — see also in the online documentation.

Tab key narrows completions. Select this check box to narrow the list of possible completions shown by typing another character and pressing **Tab**. For details, see “Narrowing Completions Shown” on page 3-23.

Tabs and Indents

The links go to the panes where you can view and set preferences for

- **Tab** key size in the Command Window, which is used when the tab completion preference is not set
- **Tab** key size and indenting preferences in the Editor/Debugger

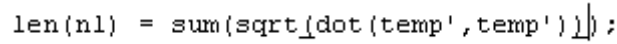
Delimiter Matching

To set these preferences, select **File > Preferences > Keyboard > Delimiter Matching** . These preferences apply to the Command Window and the Editor/Debugger.

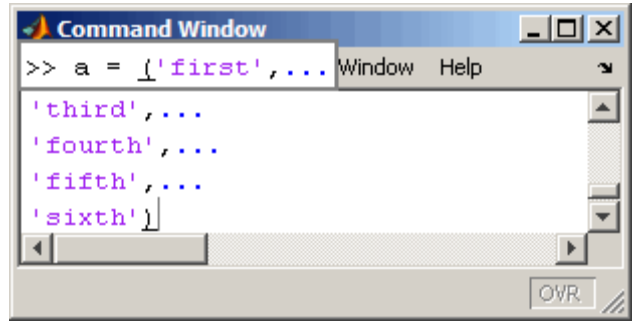
With these preferences selected, MATLAB alerts you to matched and unmatched delimiters based on the MATLAB language syntax rules. For example, when you type a parenthesis or another delimiter, MATLAB highlights the matched parenthesis or delimiter in the pair.

Delimiter pairs are parentheses ( ), brackets [ ], and braces { }. For the Editor/Debugger , paired language keywords are also matched. Paired language keywords include `for`, `if`, `while`, `else`, and `end` statements.

In the following illustration, MATLAB underlines the left parenthesis in the pair when you move over the right parenthesis using an arrow key.



If the matching delimiter is not visible on the screen, a pop-up window appears and shows the line containing the matching delimiter. In the Editor/Debugger, the line number is included. Click in the pop-up window to go to that line.



Match while typing. Select the check box if you want to be alerted to matches and mismatches in pairs of delimiters as you type them. Then choose how you want MATLAB to alert you to matches by selecting an entry from **Show match with**. When you type a closing (or opening) delimiter in the Command Window or Editor/Debugger, MATLAB alerts you based on the option you choose:

- `Balance` — The corresponding delimiter is highlighted briefly.
- `Underline` — Both delimiters in the pair are underlined briefly.
- `Highlight` — Both delimiters in the pair are highlighted briefly.

Choose how you want MATLAB to alert you to mismatches using **Show mismatch with**. When you type a closing delimiter that does not have an opening match, MATLAB alerts you based on the option you choose:

- `Beep` — MATLAB beeps.
- `Strikethrough` — The delimiter you typed is briefly crossed out.
- `None` — There is no action.

Match on arrow key. Select the check box if you want to be alerted to matches and mismatches in pairs of delimiters when you use an arrow key to move the cursor over a delimiter. Then choose how you want MATLAB to alert you to matches by selecting an entry from **Show match with**. When you move the arrow over a closing (or opening) delimiter in the Command Window or Editor/Debugger, MATLAB alerts you based on the option you choose:

- `Underline` — Both delimiters in the pair are underlined briefly.
- `Highlight` — Both delimiters in the pair are highlighted briefly.

Choose how you want MATLAB to alert you to mismatches by selecting an entry from **Show mismatch with**. When you move an arrow key over a delimiter that does not have a match, MATLAB alerts you based on the option you choose:

- `Beep` — MATLAB beeps.
- `Strikethrough` — The delimiter is briefly crossed out.
- `None` — There is no alert.

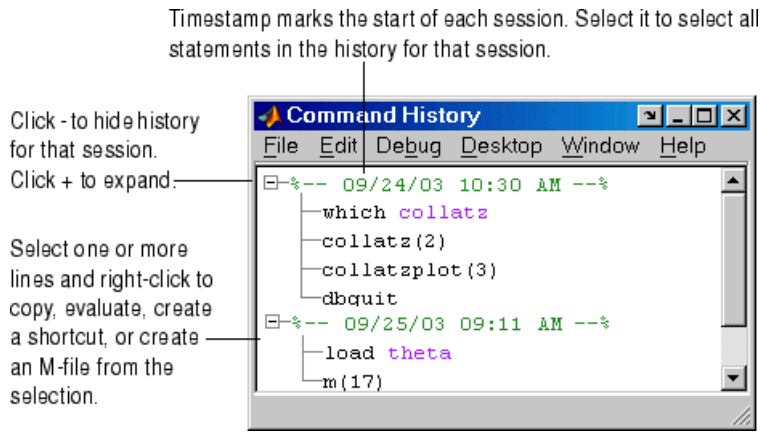
Command History Window

In this section...

Overview

The Command History window displays a log of the statements most recently run in the Command Window. If you have an active Internet connection, you can watch the Command History video demo for an overview of the major functionality.

To show or hide the Command History window, use the **Desktop** menu. Alternatively, use `commandhistory` to open the MATLAB Command History window when it is closed, or to select it when it is open. For details, see "Arranging the Desktop" on page 2-6.



MATLAB provides other options for viewing a history of statements. See also the following sections:

- "Recalling Previous Lines" on page 3-19, which describes using the up arrow in the Command Window
- The `diary` function reference page
- "Startup Options" on page 1-12, which includes the `logfile` startup option

Viewing Statements in the Command History Window

The Command History window lists statements you ran in the current session and in previous sessions. The time and date for each session appear at the top of the history of statements for that session. Use the scroll bar or the up and down arrow keys to move through the Command History window.

Click **-** to hide the history for a session, and click **+** to show it. Select a timestamp to select all entries for that session. With a timestamp selected, you can press the **+** or **-** key on the numeric keypad to show and hide entries.

The Command History window and the Command Window's statement recall feature save to the file `history.m`, which is stored in the same directory as MATLAB preferences. Type `prefdir` in the Command Window to see the location of the file. The `history.m` file is loaded when MATLAB starts, and it stores a maximum of 20,000 bytes, deleting the oldest entries as needed to maintain that size.

MATLAB automatically saves the `history.m` file throughout the session according to the **Saving** preference you specified. You can choose to automatically exclude certain statements from being written to the `history.m` file with the **Settings** preference. For details, see "Preferences for Command History" on page 3-59.

Using Statements from the Command History Window

You can select entries in the Command History window and then perform the following actions for the selected entries.

Action	How to Perform the Action
Run statements in the Command Window	Double-click an entry (entries) in the Command History window to execute the statement(s) in the entries. For example, double-click `edit myfile` to open `myfile.m` in the Editor/Debugger. You can also run the statements in an entry by right-clicking the entry and selecting **Evaluate Selection** from the context menu, or by selecting an entry and pressing **Enter** or **Return**.
Edit and run statements in the Command Window	Select an entry or entries and then select **Copy** from the context menu. Paste the selection into the Command Window. Alternatively, drag the selection to the Command Window. Then in the Command Window, edit the statements, and press **Enter** or **Return** to execute them.
Copy statements to another window	Select an entry or entries and then select **Copy** from the context menu. Paste the selection into an open M-file in the Editor/Debugger or any application. Alternatively, drag the selection from the Command History window to an open M-file or another application.
Create an M-file from statement(s)	Select an entry or entries and then right-click and select **Create M-File** from the context menu. The Editor/Debugger opens a new M-file that contains the statements you selected from the Command History window.
Create a shortcut from statement(s)	Select an entry or entries and then right-click and select **Create Shortcut** from the context menu. Alternatively, drag the selection to the **Shortcuts** toolbar. The Shortcut Editor opens and the selected statements appear in the **Callback** field. For more information, see “Shortcuts for MATLAB — Easily Run a Group of Statements” on page 2-32.

Searching in the Command History Window

There are two types of search in the Command History window:

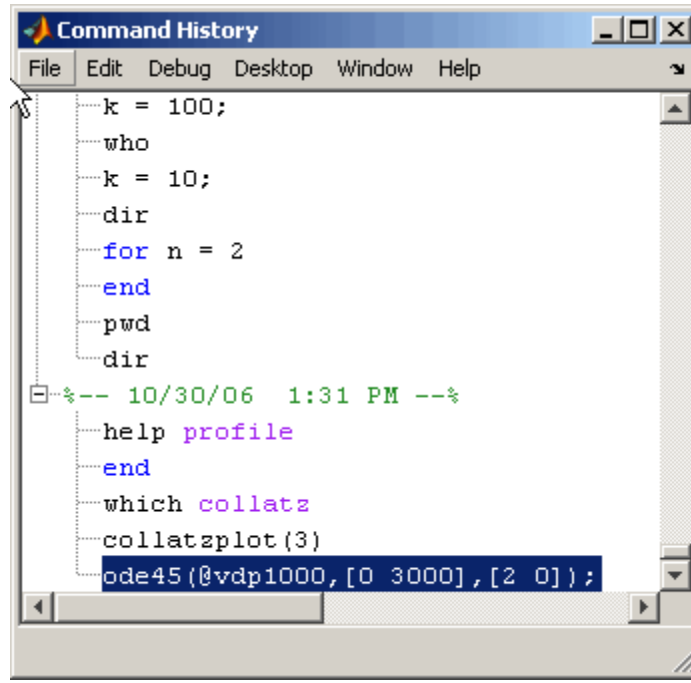
- “Finding Next Entry By Letter” on page 3-52
- “Finding Text” on page 3-56

After finding an entry, you can copy and paste it into an M-file or any file, or you can right-click and select **Evaluate Selection** to run the entry.

Finding Next Entry By Letter

Type a letter in the Command History window. The Command History window searches backwards to find the last previous entry that begins with that letter as illustrated in this example:

1 Position the cursor at anywhere in the Command History window.

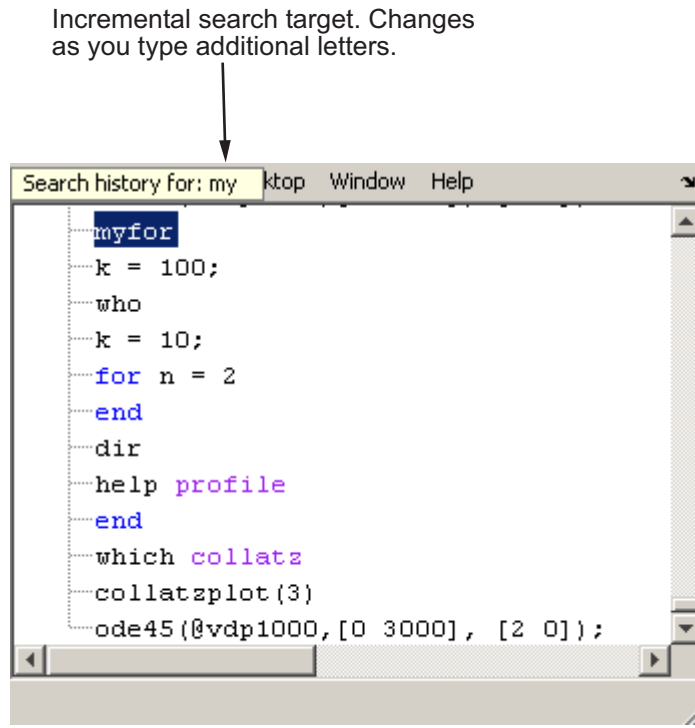


2 Type the first letters of the entry you want to find. For example, type `my`.

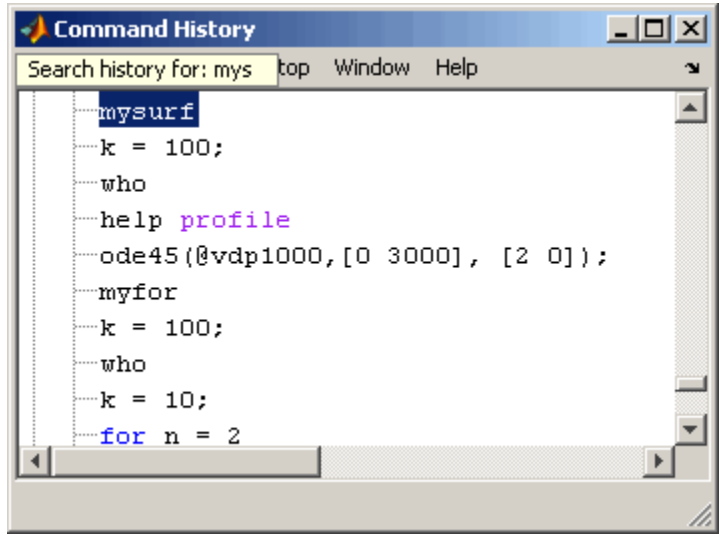
The Command History window searches backwards and selects the previous entry that begins with the letters you typed; in this example, you typed `my`, and the Command History finds `myfor`.

As you begin typing that a small yellow-background pop-up window, `Search history for:`, appears at the top of the Command History window. This window keeps track of your search target as you type additional letters to narrow the focus of your search.

If the search finds a matching entry in a sessions that is collapsed, it expands the session and selects the entry.

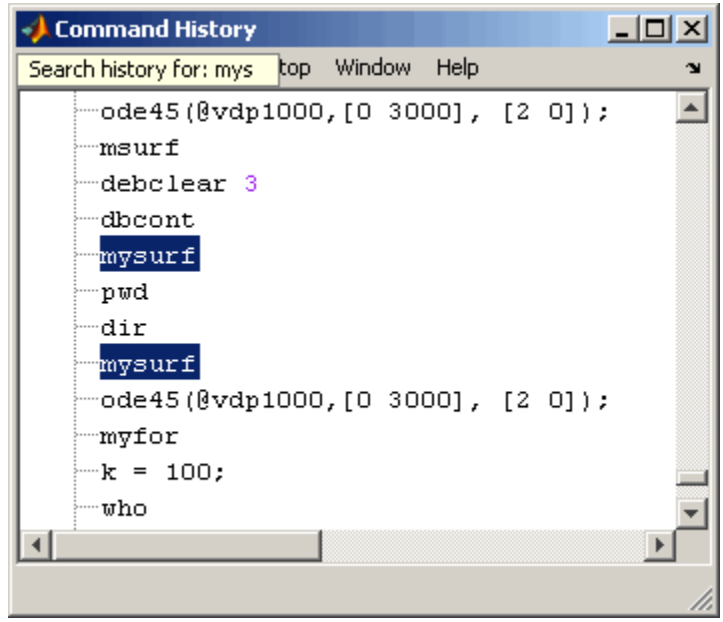


3 Now type an `s` to extend the search to `mys`. The Command History window continues to search backwards, stopping next at the function `mysurf`.

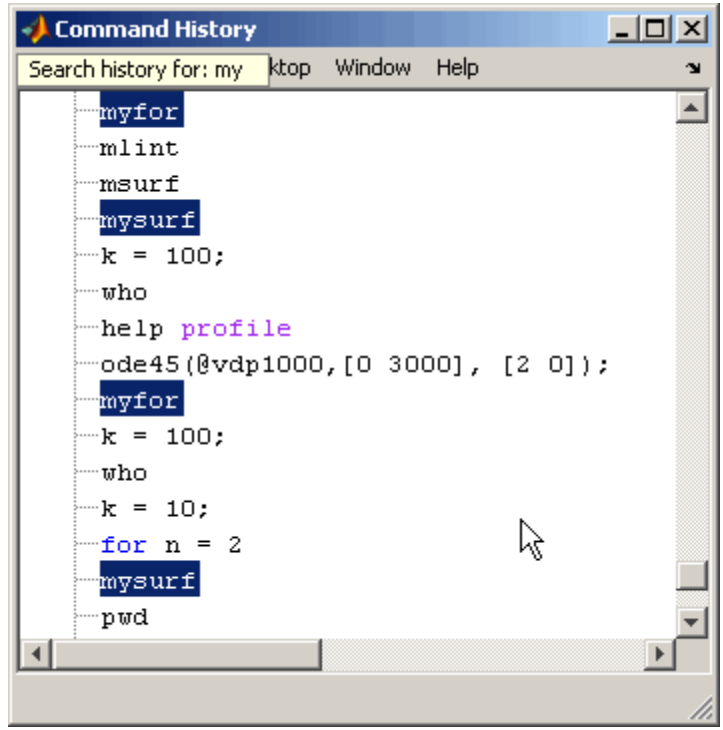


Finding Multiple Occurrences of the Entry. You can use the up and down arrow keys to search for the next or the previous occurrence of the entry you just found.

When you press **Ctrl** and the up or down arrow key, each occurrence of the entry remains highlighted while you search for additional instances.



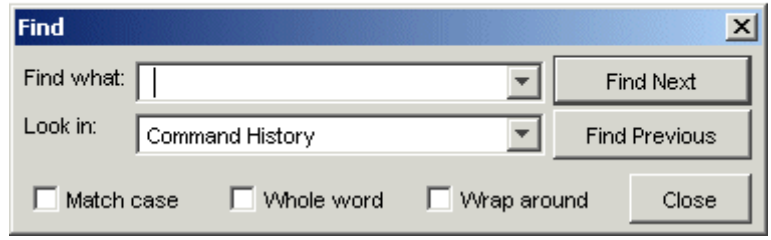
To highlight all instances of the entry, press **Ctrl+A**. In the example below, all instances of entries beginning with `my` are highlighted.



Finding Text

Select **Find** from the **Edit** menu to search for specified text using the Find dialog box. Complete the dialog box. The search begins at the current cursor position. MATLAB finds the text you specified and highlights it. Click **Find Next** or **Find Previous** to find another occurrence, or use the keyboard shortcuts **F3** and **Shift+F3**. Find looks for visible entries only, that is, it does not find entries in collapsed nodes.

Search for specified text in the Command History window.



MATLAB beeps when a search for **Find Next** reaches the end of the Command History window, or when a search for **Find Previous** reaches the top of the Command History window. If you have **Wrap around** selected, it continues searching after beeping.

Change the selection in the **Look in** field to search for the specified text in other MATLAB desktop tools.

Printing the Command History Window

To print the contents of the Command History window, select **File > Print** or **Print Selection**. Specify options for printing by selecting **File > Page Setup**. For example, you can print the history with a header. For more information, see “Printing and Page Setup Options for Desktop Tools” on page 2-52.

The printed version is sized to fit the page. If there is a long statement in the Command History, the reduced page size might be difficult to read. As a workaround, either use **Print Selection**, where the long statement is not part of the selection, or remove any extremely long statements from the Command History before printing it.

Deleting Entries from the Command History Window

Delete entries from the Command History window when you feel there are too many and it becomes inconvenient to find the ones you want. All entries remain until you delete them or until the `history.m` file exceeds its maximum size, when MATLAB automatically deletes the oldest entries—see “Viewing Statements in the Command History Window” on page 3-50.

To delete entries in the Command History window, first select the entries to delete, using one of these methods:

- Select a single entry.
- **Shift**+click or **Ctrl**+click to select multiple entries.
- Select the timestamp for a session to select all entries for that session. Then use **Shift**+click or **Ctrl**+click to select multiple timestamps with all of their entries.

Then right-click and select **Delete selection** from the context menu, or press the **Delete** key. A confirmation dialog box might appear; see preferences for "Confirmation Dialogs Preferences" on page 2-81 for more information.

To delete all entries, select **Edit > Clear Command History**, or select **Clear Entire History** from the context menu.

After deleting entries from the Command History window, you will not be able to recall those statements in the Command Window as described in "Recalling Previous Lines" on page 3-19.

Preferences for Command History

In this section...

Introduction

Using Command History preferences, you can choose to exclude statements from the `history.m` file and specify how often to save it. The `history.m` file is used for both the Command History window and statement recall in the Command Window.

To set preferences for the `history.m` file, select **File > Preferences**, and then select **Command History** in the Preferences dialog box.

Settings

Specify the types of statements to exclude from the `history.m` file. Note that when you exclude statements from the `history.m` file, you cannot recall them in the Command Window as described in "Recalling Previous Lines" on page 3-19, nor can you view them in the Command History window.

Save Exit/Quit Commands

Select the check box to save `exit` and `quit` commands in the `history.m` file.

Save Consecutive Duplicate Commands

Select the check box if you want consecutive executions of the same statement to be saved to the `history.m` file.

For example, with this option selected, run `magic(5)`, and then run `magic(5)` again. The `history.m` file saves two consecutive entries for `magic(5)`. With this option cleared, for the same example, the command history file saves only

one entry for `magic(5)`. If you then run `magic(10)`, the command history file saves both entries, `magic(5)` followed by `magic(10)`.

Saving

Use **Saving** preferences to specify how often to automatically save the `history.m` file during a MATLAB session.

Save History File On Quit

Select this option to save the `history.m` file when you end the MATLAB session. If the session does not end via a normal termination, that is, via the `exit` or `quit` functions, **File > Exit MATLAB**, or the MATLAB desktop Close box, the history file is not saved for that session.

Save After n Commands

Select this option to save the `history.m` file after `n` statements are added to the file. For example, when you select the option and set `n` to `10`, after every 10 statements are added, the history file is automatically saved. Use this option if you don't want to risk losing entries to the saved history because of an abnormal termination, such as a power failure.

Don't Save History File

Select this option if you do not want to save the `history.m` file. This feature is useful when multiple users share the same machine and do not want other users to view the statements they have run.

Note that any entries already in the `history.m` file remain. Prior to setting this preference, you might want to remove any existing entries. Follow the instructions in "Deleting Entries from the Command History Window" on page 3-57.

See Also

Additional preferences that relate to the Command History are

- "Fonts Preferences for Desktop Tools" on page 2-62
- "Confirmation Dialogs Preferences" on page 2-81

4

Help for Using MATLAB

The primary means for getting help is the Help browser, which provides documentation for all your installed products. Other forms of help are available including M-file help and Technical Support solutions. If you have an active Internet connection, you can watch the Help and Documentation video demo for an overview of the major functionality.

Help Functions (p. 4-44)	Use functions to get information, such as `help` and `doc`.
Getting Pop-Up Help for Functions (p. 4-49)	Get pop-up help (help on selection) for functions from within the Editor/Debugger and the Command Window.
Other Forms of Help (p. 4-51)	Use product-specific help features, download M-files, contact Technical Support, see documentation for other MathWorks products, view a list of other books, and participate in a MATLAB newsgroup.

Help Browser Overview

In this section...

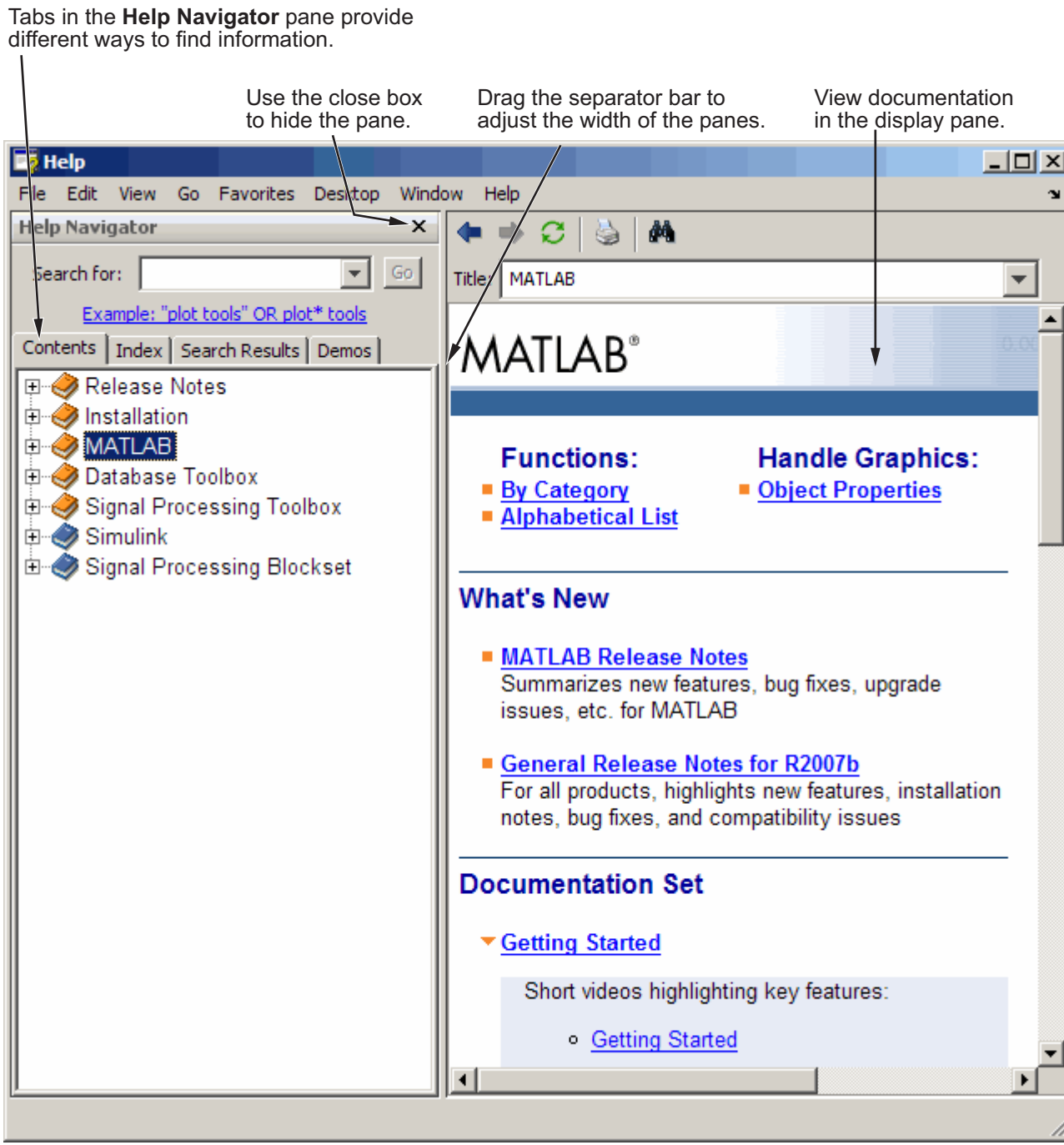
About the Help Browser

The Help browser is an HTML browser integrated with the MATLAB desktop. Use the Help browser to search and view documentation and demonstrations for MATLAB and all other installed MathWorks products. MATLAB automatically installs the documentation and demos for a product when you install that product.

Opening the Help Browser

To open the Help browser, click the Help button  in the desktop toolbar, type `helpbrowser` in the Command Window, or use the **Help** menu in any tool. There are two panes:

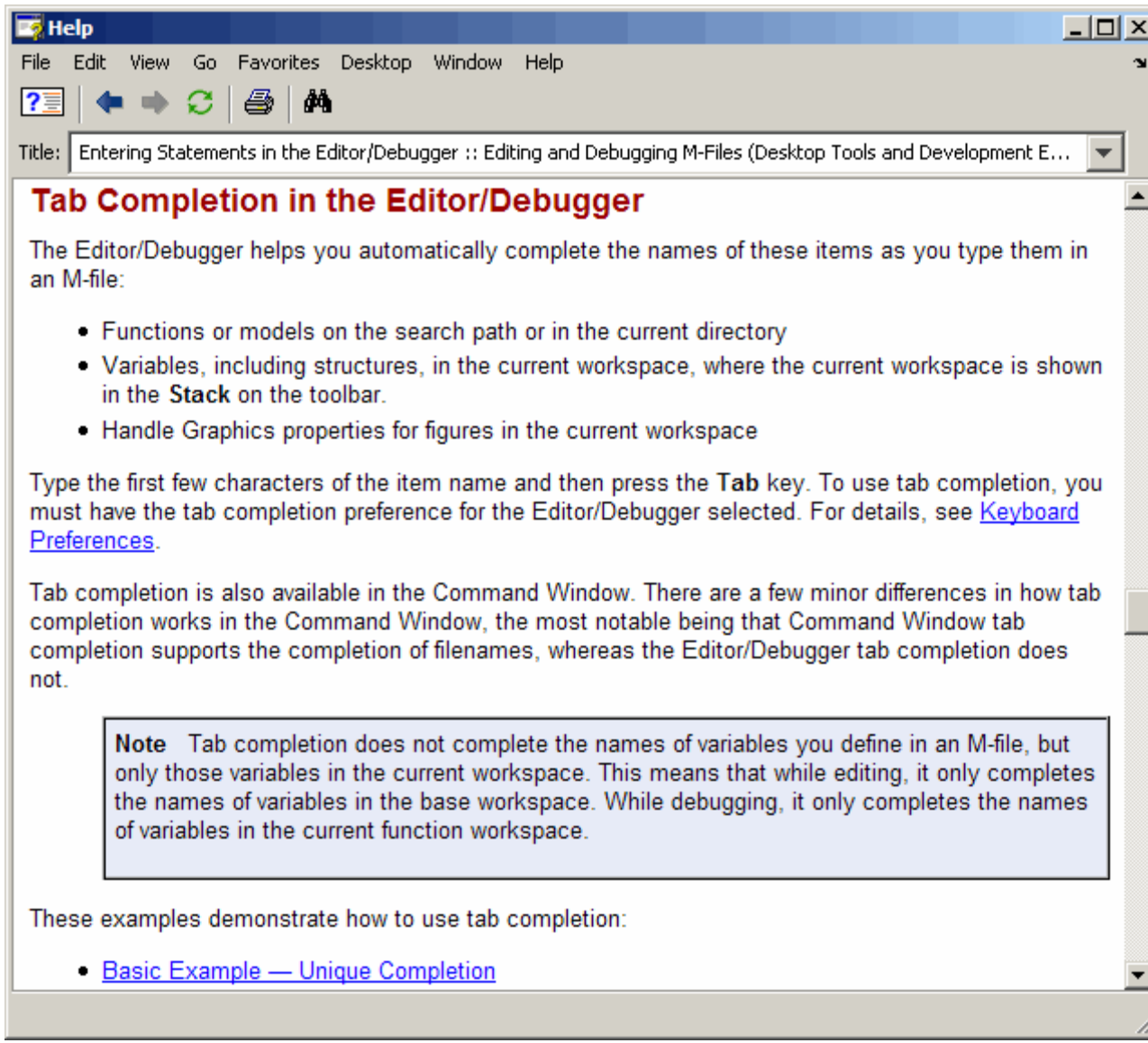
- The Help Navigator, on the left, for finding information, includes a **Search for** field, and **Contents**, **Index**, **Search Results**, and **Demos** tabs. For more information, see “Finding Information with the Help Browser” on page 4-10.
- The display pane, on the right, for viewing documentation and demos.


Tabs in the **Help Navigator** pane provide different ways to find information.
Use the close box to hide the pane.
Drag the separator bar to adjust the width of the panes.
View documentation in the display pane.
Help
File Edit View Go Favorites Desktop Window Help
Help Navigator
Search for:
Go
Example: "plot tools" OR plot* tools
Contents
Index
Search Results
Demos
Release Notes
Installation
MATLAB
Database Toolbox
Signal Processing Toolbox
Simulink
Signal Processing Blockset
Title: MATLAB
MATLAB®
Functions:
By Category
Alphabetical List
Handle Graphics:
Object Properties
What's New
MATLAB Release Notes
Summarizes new features, bug fixes, upgrade issues, etc. for MATLAB
General Release Notes for R2007b
For all products, highlights new features, installation notes, bug fixes, and compatibility issues
Documentation Set
Getting Started
Short videos highlighting key features:
Getting Started

Resizing the Help Browser

To adjust the relative width of the two panes, drag the separator bar between them. You can also change the font in either of the panes — see “Help Fonts and Colors Preferences” on page 4-39.

Once you find the documentation you want, you can close the **Help Navigator** pane so there is more screen space to view the information itself. This is shown in the following figure. To close the **Help Navigator** pane, click the Close box in the pane’s upper right corner. To open the **Help Navigator** pane from the display pane, click the Help Navigator button on the toolbar. Alternatively, use the **View** menu.


Help
File Edit View Go Favorites Desktop Window Help
Title: Entering Statements in the Editor/Debugger :: Editing and Debugging M-Files (Desktop Tools and Development E...
Tab Completion in the Editor/Debugger
The Editor/Debugger helps you automatically complete the names of these items as you type them in an M-file:
Functions or models on the search path or in the current directory
Variables, including structures, in the current workspace, where the current workspace is shown in the Stack on the toolbar.
Handle Graphics properties for figures in the current workspace
Type the first few characters of the item name and then press the Tab key. To use tab completion, you must have the tab completion preference for the Editor/Debugger selected. For details, see Keyboard Preferences.
Tab completion is also available in the Command Window. There are a few minor differences in how tab completion works in the Command Window, the most notable being that Command Window tab completion supports the completion of filenames, whereas the Editor/Debugger tab completion does not.
Note Tab completion does not complete the names of variables you define in an M-file, but only those variables in the current workspace. This means that while editing, it only completes the names of variables in the base workspace. While debugging, it only completes the names of variables in the current function workspace.
These examples demonstrate how to use tab completion:
Basic Example — Unique Completion

Types of Documentation

The Help browser and help functions provide access to the following types of information for all installed MathWorks products. The icons shown here appear in the Help browser contents listing to help you quickly identify documentation by type.

Icon	Type of Documentation	Description and When to Use
	Getting Started	Review Getting Started documentation before you begin using a product or feature for the first time. Then, to learn more, go to the user guides, reference pages, demos, and examples.
 or  or 	Product	MATLAB, toolboxes, and related products use orange book icons  . Simulink, blocksets, and related products use blue book icons  . Link and Target products use green book icons  .
	Index of Examples	Accessible via the Help browser **Contents** listing, this is an index of the major examples included in the Help browser documentation.
	User Guides (blue)	User guide material contains overviews as well as detailed instructions. Consult it after reviewing Getting Started material.
	Reference Pages (orange)	Each function has a reference page that provides the syntax, description, examples, and other information for that function. Each reference page includes links to related functions and additional information. Reference pages are also provided for blocks and properties.
	Release Notes	An overview of new products and features in a release. Release Notes also include upgrade information, links to fixed and known problems, and compatibility considerations. Review the Release Notes for all your products when you first start using a new release. Release Notes for the current version include the release notes for multiple prior versions.

Icon	Type of Documentation	Description and When to Use
	Printable Documentation	Most products provide access to the online documentation in a printable format, PDF. Access PDF files via the Help browser and print them from your PDF reader, such as Adobe Acrobat. Most PDF files reside only on the MathWorks Web site, so you need an Internet connection to view them.
none	Demos	MathWorks products come with demonstrations that run key features of the product. Many of the demos run MATLAB code. Use the Help browser **Demos** pane or **Search Results** to access demos for the products you have installed.
none	M-File Help	Get M-file help in the Command Window to quickly access basic information for a function or model. It provides a brief description of a function and its syntax. It is called M-file help because the text of the help is a series of comments at the top of the M-file for a function.

Accessing Documentation on the Web

You can access all product documentation on the MathWorks Web site at `http://www.mathworks.com/access/helpdesk/help/helpdesk.shtml`. These are some uses for the Web site version of the documentation:

- Access documentation for products you have not installed.
- Access documentation for the most current version. If you do not see the information you are looking for in the Help browser and know you are not running the most current version of MATLAB, the most current version of the documentation, which is on the Web site, might include more information. Note, though, that the documentation on the Web site might refer to features that are not part of your earlier-version product.
- Access documentation for a prior version of some products (Release 13 with Service Pack 2). Note that the release notes on this page include release notes for multiple prior versions. For example, you can find information about MATLAB Version 6.0 (Release 12) new features and changes.

- Access documentation via your system Web browser, such as when you are not running MATLAB or if you prefer your system Web browser.

To determine the URL for a page in the Help browser, see “View the Page Location” on page 4-30

PDF documentation is available only on the Web site.

You cannot read MATLAB documentation files from the MATLAB installation media. You also cannot use a Web browser to read the documentation files installed with MATLAB because the files are compressed JAR files.

Adding Help Files

You can add your own HTML help files so that they appear in the Help browser. For details, see in the online documentation.

Documentation in Other Languages

The MathWorks documentation is available in English. Japanese versions of MATLAB include documentation that has been translated into Japanese. For more information, go to `http://www.cybernet.co.jp/matlab`.

Finding Information with the Help Browser

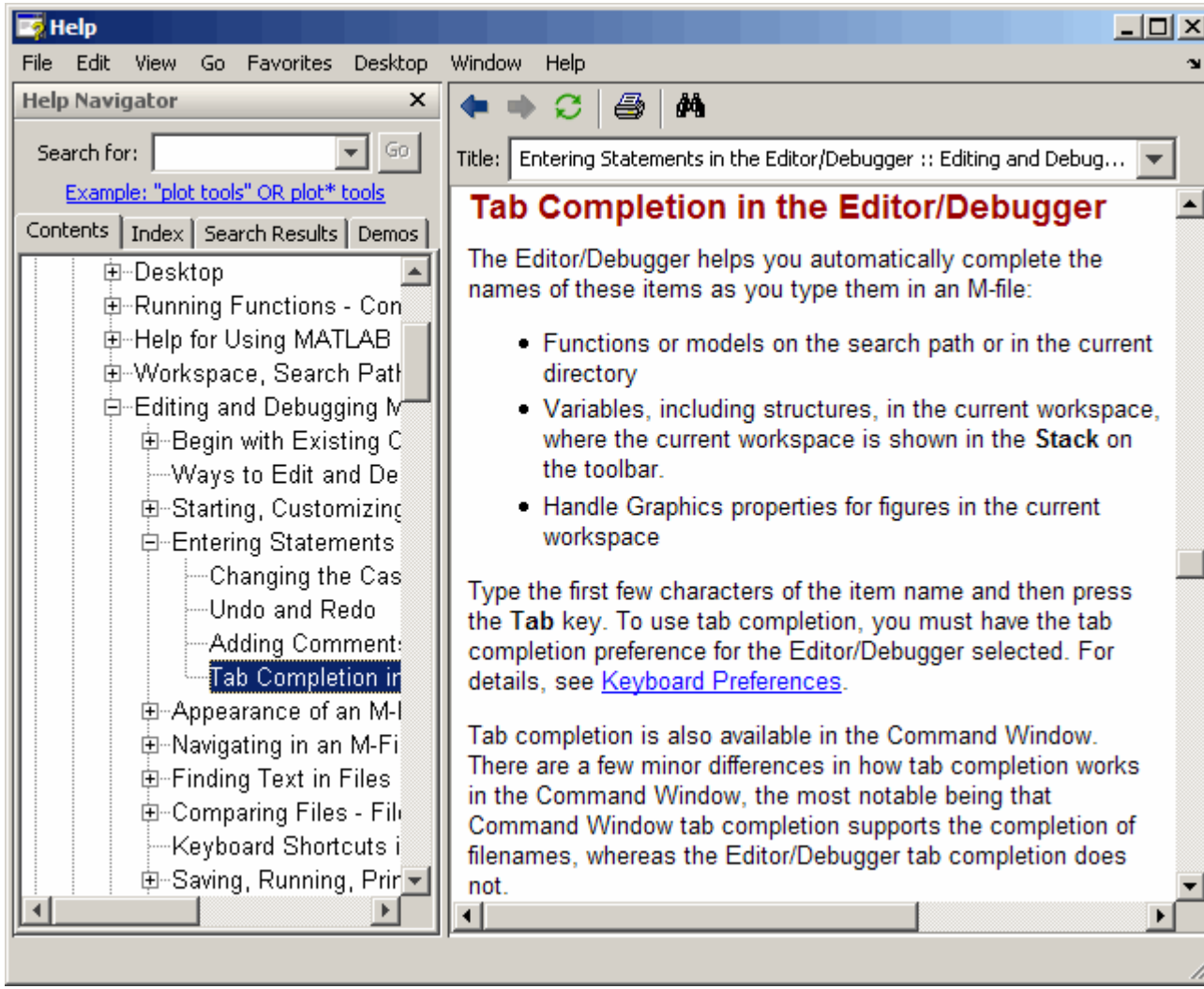
In this section...

Help Navigator

The **Help Navigator** is in the left pane of the Help browser. It provides a table of contents, an index, and a search feature to help you find information.

Contents in the Help Browser

To list the documentation titles and tables of contents for products you installed, click the **Contents** tab in the **Help Navigator** pane. To show documentation for only some of the installed products, use the product filter.



Product Roadmap

When you select a product in the **Contents** pane (any entry with a book icon), such as MATLAB or Communications Toolbox, a *roadmap* of the documentation for that product appears in the display pane. The roadmap includes links to commonly used documentation sections, including

- Function and block references pages
- An index of major examples in the documentation
- The PDF version of the documentation, which is suitable for printing (this is the only direct access from MATLAB to the printable documentation)

Navigate the Contents Listing

In the **Contents** listing, you can

- Click the **+** to the left of an item to show the first page of that document or section in the display pane and expand the listing for that item in the **Help Navigator** pane. You can alternatively: double-click the item, press the right arrow key, or press **+** on the numeric keypad.
- Click the **-** to the left of an item to collapse the listings for that item. You can alternatively: double-click the item, press the left arrow key, or press **-** on the numeric keypad.
- Select an item to show the first page of that document or section in the display pane.
- Press * on the numeric keypad to show all subentries for the selection.
- Use the down and up arrow keys to move through the list of items.

Icons in the Contents Listing

Icons for entries in the top levels of the **Contents** pane listing represent the type of documentation so you can quickly find the kind of information you need for a product. See the table for icons in “Types of Documentation” on page 4-7.

Synchronize the Contents Listing and Demos Listing with the Display Pane

By default, the topic highlighted in the **Contents** pane matches the title of the page appearing in the display pane. The **Contents** pane listing is said to be synchronized with the displayed document. This feature is useful if you access documentation with a method other than the **Contents** pane, for example, using a link in a page in the display pane, or selecting a search result. With synchronization, you know what book and section the displayed page is part of, and where that section fits within the overall book. Note that synchronization

only applies to the major headings in a document. For pages that begin with lower level headings, the **Contents** pane listing does not synchronize.

You can turn off synchronization. To do so, use preferences. See “General — Keep Contents Synchronized” on page 4-38.

Synchronization applies to the **Contents** and **Demos** panes. The page shown in the display pane does not necessarily correspond to the selection in the **Index**, or **Search Results** panes. However, if you return to the **Contents** pane (or **Demos** pane), the displayed page synchronizes with the **Contents** (or **Demos**) pane.

This example illustrates synchronization for after performing a search. When you enter `"interactive plotting"` in the **Search for** field, the **Search Results** pane displays a list of results in the Help Navigator, with the first documentation result selected. The display pane shows the page corresponding to that first result, `Plotting Tools  Interactive Plotting`. When you click the **Contents** tab, the tree automatically expands to show `MATLAB > Graphics > Plots and Plotting Tools` and selects the `Plotting Tools  Interactive Plotting` entry.

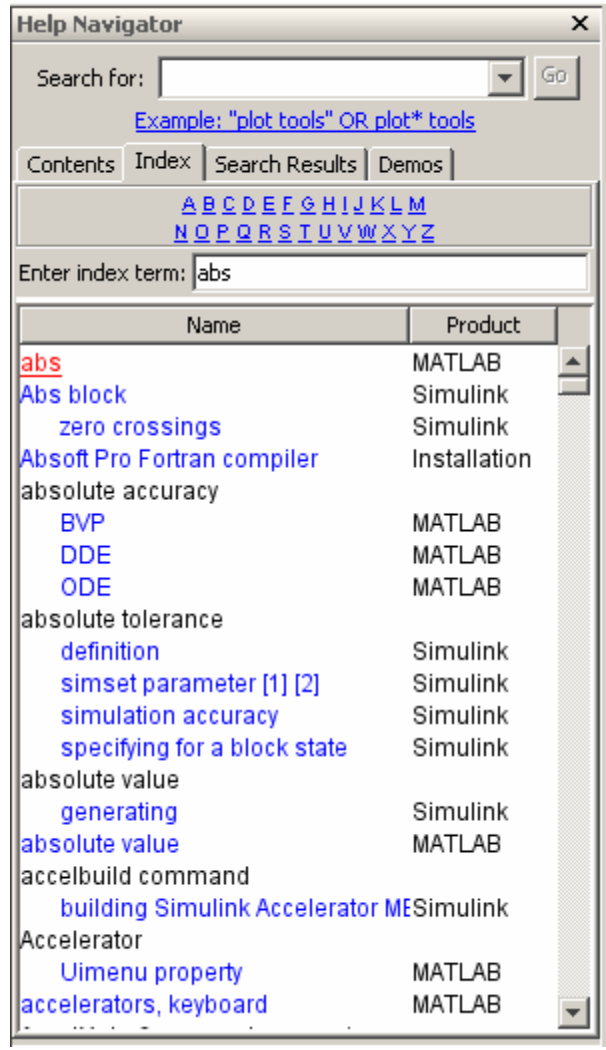
This example illustrates synchronization for demos. When you run

```
demo('matlab', 'graphics')
```

the **Demos** pane appears, with the MATLAB Graphics entry selected.

Index for the Help Browser

To find specific index entries (selected keywords) in the MathWorks documentation for installed products, use the **Index** in the **Help Navigator** pane.



1 Click the **Index** tab.

2 Type a word or words in the **Enter index term** field. As you type, the **Index** pane displays matching entries and their subentries (indented). It might take a moment for the display to appear. The index is not case

sensitive. If there is not a matching entry, it displays the page for the letter that your entry begins with.

The product whose documentation includes the matching index entry is listed next to the index entry, which is useful when there are multiple matching index entries. You might have to make the **Help Navigator** pane wider to see the product.

3 Select a blue index entry from the list (where blue represents a hyperlink) to display the page to which the term refers. Multiple links per entry are denoted by numbers in brackets following the term. (Black index entries are headings and do not link to any page.)

When you select an entry, its color becomes red. The page whose entry you selected appears in the display pane, scrolled to the location that the entry references.

4 To see more matching entries, scroll through the list.

Tips for Using the Index

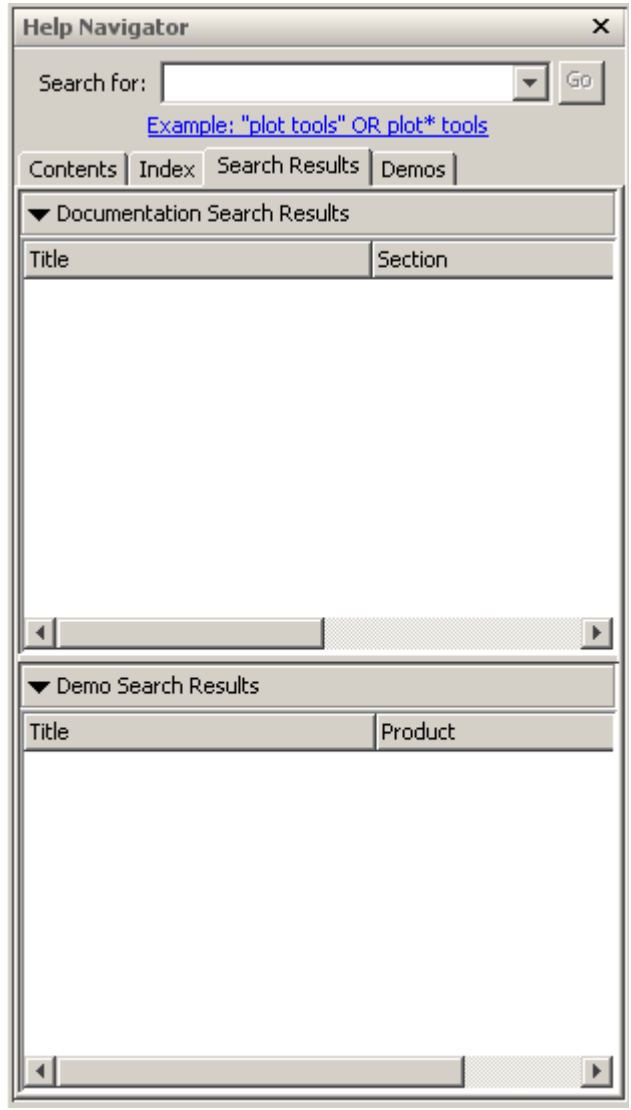
- To see entries for selected products only, select **File > Preferences > Help** and set the product filter.
- To see entries for all installed products, select **File > Preferences > Help**, and clear the **Enable product filter** check box.
- For more or different results, type a different term or reverse the order of the words you type. For example, if you are looking for an entry about tab completion for `Editor/Debugger`, a subentry does not exist. Instead type `tab completion` and there is a subentry `Editor/Debugger`.
- After selecting an entry, search for specified text in the displayed page using the **Find** tool, accessible from the Find button on the display pane toolbar.
- When there are multiple matching entries, refer to the product associated with each entry, which appears in the second column of the **Index** results. You might need to make the pane wider to see it.

- For different but related results, try using the **Search for** field—for instructions, see “Search Documentation and Demos with the Help Browser” on page 4-16.
- See “Specifying Colors for the Help Browser” on page 4-41 for information about changing the color of hyperlinks in the **Index**.

Search Documentation and Demos with the Help Browser

Searching in the Help Browser

To look for a specific word or phrase in the documentation or demos, use the **Search for** field in the **Help Navigator**.



1 To limit (or extend) the products whose documentation and demos are searched, set the product filter.

2 In the **Search for** field, type the word or words you want to find and click **Go** (or press **Enter** or **Return**). Some techniques for honing your search are

- Exact Phrase — Search for an exact phrase by enclosing words in quotation marks, for example, `"plot tools"`. For more information, see "Exact Phrases in Search" on page 4-21.
- Wildcard (Partial Word) — Search for variations of a word, also called partial word searching, by using the wildcard symbol (`*`) in place of letters in a word, for example, `plot* tools`. For more information, see "Wildcards in Search (Partial Word)" on page 4-21.
- Boolean Operators — Add the Boolean operators `AND`, `OR`, and `NOT` between search words to include or exclude words. By default, search assumes an `AND` between all search words and exact phrases. For more information, see "Boolean Operators in Search" on page 4-21.

The documents and demos containing all of the search words are listed in the **Search Results** tab. Two sets of results appear: **Documentation** and **Demo**. The number of results for each appears in parentheses at the top of the listing, and the total number for both sets of results appears in the lower left corner of the Help browser. Both sets of results have additional columns that list the **Product**, and for documentation, another column lists the **Section**. You might need to make the **Help Navigator** pane wider to see all columns.

3 Select an entry from the list of results. By default, the first documentation entry is automatically selected. If there are no documentation results, but there are demo results, the first demo entry is automatically selected.

The selected page appears in the display pane with all occurrences of the search words and exact phrases highlighted, using a different color for each search word or phrase. Highlights remain until you view another page or until you click the Refresh button  in the toolbar.

In the display pane, use the **Find** tool, accessible from the Find button  on the toolbar, to find a specified word in that page.

4 Search results are ordered by relevance. For example, for documentation, reference pages that match the search term appear first, followed by titles

that contain all search words, with pages containing a single instance of each search word appearing last. Change the display of search results to more easily find the most relevant results:

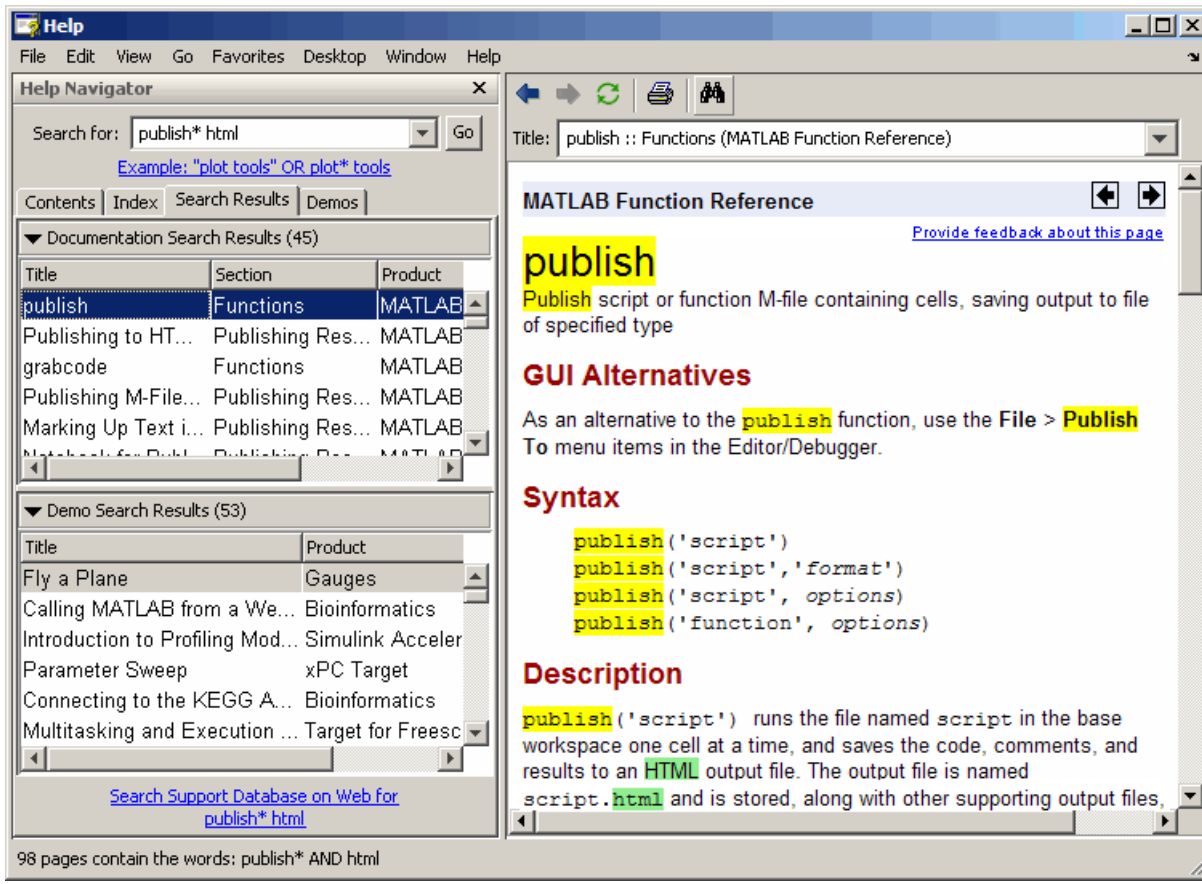
- Sort by column — Change the order of the results by clicking a column heading. For example, click **Product** to group results by product. Click **Title** to sort titles alphabetically. A triangular icon indicates the column on which you most recently sorted. Click the heading again to sort by the column but in the reverse order.

  After changing the order of results, to see results ordered by relevance, click **Go** to rerun the search.

- Reorder columns — Change the location of a column by dragging its heading to a new position. For example, you can drag the **Product** column to the middle for documentation results.

- Resize columns — Make columns wider or narrower by dragging the separator bar between the column headings. Similarly, make the Help Navigator wider or narrower.

5 For more results, you can search for the words in the Technical Support database of bug reports, solutions, and notes on the MathWorks Web site by clicking the link at the bottom of the **Search** results pane.



Function Alternative. From the Command Window, use `docsearch` to open the Help browser to the **Search** pane and search for the specified term. For example

```
docsearch('publish* html')
```

finds all pages that contain the word `publish` or its variations, such as `publishing`, `published`, and so on, and also contains the word `html`.

```
docsearch('"plot tools"')
```

finds all pages containing the exact phrase `plot tools`.

For details, see the `docsearch` reference page.

Wildcards in Search (Partial Word)

You can use the wildcard character (`*`) in place of letters or digits in your search terms. For example, `plot*` finds various forms of the word *plot*, such as *plot*, *plots*, and *plotting*. The search term `p*t` also finds those variations of *plot* as well as variations of *print* and *part*, among others.

You can use multiple wildcards in a word or search term. For example, `plot* tool*` finds *plotting tools*, among other terms. The term `p*t* tool*` finds not only *plotting tools*, but also *pointer tooltip*.

You cannot use a wildcard with just one letter or digit, nor can you include wildcards within an exact phrase. You cannot begin a word in a search term with a wildcard character. For example, these fail: `p*`, `"plot* tools"`, `plot *ool`.

Exact Phrases in Search

To find a phrase, type quotation marks around it. For example, `"plot tools"` finds only pages that include `plot tools` together, but does not find pages that include `plot` in one part of the page and `tools` in another part of the page. Specify an exact phrase to reduce the number of irrelevant results. For example, `"plot tools"` finds about 10 pages in MATLAB documentation, while `plot tools` finds about 100 pages.

You can specify more than one exact phrase, such as `"plot tools" "figure palette"` to find pages that contain both `"plot tools"` and `"figure palette"`. You cannot use a wildcard within an exact phrase.

Boolean Operators in Search

The search automatically performs a Boolean `AND` for multiple words. In the example `publish* html`, it finds all pages that have the word `publish` or its variations, and the word `html`.

You can refine the search by including the Boolean operators `NOT`, `OR`, and `AND` between words. The operators must be in all capital letters and there must be a space before and after each operator. The `NOT`s are evaluated first, followed by the `OR`s, and then the `AND`s.

Example Using Boolean Operators in Search. Type

```
plot* tools NOT time series
```

to find all pages that contain the words `plot` or its variations and `tools`, but not the phrase `time series`.

More About Search

These are the guidelines search uses:

- Insignificant words (a, an, the, of) are ignored.
- Search is not case sensitive.
- Search only finds letters and digits, but not symbols. To find a symbol, look for the word (for example, `plus` instead of +), use the **Index**, or see Operators and Special Characters in the MATLAB Functions — By Category. Another option is to search the PDF documentation, which supports searching for symbols — instructions to access the PDF file are included in "Printing the PDF Version of Documentation" on page 4-42.
- Search find words in comments or code for M-file and Model types demos. It finds comments in the M-file help for M-GUI demos. It does not search video demos.
- If you search for a function that is used in multiple products (called an overloaded function), the reference pages for all those products are listed. Use the **Product** column in **Search Results** to determine the reference page you want.

Get Fewer Results

If there are too many results for the search to be useful, try the following.

Problem	Try These Suggestions
Too many products	Select **File > Preferences > Help** and enable the product filter for specified products. For details, see "Product Filter" on page 4-37. Order results by product — click the **Product** column in **Search Results**. If you cannot see the column, make the pane wider.

Problem	Try These Suggestions
Pages are not about search word, but just mention it	Try the **Index** pane to see more important entries for that search word.
Too many irrelevant results	Type more than one word in the **Search for** field. Look for an exact phrase by enclosing words in quotations marks, such as `"plot tools"`. Use Boolean operators (in all capitals), for example, `printing AND figures NOT exporting`.
Topic is not relevant	Look at the **Section** column in **Search Results**, which provides context for the result. If you cannot see the column, make the pane wider.
Want to look only within part of a product's documentation	For products like MATLAB, you might want to search only part of the documentation. There is no way to do this in the Help browser. However, you might be able to accomplish that via PDF search. For example, you can search the "Getting Started with MATLAB" PDF file, or the "MATLAB External Interfaces" PDF file. Instructions to access the PDF file are included in "Printing the PDF Version of Documentation" on page 4-42.

Get More Results

If you want more results, try the following.

Problem	Try These Suggestions
No results for the product	Be sure the product filter is set correctly. Select **File > Preferences > Help** and disable the product filter, or at least ensure the products of interest are selected. For details, see "Product Filter" on page 4-37.

Problem	Try These Suggestions
No results but you know the word should be there	Try variations of the search word by using a wildcard symbol (`*`). For example, search for `preference*` to find all pages that contain either the word `preference` or the word `preferences`.
Not enough information	Try searching the Technical Support database of bug reports, solutions, and technical notes by clicking the link at the bottom of the **Search** results pane. If you are not running the most current version of MATLAB, try looking at the most current version on the Web site. It might contain additional information. For more information, see "Accessing Documentation on the Web" on page 4-8.

See Also. "Finding Files and Content Within Files" on page 5-49, which describes the Find Files tool you use to look for files and content within files, such as comments in M-files or code fragments.

Favorites

Favorites are bookmarks to pages in the Help browser documentation and M-file type demos.

Add Favorites

To designate the displayed page as a favorite (that is, to bookmark it),

1. Select **Favorites > Add to Favorites**.
2. The **Favorites Editor** dialog box opens. You can accept the defaults and click **Save**, or make changes to the entries:
   a. Use the **Label** provided, or change it to another term.
   b. Do not change the entry for **Callback**.
   c. Maintain the **Category** as `Help Browser Favorites` so you can access them from the **Favorites** menu.
   d. For **Icon**, keep the default `Help icon`, or choose another.

A favorite is implemented as a MATLAB shortcut, so the dialog box is the same as for the **Shortcut Editor**.

Favorites from previous releases are not migrated to a new release.

Go to Favorites

Select the **Favorites** menu to view the list of pages you previously designated as favorites (bookmarks). Select an entry and that page appears in the display pane.

Organize Favorites

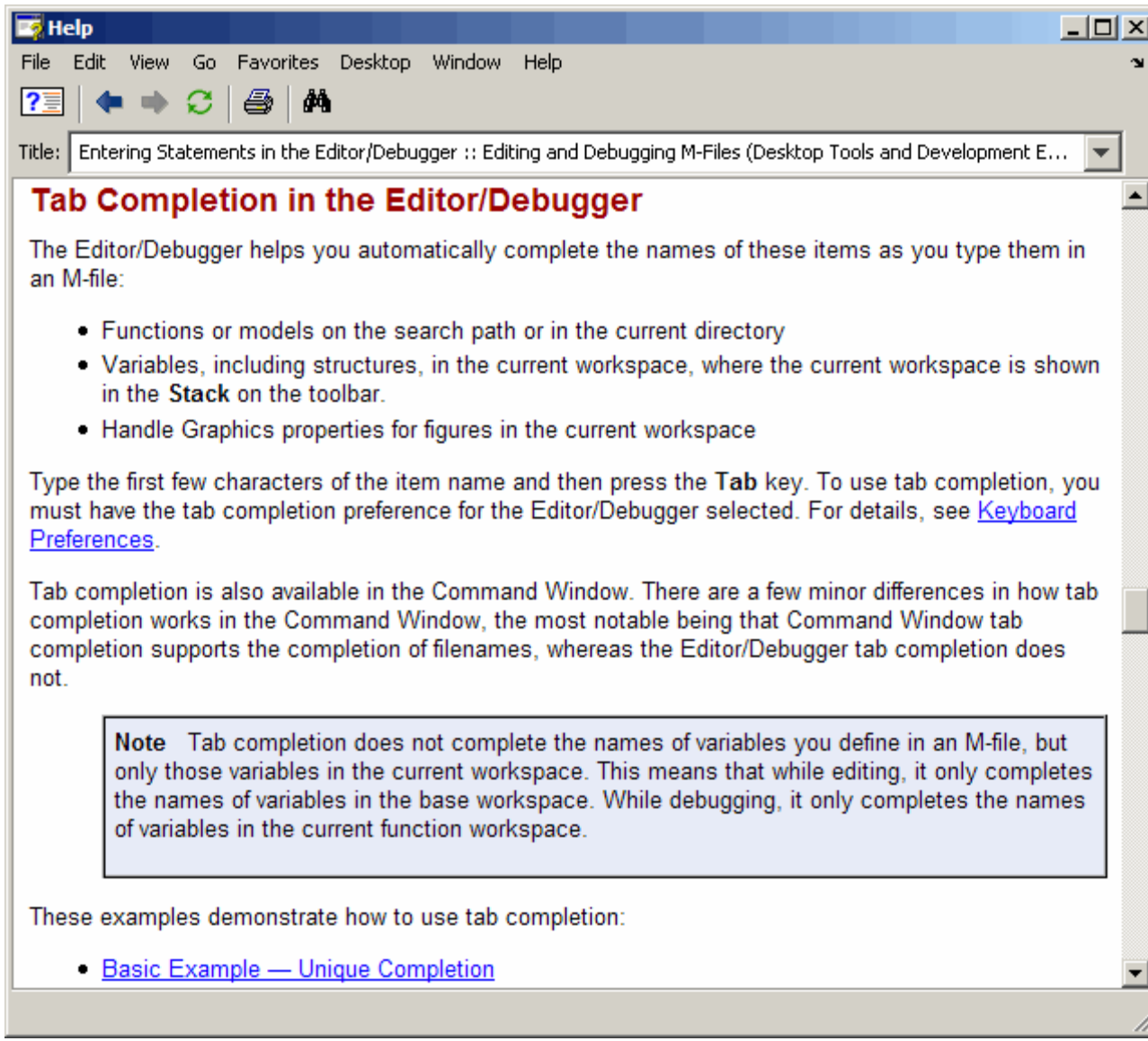
You can rename, remove, and reorder the list of favorites. Select **Favorites > Organize Favorites**. For more information, click **Help** in the **Organize Favorites** dialog box.

Viewing Documentation in the Help Browser

In this section...

About the Display Pane

After finding documentation with the **Help Navigator**, view the documentation in the display pane. The following illustration shows the **Help Navigator** closed to provide a larger area for viewing the information.



Browse to Other Pages

Use the arrow buttons in the page and in the toolbar to go to other pages.

View the next page in a document by clicking the Next page button at the top or bottom of the page. View the previous page in a document by clicking the Previous page button at the top or bottom of the page. These arrows allow you to move forward or backward within a single document. The arrows at the bottom of the page are labeled with the title of the page they go to.

View the page previously shown by clicking the Back button in the display pane toolbar. After using the Back button, view the next page shown by clicking the Forward button in the display pane toolbar. These buttons work like the forward and back buttons of popular Web browsers. You can also go back or forward by right-clicking a page and selecting **Back** or **Forward** from the context menu.

Links

Click links in the displayed page to go to a related topic for more information on the subject. Links appear underlined and in blue, while visited links appear in purple. Links to Web addresses open in the MATLAB Web Browser. Click the middle mouse button to open the linked page in a separate window.

Find Text in Displayed Pages

To find a phrase in the currently displayed page,

1. Click the Find button . In the resulting **Find** dialog box, type the word or phrase you are looking for. You can type a partial word, for example, `preference` to find all occurrences of `preference` and `preferences`. Use the check boxes to specify options. Click **Find Next**.

   The search begins at the current cursor position and the page scrolls to the first occurrence of the phrase in the page and highlights it.

2. To find more occurrences in that page, click **Find Next** or **Find Previous** in the **Find** dialog box, or use the keyboard shortcuts **F3** and **Shift+F3**.

   MATLAB beeps when a search for **Find Next** reaches the end of the page, or when a search for **Find Previous** reaches the top of the page. If you have **Wrap around** selected, it continues searching after beeping.

You can change the selection in the **Look in** field to search for the specified text in other MATLAB desktop tools.

See “Search Documentation and Demos with the Help Browser” on page 4-16 for instructions on looking through all the documentation instead of just one page.

Copy Information

To copy information from the display pane, such as code in an example, first select the information. Then right-click and select **Copy** from the context menu. You can then paste the information into another tool, such as the Command Window or Editor/Debugger, or into another application, such as a word processing application.

Evaluate a Selection

To run code examples that appear in the documentation, select the code in the display pane. Then right-click and select **Evaluate Selection** from the context menu. The statements execute in the Command Window.

Open a Selection

In a page in the display pane, select the name of a file that is provided with MATLAB, such as an M-file. Then right-click and select **Open Selection** from the context menu. The file opens in MATLAB. For example, an M-file opens in the Editor/Debugger.

Help on Selection

In a page in the display pane, select the name of a function that is provided with MATLAB. Then right-click and select **Help on Selection** from the context menu. The reference page for that function opens in Help browser.

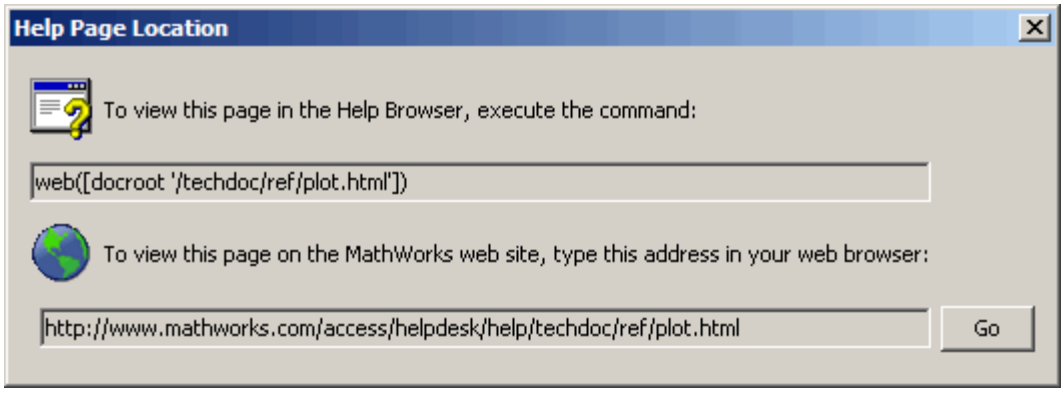
View the Page Source (HTML)

To view the HTML source for the currently displayed page, select **View > Page Source**. A read-only HTML version of the page appears in a separate window. You can copy selections from the HTML source and paste

them into other tools like the Editor/Debugger or Command Window, or into other applications.

View the Page Location

To view the location of the page currently displayed, select **View > Page Location**. The Help Page Location dialog box appears, providing the full path to the documentation file for both your local system and the MathWorks Web site.



You can copy the information from this window into an e-mail message or other tool to facilitate communication with other users or The MathWorks. For example, if you find a page of documentation that you know would be useful to a colleague running MATLAB, send them the link so they can view the page in the Help browser. Note that the `docroot` function used with the `web` function is unsupported, intended only for use by MathWorks products.

Click the **Go** button to view the same documentation page on The MathWorks Web site. This is useful if you do not see the information you are looking for on the page in view and know you are not running the most current version of MATLAB. The documentation for the most current version is on the Web site and might include more information than the documentation for your version. Note, though, that the documentation on the Web site might refer to features that are not part of your earlier-version product. See “Accessing Documentation on the Web” on page 4-8 for more information.

Demos in the Help Browser

In this section...

About Demos

MATLAB and related products include demos that you can access from the Help browser **Demos** pane.

There are four types of demos:

- M-file: Demos that tell a step-by-step story, including source code, commentary, and output. They are published from M-file scripts to HTML output using the Editor/Debugger. The first comment line of the demo M-file begins with two comment symbols (%%), and similarly, two comment symbols (%%) create a cell for each step. The MATLAB Graphics Square Wave from Sine Waves demo is an M-file type demo.
- M-GUI: Stand-alone tools for exploring a feature. An example is the MATLAB Graphics Vibrating Logo demo.
- Model: Simulink block diagrams. An example is the Engine Timing Simulation demo.
- Video: Movies that highlight key features in a tool. They play in your system browser and require the Macromedia Flash Player plug-in. Some also require an Internet connection. An example is the MATLAB Desktop and Command Window demo.

The MATLAB code and Simulink blocks used in the demos (except videos) are available for you to view and copy for use in your own applications.

See also Examples for each product in the **Contents** pane. These examples are similar to demos but are integrated in the documentation.

Using Demos

To access demos for the products you have installed,

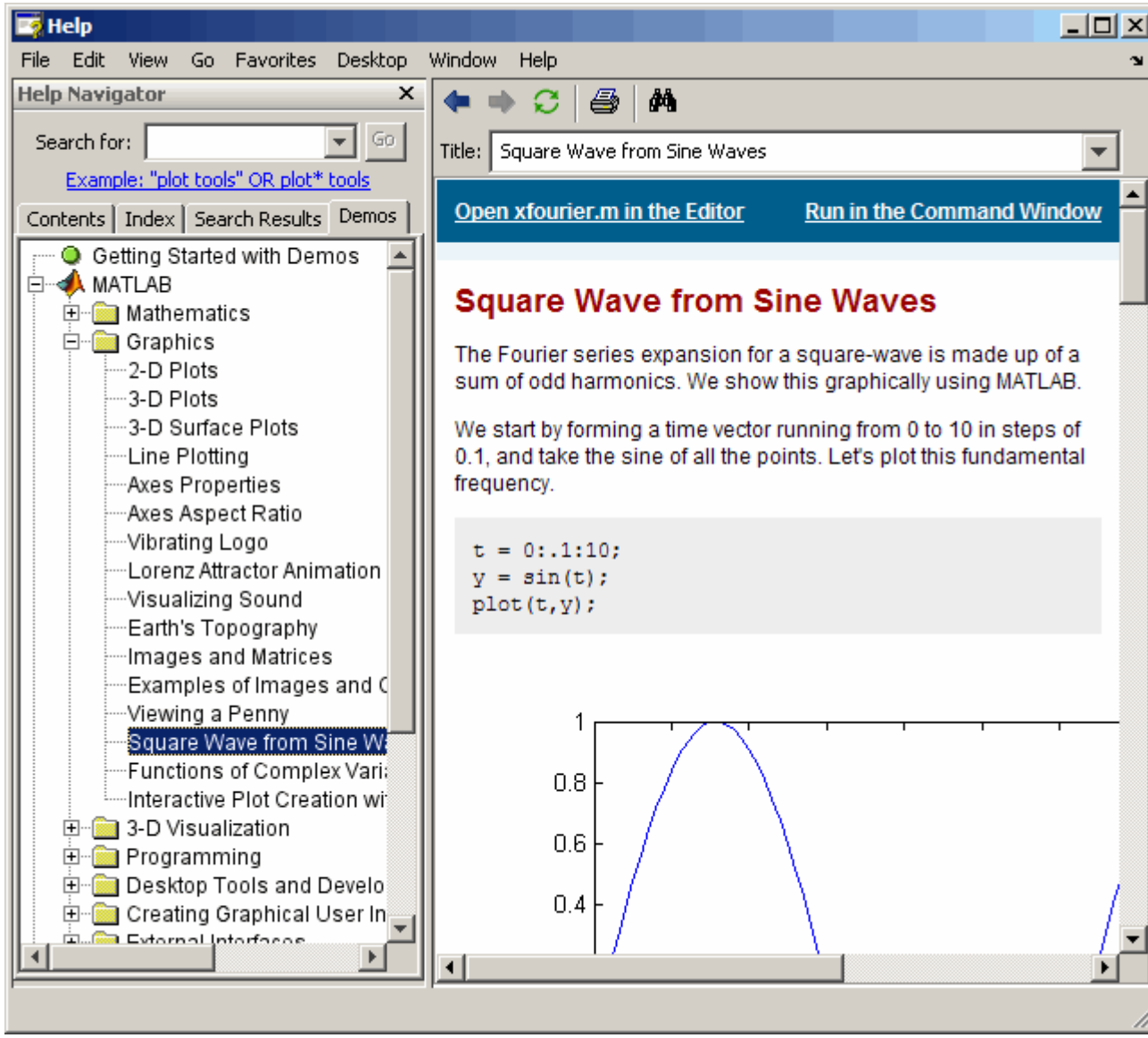
1. Click the **Demos** tab in the **Help Navigator**.

   You can also access demos from the **Start** button, by using the `demo` function, or from the **Help** menu for some tools.

2. Click the **+** for a product area to list the products or categories that have demos. Then click **+** for a product or product category to list its demos.

   All demos for that product or product area are listed in the display pane, and each includes the type of demo along with a thumbnail image that represents output from the demo.

3. Select a specific demo. Information about the demo appears in the display pane.



4 You can then view and run the demo, with specific options depending on the type of demo:

- For M-file demos, click the **Open filename in the Editor** link at the top left. This opens the M-file in the Editor/Debugger. From the Editor/Debugger, run the demo step by step by selecting **Cell > Evaluate Current Cell and Advance**.

  You can also click **Run in the Command Window**, and then follow the instructions that appear in the Command Window. You might need to scroll up to see all of the instructions.

  See also "Running Demos and Base Workspace Variables" on page 4-35.

- For M-GUI demos, click the **Open filename in the Editor** link at the top left. This opens the M-file in the Editor/Debugger.

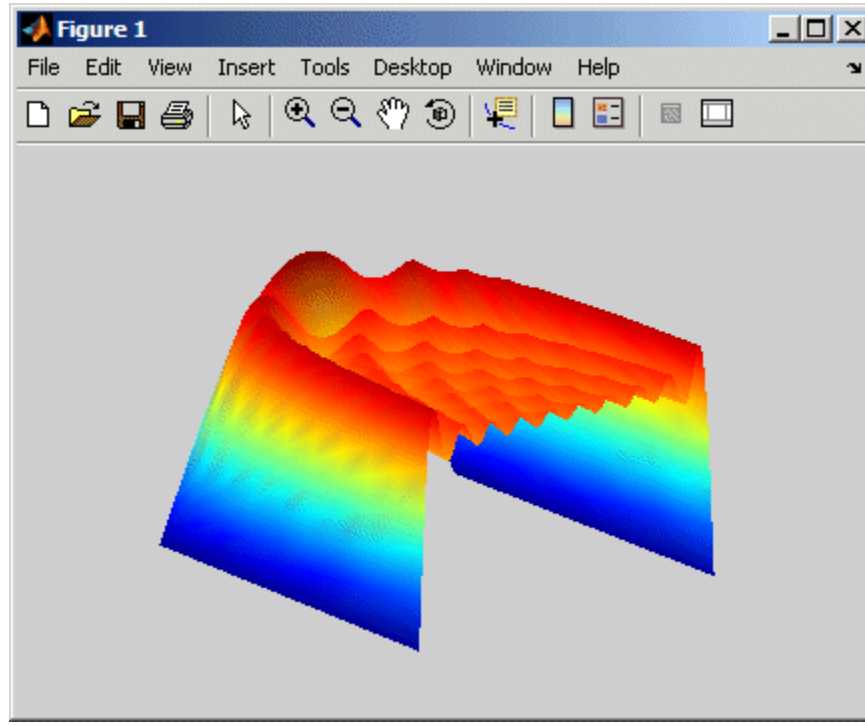
  Click the **Run this demo** link at the top right to start the GUI. Then follow the instructions in the GUI to proceed through the demo.

- For Model demos, click **Open this model** to open the block diagram.

- For Video demos, click the **Run this demo** link in the top right to play the video. These demos run in your system browser and require the Macromedia Flash Player plug-in. Some also require an Internet connection.

When you double-click a demo name in the **Help Navigator** pane, the demo file opens for M-file and Model demos, or runs for M-GUI and Video demos.

The following example shows the results of running the MATLAB Graphics Square Wave from Sine Waves demo (`xfourier`). In it, MATLAB generates a series of plots, culminating in the final one shown here.



Searching for Demos

You can use the Help browser search feature to find demos. Search find words in comments or code for M-file and Model types demos. It finds comments in the M-file help for M-GUI demos. It does not search video demos. For instructions, see “Search Documentation and Demos with the Help Browser” on page 4-16.

Running Demos and Base Workspace Variables

M-file demos run as scripts. Their variables are created in the MATLAB base workspace. If you have variables in the base workspace when you run an M-file demo, and the demo uses an identical variable name, there could be problems due to variable name conflicts. For example, a variable of yours might be overwritten by the demo. The demo’s variables remain in the base workspace until you clear them or quit MATLAB.

Function Alternative

To open the **Demos** pane in the Help browser, type `demo` in the Command Window. You can go directly to the demos for a specific product. For example

```
demo toolbox signal
```

opens the **Demos** listing for Signal Processing Toolbox.

To run an M-GUI demo, type the demo name in the Command Window. For example, type

```
vibes
```

to run the MATLAB Graphics demo showing an animated L-shaped membrane.

To run an M-file demo step by step from the Command Window, type `echodemo` followed by the demo name. For example, run

```
echodemo xfourier
```

Typing the demo name for an M-file demo runs the demo, but not step by step.

Typing the name of a model demo opens the block diagram.

Adding Your Own Demos

You can add your own demos so they appear in the **Demos** pane. For details, see in the online documentation.

Preferences for the Help Browser

In this section...

Product Filter

If you have MathWorks products in addition to MATLAB, such as Simulink, toolboxes, and blocksets, set the product filter to limit the product documentation and demos used:

1. Select **File > Preferences > Help**.
2. Under **Product filter**, select the check box for **Enable product filter**. Click **Select products**.

   The **Help Product Filter** dialog box opens.
3. Select the products whose documentation and demos you want to appear in the **Help Navigator**. Click **OK**.

   The **Help Navigator** updates to include only those products you specified. The product filter settings are saved for your next MATLAB session.
4. When you want to use documentation and demos for all installed products, in **Help Preferences**, clear the check box for **Enable product filter**.

The `Release Notes` entry in the **Help Product Filter** dialog box applies to the Release Notes overview document for a release, for example, all products in R2007b, not to the Release Notes for an individual product, for example, MATLAB Release Notes for R2007b. Release Notes for a product are considered part of the product’s documentation. For example, MATLAB Release Notes are considered part of MATLAB, and Database Toolbox Release Notes are considered part of Database Toolbox when you use the **Help Product Filter**.

Example Using the Product Filter

If you want to perform a search and have many products installed but know the information you are seeking is in MATLAB or Communications Toolbox, in the **Help Product Filter**, click **Clear All** and then select `MATLAB` and `Communications Toolbox`.

The **Contents** only shows MATLAB and Communications Toolbox documentation, the **Index** only shows entries for MATLAB and Communications Toolbox, and the **Search for** feature only looks in and shows results for MATLAB and Communications Toolbox documentation and demos. **Demos** lists only demos for MATLAB and Communications Toolbox.

PDF Reader — Specifying Its Location

If you want to view the PDF version of the documentation, use the link on the roadmap page for that product. To open the PDF file, the Help browser needs to know the location of your PDF reader (for example, Adobe Acrobat).

For Windows systems, MATLAB reads the PDF reader location from the registry, so you do not specify its location.

For UNIX systems, the default PDF reader is Acrobat and MATLAB determines its location. If a different command starts your PDF reader, specify it using preferences. Select **File > Preferences > Help**, and enter the full pathname in the **PDF reader** field or use the Browse for Folder (**...**) button to navigate your file system to select it.

General — Keep Contents Synchronized

By default, the displayed page is synchronized with the **Contents** or **Demos** listings. For more information about this feature, see “Synchronize the Contents Listing and Demos Listing with the Display Pane” on page 4-12.

To turn synchronization off, select **File > Preferences > Help**. Under **General**, clear the check box for **Keep contents tree synchronized with displayed document**. Select the check box to turn synchronization back on.

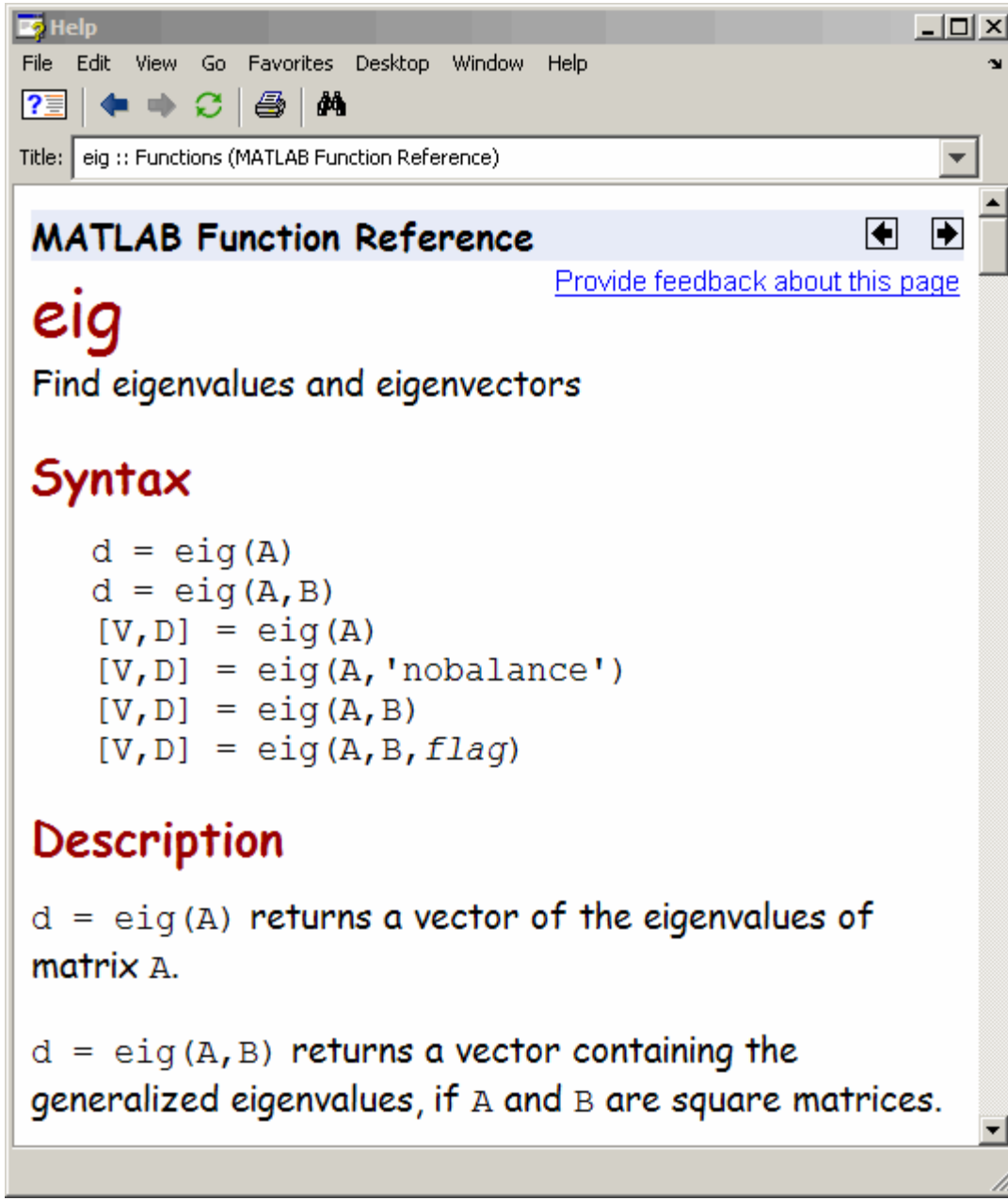
Help Fonts and Colors Preferences

Set fonts and colors for the Help browser the same way you would for other desktop tools. This section describes the process for the Help browser:

Specifying Font Name, Style, and Size

You can specify the font name (also called font type or family), style, and size used in the **Help Navigator**.

For the display pane, you can specify the font name and size for the text font, but changes do not impact the style. For the code font, your changes to size apply, but changes to name and style have no impact. The following example shows the results of specifying Comic Sans MS, bold, 14 point for the text font; note that the bold has no effect.


Help
File Edit View Go Favorites Desktop Window Help
Title: eig :: Functions (MATLAB Function Reference)
MATLAB Function Reference
Provide feedback about this page
eig
Find eigenvalues and eigenvectors
Syntax
d = eig(A)
d = eig(A,B)
[V,D] = eig(A)
[V,D] = eig(A,'nobalance')
[V,D] = eig(A,B)
[V,D] = eig(A,B,flag)
Description
d = eig(A) returns a vector of the eigenvalues of matrix A.
d = eig(A,B) returns a vector containing the generalized eigenvalues, if A and B are square matrices.

Use the same method as you would to specify fonts for any desktop tool — for more information, see “Fonts Preferences for Desktop Tools” on page 2-62. By default, the **Help Navigator** uses the desktop text font. The display pane is considered to be an HTML Proportional Text tool, and by default, uses the desktop text font.

This example changes the display pane font:

1. Select **File > Preferences > Fonts > Custom**.
2. From the **Desktop tools** list, select `HTML Proportional Text`. The Help browser display pane is considered to be an HTML Proportional Text tool, as is the MATLAB Web Browser. Changing the font preference affects both tools.
3. For **Font to Use**, select **Custom**, and then specify the font characteristics:
   - Name, for example, `Comic Sans`
   - Size in points, for example, `14`

   After you make a selection, the **Sample** area shows how the font will look.
4. Click **OK**. The Help display pane fonts use the new settings. The MATLAB Web Browser fonts also use the new settings.

Specifying Colors for the Help Browser

You can specify the background and text color used in the **Help Navigator**. Use the same method as you would to specify the background color for any desktop tool — for more information, see “Colors Preferences for Desktop Tools” on page 2-70.

If the background color preference for your desktop tools is a dark color, you might not be able to see index entries in the **Help Navigator** because they are links, for which the default color is blue. To see the links, change the **Hyperlink** color preference to a light or other contrasting color — for more information, see “Other Colors” on page 2-75.

You cannot specify colors for the Help browser display pane.

Printed Documentation

In this section...

About Printed Manuals

Generally, printed manuals are not provided for most products and tools. The printed manuals typically contain less information, and is also sometimes less current than the online documentation. If you want to purchase printed documentation, see the online store at the MathWorks Web site at `http://www.mathworks.com`.

Printing a Page from the Help Browser

To print the page currently shown in the Help browser, select **File > Print**, or click the Print button in the display pane toolbar. The Print dialog box appears.

Select **All** in the Print dialog box to print the entire page shown in the display pane. Specifying a range of pages, for example, `1 to 3`, prints the first three pages of the page currently shown in the display pane.

Complete the dialog box and click **OK** to print.

Printing the PDF Version of Documentation

If you need to print more than a few pages of documentation, or if you want the pages to appear as if they came from a printed book, print the PDF version of the documentation. PDF documentation is shown and printed using your PDF reader, usually Adobe Acrobat Reader. The PDF documentation reproduces the look and feel of a printed book. In the PDF document, use links from the table of contents, index, or within the document to go directly to the page of interest within that document. Note that some documentation available from the Help browser is not available in PDF format.

Note The Help browser accesses PDF documentation from the MathWorks Web site. Therefore, you need Internet access to view or print PDF documentation.

1. In the Help browser, click the **Contents** tab and select a product, for example, MATLAB.

   The roadmap page opens for that product, providing links to key documentation for that product.

2. Near the end of the roadmap page, listed under **Printing the Documentation Set**, are links for printing the documentation. Select the link for the item you want to print.

   The selected document is accessed from the MathWorks Web site. Your PDF reader opens, displaying the documentation.

   If you are using a UNIX platform and cannot open the PDF documentation, check the Help preferences. See "PDF Reader — Specifying Its Location" on page 4-38 for more information.

3. To print the documentation, select **Print** from the **File** menu in your PDF reader.

Help Functions

In this section...

About Help Functions

There are several help functions that provide forms of help different than the Help browser documentation, or provide alternative ways to access the Help browser information.

Summary Table of Help Functions

Function	Description
`dbtype`	Displays specified M-file with line numbers. If you want to see only the input and output arguments for a function, use `dbtype function 1`, which displays the first line of the M-file.
`demo`	Displays the **Demos** pane in the Help browser, from which you can access demonstrations for the products you have installed. With an argument, runs the specified demo.
`doc`	Displays in the Help browser, the reference page for the specified function, block, or property. Usually more extensive than results for the `help` function, the reference page provides syntax, a description, examples, illustrations, and links to related functions.
`docopt`	On UNIX systems, specifies Web browser information, used when displaying Internet Web pages.
`docsearch`	Run the Help browser search feature for the specified term.
`help`	Displays M-file help (a description and syntax) in the Command Window for the specified function. For MDL-files, displays a description of the model.

Function	Description
`helpbrowser`	Opens the Help browser, the MATLAB interface for accessing documentation.
`helpdesk`	Opens the Help browser. In previous releases, `helpdesk` displayed the Help Desk, which was the precursor to the Help browser. This function will be removed in a future release.
`helpwin`	Displays in the Help browser a list of all functions, and provides access to M-file help for the functions.
`lookfor`	Displays in the Command Window a list and brief description of all functions whose brief description includes the specified keyword.
`web`	Opens the specified URL in the specified browser. Use `web` in your own M-files to display HTML documentation you create for your work.
`whatsnew`	Displays the Release Notes in the Help browser.

View Function Reference Pages — the doc Function

To view the reference page for a function, block, or property in the Help browser, use `doc`. For example, type

```
doc format
```

to view the reference page for the `format` function.

Overloaded Functions with the doc Function

When a function name is used in multiple products, it is said to be an overloaded function. The `doc` function displays the reference page for the first function on the MATLAB search path having that name, and displays a hyperlinked list of the overloaded functions in the Command Window.

For example, using the default search path

```
doc set
```

displays the reference page for the MATLAB `set` function in the Help browser. The Command Window displays a hyperlinked list of the `set` functions located in other directories, such as

```
database/set
```

which is the `set` function for Database Toolbox. Click a link to go to that `set` reference page.

To directly get the reference page for an overloaded function, specify the name of the directory containing the function you want the reference page for, followed by the function name. For example, to display the reference page for the `set` function in Database Toolbox, type

```
doc database/set
```

Some products have more than one function with the same name. For example, MATLAB includes a built-in `get` function in the `graphics` directory and a `get` function in the MATLAB `serial` directory (for serial port functions). Type

```
doc get
```

The reference page for the MATLAB `graphics` built-in `get` function appears, and the Command Window lists overloaded functions in other products. But the list does not include any overloaded functions in the same product. Therefore, `get` in the MATLAB `serial` directory is not listed as an overloaded function. Type

```
doc ('get (serial)')
```

to display the reference page for the `get` function located in the MATLAB `serial` directory.

Getting Help in the Command Window — the help Function

To quickly view a brief description and syntax for a function in the Command Window, use the `help` function. For example, typing

```
help bar
```

displays a description and syntax for the `bar` function in the Command Window. This is called the M-file help. For other arguments you can supply, see the reference page for `help`.

Note M-file help displayed in the Command Window uses all uppercase characters for the function and variable names to distinguish them from the rest of the text. When typing function names, however, use lowercase characters. Some functions for interfacing to Java do use mixed case; the M-file help accurately reflects that, and you should use mixed case when typing them.

If you need more information than the `help` function provides, use the `doc` function, which displays the reference page in the Help browser. It can include color, images, links, and more extensive examples than the M-file help. For example, typing

```
doc bar
```

displays the reference page for the `bar` function in the Help browser.

Overloaded Functions with the help Function

When a function name is used in multiple products, it is said to be an overloaded function. The `help` function displays M-file help for the first function on the MATLAB search path having that name, and displays a hyperlinked list of the overloaded functions at the end.

For example, using the default search path

```
help set
```

displays M-file help for the MATLAB `set` function, and displays a hyperlinked list of the `set` functions residing in other directories, such as

```
database/set
```

which is the `set` function for Database Toolbox. Click a link to display the M-file help for that `set` function.

To directly get help for an overloaded function, specify the name of the directory containing the function you want help for, followed by the function name. For example, to get help for the `set` function in Database Toolbox, type

```
help database/set
```

Creating M-File Help for Your Own M-Files

You can create M-file help for your own M-files and access it using the `help` command. See the `help` reference page for details.

Help in the Current Directory Browser

The Help Report and the Contents Report provide other ways of looking at and managing help for M-files — see “Directory Reports in Current Directory Browser” on page 7-2.

You can also see the help for an M-file in the Current Directory browser if you have its preference for **Show M, MDL, and MAT file contents** selected.

Help for Model Files

Use the `help` function with an MDL filename to display the complete description for the model file. For example, run

```
help f14_dap.mdl
```

and MATLAB displays the description of the Simulink F-14 Digital Autopilot High Angle of Attack Mode, as defined in the **Model > Properties > Description**.

```
Multirate digital pitch loop control for F-14 control design
demonstration.
```

If Simulink is installed, you do not need to include the `.mdl` extension.

You can see the same description in the Current Directory browser if you have its preference for **Show M, MDL, and MAT file contents** selected.

Getting Pop-Up Help for Functions

MATLAB provides pop-up help for MATLAB functions from both the Editor/Debugger and the Command Window. Pop-up help, also referred to as "help on selection," enables you to view documentation for a function without requiring you to interrupt the flow of your work to open the Help browser.

To access pop-up help, follow these steps:

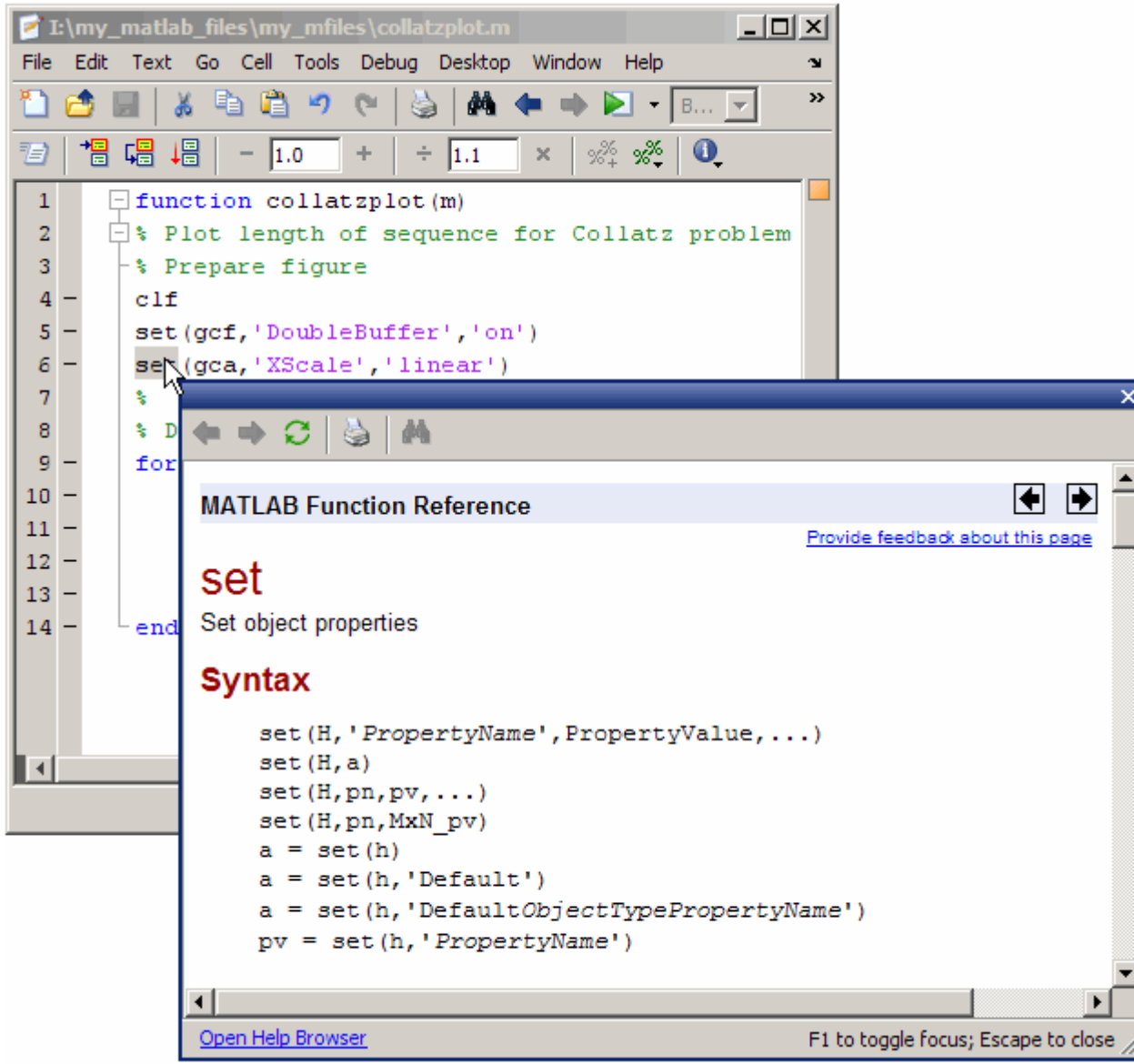
1. Select a function or click the pointer in a function for which you want information.
2. Press **F1** or right-click and select **Help on Selection**.

   Documentation for the function opens in a small pop-up window, similar to that shown in the figure that follows.
3. Move or resize the pop-up window, if desired.
4. Press the **F1** key to toggle focus between the pop-up window and Command Window or the Editor/Debugger.

   When you type in the Command Window or the Editor/Debugger, the pop-up window remains open.
5. With the pop-up window active, press the Escape (**Esc**) key to close the pop-up window.

If you want to get documentation on a different function while the pop-up window is open, there is no need to close it. Repeat steps 1 and 2—the pop-up window refreshes with documentation for the new function.

If you want to open the Help browser from the pop-up window, click the **Open Help Browser** link at the bottom of the pop-up window. The Help browser opens and the pop-up window closes.


I:\my_matlab_files\my_mfiles\collatzplot.m
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1
function collatzplot(m)
% Plot length of sequence for Collatz problem
% Prepare figure
clf
set(gcf,'DoubleBuffer','on')
set(gca,'XScale','linear')
MATLAB Function Reference
Provide feedback about this page
set
Set object properties
Syntax
set(H,'PropertyName',PropertyValue,...)
set(H,a)
set(H,pn,pv,...)
set(H,pn,MxN_pv)
a = set(h)
a = set(h,'Default')
a = set(h,'DefaultObjectTypePropertyName')
pv = set(h,'PropertyName')
Open Help Browser
F1 to toggle focus; Escape to close

Other Forms of Help

In this section...

Documentation for Other Products

The Help browser provides access to documentation for all products installed on your system. To view the online documentation for all MathWorks products, use the MathWorks Web site at `http://www.mathworks.com/access/helpdesk/help/helpdesk.shtml`.

Product-Specific Help Features

In addition to the Help browser and help functions, some products and tools allow other methods for getting help. You will encounter some methods in the course of using a product, such as entries in the **Help** menu, **Help** buttons in dialog boxes, and selecting **Help** from a context menu. These methods all display context-sensitive help. Other methods for getting help, such as pressing the **F1** key, are described in the documentation for the product or tool that uses the method.

User-Contributed M-Files

You can download M-files contributed by users and developers of MATLAB, Simulink, and related products from MATLAB Central. Before you write an M-file yourself, especially if it seems to be a generic utility, check the list of contributed files to see if someone has already written it. These files are freely contributed and can be used without charge by anyone

who downloads them. To view the files available to download, go to the MATLAB Central File Exchange page on the MathWorks Web site, `http://www.mathworks.com/matlabcentral/fileexchange/index.jsp`. You can access this from any desktop component via **Help > Web Resources**.

If you write M-files that you think would be of use to others, consider submitting them to the MATLAB Central File Exchange via the Web page.

Technical Support

Technical Support provides help for problems you have with MathWorks products:

- Find specific Technical Support information using the Help browser **Search** feature. Run a search for a specified term. The end of the results list includes a link that runs the same search on the support database. This database, on the MathWorks Web site, provides the most up-to-date solutions, bug reports, and technical notes for questions posed by users.
- Select **Help > Web Resources > Support** to go to the Support Web page (`http://www.mathworks.com/support`). The page displays in your system's default Web browser. You can find out about other types of information such as third-party books, ask questions, make suggestions, view known bugs and workarounds, and report possible bugs.
- If you cannot access the Web site, e-mail Technical Support using the address `support@mathworks.com`. You must provide your license number to obtain support. It is helpful if you also provide your operating system and MATLAB version number. You can obtain the information by running the `ver` function or by selecting **Help > About**.

Newsgroup for MathWorks Products

The Usenet newsgroup for MATLAB and related products, `comp.soft-sys.matlab`, (also known as `cssm`) is read by thousands of users worldwide. Access the newsgroup to ask for or provide help or advice. You can read and submit postings as well as view and search through a sizable archive of postings using the MATLAB Central Newsgroup Access Web page on the MathWorks Web site, `http://www.mathworks.com/matlabcentral`. You can access this via **Help > Web Resources > MATLAB Newsgroup Access** from any desktop component.

First-time users to the newsgroup should read the newsgroup FAQ, linked to from the MATLAB Central page. It is a good practice to try to solve your own problem using the documentation and Technical Support database before posting a question to the newsgroup. Be sure to post with a meaningful subject that briefly describes the nature of the issue.

Other Resources for MATLAB Information

Following are some additional resources for help with MATLAB and related products:

- Newsletters — The MathWorks publishes News and Notes twice a year, containing feature articles, technical notes, and product information for MATLAB users. More frequently, The MathWorks issues MATLAB Digest, an electronic bulletin consisting of technical notes, solutions, and timely announcements to the user community. Select **Help > Web Resources > MATLAB Newsletters** or see `http://www.mathworks.com/company/newsletters/`.
- Books — There are hundreds of MATLAB based books. For a list with descriptions, see `http://www.mathworks.com/support/books/`.
- Seminars and Training — The MathWorks regularly presents free seminars on special topics conducted in various locations. Webinars on special topics are presented via the Web. The MathWorks offers training classes for MATLAB and other products. For details, see `http://www.mathworks.com/company/events/`.
- `Mathtools.net` — This is a technical computing Web portal with links to many resources for MATLAB users. See `http://www.mathtools.net/`.

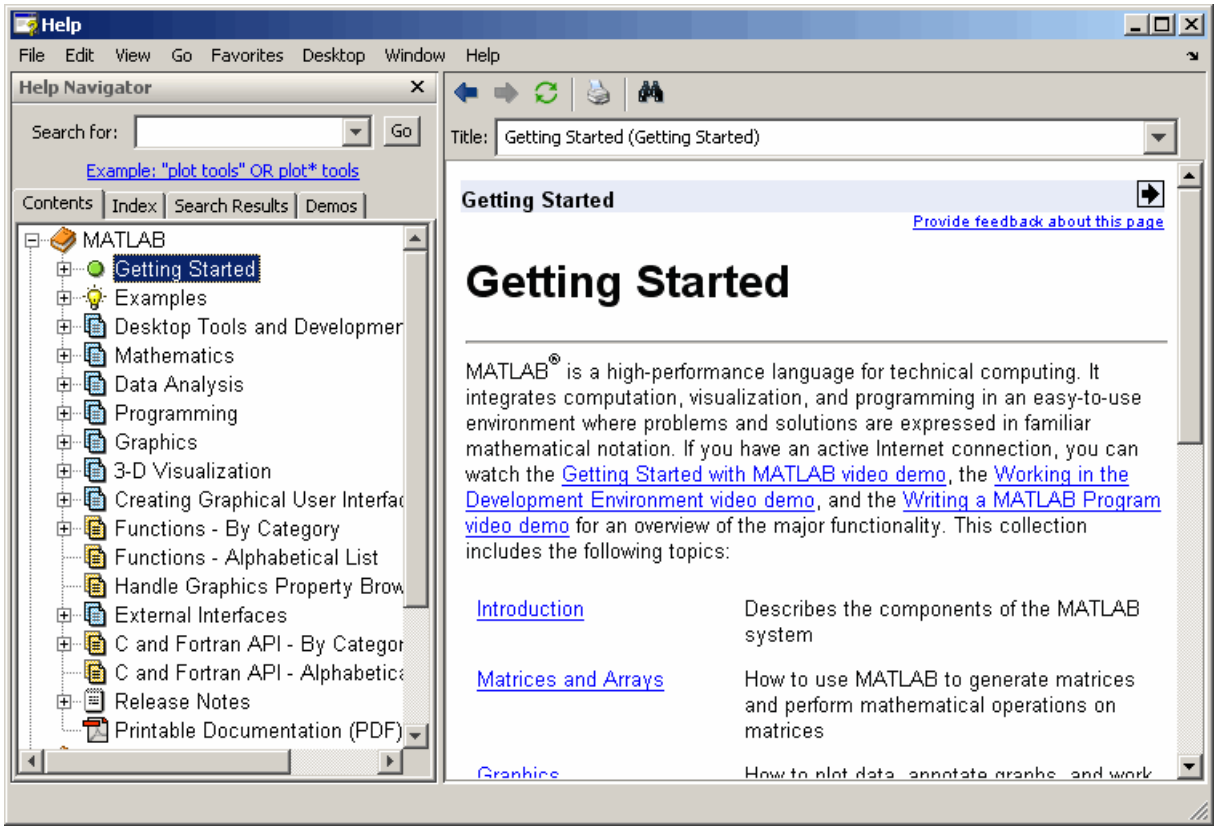
Version and License Information

If you need the version or license information for a product, select **About** from the **Help** menu for that product. The version is displayed in an About dialog box. If the product does not have a **Help** menu, use the `ver` function. To see the license number for MATLAB, type `license` in the Command Window. See also the `ver`, `version`, and `license` reference pages.

You can access information about your passcodes and licenses, as well get trial versions of products, using **Help > Web Resources > MathWorks Account**.

Provide Feedback

To report problems or provide comments or suggestions to The MathWorks about the documentation, help features, or products, use the **Provide feedback on this page** link on the top and bottom of every page in the Help browser.



5

Workspace, Search Path, and File Operations

If you have an active Internet connection, you can watch the Workspace Browser video demo, the Array Editor video demo, and the Current Directory Browser video demo for an overview of the major functionality.

MATLAB Workspace (p. 5-2)	The workspace is the set of variables maintained in memory during a MATLAB session. Use the Workspace browser or equivalent functions to view the workspace.
Viewing and Editing Workspace Variables with the Array Editor (p. 5-12)	View and make changes to variables using the Array Editor.
Search Path (p. 5-23)	MATLAB uses a search path to find M-files and other MATLAB related files. View and change the path using the Set Path dialog box or equivalent functions.
File Management Operations (p. 5-35)	Search for, view, open, and make changes to MATLAB related directories and files, using the Current Directory browser or equivalent functions.

MATLAB Workspace

In this section...

About the MATLAB Workspace

The MATLAB workspace consists of the set of variables (named arrays) built up during a MATLAB session and stored in memory. You add variables to the workspace by using functions, running M-files, and loading saved workspaces. For example, if you type

```
t = 0:pi/4:2*pi;
y = sin(t);
```

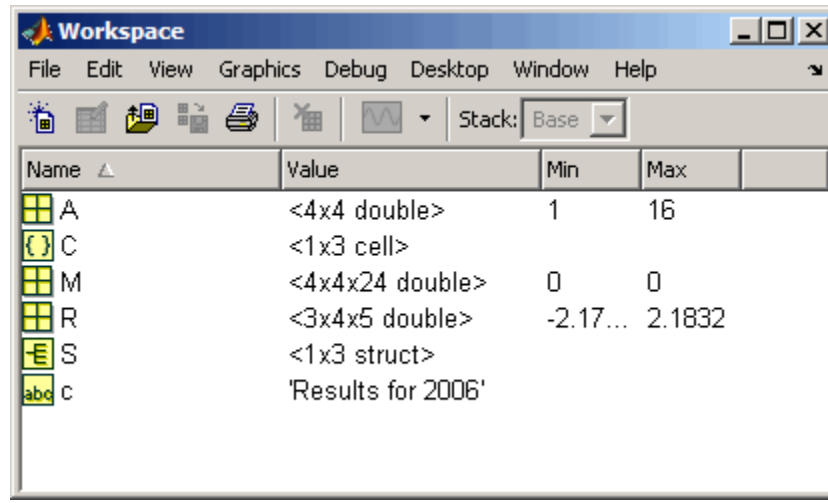
the workspace includes two variables, `y` and `t`, each having nine values.

You can perform workspace operations and related features using the Workspace browser. Equivalent functions are available and are documented with each feature of the Workspace browser. If you have an active Internet connection, you can watch the Workspace browser video demo for an overview of the major functionality:

Opening the Workspace Browser

To open the Workspace browser, select **Workspace** from the **Desktop** menu in the MATLAB desktop, or type `workspace` at the Command Window prompt.

The Workspace browser opens.



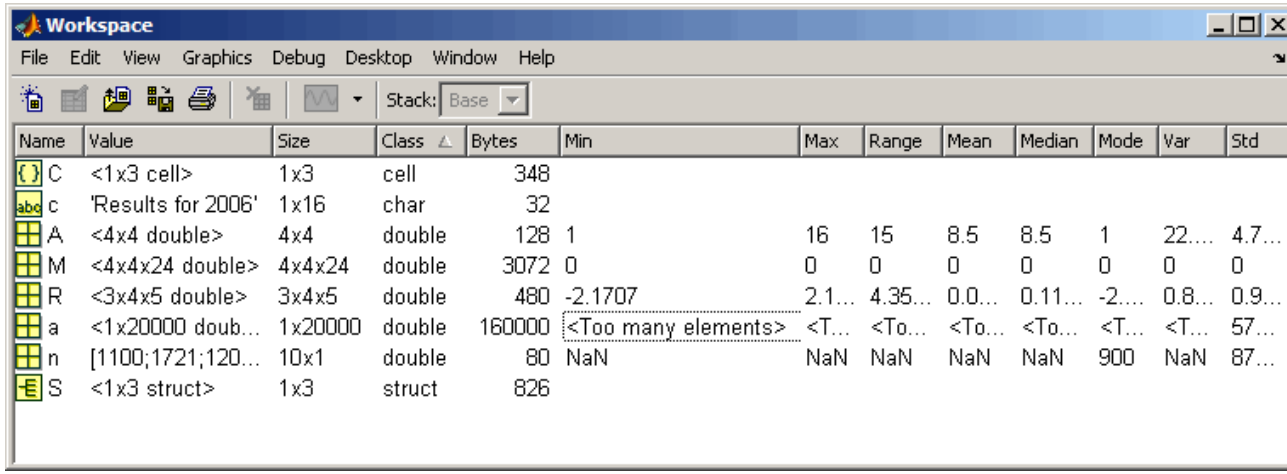
Viewing and Editing Values in the Current Workspace

The Workspace browser shows the name of each variable, its value, and the **Min** and **Max** calculations, which MATLAB computes using the `min` and `max` functions, and updates automatically. The icon for each variable denotes its class.

- You can display additional columns, including size (dimensions), size in bytes, class, and other common statistical calculations such as mode and standard deviation. To show or hide columns, select **View > Choose Columns** or right-click any column heading. To specify the size limit for calculations and how NaNs are considered, use “Preferences for the Workspace Browser” on page 5-10.
- To resize the columns of information, drag the column header borders. To reorder columns, drag a column header to a new position.

- You can select the column on which to sort as well as reverse the sort order of any column. Click a column heading to sort on that column. Click the column heading again to reverse the sort order in that column. For example, to sort on **Name**, click the column heading once. To change from ascending to descending, click the heading again. You cannot sort by the **Value** column in the Workspace browser. Alternatively, select **View > Sort By**
- You can edit variable values directly in the Workspace browser **Value** column. To edit a value, select the row to change in the **Value** column and type the new value.

This illustration shows the Workspace browser with all columns in view, sorted by the **Class** column, as indicated by the triangle in that column heading. Preferences are set to show statistical values only for arrays under `10,000` elements, so results for `a`, `(1 x 20000)`, do not appear. The preference for treatment of NaNs is set to include them in calculations, as seen for `n`. With the preference set to ignore NaNs, values other than `NaN` would appear.



Name	Value	Size	Class	Bytes	Min	Max	Range	Mean	Median	Mode	Var	Std
C	<1x3 cell>	1x3	cell	348								
c	'Results for 2006'	1x16	char	32								
A	<4x4 double>	4x4	double	128	1	16	15	8.5	8.5	1	22....	4.7...
M	<4x4x24 double>	4x4x24	double	3072	0	0	0	0	0	0	0	0
R	<3x4x5 double>	3x4x5	double	480	-2.1707	2.1...	4.35...	0.0...	0.11...	-2....	0.8...	0.9...
a	<1x20000 doub...	1x20000	double	160000	<Too many elements>	<T...	<To...	<To...	<To...	<T...	<T...	57...
n	[1100;1721;120...	10x1	double	80	NaN	NaN	NaN	NaN	NaN	900	NaN	87...
S	<1x3 struct>	1x3	struct	826								

Function Alternative

Use `who` to list the current workspace variables. Use `whos` to list the variables and information about their size and class. For example:

```
>> who
```

```
Your variables are:

A  C  M  R  S  c

>> whos
 Name      Size              Bytes  Class     Attributes

  A         4x4                 128  double
  C         1x3                 348  cell
  M         4x4x24             3072  double
  R         3x4x5               480  double
  S         1x3                 826  struct
  c         1x16                 32  char
```

Use `exist` to see if the specified variable is in the workspace.

Saving the Current Workspace

The workspace is not maintained across MATLAB sessions. When you quit MATLAB, the workspace is cleared. You can save any or all of the variables in the current workspace to a MAT-file, which is a MATLAB specific binary file. You can then load the MAT-file at a later time during the current or another session to reuse the workspace variables. MAT-files use a `.mat` extension.

Note The `.mat` extension is also used by Microsoft Access.

Saving All Variables

To save all of the workspace variables using the Workspace browser,

1. From the **File** menu, select **Save Workspace As**, or click the Save button  in the Workspace browser toolbar.

   The **Save** dialog box opens.

2. Specify the location and **File name**. MATLAB automatically supplies the `.mat` extension.

3. Click **Save**.

The workspace variables are saved under the MAT-file name you specified.

You can also save the workspace variables from the desktop by selecting **Save Workspace As** from the **File** menu.

Saving Selected Variables

To save some but not all of the current workspace variables,

1. Select the variable in the Workspace browser. To select multiple variables, **Shift**+click or **Ctrl**+click.
2. Right-click and from the context menu, select **Save As**.

   The **Save to MAT-File** dialog box opens.
3. Specify the location and **File name**. MATLAB automatically supplies the `.mat` extension.
4. Click **Save**.

   The workspace variables are saved under the MAT-file name you specified.

To specify preferences for saving MAT-files, see "MAT-Files Preferences" on page 2-79.

Function Alternative

To save workspace variables, use the `save` function followed by the filename you want to save to. For example,

```
save('june10')
```

saves all current workspace variables to the file `june10.mat`.

If you don't specify a filename, the workspace is saved to `matlab.mat` in the current working directory. You can specify which variables to save, as well as control the format in which the data is stored, such as ASCII. For these and other forms of the function, see the reference page for `save`. For a related function, see `genvarname`. MATLAB provides additional functions for saving

information — see “Data Import and Export” in the MATLAB Programming documentation.

Loading a Saved Workspace and Importing Data

To load saved variables into the workspace,

1. Click the Import Data button on the toolbar in the Workspace browser.

   The **Open** dialog box opens.

2. Select the MAT-file you want to load and click **Open**.

   The variables and their values, as stored in the MAT-file, are loaded into the current workspace. If any variables being loaded have the same names as variables in the current workspace, the values from the MAT-file replace the values in the current workspace. Any variables in the MAT-file that are not in the workspace are added to the workspace.

Function Alternative

Use `load` to open a saved workspace. For example,

```
load('june10')
```

loads all workspace variables from the file `june10.mat`.

Importing Data

MATLAB provides other methods and functions for loading information. You can use one of these methods, the Import Wizard, from the Workspace browser — select **Edit > Paste to workspace** or use **Ctrl+V** to import data to MATLAB using the Import Wizard. For more information on the Import Wizard and other methods for loading information, see the “Using the Import Wizard”.

Viewing Variables in MAT-Files

Use the Current Directory browser to view the contents of a MAT-file without loading the file into MATLAB. For details, see “Finding Files and Content Within Files” on page 5-49.

Function Alternative. Use `whos` with the `-file` option.

Changing and Copying Variable Names

To rename a variable in the workspace, right-click the variable in the Workspace browser and select **Rename** from the context menu. Type the new variable name over the existing name and press **Enter** or **Return**.

To copy variable names to the clipboard, select the workspace variables and select **Edit > Copy**. You can then paste the names, for example, into the Command Window. Multiple variables are comma separated.

Deleting Workspace Variables

You can delete a variable, which removes it from the workspace. To delete a variable using the Workspace browser,

1. In the Workspace browser, select the variable, or **Shift**+click or **Ctrl**+click to select multiple variables. To select all variables, choose **Select All** from the **Edit** or context menus.
2. Press the **Delete** key on your keyboard or click the Delete button on the Workspace browser toolbar.
3. A confirmation dialog box might appear. If it does, click **OK** to clear the variables.

   The confirmation dialog box appears if you selected that preference. For more information, see “Confirmation Dialogs Preferences” on page 2-81.

To delete all variables, select **Edit > Clear Workspace** from any desktop tool.

Function Alternative

Use the `clear` function to clear variables from the workspace. For example,

```
clear A M
```

clears the variables `A` and `M` from the workspace.

Viewing Base and Function Workspaces Using the Stack

When you run M-files, MATLAB assigns each function its own workspace, called the function workspace, which is separate from the MATLAB base workspace. To access the base and function workspaces when running or debugging M-files, use the **Stack** field in the Workspace browser. The **Stack** field is only available in debug mode and otherwise is grayed out. The **Stack** field is also accessible from the Array Editor and the Editor/Debugger. See "Finding Errors, Debugging, and Correcting M-Files" on page 6-84for more information. See also the `dbstack` and `evalin` functions.

Creating Plots from the Workspace Browser

From the Workspace browser, you can generate a plot of a variable. To create a plot, click the Plot button  on the Workspace browser toolbar and select the plot type. The plot appears in a figure window. The button itself changes to reflect the currently selected style of plot, for example bar or stem.

This feature is only available for variables whose data types can be plotted, such as numeric. Open the variable in the Array Editor for additional plotting options.

In addition, you can right-click the variable you want to plot. From the context menu, choose the type of plot you want to create.

You can also **Shift**+click or **Ctrl**+click to select multiple variables to plot together. When one of the variables is named `time`, `t`, or `T`, MATLAB assumes it is the independent variable.

For more information about creating graphs in MATLAB, see the *MATLAB Graphics* documentation.

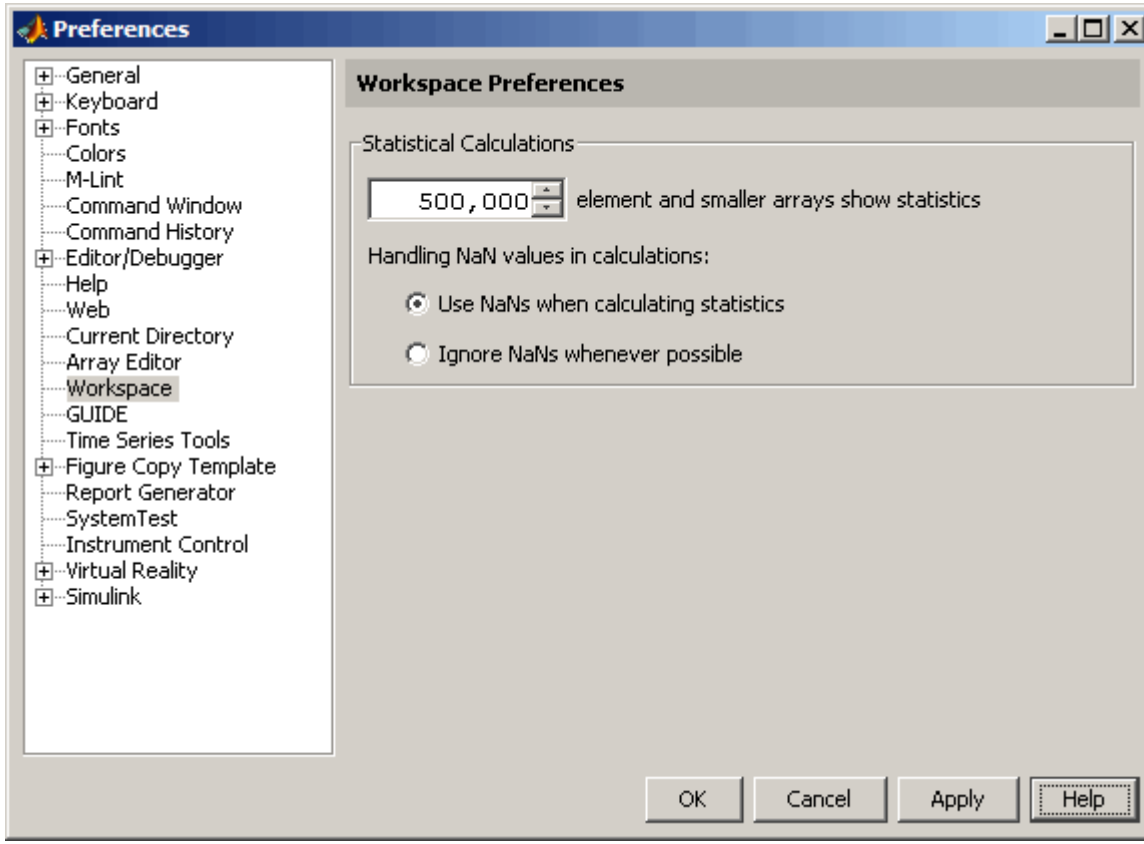
Opening Variables and Objects for Viewing and Editing

In the Workspace browser, double-click a variable and it opens in the Array Editor, where you can view and edit the contents of the variable. See "Viewing and Editing Workspace Variables with the Array Editor" on page 5-12 for more information about opening arrays.

Some toolboxes allow you to double-click an object in the Workspace browser to open a viewer or other tool appropriate for that object. For details, see the toolbox documentation for that object type.

Preferences for the Workspace Browser

The Workspace browser displays statistical calculations for variables. Use preferences to restrict the size of arrays on which calculations are performed and to specify if NaNs are included or ignored in calculations. Select **File > Preferences** to open the dialog box. Make changes and click **OK**.



Specify Maximum Array Size on Which to Compute Statistics

If you show statistical columns in the Workspace browser, and if you work with very large arrays, you might experience performance issues when the data changes as MATLAB updates the statistical results. In that event, show only the columns of interest to you and hide those you do not need.

Another step you can take is specify via a preference that the Workspace browser *not* perform statistical calculations on the largest arrays. Use the arrows to change the value of the maximum array size for which you want the Workspace browser to perform statistical calculations. The default value is `500,000` elements. Any variable exceeding that size reports `<Too many elements>` instead of statistical results.

Handling NaN Values in Calculations

If your data includes NaNs, you can specify that the statistical calculations consider the NaNs, or ignore the NaNs. For example, if a variable includes a `NaN` and the preference is set to **Use NaNs when calculating statistics**, the values for **Min**, **Max**, **Var** and some others will appear as `NaN`, although **Mode**, for example, shows a numeric result. With the preference set to **Ignore NaNs whenever possible**, numeric results appear for most of the statistical columns including **Min** and **Max**, however, **Var** is still reported as `NaN`.

For more information about statistical values in the Workspace browser, see “Viewing and Editing Values in the Current Workspace” on page 5-3.

Viewing and Editing Workspace Variables with the Array Editor

In this section...

About the Array Editor

You use the Array Editor to view and edit a visual representation of one or two-dimensional numeric arrays, strings, cell arrays of strings, and structures. You can also view the contents of multidimensional arrays. If you have an active Internet connection, watch the Array Editor video demo for an overview of the major functionality.

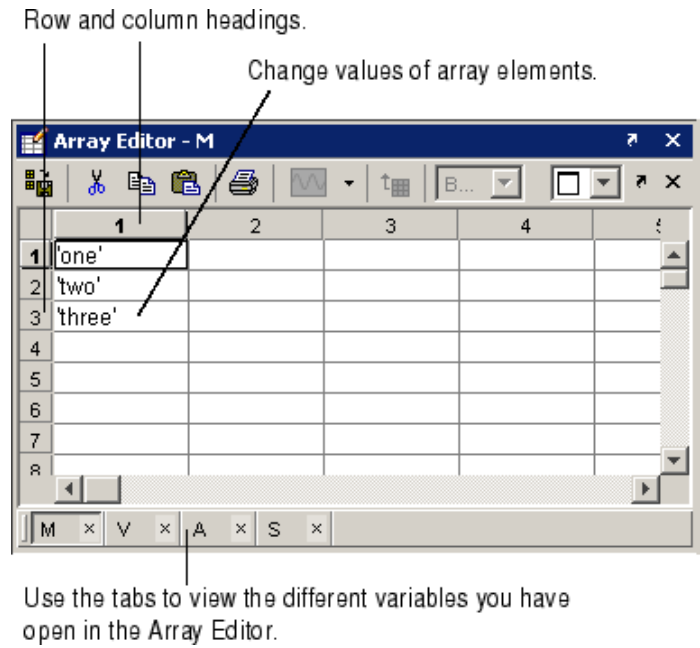
Opening the Array Editor

To open the Array Editor from the Workspace browser,

1 In the Workspace browser, select the variable you want to open. **Shift**+click or **Ctrl**+click to select multiple variables, or use **Ctrl+A** to select all variables to open.

2 Click the Open Selection button on the toolbar. For one variable, you can also open it by double-clicking it.

The Array Editor opens, displaying the values for the selected variable.



Repeat the steps to open additional variables in the Array Editor. Access each variable via its tab at the bottom of the window, or use the **Window** menu.

Changes you make to variables via the Command Window or other operations automatically update the information for those variables in the Array Editor.

Note The maximum array size that you can open in the Array Editor is not limited by MATLAB, but is based on your operating system or the amount of physical memory installed on your system.

Function Alternatives

To open a variable in the Array Editor, use `openvar` with the name of the variable you want to open as the argument. For example, type

```
openvar('M')
```

MATLAB opens `M` in the Array Editor.

To see the contents of a variable in the workspace, just type the variable name at the Command Window prompt. For example, type

```
M
```

and MATLAB returns

```
M =
'one'
'two'
'three'
```

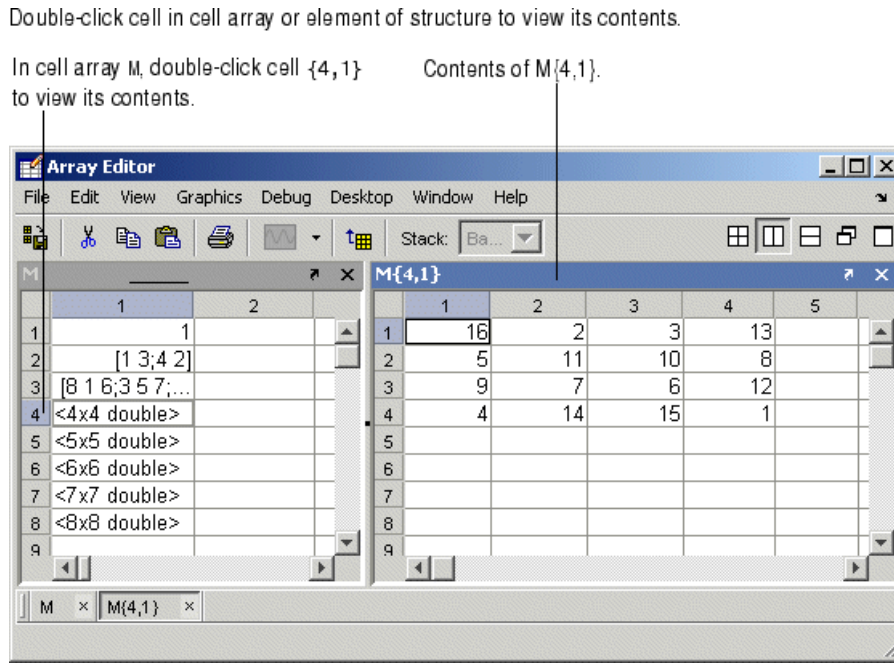
Viewing and Editing Cell Arrays, Structures, and Multidimensional Arrays

Cell Arrays and Structures in the Array Editor

You can view and edit the content of cell arrays and structures in the Array Editor.

In the Array Editor, double-click an element of a structure to open it as its own Array Editor document. You can then view and edit the contents of that element.

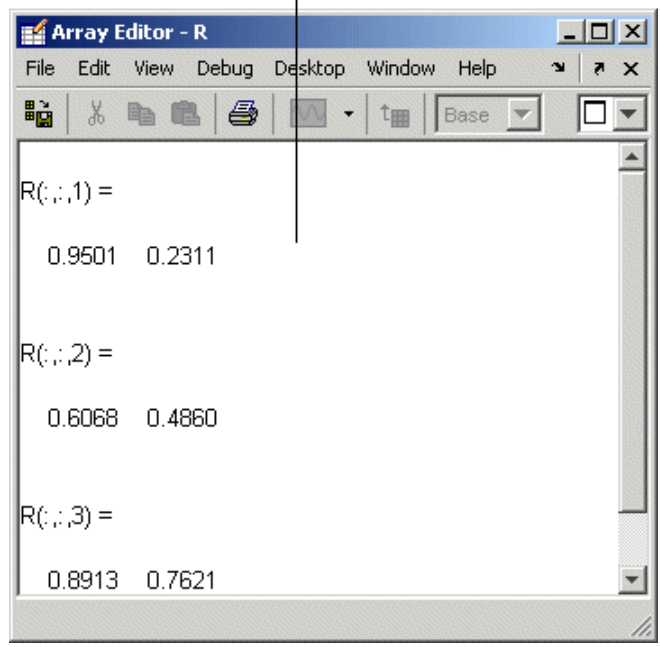
Similarly, double-click a cell in a cell array to view and edit its contents. The following illustration shows an 8-by-1 cell array, `M`, and the contents of `M{4,1}`.



Multidimensional Arrays in the Array Editor

You can view the contents of multidimensional arrays in the Array Editor. When you open a multidimensional array in the Array Editor, it does not have usual grid structure, because multidimensional arrays do not fit that format. You cannot double-click an element in a multidimensional array to edit it. The following illustration shows `R = rand(1,2,3)`.

You can view but cannot edit the contents of a multidimensional array in the Array Editor.



Navigating and Editing Shortcut Keys for the Array Editor

Use the following shortcut keys (sometimes called hot keys) to move among elements in the Array Editor. Navigating in the Array Editor is much like navigating in Microsoft Excel.

Key	Result
Enter	Commit any changes to the element and move to next element, where next element is specified using “Preferences for the Array Editor” on page 5-22 (default is down)

Key	Result
Tab	Move right Within a selection, also moves from the last column to the first column in the next row
Shift+Enter or **Shift+Tab**	Move in opposite direction of **Enter** or **Tab**
Page Up	Move up m rows, where m is the number of visible rows
Page Down	Move down m rows, where m is the number of visible rows
Home	Move to column 1
Ctrl+Home	Move to row 1, column 1
Shift+Home	Select to column 1
End	Move to last column in current row
F2 (**Ctrl+U** on Macintosh)	Edit current element, positioning cursor at the end of the element

Changing Array Size, Content, and Format of Elements in the Array Editor

To increase the size of an array, scroll to the desired location in the array and enter a value. The array will automatically expand to accommodate the new value. Empty cells are filled with zeros, if numeric, or empty arrays, if a cell array. To decrease the size of an array, select the rows or columns that you want to remove by clicking in the row or column header to select the entire row, right-clicking, and selecting **Delete**.

To change the value of an element in the Array Editor, click in that element and type a new value. Press **Enter** or **Return**, or click in another element to make the change take effect. You can specify where the cursor moves to after you press **Enter** — see “Preferences for the Array Editor” on page 5-22.

If you want to change the display format for the Array Editor, select the **View** menu and choose a format. To change the default format for future use, use

the **Preferences** dialog. For more information, see “Preferences for the Array Editor” on page 5-22.

If you opened an existing MAT-file and made changes to it using the Array Editor, save that MAT-file if you want the changes to be saved. For instructions, see “Saving the Current Workspace” on page 5-5.

Cut, Copy, Paste, and Clear Contents in the Array Editor

You can cut or copy selected elements, rows, and columns in an array and paste them to another position in that or another open array. To select a column or row, click in the row or column heading (the element that shows the row or column number). **Shift**+click to choose contiguous elements, rows, or columns in the array, or **Ctrl+A** to select all elements. For the cut, copy, and paste operations, use the **Edit** menu, the context menu, or the toolbar buttons. You can undo the last operation you performed in the Array Editor.

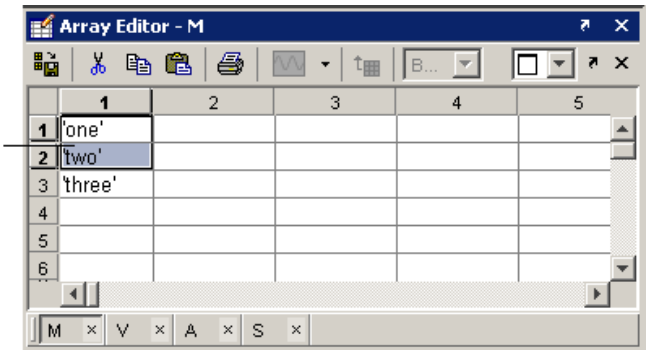
When you cut elements, the value of each element you cut becomes `0` if numeric or `[]` if a cell array. After cutting, select the elements whose value you want to replace with the cut elements and then choose **Paste**. If the shape of the elements you cut differs from the shape of the elements into which you are pasting, the Array Editor pastes all the elements, either by expanding the selection to be pasted into, or by expanding the array size to allow all the elements to be pasted. Pasting copied elements is the same as pasting cut elements, but the elements copied maintain their value rather than becoming `0`.

Select elements, rows, or columns and then select **Edit > Clear Contents**. The value of the selected elements becomes `0`. It differs from a performing a **Cut** because the data from the selected elements does not move to the clipboard; any clipboard content is unaffected by **Clear Contents**.

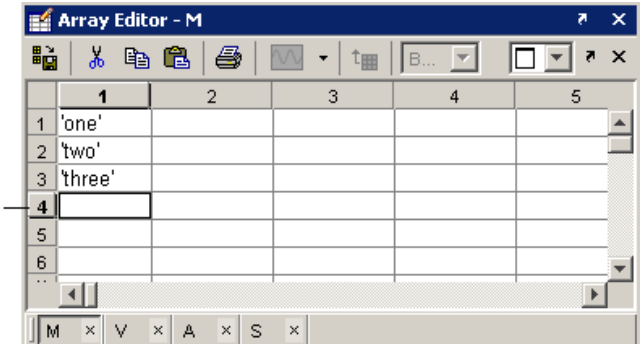
Example Copying and Pasting Array Elements

In this example, two elements are copied but the selected area for pasting is only one element, so the Array Editor expands the selected area for pasting.

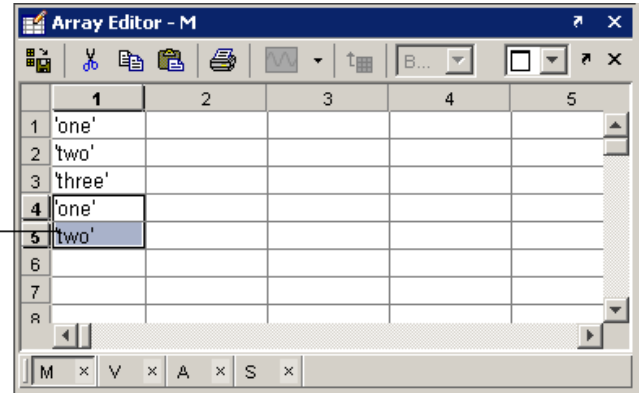
Two elements are selected and copied.



One element is selected as the paste area.

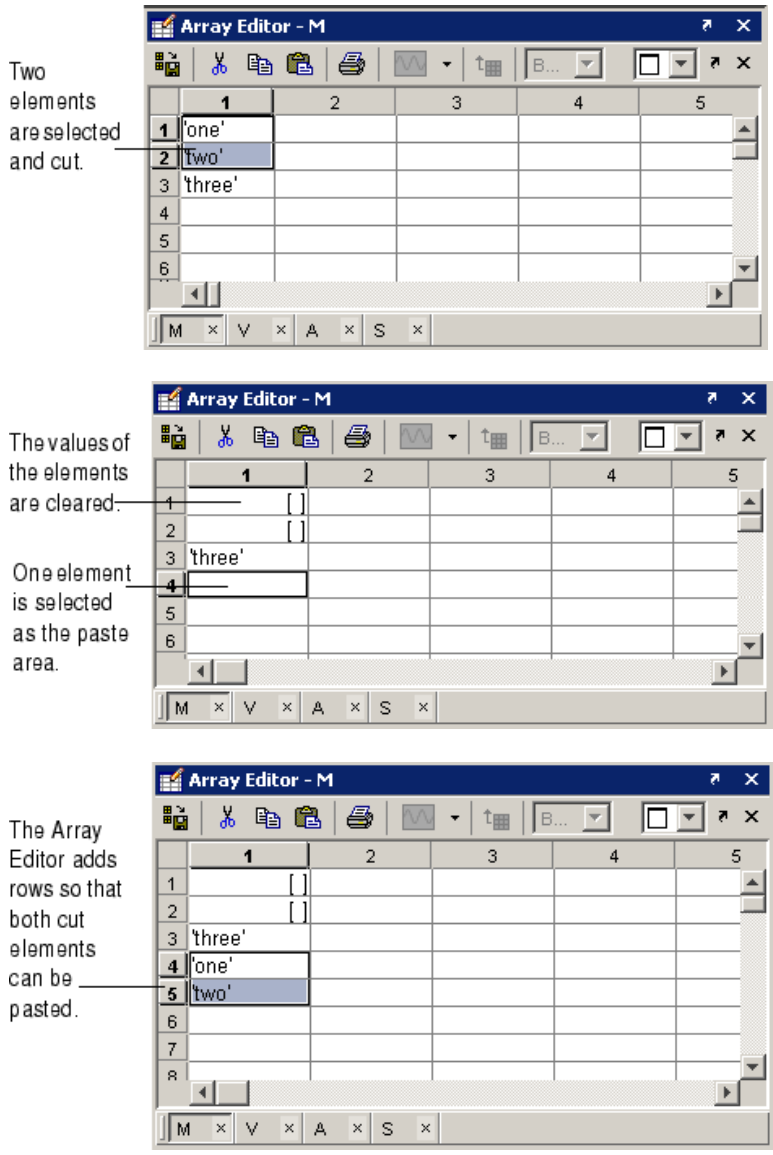


The Array Editor pastes all of the copied elements.



Example Cutting and Pasting Array Elements

In this example, the area selected for pasting requires the Array Editor to expand the array size in order for all cut elements to be pasted.



Insert and Delete in the Array Editor

You can insert and delete elements, rows, and columns in the Array Editor. When you select **Edit > Insert**, or **Edit > Delete**, a dialog box appears in which you specify cells, rows, or columns, and for cells, the direction for shifting.

Undo and Redo in the Array Editor

You can undo the last action you performed in the Array Editor, or redo a change after choosing undo. Select **Edit > Undo** or **Edit > Redo**. The actions supported are a change to a value you make by editing it in the Array Editor, cutting, pasting, inserting, deleting, clearing contents, and pasting Excel data.

Exchanging Data with the Command Window

You can copy data from the Array Editor and paste it into the Command Window. You can also copy a value from the Command Window and paste it into an element in the Array Editor. Be sure the data types are compatible. For example, you cannot paste text from the Command Window into a numeric array in the Array Editor.

Exchanging Data with Excel

You can cut or copy cells from Microsoft Excel and paste them into the Array Editor—use **Edit > Paste from Excel**. You can also cut or copy elements from the Array Editor and paste them into Excel.

Be sure the data types are compatible. For example, you cannot paste text from Excel into a numeric array in the Array Editor.

Creating Graphs and Variables from the Current Selection

You can create graphs and new variables from the Array Editor. To create a graph, select a cell, row, or column, and in the right-click context menu, choose the graph type. MATLAB presents allowable options for the selected data. In some cases, MATLAB makes assumptions, such as using `cell2mat` to convert selected cell array data, which cannot be plotted directly.

To create a new variable, select a cell, row, or column in the Array Editor, right-click, and from the context menu, select **Create Variable from Selection**.

Preferences for the Array Editor

To set preferences for the Array Editor, select **File > Preferences**. The Preferences dialog box opens showing **Array Editor Preferences**.

Format

Specify the default array output format of numeric values displayed in the Array Editor. This affects only how numbers are displayed, not how MATLAB computes or saves them. For more information, see the reference page for `format`.

Editing

You can specify where the cursor moves to after you type in an element and press **Enter**:

- If you want the cursor to remain at the element where you just typed, clear the **Move selection after Enter** check box.
- If you want the cursor to move to another element, select the **Move selection after Enter** check box, and then use **Direction** to specify how you want the cursor to move. For example, if you want the cursor to move right one element after you press **Enter**, select **Right**.

International Number Handling

You can specify how you want decimal numbers to be formatted when you cut or copy cells from the Array Editor and paste them into text files or other applications. The **Decimal separator to use when copying** edit field is by default `"."` (period). If you are working in or providing data to a locale that uses a different character to delimit decimals, type that character in this preference and click **OK** or **Apply**.

Search Path

In this section...

About the Search Path

MATLAB uses a *search path* to find M-files and other MATLAB related files, which are organized in directories on your file system. By default, the files supplied with MATLAB and MathWorks products are included in the search path. This default search path includes many of the directories and files under *`matlabroot`*`/toolbox`, where `matlabroot` is the directory in which MATLAB was installed, as returned by running the `matlabroot` function.

Any file you want to run or debug in MATLAB must reside in a directory that is on the search path, or in the current directory. If you create any MATLAB related files, add the directories containing the files to the MATLAB search path.

If you try to run or debug a file that is in a directory not on the path, the action fails. When this occurs in the Command Window, MATLAB errors. In the Editor/Debugger, a dialog box prompts you to

- **Change Directory** — Makes the directory containing the file you want to run become the current directory
- **Add to Path** — Adds the directory containing the file to the top of the search path

If there is already another file of the same name on the search path, the file you want to run is considered shadowed. You might need to perform some

action so that MATLAB runs the file you want—for more information, see "How the Search Path Determines Which Function to Use" on page 5-24.

The search path is also referred to as the *MATLAB path*. Directories included are considered to be *on the path*. When you include a directory in the search path, you *add it to the path*. Subdirectories must be explicitly added to the path; they are not on the path just because their parent directories are. Adding directories to the path is similar to performing an include or import in some other applications.

For instructions to view the search path and add directories to it, see "Viewing and Setting the Search Path" on page 5-26, including "Caution Against Saving Files in matlabroot/toolbox" on page 5-31.

How the Search Path Determines Which Function to Use

The order of directories on the path is relevant. MATLAB looks for a named element, for example, `foo`, as described here. If you enter `foo` at the MATLAB prompt, MATLAB performs the following actions:

1. Looks for `foo` as a variable.
2. Looks in the current directory for a file named `foo.m`.
3. Searches the directories on the MATLAB search path, in order, for `foo` as a built-in function, followed by `foo.m` which is not built-in.

If there is more than one function with the same name, the order of directories on the path determines which of those functions MATLAB uses. When MATLAB looks for that function, it uses the first one found in the search path:

- To use a function with the same name that is located in a directory further down on the search path, called a *shadowed* function, make its location the current directory. For M-file scripts, you can use `run` with the full pathname for the M-file. For example, use `run d:/mymfiles/foo.m` to ensure that version of `foo` runs. Another option is to move the directory containing the shadowed function to a position in the search path ahead of the directory containing the file of the same name. For example, add it to the top of the search path.

- If you are not sure of the function MATLAB is using, run `which` for a specified function and MATLAB returns the full path to the function.

Although the actual search path rules are more complicated because of the restricted scope of private functions, subfunctions, object-oriented functions, P-files, and MAT-files, this simplified perspective is accurate for the ordinary M-files you usually work with. For more information, see "Determining Which Function Is Called" in the MATLAB Programming documentation.

How MATLAB Finds the Search Path, pathdef.m

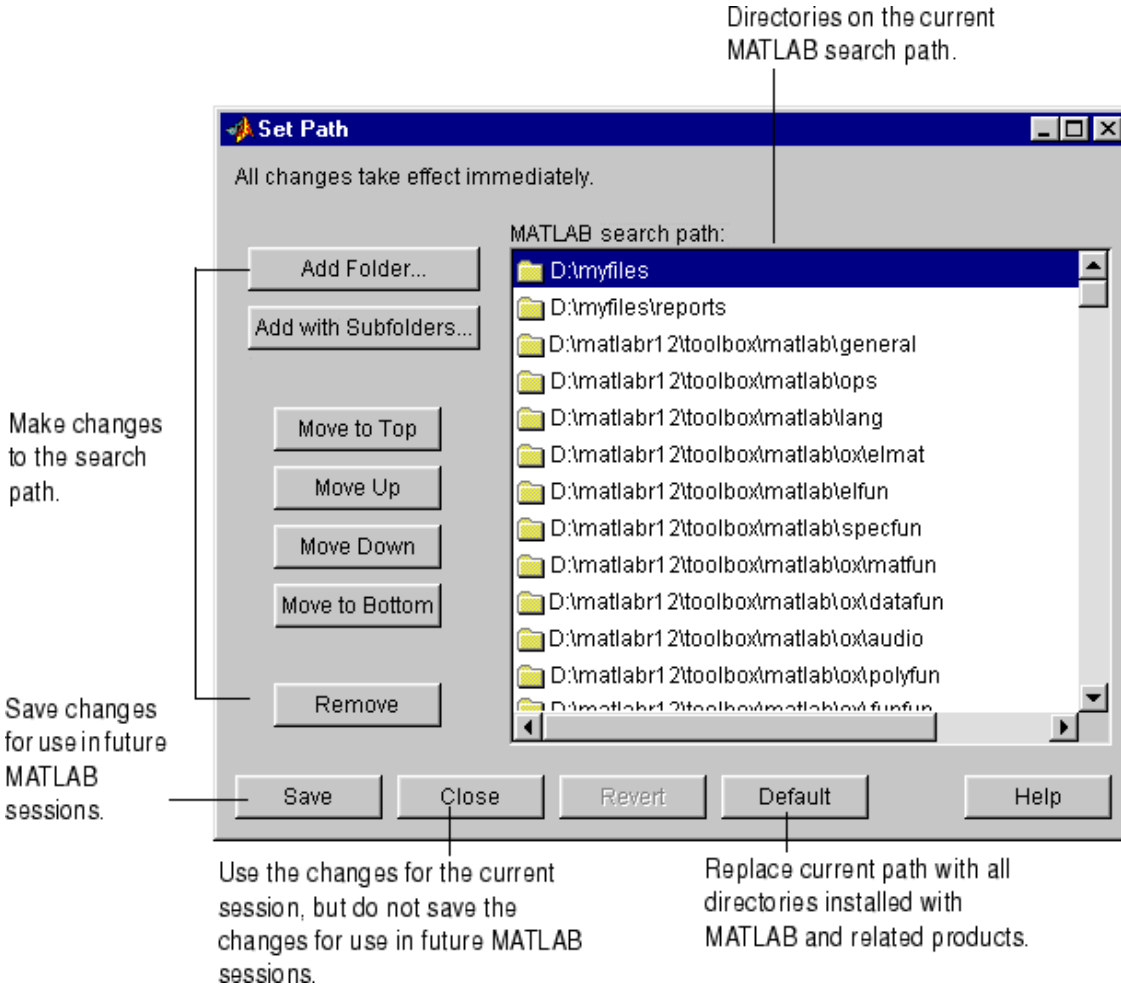
The search path is stored in the file `pathdef.m`, which by default, is located in *`matlabroot`*`/toolbox/local`. You can store it in the MATLAB startup directory, and modify it for the current session or for all future sessions.

When MATLAB starts, it looks for a `pathdef.m` file in its startup directory. If none is found, it uses `pathdef.m` in `matlabroot/toolbox/local`. MATLAB modifies the path based on any path statements in a `startup.m` file. During a session, you can save changes to the path using the **Set Path** dialog box or the `savepath` function, and MATLAB uses the path you saved to for the remainder of the session.

If MATLAB finds a `pathdef.m` in the current directory, it uses that version instead. To avoid problems, do not maintain a `pathdef.m` file in a directory other than the MATLAB startup directory or *`matlabroot`*`/toolbox/local`.

Viewing and Setting the Search Path

Use the **Set Path** dialog box to view and modify the MATLAB search path. Equivalent functions are documented for each feature of the **Set Path** dialog box. Select **Set Path** from the **File** menu, or type `pathtool` at the Command Window prompt. The **Set Path** dialog box opens.



Use the **Set Path** dialog box for the following actions. Equivalent functions are listed as well:

See also

Viewing the Search Path

The **MATLAB search path** field in the **Set Path** dialog box lists all of the directories on the search path. The top of the list is the start of the search path, while the bottom of the list is the end.

Function Alternative. Use the `path` function to view the search path.

Adding Directories to the Search Path

Add directories to the search path when you want to run M-files in those directories.

To add directories to the MATLAB search path using the **Set Path** dialog box,

1. Click **Add Folder** or **Add with Subfolders**.

   - If you want to add only the selected directory but do not want to add all of its subdirectories, click **Add Folder**.
   - If you want to add the selected directory and all of its subdirectories, click **Add with Subfolders**.

The **Browse for Folder** dialog box opens.

2 In the Browse for Folder dialog box, use the view of your file system to select the directory to add, and then click **OK**.

The selected directory, and subdirectories if specified in step 1, are added to the top of the search path.

3 To use the newly modified search path in future sessions, click **Save**. For more information about saving the path, see "Saving Settings to the Path" on page 5-30.

4 Click **Close**. If you did not save the changes in the previous step, the directories you added remain on the search path until you end the current MATLAB session.

You cannot add method directories (directories that start with @) and private directories to the MATLAB search path.

Adding Directories to the Path from the Current Directory Browser. In the Current Directory browser, select a directory, right-click, and select **Add to Path** from the context menu. Then select one of the submenus, for example, **Selected Folder and Subfolders**.

Function Alternative. To add directories to the top or the end of the search path, use `addpath`. The `addpath` function offers an option to get the path as a string and to concatenate multiple strings to form a new path.

You can include `addpath` statements in your `startup` M-file to automatically modify the path when MATLAB starts. For details, see "Modifying the Path in a startup.m File" on page 5-31.

Moving Directories Within the Search Path

The order of files on the search path is relevant — for more information, see "How the Search Path Determines Which Function to Use" on page 5-24.

To modify the order of directories within the search path,

1 Select the directory or directories you want to move.

2 Click one of the **Move** buttons, such as **Move to Top**. The order of the directories changes.

3 To use the newly modified search path in future sessions, click **Save**. For more information about saving the path, see "Saving Settings to the Path" on page 5-30.

4 Click **Close**. If you did not save the changes in the previous step, the new order of files on the search path remains in effect until you end the current MATLAB session.

Function Alternative. While there is not a specific function to move directories, you can edit the `pathdef.m` file with any text editor to change the order of the directories. Use caution when editing the file so that you do not make MATLAB and toolbox functions unusable.

Removing Directories from the Search Path

To remove directories from the MATLAB search path using the **Set Path** dialog box,

1 Select the directories to remove.

2 Click **Remove**. The directories are removed from the path.

3 To use the newly modified search path in future sessions, click **Save**. For more information about saving the path, see "Saving Settings to the Path" on page 5-30.

4 Click **Close**. If you did not save the changes in the previous step, the directories are removed from the search path until you end the current MATLAB session.

Function Alternative. To remove directories from the search path, use `rmpath`.

You can include `rmpath` statements in your `startup` M-file to automatically modify the path when MATLAB starts. For details see "Modifying the Path in a startup.m File" on page 5-31.

Restoring the Default Search Path

To restore the default search path, click **Default** in the **Set Path** dialog box. This changes the search path so that it includes only the directories installed with MATLAB and related products.

Reverting to the Previous Path

To restore the previous path, click **Revert** in the **Set Path** dialog box. This cancels any unsaved changes you have made in the **Set Path** dialog box.

Saving Settings to the Path

When you make changes to the search path, they remain in effect during the current MATLAB session. To keep the changes in effect for subsequent sessions, you need to save them. To save changes using the **Set Path** dialog box, click **Save**.

If you want to automatically use this search path in future sessions, save the path to your MATLAB startup directory, which saves `pathdef.m` to that location. You can save the changes to the default `pathdef.m` file, in *`matlabroot`*`/toolbox/local` if you have write permission for that directory but see the following caution. Alternatively, you can include `addpath` and `rmpath` statements in a `startup.m` file, which avoids some problems you might have with saving the path, for example, using the same path with both Windows and UNIX platforms. For more information, see "Using the Path in Future Sessions" on page 5-31.

Note When MATLAB starts up, it may provide in the Command Window a list of invalid directories. These directories were previously in `pathdef.m` but were subsequently deleted. The directories do not now appear on the search path listing in the **Set Path** dialog box. Click **Save** to overwrite the `pathdef` M-file, thereby eliminating future reporting of these nonexistent directories.

Caution Against Saving Files in matlabroot/toolbox. Save any M-files you create and any MathWorks supplied M-files that you edit in a directory that is not in the *matlabroot*/toolbox directory tree. If you keep your files in *matlabroot*/toolbox directories, they can be overwritten when you install a new version of MATLAB. Also note that locations of files in the *matlabroot*/toolbox directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to *matlabroot*/toolbox directories using an external editor or add or remove in from these directories using file system operations, run rehash toolbox before you use the files in the current session. If you make changes to existing files in *matlabroot*/toolbox directories using an external editor, run clear functionname before you use the files in the current session. For more information, see rehash or "Toolbox Path Caching in MATLAB" on page 1-17.

Function Alternative. Use savepath to save the current path to pathdef.m. Locate pathdef.m in your MATLAB startup directory to automatically use it in future sessions. Consider using savepath in your finish.m file. To modify the default path upon startup, include addpath and rmpath functions in your startup.m file. For more information, see "Modifying the Path in a startup.m File" on page 5-31.

Using the Path in Future Sessions

There are three basic ways for MATLAB to automatically use a search path you specify, each with advantages and disadvantages:

- "Modifying the Path in a startup.m File" on page 5-31
- "Saving the Path in the MATLAB Startup Directory" on page 5-32
- "Saving the Path in matlabroot/toolbox/local" on page 5-32

For background information, see "How MATLAB Finds the Search Path, pathdef.m" on page 5-25.

Modifying the Path in a startup.m File

Put addpath and rmpath statements in a startup.m file, and include the startup file in MATLAB's startup directory. When MATLAB starts, it uses the search path defined in pathdef.m in *matlabroot*/toolbox/local and modifies it based on the commands in the startup.m file.

By maintaining an unaltered `pathdef.m` in *`matlabroot`*`/toolbox/local`, you avoid inadvertently removing directories supplied by The MathWorks from the path. This method continues working even when you update to a new version of MATLAB. If you run MATLAB on both Windows and UNIX platforms, this method works well — for example, for each platform, include separate `addpath` sections in the `startup.m` file, with each section preceded by an `ispc` or `isunix` statement.

One disadvantage of this method is that changes you make to the path using the **Set Path** dialog box are not incorporated in the `startup.m` file.

Saving the Path in the MATLAB Startup Directory

Copy `pathdef.m` from *`matlabroot`*`/toolbox/local` to the MATLAB startup directory. Make changes to the path using the **Set Path** dialog box, and with `addpath` and `rmpath` functions — choose whichever suits your needs. You can use this method if you do not have write access to *`matlabroot`*`/toolbox/local`.

There are some disadvantages to this method. You might inadvertently remove directories supplied by The MathWorks from the path. When you update to a new version of MATLAB, you cannot use the `pathdef.m` file in the startup directory, but must delete it and create a new version. If you run MATLAB on both Windows and UNIX platforms, you need to maintain a separate `pathdef.m` file for each.

Saving the Path in matlabroot/toolbox/local

If you have write access to *`matlabroot`*`/toolbox/local`, make and save changes to the path using the **Set Path** dialog box, and with `addpath` and `rmpath` functions — choose whichever suits your needs.

There are some disadvantages to this method. You cannot maintain this file when you update to a new version of MATLAB, but will need to use the new default `pathdef.m` and make changes to it. If you run MATLAB on both Windows and UNIX platforms, you need to maintain a separate `pathdef.m` file for each.

Recovering from Problems with the Search Path

If you get unexpected results that are related to the search path, you can try to correct the path file or restore the default path. You might experience path problems if you save the path on a Windows platform and then try to use the same `pathdef.m` file on a UNIX platform. Similarly, you might experience problems if you edit the `pathdef.m` file directly and make it invalid, or if the file becomes corrupt, renamed, or lost.

For example, if an error message similar to the following appears when you start MATLAB

```
Warning: MATLAB did not appear to successfully set the search
path...
```

it indicates a problem with the search path and you will not be able to use MATLAB successfully.

To recover from problems with the search path, try the following, in order, proceeding to the next step only if needed:

1. View the `pathdef.m` and `startup.m` files, looking for obvious problems. Make changes and save them. If path problems appear to be resolved, start MATLAB again to be sure the problem does not reappear. Depending on the problem, you might not be able to even view the `pathdef.m` file.

2. Use the default path for MathWorks products. In the **Set Path** dialog box, select **Default**, then **Save**, then **Close**. Depending on the problem, you might not be able to even open the dialog box.

3. Run `restoredefaultpath`. This sets the search path to include only installed products from The MathWorks. If that seems to have corrected the problem, run `savepath`. Start MATLAB again to be sure the problem does not reappear.

   Depending on the problem, this might generate a message such as

   ```
   The path may be bad. Please save your work (if desired), and quit.
   ```

   If so, perform step 4.

4. Perform these steps after trying step 3.

a Run

```
restoredefaultpath; matlabrc
```

This might run for a few minutes. It sets the search path to include only installed products from The MathWorks and corrects path problems encountered during startup.

b If there is a `pathdef.m` in your startup directory for MATLAB, it caused the problem. So either remove the bad `pathdef.m` file or replace the with a good `pathdef.m` file, for example, one you can generate at this point with

```
savepath('path_to_your_startup_directory/pathdef.m')
```

c Start MATLAB again to be sure the problem does not reappear.

File Management Operations

In this section...

Note You generally cannot perform operations on files and directories for which you do not have proper permission. For example, you cannot copy a file to a read-only directory using the Current Directory browser, however, you can do so using `movefile` with the appropriate option.

About MATLAB File Operations

MATLAB file operations use the current directory and the MATLAB search path as reference points. Any file you want to run must either be in the current directory or on the search path. The key tools for performing file operations are

Current Directory Field

A quick way to view or change the current directory is by using the current directory field in the desktop toolbar.

To change the current directory from this field, do one of the following:

- In the field, type the path for the new current directory.
- Click the down arrow to view a list of previous working directories, and select an item from the list to make that directory become the MATLAB current working directory. The directories are listed in order, with the most recently used at the top of the list. You can clear the list and set the number of directories saved in the list — see “Preferences for the Current Directory Browser” on page 5-54.
- Click the Browse for Folder button (**...**) to set a new current directory.
- Use the Go Up One Level button to move the current directory up one level.

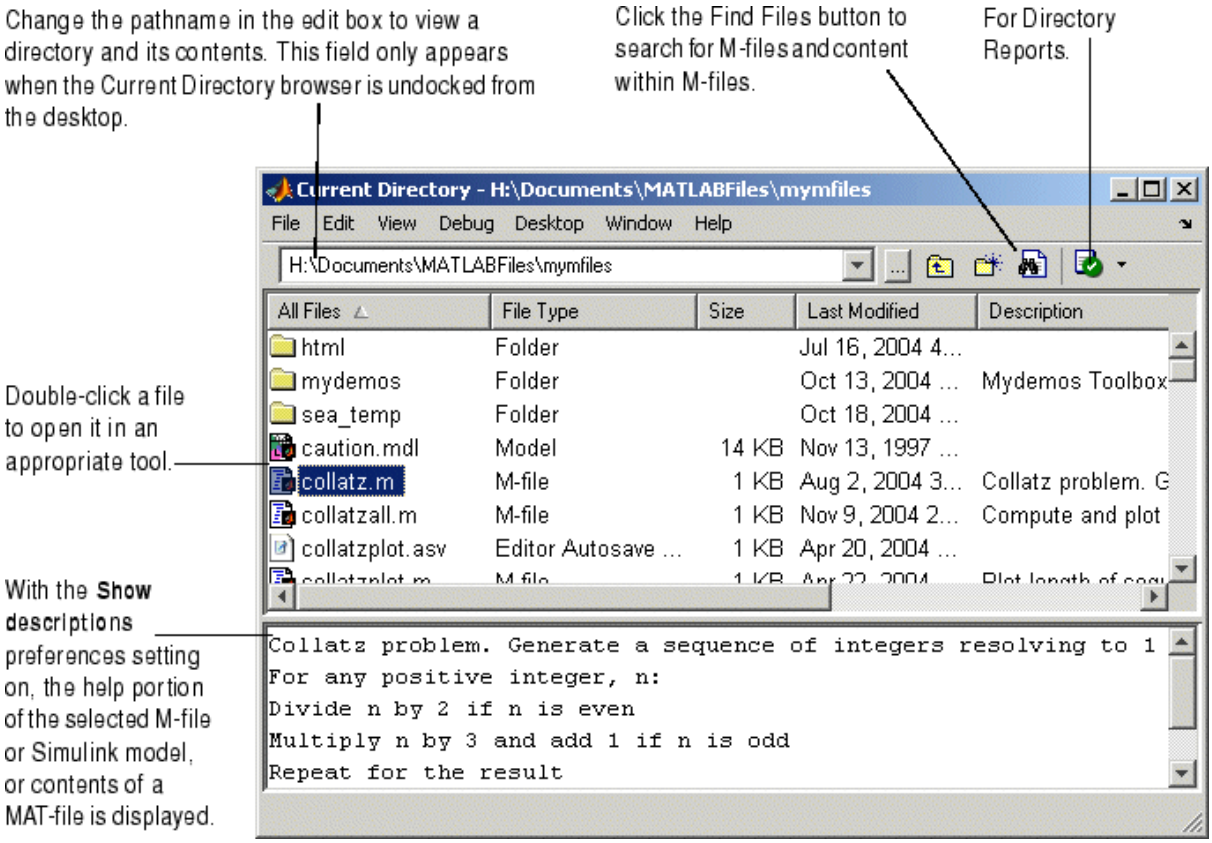
The current directory field in the desktop also appears in the Current Directory browser, when the Current Directory browser is undocked. Consider it to be one tool with two different means of accessing it.

Current Directory Browser

To search for, view, open, find, and make changes to MATLAB related directories and files, use the MATLAB Current Directory browser. Most features of the Current Directory browser have equivalent functions that perform similar actions. If you have an active Internet connection, you can watch the Current Directory Browser video demo for an overview of the major functionality.

In addition to the features described here, the Current Directory browser includes tools to help you manage your M-files — see “Directory Reports in Current Directory Browser” on page 7-2.

To open the Current Directory browser, select **Desktop > Current Directory** from the MATLAB desktop, or type `filebrowser` at the Command Window prompt. The Current Directory browser opens.



The main tasks you perform with the Current Directory browser are

- “Viewing and Making Changes to Directories” on page 5-38
- “Creating, Renaming, Copying, and Removing Directories and Files” on page 5-43
- “Opening and Running Files” on page 5-47
- “Finding Files and Content Within Files” on page 5-49
- “Accessing Source Control Features” on page 5-54
- Setting “Preferences for the Current Directory Browser” on page 5-54

Viewing and Making Changes to Directories

You can change the current directory, view its contents, add directories to the MATLAB search path, and change the way the Current Directory browser presents entries.

Changing the Current Working Directory and Viewing Its Contents

To change the current directory, use the current directory field. The Current Directory browser lists the files and directories in the current directory.

To view the contents of a subdirectory, double-click it, or select the subdirectory and press **Enter** or **Return**.

To move up one level in the directory structure, press the backspace (**<-**) key.

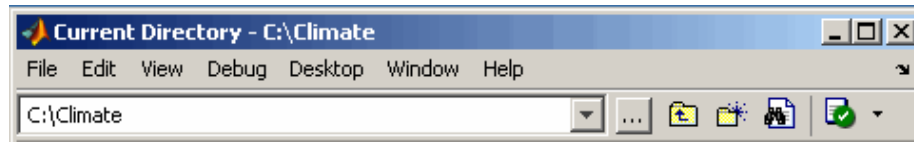
Function Alternative. Use `dir` to view the contents of the current working directory or another specified directory. Use a return argument with `dir` to get a structure containing information including the names of the files in the directory and their last modified date and time.

Use `what` with no arguments to display the MATLAB related files in the current working directory. Use `which` to display the pathname for a specified function. Use `exist` to see if a directory or file exists. Use `fileattrib` to see or set file attributes, much like `attrib` in DOS or `chmod` in UNIX.

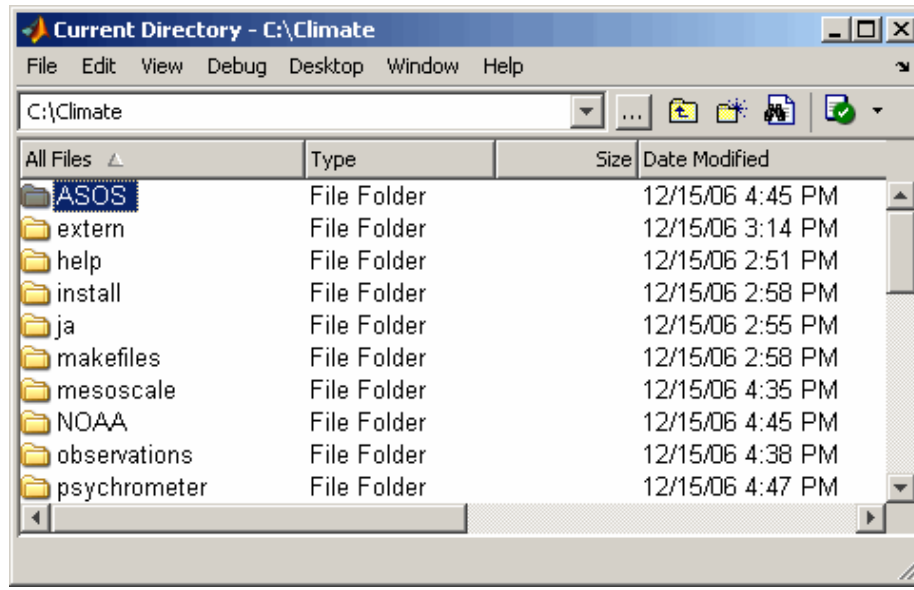
Searching in the Current Directory Browser

You can search the Current Directory browser for files and directories by typing directly in the window. As you type the Current Directory browser searches downward from the top of the window to find an entry that matches what you have typed. For example:

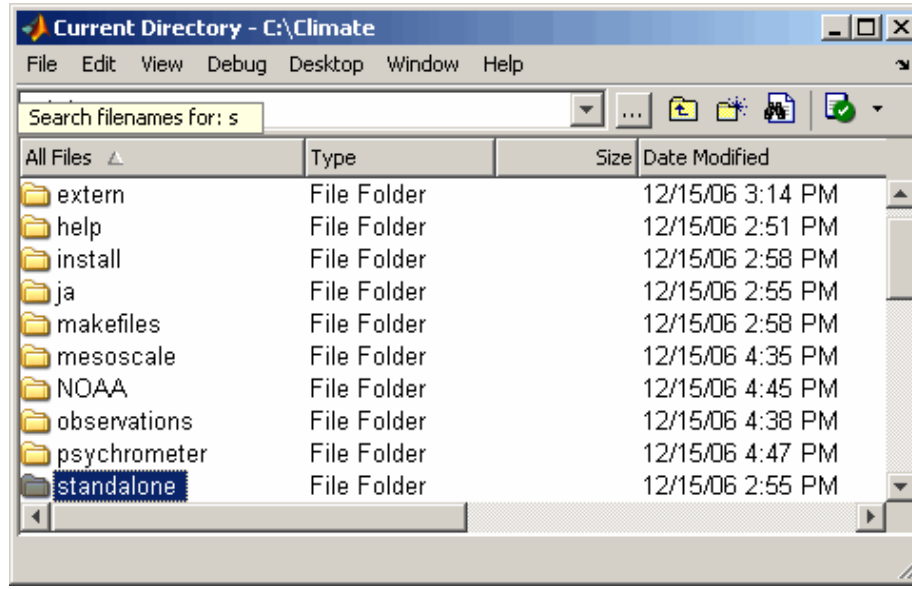
1 Set `C:\Climate` as your current directory.



2 Assume that you want to search the Current Directory for the file named `standard.m`. Begin your search by positioning the cursor anywhere within the Current Directory browser.

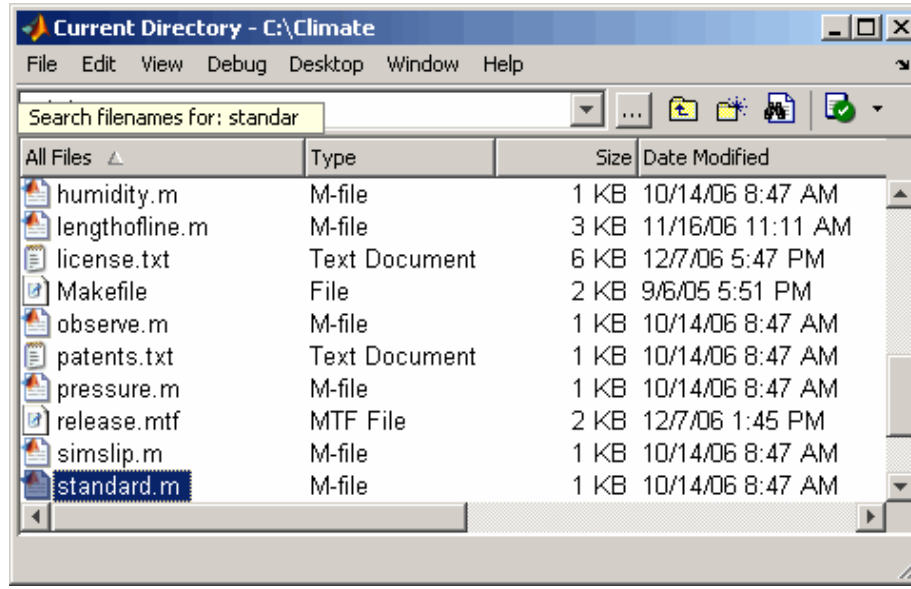


3 Type the letter `s`. The Current Directory browser searches to find the first entry beginning with the letter `s`. In this example it stops at the directory named `standalone`. Note as you begin typing that a small yellow-background `Search filenames for:` dialog box appears at the top of the Current Directory browser window. This dialog box keeps track of your search target as you type additional letters to narrow the focus of your search.



4 Since the `standalone` directory is not your intended search target, continue typing additional letters that identify your search target, eventually entering the letters `standar`.

5 When you have entered the letters `standar`, the search resumes, stopping this time at the `standard.m` file, your intended search target.



Changing the Display

Types of Files. To specify the types of files shown in the Current Directory browser, use the **View** menu. For example, you can show only M-files. If **All Files** is selected and you want to see specific file types, first clear the selection for **All Files** and then select the specific file types.

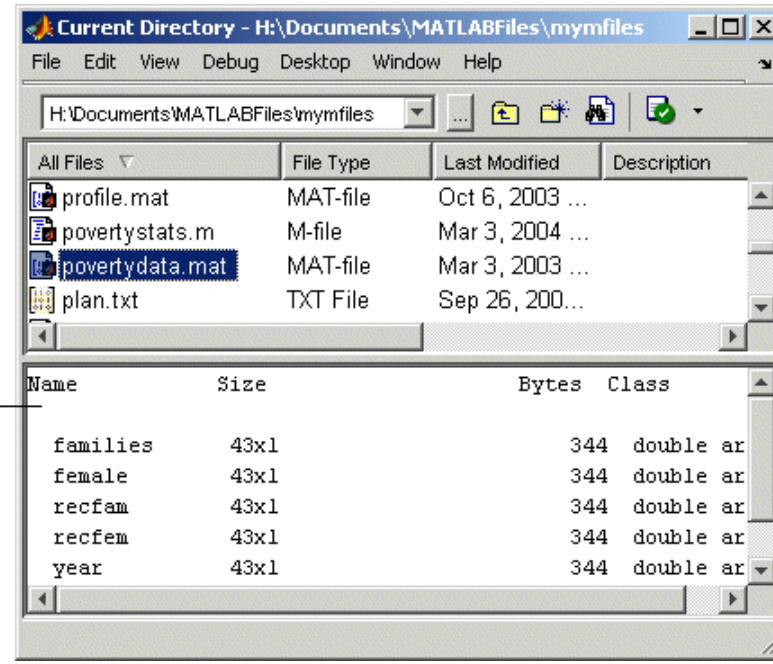
Columns. To show or hide columns, use preferences for the Current Directory browser. Select **File > Preferences > Current Directory** and select or clear the check boxes for **Browser display options**. For more information, see “Browser Display Options” on page 5-55.

You can sort the information shown in the Current Directory browser by column. Click the title of column on which you want to sort. The display is sorted, with the information in that column shown in ascending order, and an up arrow icon indicating the direction. Click a second time on the column title to sort the information in descending order.

Contents. In the Current Directory browser, select a file and then view information about the file in the Current Directory browser's lower pane. To view this, you must first select **File > Preferences > Current Directory** and under **Browser display options**, select the check box **Show M, MDL and MAT file contents**.

For an M-file, it shows the M-file help. For a Simulink model, it shows the complete description, allowing you to view information about a model without having to start Simulink. For a MAT-file, it displays the names of its variables along with their size, bytes, and class, allowing you to view the content of a MAT-file without loading it.

View variables in selected MAT-file without loading the file.



You can view more extensive help for the M-file selected in the Current Directory browser. From the context menu, select **View Help**. The reference page for that function appears in the Help browser.

Adding Directories to the MATLAB Search Path

From the Current Directory browser, you can add directories to the MATLAB search path. Right-click and from the context menu, select **Add to Path**. Then select one of the options:

- **Current Directory** — Adds the current directory to the path.
- **Selected Folders** — Adds the directories selected in the Current Directory browser to the path.
- **Selected Folder and Subfolders** — Adds the directory selected in the Current Directory browser to the path, and adds all of its subdirectories to the path.

Creating, Renaming, Copying, and Removing Directories and Files

General Notes

If you have write permission, you can create, copy, remove, and rename MATLAB related files and directories for the directory shown in the Current Directory browser. If you do not have write permission, you can still copy files and directories to another directory, or you can use equivalent functions, such as `movefile`.

To run functions whose arguments require the use of a pathname or filename, use the function form rather than the unquoted or command form of the syntax when the pathname or filename includes spaces. For example, the command form

```
delete my file.m
```

generates a warning and does not delete `my file.m`. Instead use the function form of the syntax:

```
delete('my file.m')
```

Creating New Files

To create a new file in the current directory,

1. Select **New** from the context menu or **File** menu and then select the type of file to create.

   An icon for that file type, for example, an M-file icon  , with the default name `Untitledn`, appears at the end of the list of files shown in the Current Directory browser.

2. Type over `Untitledn` with the name you want to give to the new file.

3. Press **Enter** or **Return**.

   The file is added.

4. To enter the contents of the new M-file, open the file—see "Opening and Running Files" on page 5-47. If you created the file using the context menu, the new file opens in the Editor/Debugger with a template for writing an M-file function.

Function Alternative. Use the `edit` function to create a new M-file or other type of text file in the Editor/Debugger.

Creating New Directories

To create a new directory in the current directory,

1. Click the New Folder button  in the Current Directory browser toolbar, or select **New > Folder** from the context menu.

   An icon, with the default name `NewFoldern` appears at the end of the list of files shown in the Current Directory browser.

2. Type over `NewFoldern` with the name you want to give to the new directory.

3 Press the **Enter** or **Return** key.

   The directory is added.

Function Alternative. To create a directory, use the `mkdir` function. For example,

```
mkdir newdir
```

creates the directory `newdir` within the current directory.

Renaming Files and Directories

To rename a file or directory, select the item, right-click, and select **Rename** from the context menu. Type over the existing name with the new name for the file or directory, and press **Enter** or **Return**. The file or directory is renamed.

Function Alternative. You can use `movefile` to rename a file or directory. For example,

```
movefile('myfile.m','projectresults.m')
```

renames `myfile.m` to `projectresults.m`.

Cutting or Deleting Files and Directories

To cut or delete files and directories,

1 Select the files and directories to remove. Use **Shift**+click or **Ctrl**+click to select multiple items.

2 Right-click and select **Cut** or **Delete** from the context menu.

   The files and directories are removed.

Files and directories you delete from the Current Directory browser go to the Recycle Bin on Windows (or the Trash Can on Macintosh platforms). If you do not want the selected items to go to the Recycle Bin, press **Shift+Delete**. A confirmation dialog box displays before the items are deleted if you have set that option in your operating system. For example, on Windows, right-click

the Recycle Bin, select **Properties** from the context menu, and then, under the **Global** tab, select the check box to **Display delete confirmation dialog**.

Function Alternative. To delete a file, use the `delete` function. For example,

```
delete('d:/mymfiles/testfun.m')
```

deletes the file `testfun.m`. You can recover deleted files if you use the `recycle` function or the equivalent preference described in "Default Behavior of the Delete Function" on page 2-78.

To delete a directory and optionally its contents, use `rmdir`. For example,

```
rmdir('myfiles')
```

removes the directory `myfiles` from the current directory.

Copying and Pasting Files and Directories

Use the Current Directory browser, to copy (or cut) and paste files and directories:

1. Select the files or directories to copy. Use **Shift**+click or **Ctrl**+click to select multiple items. For a directory, the entire contents are copied, including all subdirectories and files.
2. Right-click and select **Copy** from the context menu.
3. Navigate to the file or directory where you want to paste the items you just copied.
4. Right-click and select **Paste** from the context menu.

You can also copy and paste files and directories to and from tools outside of MATLAB, such as Windows Explorer. You can use Current Directory browser menu items or keyboard shortcuts, or you can drag the items.

Function Alternative. Use `movefile` or `copyfile` to cut and paste or to copy and paste files or directories. For example, to make a copy of the file `myfun.m` in the current directory, assigning it the name `myfun2.m`, type

```
copyfile('myfun.m','myfun2.m')
```

Opening and Running Files

Opening Files

You can open a file from the Current Directory browser and the file opens in the tool associated with that file type.

To open a file, select one or more files and perform one of the following actions:

- Press the **Enter** or **Return** key.
- Right-click and select **Open** from the context menu.
- Double-click the file(s).

The file opens in the appropriate tool, provided that the tool has been installed on your system. For example, the Editor/Debugger opens for M-files, and Simulink opens for model (`.mdl`) files.

To open a file in the Editor/Debugger, no matter what type it is, select **Open as Text** from the context menu. One exception is P-files (`.p`), which you cannot open.

To open a file using an external application, select **Open Outside MATLAB** from the context menu. For example, if you select `myfile.doc`, **Open Outside MATLAB** opens `myfile.doc` in Microsoft Word, assuming you have the `.doc` file association configured to start Word. This is also useful for file types associated with MATLAB that are also associated with an external application in Windows. For example, `.mat` is the extension for MATLAB data files as well as Microsoft Access files. When you double-click a `.mat` file in the Current Directory browser, it loads the MATLAB data file into the workspace. If instead you want to open the `.mat` file in Access, right-click it and select **Open Outside MATLAB**. MATLAB opens the file using the applications you associated with that file type in Windows. For more information, see "Changing File Associations for MATLAB from Windows" on page 1-5.

You can also import data from a file. Select the file, right-click, and select **Import Data** from the context menu. The Import Wizard opens. See the Import Wizard documentation for instructions to import the data.

You can run a Windows shortcut directly from the Current Directory browser. Double-click the shortcut icon in the Current Directory browser to perform the Windows operation.

Function Alternative. Use the `open` function to open a file in the tool appropriate for the file, given its file extension. Default behavior is provided for standard MATLAB file types. You can add other file types and override the default behavior for the standard files. For `name.ext`, `open` performs the following actions.

File Type	Extension	Action
Figure file	`fig`	Opens figure `name.fig` in a figure window.
HTML file	`html`	Opens HTML file `name.html` in the MATLAB Web browser.
M-file	`m`	Opens M-file `name.m` in the Editor/Debugger.
MAT-file	`mat`	Opens MAT-file `name.mat` in the Import Wizard.
Model	`mdl`	Opens model `name.mdl` in Simulink.
PDF file	`pdf`	Opens the PDF file `name.pdf` in the installed PDF reader, for example, Adobe Acrobat.
Project file	`prj`	Opens the project file `name.prj` in the MATLAB Compiler Deployment Tool. If the MATLAB Compiler or Deployment Tool is not installed, opens the project file in a text editor.

File Type	Extension	Action
Variable	none	Opens the numeric or string array `name` in the Array Editor; `open` calls `openvar`.
Other	`custom`	Opens `name.custom` by calling the helper function `opencustom`, where `opencustom` is a user-defined function.

Use `winopen` to open a file using an external application on Windows platforms.

To view the content of an ASCII file, such as an M-file, use the `type` function. For example

```
type('startup')
```

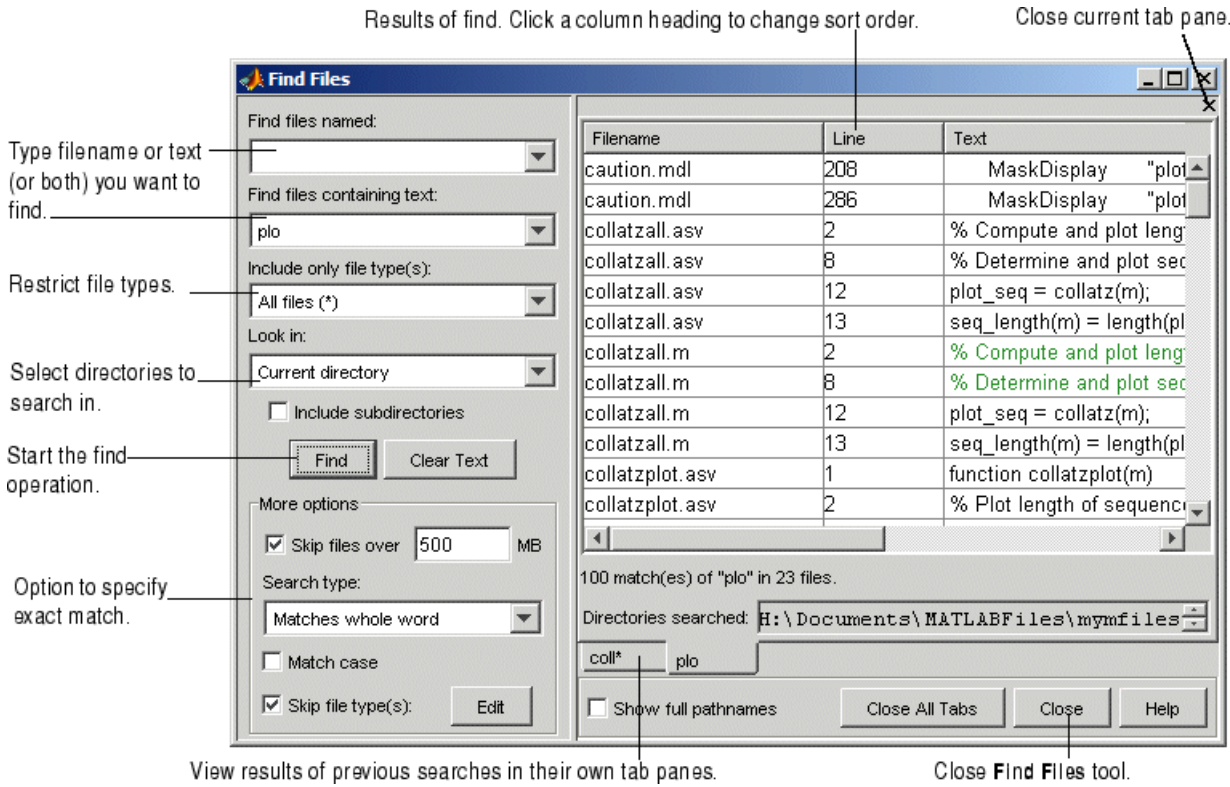
displays the contents of the file `startup.m` in the Command Window.

Running M-Files

To run an M-file from the Current Directory browser, select it, right-click, and select **Run** from the context menu. The results appear in the Command Window.

Finding Files and Content Within Files

Use the Find Files tool to search for files or for specified text within files.



To search for files in one or more directories, or to search for specified text in files, follow these instructions:

1 Open the **Find Files** tool by clicking the Find Files button in the Current Directory browser toolbar, or by selecting **Edit > Find Files** from any desktop tool, such as the Current Directory browser or the Editor/Debugger.

   The **Find Files** dialog box opens.

2 Type the filename and/or text you are searching for:

   - To search for files, type the filename in the **Find files named** field. You can use the wildcard character (`*`) in the filename. For example, type `coll*` to search for filenames that start with `coll`.

- To search for text within files, type the text in the **Find files containing text** field. For example, search for `plot`. Alternatively, you can select text in the Command Window or Editor/Debugger and that text appears in the **Find files containing text** field.

  Under **More options**, use the **Search type** to specify **Matches whole word**, or specify a partial match by selecting **Contains text**.

- To search for text in specified filenames only, type entries in both fields. Use the **Clear Text** button to clear the entries in both fields.

Click the down arrow next to each field to select previous entries from the current MATLAB session.

3 You can restrict the types of files to search by selecting an option in **Include only file type(s)**. For example, select `*.m` to limit the search to M-files only.

With **All files (*)** selected, use **Skip file types** (under **More options**) to ignore files of the specified type. For details, see "Skip File Types in Find Files" on page 5-52.

4 From the **Look in** list box, select the directories to search in. Select the MATLAB current directory or MATLAB search path, or use the `Browse` option to select another directory. You can instead type the full pathname for one or more directories into this field, with each pathname separated by a semicolon (`;`). To include subdirectories in the search, select the **Include subdirectories** check box.

5 Use additional entries under **More options** to further restrict the search:

- **Skip files over** the specified size to ignore large files that might take a long time to search through. This option is only available when you are searching for text within files.
- **Match case** when lower or upper case is relevant.

6 To execute the search, click **Find**. While the search is in progress, the **Find** button label changes to **Stop Find**. To abort a search, click **Stop Find**.

Search results appear in the pane on the right side of the **Find Files** dialog box, with a summary of the results at the bottom of the pane. For text searches, the line number and line of code are shown. To see the full pathnames for the files, select the **Show full pathnames** check box.

7 Click a column heading to sort the results based on that column. Click the column heading again to reverse the sort order for that column. For example, click **Line** to sort results by line number.

Opening Files from Find Files

To open files shown in the results list, do one of the following:

- Double-click the file
- Select the files and press **Enter** or **Return**
- Right-click the selected files and select **Open** from the context menu

The files open in the Editor/Debugger. For text searches, the file opens scrolled to the line number shown in the results section of the **Find Files** dialog box. Once in the Editor/Debugger, you can use the Find & Replace tool to change specified text.

Previous Results of Find Files

To see the results of a previous search, select its tab at the bottom of the results pane. **Find Files** shows up 10 search result tabs while the tool is open, but does not maintain the results after you close the tool.

MATLAB maintains the state for options in the Find Files tool even after you end the session.

Skip File Types in Find Files

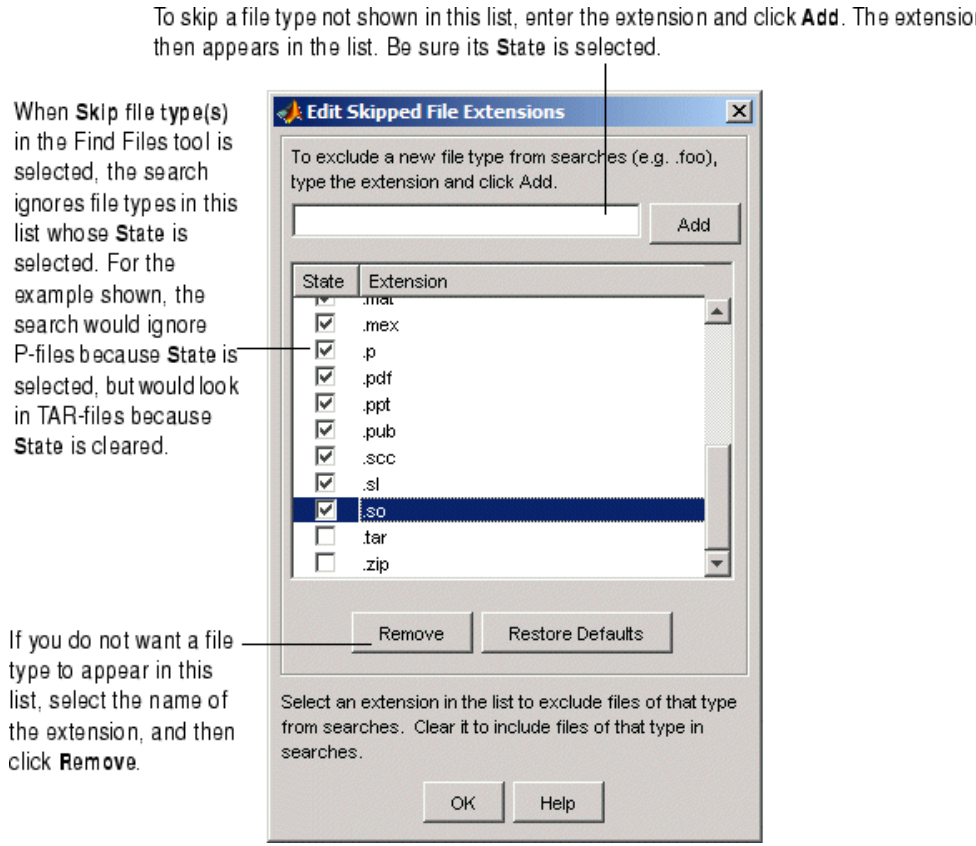
In the Find Files tool, you can restrict the search to look in all file types except those you specify:

1 For **Include only file type(s)**, select **All files (*)**.

2 Select the **Skip file type(s)** check box.

3 Click **Edit** to view or change the list of file types the search ignores.

The **Edit Skipped File Extensions** dialog box opens.



4 Find Files will not look in any file type in the list whose **State** check box is selected. It will look in any file type in the list whose **State** check box is cleared.

   a Clear or select the **State** check box as needed to instruct Find Files about file types to skip.

   b If you want Find Files to skip a file type not shown in the list, enter the file extension in the field at the top of the dialog box and click **Add**. The type appears in the list. Be sure its **State** check box is selected. For the example shown, the `scc` file type was added.

   c You can reduce the size of the list by removing any file extensions. Select the name of the extension and click **Remove**.

5 Click **OK** to accept the changes and close the **Edit Skipped File Extensions** dialog box.

6 When you click **Find** in the Find Files tool, the search ignores the selected file types.

Function Alternative

Use `lookfor` to search for the specified text in the first line of help for all M-files on the search path.

Comparing Files

The MATLAB File Comparison tool highlights the line-by-line differences between two files. Access it by selecting a file in the Current Directory browser, then right-click, and from the context menu, select **Compare Against > Browse** and navigate to the file to compare. Select it and click **Open**.

For two files in the same directory, select the files, right-click, and from the context menu, select **Compare Selected Files**.

For information about using the results and other options, see "Comparing Files — File Comparison Tool" on page 6-54.

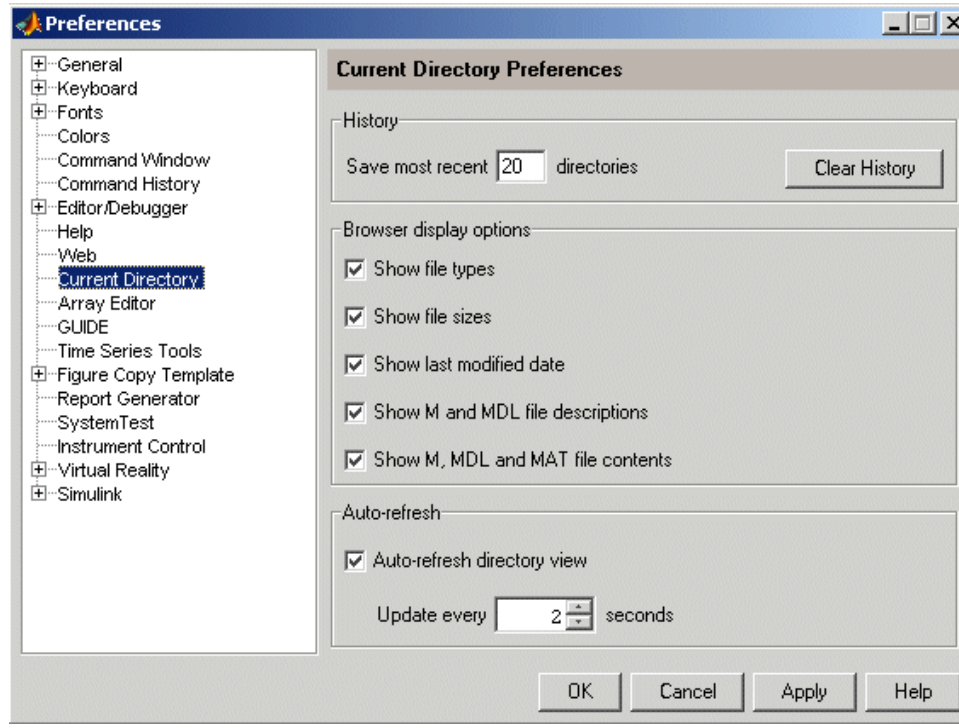
Accessing Source Control Features

Select a file or files in the Current Directory browser and right-click to view the context menu. From there you can access features for Source Control. For details on these features, see Chapter 9, "Source Control Interface".

Preferences for the Current Directory Browser

Using preferences, you can specify the number of recently used current directories to maintain in the history list as well as the type of information to display in the Current Directory browser.

From the Current Directory browser, select **File > Preferences**. The **Current Directory Preferences** pane appears in the Preferences dialog box.



History

The drop-down list in the current directory field shows the history of current directories, that is, the most recently used current directories.

Saving Directories. When the MATLAB session ends, the list of directories will be maintained. Use the **Save most recent directories** field to specify how many directories will appear on the list at the start of the next MATLAB session.

Removing Directories. To remove all entries in the list, click **Clear History**. The list is cleared immediately.

Browser Display Options

In the Current Directory browser, you can view or hide the following information by selecting the appropriate **Browser display options**:

- File type
- File size
- Last modified date
- M-file descriptions (the first comment line in the M-file, also called the H1 line) and the start of MDL file descriptions (approximately the first 128 characters)
- M-file help, MDL complete descriptions, and MAT-file contents

For more information, see “Changing the Display” on page 5-41.

Auto-Refresh

By default, the **Auto-refresh directory view** check box is selected, with an update time of 2 seconds. This means that every 2 seconds, the Current Directory browser checks for and reflects any changes you made to files and directories in the current directory using other applications.

In some cases when the current directory is on a network, MATLAB becomes slow because of the auto-refresh feature in the Current Directory browser. If you experience general slowness in MATLAB and have the Current Directory browser open, try increasing the default update time to alleviate this problem. For extremely slow performance situations, clear the check box to turn auto-refresh off. You can then right-click and select **Refresh** from the context menu to update the Current Directory browser display.

6

Editing and Debugging M-Files

MATLAB provides powerful tools for creating, editing, and debugging files, as detailed here. For information about the MATLAB language and writing M-files, see the MATLAB Programming documentation.

Navigating in an M-File (p. 6-42)	Go to a line number, function, bookmark, back and forward, and open a selection.
Finding Text in Files (p. 6-49)	Find and replace text in the current file or multiple files. Incremental search tool.
Comparing Files — File Comparison Tool (p. 6-54)	View differences between two files.
Keyboard Shortcuts in the Editor/Debugger (p. 6-59)	Use the keyboard to navigate in or perform other common actions in a file.
Saving, Printing, and Closing Files in the Editor/Debugger (p. 6-62)	Save and autosave features, printing and page setup, and closing files.
Running M-Files in the Editor/Debugger (p. 6-66)	Running M-files from the Editor/Debugger, with no input arguments or with input arguments
Finding Errors, Debugging, and Correcting M-Files (p. 6-84)	Automatically analyze code using M-Lint to find errors and make improvements, and use debugging features to isolate run-time problems.
M-Lint Code Analyzer (p. 6-87)	Check your code for problems and get recommendations to maximize performance and maintainability.
Debugging Process and Features (p. 6-103)	Graphical debugging tools and functions for debugging in the Command Window.
Using Cells for Rapid Code Iteration and Publishing Results (p. 6-133)	Define sections of your M-files as cells. Use cells for publishing M-files to formats like HTML. Also use cells to experiment and incrementally modify values in M-files.

Begin with Existing Code

In this section...

Create M-Files from Command Window and History

Before you begin writing MATLAB code in a blank file, consider starting with existing resources for the code, and then use the MATLAB Editor/Debugger to modify the code.

In many cases, you create and run MATLAB statements in the Command Window, modify those statements to your satisfaction, and then create an M-file that includes the statements. To facilitate this process, in the Command History, select the MATLAB statements you want to include in the M-file. Right-click and select **Create M-File**. The Editor/Debugger opens a new file that includes the statements you selected from the Command History. You can also copy the statements from the Command History and paste them into an existing M-file.

Use Existing M-Files and Examples

If you can find existing M-files that accomplish what you want to do, copy and use the code in your own M-file, assuming you have legal permission to do so. Following are some resources you can use.

MATLAB and Toolbox M-Files

You can access and reuse the code in most MATLAB and toolbox functions that have a `.m` file extension. You cannot use MATLAB and toolbox functions that are built-in. They are efficient but their code is not accessible.

If there is a MATLAB function that is similar to what you need to do and it is not built-in, open the file in the Editor/Debugger and use it as a basis for your file. Be sure to save the file using a different name and in a directory that is not in `matlabroot/toolbox`. See “Saving Files” on page 6-62 for details.

Demos and Examples

MATLAB and its toolboxes include demonstration programs. You can view the code in the demos and copy it for use in your own M-files. To see the demos, type `demo`, which opens the Help browser to the **Demos** pane. For more information about demos, see "Demos in the Help Browser" on page 4-31.

There are also code examples in the online documentation. To see a list of examples for a product, type `helpbrowser` to open the Help browser. In the **Contents** pane, click **+** for a product to view the help topics, and then select the **Examples** entry.

File Exchange

The MathWorks Web site features a user-contributed code library, from which you can download free M-files contributed by users and developers of MATLAB, Simulink, and related products. To view the files available to download, go to the MATLAB Central File Exchange page on the MathWorks Web site, `http://www.mathworks.com/matlabcentral/fileexchange/index.jsp`, or access it via the **Help > Web** menu in any desktop component.

Ways to Edit, Evaluate, and Debug M-Files

There are several methods for creating, editing, evaluating, and debugging files with MATLAB.

Creating and Editing Files — Options	Instructions
MATLAB Editor/Debugger	See "Starting, Customizing, and Closing the Editor/Debugger" on page 6-7, and "Saving, Printing, and Closing Files in the Editor/Debugger" on page 6-62. You can create, open, edit and save M-files as well as other file types in the MATLAB Editor/Debugger — see "Creating and Editing Other Text File Types" on page 6-12.
Any text editor, such as Emacs or vi	To specify another editor as the default for use with MATLAB, select **File > Preferences > Editor/Debugger**, and for **Editor**, specify the **Text editor**. Click the **Help** button in the Preferences dialog box for details. Use that editor by default, or use any other editor you open. Regardless of the editor you use, you can debug M-files using the MATLAB Editor/Debugger or debugging functions.

Debugging M-Files – Options	Instructions
General debugging tips	See "Finding Errors, Debugging, and Correcting M-Files" on page 6-84.
MATLAB Editor/Debugger	See • "M-Lint Code Analyzer" on page 6-87 to identify errors and make improvements. • "Debugging Process and Features" on page 6-103 to help you isolate run-time problems.
MATLAB debugging functions (for use in the Command Window)	See function alternatives in "Debugging Process and Features" on page 6-103.

Use preferences for the Editor/Debugger to set up the editing and debugging environment to best meet your needs.

For information about the MATLAB language and writing M-files, see the MATLAB Programming documentation.

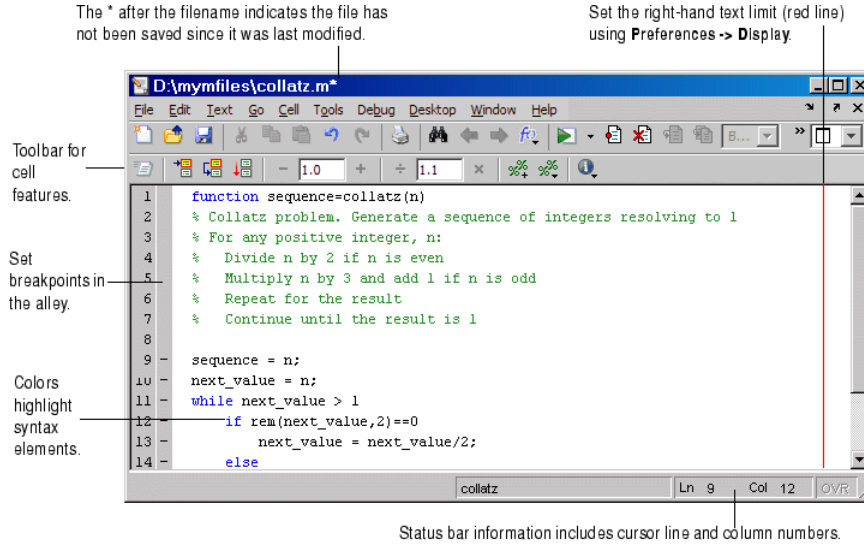
Starting, Customizing, and Closing the Editor/Debugger

In this section...

Starting the Editor/Debugger

The MATLAB Editor/Debugger provides a graphical user interface for basic text editing features for any file type, as well as for M-file debugging. The Editor/Debugger is a single tool that you can use for editing, debugging, or both. There are various ways to start the Editor/Debugger. The Editor/Debugger automatically starts when you open a document or create a new one. Once started, you can customize the Editor/Debugger to suit your needs.

This figure shows an example of the Editor/Debugger outside of the desktop opened to an existing M-file, and calls out some of the tool's useful features.



Creating a New File in the Editor/Debugger

To create a new text file in the Editor/Debugger, either click the New M-file button on the MATLAB desktop toolbar, or select **File > New > M-File** from the MATLAB desktop. The Editor/Debugger opens, if it is not already open, with an untitled file in the MATLAB current directory, in which you can create an M-file or another type of text file.

The location of the new file and the Editor/Debugger are determined by document positioning guidelines. You can rearrange the documents to suit your needs. For details, see “Opening and Arranging Documents” on page 2-8.

If the Editor/Debugger is open, create more new files by using the New M-file button on the toolbar, or select **File > New > M-File**.

Other tools also provide features for creating new M-files. For example, in the Command History, select statements, right-click, and select **Create M-File** from the context menu. Similarly, create a new file from the context menu in the Current Directory browser — see “Creating New Files” on page 5-44.

Function Alternative

Type `edit` in the Command Window to create a new file in the Editor/Debugger.

Type `edit filename.ext` to create the file `filename.ext`. If `filename.ext` already exists in the current directory or on the MATLAB search path, this opens the existing file. If `filename.ext` does not exist in the current directory or on the MATLAB search path, a confirmation dialog box might appear asking if you want to create a new file titled `filename.ext`:

- If you click **Yes**, the Editor/Debugger creates a blank file titled `filename.ext`. If you do not want the dialog to appear in this situation, select that check box in the dialog. Then, the next time you type `edit filename.ext`, the file is created without first prompting you.
- If you click **No**, the Editor/Debugger does not create a new file. If you do not want the dialog to appear in this situation, select that check box in the dialog. In that case, the next time you type `edit filename.ext`, a "`file not found`" message appears.

For more information about the confirmation dialog box, see preferences for "Confirmation Dialogs Preferences" on page 2-81.

Opening Existing Files in the Editor/Debugger

To open an existing file in the Editor/Debugger, click the Open file button on the desktop or Editor/Debugger toolbar, or select **File > Open**.

The **Open** dialog box appears, listing all M-files. You can see different files by changing the selection for **Files of type** in the dialog box. Type or select a filename, and click **Open**. If you access the **Open** dialog box from the desktop, the current directory files are shown, but if you access it from the Editor/Debugger, the files in the directory for the current file are shown.

The Editor/Debugger opens, if it is not already open, with the file displayed. You can have multiple Editor/Debugger files open at once, and the location of the files and the Editor/Debugger are determined by document positioning guidelines. You can rearrange the documents to suit your needs. For details, see "Opening and Arranging Documents" on page 2-8.

To make a document in the Editor/Debugger become the current document, click it, or select it from the **Window** menu or document bar.

M-File Cells

If you open an M-file that contains M-file cells, yellow highlighting and gray horizontal lines might appear in the M-file, along with an information toolbar. Cell mode is used for publishing results and rapid code iteration. An M-file cell is denoted by a %% at the start of a line. MATLAB interprets any M-file that contains %% at the start of a line as including cells and the Editor/Debugger reflect the cell toolbar state and the cell display preferences, such as yellow highlighting of the current cell and gray lines between cells.

The first time you open an M-file that contains cells, an information bar appears below the cell toolbar, providing links for details about cell mode. To dismiss the information bar, click the close box on the right side of the bar. The information bar does not appear again, but you can get the same quick access to the information about M-file cells from the information button on the cell toolbar.

To hide the cell toolbar, right-click in the toolbar and select **Cell Toolbar** from the context menu. If you do not want cell mode enabled, select **Cell > Disable Cell Mode**. Because MATLAB remembers the cell mode between sessions, if cell mode is disabled when you quit MATLAB, it is disabled the next time you start MATLAB, and the converse is true.

Other Methods for Opening Files in the Editor/Debugger

These are other ways to open files in the Editor/Debugger:

- Drag a file from another MATLAB desktop tool or a Windows tool into the Editor/Debugger. For example, drag files from the Current Directory browser, or from Windows Explorer.
- Open files from the Current Directory browser — see “Opening Files” on page 5-47.
- Select a file to open from the most recently used files, which are listed at the bottom of the **File** menu in the Editor/Debugger and all other desktop tools. You can change the number of files appearing on the list — select

File > Preferences > Editor/Debugger and in the **Most recently used file list**, specify the **Number of entries**.

- In the Editor/Debugger or another desktop tool such as the Command Window, select a filename, right-click, and select **Open Selection** from the context menu to open that file. For details, see "Opening a Selection in an M-File" on page 6-48.
- Set a preference that instructs MATLAB, upon startup, to automatically open the files that were open when the previous MATLAB session ended. Select **File > Preferences > Editor/Debugger** and in the **Opening files in editor area**, select the check box for **On restart reopen files from previous MATLAB session**.

Function Alternative for Opening an M-File. Use the `edit` or `open` function to open an existing file in the Editor/Debugger. For example, type

```
edit collatz.m
```

to open the file `collatz.m` in the Editor/Debugger, where `collatz.m` is on the search path or in the current directory. Use the relative or absolute pathname for the file you want to open if it is not on the search path or in the current directory.

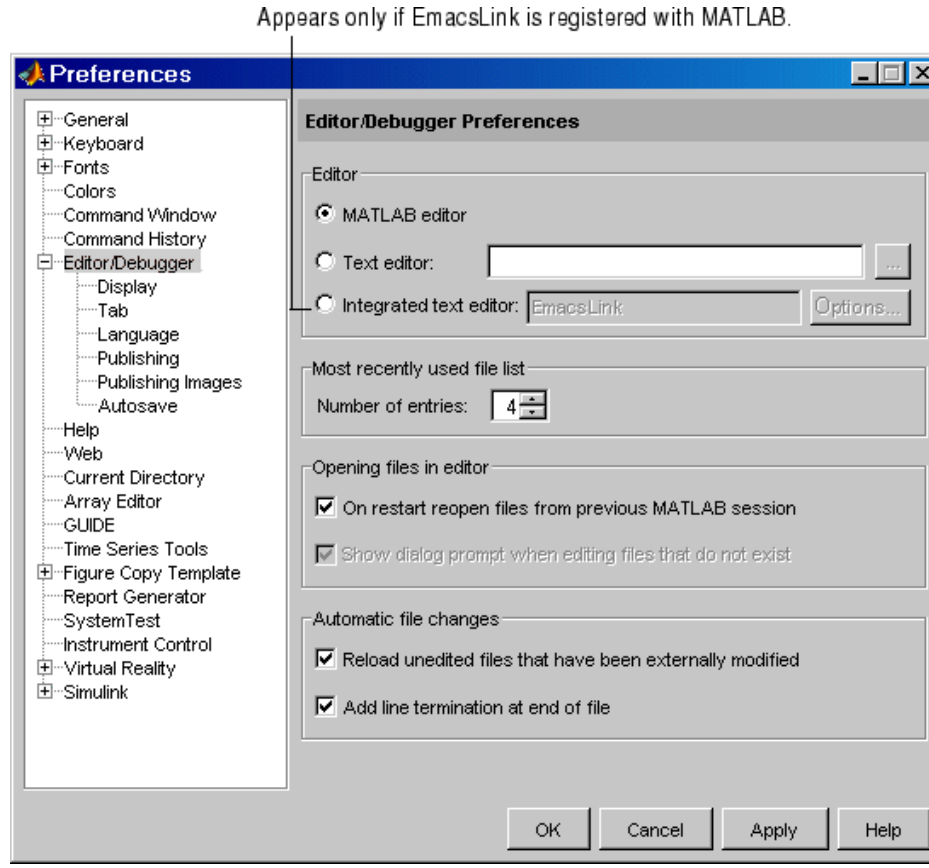
Arranging Editor/Debugger Documents

You can arrange the size and location of M-files and other text documents you open in the Editor/Debugger. Editor/Debugger documents follow the same arrangement practices as other desktop documents. For details, see "Opening and Arranging Documents" on page 2-8.

Preferences for the Editor/Debugger

Using preferences, you can specify the default behavior for various aspects of the Editor/Debugger.

To set preferences for the Editor/Debugger, select **File > Preferences**. The Preferences dialog box opens showing **Editor/Debugger Preferences**.



Click the **+** next to Editor/Debugger in the left pane to view all categories of Editor/Debugger preferences. Select a category and that preference pane displays. Make changes and click **Apply** or **OK**.

Click the **Help** button in the Preferences dialog box for details about Editor/Debugger preferences.

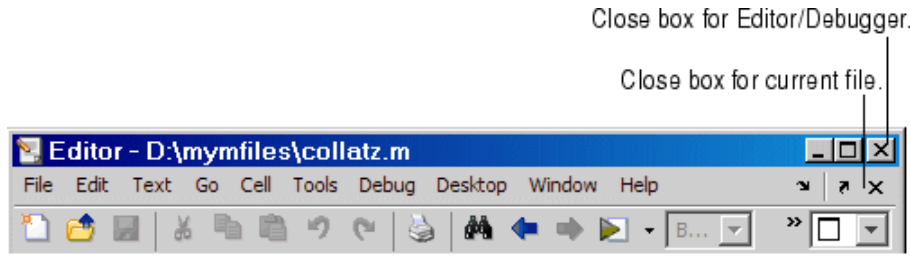
Creating and Editing Other Text File Types

You can edit any type of text file using the MATLAB Editor/Debugger. For example, you can open and edit an HTML file. Note that you can run or debug only M-files from the Editor/Debugger.

When working with files created for C/C++, Java, and HTML, you can specify syntax highlighting and indenting preferences appropriate to those languages. Select **File > Preferences > Editor/Debugger > Language**. See details in the online documentation for Editor/Debugger language preferences, or click the **Help** button in the dialog box.

Closing the Editor/Debugger

To close the Editor/Debugger, click the **Close** box in the title bar of the Editor/Debugger. This is different from the **Close** box in the menu bar of the Editor/Debugger, which closes the current file when multiple files are open in a single window.



If multiple files are open, with each in a separate window, close each window separately. To close all files at once, select **Close All Documents** from the **Window** menu. Note that this will close other desktop documents as well, such as arrays in the Array Editor, and it will close the tools as well, that is, the Editor/Debugger and Array Editor, for example.

When you close the Editor/Debugger and any of the open files have unsaved changes, you are prompted to save the files.

Entering Statements in the Editor/Debugger

In this section...

Use Command Window Features in the Editor/Debugger

After opening an existing file or creating a new file in the Editor/Debugger, enter statements in the file. Follow the same rules you would use for entering statements in the Command Window as described in Chapter 3, "Running Functions — Command Window and History":

- "Case and Space Sensitivity" on page 3-15
- "Matching Delimiters (Parentheses)" on page 3-17
- "Entering Multiple Functions in a Line" on page 3-18
- "Entering Long Statements (Line Continuation)" on page 3-18
- "Suppressing Output" on page 3-30
- "Formatting and Spacing Numeric Output" on page 3-31

In addition, utilize the Editor/Debugger features described in the remaining parts of this section.

Changing the Case of Selected Text

To change the case of text in the Editor/Debugger, select the text and then from the **Text** menu, select one of the following:

- **Change to Upper Case** to change all text to uppercase
- **Change to Lower Case** to change all text to lowercase

- **Reverse Case** to change the case of each letter

This is useful, for example, when copying syntax from help in an M-file, where function and variable names are distinguished by the use of uppercase. But because of that, the code will not run in MATLAB. In this example, the text was copied and pasted from the output of `help get`.

```
V = GET(H, 'Default')
```

Select all of the text. Select **Text > Change to Lower Case**. The text becomes

```
v = get(h, 'default')
```

If instead you select **Reverse Case** for

```
V = GET(H, 'Default')
```

the case changes to

```
v = get(h, 'dEFAULT')
```

Undo and Redo

You can undo many of the Editor/Debugger actions listed in **Edit** and **Text** menus. Select **Edit > Undo**. You can undo multiple times in succession until there are no remaining actions to undo. Select **Edit > Redo** to reverse an undo.

Adding Comments

Comments in an M-file are strings or statements that do not execute. Add comments in an M-file to describe the code or how to use it. Comments determine what text displays when you run `help` for a filename. Use comments when testing your files or looking for errors—temporarily turn lines of code into comments to see how the M-file runs without those lines. These topics provide details:

- “Commenting in M-Files Using the MATLAB Editor/Debugger” on page 6-16

Commenting in M-Files Using the MATLAB Editor/Debugger

You can comment the current line or a selection of lines in an M-file:

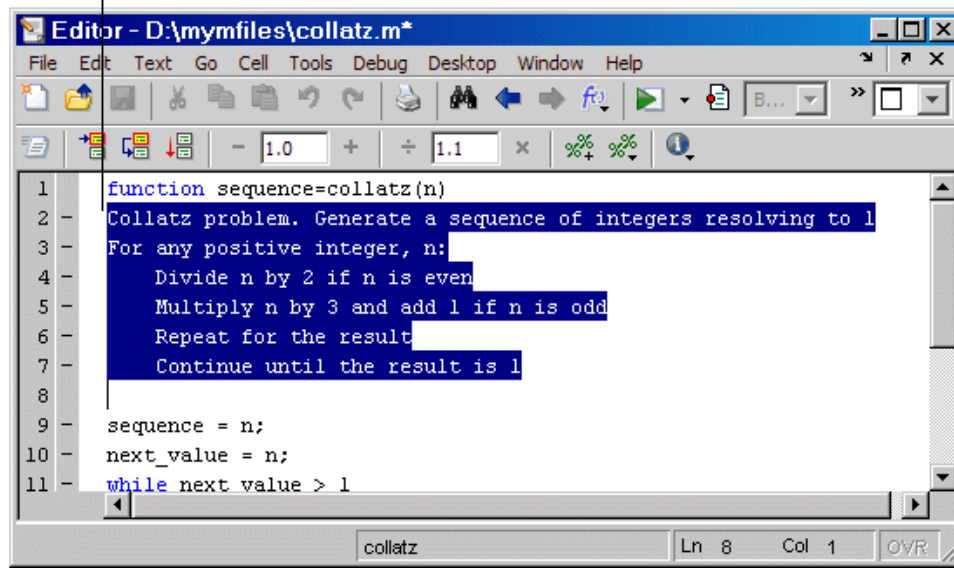
1. For a single line, position the cursor in that line. For multiple lines, click in the line and then drag or **Shift**+click to select multiple lines.
2. Select **Comment** from the **Text** menu, or right-click and select it from the context menu.

A comment symbol, `%`, is added at the start of each selected line, and the color of the text becomes green or the color specified for comments — see "Syntax Highlighting" on page 6-28.

To uncomment the current line or a selected group of lines, select **Uncomment** from the **Text** menu, or right-click and select it from the context menu.

Click in the area to the left of a line to select that line.
To select multiple lines, click+drag or **Sh**ift+click.

Select **Text -> Comment** to make all the selected lines comments.



Commenting in Java and C/C++ Files Using the MATLAB Editor/Debugger

For Java and C/C++ files, selecting **Text > Comment** adds the // symbols at the front of the selected lines. Similarly, **Text > Uncomment** removes the // symbols from the front of selected lines in Java and C/C++ files.

Commenting in M-File Using Any Text Editor

You can make any line in an M-file a comment by typing % at the beginning of the line. To put a comment within a line, type % followed by the comment text; MATLAB treats all the information after the % on a line as a comment.

```
% This is a comment.
```
——MATLAB ignores this comment line when you run the M-file.

```
This is not a comment.
```
——This line produces an error when you run the M-file.

To uncomment any line, delete the comment symbol, %.

To comment a contiguous group of lines, type `%{` before the first line and `%}` after the last line you want to comment. This is referred to as a block comment. The lines that contain `%{` and `%}` can contain spaces, but not contain any other text. After typing the opening block comment symbol, `%{`, all subsequent lines assume the syntax highlighting color for comments until you type the closing block comment symbol, `%}`. Remove the block comment symbols, `%{` and `%}`, to uncomment the lines.

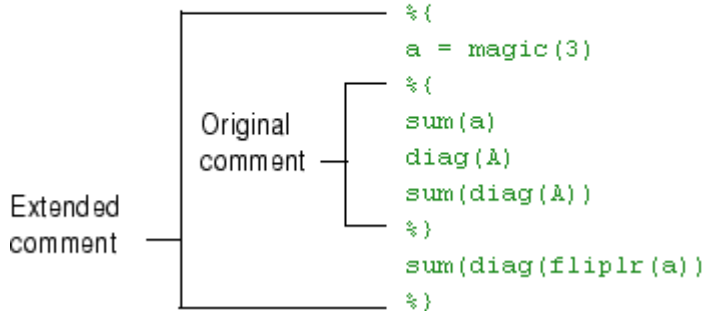
This examples shows some lines of code commented out. When you run the M-file, the commented lines will not execute. This is useful when you want to identify the section of a file that is not working as expected.

Comment a block of code by adding `%{` before the first line and `%}` after the last line.

```
a = magic(3)
%{
sum(a)
diag(A)
sum(diag(a))
%}
sum(diag(fliplr(a)))
```

You can easily extend a block comment without losing the original block comment, that is, create a nested block comment, as shown in the following example.

Create a nested comment, that is, a block comment within a block comment.



Commenting Out Part of a Statement

To comment out the end of a statement in an M-file, put the comment character, `%`, before the comment. When you run the file, MATLAB ignores any text on the line after the `%`.

Any text following a `%` within a line is considered to be a comment.

```
a = zeros(10) % Initialize matrix
```

To comment out text within a multiline statement, use the ellipsis (`...`). MATLAB ignores any text appearing after the `...` on a line and continues processing on the next line. This effectively makes a comment out of anything on the current line that follows the `...`. The following example comments out the `Middle Initial` line.

```
header = ['Last Name, '...
'First Name, '...
... 'Middle Initial, '...
'Title']
```

MATLAB ignores the text following the `...` on the line

```
... 'Middle Initial, '...
```

Note that `Middle Initial` is green, which is the syntax highlighting color for a comment.

MATLAB continues processing the statement with the next line

```
'Title']
```

MATLAB effectively runs

```
headers = ['Last Name, ' ...
'First Name, ' ...
'Title']
```

Formatting Comments in M-Files

To make comment lines in M-files wrap when they reach a certain column,

1. Specify the maximum column number using preferences for the Editor/Debugger. Select **Language > M**. For **Comment formatting**, set the **Max width**.

2. Select contiguous comment lines that you want to limit to the specified maximum width.

3. Select **Text > Wrap Selected Comments**.

   The selected comment lines are reformatted so that no comment line in the selected area is longer than the maximum. Lines that were shorter than the specified maximum are merged to make longer lines if they are at the same level of indentation.

To automatically limit comment lines to the maximum width while you type, select the **Comment formatting** preference to **Autowrap comments**.

For example, assume you select **Autowrap comments** and set the maximum width to be 75 characters, which is the width that will fit on a printed page using the default font for the Editor/Debugger. When typing a comment line,

as you reach the 75th column, the comment automatically continues on the next line.

Tab Completion in the Editor/Debugger

The Editor/Debugger helps you automatically complete the names of these items as you type them in an M-file:

- Functions or models on the search path or in the current directory
- Variables, including structures, in the current workspace, where the current workspace is shown in the **Stack** on the toolbar.
- Handle Graphics properties for figures in the current workspace

Type the first few characters of the item name and then press the **Tab** key. To use tab completion, you must have the tab completion preference for the Editor/Debugger selected. For details, see “Keyboard Preferences” on page 3-43.

Tab completion is also available in the Command Window. There are a few minor differences in how tab completion works in the Command Window, the most notable being that Command Window tab completion supports the completion of filenames, whereas the Editor/Debugger tab completion does not.

Note Tab completion does not complete the names of variables you define in an M-file, but only those variables in the current workspace. This means that while editing, it only completes the names of variables in the base workspace. While debugging, it only completes the names of variables in the current function workspace.

These examples demonstrate how to use tab completion:

- “Basic Example — Unique Completion” on page 6-22
- “Multiple Possible Completions” on page 6-23
- “Narrowing Completions Shown” on page 6-24

Basic Example — Unique Completion

This example illustrates a basic use for tab completion in the Editor/Debugger. In an M-file opened in the Editor/Debugger, type the beginning of a function or model on the MATLAB search path or in the current directory, for example,

```
horz
```

and press **Tab**. The Editor/Debugger automatically completes the name, which for this example displays the function name

```
horzcat
```

Complete the statement, adding any arguments, operators, or options. If the Editor/Debugger does not complete the name `horzcat` but instead moves the cursor to the right, you do not have the preference set for tab completion. The Editor/Debugger also moves the cursor to the right when you try to complete a filename; filename tab completion is not supported in the Editor/Debugger, but is supported in the Command Window.

You can use tab completion anywhere in the line, not just at the beginning. For example, if you type

```
a = horz
```

and press **Tab**, the Editor/Debugger completes `horzcat`.

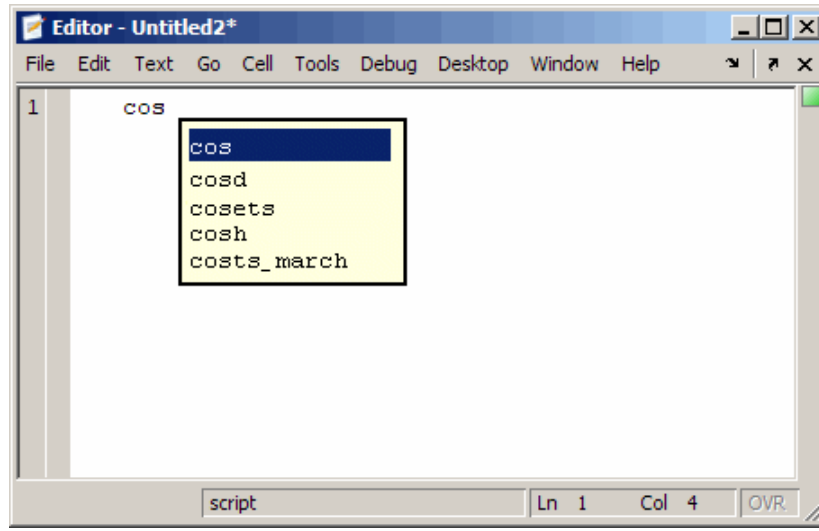
The Editor/Debugger also completes the names of variables in the current workspace. For example, if there is a variable `costs_march` in the currently selected workspace, type `cost` and press **Tab**. The Editor/Debugger completes the variable name `costs_march`. If MATLAB displays `No Completions Found`, `costs_march` does not exist in the current workspace.

Multiple Possible Completions

If there is more than one name that starts with the characters you typed, when you press the **Tab** key, the Editor/Debugger displays a list of all names that start with those characters. For example, assume you had created the variable `costs_march` in the base workspace. In an M-file in the Editor/Debugger, type

```
cos
```

and press **Tab**. The Editor/Debugger displays



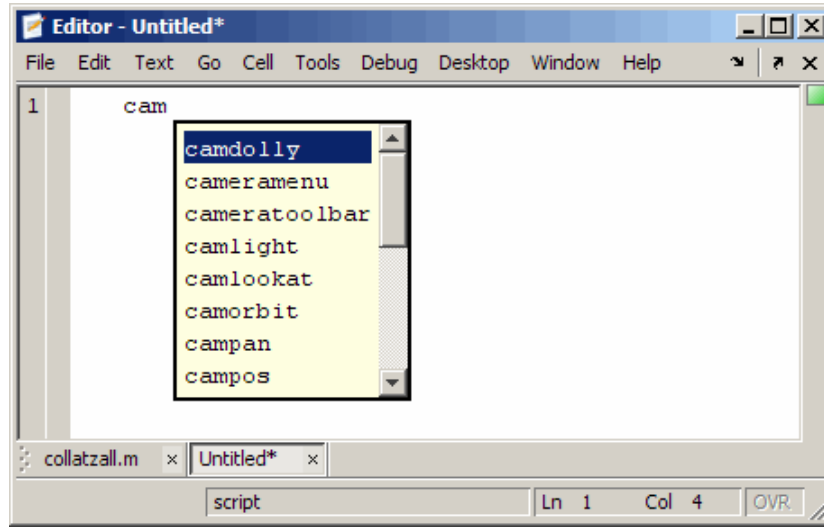
The resulting list of possible completions includes the variable name you created, `costs_march`, but also includes functions and models that begin with `cos`, including `cosets` from Communications Toolbox, if it is installed and on the MATLAB search path.

Continue typing to make your entry unique. For example, type the next character, such as `t` in the example. The Editor/Debugger selects the first item in the list that matches what you typed, in this case, `costs_march`. Press **Enter** (or **Return**) or **Tab** to select that item, which completes the name in the M-file. In the example, MATLAB displays `costs_march` at the prompt.

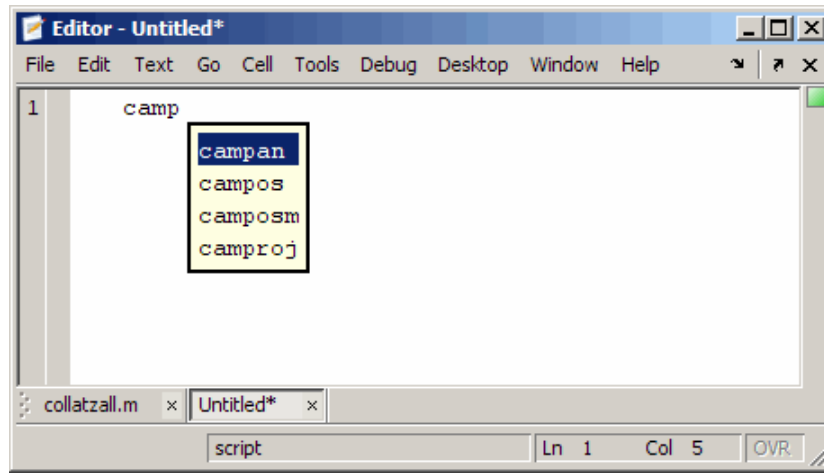
You can navigate the list of possible completions using up and down arrow keys, and **Page Up** and **Page Down** keys. You can clear the list without selecting anything by pressing **Esc**. The list of possible completions might include items that are not valid commands, such as private functions.

Narrowing Completions Shown

You can narrow the list of completions shown by typing a character and then pressing **Tab** if the **Keyboard** preference **Tab key narrows completions** is selected. This is particularly useful for large lists. For example, type `cam` and press **Tab** to see the possible completions. There is a scroll bar with the list because there are too many completions to be seen at once.



Type `p` and press **Tab** again. The Editor/Debugger narrows the list, showing only all possible `camp` completions.



Continue narrowing the list in the same way. For the above example, type `o` and press **Tab** to further narrow the list. Press **Enter** or **Return** to select an item, which completes the name at the prompt.

Tab Completion for Structures

For structures that are in the current workspace, after the period separator, press **Tab**. For example, type

```
mystruct.
```

and press **Tab** to display all fields of `mystruct`. If you type a structure and include the start of a unique field after the period, pressing **Tab** completes that structure and field entry.

For example, type

```
mystruct.n
```

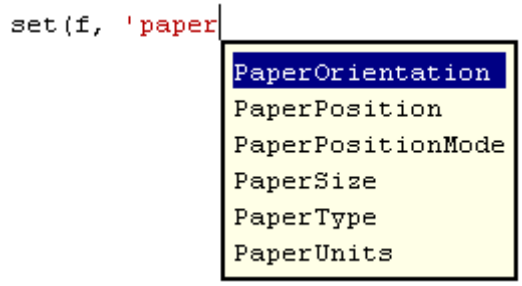
and press **Tab**, which completes the entry `mystruct.name`, where `mystruct` is in the current workspace and contains no other fields that begin with `n`.

Tab Completion for Properties

Complete property names for figures in the current workspace using tab completion, as in this graphics example. Here, `f` is a figure. Type

```
set(f, 'pap
```

and press **Tab**. The Editor/Debugger displays



Select a property from the list. For example, type

```
u
```

and press **Enter**. The Editor/Debugger completes the property, including the closing quote.

```
set(f, 'paperunits'
```

Continue adding to the statement, as in this example,

```
set(f, 'paperunits', 'c
```

and press **Tab**. The Editor/Debugger automatically completes the property

```
set(f, 'paperUnits', 'centimeters'
```

because `centimeters` is the only possible completion.

Using Tab for Spacing

If the preference for tab completion is selected, and you want to also use the **Tab** key to add spacing within your statements, add a space before pressing **Tab**. For example, to create this statement

```
if a=mate     %test input value
```

add a space after `mate` and then press **Tab**. If you do not include the space, the following happens instead:

```
if a=material
```

This is because the tab completion feature automatically causes `mate` to complete as the `material` function.

Alternatively, turn off the tab completion preference to use **Tab** for spacing in the Editor/Debugger.

Appearance of an M-File — Making Files More Readable

In this section...

Note You can specify the default behaviors for some of these features—see "Preferences" on page 2-59.

Syntax Highlighting

Some entries appear in different colors to help you better find matching elements, such as `if/else` statements. Similarly, unterminated strings have a different color than terminated strings. This is called syntax highlighting and is used in the Command Window and History, as well as in the Editor/Debugger. For more information, see the Command Window documentation for "Syntax Highlighting" on page 3-16.

When you paste or drag a selection from the Editor/Debugger to another application, such as Microsoft Word, the pasted text maintains the syntax highlighting colors and font characteristics from the Editor/Debugger. MATLAB pastes the selection to the clipboard in RTF format, which many Windows and Macintosh applications support.

Indenting

Automatic Indenting

You can set an indenting preference so that program control entries are automatically indented to make reading loops, such as `while`/`end` statements, easier. To do so, select **File > Preferences > Editor/Debugger > Language**, and select a **Language**, for example, `M`. For **Indenting for Enter key**, select **Smart indenting** or **Block indent**, and then click **OK**. Use **No indent** instead if you want to indent manually. For more information about indenting preferences, see the in the online documentation. Specify the indenting size and other options using in the online documentation.

Manual Indenting

You can manually apply smart indenting to selected lines — select the lines and then select **Smart Indent** from the **Text** menu, or right-click and select it from the context menu. This feature indents lines that start with keyword functions or that follow lines containing certain keyword functions. Smart indenting can help you to follow the code sequence.

To move the current or selected lines further to the left, select **Decrease Indent** from the **Text** menu. To move the current or selected lines further to the right, select **Increase Indent** from the **Text** menu.

You can also indent a line by pressing the **Tab** key at the start of a line. Or select a line or group or lines and press the **Tab** key. Press **Shift+Tab** to decrease the indent for the selected lines. This works differently if you select the Editor/Debugger **Tab** preference for **Emacs-style Tab key smart indenting** — when you position the cursor in any line or select a group of lines and press **Tab**, the lines indent according to smart indenting practices.

For more information about manual indenting, see in the online documentation.

Function Indenting

If you select the language preference for smart indent, you can select from three indenting options when you enter a subfunction or a nested function

(a function within a function) in the Editor/Debugger. For details, see in the online documentation.

Line and Column Numbers

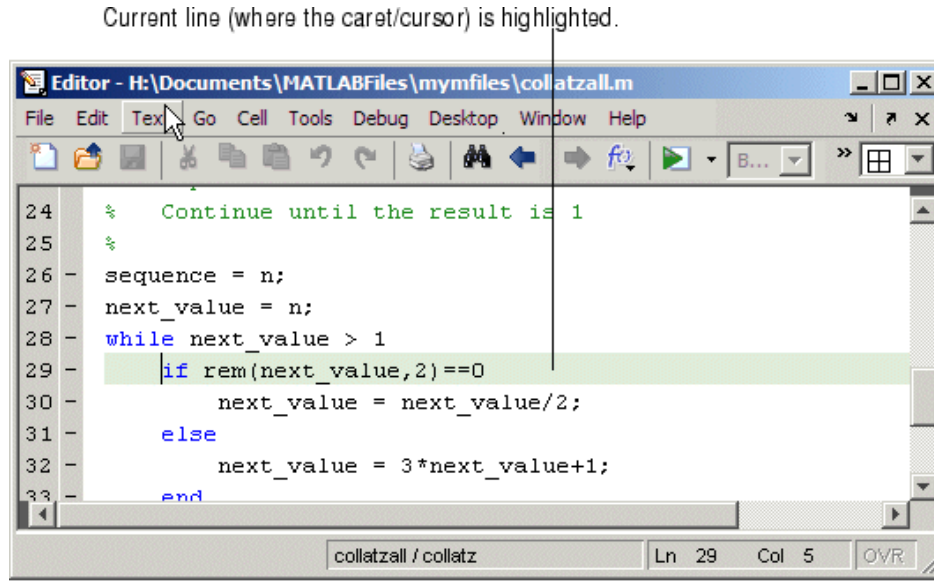
Line numbers are displayed along the left side of the Editor/Debugger window. You can elect not to show the line numbers using preferences — for details, see in the online documentation.

The line and column numbers for the current cursor position are shown in the far right side of the status bar in the Editor/Debugger.

Highlight Current Line

You can set a preference to highlight the current line, that is the line with the caret (also called the cursor). This is useful, for example, to help you see where copied text will be inserted when you paste.

To highlight the current line, select **File > Preferences > Editor/Debugger > Display**, and under **General display options**, select the check box for **Highlight current line**. You can also specify the color used to highlight the line.



Right-Hand Text Limit

By default, a light red vertical line (rule) appears at column 75 in the Editor/Debugger, providing a cue as to when a line becomes wider than desired, which is useful if you plan to print the file, for example. You can hide the line or change the column number at which it appears — see in the online documentation.

View Function or Subfunction

The function or subfunction the cursor is currently at is shown at the right side of the status bar in the Editor/Debugger.

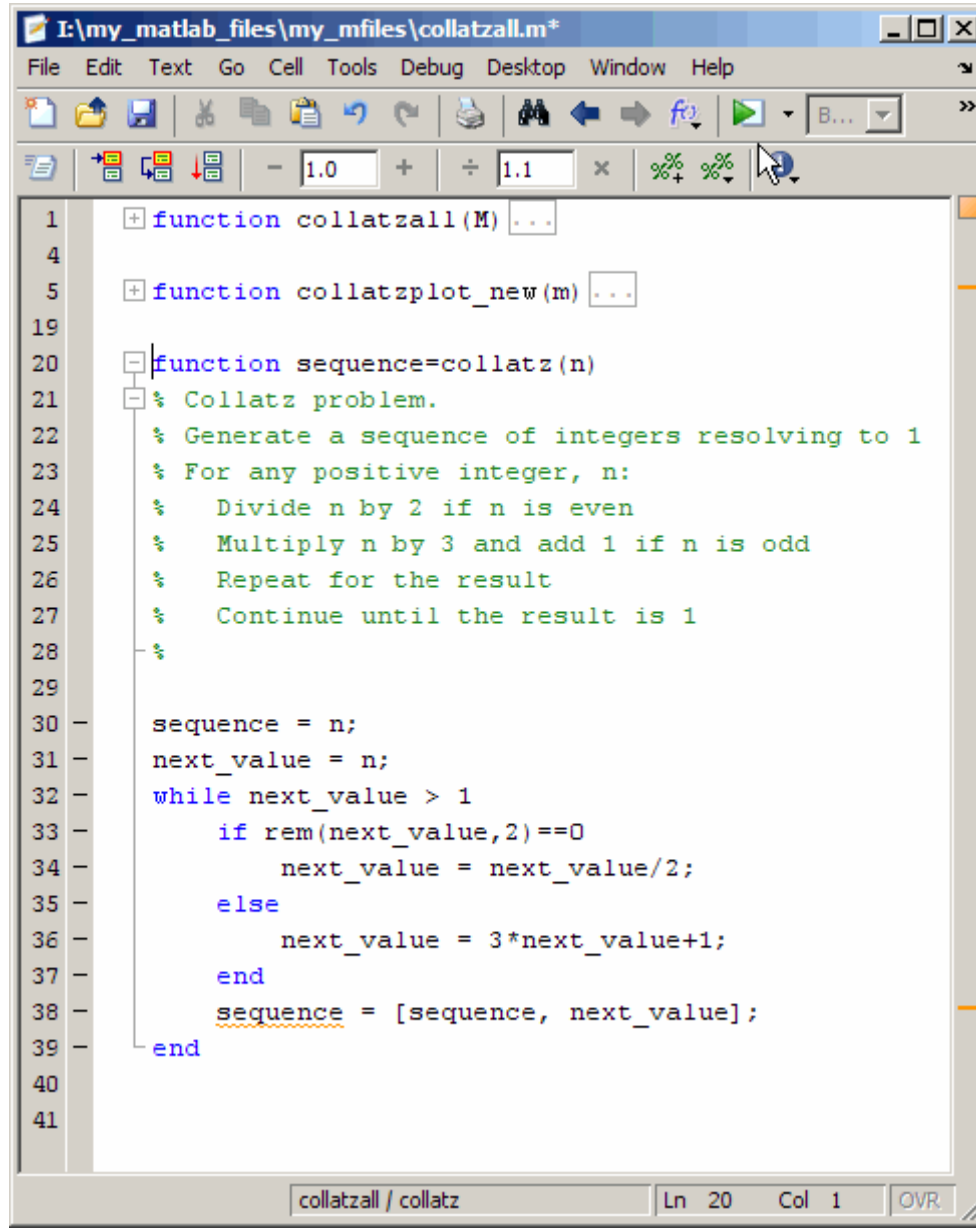
Code Folding—Expanding and Collapsing M-File Constructs

You can enable or disable the ability to expand and collapse M-file programming constructs; this ability is referred to as code folding. Programming constructs include functions and function help.

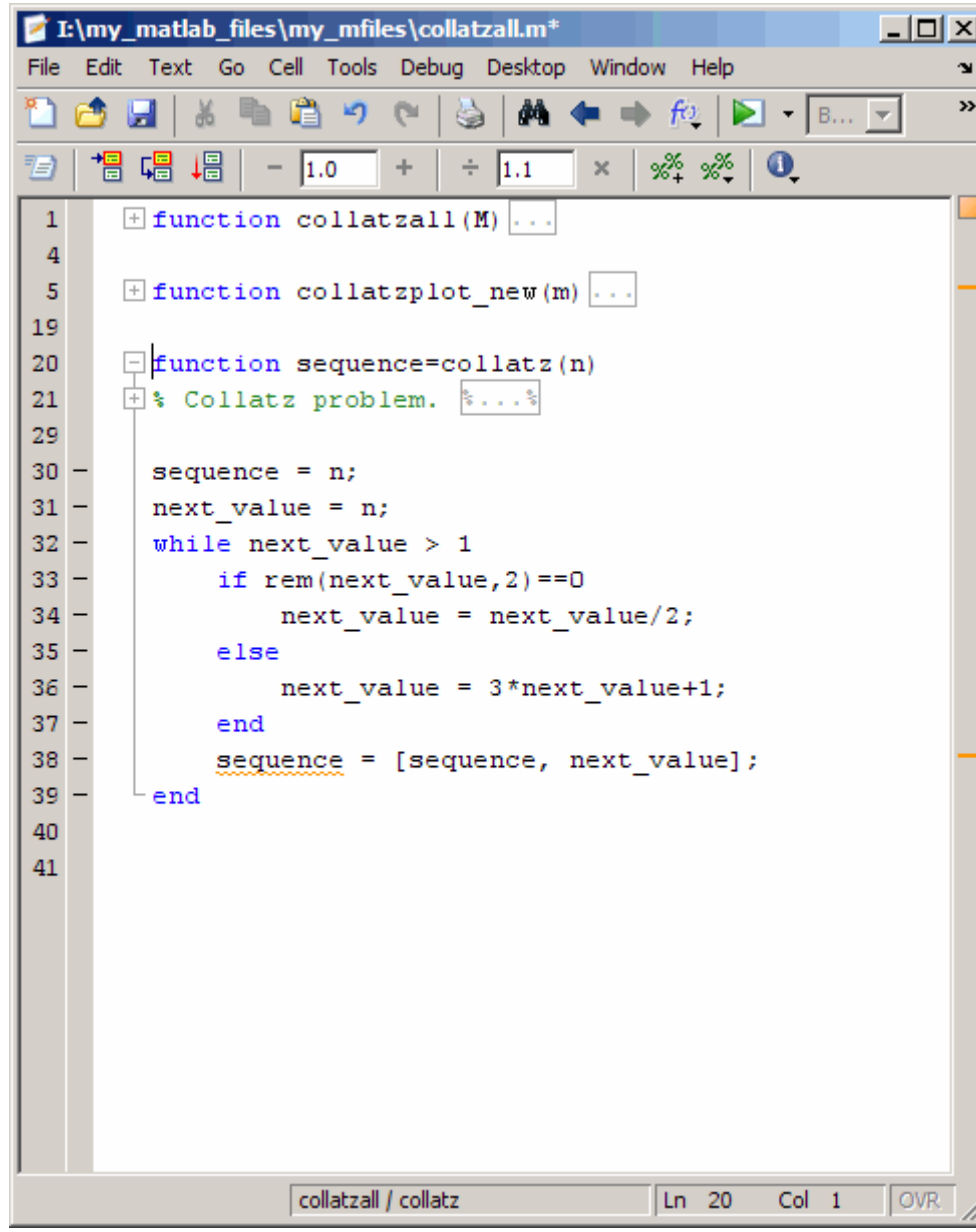
Code folding is particularly useful for improving readability when an M-file contains numerous subfunctions and you want to hide code on which you are not currently working.

When you fold a function, all the code associated with that function (including any help code) is collapsed such that the Editor/Debugger displays only the function definition line. The function definition line is appended with an ellipsis icon  to indicate there is more function code.

The following image shows the `collatz` function code expanded and the `collatzall` and `collatzplot_new` functions collapsed:


I:\my_matlab_files\my_mfiles\collatzall.m*
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1
1 function collatzall(M) ...
4
5 function collatzplot_new(m) ...
19
20 function sequence=collatz(n)
21 % Collatz problem.
22 % Generate a sequence of integers resolving to 1
23 % For any positive integer, n:
24 % Divide n by 2 if n is even
25 % Multiply n by 3 and add 1 if n is odd
26 % Repeat for the result
27 % Continue until the result is 1
28 %
29
30 sequence = n;
31 next_value = n;
32 while next_value > 1
33 if rem(next_value,2)==0
34 next_value = next_value/2;
35 else
36 next_value = 3*next_value+1;
37 end
38 sequence = [sequence, next_value];
39 end
40
41
collatzall / collatz
Ln 20 Col 1
OVR

When you expand a function, but collapse its associated help code, the Editor/Debugger displays all the function code and just the H1 line of the help code. The H1 line ends with a commented ellipsis icon  to indicate there is additional help code, as shown in the following image:


I:\my_matlab_files\my_mfiles\collatzall.m*
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1
1 function collatzall(M) ...
4
5 function collatzplot_new(m) ...
19
20 function sequence=collatz(n)
21 % Collatz problem. %...%
29
30 sequence = n;
31 next_value = n;
32 while next_value > 1
33 if rem(next_value,2)==0
34 next_value = next_value/2;
35 else
36 next_value = 3*next_value+1;
37 end
38 sequence = [sequence, next_value];
39 end
40
41
collatzall / collatz
Ln 20 Col 1
OVR

To expand code for a construct that is currently collapsed, do one of the following:

- Click the plus sign icon ⊞ to the left of the construct that you want to expand.
- Place your cursor in the code that you want to expand, right-click, and then select **Code Folding > Expand** from the context menu.

To collapse code for a construct that is currently expanded, do one of the following:

- Click the minus sign icon ⊟ to the left of the construct that you want to collapse.
- Place your cursor in the code that you want to collapse, right-click, and then select **Code Folding > Collapse** from the context menu.

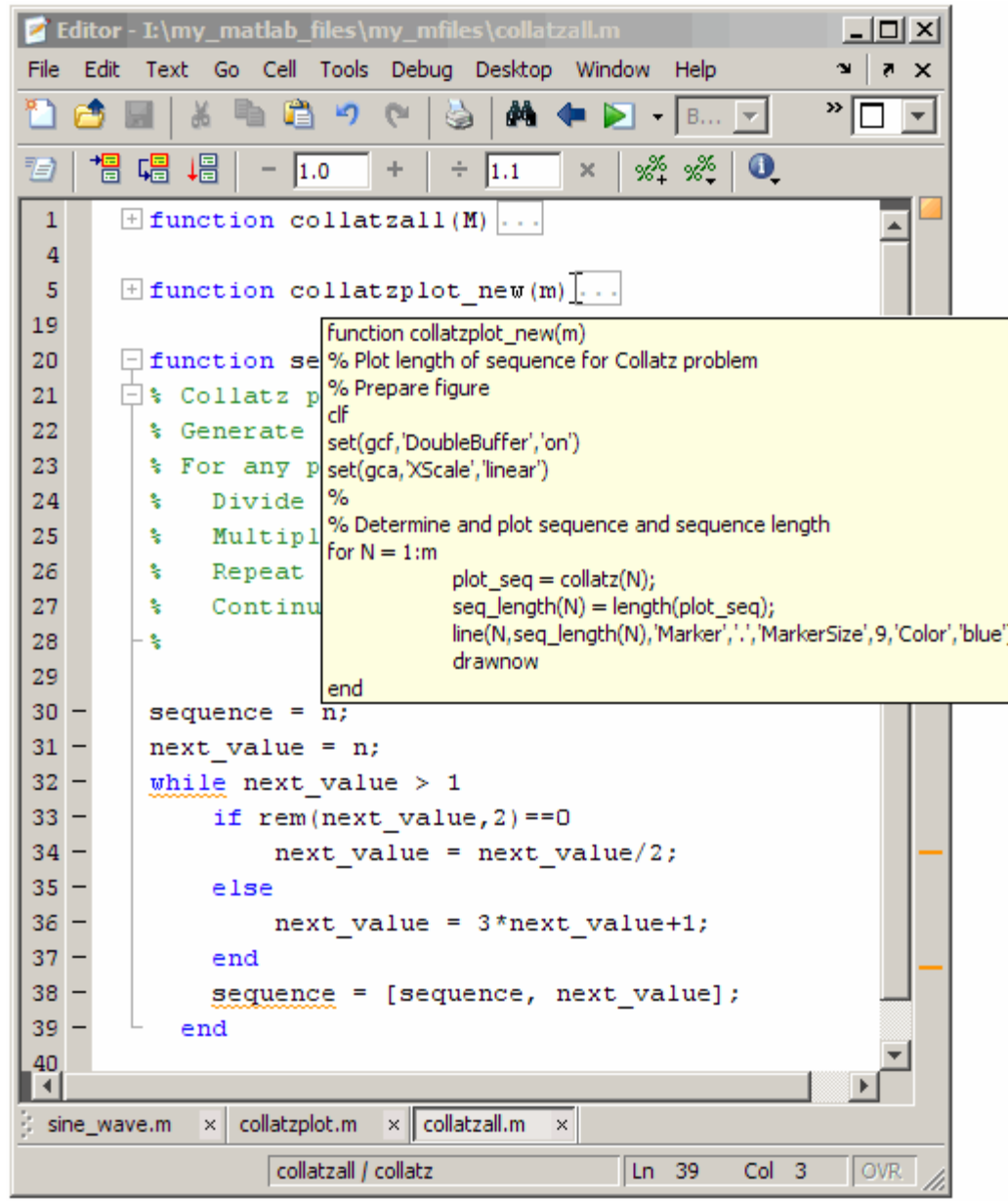
To expand or collapse all of the code in an M-file, place your cursor anywhere within the M-file, right-click, and then select **Code Folding > Expand All** or **Code Folding > Collapse All** from the context menu.

For information on the structure of an M-file, including a description of a function definition line and an H1 line, see Basic Parts of an M-File in the MATLAB Programming documentation

Viewing Folded Code in a Tooltip

You can view code that is currently folded by positioning the pointer over its ellipsis icon ... . The code displays in a tooltip. This enables you to get a quick view of the code without requiring you to unfold it.

The following image shows the tooltip that displays when the pointer is placed over the ellipsis icon on line 5 of collatzall.m when the collatzplot_new function is folded.


Editor - I:\my_matlab_files\my_mfiles\collatzall.m
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1
function collatzall(M) ...
function collatzplot_new(m) ...
function collatzplot_new(m)
% Plot length of sequence for Collatz problem
% Prepare figure
clf
set(gcf,'DoubleBuffer','on')
set(gca,'XScale','linear')
%
% Determine and plot sequence and sequence length
for N = 1:m
plot_seq = collatz(N);
seq_length(N) = length(plot_seq);
line(N,seq_length(N),'Marker','.','MarkerSize',9,'Color','blue')
drawnow
end
sequence = n;
next_value = n;
while next_value > 1
if rem(next_value,2)==0
next_value = next_value/2;
else
next_value = 3*next_value+1;
end
sequence = [sequence, next_value];
end
sine_wave.m
collatzplot.m
collatzall.m
collatzall / collatz
Ln 39 Col 3
OVR

Code Folding Behavior and Preferences

Be aware of the following:

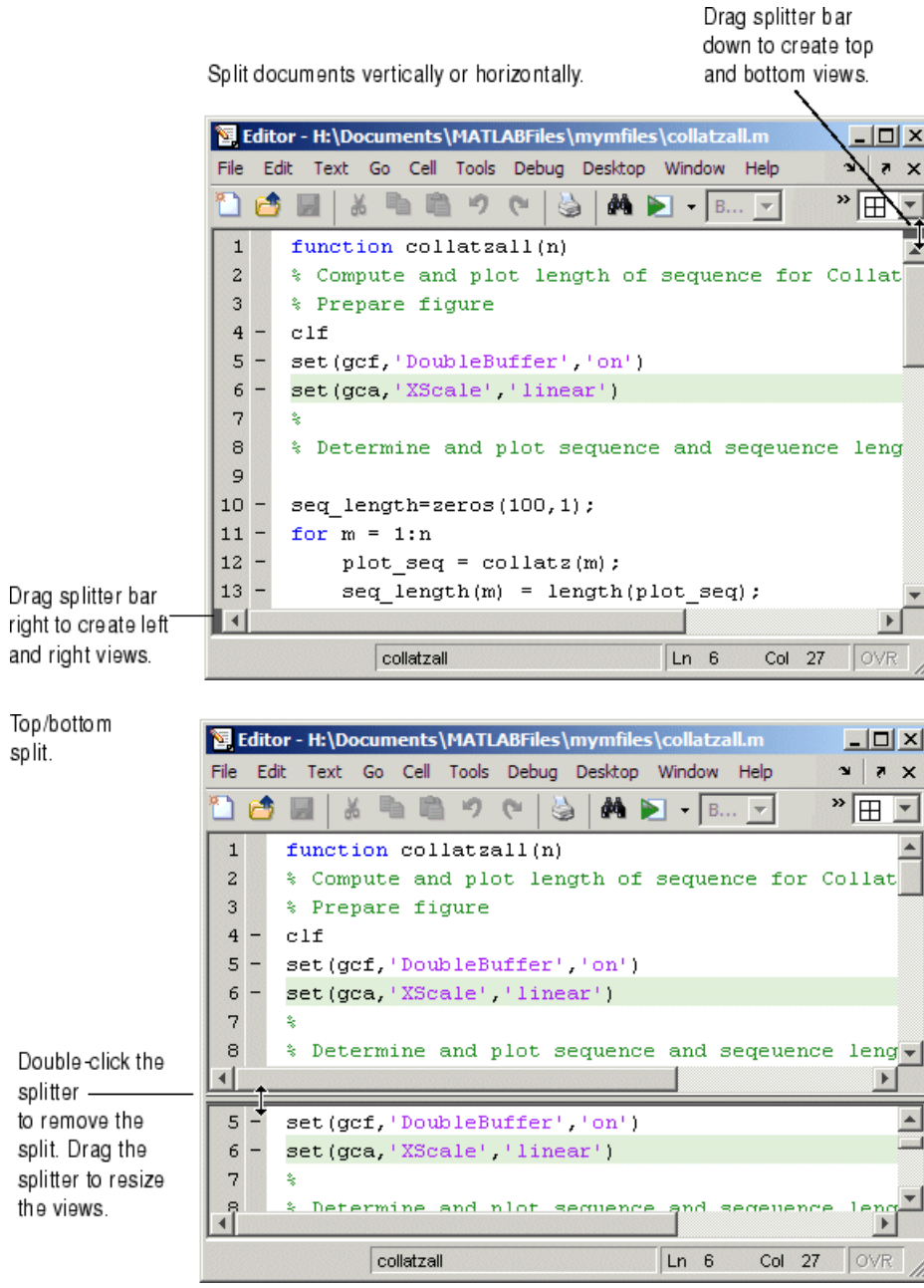
- You can change the current code folding settings, by selecting **File > Preferences > Editor/Debugger > Code Folding** and then clicking **Help** for assistance.
- By default, the first time you open an existing M-file in MATLAB release 7.5 (R2007b), code folding is enabled and all constructs are expanded.
- Constructs that are collapsed when you close an M-file remain collapsed when you reopen the file.
- If you copy a collapsed construct from one region of an M-file and paste it in another region, the construct is expanded in the pasted location.
- If you print a file with one or more collapsed constructs, those constructs are expanded in the printed version of the file.

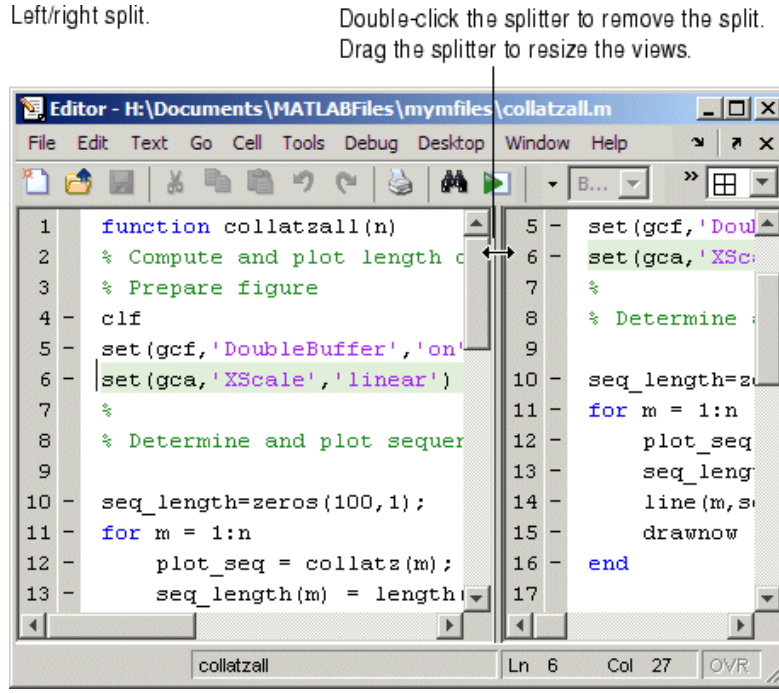
Split Screen Display

You can simultaneously display two different parts of a file in the Editor/Debugger. This makes it easy to compare different lines in a file or to copy and paste from one part of a file to another.

Split the screen horizontally by selecting **Window > Split Screen > Top/Bottom**. Or to split it vertically, select **Left/Right**.

Alternatively, when there is a scroll bar, split the document into top and bottom views by dragging the splitter bar, (as shown in the following illustration), down from above the vertical scroll bar. Similarly, to split into left and right views, drag the splitter bar from the left of the horizontal scroll bar. The pointer assumes a double-headed arrow shape when it is positioned on the splitter bar.


Split documents vertically or horizontally.
Drag splitter bar down to create top and bottom views.
Drag splitter bar right to create left and right views.
Top/bottom split.
Double-click the splitter to remove the split. Drag the splitter to resize the views.
Editor - H:\Documents\MATLABFiles\mymfiles\collatzall.m
File Edit Text Go Cell Tools Debug Desktop Window Help
function collatzall(n)
% Compute and plot length of sequence for Collat
% Prepare figure
clf
set(gcf,'DoubleBuffer','on')
set(gca,'XScale','linear')
%
% Determine and plot sequence and seqeuence leng
seq_length=zeros(100,1);
for m = 1:n
plot_seq = collatz(m);
seq_length(m) = length(plot_seq);
collatzall
Ln 6 Col 27
OVR



Adjust the size of the views by dragging the splitter. The pointer assumes an arrow shape when it is positioned on the splitter.

Only one view is active at any time, meaning, you will only see the cursor in one of the views. To change the active view use **Window > Split Screen > Switch Focus** or its keyboard equivalent, which is shown with the menu item. The cursor returns to its last position in that view.

Make changes to the document in either view. Both views of the file are always current, so you see the changes in either view.

You split each open document individually, so there can be multiple configurations at once. You can split some documents horizontally, others vertically, and leave others unsplit. When you open a document, it always opens unsplit, regardless of its split status when you last had it open.

You can remove a document split using any of these methods:

- Drag the splitter to an edge of the window.
- Double-click the splitter.
- Select **Window > Split > Screen > Off**.

See also “Summary of Actions for Arranging Documents” on page 2-11 for instructions to display multiple documents simultaneously.

Navigating in an M-File

In this section...

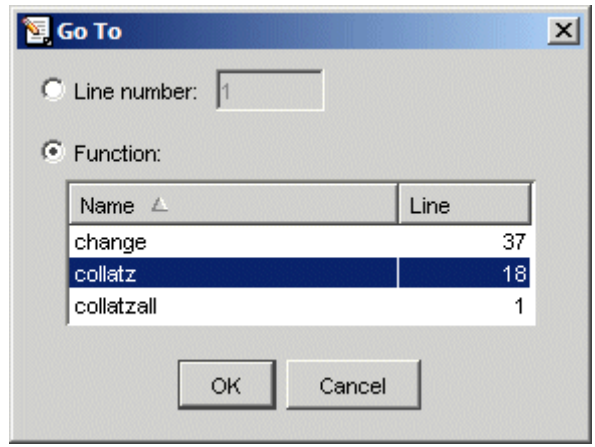
Note See also "Finding Text in Files" on page 6-49.

Going to a Line Number

Select **Go > Go To**. In the resulting **Go To** dialog box, select the **Line number** option, enter a line number, and click **OK**. The cursor moves to that line number in the current M-file.

Going to a Function (Subfunctions and Nested Functions)

To go to a function within an M-file (either a subfunction or a nested function), select **Go > Go To**. In the resulting **Go To** dialog box, select the **Function** option, and then select an entry from the list of subfunctions and nested functions in the file. Click **OK**.



Functions in the list appear alphabetically by name. To order them by their position in the file, click the **Line** column heading. The list does not include functions that are called from the M-file, but only shows lines in the current M-file that begin with a `function` statement.

Alternatively, click the Show Functions button on the toolbar. Then select the subfunction or nested function you want to go to from the list. The functions are listed in order of position in the file.

Note that the status bar shows the function and subfunction the current line is part of.

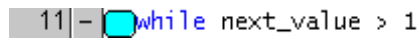
Going to a Cell

For M-file scripts that contain cells for rapid code iteration or publishing, the **Go To** dialog box lists cell titles.

Going to a Bookmark

You can set a bookmark at a line in a file in the Editor/Debugger so you can quickly go to the bookmarked line. This is particularly useful in long files. For example, while working on a line, if you need to look at another part of the file and then return, set a bookmark at the current line, go to the other part of the file, and then go back to the bookmark.

To set a bookmark, position the cursor anywhere in the line and select **Go > Set/Clear Bookmark**. A bookmark icon appears to the left of the line.



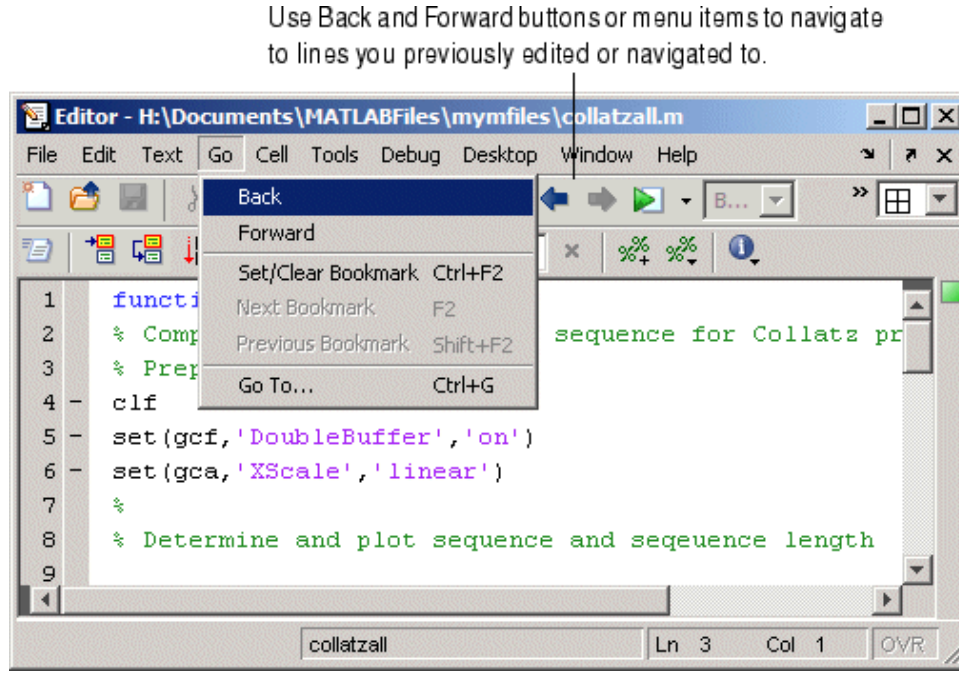
To go to a bookmark, select **Next Bookmark** or **Previous Bookmark** from the **Go** menu.

To clear a bookmark, position the cursor anywhere in the line and select **Go > Select/Clear Bookmark**.

Bookmarks are not maintained after you close a file.

Navigating Back and Forward in Files

Use **Go > Back** (and **Go > Forward**) to go to lines you previously edited or navigated to in a file. The feature goes to the lines in the sequence you accessed them. As an alternative to the menu items, use the Back and Forward buttons on the toolbar.



For example, if you open a file and make changes at lines 3, 9, and 6, use **Go > Back** to return to line 9, then 3, then 1, and then use **Go > Forward** to go from 1 to 3 to 9 to 6, and then return to 3. Detailed instructions to accomplish this are

1. Select **Go > Back** to return from line 6 to line 9.
2. Select **Go > Back** again to return to line 3.
3. Select **Go > Back** again to return to line 1, which is the first line you originally navigate to in a file by virtue of opening it.
4. Use **Go > Forward** to reverse the direction of the feature—select **Go > Forward** to navigate to line 3.
5. Select **Go > Forward** to navigate to line 9.
6. Reverse the direction of the feature again—select **Go > Back** to navigate to line 3.

Lines Navigated to Using Go Back

Use **Go > Back** and **Forward** to go to lines you previously edited or navigated to via these features:

Feature	Examples	Notes
Opening a file (first line in the file)	**File > Open**	None
Changes made using text-editing tools	**Delete** key, or **Text > Increase Indent**	Edits made to a selection of lines are represented by the first line in the selection. Changes made using **Cell > Insert Cell Divider** and **Cell > Insert Text Markup** are not considered as having been previously navigated to.
Changes made using Find and Replace	**Edit > Find and Replace**	Changes made using **Replace All** are not considered as having been previously navigated to.
Find features	**Edit > Find and Replace**, **Find Next**, **Find Previous**, and **Find Selection**	None
Incremental search	**Ctrl+S** and **Ctrl+R**	None
Show Function button		None
Opening a selection	**File > Open Selection**	None
Go to	**Go > Go To** line number, function, or cell title	None
Bookmark navigation	**Go > Next Bookmark** and **Previous Bookmark**	A line at which you **Set/Clear Bookmark** is not considered as having been previously navigated to.
Hyperlink access	From warnings or errors in the Command Window, from **Find Files** results, and from reports like the Profiler	None

Feature	Examples	Notes
Debugging navigation	Lines with breakpoints that were stopped at while running, and lines stepped to	A line at which you set a breakpoint is not considered as having been previously navigated to, unless it was actually stopped at during execution.
Cell mode navigation	**Cell > Next Cell** and **Previous Cell**, and **Cell > Evaluate Current Cell and Advance**	Lines accessed using **Cell > Evaluate Current Cell** are not considered as having been previously navigated to.

Interrupting the Sequence of Go Back and Forward

If you use **Go > Back** and **Go > Forward**, and then edit another line or navigate to another line using the list of features described in the above table, the **Go > Back** or **Go > Forward** feature sequence is interrupted. You can still go to the lines preceding the interruption point in the sequence, but you cannot go to any lines after that point. Any lines you edit or navigate to after interrupting the sequence are added to the sequence after the interruption point.

For example,

1. Open a file and edit lines 2, then 4, and then 6.
2. Use **Go > Back** to move back to line 4, and then back to line 2.
3. You could then **Go > Forward** to lines 4 and 6, or **Go > Back** to line 1.

   Instead, make an edit at line 3. Now you cannot **Go > Forward** to lines 4 and 6 and you can only **Go > Back** to line 2 and then line 1.

Closed Files and Behavior of Go Back and Forward

Go > Back and **Forward** do not go to lines in closed files.

Split Screen and Behavior of Go Back and Forward

When you have a split screen display, **Go > Back** and **Forward** go to the view in which the line was originally navigated to or edited in. If you remove

the split, **Go > Back** and **Forward** do not go to any lines that were visited in the lower (or right) view.

Opening a Selection in an M-File

You can open a subfunction, function, file, variable, or Simulink model from within a file in the Editor/Debugger. Position the cursor in the name and then right-click and select **Open Selection** from the context menu. Based on what the selection is, the Editor/Debugger performs a different action.

Selection	Action
Subfunction	Cursor moves to the subfunction within the current M-file. If no subfunction by that name is found in the current M-file, the Editor/Debugger runs the `open` function on the selection, which opens the selection in the appropriate tool, as shown for the other selection types in this table.
M-file or other text file	Opens in the Editor/Debugger.
Figure file (`.fig`)	Opens in a figure window.
Variable	Opens in the Array Editor.
Model	Opens in Simulink.
Other	If the selection is some other type, **Open selection** looks for a matching file in a private directory in the current directory and performs the appropriate action.

After selecting a name, you can also choose **Help on Selection** from the context menu to see documentation for the item. For example, select a function, right-click and select **Help on Selection**. The reference page for that function opens in a popup window, or if the reference page does not exist, the M-file help appears. For more information, see “Getting Pop-Up Help for Functions” on page 4-49

Finding Text in Files

In this section...

Finding Text in the Current File

Within the current file, select the text you want to find. From the **Edit** menu, select **Find Selection**. The next occurrence of that text is selected. Select **Find Selection** again (or **Find Next**) to continue finding more occurrences of the text.

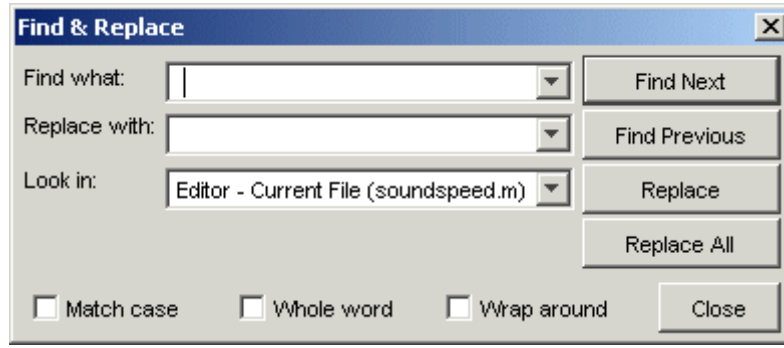
To find the previous occurrence of selected text (find backwards) in the current file, select **Find Previous** from the **Edit** menu. The previous occurrence of the text is selected. Repeat to continue finding the previous occurrences of the text.

Finding and Replacing Text in the Current File

You can search for specified text within multiple files, and then replace the text within a file.

Finding Text

To search for text in files, click the Find button in the Editor/Debugger toolbar, or select **Edit > Find and Replace**. Complete the resulting **Find & Replace** dialog box.



The search begins at the current cursor position. MATLAB finds the text you specified and highlights it. To find another occurrence, click **Find Next** or **Find Previous**, or use the keyboard shortcuts **F3** and **Shift+F3** (or **Command+F3** and **Command+Shift+F3** with Macintosh key bindings).

MATLAB beeps when a search for **Find Next** reaches the end of the file, or when a search for **Find Previous** reaches the top of the file. If you have **Wrap around** selected, it continues searching after beeping.

Use **F3** and **Shift+F3** to continue finding the specified text even after closing the **Find & Replace** dialog box. You can go to another file and find the specified text in it.

Change the selection in the **Look in** field to search for the specified text in other MATLAB desktop tools.

Replacing Text

After finding text using the **Find & Replace** dialog box, you can replace the text in the current file:

1. In the **Replace with** field, type the text that is to replace the found text.
2. Click **Replace** to replace the text currently selected, or click **Replace All** to replace all instances in the current file.

   The text is replaced. For **Replace All**, the number of instances that were replaced appears in the Editor/Debugger status bar.

3 To save the changes to the file, select **Save** from the **File** menu.

You can repeat this for multiple files.

Function Alternative for Finding Text

Use `lookfor` to search for the specified text in the first line of help for all M-files on the search path.

Finding Files or Text in Multiple Files

To find directories and filenames that include specified text, or whose contents contain specified text, use **Edit > Find Files**. For details, see "Finding Files and Content Within Files" on page 5-49.

Incremental Search

With the incremental search feature, the cursor moves to the next or previous occurrence of the specified text in the current file. It is similar to the Emacs search feature. Incremental search is also available in the Command Window — see "Incremental Search" on page 3-35.

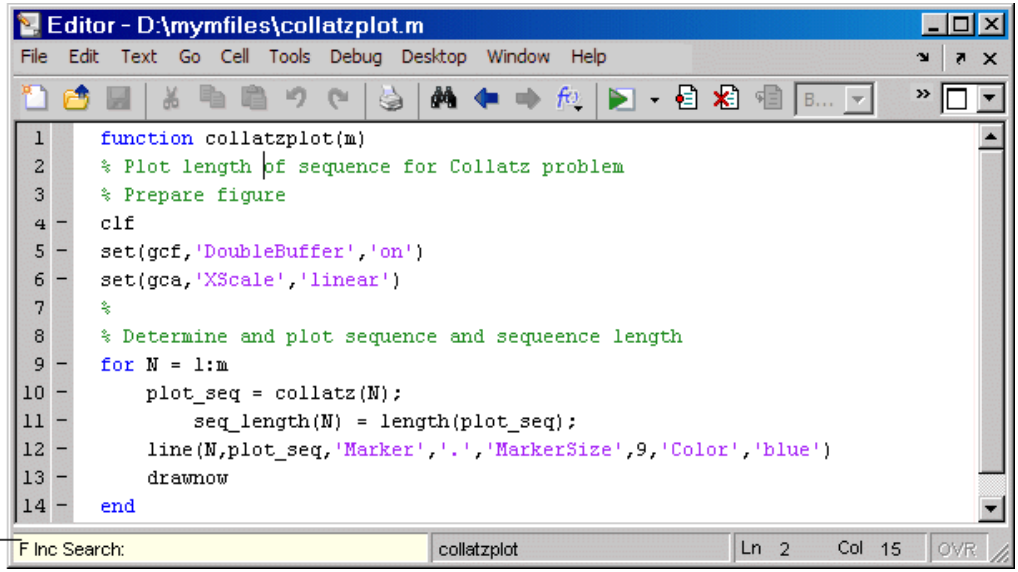
To use the incremental search feature in the Editor/Debugger, follow these steps:

1 Position the cursor where you want the search to begin.

2 How you begin the incremental search depends on your setting for theEditor/Debugger key bindings preference and in which direction you want to search:

- Press **Ctrl+S** to search forward or **Ctrl+R** to search backward for Emacs and Macintosh key bindings.
- Press **Ctrl+Shift+S** to search forward or **Ctrl+Shift+R** to search backward for Windows key bindings.

An incremental search field appears in the left side of the status bar of the current file window. **F Inc Search** means search *F*orward from the cursor. The field label is instead **R Inc Search** when you search backwards.



Incremental search field. **F** means search *Forward* from the cursor.

3 In the incremental search field, type the text you want to find. For example, type `plot`.

As you type the first letter, `p`, the first occurrence of that letter after the cursor is highlighted. In the example shown, the cursor is in the middle of line 2, so the first occurrence of `p`, the `p` in `problem` on line 2, is highlighted.

```
1   function collatzplot(m)
2   % Plot length of sequence for Collatz problem
3   % Prepare figure
```

Incremental search is case sensitive for uppercase letters. In the above example, searching for uppercase `P`, would instead find the `P` in `Prepare` on line 3.

When you type the next letter in the term you are searching for, the first occurrence of the term becomes highlighted. In the example, when you add the letter `l` to the `p` so that the incremental search field now has `pl`, the `pl` in `plot` on line 8 is highlighted. When you add `ot` to the term in the incremental search field, the whole word `plot` in line 8 is highlighted.

- If you mistype in the incremental search field, use the backspace key to remove the last letters and make corrections.
- After finding the `p`, press **Ctrl+W** to highlight the rest of the word found, in this case `plot`, which also puts the complete word in incremental search field.

4. To find the next occurrence of `plot` in the file, press **Ctrl+S**. To find the previous occurrence of the text, press **Ctrl+R**.

5. If MATLAB beeps, it either means the search is at the end or beginning of the file, or it means that the text was not found.

   - When the text is not found, `Failing` appears in the incremental search field. Modify the search term in the incremental search field and try again. Use **Ctrl+G** to automatically remove characters back to the last successful search. For example, if `plode` fails, **Ctrl+G** removes the `de` from the search term because `plo` does exist in the file.
   - When at the end or beginning of the file, press **Ctrl+S** or **Ctrl+R** again to wrap to the beginning (or end) of the file and continue the search. Use **Ctrl+G** after a finding a string to clear the search and return the cursor to the starting point.

6. To end the incremental search, press **Esc** or **Enter**, or any other noncharacter or number key except **Tab** or backspace.

   The incremental search field no longer appears in the status bar. The cursor is now located at the position where the string was last found.

If you press **Ctrl+S** or **Ctrl+R** after displaying the blank incremental search field, the search term from your previous incremental search appears in the field. Then the backspace key deletes the entire previous search term, rather than just the last letter.

Comparing Files — File Comparison Tool

In this section...

What Is the File Comparison Tool?

The File Comparisons tool identifies differences line by line between two files. Some other applications refer to this as a file diff. As an example, you can use this to easily compare an autosaved version of a file to the latest version.

Running the File Comparison Tool

To use the tool, follow these steps:

1. Open one of the files you want to compare in the Editor/Debugger.

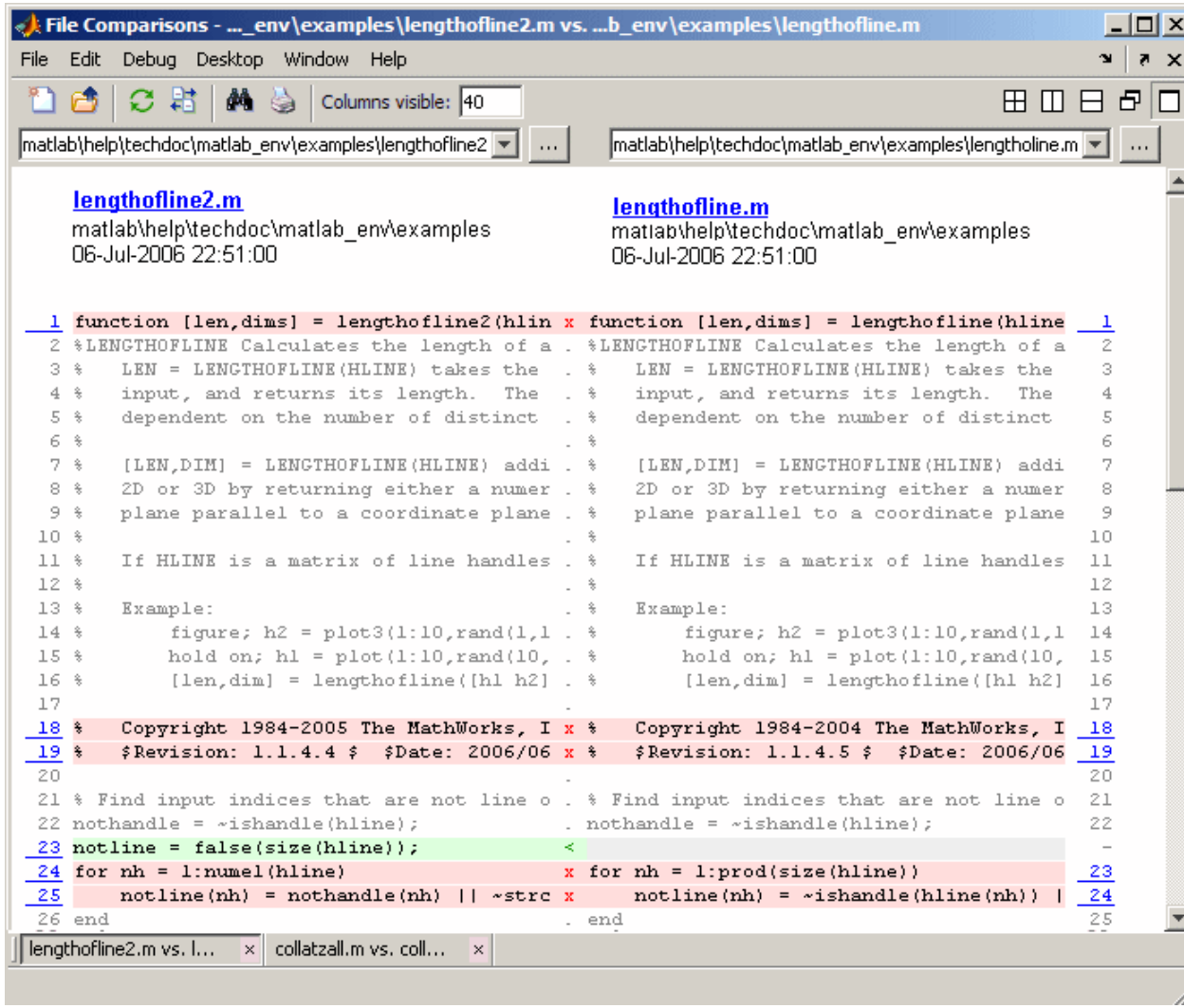
   To open the example file provided, `lengthofline.m`, run

   ```
   open(fullfile(matlabroot,'help','techdoc','matlab_env',...
    'examples','lengthofline.m'))
   ```

2. Select **Tools > Compare Against > Browse**. Navigate to the file you want to compare against, select the file, and click **Open**. To open the example file provided, select `lengthofline2.m`. Other options available are

- **Tools > Compare Against > Autosave Version** to compare the open file to the Editor/Debugger's automatic copy, *filename*.asv. For more information, see "Autosave" on page 6-63.
- **Tools > Compare Against Version on Disk** to compare an open file that has been changed but not saved to the saved version.

3 The File Comparisons tool opens, displaying the files side by side and highlighting lines that do not match. Pink highlighting and an x at the start of a line indicates that the content of the lines differs between the two files. Green highlighting and a > at the start of a line indicates a line that exists in the file presented on the right side of the page but not in the file presented on the left side of the page. Green highlighting and a < at the end of a line indicates a line that exists in the file presented on the left side of the page but not in the file presented on the right side of the page.



4 Use features of the File Comparisons tool to work with the results.

Increase or Decrease Line Lengths Shown

By default, the display is 60 columns wide for each file. To see the full line length for each file, use a high number for **Columns visible**, and drag the

vertical edges of the window to make it wider. With a narrower window size, if there are more columns shown in the left file but fewer in the right file, reduce the number for **Columns visible** to see enough in both sides.

Exchange Positions

To move the file on the left side to the right side and vice-versa, select **File > Swap Sides**, or click the equivalent toolbar button  .

Show Updated Files

After making changes to and saving the files in the Editor/Debugger, update the results in the File Comparisons tool by selecting **File > Refresh** or clicking the equivalent toolbar button  .

Find Text in Files

To find a phrase in the currently displayed files, select **Edit > Find**, or click the equivalent toolbar button. The resulting Find dialog box is the same as the one you use in the Command Window—for more information, see “Find Dialog Box” on page 3-34.

Compare to Other Files

You can replace an existing file in the tool. Drag a different filename from the Current Directory browser or Windows Explorer to the left or right side of the File Comparisons window, replacing the file currently shown there. Alternatively, type the path to a file or browse to find a file using the field below the File Comparisons toolbar. Another option is to use **File > Open**.

Perform New and View Previous Comparisons

You can perform another comparison by selecting **File > New File Comparison**. Supply the files to compare using the techniques described in “Compare to Other Files” on page 6-57. You can see the results of either pair of comparisons by selecting its entry in the document bar (as shown at the bottom of the window in the above example illustration).

Alternative Ways to Access the Tool

These are additional ways you can access the tool:

- From the MATLAB desktop, select **Desktop > File Comparisons**.
- From the Current Directory browser, select a file, right-click, and from the context menu, select **Compare Against**.
- For two files in the same directory, from the Current Directory browser, select the files, right-click, and from the context menu, select **Compare Selected Files**.

Supply the files to compare using the techniques described in “Compare to Other Files” on page 6-57.

Keyboard Shortcuts in the Editor/Debugger

Following is the list of keys that serve as shortcuts for using the Editor/Debugger. This list does not include shortcut keys (sometimes called hot keys) for menu items—you can view those on the menus. If you select the **Emacs** "Editor/Debugger Key Bindings" on page 3-45 preference, you can also use the **Ctrl+key** combinations shown. See also general desktop "Keyboard Shortcuts" on page 2-40.

Key or Mouse Action for Windows Preference	Additional Control Key for Emacs Preference	Key or Mouse Action for Macintosh Preference	Operation
↑	**Ctrl+P**	↑	Move to *p*revious line.
↓	**Ctrl+N**	↓	Move to *n*ext line.
Ctrl+Home	None	**Cmd+Home**	Move to top of file.
Ctrl+End	None	**Cmd+End**	Move to end of file.
Ctrl+↑	None	**Home**	Scroll up without moving cursor position (with cell mode disabled). Move to top of current cell or top of previous cell (with cell mode enabled).
Ctrl+↓	None	**End**	Scroll down without moving cursor position (with cell mode disabled). Move to top of next cell (with cell mode enabled).
Page Down	**Ctrl+V**	**Page Down**	Move down one screen.
Page Up	**Alt+V**	**Page Up**	Move up one screen.
←	**Ctrl+B**	←	Move *b*ack one character.
→	**Ctrl+F**	→	Move *f*orward one character.
Ctrl+←	None	**Option+**←	Move *l*eft one word.
Ctrl+→	None	**Option+**→	Move *r*ight one word.

Key or Mouse Action for Windows Preference	Additional Control Key for Emacs Preference	Key or Mouse Action for Macintosh Preference	Operation
Home	**Ctrl+A**	**Cmd+←**	Move to beginning of line.
End	**Ctrl+E**	**Cmd+→**	Move to *e*nd of line.
Delete	**Ctrl+D**	**Forward Delete**	Delete character after cursor.
Backspace	none	**Delete**	Delete character before cursor.
None	**Ctrl+K**	None	Cut contents (*k*ill) to end of line.
Double-click	None	Double-click	Select current word. To select additional words, hold mouse after second click and continue dragging left or right.
Triple-click	None	Triple-click	Select current line. To select additional lines, hold mouse after second click and continue dragging up or down.
Ctrl+Shift+←	None	**Option+Shift+←**	Select word to the left
Ctrl+Shift+→	None	**Option+Shift+→**	Select word to the right.
Shift+Home	None	**Cmd+Shift+←**	Select to beginning of line.
Shift+End	None	**Cmd+Shift+→**	Select to end of line.
Shift+Page Up	**Ctrl+Shift+V**	**Shift+Page Up**	Select one screen up.
Shift+Page Down	**Alt+Shift+V**	**Shift+Page Down**	Select one screen down.
Ctrl+Shift+Home	None	**Cmd+Shift+Home**	Select to top of file.
Ctrl+Shift+End	None	**Cmd+Shift+End**	Select to end of file.
Shift+Enter	None	**Shift+Enter**	Add a new line that is not indented.

Key or Mouse Action for Windows Preference	Additional Control Key for Emacs Preference	Key or Mouse Action for Macintosh Preference	Operation
Insert	None	None	Change to overwrite mode from insert mode, or change to insert mode from overwrite mode. View current mode in the status bar: OVR is gray for insert mode. In overwrite mode, what you type replaces existing text, and the cursor is a wide block. (Not supported on Macintosh platforms.)
Shift+F5	None	**Shift+F5**	Exit debug mode. Equivalent to typing `dbquit`. The Command Window displays the standard prompt `>>`.

Saving, Printing, and Closing Files in the Editor/Debugger

In this section...

Saving Files

After making changes to an file, you see an asterisk (*) next to the filename in the title bar of the Editor/Debugger. This indicates there are unsaved changes to the file.

To save the changes, use one of the **Save** commands in the **File** menu:

- **Save** — Saves the file using its existing name. If the file is newly created, the **Save file as** dialog box opens, where you assign a name to the file before saving it. Another way to save is by using the Save button on the toolbar. If the file has not been changed, **Save** is grayed out, but you can instead use **Save As** from the **File** menu to save to a different filename.
- **Save As** — The **Save file as** dialog box opens, where you assign a name to the file and save it. By default, if you do not type an extension, MATLAB automatically assigns the `.m` extension to the filename. If you do not want an extension, type a `.` (period) after the filename.
- **Save All** — Saves all named files to their existing filenames. For all newly created files, the **Save file as** dialog box opens, where you assign a name to each untitled file and save it.

You cannot save an M-file while in debug mode. If you try to, MATLAB displays a dialog box asking if you want to exit debug mode and then save the file. While debugging, you can execute sections of an M-file even though there are unsaved changes — see “Running Sections in M-Files That Have Unsaved Changes” on page 6-125.

Note Save any M-files you create and any M-files from The MathWorks that you edit in a directory that is not in the *matlabroot*/toolbox directory tree. If you keep your files in *matlabroot*/toolbox directories, they can be overwritten when you install a new version of MATLAB. Also note that locations of files in the *matlabroot*/toolbox directory tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to *matlabroot*/toolbox directories using an external editor or add or remove files from these directories using file system operations, run `rehash toolbox` before you use the files in the current session. If you make changes to existing files in *matlabroot*/toolbox directories using an external editor, run `clear functionname` before you use the files in the current session. For more information, see `rehash` or "Toolbox Path Caching in MATLAB" on page 1-17.

Autosave

As you make changes to a file in the Editor/Debugger, every five minutes the Editor/Debugger automatically saves a copy of the file to a file of the same name but with an `.asv` extension. The autosave copy is useful if you have system problems and lose changes made to your file. In that event, you can open the autosave version, `filename.asv`, and then save it as `filename.m` to use the last good version of `filename`. For example, if you edit `filename.m` and do not save it for five minutes, MATLAB saves the file including the unsaved changes, to `filename.asv`.

Use autosave preferences to turn the autosave feature off or on, to specify the number of minutes between automatic saves, and to specify the file extension and location for autosave files. For details, see in the online documentation.

If the file you are editing is in a read-only directory and the autosave preference for location is the source file directory, an autosave copy of the file is *not* made.

Deleting Autosave Files. By default, autosave files are not automatically deleted when you delete the source file. To keep autosave to M-file relationships clear and current, it is a good practice when you rename or remove an M-file to delete or rename its corresponding autosave file.

There is a preference to **Automatically delete autosave files**. With this preference selected, when you close an M-file in the Editor/Debugger, MATLAB automatically deletes the corresponding autosave file.

Accessing Your Source Control System

If you use a source control system for M-files, you can access it from within the Editor/Debugger using **File > Source Control**. For more information, see Chapter 9, “Source Control Interface”.

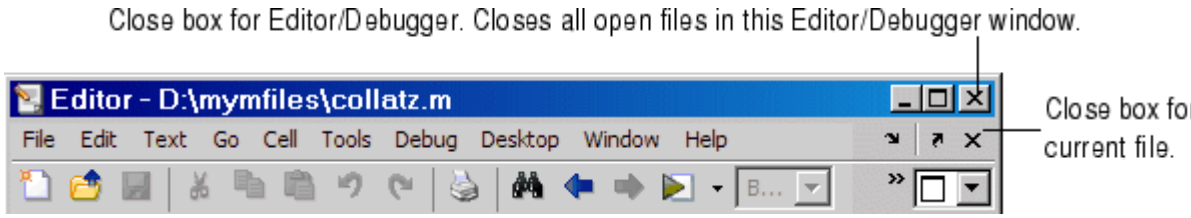
Printing M-Files

To print an entire M-file, select **File > Print**, or click the Print button on the toolbar. To print the current selection, select **File > Print Selection**. Complete the standard print dialog box that appears.

Specify printing options for the Editor/Debugger by selecting **File > Page Setup**. For example, you can specify printing with a header. For more information, see “Printing and Page Setup Options for Desktop Tools” on page 2-52.

Closing M-Files

To close the current M-file, select **Close filename** from the **File** menu, or click the Close box in the Editor/Debugger menu bar. This is different from the Close box in the titlebar of the Editor/Debugger, which closes all open files in that Editor/Debugger window.



To close all files within the Editor/Debugger, select **Window > Close Editor Documents**. This does not close any files undocked from the Editor/Debugger. The Editor/Debugger remains open with no files in it.

If each file is open in a separate window, close all the files at once using the **Close All Documents** item in the **Window** menu. Note that this also closes desktop documents of all types, including Array Editor documents.

When you close a file that has unsaved changes, you are prompted to save the file. If you do not want to be prompted, hold **Ctrl** and click the Close box. The prompt will not appear and the document will close without saving any unsaved changes.

Running M-Files in the Editor/Debugger

In this section...

Running M-Files with No Input Arguments in the Editor/Debugger

In the Editor/Debugger, to run a script M-file, or a function M-file that requires no input arguments, click the Run button  on the toolbar. The button's tooltip includes the name of the file to be run, which is useful when you have more than one file open. Alternatively, select **Debug > Run filename**.

If the file is not in a directory on the search path or in the current directory, a dialog box appears, presenting you with options that allow you to run the file. You can either change the current directory to the directory containing the file, or you can add the directory containing the file to the search path.

If the file has unsaved changes, running it from the Editor/Debugger automatically saves the changes before running. In that event, the menu item is **Save File and Run filename**.

If the M-file is a script, you can view the value of a variable in the file, which is called a *datatip* (like a tooltip for data). You need to set the preference to show datatips in edit mode — select **File > Preferences > Display**, and for

General Display Options, select the check box for **Enable datatips in edit mode**.

Using Configurations — Running M-Files with Input Arguments in the Editor/Debugger

In the Editor/Debugger, you can provide values for a function's input arguments using a configuration, and then run that configuration to use the assigned values. When you are editing a function M-file, use a configuration as an alternative to running the function in the Command Window. You can associate multiple configurations with an M-file, for different input values. MATLAB saves the configurations between sessions.

For example, the function `collatzplot_new.m`, which computes and plots the Collatz sequence for any given positive integer, requires you to specify the integer as an input value. You cannot simply run `collatplot_new.m` in the Editor/Debugger because the input value is not defined. One way to specify the input value is to run the M-file in the Command Window. Configurations allow you to run `collatzplot_new(`*`specific value`*`)` in the Editor/Debugger.

You can also use configurations to provide preparatory or setup information prior to running an M-file, whether it takes input arguments or not.

Create and Run a Configuration for an M-file

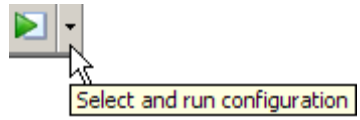
Follow these steps to create and run a configuration for an M-file in the Editor/Debugger. These steps specify Editor/Debugger toolbar buttons, but you can also use equivalent items in the **Debug** menu.

1. Open the file you want to run in the Editor/Debugger. For example, open `collatzplot_new.m` by running

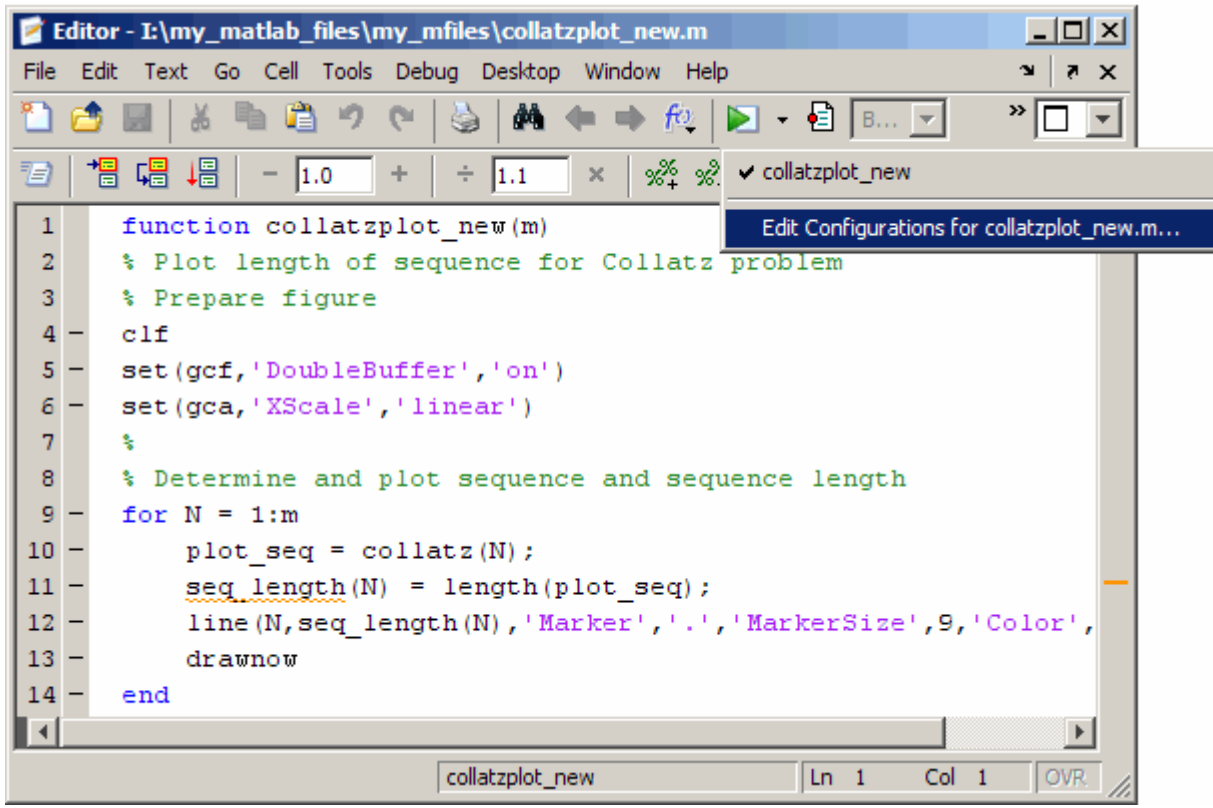
   ```
   cd ([matlabroot '/help/techdoc/matlab_env/examples'])
   edit collatzplot_new.m
   ```

   To work with `collatzplot_new.m` on your system, save the file to a directory for which you have write permission. In the example, the file is saved to `I:\my_matlab_files\my_mfiles\collatzplot_new.m`

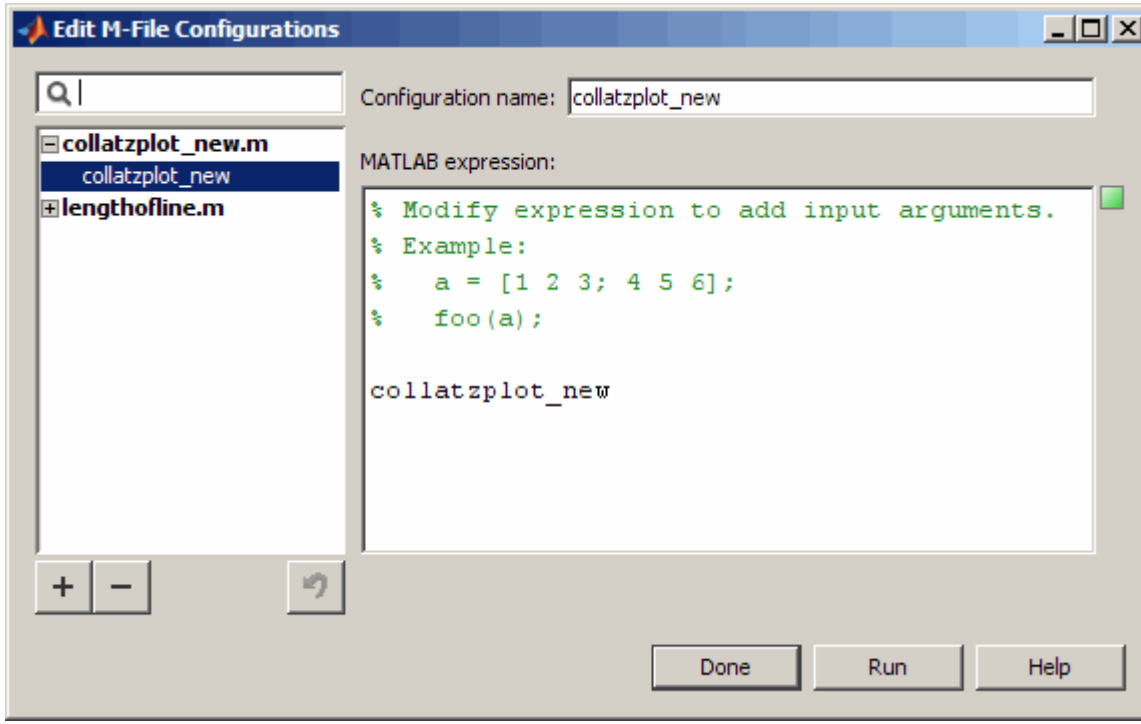
2 Click the down arrow on the Run button in the Editor/Debugger toolbar



and select **Edit Configurations for *filename***, where filename in this example is `collatzplot_new.m`.

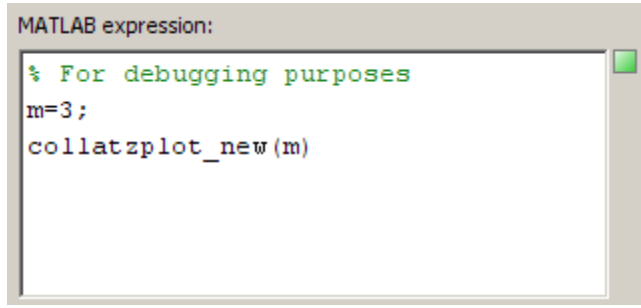


The Edit M-File Configurations dialog box opens, with a default configuration template for `collatzplot_new.m`.



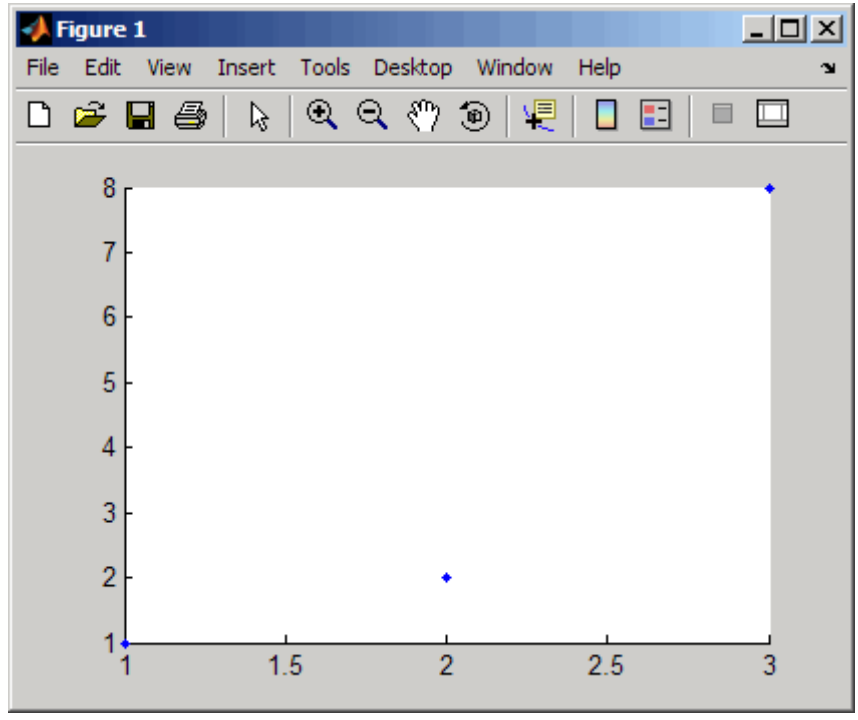
3 In the dialog box, enter MATLAB statements in the **MATLAB expression** area of the dialog box, specifying what you want to run. Delete the existing comments or replace them with comments relevant to your configuration. To undo and redo, use the keyboard shortcuts for your platform, such as **Ctrl+Z** and **Ctrl+Y** for Windows.

In this example, set `m` equal to `3`, a small value useful for debugging purposes. Complete the statement to run `collatzplot_new(m)`.



The MATLAB expression area provides syntax highlighting and shows M-Lint messages, similar to the Editor/Debugger.

4 To ensure your configuration runs as expected, click **Run** to execute the statements in the **MATLAB expression** area of the dialog box. In this example, `collatzplot_new(3)` runs, and a figure window displays the plot.



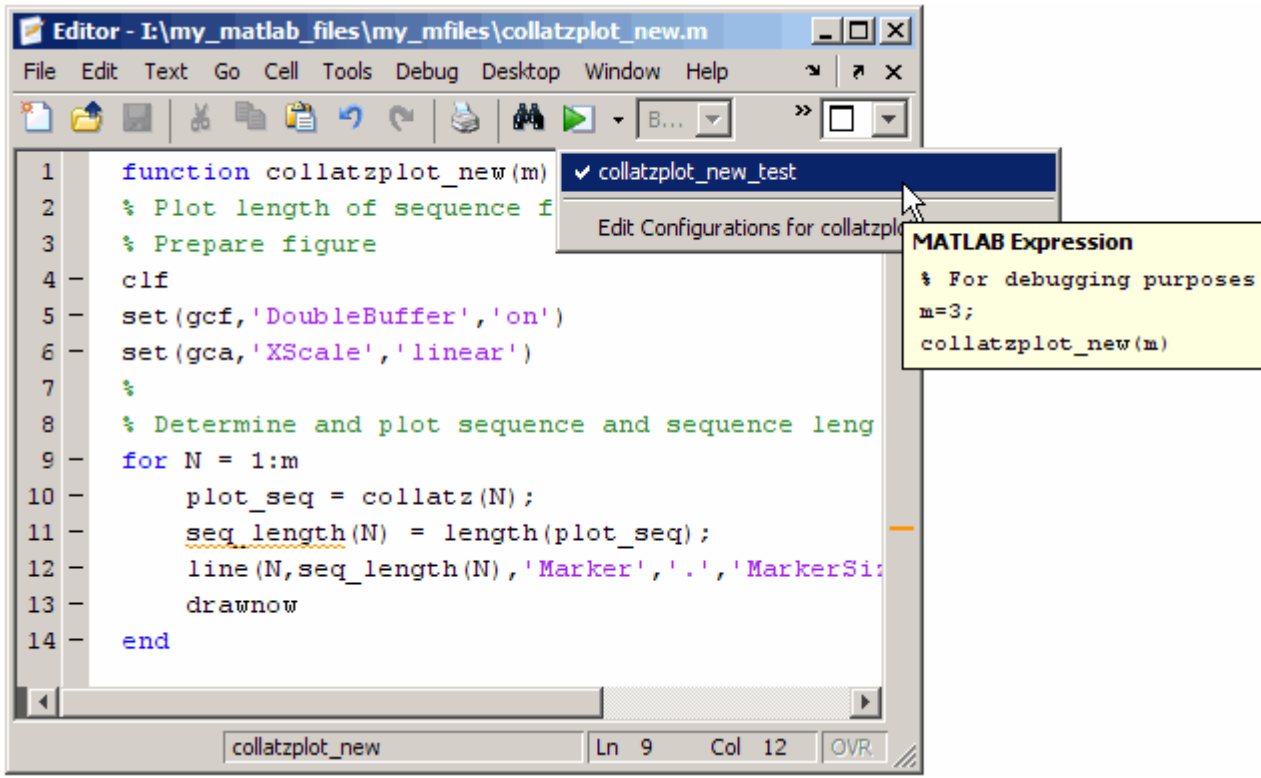
5 You can modify the statements in the **MATLAB expression** area of the dialog box and click **Run** to see the results of the changes. You can also modify the M-file and save the changes while the Edit M-File Configurations dialog box is open, and then click **Run** to see the results of the M-file changes.

6 You can assign a name using **Configuration name** in the Edit M-File Configurations dialog box. By default, the configuration name is the M-file name. If you expect to create multiple configurations for an M-file, assign each a name that helps you identify the configuration. In this example, name the configuration `collatzplot_new_test`.

7 To close the Edit M-File Configurations dialog box, click **Done**. MATLAB saves the configuration and its association with the M-file.

8 After creating a configuration, you can view and run the configuration without opening the Edit M-File Configurations dialog box.

In the Editor/Debugger toolbar, click the down arrow on the Run button  and position the pointer on a configuration name. MATLAB displays a tooltip showing the configuration's expression so you can see what will run.



To run the configuration, select the configuration name. MATLAB runs the expression you specified in the configuration. For example, select `collatzplot_new_test`, and MATLAB runs `collatzplot_new(3)`, as specified in step 3. You can modify the M-file, save it, and run the configuration from the toolbar to see the effects of the M-file changes.

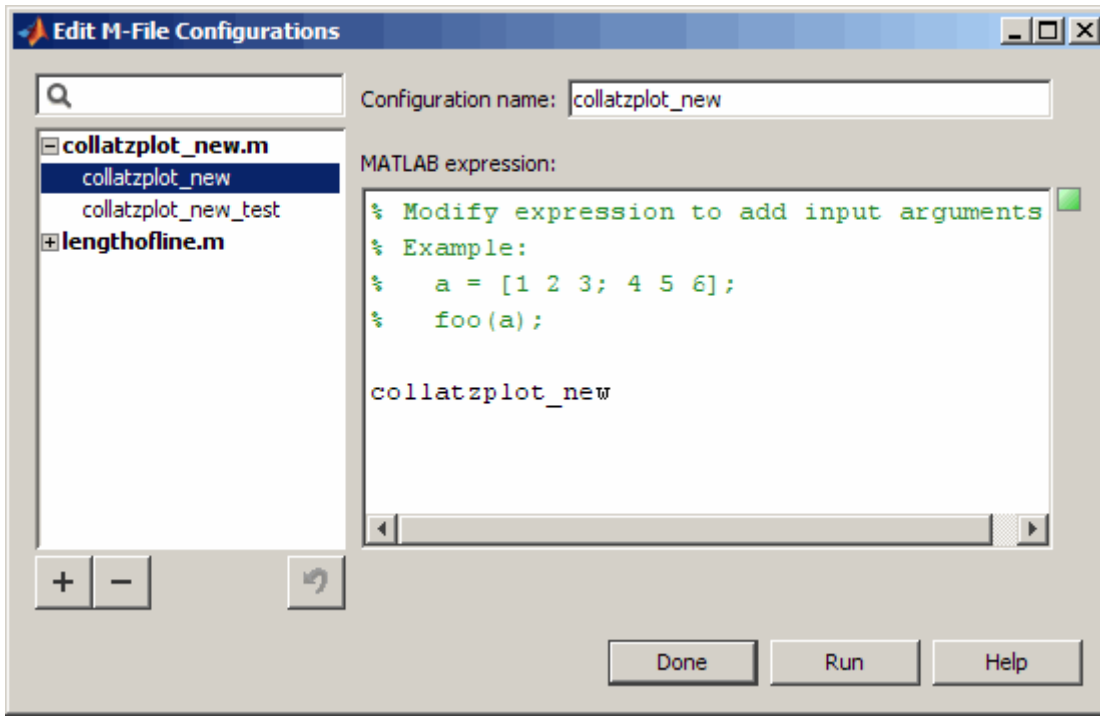
Create and Run Multiple Configurations for an M-File

You can create multiple configurations for a given M-file, allowing you to run with different values for input arguments, each for a different purpose. Create a named configuration for each purpose, all associated with the M-file. Then any time you open the M-file, chose and run the configuration you want. For example, for `collatzplot_new(m)` you might use three values for `m` and have three configurations:

- Small value, e.g., 3, for debugging and testing
- Realistic value, e.g., 200 or more, for a specific project
- Random value to observe changes

1. Open the Edit M-File Configurations dialog box. Select the M-file to which you want to add a configuration, or select a configuration associated with that M-file. Click the Add button  (under the list of M-files and configurations). MATLAB creates a new default configuration template, in this example, `collatzplot_new`.

   The example shows `collatzplot_new` and its default expression, as well as one previously created configuration associated with `collatzplot_new.m`, `collatzplot_new_test`.



2. In the Edit M-File Configurations dialog box, modify, run, and name the new configurations as you did for the initial configuration,

`collatplot_new_test`, described in "Create and Run a Configuration for an M-file" on page 6-67.

For example, rename `collatzplot_new` to `collatzplot_new_largevalue`, and replace the default template expression with

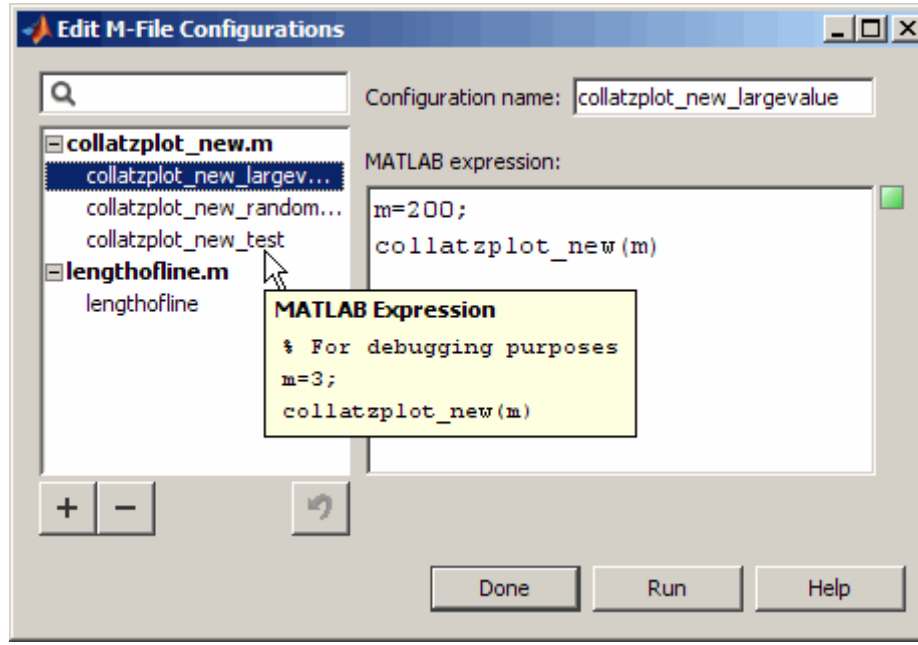
```
m=200;
collatzplot_new(m)
```

To create another configuration, click the Add button  . Rename `collatzplot_new` to `collatzplot_new_random` and replace the default template expression with

```
% Random value
m=int16(rand*50);
collatzplot_new(m)
clear all
```

3. Select a configuration in the listing to see and modify its expression, or to rename the configuration. Click the expander next to an M-file name (**+** and **-** icon on Windows) to see or hide all the configurations associated with that M-file.

4. To get a quick view of the expression in a configuration, position the pointer on the name of a configuration without selecting it. In this example, `collatzplot_new_largevalue` is selected and you can edit its expression or name. The pointer is positioned on `collatzplot_new_test` and you can see the statements in it.



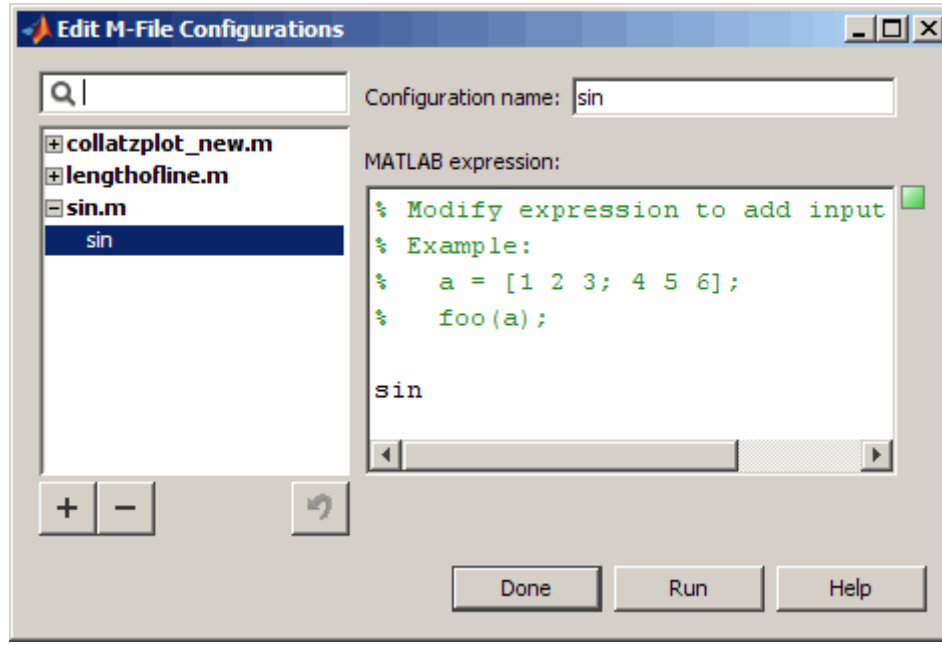
5 Click **Done** to close the Edit M-File Configurations dialog box.

Find Configurations

1 Open any M-file in Editor/Debugger. For example, open the MATLAB function `sin`.

2 Open the Edit M-File Configurations dialog box. MATLAB automatically creates a default configuration for `sin.m`, if none exists.

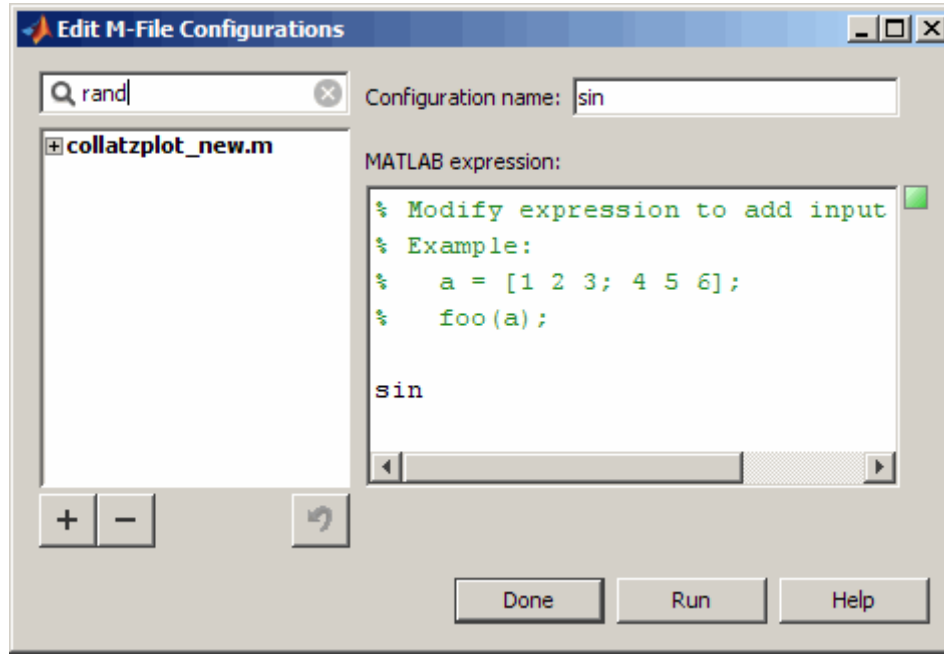
In the left panel, MATLAB displays a list of all M-files containing configurations.



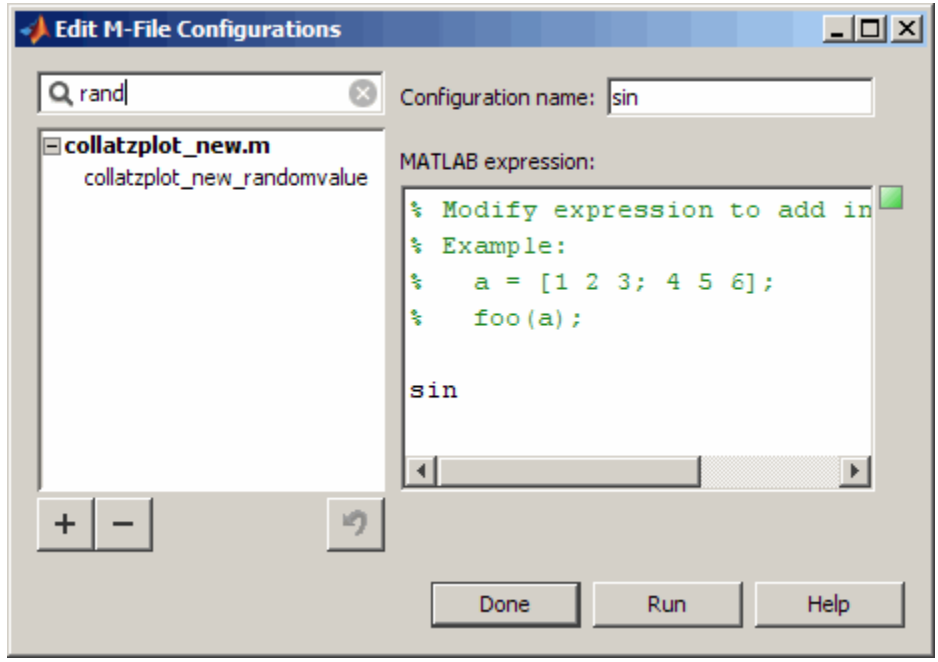
3 Type a term in the filter field  to find an M-file or configuration by name.

MATLAB displays only those M-files whose names contain the term, or whose associated configurations contain the term in their name. As you type, MATLAB filters out files and configurations that do not contain the term.

For example, type `rand`. In this example, only one M-file, `collatzplot_new.m`, has a configuration that contains the term `rand`.



4 Expand the M-file name to see associated configurations whose names contain the term you entered in the filter field. In this example, click the expander (**+** on Windows) for `collatzplot_new.m` to see configurations whose names contain `rand`. If you cannot view the entire name of a configuration, drag the separator bar to the right of the list, making the left panel wider. For the example, there is only one configuration, `collatzplot_new_randomvalue`.



5 To see the expression in that configuration, select the configuration, or position the pointer over the name.

6 As you type additional letters in the filter field, fewer M-files remain in the list of results. Use the backspace key to modify the term. If there are no M-files or configurations containing the term, the list will be empty.

7 To clear the filter and show all M-files with configurations, click the clear button in the filter field.

Remove Configurations

If you no longer need a configuration because you do not use it or because you deleted the M-file with which it is associated, it is a good practice to delete the configuration.

1 Open any M-file in the Editor/Debugger.

2 Open the Edit M-File Configurations dialog box.

3 Select the configuration you want to delete.

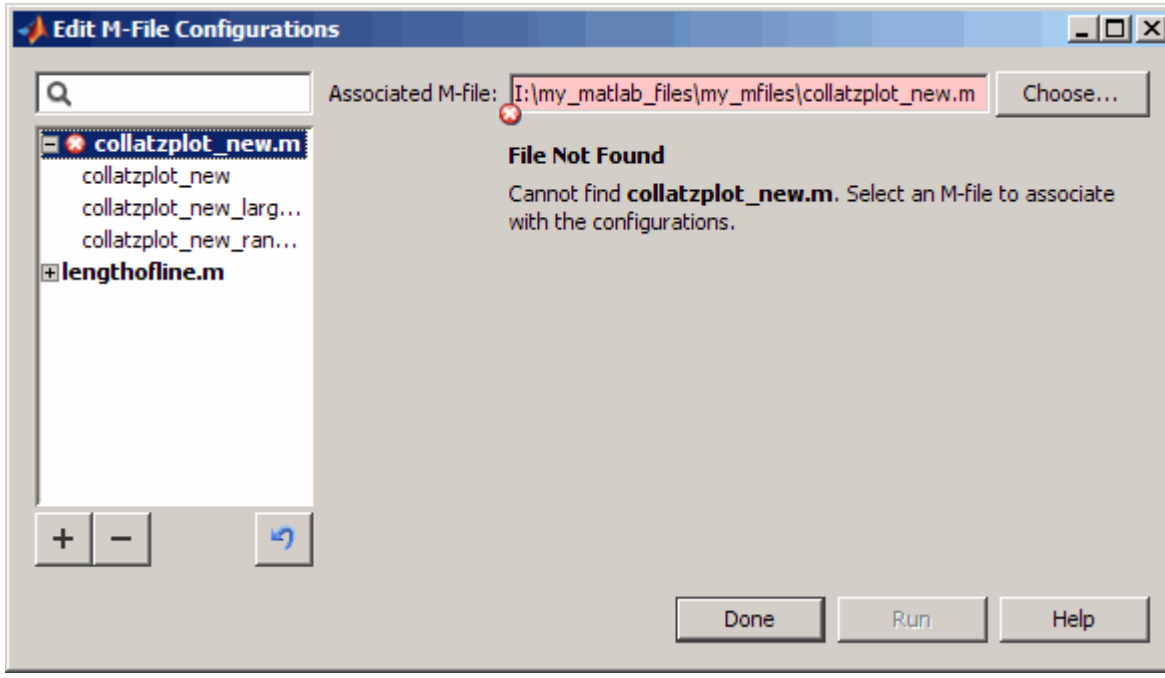
4 Click the Remove button  .

5 To undo the last deletion, click the Undo button  . You cannot undo the last deletion after having closed the Edit M-File Configuration dialog box.

Reassociate and Rename Configurations

Each configuration is associated with a specific M-file. If you move or rename the M-file, you need to redefine the association. If you delete an M-file, you might want to delete the associated configurations, or associate them with a different M-file. You might also need to modify the statements in the configurations so they will run.

When MATLAB cannot associate a configuration with an M-file, the Edit M-File Configurations dialog box displays the M-file name in red, displays a **File Not Found** message, and allows you to find the M-file to which you want to associate the configuration. In this example, MATLAB cannot find the file `collatzplot_new.m`, which has three configurations associated with it. For this example, `collatzplot_new.m` had been renamed to `collatzplot.m`, so the configurations associated with `collatzplot_new.m` need to be reassociated.



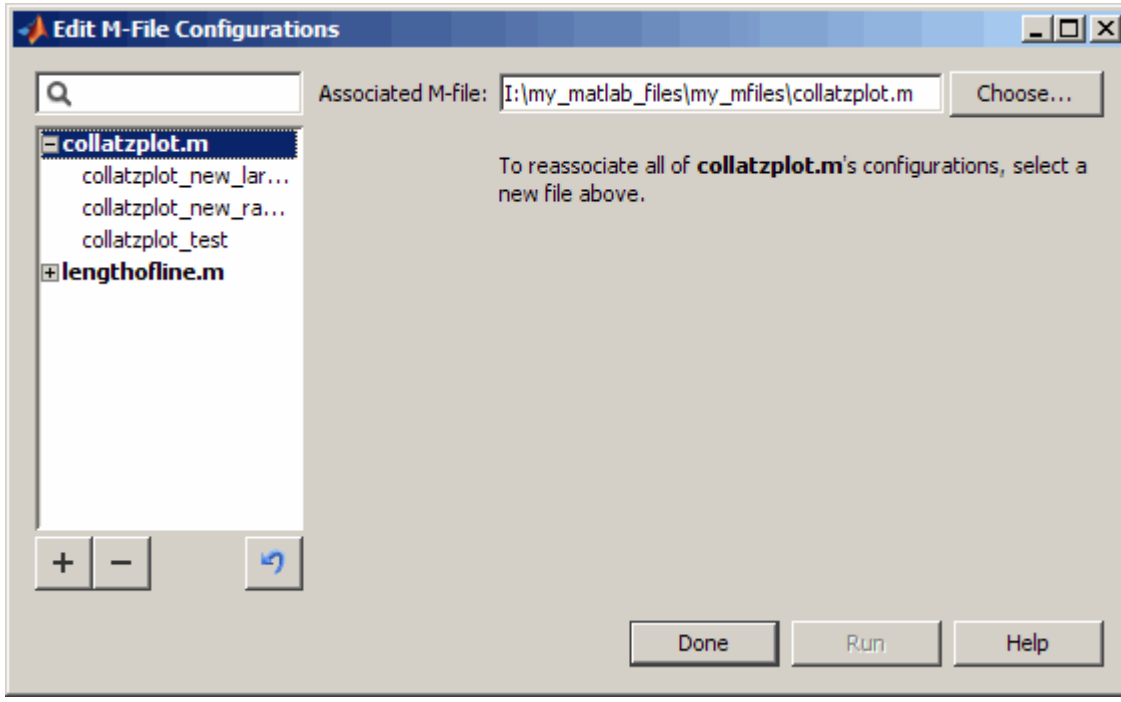
To reassociate configurations

1. In the list of configurations (left pane), select the M-file. The **Associated M-file** displays the full path to the M-file that was associated with the configurations. Click **Choose**.

2. In the resulting Open dialog box, navigate to and select the M-file with which you now want to associate the configurations. Click **Open**

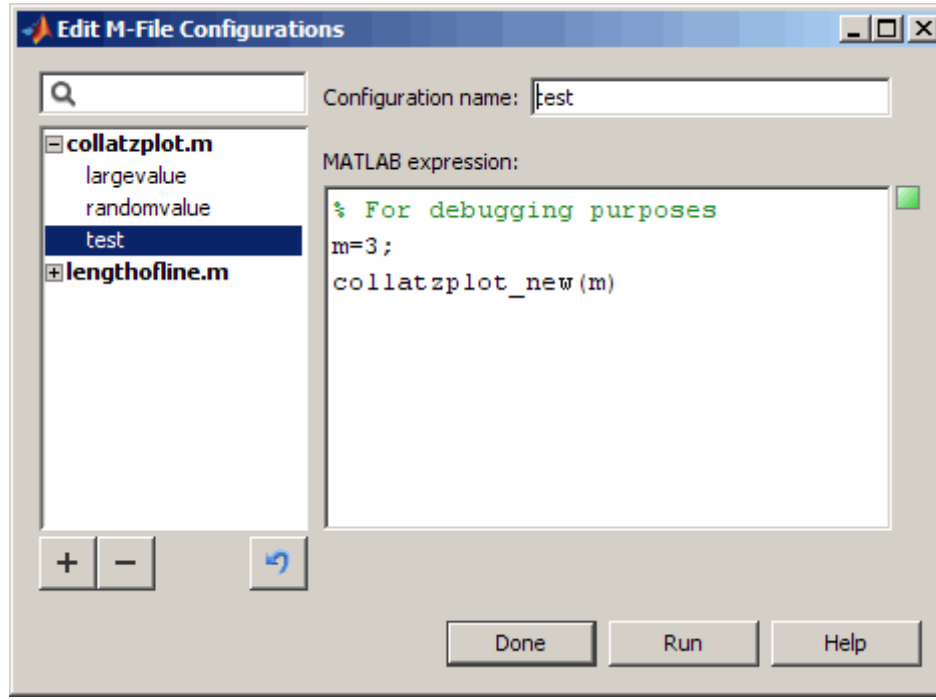
   In this example, you want to associate the configurations with `collatzplot.m`; select `collatzplot.m`.

3. In the Edit M-File Configurations dialog box, the **Associated M-file** value reflects the change you made and the **File Not Found** message no longer appears.

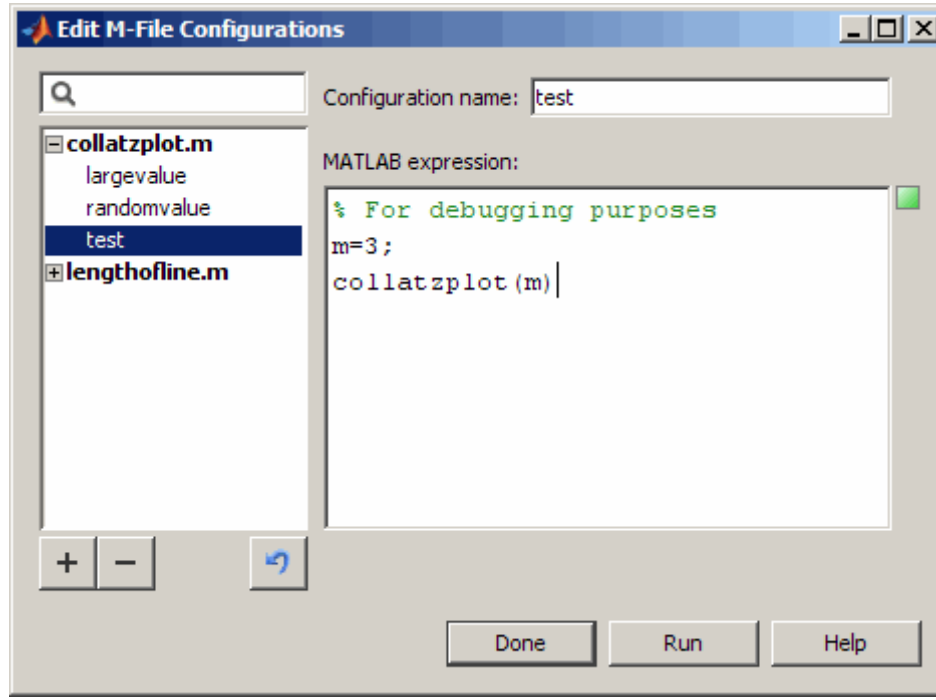


4 You might want to rename the configurations to be consistent with the new M-file name, or at least to not reflect the former M-file name. This is not required, but it is a good practice. To do so, select a configuration from the list in the left pane. In the right pane, edit the value for **Configuration name**. Repeat for all configurations associated with the M-file.

   In this example, remove `collatzplot_new` from the start of each configuration name.



5 For an M-file name change, you might need to modify the configuration statements to run correctly. For this example, modify the `collatzplot_new(m)` statement in each configuration to use `collatzplot(m)`.



See Also — Other Ways to Run M-Files from the Editor/Debugger

- See “Running an M-File with Breakpoints” on page 6-111 for additional information about running M-files while debugging.
- While debugging, you can execute sections of an M-file even though there are unsaved changes—see “Running Sections in M-Files That Have Unsaved Changes” on page 6-125.
- You can execute M-files one section at a time and quickly modify values incrementally using the toolbar—for more information, see “Using Cells for Rapid Code Iteration and Publishing Results” on page 6-133.

Finding Errors, Debugging, and Correcting M-Files

This section introduces general techniques for finding errors and using M-Lint automatic code analyzer to detect possible improvements in M-files. It then illustrates MATLAB debugger features found in the Editor/Debugger, as well equivalent Command Window debugging functions, using a simple example.

There are two kinds of errors:

- Syntax errors — For example, misspelling a function name or omitting a parenthesis.
- Run-time errors — These errors are usually algorithmic in nature. For example, you might modify the wrong variable or code a calculation incorrectly. Run-time errors are usually apparent when an M-file produces unexpected results. Run-time errors are difficult to track down because the function's local workspace is lost when the error forces a return to the MATLAB base workspace. The process of isolating and fixing these run-time problems is referred to as *debugging*.

In addition to finding and fixing problems with your M-files, you might want to improve the performance and make other enhancements using MATLAB tools.

Use the following techniques to isolate the causes of errors and improve your M-files.

Technique or Tool	Description	For More Information
Syntax highlighting and Delimiter matching	Syntax highlighting helps you identify unterminated strings in an M-file before you run the file. Delimiter matching helps you correctly match pairs of parentheses, brackets, braces, and keywords.	"Syntax Highlighting" on page 6-28 "Matching Delimiters (Parentheses)" on page 3-17

Technique or Tool	Description	For More Information
Error Messages	When you run an M-file with a syntax error, MATLAB will most likely detect it and display an error message in the Command Window describing the error and showing its line number in the M-file. Click the underlined portion of the error message, or position the cursor within the message and press **Ctrl+Enter**. The offending M-file opens in the Editor/Debugger, scrolled to the line containing the error. To check for syntax errors in an M-file without running the M-file, use the `pcode` function.	None
M-Lint	Use the M-Lint code analyzer to help you verify the integrity of your code and learn about potential improvements. Access M-Lint messages automatically while you work in a file in the Editor/Debugger, or run an M-Lint report for an existing file. To evaluate the McCabe complexity (also known as the cyclomatic complexity) of an M-File, use the `mlint` function with the `-cyc` option.	"M-Lint Code Analyzer" on page 6-87 and the reference page for the `mlint` function
Editor/ Debugger Graphical Debugger and MATLAB Debugging Functions	The MATLAB Editor/Debugger graphical debugger and MATLAB debugging functions are useful for correcting run-time problems because you can access function workspaces and examine or change the values they contain. You can set and clear *breakpoints*, indicators that temporarily halt execution in an M-file. While stopped at a breakpoint, you can change workspace contexts, view the function call stack, and execute the lines in an M-file one by one.	"Debugging Process and Features" on page 6-103

Technique or Tool	Description	For More Information
Other Debugging Techniques	• Add `keyboard` statements to the M-file — `keyboard` statements stop M-file execution at the point where they appear and allow you to examine and change the function's local workspace. This mode is indicated by a special `K>>`prompt. Resume function execution by typing `return` and pressing the **Enter** key. For more information, see the `keyboard` reference page. • Remove selected semicolons from the statements in your M-file—semicolons disable the display of output in the M-file. By removing the semicolons, you instruct MATLAB to display these results on your screen as the M-file executes. • List dependent functions—use the `depfun` function to see the dependent functions.	Reference pages for `keyboard` and `depfun` function
Cells	In the Editor/Debugger, isolate sections of an M-file, called cells, so you can easily make changes to and run a single section.	"Using Cells for Rapid Code Iteration and Publishing Results" on page 6-133
Profiler	Use the Profiler to help you improve performance and detect problems in your M-files. Access the Profiler from the Editor/Debugger by selecting **Tools > Open Profiler**.	"Profiling for Improving Performance" on page 7-27
Directory Reports	The M-file Directory Reports help you polish and package M-files before providing them to others to use. Access all of these tools from the Current Directory browser. You can access some of these directly from the Editor/Debugger **Tools** menu.	"Directory Reports in Current Directory Browser" on page 7-2

M-Lint Code Analyzer

In this section...

What Is M-Lint?

The M-Lint code analyzer checks your code for problems and recommends modifications to maximize performance and maintainability.

Ways to Use M-Lint

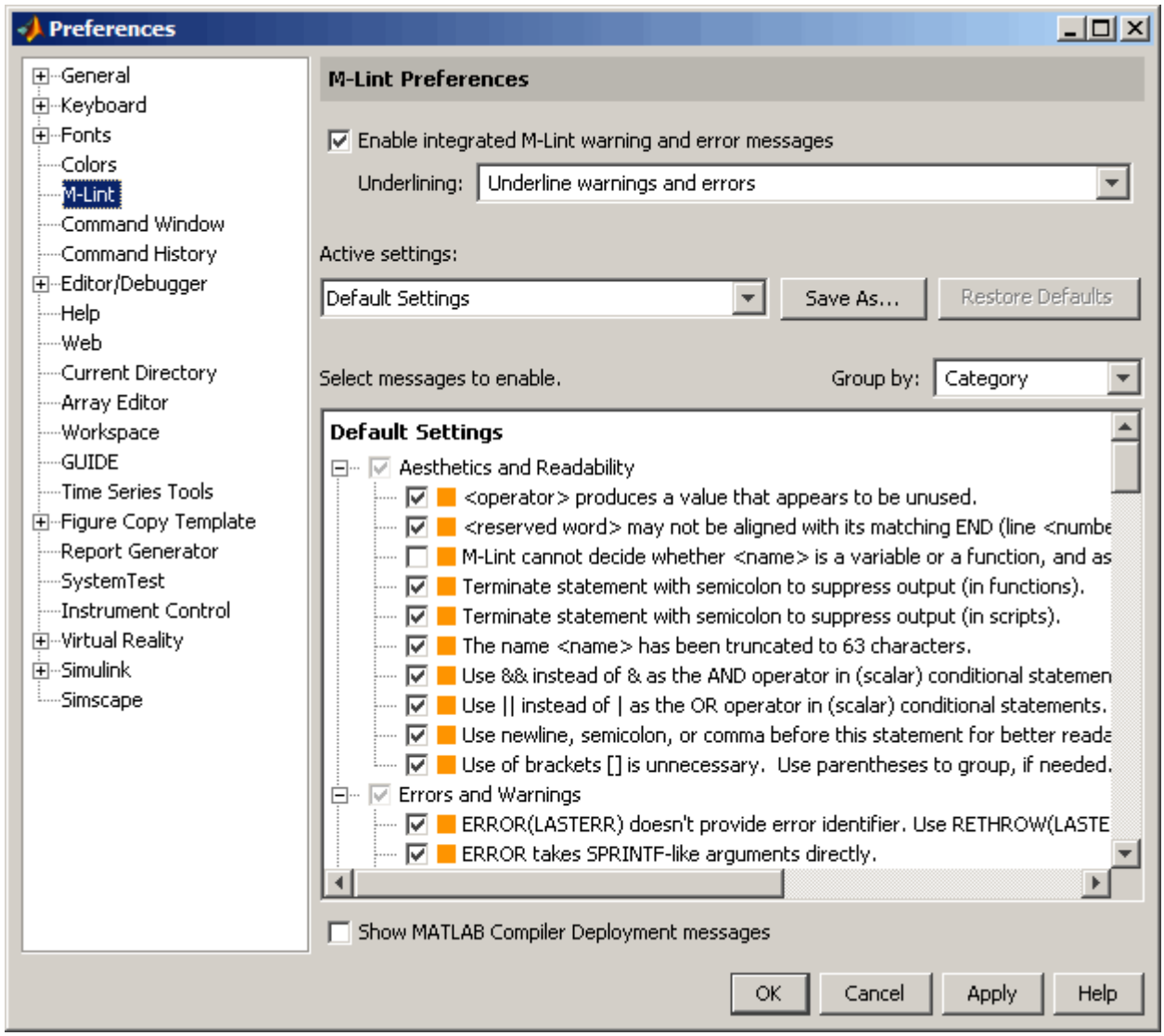
You can use M-Lint in two different ways, both of which report the same information:

- Run a report for an existing M-file or group of M-files. To do so, from an M-file in the Editor/Debugger, select **Tools > M-Lint > Show M-Lint Report**. Make any changes to your file based on the M-Lint messages in the report. After making changes, you must save the file and rerun the report to see if your changes addressed the issues noted in M-Lint messages. To run M-Lint for all files in a directory, access M-Lint from the Current Directory browser — select **View > Directory Reports > M-Lint Code Check Report**. For details, see "M-Lint Code Check Report" on page 7-16.
- Continuously check code in the Editor/Debugger while you work. View M-Lint messages and make changes to your file based on the messages. The code analyzer updates automatically and continuously so you can see if your changes addressed the issues noted in the M-Lint messages. For some messages, M-Lint offers automatic code correction. For details about specific M-Lint messages, see "M-Lint Code Check Report" on page 7-16. Information about using the continuous checking and correction interface in the Editor/Debugger is explained here.

M-Lint Automatic Code Analyzer in the Editor/Debugger

To use the M-Lint continuous code checking in an M-file in the Editor/Debugger, perform the following steps:

1. Ensure the M-Lint messaging preference is enabled: Select **File > Preferences > M-Lint** and select the **Enable integrated M-Lint warning and error messages** check box. To follow these instructions, be sure the **Underlining** option is set to `Underline warnings and errors`. Click **OK**.



2 Open an M-file in the Editor/Debugger. This example uses the sample file `lengthofline.m`:

a Open the example file:

```
open(fullfile(matlabroot,'help','techdoc','matlab_env','examples','lengthofline.m'))
```

b Save the example file to a directory to which you have write access. For the example, `lengthofline.m` is saved to `I:\MATLABFiles\mymfiles`.

3 The M-Lint message indicator at the top right edge of the window conveys the M-Lint messages reported for the file:

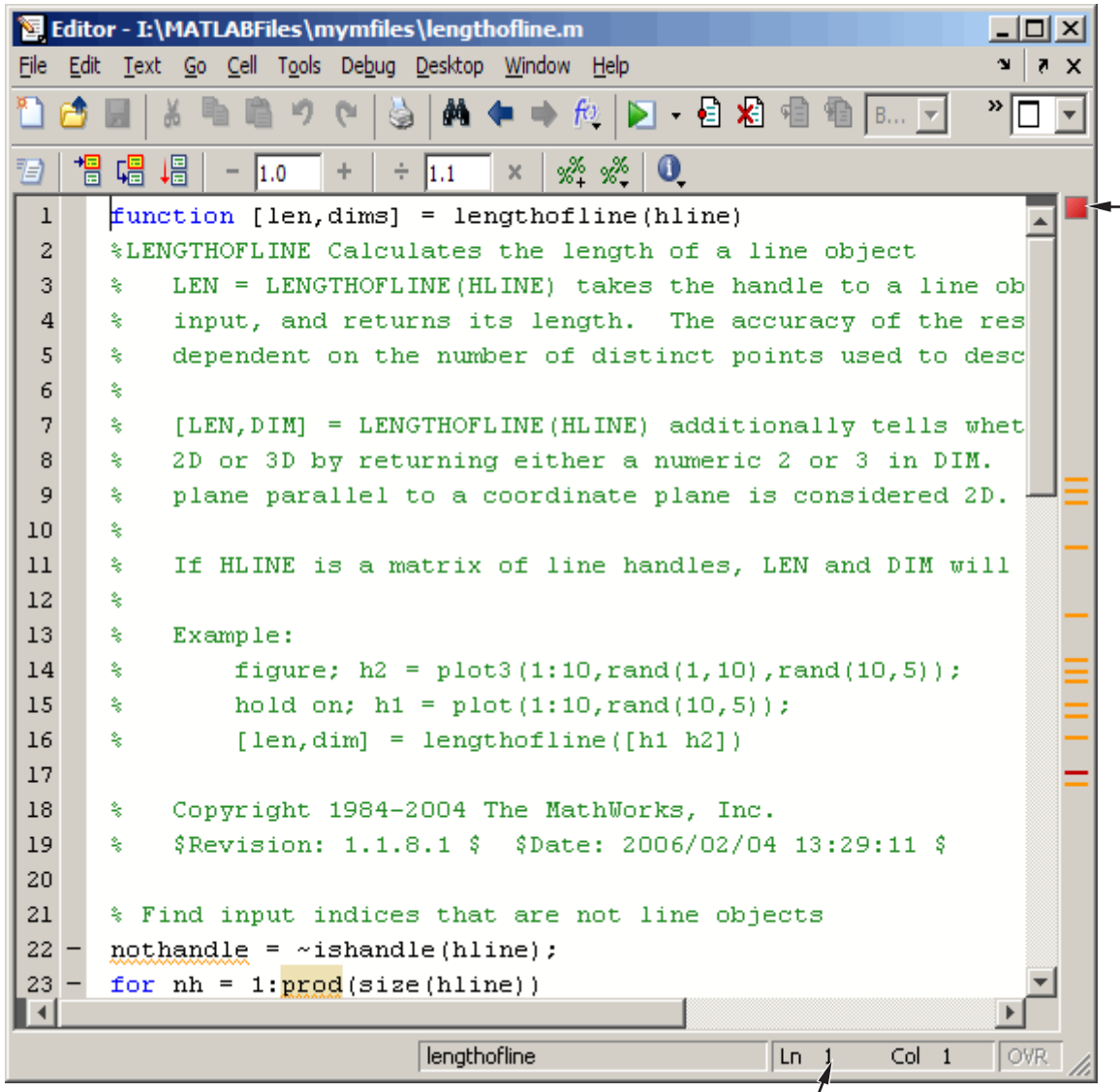
- Red means syntax errors were detected. Another way to detect some of these errors is using syntax highlighting to identify unterminated strings, and delimiter matching to identify unmatched keywords, parentheses, braces, and brackets.
- Orange means warnings or opportunities for improvement were detected, but no errors were detected.
- Green means no errors, warnings, or opportunities for improvement were detected.

For the example, the indicator is red, meaning there is at least one error in the file.

M-Lint message indicator for all messages in entire file:

- Red means errors detected
- Orange means warnings or improvement opportunities detected
- Green means none detected

Click indicator to go to next line that has an associated M-Lint message.



Current cursor position

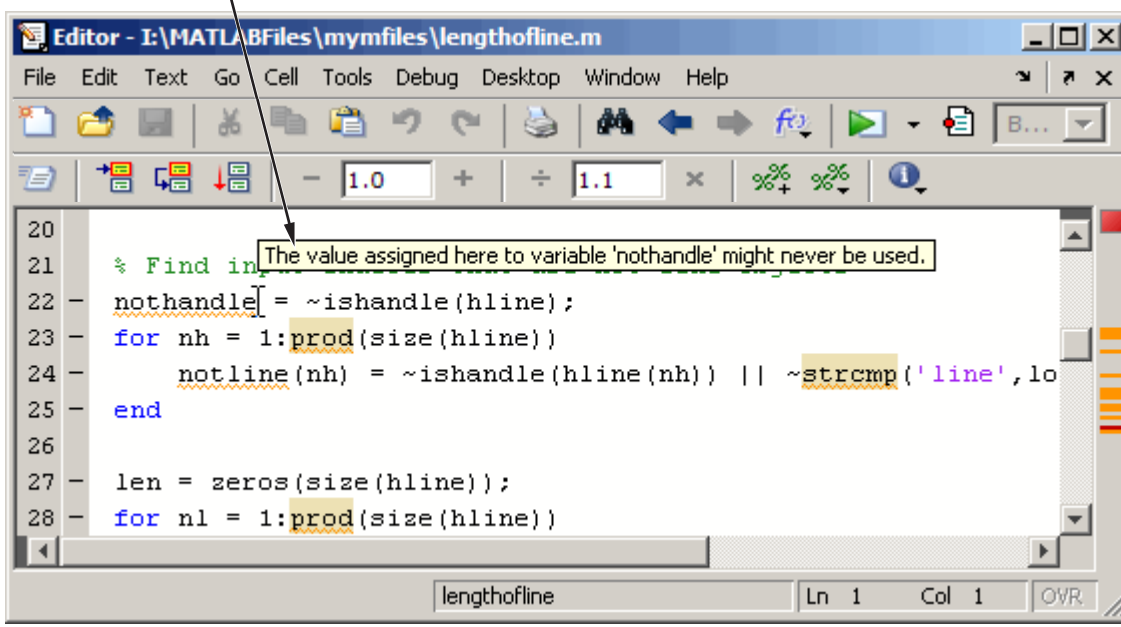
4 Click the M-Lint message indicator to go to the next code fragment containing an M-Lint message. The next code fragment is relative to the current cursor position, viewable in the status bar.

   In the `lengthofline` example, the first message is at line 22. The cursor moves to the beginning of line 22.

5 The code fragment for which there is an M-Lint message is underlined in either red for errors or orange for warnings and improvement opportunities.

   To view the M-Lint message, move the pointer within the underlined fragment. The message appears with a yellow highlighted background, similar to datatips (see "Viewing Values as Datatips in the Editor/Debugger" on page 6-115).

Position cursor within orange underlined code fragment and Editor/Debugger displays the related M-Lint message.



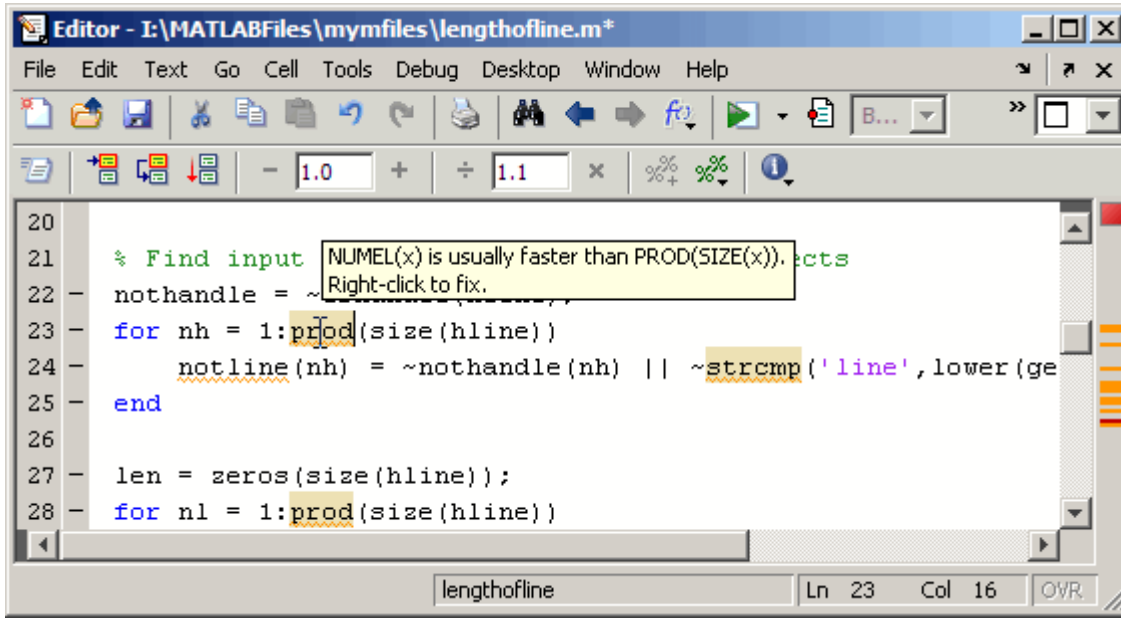
   This message means that in line 22, `nothandle` is assigned a value, but is probably not used anywhere after that in the file. The line might be

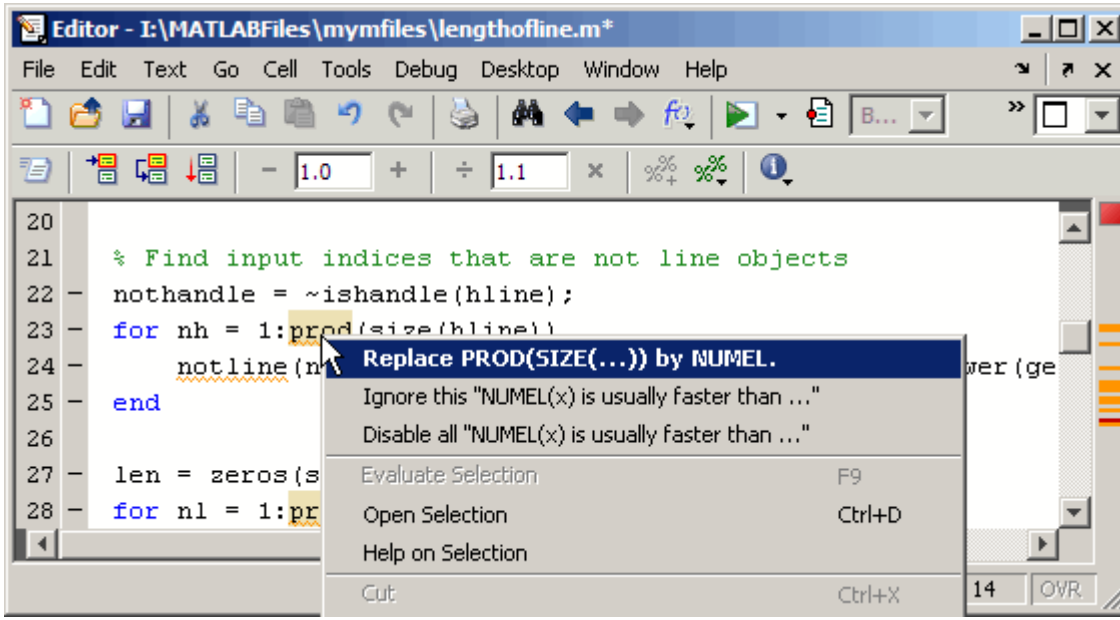
extraneous and you could delete it. But it might be that you actually intended to use the variable, as shown in step 6 of this example.

6 Make changes to your code as needed. The M-Lint indicator and underlining automatically update to reflect the changes you make, even if you do not save the file.

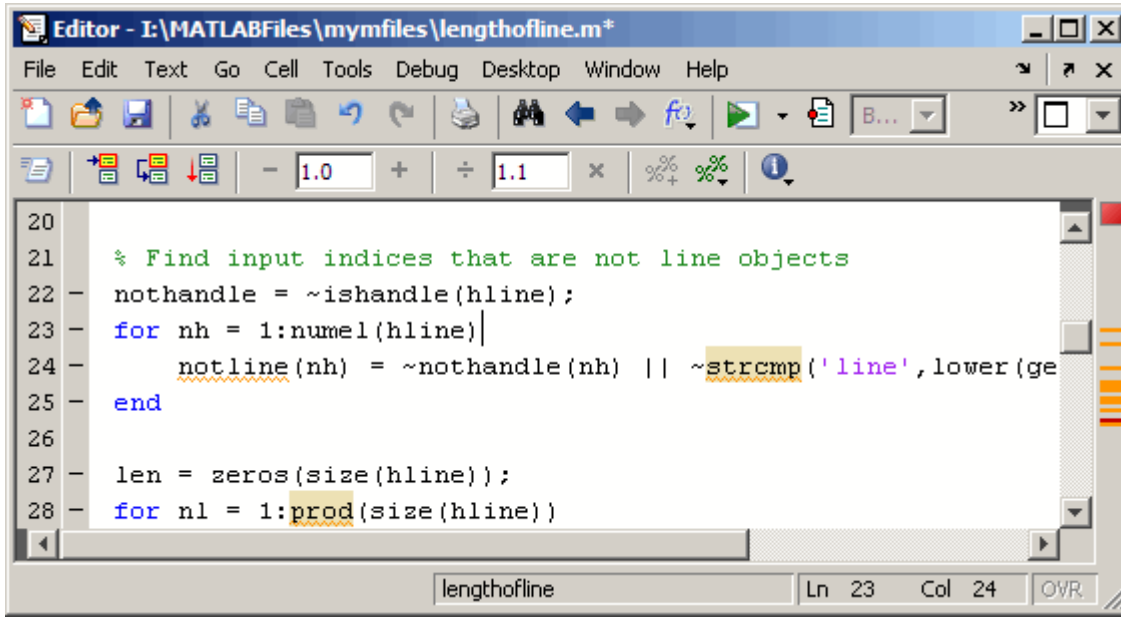
In this example, the intention was to use `nothandle` as a performance improvement by determining the value prior to the loop. Changing `~ishandle(hline(nh))` in line 24 to `nothandle(nh)` means there is no longer an M-Lint message associated with line 22. For more information about what the warning and improvement messages in this example mean and actions you can take to address them, see "Messages and Resulting Changes for the lengthofline Example" on page 7-21.

7 Some errors and warnings are highlighted, indicating M-Lint can automatically fix the code. For example, in `lengthofline`, line 23, `prod` is underlined because there is an M-Lint warning, and it is highlighted because an automatic fix is available. When you view the M-Lint message, it also indicates the auto-fix that is available.





Right-click the highlighted code (for a single-button mouse, use **Ctrl**+click). The first item in the context menu indicates the automatic fix that M-Lint can perform. Select it and M-Lint automatically corrects the code. In this example, M-Lint replaces `prod(size(hline))` with `numel(hline)`.



There is a preference you can set for the color—for more information, see "Other Colors" on page 2-75.

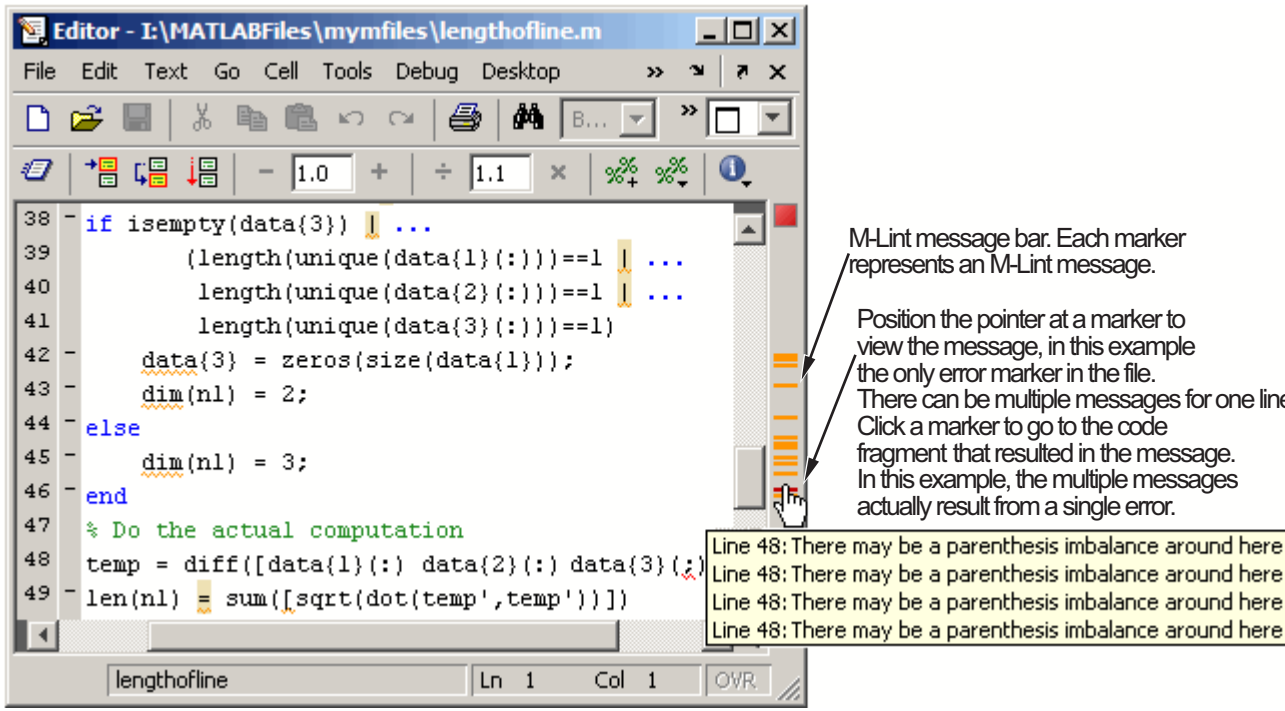
8 You might want to ignore certain M-Lint messages and do not want the messages to display; for more information, see "Suppressing M-Lint Indicators and Messages" on page 6-98.

9 You can click the M-Lint message indicator to go to the next message, or use the other way to view messages, which is the M-Lint message bar. Each marker in the bar represents a line that has associated M-Lint messages. A red marker means there is an error at that line, while an orange marker means there are warnings or suggested improvements, but no errors at that line.

   a Position the pointer at a marker in the message bar to view the message. For example, to see an error in `lengthofline`, position the pointer at a red marker in the message bar. There is only one error in the file and with the pointer positioned over it, the associated M-Lint messages appears. Click the marker to go to the first code fragment in the line that resulted in an M-Lint message. For the example, click the red marker, which takes you to the first suspect code fragment in line 48.

```
temp = diff([data{1}(:) data{2}(:) data{3}(;)]);
```

Multiple messages can represent a single problem or multiple problems. Addressing one might address all of them, or after addressing one, the other messages might change or what you need to do might become clearer.



b Make changes to address the problem noted in the M-Lint message—the M-Lint indicators update automatically.

In the example, the M-Lint message suggest a delimiter imbalance. You can check that by moving the arrow key over each of the delimiters to see if MATLAB indicates a mismatch. This requires that **File > Preferences > Keyboard > Delimiter Matching** has the **Match on arrow key** option selected. There are no mismatched delimiters. The actual problem is the semicolon in parentheses, `data{3}(;)`, is incorrect and should be a colon. In line 48, change `data{3}(;)` to `data{3}(:)`. When you make the change, the underline no longer appears in line 48.

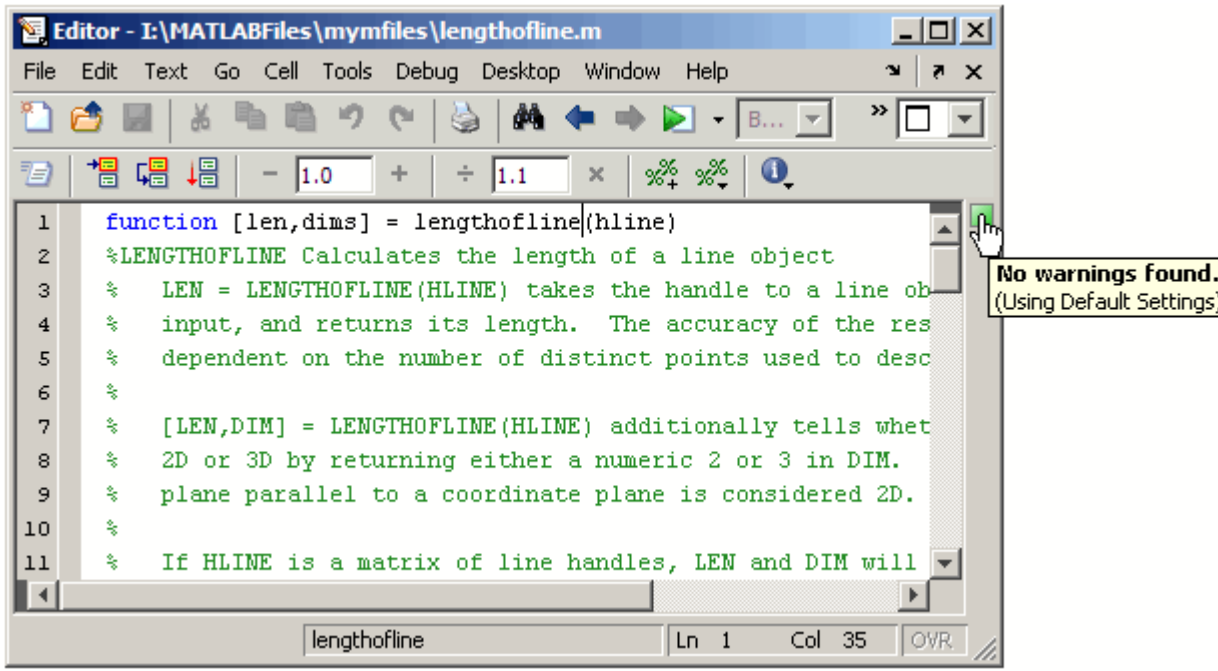
That single change addressed the issues in all of the M-Lint messages for line 48.

Because the change you made removed the only error in the file, the M-Lint message indicator at the top of the bar changes from red to orange, indicating that only warnings and potential improvements remain.

c If there are multiple messages associated with a line, there might be multiple underlined code fragments that are adjacent, as in the above example, making it difficult to display the message of interest. In those cases, it might be easier to view the messages via the marker on the message bar.

10 After making changes to address all M-Lint messages, or disabling designated messages, the M-Lint message indicator becomes green The example file with all M-Lint messages addressed has been saved as `lengthofline2.m`, which you can open by running

```
open(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','lengthofline2.m'))
```



Suppressing M-Lint Indicators and Messages

Depending on what stage you are at in completing the M-file, you might want to restrict the underlining, which you can do via the M-Lint preference referred to in step 1, above. For example, when first coding, you might prefer no underlines because they would be distracting. For details, click the **Help** button in the Preferences dialog box.

M-Lint does not provide perfect information about every situation and in some cases, you might not want to make any changes based on an M-Lint message. In the event you do not want to change the code but you also do not want to see the M-Lint indicator and message for that line, instruct M-Lint to suppress them. For the `lengthofline` example, in line 49, the first M-Lint message is `Terminate statement with semicolon to suppress output (in functions)`. Adding a semicolon to the end of a statement suppresses output and is a common practice. M-Lint alerts you to lines that produce output but lack the terminating semicolon. If you want to view output from line 49, do not add the semicolon as M-Lint suggests.

There are a few different ways to suppress the M-Lint indicators and messages:

- "Ignore Only a Specific Instance" on page 6-99
- "Disable All Instances in All Files" on page 6-101
- "Disable Specified Messages or in Selected Files as Needed" on page 6-101

Note that you cannot suppress M-Lint error messages such as syntax errors, and therefore, the following options do not apply.

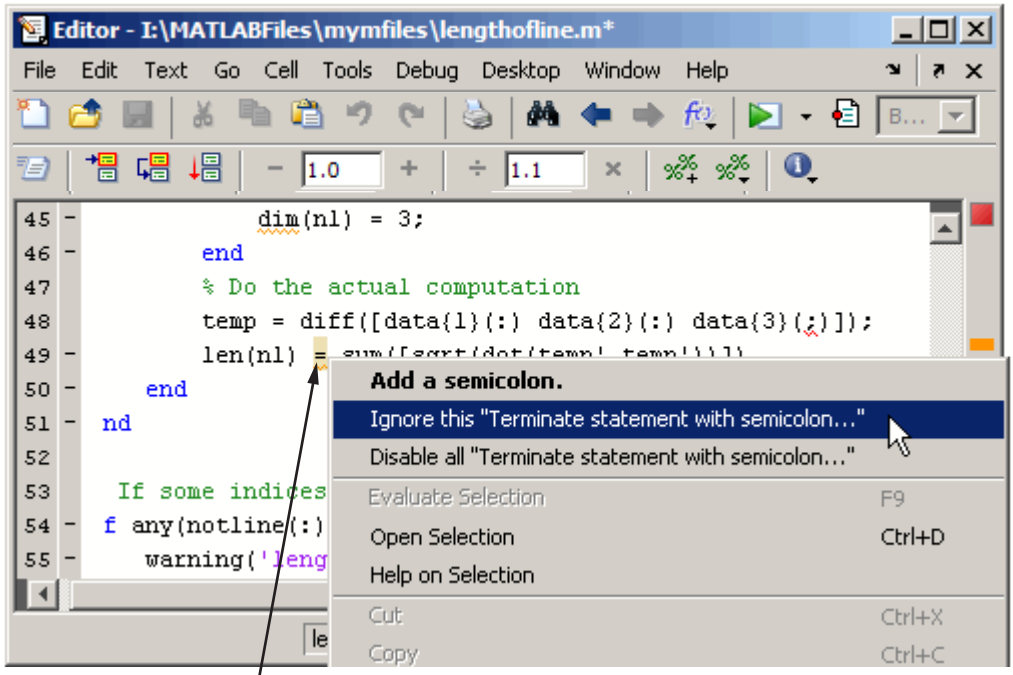
Ignore Only a Specific Instance

Right-click at the M-Lint underline (for a single-button mouse, use **Ctrl**+click). From the context menu, select `Ignore this "Terminate statement with semicolon..."`. M-Lint adds a `%#ok<NOPRT>` to the end of the line, which instructs MATLAB not to check for a terminating semicolon at that line. M-Lint removes the underline and mark in the M-Lint indicator bar for that message.

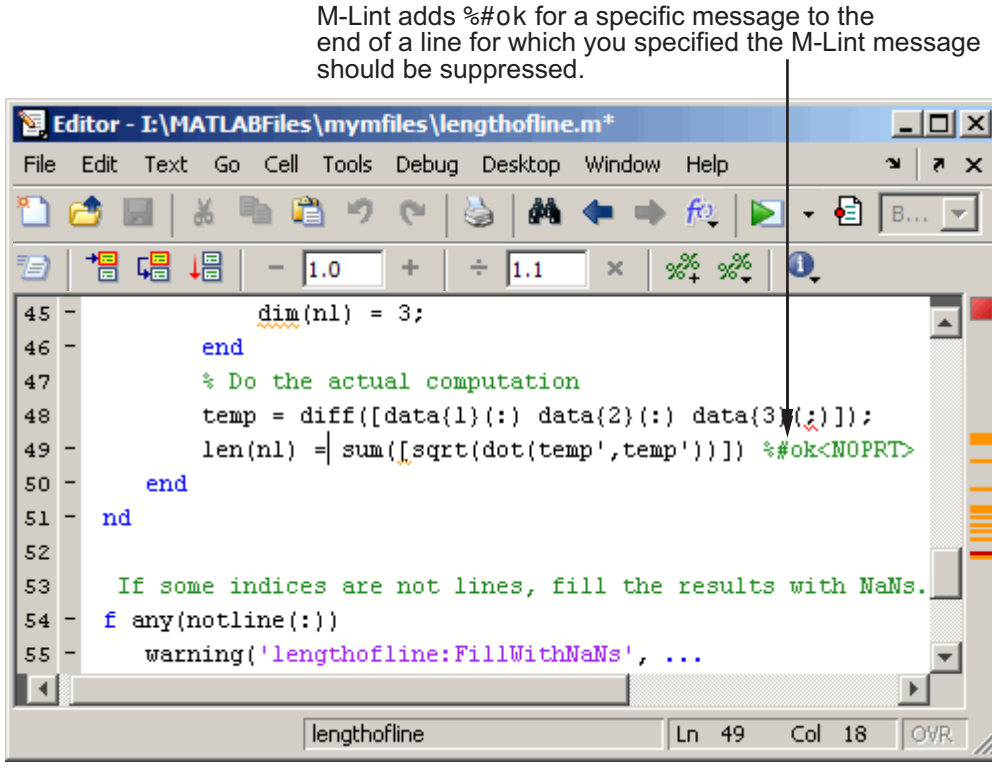
If there are two messages on a line that you do not want M-Lint to display, right-click separately at each underline and select the appropriate entry from the context menu. M-Lint expands the `%#ok` syntax. For the example, ignoring both messages for line 49 would add `%#ok<NBRAK,NOPRT>`.

For more information about `%#ok`, see the `mlint` function reference page.

This method of suppressing the messages changes the M-file. If M-Lint preferences are to set to enable this message, the specific instance of the message suppressed in this way will not appear because the `%#ok` takes precedence over the preference setting. If you later decide you want M-Lint to check for a terminating semicolon at that line, delete the `%#ok<NOPRT>` from the line.



Right-click at an M-Lint underline and select the option instructing M-Lint to ignore only this instance of the message, as shown in this example.



Disable All Instances in All Files

Right-click at the M-Lint underline (for a single-button mouse, use **Ctrl**+click). From the context menu, select `Disable all "Terminate statement with semicolon..."`. Doing so modifies the M-Lint preference setting, which applies to all occurrences in all files, unless a line includes a `%#ok` for that message. For more information about the M-Lint preference, including how to restore MATLAB default settings, select **File > Preferences > M-Lint**, and click **Help**.

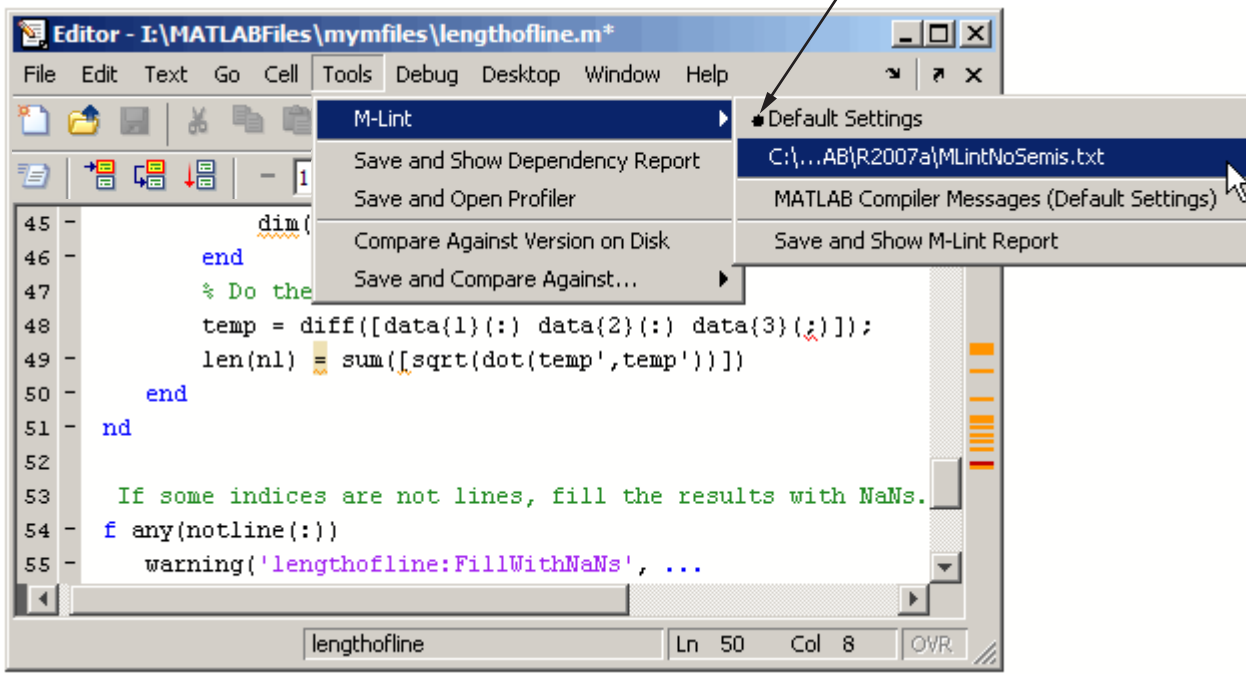
Disable Specified Messages or in Selected Files as Needed

Use M-Lint preferences by selecting **File > Preferences > M-Lint**. Then enable specific messages or categories of messages and save the settings to a

`txt` file. You can reuse the settings for any M-file, or provide the settings file to another user.

To use the saved settings, either select the settings file in M-Lint preferences, or in the Editor/Debugger. In the Editor/Debugger, right-click the M-Lint message bar (for a single-button mouse, use **Ctrl**+click), or select **Tools > M-Lint**. The currently-selected setting choice is shown, preceded by a bullet point. You can choose from any of the settings files, such as the `MLintNoSemis` example, as shown here.

M-Lint default settings are currently selected (as indicated by the bullet point preceding that menu item). Select any `txt` file to use the M-Lint settings specified in that file.



For more about M-Lint settings and preferences, click **Help** in the M-Lint preferences panel.

Debugging Process and Features

In this section...

Ways to Debug M-Files

You can debug the M-files using the Editor/Debugger, which is a graphical user interface, as well as by using debugging functions from the Command Window. You can use both methods interchangeably. These topics and the example describe both methods.

Preparing for Debugging

Do the following to prepare for debugging:

- Open the file — To use the Editor/Debugger for debugging, open it with the file to run.
- Save changes — If you are editing the file, save the changes before you begin debugging. If you try to run a file with unsaved changes from within the Editor/Debugger, the file is automatically saved before it runs. If you run a file with unsaved changes from the Command Window, MATLAB runs the saved version of the file, so you will not see the results of your changes.
- Add the files to a directory on the search path or put them in the current directory — Be sure the file you run and any files it calls are in directories

that are on the search path. If all files to be used are in the same directory, you can instead make that directory be the current directory.

Debugging Example — The Collatz Problem

The debugging process and features are best described via an example. To prepare to use the example, create two M-files, `collatz.m` and `collatzplot.m`, that produce data for the Collatz problem.

For any given positive integer, `n`, the `Collatz` function produces a sequence of numbers that always resolves to 1. If `n` is even, divide it by 2 to get the next integer in the sequence. If `n` is odd, multiply it by 3 and add 1 to get the next integer in the sequence. Repeat the steps until the next integer is 1. The number of integers in the sequence varies, depending on the starting value, `n`.

The Collatz problem is to prove that the Collatz function will resolve to 1 for all positive integers. The M-files for this example are useful for studying the Collatz problem. The file `collatz.m` generates the sequence of integers for any given `n`. The file `collatzplot.m` calculates the number of integers in the sequence for all integers from 1 through `m`, and plots the results. The plot shows patterns that can be further studied.

Following are the results when `n` is 1, 2, or 3.

n	Sequence	Number of Integers in the Sequence
1	1	1
2	2 1	2
3	3 10 5 16 8 4 2 1	8

M-Files for the Collatz Problem. Following are the two M-files you use for the debugging example. To create these files on your system, open two new M-files. Select and copy the following code from the Help browser and paste it into the M-files. Save and name the files `collatz.m` and `collatzplot.m`. Save them to your current directory or add the directory where you save them to the search path. One of the files has an embedded error to illustrate the debugging features.

Code for `collatz.m`.

```
function sequence=collatz(n)
% Collatz problem. Generate a sequence of integers resolving to 1
% For any positive integer, n:
%  Divide n by 2 if n is even
%  Multiply n by 3 and add 1 if n is odd
%  Repeat for the result
%  Continue until the result is 1%

sequence = n;
next_value = n;
while next_value > 1
 if rem(next_value,2)==0
  next_value = next_value/2;
 else
  next_value = 3*next_value+1;
 end
 sequence = [sequence, next_value];
end
```

Code for `collatzplot.m`.

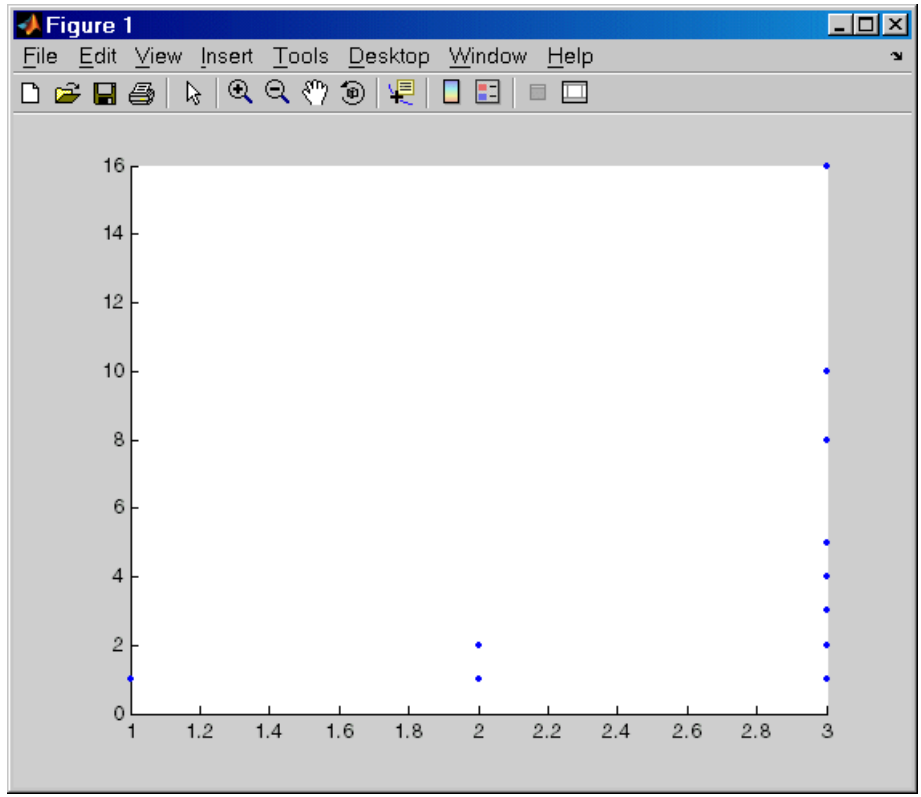
```
function collatzplot(m)
% Plot length of sequence for Collatz problem
% Prepare figure
clf
set(gcf,'DoubleBuffer','on')
set(gca,'XScale','linear')
%
% Determine and plot sequence and sequence length
for N = 1:m
 plot_seq = collatz(N);
 seq_length(N) = length(plot_seq);
 line(N,plot_seq,'Marker','.','MarkerSize',9,'Color','blue')
 drawnow
end
```

Trial Run for Example. Open the file `collatzplot.m`. Make sure the current directory is the directory in which you saved `collatzplot`.

Try out `collatzplot` to see if it works correctly. Use a simple input value, for example, 3, and compare the results to those shown in the preceding table. Typing

```
collatzplot(3)
```

produces the plot shown in the following figure.



The plot for n = 1 appears to be correct—for 1, the Collatz series is 1, and contains one integer. But for n = 2 and n = 3, it is wrong because there should be only one value plotted for each integer, the number of integers in the sequence, which the preceding table shows to be 2 (for n = 2) and 8 (for n = 3). Instead, multiple values are plotted. Use MATLAB debugging features to isolate the problem.

Setting Breakpoints

Set breakpoints to pause execution of the M-file so you can examine values where you think the problem might be. You can set breakpoints in the Editor/Debugger, using functions in the Command Window, or both.

There are three basic types of breakpoints you can set in M-files:

- A standard breakpoint, which stops at a specified line in an M-file. For details, see "Setting Standard Breakpoints" on page 6-108.
- A conditional breakpoint, which stops at a specified line in an M-file only under specified conditions. For details, see "Conditional Breakpoints" on page 6-126.
- An error breakpoint that stops in any M-file when it produces the specified type of warning, error, or `NaN` or infinite value. For details, see "Error Breakpoints" on page 6-129.

You can disable standard and conditional breakpoints so that MATLAB temporarily ignores them, or you can remove them. For details, see "Disabling and Clearing Breakpoints" on page 6-119. Breakpoints are not maintained after you exit the MATLAB session.

You can only set valid standard and conditional breakpoints at executable lines in saved files that are in the current directory or in directories on the search path. When you add or remove a breakpoint in a file that is not in a directory on the search path or in the current directory, a dialog box appears, presenting you with options that allow you to add or remove the breakpoint. You can either change the current directory to the directory containing the file, or you can add the directory containing the file to the search path.

Do not set a breakpoint at a `for` statement if you want to examine values at increments in the loop. For example, in

```
for n = 1:10
    m = n+1;
end
```

MATLAB executes the `for` statement only once, which is efficient. Therefore, when you set a breakpoint at the `for` statement and step through the file, you

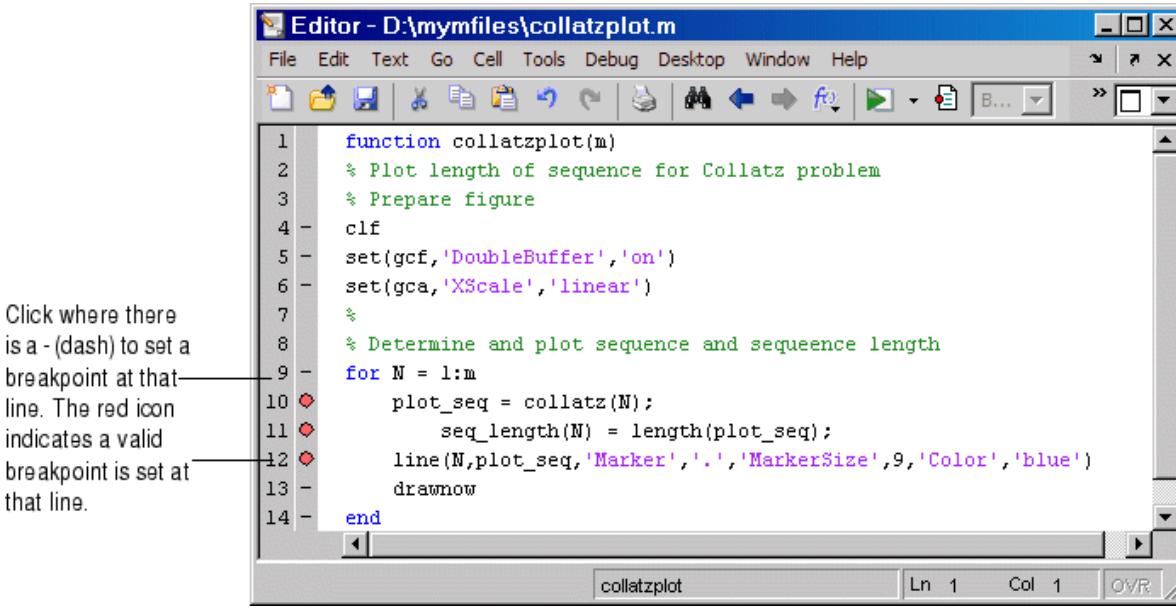
only stop at the `for` statement once. Instead place the breakpoint at the next line, `m=n+1` to stop at each pass through the loop.

You cannot set breakpoints while MATLAB is busy, for example, running an M-file, unless that M-file is paused at a breakpoint.

Setting Standard Breakpoints

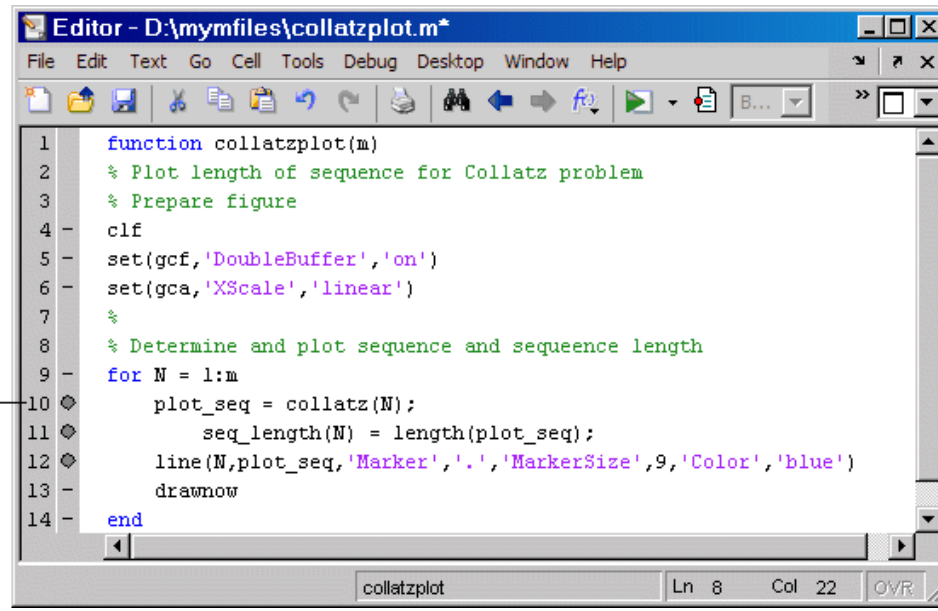
To set a standard breakpoint using the Editor/Debugger, click in the breakpoint alley at the line where you want to set the breakpoint. The breakpoint alley is the narrow column on the left side of the Editor/Debugger, just right of the line number. Set breakpoints at lines that are preceded by a - (dash). Lines not preceded by a dash, such as comments or blank lines, are not executable — if you try to set a breakpoint there, it is actually set at the next executable line. Other ways to set a breakpoint are to position the cursor in the line and then click the Set/Clear Breakpoint button  on the toolbar, or select **Set/Clear Breakpoint** from the **Debug** menu or the context menu. A breakpoint icon appears.

Set Breakpoints for the Example. It is unclear whether the problem in the example is in `collatzplot` or `collatz`. To start, set breakpoints in `collatzplot.m` at lines 10, 11, and 12. The breakpoint at line 10 allows you to step into `collatz` to see if the problem might be there. The breakpoints at lines 11 and 12 stop the program where you can examine the interim results.



Valid (Red) and Invalid (Gray) Breakpoints. Red breakpoints are valid standard breakpoints. If breakpoints are instead gray, they are not valid.

When breakpoints are gray, they are not valid. In this example, it is because the file has not been saved since changes were made to it. Save the file to make the breakpoints valid (red).



Breakpoints are gray for either of these reasons:

- The file has not been saved since changes were made to it. Save the file to make breakpoints valid. The gray breakpoints become red, indicating they are now valid. Any gray breakpoints that were entered at invalid breakpoint lines automatically move to the next valid breakpoint line with the successful file save.
- There is a syntax error in the file. When you set a breakpoint, an error message appears indicating where the syntax error is. Fix the syntax error and save the file to make breakpoints valid.

Function Alternative for Setting Breakpoints

To set a breakpoint using the debugging functions, use `dbstop`. For the example, type

```
dbstop in collatzplot at 10
dbstop in collatzplot at 11
dbstop in collatzplot at 12
```

Some useful related functions are

- `dbtype` — Lists the M-file with line numbers in the Command Window.
- `dbstatus` — Lists breakpoints.

Running an M-File with Breakpoints

After setting breakpoints, run the M-file from the Command Window or the Editor/Debugger.

Running the Example

For the example, run `collatzplot` for the simple input value, 3, by typing in the Command Window

```
collatzplot(3)
```

The example, `collatzplot`, requires an input argument and therefore runs only from the Command Window and not from the Editor/Debugger.

Results of Running an M-File Containing Breakpoints

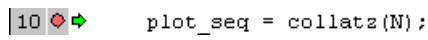
Running the M-file results in the following:

- The prompt in the Command Window changes to

  ```
  K>>
  ```

  indicating that MATLAB is in debug mode.

- The program pauses at the first breakpoint. This means that line will be executed when you continue. The pause is indicated in the Editor/Debugger by the green arrow just to the right of the breakpoint, which in the example, is line 10 of `collatzplot` as shown here.

  If you use debugging functions from the Command Window, the line at which you are paused is displayed in the Command Window. For the example, it would show

  ```
  10   plot_seq = collatz(N);
  ```

- The function displayed in the **Stack** field on the toolbar changes to reflect the current function (sometimes referred to as the caller or calling workspace). The call stack includes subfunctions as well as called functions. If you use debugging functions from the Command Window, use `dbstack` to view the current call stack.
- If the file you are running is not in the current directory or a directory on the search path, you are prompted to either add the directory to the path or change the current directory.

In debug mode, you can set breakpoints, step through programs, examine variables, and run other functions.

Note that MATLAB could become nonresponsive if it stops at a breakpoint while displaying a modal dialog box or figure that your M-file creates. In that event, use **Ctrl+C** to go the MATLAB prompt.

Stepping Through an M-File

While debugging, you can step through an M-file, pausing at points where you want to examine values.

Use the step buttons or the step items in the **Debug** menu of the Editor/Debugger or desktop, or use the equivalent functions.

Toolbar Button	Debug Menu Item	Description	Function Alternative
	**Continue** or **Run** or **Save and Run**	Continue execution of M-file until completion or until another breakpoint is encountered. The menu item says **Run** or **Save and Run** if a file is not already running.	`dbcont`
None	**Go Until Cursor**	Continue execution of M-file until the line where the cursor is positioned. Also available on the context menu.	None

Toolbar Button	Debug Menu Item	Description	Function Alternative
	**Step**	Execute the current line of the M-file.	`dbstep`
	**Step In**	Execute the current line of the M-file and, if the line is a call to another function, step into that function.	`dbstep in`
	**Step Out**	After stepping in, run the rest of the called function or subfunction, leave the called function, and pause.	`dbstep out`

Continue Running in the Example

In the example, `collatzplot` is paused at line 10. Because the problem results are correct for `N/n = 1`, continue running until `N/n = 2`. Press the Continue button three times to move through the breakpoints at lines 10, 11, and 12. Now the program is again paused at the breakpoint at line 10.

Stepping In to Called Function in the Example

Now that `collatzplot` is paused at line 10 during the second iteration, use the Step In button or type `dbstep in` in the Command Window to step into `collatz` and walk through that M-file. Stepping into line 10 of `collatzplot` goes to line 9 of `collatz`. If `collatz` is not open in the Editor/Debugger, it automatically opens if you have selected **Debug > Open M-Files When Debugging**.

The pause indicator at line 10 of `collatzplot` changes to a hollow arrow ⇨, indicating that MATLAB control is now in a subfunction called from the main program. The call stack shows that the current function is now `collatz`.

In the called function, `collatz` in the example, you can do the same things you can do in the main (calling) function—set breakpoints, run, step through, and examine values.

Examining Values

While the program is paused, you can view the value of any variable currently in the workspace. Examine values when you want to see whether a line of code has produced the expected result or not. If the result is as expected, continue running or step to the next line. If the result is not as expected, then that line, or a previous line, contains an error. Use the following methods to examine values:

Many of these methods are used in "Examining Values in the Example" on page 6-117.

Selecting the Workspace

Variables assigned through the Command Window and created using scripts are considered to be in the base workspace. Variables created in a function belong to their own function workspace. To examine a variable, you must first select its workspace. When you run a program, the current workspace is shown in the **Stack** field. To examine values that are part of another workspace for a currently running function or for the base workspace, first select that workspace from the list in the **Stack** field.

If you use debugging functions from the Command Window, use `dbstack` to display the call stack. Use `dbup` and `dbdown` to change to a different workspace. Use `who` or `whos` to list the variables in the current workspace.

Workspace in the Example. At line 10 of `collatzplot`, you stepped in, so the current line is 9 in `collatz`. The **Stack** shows that `collatz` is the current workspace.

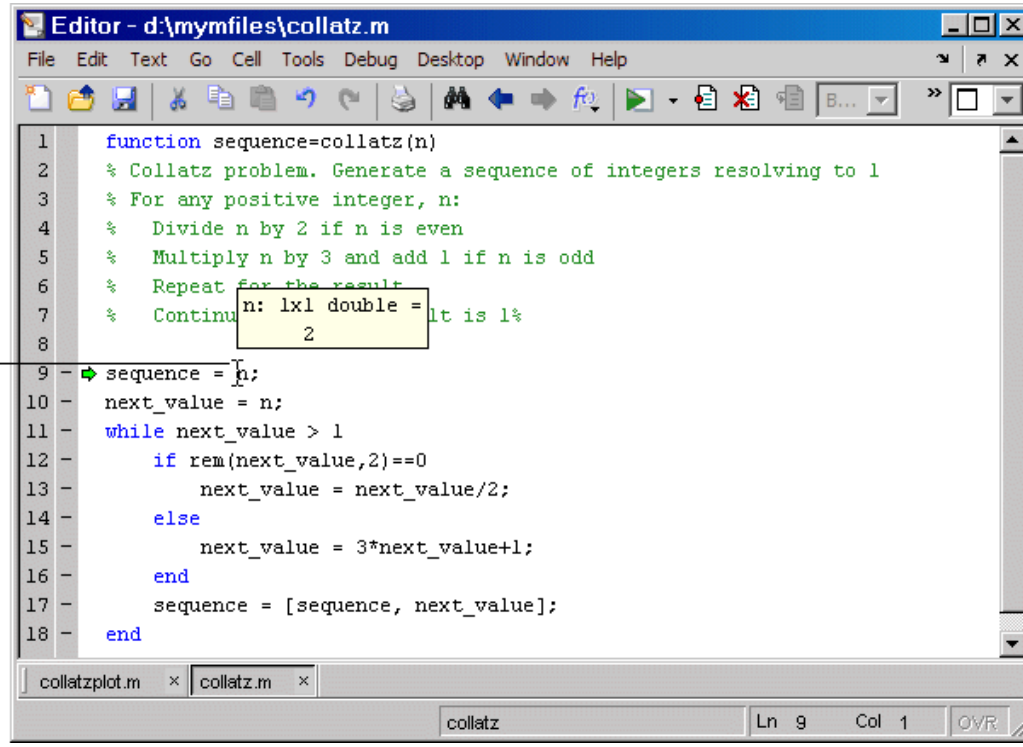
Viewing Values as Datatips in the Editor/Debugger

In the Editor/Debugger, position the pointer to the left of a variable on that line. Its current value appears — this is called a datatip, which is like a tooltip for data. The datatip stays in view until you move the pointer. If you have trouble getting the datatip to appear, click in the line and then move the pointer next to the variable.

A related function is `datatipinfo`.

Datatips in the Example. Position the pointer over `n` in line 9 of `collatz`. The datatip shows that `n = 2`, as expected.

Hold the pointer over a variable. Its current value temporarily displays as a datatip.




```
1   function sequence=collatz(n)
2   % Collatz problem. Generate a sequence of integers resolving to 1
3   % For any positive integer, n:
4   %   Divide n by 2 if n is even
5   %   Multiply n by 3 and add 1 if n is odd
6   %   Repeat for the result
7   %   Continu                    lt is 1%
8
9   sequence = n;
10  next_value = n;
11  while next_value > 1
12      if rem(next_value,2)==0
13          next_value = next_value/2;
14      else
15          next_value = 3*next_value+1;
16      end
17      sequence = [sequence, next_value];
18  end
```

Viewing Values in the Command Window

You can examine values while in debug mode at the `K>>` prompt. To see the variables currently in the workspace, use `who`. Type a variable name in the

Command Window and MATLAB displays its current value. For the example, to see the value of `n`, type

```
n
```

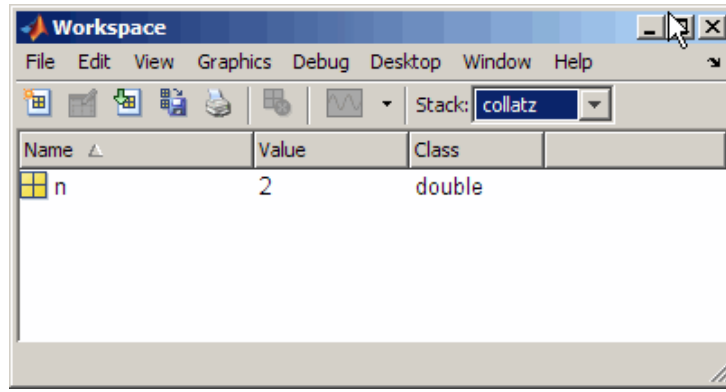
MATLAB returns the expected result

```
n =
 2
```

and displays the debug prompt, `K>>`.

Viewing Values in the Workspace Browser and Array Editor

You can view the value of variables in the **Value** column of the Workspace browser. The Workspace browser displays all variables in the current workspace. Use the **Stack** in the Workspace browser to change to another workspace and view its variables.

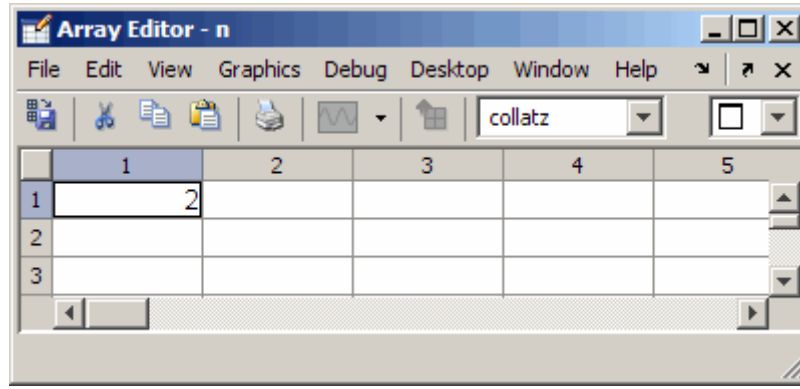


The **Value** column does not show all details for all variables. To see details, double-click a variable in the Workspace browser. The Array Editor opens, displaying the content for that variable. You can open the Array Editor directly for a variable using `openvar`.

To see the value of `n` in the Array Editor for the example, type

```
openvar n
```

and the Array Editor opens, showing that n = 2 as expected.



Evaluating a Selection

Select a variable or equation in an M-file in the Editor/Debugger. Right-click and select **Evaluate Selection** from the context menu (for a single-button mouse, use **Ctrl**+click). MATLAB displays the value of the variable or equation in the Command Window. You cannot evaluate a selection while MATLAB is busy, for example, running an M-file.

Examining Values in the Example

Step from line 9 through line 13 in `collatz`. Step again, and the pause indicator jumps to line 17, just after the `if` loop, as expected. Step again, to line 18, check the value of `sequence` in line 17 and see that the array is

```
2   1
```

as expected for n = 2. Step again, which moves the pause indicator from line 18 to line 11. At line 11, step again. Because `next_value` is now 1, the `while` loop ends. The pause indicator is at line 11 and appears as a green down arrow ⬇. This indicates that processing in the called function is complete and program control will return to the calling program. Step again from line 11 in `collatz` and execution is now paused at line 10 in `collatzplot`.

Note that instead of stepping through `collatz`, the called function, as was just done in this example, you can step out from a called function back to the

calling function, which automatically runs the rest of the called function and returns to the next line in the calling function. To step out, use the Step Out button or type `dbstep out` in the Command Window.

In `collatzplot`, step again to advance to line 11, then line 12. The variable `seq_length` in line 11 is a vector with the elements

```
1   2
```

which is correct.

Finally, step again to advance to line 13. Examining the values in line 12, `N = 2` as expected, but the second variable, `plot_seq`, has two values, where only one value is expected. While the value for `plot_seq` is as expected

```
2   1
```

it is the incorrect variable for plotting. Instead, `seq_length(N)` should be plotted.

Correcting Problems and Ending Debugging

These are some of the ways to correct problems and end the debugging session:

Many of these features are used in "Completing the Example" on page 6-121.

Changing Values and Checking Results

While debugging, you can change the value of a variable in the current workspace to see if the new value produces expected results. While the

program is paused, assign a new value to the variable in the Command Window, Workspace browser, or Array Editor. Then continue running or stepping through the program. If the new value does not produce the expected results, the program has a different problem.

Ending Debugging

After identifying a problem, end the debugging session. You must end a debugging session if you want to change and save an M-file to correct a problem, or if you want to run other functions in MATLAB.

Note It is recommended that you quit debug mode before editing an M-file. If you edit an M-file while in debug mode, you can get unexpected results when you run the file. If you do edit an M-file while in debug mode, breakpoints turn gray, indicating that results might not be reliable. See "Valid (Red) and Invalid (Gray) Breakpoints" on page 6-109 for details.

If you attempt to save an edited M-file while in debug mode, a dialog box appears allowing you to exit debug mode and save the file.

To end debugging, click the Exit Debug Mode button  , or select **Exit Debug Mode** from the **Debug** menu.

You can instead use the function `dbquit` or the **Shift+F5** keyboard shortcut to end debugging.

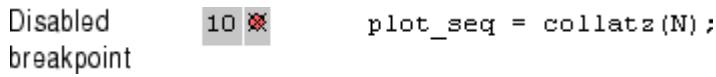
After quitting debugging, pause indicators in the Editor/Debugger display no longer appear, and the normal prompt `>>` appears in the Command Window instead of the debugging prompt, `K>>`. You can no longer access the call stack.

Disabling and Clearing Breakpoints

Disable a breakpoint to temporarily ignore it. Clear a breakpoint to remove it.

Disabling and Enabling Breakpoints. You can disable selected breakpoints so the program temporarily ignores them and runs uninterrupted, for example, after you think you identified and corrected a problem. This is especially useful for conditional breakpoints — see “Conditional Breakpoints” on page 6-126.

To disable a breakpoint, right-click the breakpoint icon and select **Disable Breakpoint** from the context menu, or click anywhere in a line and select **Enable/Disable Breakpoint** from the **Debug** or context menu. You can also disable a conditional breakpoint by clicking the breakpoint icon. This puts an `X` through the breakpoint icon as shown here.



After disabling a breakpoint, you can enable it to make it active again, or clear it. To enable it, right-click the breakpoint icon and select **Enable Breakpoint** from the context menu, or click anywhere in a line and select **Enable/Disable Breakpoint** from the **Breakpoints** or context menu. The `X` no longer appears on the breakpoint icon and program execution will pause at that line.

When you run `dbstatus`, the resulting message for a disabled breakpoint is

```
Breakpoint on line 10 has conditional expression 'false'.
```

Clearing (Removing) Breakpoints. All breakpoints remain in a file until you clear (remove) them or until they are automatically cleared. Clear a breakpoint after determining that a line of code is not causing a problem.

To clear a breakpoint in the Editor/Debugger, click anywhere in a line and select **Set/Clear Breakpoint** from the **Debug** or context menu. The breakpoint for that line is cleared. Another way to clear a breakpoint is to click a standard breakpoint icon, or a disabled conditional breakpoint icon.

To clear all breakpoints in all files, select **Debug > Clear Breakpoints in All Files**, or click its equivalent button on the toolbar.

The function that clears breakpoints is `dbclear`. To clear all breakpoints, use `dbclear all`. For the example, clear all of the breakpoints in `collatzplot` by typing

```
dbclear all in collatzplot
```

Breakpoints are automatically cleared when you

- End the MATLAB session
- Clear the M-file using `clear name` or `clear all`

Note When `clear name` or `clear all` is in a statement in an M-file that you are debugging, it clears the breakpoints.

Saving Breakpoints

You can use the `s=dbstatus` syntax and then save `s` to save the current breakpoints to a MAT-file. At a later time, you can load `s` and restore the breakpoints using `dbstop(s)`. For more information, including an example, see the `dbstatus` reference page.

Correcting an M-File

To correct a problem in an M-file,

1. Quit debugging.

   Do not make changes to an M-file while MATLAB is in debug mode. If you do edit an M-file while in debug mode, breakpoints turn gray, indicating that results might not be reliable. See“Valid (Red) and Invalid (Gray) Breakpoints” on page 6-109 for details.

2. Make changes to the M-file.

3. Save the M-file.

4. Set, disable, or clear breakpoints, as appropriate.

5. Run the M-file again to be sure it produces the expected results.

Completing the Example

To correct the problem in the example, do the following:

1. End the debugging session. One way to do this is to select **Exit Debug Mode** from the **Debug** menu.

2. In `collatzplot.m` line 12, change the string `plot_seq` to `seq_length(N)` and save the file.

3. Clear the breakpoints in `collatzplot.m`. One way to do this is by typing

   ```
   dbclear all in collatzplot
   ```

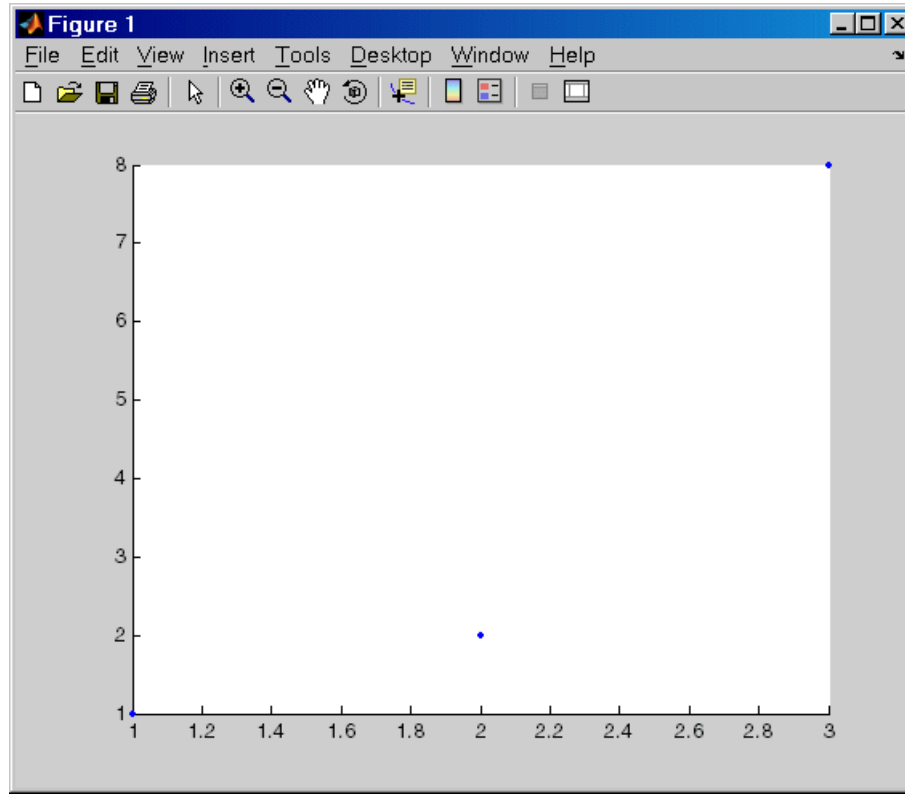
   in the Command Window.

4. Run `collatzplot` for `m` = 3 by typing

   ```
   collatzplot(3)
   ```

   in the Command Window.

5. Verify the result. The figure shows that the length of the Collatz series is 1 when `n` = 1, 2 when `n` = 2, and 8 when `n` = 3, as expected.



6 Test the function for a slightly larger value of `m`, such as 6, to be sure the results are still accurate. To make it easier to verify `collatzplot` for `m = 6` as well as the results for `collatz`, add this line at the end of `collatz.m`

```
sequence
```

which displays the series in the Command Window. The results for when `n = 6` are

```
sequence =

     6     3    10     5    16     8     4     2     1
```

Then run `collatzplot` for `m = 6` by typing

```
collatzplot(6)
```

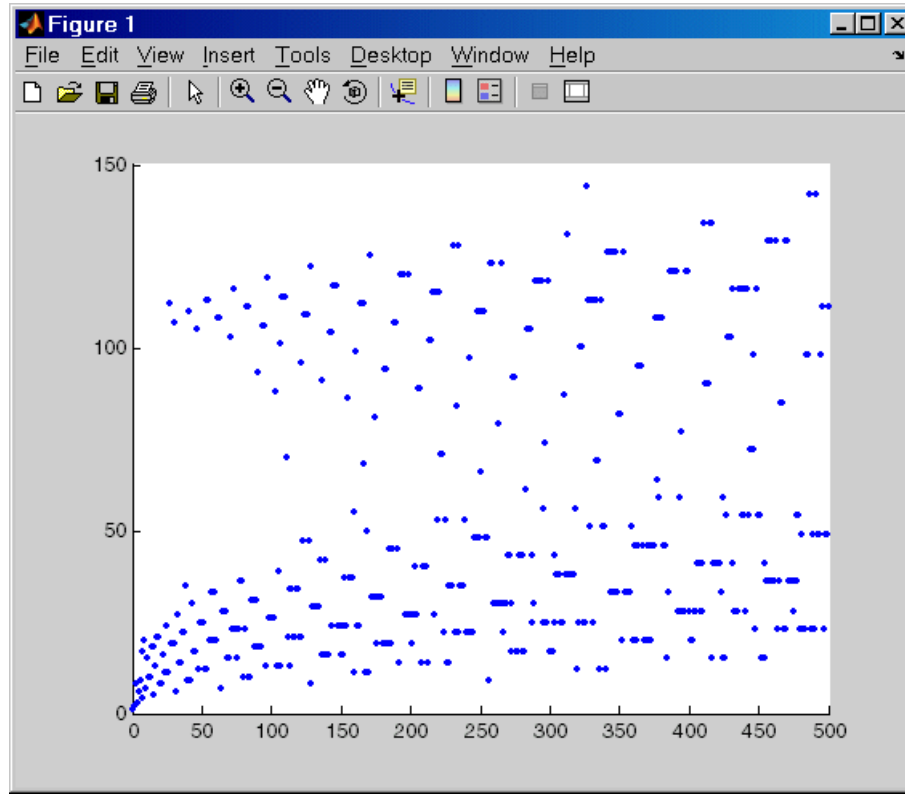
7 To make debugging easier, you ran `collatzplot` for a small value of `m`. Now that you know it works correctly, run `collatzplot` for a larger value to produce more interesting results. Before doing so, you might want to disable output for the line you just added in step 6, line 19 of `collatz.m`, by adding a semicolon to the end of the line so it appears as

```
sequence;
```

Then run

```
collatzplot(500)
```

The following figure shows the lengths of the Collatz series for n = 1 through n = 500.



Running Sections in M-Files That Have Unsaved Changes

It is a good practice to make changes to an M-file after you quit debugging, and to save the changes and then run the file. Otherwise, you might get unexpected results. But there are situations where you might want to experiment during debugging, to make a change to a part of the file that has not yet run, and then run the remainder of the file without saving the change.

To do this, while stopped at a breakpoint, make a change to a part of the file that has not yet run. Breakpoints will turn gray, indicating they are invalid. Then select all of the code after the breakpoint, right-click, and

choose **Evaluate Selection** from the context menu. You can also use cell mode to do this.

Conditional Breakpoints

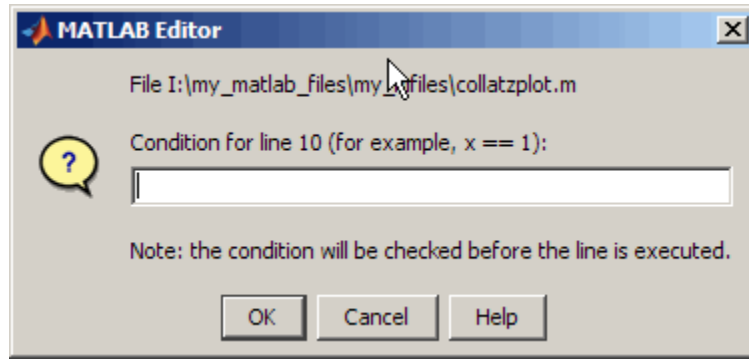
Set conditional breakpoints to cause MATLAB to stop at a specified line in a file only when the specified condition is met. One particularly good use for conditional breakpoints is when you want to examine results after a certain number of iterations in a loop. For example, set a breakpoint at line 10 in `collatzplot`, specifying that MATLAB should stop only if `N` is greater than or equal to `2`. This section covers the following topics:

Setting Conditional Breakpoints

To set a conditional breakpoint, follow these steps:

1. Click in the line where you want to set the conditional breakpoint. Then select **Set/Modify Conditional Breakpoint** from the **Debug** or context menu. If a standard breakpoint already exists at that line, use this same method to make it conditional.

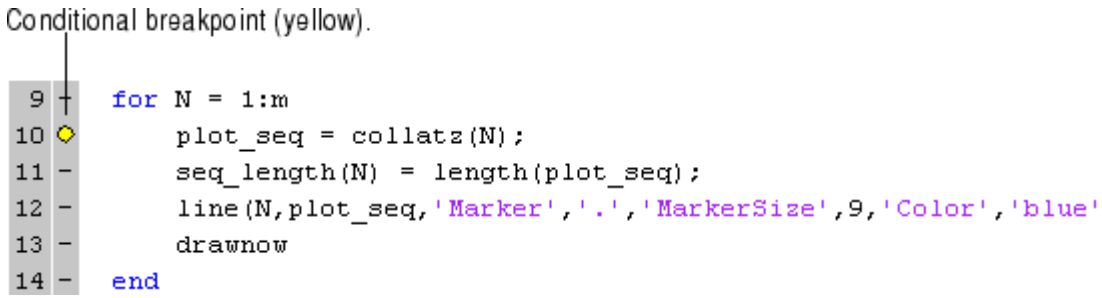
The **MATLAB Editor** conditional breakpoint dialog box opens as shown in this example.



2 Type a condition in the dialog box, where a condition is any legal MATLAB expression that returns a logical scalar value. Click **OK**. As noted in the dialog box, the condition is evaluated before running the line. For the example, at line 10 in `collatzplot`, enter

```
N>=2
```

as the condition. A yellow breakpoint icon (indicating the breakpoint is conditional) appears in the breakpoint alley at that line.



When you run the file, MATLAB enters debug mode and pauses at the line only when the condition is met. In the `collatzplot` example, MATLAB runs through the `for` loop once and pauses on the second iteration at line 10 when `N` is `2`. If you continue executing, MATLAB pauses again at line 10 on the third iteration when `N` is `3`.

Copying, Modifying, Disabling, and Clearing Conditional Breakpoints

To copy a conditional breakpoint, right-click the icon in the breakpoint alley and select **Copy** from the context menu. Then right-click in the breakpoint alley at the line where you want to paste the conditional breakpoint and select **Paste** from the context menu.

Modify the condition for the breakpoint in the current line by selecting **Set/Modify Conditional Breakpoint** from the **Debug** or context menu.

Click a conditional breakpoint icon to disable it. Click a disabled conditional breakpoint to clear it.

Function Alternative for Conditional Breakpoints

Use the `dbstop` function with appropriate arguments to set conditional breakpoints from the Command Window, and use `dbclear` to clear them. Use `dbstatus` to view the breakpoints currently set, including any conditions, which are listed in the `expression` field. If no condition exists, the value in the `expression` field is `[ ]` (empty). For details, see the function reference pages: `dbstop`, `dbclear`, and `dbstatus`.

Breakpoints in Anonymous Functions

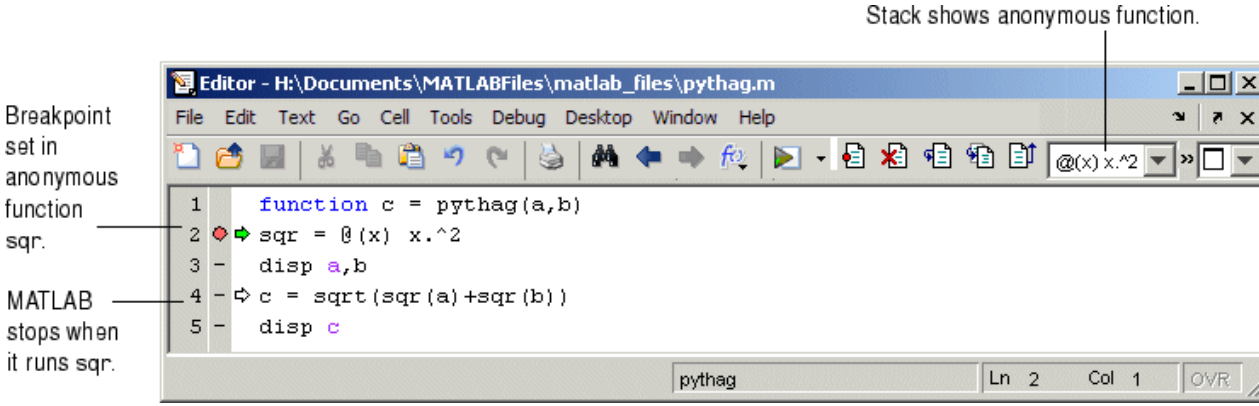
There can be multiple breakpoints in an M-file line that contains anonymous functions. There can be a breakpoint for the line itself (MATLAB stops at the start of the line), as well as a breakpoint for each anonymous function in that line. When you add a breakpoint to a line containing an anonymous function, the Editor/Debugger asks exactly where in the line you want to add the breakpoint. If there is more than one breakpoint in a line, the breakpoint icon is blue .

When there are multiple breakpoints set on a line, the icon is always blue, regardless of the status of any of the breakpoints on the line. Position the mouse on the icon and a tooltip displays information about all breakpoints in that line.

To perform a breakpoint action for a line that can contain multiple breakpoints, such as **Clear Breakpoint**, right-click the breakpoint alley at

that line and select the action. MATLAB prompts you to specify the exact breakpoint on which to act in that line.

When you set a breakpoint in an anonymous function, MATLAB stops when the anonymous function is called. The following illustration shows the Editor/Debugger when you set a breakpoint in the anonymous function `sqr` in line 2, and then run the file. MATLAB stops when it runs `sqr` in line 4. After you continue execution, MATLAB stops again when it runs `sqr` the second time in line 4. Note that the **Stack** display shows the anonymous function.



Error Breakpoints

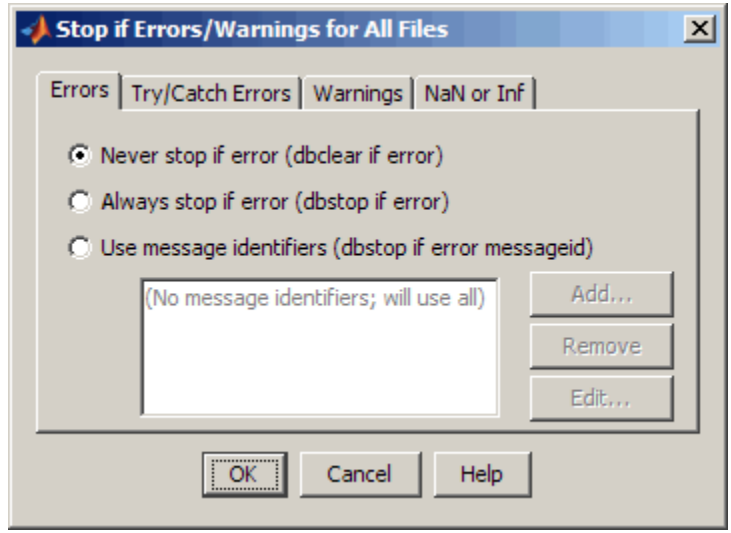
Set error breakpoints to stop program execution and enter debug mode when MATLAB encounters a problem. Unlike standard and conditional breakpoints, you do not set these breakpoints at a specific line in a specific file. Rather, once set, MATLAB stops at any line in any file when the error condition specified via the error breakpoint occurs. MATLAB then enters debug mode and opens the file containing the error, with the pause indicator at the line containing the error. Files open only when the you select **Debug > Open M-Files** . Error breakpoints remain in effect until you clear them or until you end the MATLAB session. You can set error breakpoints from the **Debug** menu in any desktop tool. This section covers the following topics:

- “Setting Error Breakpoints” on page 6-130
- “Error Breakpoint Types and Options” on page 6-130

• "Function Alternative for Error Breakpoints" on page 6-132

Setting Error Breakpoints

To set error breakpoints, select **Debug > Stop if Errors/Warnings**. In the resulting **Stop if Errors/Warnings for All Files** dialog box, specify error breakpoints on all appropriate tabs and click **OK**. To clear error breakpoints, select the **Never stop if ...** option for all appropriate tabs and click **OK**.



For example, to pause execution when a warning occurs, select the **Warnings** tab, and from it select **Always stop if warning**, then click **OK**. When you run an M-file and MATLAB produces a warning, execution pauses, MATLAB enters debug mode, and the file opens in the Editor/Debugger at the line that produced the warning. To remove the warning breakpoint, select **Never stop if warning** in the **Warnings** tab and click **OK**.

Error Breakpoint Types and Options

The four basic types of error breakpoints you can set are **Errors**, **Try/Catch Errors**, **Warnings**, and **NaN or Inf**. Select the **Always stop if ...** option for each tab to set that type of breakpoint. Select the **Use message identifiers ...** option to limit each type of error breakpoint (except NaN or Inf) so that execution stops only for specific errors.

Errors. When an error occurs, execution stops, unless the error is in a `try...catch` block. MATLAB enters debug mode and opens the M-file to the line in the `try` portion of the block that produced the error. You cannot resume execution.

Try/Catch Errors. When an error occurs in a `try...catch` block, execution pauses. MATLAB enters debug mode and opens the M-file to the line that produced the error. You can resume execution or use debugging features.

Warnings. When a warning is produced, MATLAB pauses, enters debug mode, and opens the M-file, paused at the line that produced the warning. You can resume execution or use debugging features.

NaN or Inf. When an operator, function call, or scalar assignment produces a `NaN` (not-a-number) or `Inf` (infinite) value, MATLAB pauses, enters debug mode, and opens the M-file, paused immediately after the line that encountered the value. You can resume execution or use debugging features.

Use Message Identifiers. Execution stops only when MATLAB encounters one of the specified errors. This option is not available for the **NaN or Inf** type of error breakpoint. To use this feature:

1. Select the **Errors**, **Try/Catch Errors,** or **Warnings** tab.
2. Select the **Use Message Identifiers** option.
3. Click **Add**.
4. In the resulting **Add Message Identifier** dialog box, supply the message identifier of the specific error you want to stop for, where the identifier is of the form `component:message`, and click **OK**.
5. The message identifier you added appears in the **Stop If Errors/Warnings for All Files** dialog box, where you click **OK**.

   You can add multiple message identifiers, and edit or remove them.

One way to obtain an error message identifier generated by a MATLAB function for example, is to produce the error, and then run the `lasterror` function. MATLAB returns the error message and identifier. Copy the identifier from the Command Window output and paste it into the **Add**

Message Identifier dialog box. An example of an error message identifier is `MATLAB:UndefinedFunction`. Similarly, to obtain a warning message identifier, produce the warning and then run `[m,id] = lastwarn`; MATLAB returns the last warning identifier to `id`. An example of a warning message identifier is `MATLAB:divideByZero`.

Function Alternative for Error Breakpoints

The function equivalent for each option appears in the **Stop if Errors/Warnings for All Files** dialog box. For example, the function equivalent for **Always stop if error** is `dbstop if error`. Use the `dbstop` function with appropriate arguments to set error breakpoints from the Command Window, and use `dbclear` to clear them. Use `dbstatus` to view the error breakpoints currently set. Error breakpoints are listed in the `cond` field and message identifiers for breakpoints are listed in the `identifier` field of the `dbstatus` output.

Using Cells for Rapid Code Iteration and Publishing Results

In this section...

What Are Cells?

M-files often have a natural structure consisting of multiple sections. Especially for larger files, you typically focus efforts on a single section at a time, working with the code in just that section. Similarly, when conveying information about your M-files to others, often you describe the sections of the code. To facilitate these processes, use M-file *cells*, where *cell* means a section of code. Specifically, MATLAB uses cells for

- Rapid code iteration in the Editor/Debugger — This makes the experimental phase of your work with M-file scripts easier. The next section, "Rapid Code Iteration Overview" on page 6-133, outlines the process, and is followed by details for defining, evaluating, and modifying values in cells.
- Publishing M-files — This allows you to include code and results in a presentation format such as HTML. Publishing using cells requires you to define cells using the same instructions as for rapid code iteration. You can also make use of the cell navigation and evaluation features used for rapid code iteration. See "Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells" on page 8-2 for complete details.

Rapid Code Iteration Overview

When working with MATLAB, you often experiment with your code — modifying it, testing it, and updating it — until you have an M-file that does what you want. Use the MATLAB Editor/Debugger cells features with M-file scripts to facilitate this process. You can also use cell features with function M-files, but there are some restrictions — see "Using Cells in Function M-Files" on page 6-144.

If you have an active Internet connection, you can watch the Rapid Code Iteration Using Cells video demo for an overview of the major functionality.

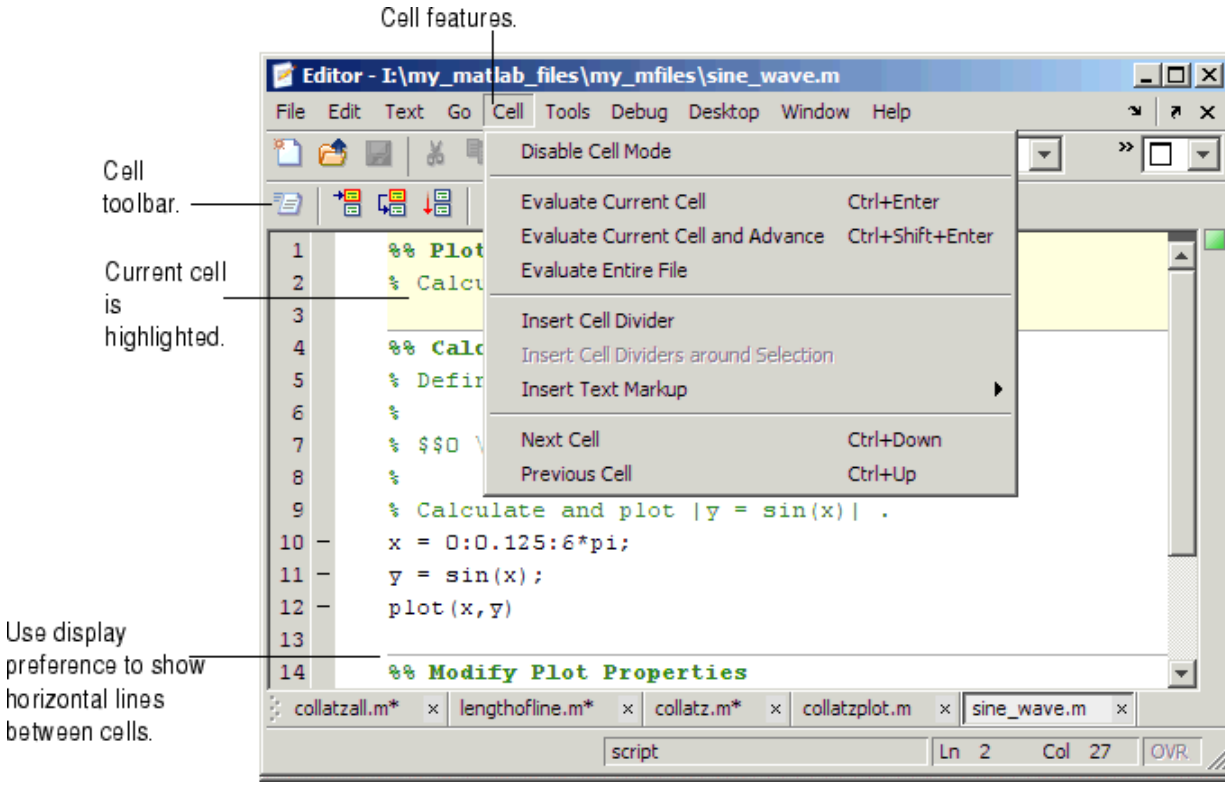
This is the overall process of using cells for rapid code iteration:

1. In the MATLAB Editor/Debugger, enable cell mode by selecting **Cell > Enable Cell Mode**. Items in the **Cell** menu become selectable. The cell toolbar appears, unless you had previously hidden it. With cell mode enabled, hide or show the toolbar by right-clicking in the Editor/Debugger menu bar or toolbars and selecting **Cell Toolbar** from the context menu.

2. Define the boundaries of the cells in an M-file script using cell features. Cells are denoted by a specialized comment syntax, %%. For details, see "Defining Cells" on page 6-135.

3. Once you define the cells, use cell features to navigate quickly from cell to cell in your file, evaluate the code in a cell in the base workspace, and view the results. To facilitate experimentation, use cell features to modify values in cells and then reevaluate them, to see how different values impact the result. For details, see "Navigating and Evaluating with Cells" on page 6-139.

4. Cells also facilitate sharing your code and results via cell publishing to a presentation format. For details, see "Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells" on page 8-2.



Defining Cells

Cell features operate on cells, where a cell is contiguous lines of code you want to evaluate as a whole in an M-file script. To define a cell, first be sure that cell mode is enabled (see step). Position the cursor just before the line at which you want to start the cell and select **Cell > Insert Cell Divider** or click the Insert Cell Divider button . MATLAB inserts a line after the cursor that consists of two percent signs (%%), which is the “start new cell” indicator to MATLAB. A cell consists of the line starting with %% and the lines that follow, up to the start of the next cell, which is identified by %% at the start of a new line.

You can also define a cell by entering two percent signs (%%) at the start of the line where you want to begin the new cell. Alternatively, select the

lines of code you want in a cell and then select **Cell > Insert Cell Dividers Around Selection**.

You can define a cell at the start of a new empty file, enter code for the cell, define the start of the next cell, enter its code, and so on. Redefine cells by defining new cells, removing existing cells, and moving lines of code.

You can set an Editor/Debugger preference to show a faint gray horizontal line (rule) above each cell that helps distinguish the cells. Select **File > Preferences > Editor/Debugger > Display** and in **Cell display options**, use **Show lines between cells**. The horizontal lines do not appear in the M-file when you print it.

MATLAB does not execute the code in lines beginning with `%%`, so be sure to put any executable code for the cell on the following line. For program control statements, such as `if ... end`, a cell must contain both the opening and closing statements, that is, it must contain both the `if` and the `end` statements.

Note that the first cell in a file does not have to begin with `%%`. MATLAB automatically understands any lines above the first `%%` line to be a cell. If there are no cell dividers in an M-file, MATLAB understands the entire file to be a single cell.

Cell Titles and Highlighting

After the `%%`, type a space, followed by a description of the cell. The Editor/Debugger emphasizes the special meaning of the start of a cell by making any text following the percent signs appear bold. The text on the `%%` line is called the *cell title* (like a section title). Including text in cell titles is optional, however, they improve readability of the file and are used for cell publishing features.

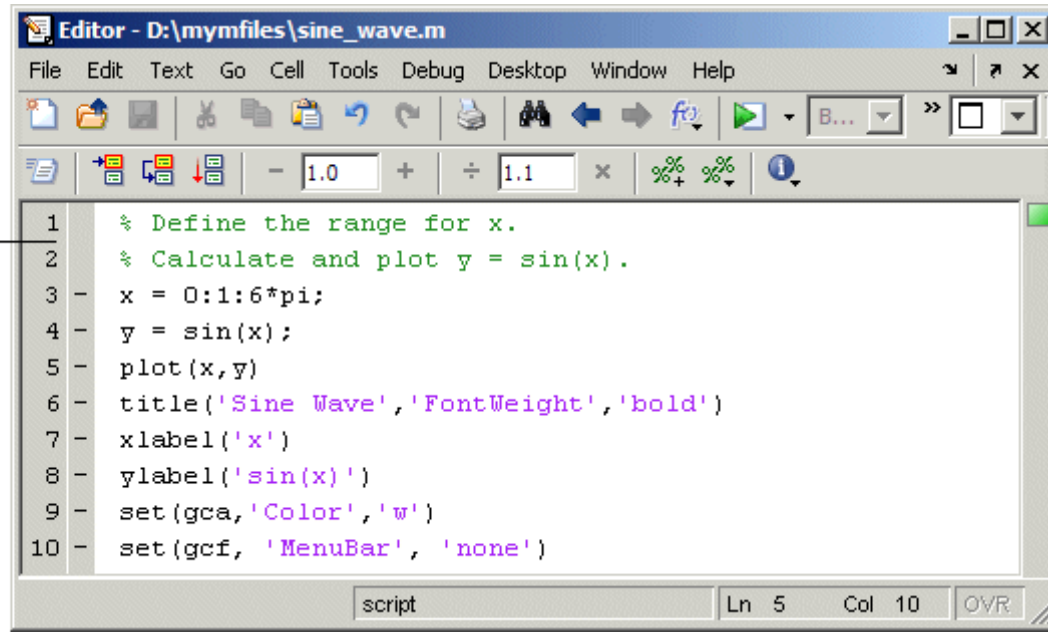
When the cursor is positioned in any line within a cell, the Editor/Debugger highlights the entire cell with a yellow background. This identifies it as the *current cell*. For example, it is used when you select the **Evaluate Current Cell** option on the **Cell** menu. If you do not want yellow highlighting for the current cell, change it using preferences. Select **File > Preferences > Editor/Debugger > Display** and change the appropriate **Cell display options**.

Example — Define Cells

This example defines two cells for a simple M-file called `sine_wave`, shown in the following code and figure. The first cell creates the basic results, while the second label the plot. The two cells in this example allow you to experiment with the plot of the data first, and then when that is final, change the plot properties to affect the style of presentation.

```
% Define the range for x.
% Calculate and plot y = sin(x).
x = 0:1:6*pi;
y = sin(x);
plot(x,y)
title('Sine Wave','FontWeight','bold')
xlabel('x')
ylabel('sin(x)')
set(gca,'Color','w')
set(gcf, 'MenuBar', 'none')
```

M-file before defining cells.



1 Select **Cell > Enable Cell Mode**, if it is not already enabled.

2. Position the cursor at the start of the first line. Select **Cell > Insert Cell Divider**.

   The Editor/Debugger inserts `%%` as the first line and moves the rest of the file down one line. All lines are highlighted in yellow, indicating that the entire file is a single cell, unless you do not have that display preference for cells selected.

3. Enter a cell title following the `%%`. Type a space first, followed by the description.

   ```
   Calculate and Plot Sine Wave
   ```

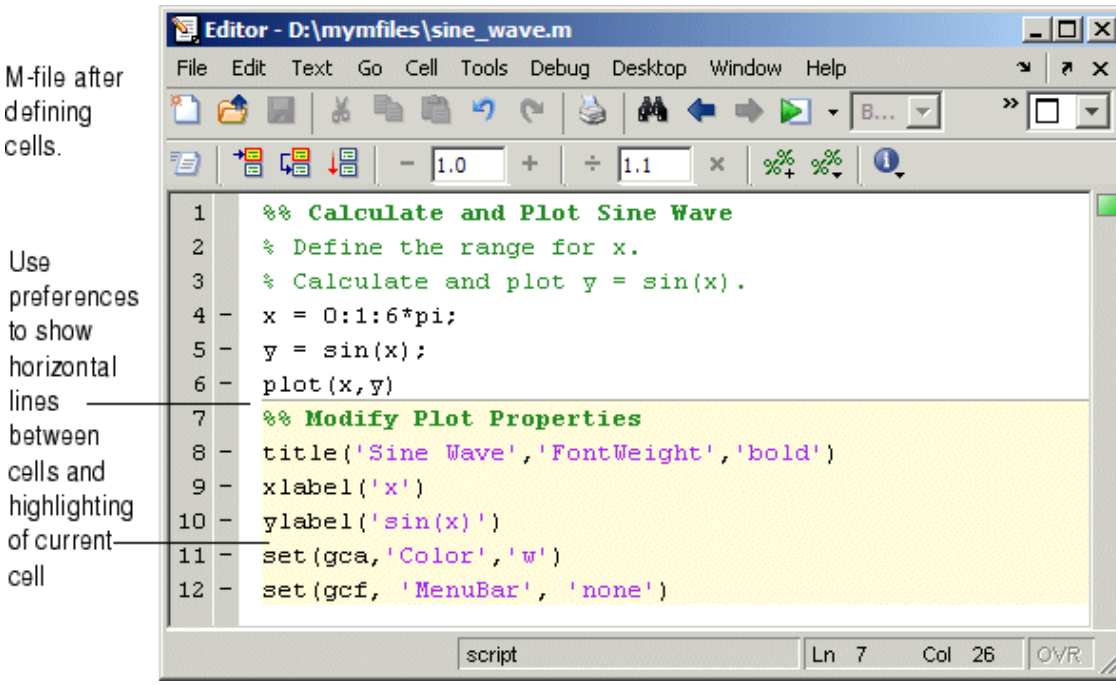
4. Position the cursor at the start of line 7, `title...`. Select **Cell > Insert Cell Divider**.

   The Editor/Debugger inserts a line containing only `%%` at line 7 and moves the remaining lines down by one line. A horizontal line that helps you distinguish the two cells appears above the `%%` line, unless you do not have that display preference for cells selected. Lines 7 through 12 are highlighted in yellow, indicating they comprise the current cell.

5. Enter a cell title for the new cell. On line 7, type a space after the `%%`, followed by the description

   ```
   Modify Plot Properties
   ```

   Save the file. The file appears as shown in this figure.



Removing Cells

To remove a cell, delete one of the percent signs (%) from the line that starts the cell. This changes the line from a cell to a standard comment and merges the cell with the preceding cell. You can also just delete the entire line that contains the %%.

Navigating and Evaluating with Cells

While you develop an M-file, you can use these Editor/Debugger cell features:

Navigating Among Cells in an M-File

To move to the next cell, select **Cell > Next Cell**. To move to the previous cell, select **Cell > Previous Cell**. To move to a specific cell, click the Show Cell Titles button and from it, select the cell title to which you want to move. You can also go to cells by selecting **Edit > Go To**.

Evaluating Cells in an M-File

To evaluate the code in a cell, use the **Cell** menu evaluation items or equivalent buttons in the cell toolbar. When you evaluate a cell, the results display in the Command Window, figure window, or otherwise, depending on the code evaluated.

The cell evaluation features run the code currently shown in the Editor/Debugger, even if the file contains unsaved changes. The file does not have to be on the search path. To evaluate a cell, it must contain all the values it requires, or the values must already exist in the MATLAB workspace.

Note While you can set breakpoints and debug a file containing cells, when you evaluate a file from the **Cell** menu or cell toolbar, breakpoints are ignored. To run the file and stop at breakpoints, use **Run/Continue** in the **Debug** menu. This means you cannot debug while running a single cell.

Evaluate Current Cell. Select **Cell > Evaluate Current Cell** or click the Evaluate Cell button to run the code in the current cell.

Evaluate and Advance. Select **Cell > Evaluate Current Cell and Advance** or click the Evaluate Cell and Advance button to run the code in the current cell and move to the next cell.

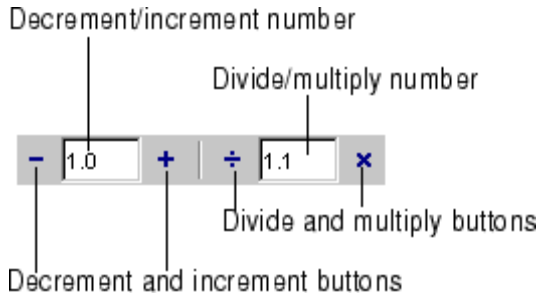
Evaluate File. Select **Cell > Evaluate Entire File** or click the Evaluate Entire File button to run all of the code in the file.

Note A beep means there is an error. See the Command Window for the error message.

Modifying Values in a Cell

You can use cell features to modify numbers in a cell, which also automatically reevaluates the cell. This helps you experiment with and fine-tune your code.

To modify a number in a cell, select the number (or place the cursor near it) and use the value modification tool in the cell toolbar. Using this tool, you can specify a number and press the appropriate math operator to add (increment), subtract (decrement), multiply, or divide the number. The cell then automatically reevaluates.



You can use the numeric keypad operator keys (**-**, **+**, **/**, and *) instead of the operator buttons on the toolbar.

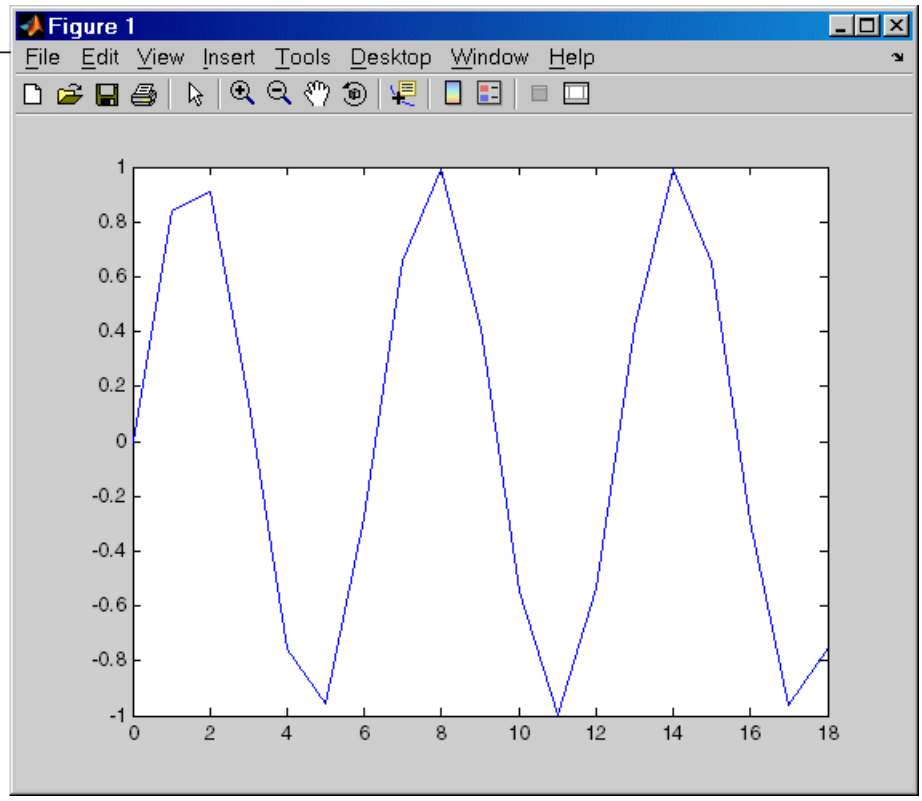
Note MATLAB does not automatically save changes you make to values using the cell toolbar. To save changes, select **File > Save**.

Example — Evaluate Cells

In this example, modify the values for `x` in `sine_wave.m`:

1. Run the first cell in `sine_wav.m`. Click somewhere in the first cell, that is, between lines 1 and 6. Select **Cell > Evaluate Current Cell**. The following figure appears.

Plot generated by running `sine_wave.m`.



2 Assume you want to produce a smoother curve. Use more values for `x` in `0:1:6*pi`. Position the cursor in line 4, next to the `1`. In the cell toolbar, change the `1.1` default multiply/divide by value to `2`. Click the Divide button ÷.

Line 4 becomes

```
4 -   x = 0:0.5:6*pi;
```

and the length of `x` doubles. The plot automatically updates. The curve still has some rough edges.

3 To add more values for `x`, click the Divide button three more times. Line 4 becomes

```
4 -   x = 0:0.0625:6*pi;
```

The curve is smooth, but because there are more values, processing time is slower. It would be better to find a smaller `x` that still produces a smooth curve.

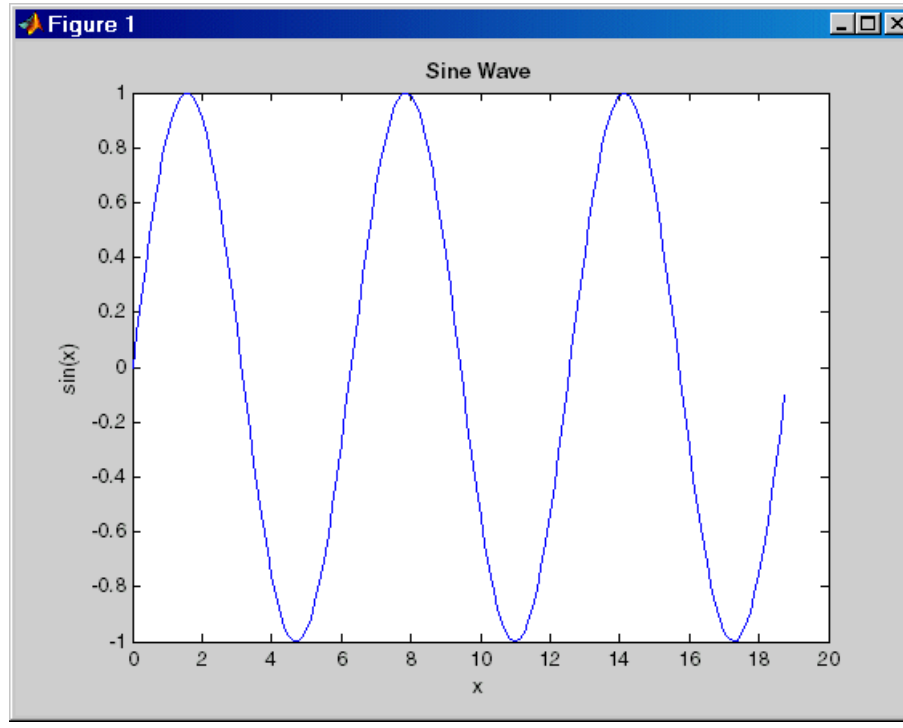
4. In the cell toolbar, click the Multiply button once. The increment for `x` as shown in line 4 changes from `0.0625` to `0.125`.

   The resulting curve is still smooth.

5. Save these changes. Select **File > Save**.

6. Now you can apply the plot properties, defined in the second cell, that is, lines 7 through 12. You do not need to evaluate the entire file to apply the plot properties. Instead, position the cursor in the second cell and use the shortcut **Ctrl+Enter** to evaluate the current cell. (The shortcut appears with the menu item, **Cell > Evaluate Current Cell**.)

MATLAB updates the figure.



Using Cells in Function M-Files

You can define and evaluate cells in function M-files as long as the variables referred to in the cell are in your workspace. For example, this can be useful during debugging. If execution is stopped at a breakpoint, you can define cells and execute them without saving the file. If you are not debugging, add the necessary variables to the base workspace and then execute the cells.

7

Tuning and Managing M-Files

This set of tools provides useful information about the M-files in a directory that can help you refine the files and improve performance. The tools can help you polish M-files before providing them to others to use. If you have an active Internet connection, you can watch the Directory Reports video demo for an overview of the major functionality.

Directory Reports in Current Directory Browser (p. 7-2)	HTML reports about files in the current directory: TODO/FIXME, Help, Contents, Dependency, Coverage (for Profiling), and M-Lint Code Check.
M-Lint Code Check Report (p. 7-16)	Report that identifies potential errors, problems, and opportunities for improvement in your code.
Profiling for Improving Performance (p. 7-27)	Report that identifies where an M-file spends the most time, indicating where to focus when looking for performance improvements.

Directory Reports in Current Directory Browser

In this section...

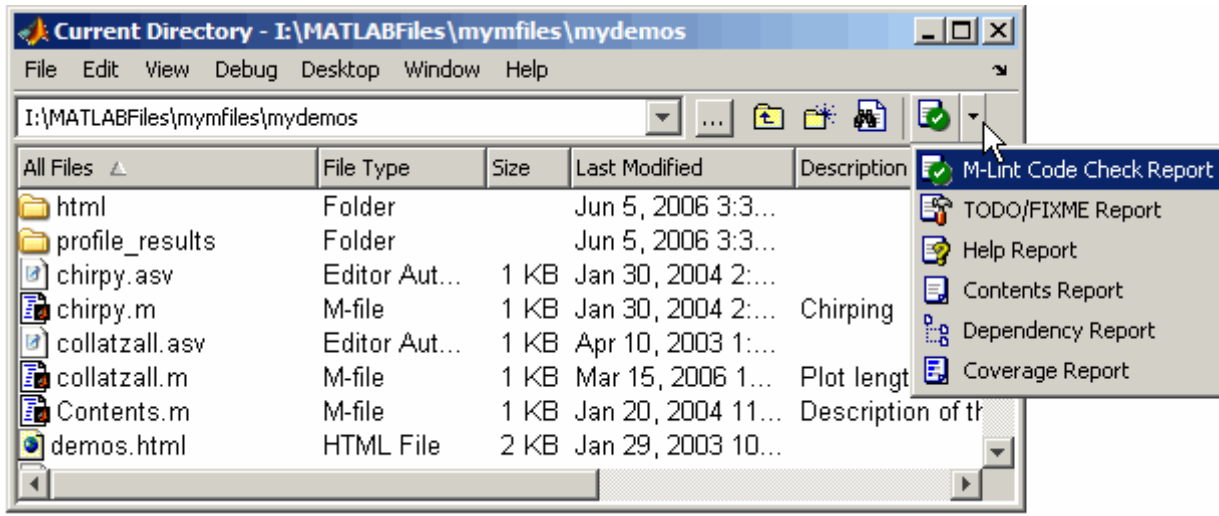
See also another Directory Report, "M-Lint Code Check Report" on page 7-16, and the File Comparisons tool.

Accessing and Using Directory Reports

Directory reports help you refine the M-files in a directory and improve their performance. They are also useful for when you prepare files for use by others, such as for a finished project, to share on MATLAB Central, or for a toolbox to be distributed.

Access directory reports from the MATLAB Current Directory browser. To display the Current Directory browser, select **Desktop > Current Directory**. For more information, see "Current Directory Browser" on page 5-36.

Navigate to the directory whose M-files you want to produce reports about. Then, in the Current Directory browser toolbar, click the down arrow button and select the type of report you want to run for all the M-files in the current directory.



The report you selected appears as an HTML document in the MATLAB Web Browser:

- In a report, click a filename to open that file in the Editor/Debugger, where you can view it or make changes to it. Click a line number to open the file at that line.
- To update a report after making changes to the report options, or after changing any files in the directory, click **Rerun This Report**. Note that this reruns the report for the directory shown in the report, not for the MATLAB current directory.
- While a report is displayed, you can change the MATLAB current directory and then click **Run Report on Current Directory** to generate the same type of report for the new current directory.
- When you run a report, it replaces the report currently displayed. Use the Back and Forward buttons in the toolbar to see a previously run report and then return to the most recent.

You cannot run directory reports when the path is a UNC (Universal Naming Convention) pathname, that is, starts with \\. Instead, use an actual hard drive on your system, or a mapped network drive.

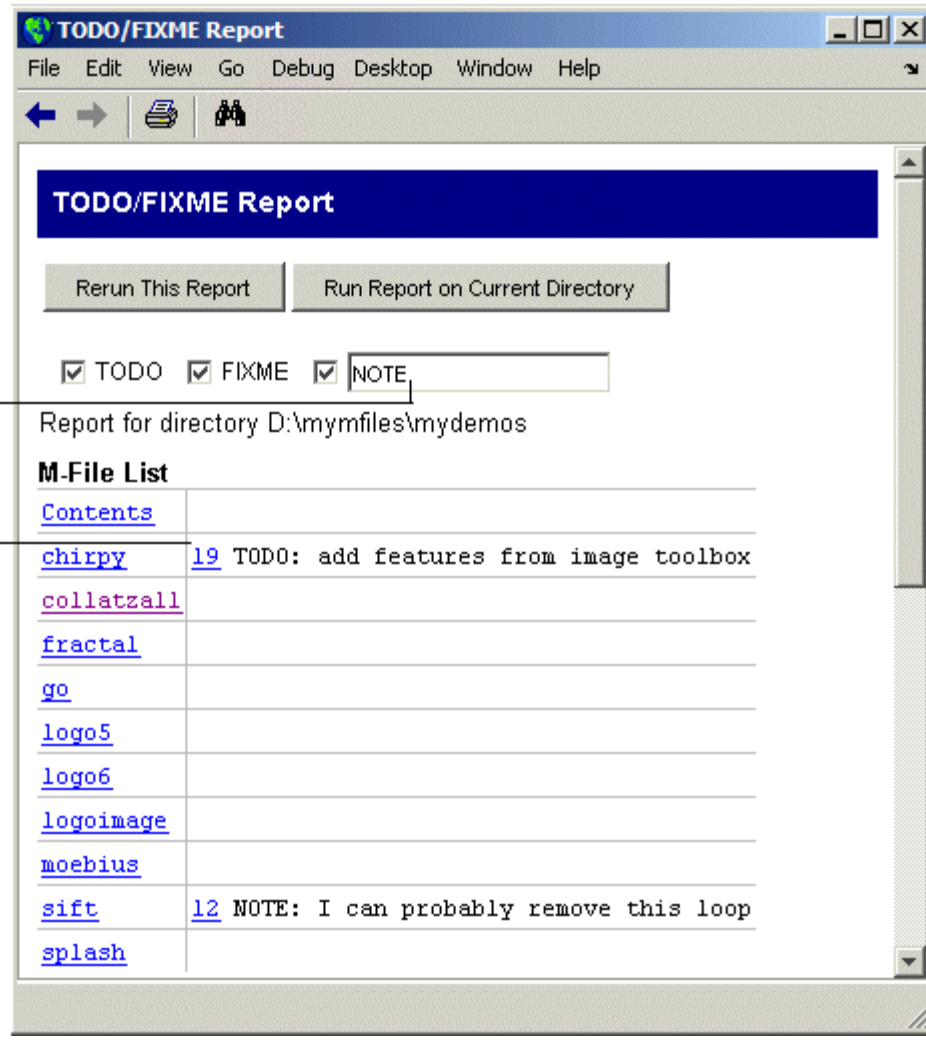
TODO/FIXME Report

The TODO/FIXME Report shows M-files that contain text strings you included as notes to yourself, such as `TODO`. Use this report to easily identify M-files that still require work or some other actions.

To access this report, follow the instructions in “Accessing and Using Directory Reports” on page 7-2.

In the report, select one or more check boxes to display lines containing the specified strings (`TODO` and `FIXME`), and click **Rerun This Report**. You can also select the check box for the text field and enter any text string in the field, such as `NOTE` or `TBD` to identify lines containing that string.

The TODO/FIXME Report identifies lines containing specified text strings, such as TODO and NOTE shown here, so you can easily see reminders you included.



Enter any text string in this field.

Click the line number to open the file at that line in the Editor/Debugger, where you can make changes.

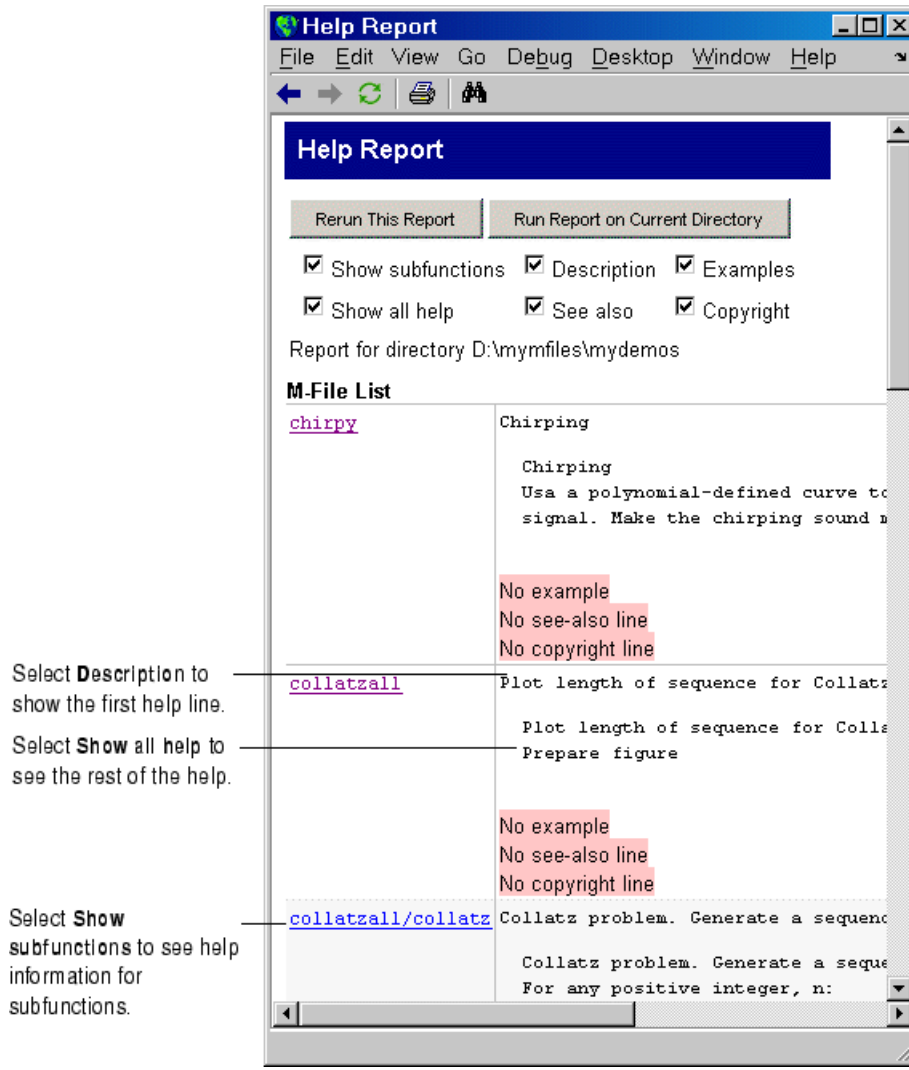
Help Report

The Help Report presents a summary view of the help component of your M-files. In MATLAB, the M-file help component is all contiguous nonexecutable lines (comment lines and blank lines), starting with the second line of a function M-file or the first line of a script M-file. For more information about creating help for your own M-files, see the reference page for the `help` function.

To access this report, follow the instructions in “Accessing and Using Directory Reports” on page 7-2.

Select one or more check boxes to display the specified help information and click **Rerun This Report**.

Use this information to help you identify files of interest or files that lack help information. It is a good practice to provide help for your files not only to help you recall their purpose, but to help others who might use the files.



Show Subfunctions

With **Show subfunctions** selected, the Help Report displays help information for all subfunctions called by each function. Help information for subfunctions is highlighted in gray.

Description

With **Description** selected, the Help Report displays the first line of help in the M-file. If the first comment line is empty, or if there is not a comment before the executable code, **No description line**, highlighted in pink, appears instead.

Examples

With **Examples** selected, the Help Report displays the line number where the examples section of the M-file help begins. The Help Report looks for a line in the M-file help that begins with the string `example` or `Example` and displays any subsequent nonblank comment lines. Select this option to easily locate and go to examples in your M-files.

It is a good practice to include examples in the help for your M-files. If you do not have examples in the help for all your M-files, use this option to identify those without examples. If the report does not find examples in the M-file help, **No example**, highlighted in pink, appears.

Show All Help

With **Show all help** selected, the Help Report displays complete M-file help, which is all contiguous nonexecutable lines (comment lines and blank lines), starting with the second line of a function M-file, or the first line of a script M-file. The M-file help shown also includes overloaded functions and methods, which are not actually part of the M-file help comments, but are automatically generated when `help` runs.

If the comment lines before the executable code are empty, or if there are no comments before the executable code, **No help**, highlighted in pink, appears instead.

See Also

With **See Also** selected, the Help Report displays the line number for the see also line in the M-file help. The see also line in M-file help lists related functions. When MATLAB displays the help for an M-file, any function name listed on the see also line appears as a link you can click to display its help. It is a good practice to include a see also line in the help for your M-files.

The report looks for a line in the M-file help that begins with the string `See also`. If the report does not find a see also line in the M-file help, **No see-also line**, highlighted in pink, appears. This helps you identify those M-files without a see also line, should you want to include one in each M-file.

The report also indicates when an M-file noted in the see also line is not in a directory on the search path. You might want to move that file to a directory that is on the search path. If not, you will not be able to click the link to get help for the file, unless you then add its directory to the path or make its directory become the current directory.

Copyright

With **Copyright** selected, the Help Report displays the line number for the copyright line in the M-file. The report looks for a comment line in the M-file that begins with the string `Copyright` and is followed by `year1-year2` (with no spaces between the years and the hyphen that separates them). It also notes if the end of the date range is not the current year.

It is a good practice to include a copyright line in the help for your M-files, that notes the year you created the file and the current year. For example, for an M-file you created in 2001, include this line

```
% Copyright 2001-2006
```

If the report does not find a copyright line in the M-file help, **No copyright line**, highlighted in pink, appears. This helps you identify those files without a copyright line, should you want to include one in each M-file.

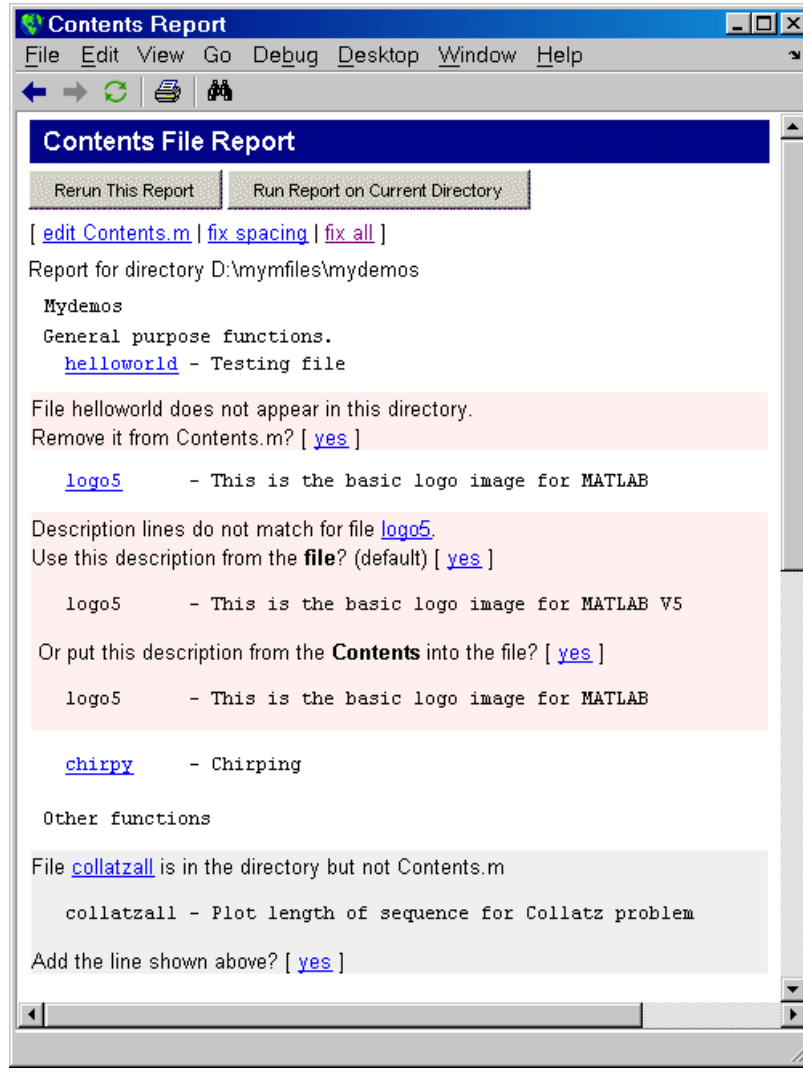
Contents Report

The Contents Report displays information about the integrity of the `Contents.m` file for the directory. A `Contents.m` file includes the filename and a brief description of each M-file in the directory. When you type `help` followed by the directory name, such as `help mydemos`, MATLAB displays the information in the `mydemos/Contents.m` file. For more information, see “Providing Help for Your Program” in the MATLAB Programming documentation.

To access this report, follow the instructions in “Accessing and Using Directory Reports” on page 7-2.

If there is no `Contents.m` file for the directory and you run the Contents Report, the report tells you the `Contents.m` file does not exist and asks if you want to create one. Click **yes** to automatically create the `Contents.m` file. Edit the `Contents.m` file in the Editor/Debugger to include the names of files you plan to create, or to remove files that you do not want to expose when displaying help for the directory, such as files for internal use.

You need to update the `Contents.m` file to reflect changes you make to files in the directory. For example, when you remove a file from a directory, remove its entry from the `Contents.m` file. The Contents Report helps you to maintain the `Contents.m` file. It displays discrepancies between the `Contents.m` file and the M-files in the directory.



Use the links displayed for each line, or edit the `Contents.m` file directly, or edit the M-files to make the changes. To make all of the suggested changes at once, click **fix all**. To automatically align the filenames and descriptions in the `Contents.m` file, click **fix spacing**.

If you always want the `Contents.m` file to reflect all files in the directory, you can automatically generate a new `Contents.m` file rather than changing the file based on the Contents Report. To do this, first delete the existing `Contents.m` file, run the Contents Report, and click `yes` when prompted for MATLAB to automatically create one.

Messages in the Contents File Report

No Contents File. This message appears if there is no `Contents.m` file in the directory. Click **yes** to automatically create a `Contents.m` file, which contains the filenames and descriptions for all M-files in the directory.

```
No Contents.m file. Make one? [ yes ]
```

File Not Found. This message appears when a file included in `Contents.m` is not in the directory. These messages are highlighted in pink. For example, a message such as

```
File helloworld does not appear in this directory.
Remove it from Contents.m? [ yes ]
```

means the `Contents.m` file includes an entry for `helloworld`, but that file is not in the directory. This might be because you removed the file `helloworld`, or you manually added it to `Contents.m` because you planned to create the file but have not as yet, or you renamed `helloworld`.

Description Lines Do Not Match. This message appears when the description line in the M-file help does not match the description provided for the M-file in `Contents.m`. These messages are highlighted in pink. Click **yes** to replace the description in the `Contents.m` file with the description from the M-file. Or select the option to replace the description line in the M-file help using the description for that file in `Contents.m`.

```
Description lines do not match for file logo5.
Use this description from the file? (default) [ yes ]
 logo5      - This is the basic logo image for MATLAB V5
Or put this description from the Contents into the file? [ yes ]
 logo5 - This is the basic logo image for MATLAB
```

Files Not In Contents.m. This message appears when a file in the directory is not in `Contents.m`. These messages are highlighted in gray. Click **yes** to add the filename and its description line from the M-file help to the `Contents.m` file.

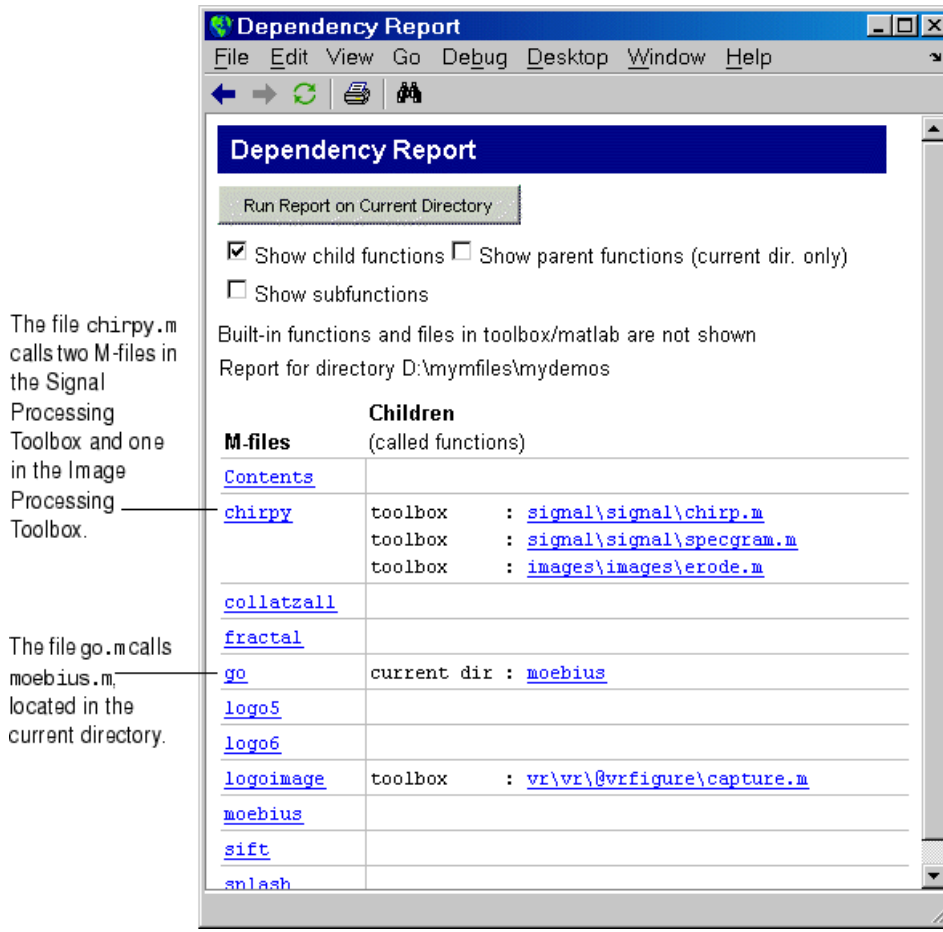
```
collatzall is in the directory but not Contents.m
 collatzall - Plot length of sequence for Collatz problem
Add the line shown above? [ yes ]
```

Dependency Report

The Dependency Report shows dependencies among M-files in a directory. This helps you determine all the M-files you need to provide when you tell someone to run a particular M-file. If you do not provide all the dependent M-files along with the M-file you want them to run, they will not be able run the file. The report does not list as dependencies the M-files in the `toolbox/matlab` directory because every MATLAB user already has those files.

To access this report, follow the instructions in “Accessing and Using Directory Reports” on page 7-2. You can also access the report from the Editor/Debugger **Tools** menu.

Select **Show child functions** to see a list of all M-files (children) called by each M-file in the directory (parent). The report also indicates where each child function resides, for example, in a specified toolbox. If a child function’s location is listed as unknown, it could be because the child function is not on the search path or in the current directory.



The Dependency Report is similar to running the `depfun` function, although the two do not provide the exact same results. For performance purposes, the Dependency Report limits the functions considered.

Select **Show parent functions** to list the M-files that call each M-file. The report limits the parent (calling) functions to those in the current directory. Select **Show subfunctions** to include subfunctions in the report. Subfunctions are listed directly after the main function and are highlighted in gray.

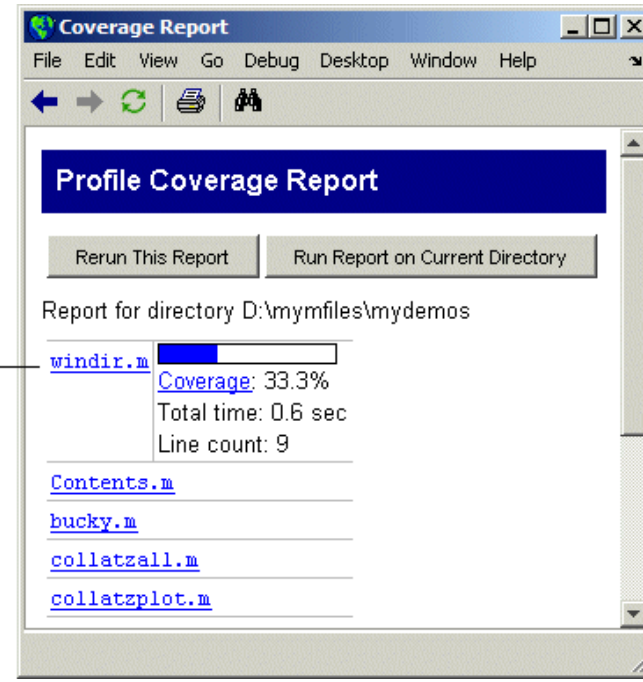
Coverage Report

Run the Coverage Report after you run the Profiler to identify how much of a file ran when it was profiled. For example, when you have an `if` statement in your code, that section might not run during profiling, depending on conditions.

You can run the Coverage Report from the Profiler, or follow these steps:

1. In the MATLAB desktop, select **Desktop > Profiler**. Profile an M-file in the Profiler. For detailed instructions, see "Profiling for Improving Performance" on page 7-27.

2. In the Current Directory browser, select `Coverage Report`. The **Coverage Report** appears, providing a summary of coverage for the M-file you profiled.



The Coverage Report shows the percentage of a file that ran when it was profiled.

3. Click the **Coverage** link to see the Profile Detail Report for the file.

M-Lint Code Check Report

In this section...

Running the M-Lint Code Check Directory Report

The M-Lint Code Check Report displays potential errors and problems, as well as opportunities for improvement in your code. The term "lint" is the name given to similar tools used with other programming languages such as C. In MATLAB, M-Lint produces a message for each line of an M-file that it determines might be improved. For example, a common M-Lint message is that a variable `foo` in line 12 is defined but never used in the M-file.

To run the M-Lint code check directory report, follow these steps:

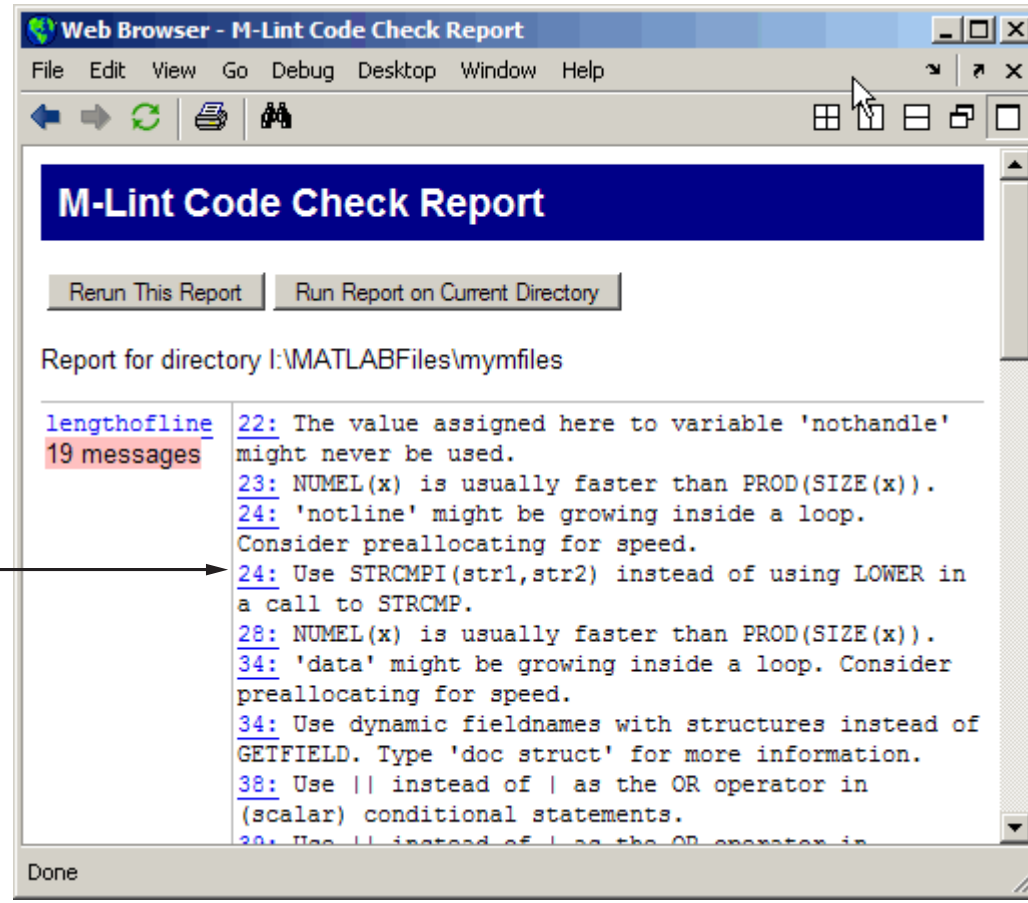
1. In the Current Directory browser, navigate to the directory that contains the M-files you want to check with M-Lint. To use the example shown in this documentation, `lengthofline.m`, you can change the current directory by running

   ```
   cd(fullfile(matlabroot,'help','techdoc','matlab_env','examples'))
   ```

2. If you plan to modify the example, save the file to a directory for which you have write access, and then make that directory the current MATLAB directory. In this example, the file is saved to `I:\MATLABFiles\mymfiles`.

3. In the Current Directory browser toolbar, select the `M-Lint Code Check Report` from the Directory Reports listing—for details, see "Accessing and Using Directory Reports" on page 7-2.

   The M-Lint Code Check Report displays in the MATLAB Web Browser, showing those M-files that M-Lint identified as having potential problems or opportunities for improvement.



Line number and message describing a potential problem or improvement opportunity.

Click a line number to open the M-file in the Editor/Debugger at the line.

4 For each message, review the suggestion and your code, click the line number to open the M-file in the Editor/Debugger at that line, and make changes based on the message. Use the following general advice:

- If you are not sure what a message means or what to change in the code as a result, use the Help browser to look for related topics in the online documentation. For examples of messages and what to do about them, including specific changes to make for the example, `lengthofline.m`, see “Making Changes Based on M-Lint Messages” on page 7-18.

- M-Lint does not provide perfect information about every situation and in some cases, you might not want to make any changes based on the M-Lint message. In the event you do not want to change the code but you also do not want to see the M-Lint message for that line in the M-Lint Report, instruct M-Lint to ignore a line by adding `%#ok` to the end of the line in the M-file. (You can override the `%#ok` by running the `mlint` function with the `'-notok'` tag.)
- If there are certain messages or types of messages you do not want to see, you can set a preference so that M-Lint does not report them. Select **File > Preferences > M-Lint**. In **Select messages to enable**, clear the check box for messages you do not want to see. Review the settings for all messages to ensure you are seeing those pertinent to your file. Click **OK**. For more information, click the **Help** button in the **M-Lint Preferences** pane. The next time you run the report, the messages will not appear. You can use `%#ok` with a specific message ID so that only that type of message is suppressed—for more information, see the reference page for `mlint`.

5. After making changes, save the M-file. Consider saving the file to a different name if you made significant changes that might introduce errors. Then you can refer to the original file if needed to resolve problems with the updated file. Use **Tools > Compare Against** in the Editor/Debugger to help you identify the changes you made to the file. For more information, see “Comparing Files — File Comparison Tool” on page 6-54.

6. Run and debug the file(s) again to be sure you have not introduced any inadvertent errors.

7. If the M-Lint Code Check Report is already displayed, click **Rerun This Report** to update the report based on the changes you made to the file, or run the report from the Current Directory browser toolbar. Ensure the M-Lint messages are gone, based on the changes you made to the M-files.

Making Changes Based on M-Lint Messages

For information on how to correct the potential problems presented by M-Lint, use the following resources:

- Look for relevant topics in the MATLAB Programming and “Programming Tips” documentation.

- Use the Help browser **Search** and **Index** panes to find documentation about terms presented in the M-Lint messages.

Other techniques to help you identify problems in and improve your M-files are in these topics:

- "Syntax Highlighting" on page 6-28 in the Command Window and Editor/Debugger
- "Examining Errors" on page 3-9 generated when you run the M-file
- "Finding Errors, Debugging, and Correcting M-Files" on page 6-84, namely the Editor/Debugger and debugging functions
- "Profiling for Improving Performance" on page 7-27 for improving performance

Example Using M-Lint Messages to Improve Code

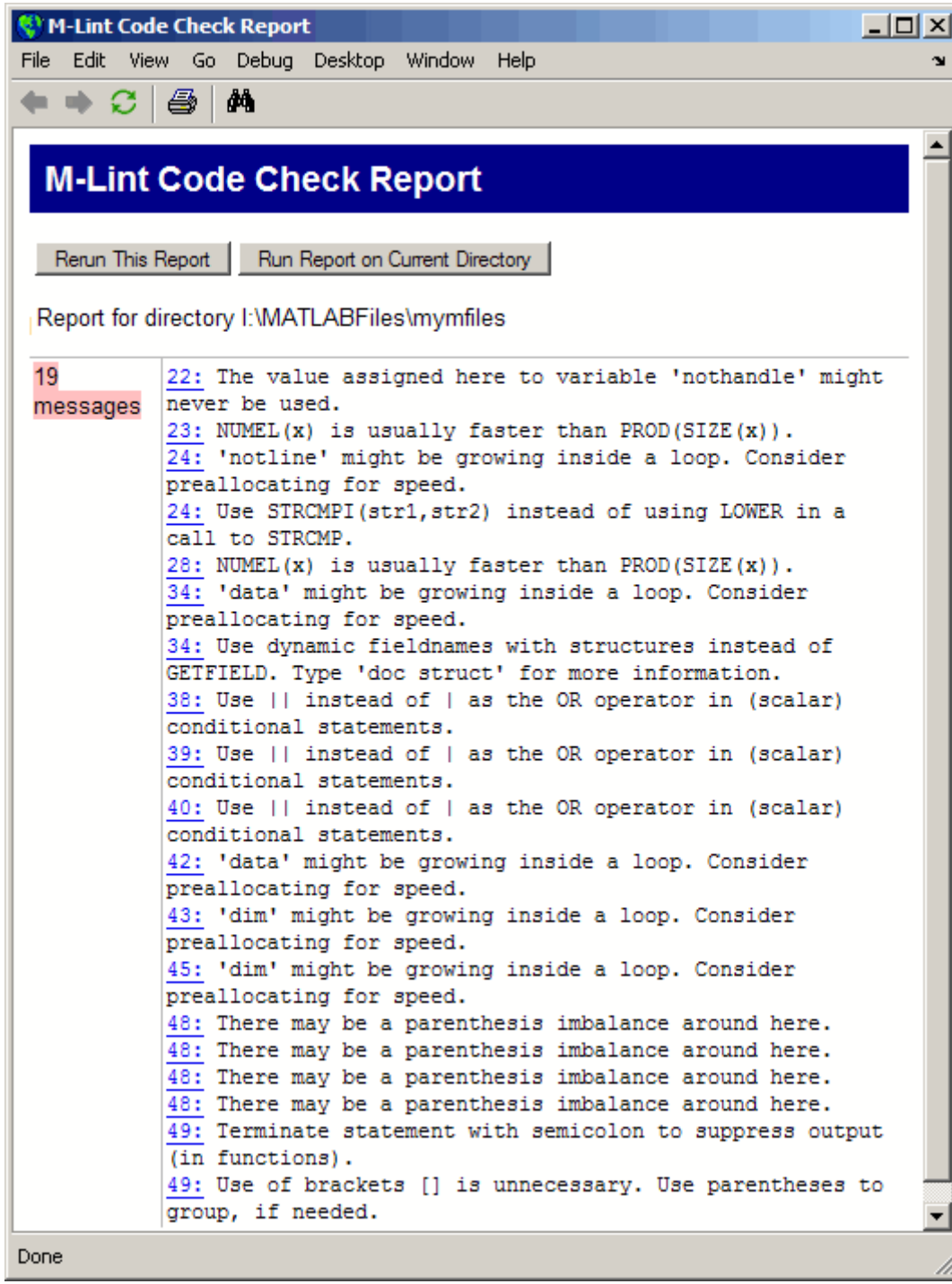
An example file, `lengthofline.m`, is included with MATLAB in *`matlabroot`*`/matlab/help/techdoc/matlab_env/examples`.

To run the M-Lint Code Check Report for `lengthofline.m`, change the current directory to its location by running

```
cd(fullfile(matlabroot,'help','techdoc','matlab_env','examples'))
```

In the Current Directory browser, select the **M-Lint Code Check Report** from the list of directory reports on the toolbar.

The M-Lint Code Check Report appears, with its list of messages suggesting improvements you can make to `lengthofline.m` and any other files in the directory.


M-Lint Code Check Report
File Edit View Go Debug Desktop Window Help
M-Lint Code Check Report
Rerun This Report
Run Report on Current Directory
Report for directory I:\MATLABFiles\mymfiles
19 messages
22: The value assigned here to variable 'nothandle' might never be used.
23: NUMEL(x) is usually faster than PROD(SIZE(x)).
24: 'notline' might be growing inside a loop. Consider preallocating for speed.
24: Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP.
28: NUMEL(x) is usually faster than PROD(SIZE(x)).
34: 'data' might be growing inside a loop. Consider preallocating for speed.
34: Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for more information.
38: Use || instead of | as the OR operator in (scalar) conditional statements.
39: Use || instead of | as the OR operator in (scalar) conditional statements.
40: Use || instead of | as the OR operator in (scalar) conditional statements.
42: 'data' might be growing inside a loop. Consider preallocating for speed.
43: 'dim' might be growing inside a loop. Consider preallocating for speed.
45: 'dim' might be growing inside a loop. Consider preallocating for speed.
48: There may be a parenthesis imbalance around here.
48: There may be a parenthesis imbalance around here.
48: There may be a parenthesis imbalance around here.
48: There may be a parenthesis imbalance around here.
49: Terminate statement with semicolon to suppress output (in functions).
49: Use of brackets [] is unnecessary. Use parentheses to group, if needed.
Done

Messages and Resulting Changes for the lengthofline Example. The following table describes each message and demonstrates a way to change the file, based on the message.

Message — Code (Original Line Numbers)	Explanation and Updated Code (New Line Numbers)

`22:  The value assigned here to variable 'nothandle' might never be used.`
`— — — — — — — — — — — — — — — — — — — —`
`22 nothandle = ~ishandle(hline);`
`23 for nh = 1:prod(size(hline))`
`24 notline(nh) = ~ishandle(hline(nh))`
`...`
	In line 22, `nothandle` is assigned a value, but `nothandle` is probably not used anywhere after that in the file. The line might be extraneous and you could delete it. But it might be that you actually intended to use the variable, which is the case for the `lengthofline` example. Update line 24 to use `nothandle`, which is faster than computing `~ishandle` for each iteration of the loop, as shown here.
	`22 nothandle = ~ishandle(hline);`
	`23 for nh = 1:numel(hline)`
	`24 notline(nh) = nothandle(nh) ...`

`23:  NUMEL(x) is usually faster than PROD(SIZE(x)).`
`— — — — — — — — — — — — — — — — — — — —`
`23 for nh = 1:prod(size(hline))`
	While `prod(size(x))` returns the number of elements in a matrix, the `numel` function was designed to do just that, and therefore is usually more efficient. Type `doc numel` to see the `numel` reference page. Change the line to
	`23 for nh = 1:numel(hline)`

Message — Code (Original Line Numbers)	Explanation and Updated Code (New Line Numbers)

`24: 'notline' might be growing inside a loop. Consider preallocating for speed.`
`— — — — — — — — — — — — — — — — — — —`
`22 nothandle = ~ishandle(hline);`
`23 for nh = 1:numel(hline)`
`24 notline(nh) = ~ishandle(hline(nh))`
`...`
	When you increase the size of an array within a loop, it is inefficient. Before the loop, preallocate the array to its maximum size to improve performance. For more information, see "Preallocating Memory" in the MATLAB Programming documentation. In the example, add a new line to preallocate `notline` before the loop.
	`23 notline = false(size(hline));`
	`24 for nh = 1:numel(hline)`
	`25 notline(nh) = nothandle(nh) ...`

`24: Use STRCMPI(str1,str2) instead of using LOWER in a call to STRCMP.`
`— — — — — — — — — — — — — — — — — — —`
`24 notline(nh)=~ishandle(hline(nh)) || ~strcmp('line',lower(get(hline(nh), 'type')));`
	While
	`strcmp ('line',lower(get(hline(nh)'type'))`
	converts the result of the `get` function to a lowercase string before doing the comparison, the `strcmpi` function ignores the case while performing the comparison, with advantages that include more efficiency. Change line 25 to
	`notline(nh) = nothandle(nh) || ~strcmpi('line',get(hline(nh),'type'));`

`28: NUMEL(x) is usually faster than PROD(SIZE(x)).`
`— — — — — — — — — — — — — — — — — — —`
`28 for nl = 1:prod(size(hline))`
	See the same message and explanation reported for line 23. Change the line 29 to
	`for nl = 1:numel(hline)`

Message — Code (Original Line Numbers)	Explanation and Updated Code (New Line Numbers)

`34: 'data' might be growing inside a loop. Consider preallocating for speed.`
`— — — — — — — — — — — — — — — — — — —`
`33 for nd = 1:length(fdata)`
`34 data{nd} = getfield(flds,fdata{nd});`
	See the same message and explanation reported for line 24. Add this line, 34, before the loop
	`data = cell(size(fdata));`

`34: Use dynamic fieldnames with structures instead of GETFIELD. Type 'doc struct' for more information.`
`— — — — — — — — — — — — — — — — — — —`
`34 data{nd} = getfield(flds,fdata{nd});`
	You can access a field in a structure as a variable expression that MATLAB evaluates at run-time. This is more efficient than using `getfield`. For more information, type `doc struct` to see the reference page for structures, or see "Using Dynamic Field Names" in the MATLAB Programming documentation. Change line 37 to
	`data{nd} = flds.(fdata{nd});`

`38: Use || instead of | as the OR operator in (scalar) conditional statements.`
`39: Use || instead of | as the OR operator in (scalar) conditional statements.`
`40: Use || instead of | as the OR operator in (scalar) conditional statements.`
`— — — — — — — — — — — — — — — — — — —`
`38 if isempty(data{3}) | ...`
`39 (length(unique(data{1}(:)))==1 | ...`
`40 length(unique(data{2}(:)))==1 | ...`
`41 length(unique(data{3}(:)))==1)`
	While `|` (the element-wise logical `OR` operator) performs the comparison correctly, use the `||` (short circuit `OR` operator) for efficiency. For details, see "Logical Operators" in the MATLAB Programming documentation. Change lines 40, 41, and 42 to
	`if isempty(data{3}) || ...`
	`(length(unique(data{1}(:)))==1 || ...`
	`length(unique(data{2}(:)))==1 || ...`

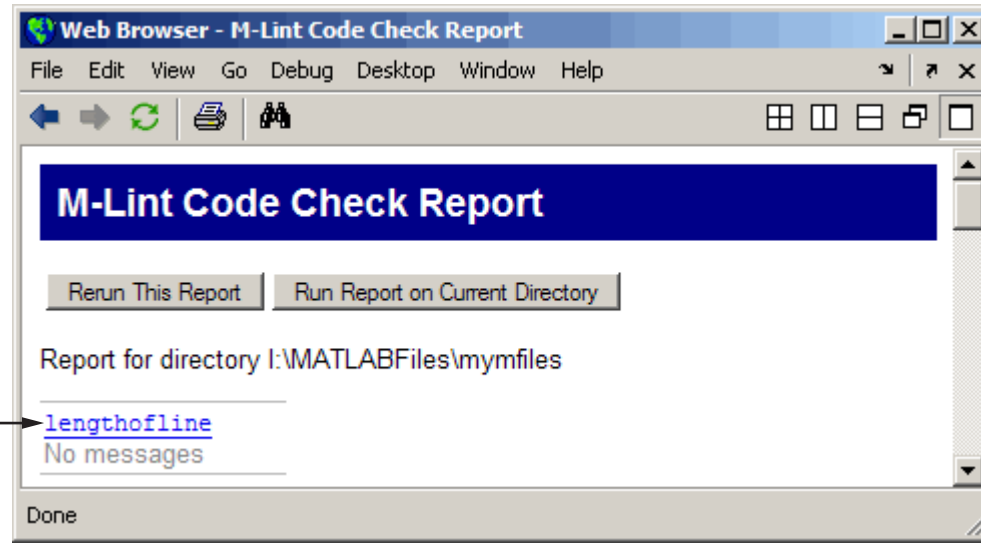
Message — Code (Original Line Numbers)	Explanation and Updated Code (New Line Numbers)

`42: 'data' might be growing inside a loop. Consider preallocating for speed.`
— — — — — — — — — — — — — — — — — —
`42 data{3} = zeros(size(data{1}));`
	This message no longer appears due to the change made to line 34 `data{nd} = getfield(flds,fdata{nd});`. Sometimes fixing code in one line automatically clears a message for another line. If the reason for a message or the action to take for a message is not obvious at first, it could be because another line is causing the message. Address the issues that are easy to fix first and rerun the report. Do not make any changes to line 44.

`43: 'dim' might be growing inside a loop. Consider preallocating for speed.` `43 dim(nl) = 2;`	See the same message and explanation reported for line 24. Add this line before the first line of the loop `dim = len;`
`48: There may be a parenthesis imbalance around here.` `48: There may be a parenthesis imbalance around here.` `48: There may be a parenthesis imbalance around here.` `48: There may be a parenthesis imbalance around here.`	There is an error in this line, which you would see by running `lengthofline`. M-Lint suggests that it might be due to a parenthesis imbalance. You can check that by moving the arrow key over each of the delimiters, to see if MATLAB indicates a mismatch. This requires that **File > Preferences > Keyboard > Delimiter Matching** has the **Match on arrow key** option selected. There are no mismatched delimiters. The actual problem is the semicolon in parentheses, `data{3}(:)` is incorrect and should be a colon. In line 51, change `data{3}(;)` to `data{3}(:)`. That single change addressed the issues in all the messages for that line.

Message — Code (Original Line Numbers)	Explanation and Updated Code (New Line Numbers)
`49:  Terminate statement with semicolon to suppress output (in functions).`	Adding a semicolon to the end of a statement suppresses output and is a common practice. M-Lint alerts you to lines that produce output but lack the terminating semicolon. If you want to view output from this line, do not add the semicolon. You can instruct M-Lint to ignore all messages on this line so that the messages on it will not appear by adding `%#ok` to the end of the line. However, because there is currently another message on the line, do not add `%#ok` until you have addressed the other message. Alternatively, you can add `%#ok` with the message ID for the specific message you want to suppress. To determine the message ID, run `mlint('lengthofline.m', '-id')`, which indicates the ID is `NOPRT`—for more information, see the `mlint` function reference page. For this example, assume you want to display the output and suppress the M-Lint message. To do so, add `%#ok<NOPRT>` to the end of the line. Note that there is a similar message for M-file scripts. This is so you can suppress the message for M-files that are cell-mode scripts, because they are often intended as demos and the display of output is intentional.

`49:  Use of brackets [] is unnecessary. Use parentheses to group, if needed.`
 	
`— — — — — — — — — — — — — — — — — — —`
 	
`49 len(nl) =`
`sum([sqrt(dot(temp',temp'))])`
	For more information about the use of brackets and parentheses, see the Special Characters reference page. In this example, remove the brackets because they are not needed. They add processing time because MATLAB concatenates unnecessarily. Change line 52 to
	
	`len(nl) = sum(sqrt(dot(temp',temp'))) %#ok`



Updated M-Lint Code Check Report after making changes to the lenghtofline file based on M-Lint messages. Now, no messages are reported.

The M-file that includes all of the changes suggested by M-Lint is `lengthofline2.m`. To view it, run

```
edit(fullfile(matlabroot,'help','techdoc','matlab_env',...
'examples','lengthofline2.m')).
```

Other Ways to Access M-Lint

You can get M-Lint messages using any of the following methods. Each provides the same M-Lint messages, but in a different format:

- Access the M-Lint Code Check report for an M-file from the Editor/Debugger Tools menu or from the Profiler detail report.
- Run the `mlint` function, which analyzes the specified file and displays messages in the Command Window, or `mlintrpt`, which runs `mlint` and displays the messages in the Web Browser.
- Use automatic M-Lint analysis and code correction while you work on a file in the Editor/Debugger — see “M-Lint Code Analyzer” on page 6-87.

Profiling for Improving Performance

In this section...

What Is Profiling?

Profiling is a way to measure where a program spends time. To assist you in profiling, MATLAB provides a graphical user interface, called the Profiler, which is based on the results returned by the `profile` function. Once you identify which functions are consuming the most time, you can determine why you are calling them and look for ways to minimize their use and thus improve performance. It is often helpful to decide whether the number of times a particular function is called is reasonable. Because programs often have several layers, your code may not explicitly call the most time-consuming functions. Rather, functions within your code might be calling other time-consuming functions that can be several layers down in the code. In this case it is important to determine which of your functions are responsible for such calls.

Profiling helps to uncover performance problems that you can solve by

- Avoiding unnecessary computation, which can arise from oversight
- Changing your algorithm to avoid costly functions
- Avoiding recomputation by storing results for future use

When you reach the point where most of the time is spent on calls to a small number of built-in functions, you have probably optimized the code as much as you can expect.

Profiling Process and Guidelines

Here is a general process you can follow to use the Profiler to improve performance in your M-files. This section also describes how you can use profiling as a debugging tool and as a way to understand complex M-files.

Note Premature optimization can increase code complexity unnecessarily without providing a real gain in performance. Your first implementation should be as simple as possible. Then, if speed is an issue, use profiling to identify bottlenecks.

---

1. In the summary report produced by the Profiler, look for functions that used a significant amount of time or were called most frequently. See "Profile Summary Report" on page 7-33 for more information.

2. View the detail report produced by the Profiler for those functions and look for the lines that use the most time or are called most often. See "Profile Detail Report" on page 7-35 for more information.

   You might want to keep a copy of your first detail report to use as a reference to compare with after you make changes, and then profile again.

3. Determine whether there are changes you can make to the lines most called or the most time-consuming lines to improve performance.

   For example, if you have a `load` statement within a loop, `load` is called every time the loop is called. You might be able to save time by moving the `load` statement so it is before the loop and therefore is only called once.

4. Click the links to the files and make the changes you identified for potential performance improvement. Save the files and run `clear all`. Run the Profiler again and compare the results to the original report. Note that there are inherent time fluctuations that are not dependent on your code. If you profile the exact same code twice, you can get slightly different results each time.

5. Repeat this process to continue improving the performance.

Using Profiling as a Debugging Tool

The Profiler is a useful tool for isolating problems in your M-files.

For example, if a particular section of the file did not run, you can look at the detail reports to see what lines did run, which might point you to the problem.

You can also view the lines that did not run to help you develop test cases that exercise that code.

If you get an error in the M-file when profiling, the Profiler provides partial results in the reports. You can see what ran and what did not to help you isolate the problem. Similarly, you can do this if you stop the execution using **Ctrl+C**, which might be useful when a file is taking much more time to run than expected.

Using Profiling for Understanding an M-File

For lengthy M-files that you did not create or that you have not used for awhile and are unfamiliar with, you can use the Profiler to see how the M-file actually worked. Use the Profiler detail reports to see the lines actually called.

If there is an existing GUI tool (or M-file) similar to one that you want to create, start profiling, use the tool, then stop profiling. Look through the Profiler detail reports to see what functions and lines ran. This helps you determine the lines of code in the file that are most like the code you want to create.

Using the Profiler

Use the Profiler to help you determine where you can modify your code to make performance improvements. The Profiler is a tool that shows you where an M-file is spending its time. This section covers

For information about the reports generated by the Profiler, see “Profile Summary Report” on page 7-33 and “Profile Detail Report” on page 7-35.

Opening the Profiler

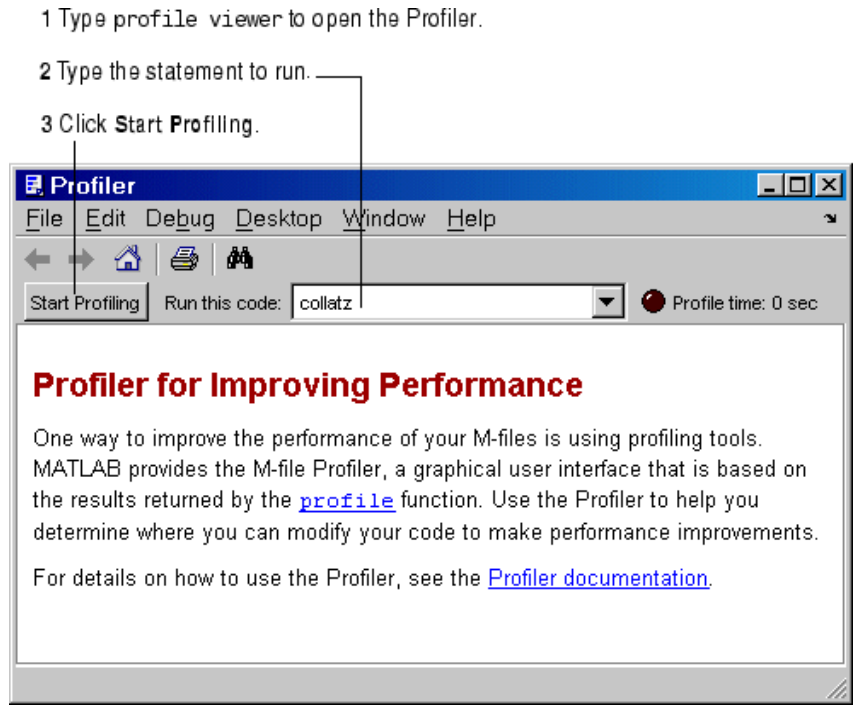
You can use any of the following methods to open the Profiler:

- Select **Desktop > Profiler** from the MATLAB desktop.
- Click the Profiler button in the MATLAB desktop toolbar.
- With a file open in the MATLAB Editor/Debugger, select **Tools > Open Profiler**.
- Select one or more statements in the Command History window, right-click to view the context menu, and choose **Profile Code**.
- Enter the following function in the Command Window:

```
profile viewer
```

Running the Profiler

The following illustration summarizes the steps for profiling.



To profile an M-file or a line of code, follow these steps:

1 In the **Run this code** field in the Profiler, type the statement you want to run.

   You can run this example

   ```
   [t,y] = ode23('lotka',[0 2],[20;20])
   ```

   as the code is provided with MATLAB demos. It runs the Lotka-Volterra predator-prey population model. For more information about this model, type `lotkademo`, which runs the demonstration.

   To run a statement you previously profiled in the current MATLAB session, select the statement from the list box — MATLAB automatically starts profiling the code, so skip to step 3.

2 Click **Start Profiling** (or press **Enter** after typing the statement).

While the Profiler is running, the **Profile time** indicator (at the top right of the Profiler window) is green and the number of seconds it reports increases.

When the profiling is finished, the **Profile time** indicator becomes black and shows the length of time the Profiler ran. The statements you profiled are shown as having been executed in the Command Window.

This is not the actual time that your statements took to run; it is the wall clock (or `tic/toc`) time elapsed from when you clicked **Start Profiling** until profiling stops. If the time reported is much different from what you expected (for example, hundreds of seconds for a simple statement), you might have had profiling on longer than you realized. This also does not match the time reported in Profiler statistics, which is based on `cpu` time by default, not wall clock time.

3 When profiling is complete, the Profile Summary report appears in the Profiler window. For more information about this report, see “Profile Summary Report” on page 7-33.

Profiling a Graphical User Interface

You can run the Profiler for a graphical user interface, such as the Filter Design and Analysis tool included with Signal Processing Toolbox. You can also run the Profiler for an interface you created, such as one built using GUIDE.

To profile a graphical user interface, follow these steps:

1 In the Profiler, click **Start Profiling**. Make sure that no code appears in the **Run this code** field.

2 Start the graphical user interface. (If you do not want to include its startup process in the profile, do not click **Start Profiling**, step 1, until after you have started the graphical interface.)

3 Use the graphical interface. When you are finished, click **Stop Profiling** in the Profiler.

The Profile Summary report appears in the Profiler.

Profiling Statements from the Command Window

To profile more than one statement, follow these steps:

1 In the Profiler, clear the **Run this code** field and click **Start Profiling**.

2 In the Command Window, enter and run the statements you want to profile.

3 After running all the statements, click **Stop Profiling** in the Profiler.

The Profile Summary report appears in the Profiler.

Changing Fonts for the Profiler

To change the fonts used in the Profiler, follow these steps:

1 Select **File > Preferences > Fonts** to open the **Font** Preferences dialog box.

2 In the Font Preferences dialog box, select the code or text font that you want to use in the Profiler. The Profiler is an HTML Proportional Text tool. For more information, click the **Help** button in the dialog box.

3 Click **Apply** or **OK**. The Profiler font reflects the changes.

Profile Summary Report

The Profile Summary report presents statistics about the overall execution of the function and provides summary statistics for each function called. The report formats these values in four columns.

- **Function Name** — A list of all the functions and subfunctions called by the profiled function. When first displayed, the functions are listed in

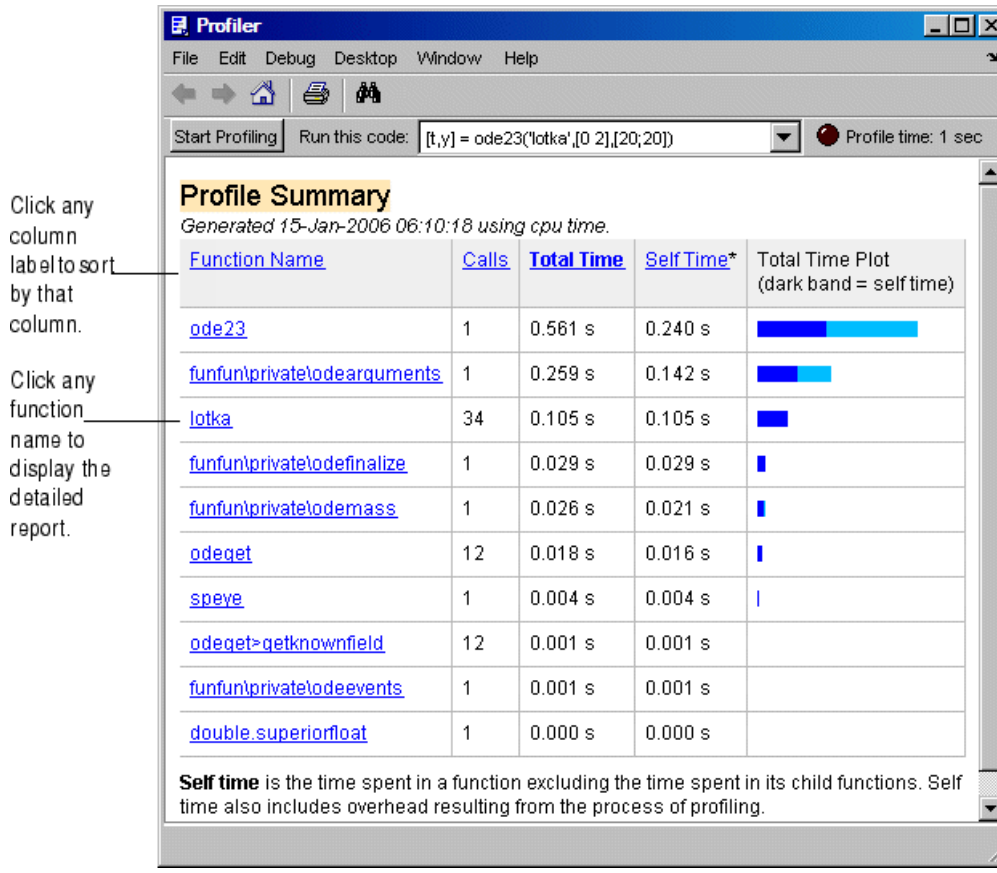
order by the amount of time they took to process. To sort the functions alphabetically, click the **Function Name** link at the top of the column.

- **Calls** — The number of times the function was called while profiling was on. To sort the report by the number of times functions were called, click the **Calls** link at the top of the column.
- **Total Time** — The total time spent in a function, including all child functions called, in seconds. The time for a function includes time spent on child functions. To sort the functions by the amount of time they consumed, click the **Total Time** link at the top of the column. By default, the summary report displays profiling information sorted by **Total Time**. Note that the Profiler itself uses some time, which is included in the results. Also note that total time can be zero for files whose running time was inconsequential.
- **Self Time** — The total time spent in a function, *not* including time for any child functions called, in seconds. To sort the functions by this time value, click the **Self Time** link at the top of the column.
- **Total Time Plot** — Graphic display showing self time compared to total time.

Following is the summary report for the Lotka-Volterra model described in “Example: Using the profile Function” on page 7-43.

To print a summary report, click the Print button  .

To get more detailed information about a particular function, click its name in the **Function Name** column. See “Profile Detail Report” on page 7-35 for more information.



Profile Detail Report

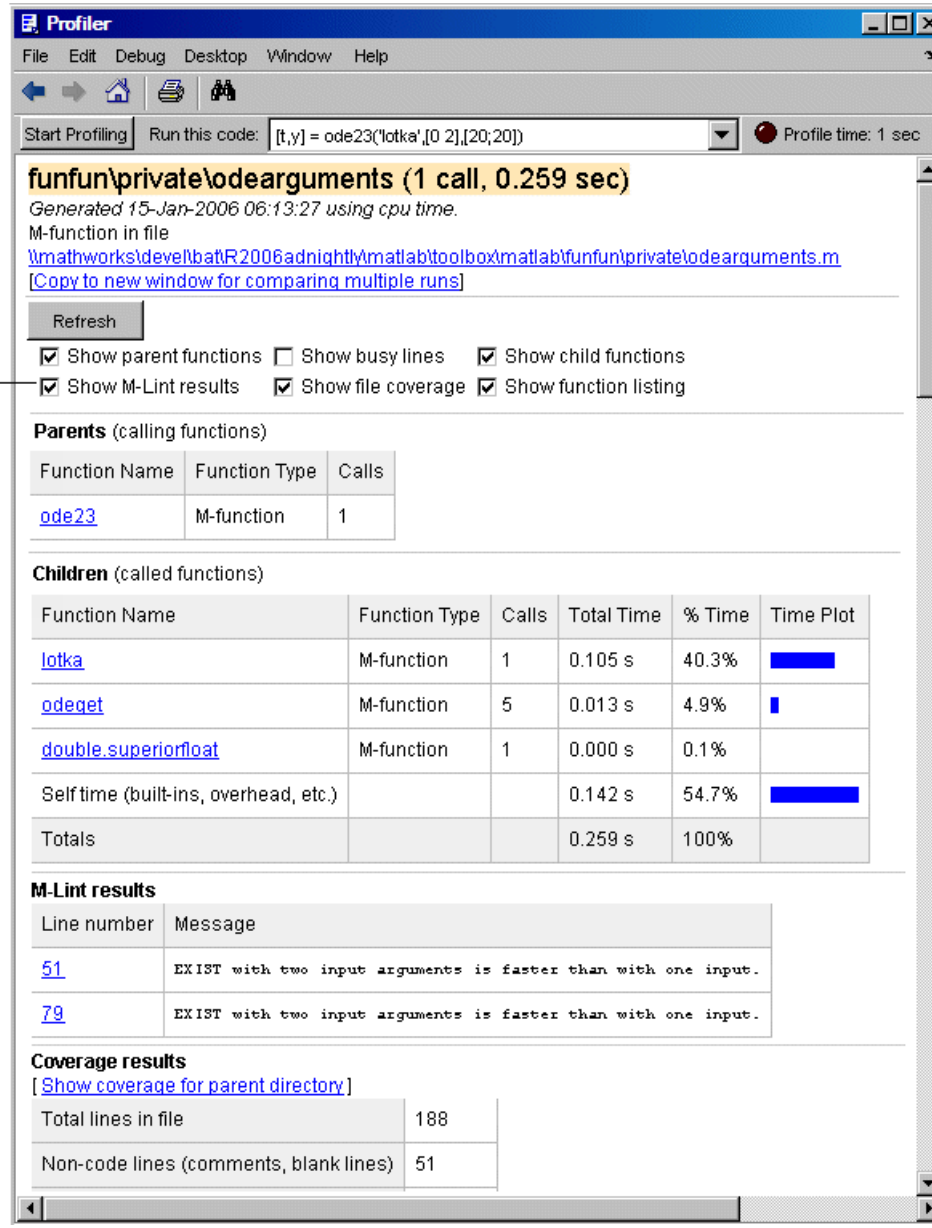
The Profile Detail report shows profiling results for a selected function that was called during profiling. A Profile Detail report is made up of seven sections, summarized below. By default, the Profile Detail report includes all seven sections, although, depending on the function, not every section contains data. To return to the Profile Summary report from the Profile Detail report, click the Home button in the toolbar. The following sections provide more detail:

Controlling the Contents of the Detail Report Display (p. 7-36)	Customize display to include only sections you are interested in.
Profile Detail Report Header (p. 7-38)	Provides general information about the function.
Parent Functions (p. 7-38)	Provides information about the parent function.
Busy Lines (p. 7-38)	Lists the lines in the function that used the greatest amount of processing time.
Child Functions (p. 7-39)	
M-Lint Results (p. 7-40)	Lists the lines in the functions that M-Lint highlighted.
File Coverage (p. 7-40)	Provides statistics about the lines of code in the function that executed while profiling was on.
Function Listing (p. 7-41)	Includes the source code for the function, if it is an M-file.

Controlling the Contents of the Detail Report Display

You can determine which sections are included in the display by selecting them and then clicking the **Refresh** button. The following sections provide more detail about each section of this report.

Select report options to display, and then click **Refresh**.

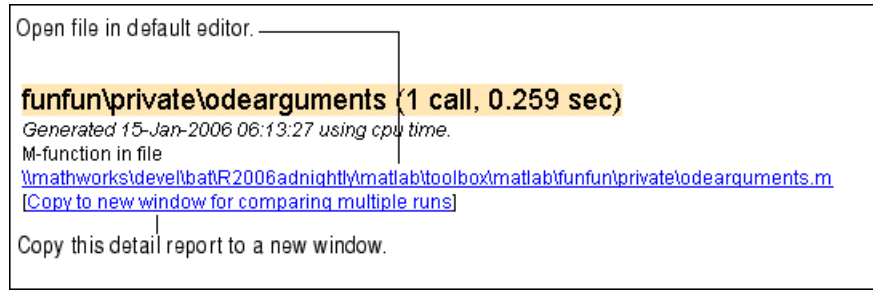


Profile Detail Report Header

The detail report header includes the name of the function that was profiled, the number of times it was called in the parent function, and the amount of time it used.

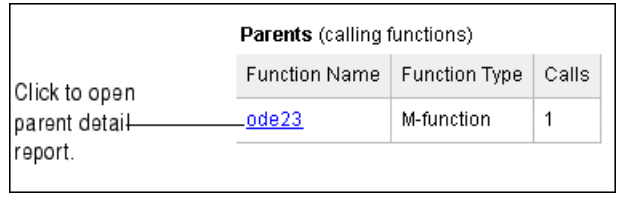
The header includes a link that opens the function in your default text editor.

The header also includes a link that copies the report to a separate window. Creating a copy of the report can be helpful when you make changes to the file, run the Profiler for the updated file, and compare the Profile Detail reports for the two runs. Do not make changes to M-files provided with MathWorks products, that is, files in *`matlabroot`*`/toolbox` directories.



Parent Functions

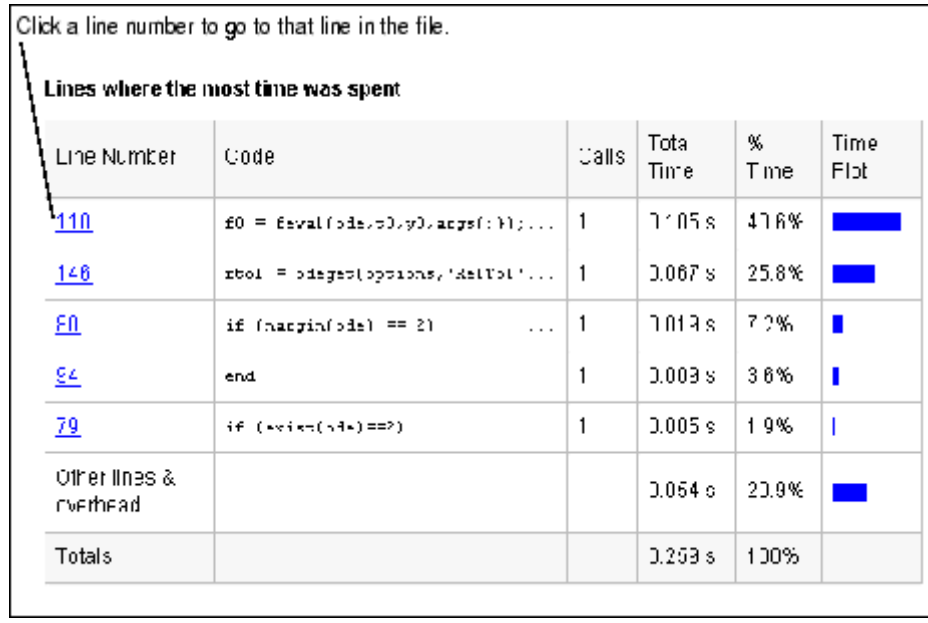
To include the **Parents** section in the detail report, select the **Show parent functions** check box. This section of the report provides information about the parent functions, with links to their detail reports.



Busy Lines

To include information about the lines of code that used the most amount of processing time in the detail report, select the **Show busy lines** check box.

Note that this was not selected in the example. Click a line number to view that line of code in the source listing.

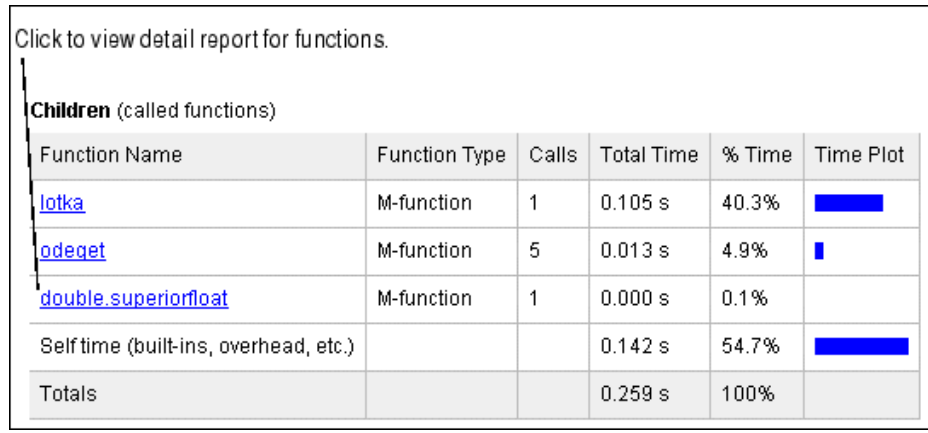

Click a line number to go to that line in the file.

Lines where the most time was spent

Line Number	Code	Calls	Tota Time	% Time	Time Plot
110	f0 = feval(p1s,p3),y),args{:});...	1	1.105 s	47.6%	
146	xbol = p1sgns(options,'dellol'...	1	0.067 s	25.8%	
50	if (nargin(p1s) == 2) ...	1	0.013 s	7.2%	
82	end	1	0.009 s	3.6%	
79	if (exist(p1s)==2)	1	0.005 s	1.9%	
Other lines & overhead			0.054 s	20.9%	
Totals			0.259 s	100%	

Child Functions

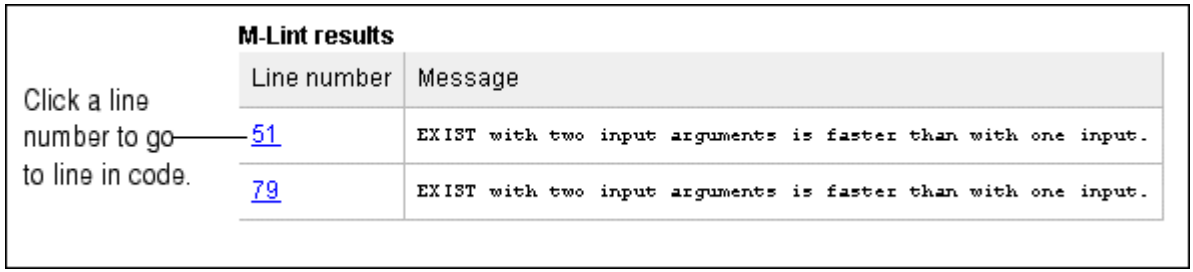
To include the **Children** section of the detail report, select the **Show child functions** check box. This section of the report lists all the functions called by the profiled function. If the called function is an M-file, you can view the source code for the function by clicking its name.



Function Name	Function Type	Calls	Total Time	% Time	Time Plot
lotka	M-function	1	0.105 s	40.3%	
odeget	M-function	5	0.013 s	4.9%	
double.superiorfloat	M-function	1	0.000 s	0.1%	
Self time (built-ins, overhead, etc.)			0.142 s	54.7%	
Totals			0.259 s	100%	

M-Lint Results

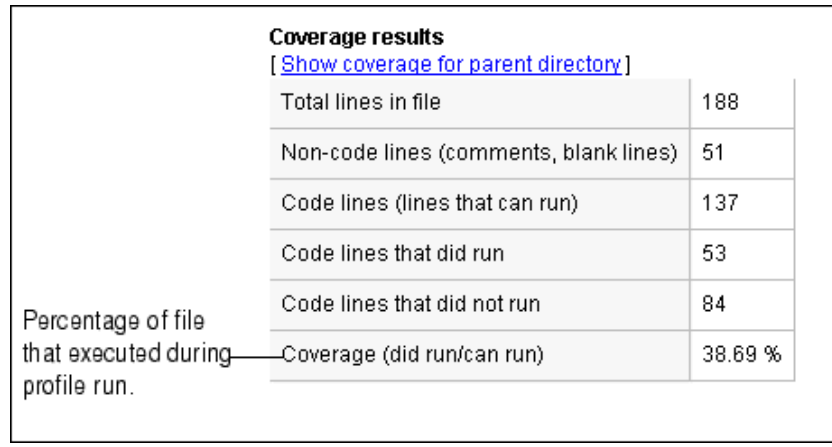
To include the **M-Lint results** section in the detail report display, select the **Show M-Lint results** check box. This section of the report provides information about problems and potential improvements, generated by M-Lint about the function. For more information about M-Lint, see "M-Lint Code Check Report" on page 7-16.



Line number	Message
51	EXIST with two input arguments is faster than with one input.
79	EXIST with two input arguments is faster than with one input.

File Coverage

To include the **Coverage results** section in the detail report display, select the **Show file coverage** check box. This section of the report provides statistical information about the number of lines in the code that executed during the profile run.

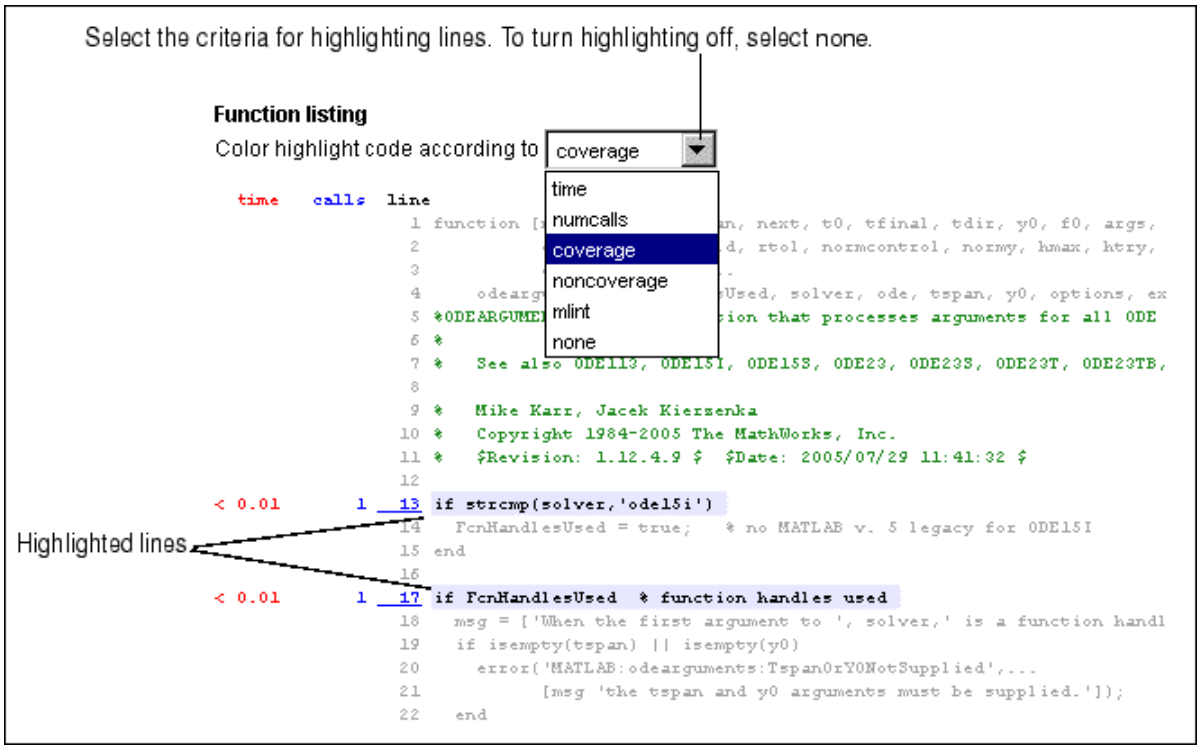


Function Listing

To include the **Function listing** section in the detail report display, select the **Show function listing** check box. If the file is an M-file, the Profile Detail report includes a column listing the execution time for each line, a column listing the number of times the line was called, and the source code for the function.

In the function listing, comment lines appear in green, lines of code that executed appear in black, and lines of code that did not execute appear in gray. If you click a function name in the listing, you can view its detail report.

By default, the Profile Detail report uses the color red to highlight the lines of code with the longest execution time. The darker the shade of red, the longer the line of code took to execute. Using the menu in this section of the detail report you can change this default and choose to highlight lines of code based on other criteria such as the lines called the most, lines called out by M-Lint, or lines of code that were (or were not) executed. Using this menu, you can also turn off highlighting completely.



The profile Function

The Profiler is based on the results returned by the `profile` function. This section describes

profile Function Syntax Summary

Here is a summary of some of the main forms of `profile`. For details about these and other options, type `doc profile`. Some people use `profile` simply to see the child functions; see also `depfun` for that purpose.

Syntax	Description
`profile on`	Starts `profile`, clearing previously recorded statistics.
`profile on -detail level`	Specifies the level of function to be profiled, where *`level`* can be either: `'mmex'` — M-functions, M-subfunctions, and MEX-functions `'builtin'` — M-functions, M-subfunctions, MEX-functions, and built-ins
`profile on -history`	Specifies that the exact sequence of function calls is to be recorded.
`profile off`	Suspends `profile`.
`profile resume`	Restarts `profile` without clearing previously recorded statistics.
`profile clear`	Clears the statistics recorded by `profile`.
`profile viewer`	Opens the Profiler, a graphical user interface and displays the information gathered as an HTML-formatted report. Note: If you run the obsoleted syntax `profile report`, the `profile` function calls this syntax.
`s = profile('status')`	Displays a structure containing the current `profile` status.
`stats = profile('info')`	Suspends `profile` and displays a structure containing `profile` results.

Example: Using the profile Function

This example demonstrates how to run `profile`:

1 To start `profile`, type in the Command Window

```
profile on
```

2 Execute an M-file. This example runs the Lotka-Volterra predator-prey population model. For more information about this model, type `lotkademo`, which runs a demonstration.

```
[t,y] = ode23('lotka',[0 2],[20;20]);
```

3 Generate the profile report and display it in the Profiler window. This suspends `profile`.

```
profile viewer
```

4 Restart `profile`, without clearing the existing statistics.

```
profile resume
```

The `profile` function is now ready to continue gathering statistics for any more M-files you run. It will add these new statistics to those generated in the previous steps.

5 Stop `profile` when you finish gathering statistics.

```
profile off
```

6 To view the profile data, call `profile` specifying the `'info'` argument. The profile function returns data in a structure.

```
p = profile('info')

p =
      FunctionTable: [10x1 struct]
    FunctionHistory: [2x0 double]
     ClockPrecision: 3.3333e-010
         ClockSpeed: 3.0000e+009
               Name: 'MATLAB'
```

The `FunctionTable` indicates that statistics were gathered for 10 functions.

7 To save the profile report, use the `profsave` function. This function stores the profile information in separate HTML files, for each function listed in `FunctionTable` of `p`.

```
profsave(p)
```

By default, `profsave` puts these HTML files in a subdirectory of the current directory named `profile_results`, and displays the summary report in your system browser. You can specify another directory name as an optional second argument to `profsave`.

Accessing Profiler Results

The `profile` function returns results in a structure. This example illustrates how you can access these results:

1 To start `profile`, specifying the `detail` and `history` options, type in the Command Window.

```
profile on -detail builtin -history
```

The detail option specifies that built-ins should be included in the profile data. The history option specifies that the report include information about the sequence of functions as they are entered and exited during profiling.

2 Execute an M-file. This example runs the Lotka-Volterra predator-prey population model. For more information about this model, type `lotkademo`, which runs a demonstration.

```
[t,y] = ode23('lotka',[0 2],[20;20]);
```

3 Stop profiling.

```
profile off
```

4 Get the structure containing profile results.

```
stats = profile('info')
stats =
      FunctionTable: [43x1 struct]
    FunctionHistory: [2x754 double]
     ClockPrecision: 3.3333e-010
```

```
          ClockSpeed: 3.0000e+009
                Name: 'MATLAB'
```

5 The `FunctionTable` field is an array of structures, where each structure represents an M-function, M-subfunction, MEX-function, or, because the `builtin` option is specified, a MATLAB built-in function.

```
stats.FunctionTable

ans =

41x1 struct array with fields:
    CompleteName
    FunctionName
    FileName
    Type
    NumCalls
    TotalTime
    TotalRecursiveTime
    Children
    Parents
    ExecutedLines
    IsRecursive
    PartialData
```

6 View the second structure in `FunctionTable`.

```
stats.FunctionTable(2)

ans =
           CompleteName: [1x79 char]
           FunctionName: 'ode23'
               FileName: [1x73 char]
                   Type: 'M-function'
               NumCalls: 1
              TotalTime: 0.3902
    TotalRecursiveTime: 0
               Children: [20x1 struct]
                Parents: [0x1 struct]
          ExecutedLines: [139x3 double]
```

```
    IsRecursive: 0
    PartialData: 0
```

7 To view the history data generated by `profile`, view the `FunctionHistory`, for example, `stats.FunctionHistory`. The history data is a 2-by-n array. The first row contains Boolean values, where `0` (zero) means entrance into a function and `1` means exit from a function. The second row identifies the function being entered or exited by its index in the `FunctionTable` field. To see how to create a formatted display of history data, see the example on the `profile` reference page.

Saving Profile Reports

To save the profile report, use the `profsave` function.

This function stores the profile information in separate HTML files, for each function listed in the `FunctionTable` field of the structure, `stats`.

```
profsave(stats)
```

By default, `profsave` puts these HTML files in a subdirectory of the current directory named `profile_results`. You can specify another directory name as an optional second argument to `profsave`.

```
profsave(stats,'mydir')
```

8

Publishing Results

MATLAB provides two different approaches for publishing: using cells and with the Notebook features for Microsoft Word.

Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells

In this section...

About Publishing M-Files

When you have completed writing and debugging an M-file, use the M-file cell features in the Editor/Debugger to quickly publish the M-file and its results in any of several presentation formats: HTML, XML, LaTeX, or, when the applications are installed, Microsoft Word or PowerPoint. This allows you to share your work with others, presenting not only the code, but also commentary on the code and results from running the file.

Publishing features evaluate M-files cell-by-cell and display the contents of a cell in a presentation quality document. For M-file scripts, publishing also displays the results. For example, published documents include bold headings for each section of the file. For M-file scripts, published documents include output to the Command Window and figures. The cells in the Editor/Debugger that you use for publishing are the same ones you might already have used for rapid code iteration—see "Using Cells for Rapid Code Iteration and Publishing Results" on page 6-133.

If you have an active Internet connection, you can watch the Publishing M Code from the Editor/Debugger video demo for an overview of the major functionality.

This is the overall process to publish an M-file using cell features in the Editor/Debugger:

1. Enable cell mode and define cells as described in steps 1 through 3 in "Using Cells for Rapid Code Iteration and Publishing Results" on page 6-133. When you publish the file without adding any text markup, comments at

the start of a cell appear as plain text. Comments appearing after code in a cell appear as unformatted M-file comments in the published document.

2. Use **Cell > Insert Text Markup** to insert markup symbols in the M-file comments to stylize the text for the output, for example, to display specified text as bold or monospaced. For details, see "Marking Up Text in Cells for Publishing" on page 8-11.

3. When publishing an M-file function, select **File > Preferences > Editor/Debugger > Publishing**, and clear the **Evaluate code** option.

4. Select **File > Publish To**, and select the format in which you want to publish the M-file: HTML, XML, LaTeX, Word, or PowerPoint. For details, see "Publishing M-Files Using Cells" on page 8-24.

5. Change the **Editor/Debugger Publishing** and **Publishing Images** preferences to adjust the output. For example, you can choose to include or exclude the code from the output. For details, see "Modifying Published Output Using Preferences" on page 8-26.

MATLAB publishes the M-file by writing the cell titles, comment text, and code to a file using the specified format. For M-file scripts, MATLAB also evaluates the cells and writes the results of the evaluation to the output file. Any figures created or modified during the evaluation are saved as graphics files, and are shown with the results.

Publishing Scripts and Functions—Differences

When you publish M-file scripts, MATLAB runs the code and the published output includes the results.

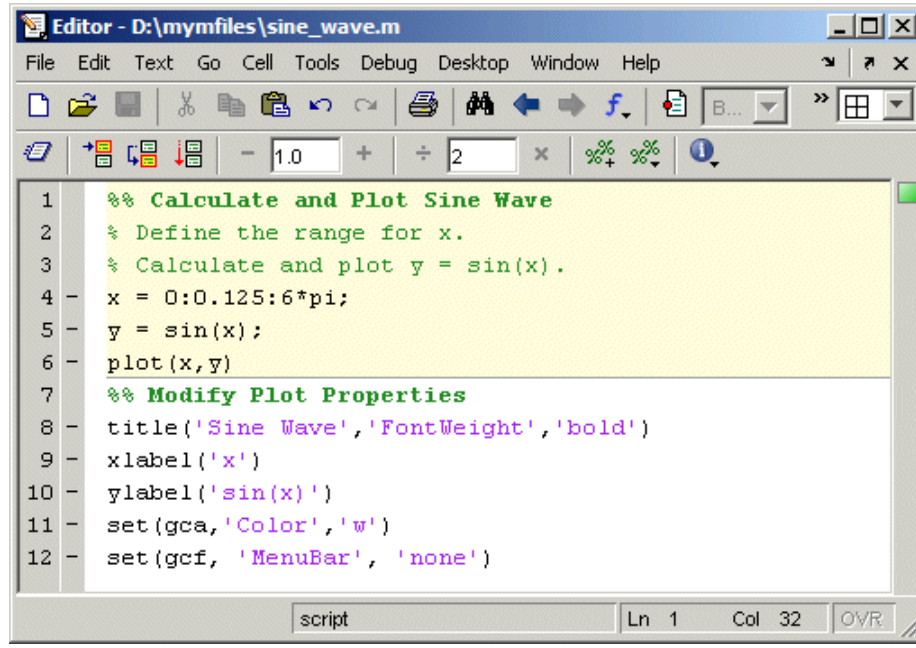
When you publish M-file functions, MATLAB does not run the code, so the published output does not include results. Publishing an M-file function allows you to effectively save the M-file code along with formatted M-file comments. The instructions for publishing M-files apply to M-file functions, with the exception of steps that involve evaluating code and displaying results from running the file.

If you want to publish an M-file function that includes results, change the M-file function to a script. Remove the function declaration statement and

within the M-file script, supply any input values that you had passed when you ran the function.

Example of Publishing Without Text Markup

This is based on the M-file script used in “Example — Evaluate Cells” on page 6-141, as shown here. Instructions for preparing and publishing the file follow.



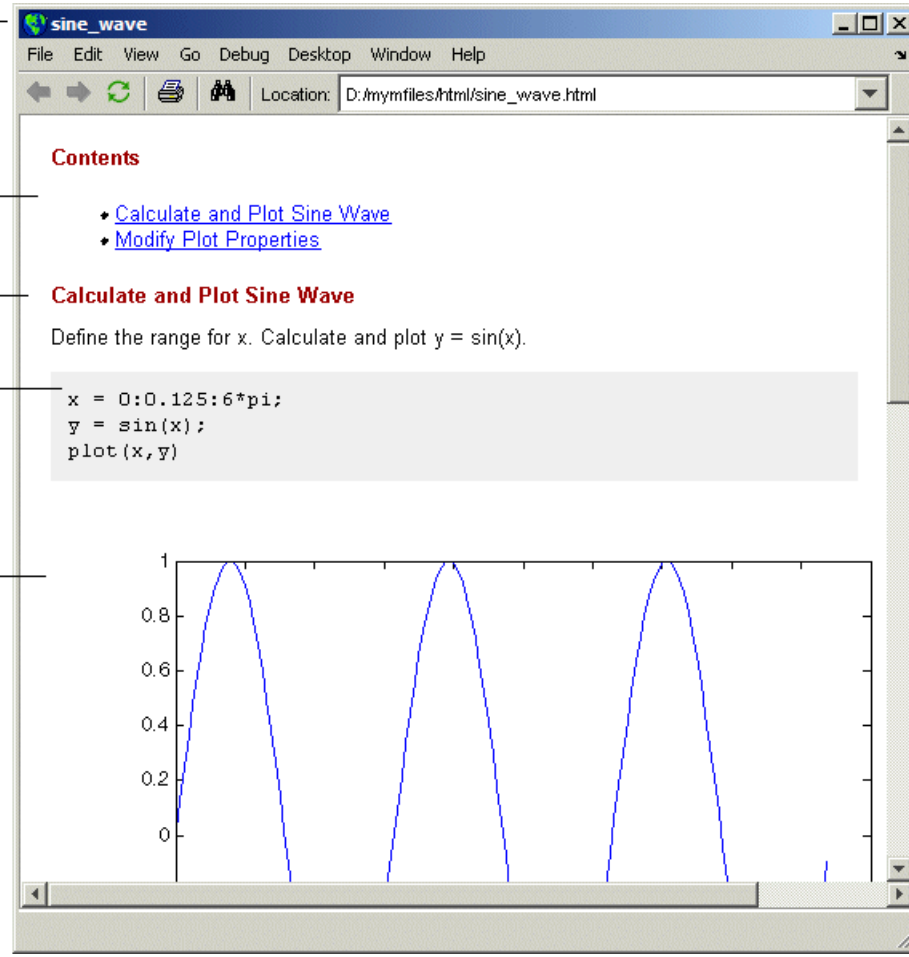
Select **File > Publish to HTML** to produce the following result.

M-file published to HTML format.

Publishing automatically generates a cell contents listing with links to all cell (section) titles.

Automatic formatting makes cell titles bold and shows code in a monospace font. Executable code appears with a gray background.

Results from running the M-file are included in the published document.



Example of Publishing with Text Markup

This simple example adds text markup to the `sine_wave.m` file used in “Example of Publishing Without Text Markup” on page 8-4 to produce the following published HTML document.

Published file after applying text markup.

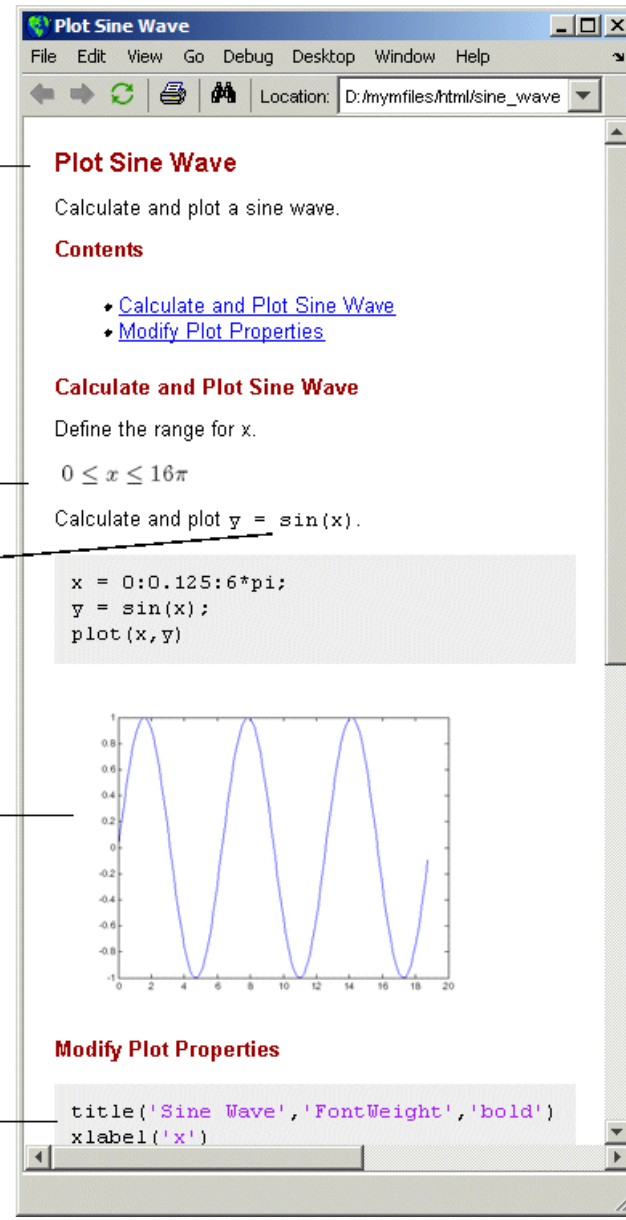
Add a title for the document.

Display comment lines using TeX format.

Make selected comment text appear in monospace.

Reduce the size of the figure output in the document using **Preferences for Publishing Images**.

MATLAB code appears with gray background to distinguish it from results.



1. Add an overall title and introduction for the published document

   a. Select **Cell > Insert Text Markup > > Document Title and Introduction**. MATLAB adds the following at the top of the file.

   ```
   %% DOCUMENT TITLE
   % INTRODUCTORY TEXT
   ```

   The two percent signs (`%%`) indicate the start of a new cell, where a cell is a section of an M-file. A single percent sign indicates a comment line.

   b. Replace **`DOCUMENT TITLE`** with **`Plot Sine Wave`**.

   c. On line 2, after the single percent sign, replace `INTRODUCTORY TEXT` with a comment about the overall file. For example, `Calculate and plot a sine wave`.

   d. On line 3, insert a blank line for better readability.

   You can add any overall comments about the file in the lines following the title. You cannot add executable code between document title and before the next cell (a line starting with `%%`) if you want the document title to appear as the overall document title.

2. Display equations in comments with symbols and Greek characters using the TeX format. For a list of symbols you can display and the character sequence to create them, see the `String` property on the MATLAB reference page for graphics `Text Properties`. In this example, to create a comment containing the following equation in the published document: $0 \le x \le 6\pi$, use text markup as follows:

   a. Position the cursor at the end of line 5, where the text`Define the range for x` appears.

   b. Select **Cell > Insert Text Markup > TeX Equation**.

   MATLAB inserts the following lines:

   ```
   %
   % $$e^{\pi i} + 1 = 0$$
   %
   ```

   The sample equation, which is the text between the set of two dollar signs (`$$`), is highlighted.

c Replace the sample equation with the following TeX equation:

```
0 \leq x \leq 6\pi
```

The three lines that display the TeX equation in the published document now appear as follows in the M-file.

```
%
% $$0 \leq x \leq 6\pi$$
%
```

3 Display a selected comment text in a monospace font, as follows:

a Position the cursor in the following comment, which appears in line 9.

```
% Calculate and plot y = sin(x).
```

b To make the equation `y = sin(x)` appear in monospace font in the published document, select the equation and then select **Cell> Insert Text Markup> Monospaced Text**. Line 9 should appear as follows:

```
% Calculate and plot |y = sin(x)|.
```

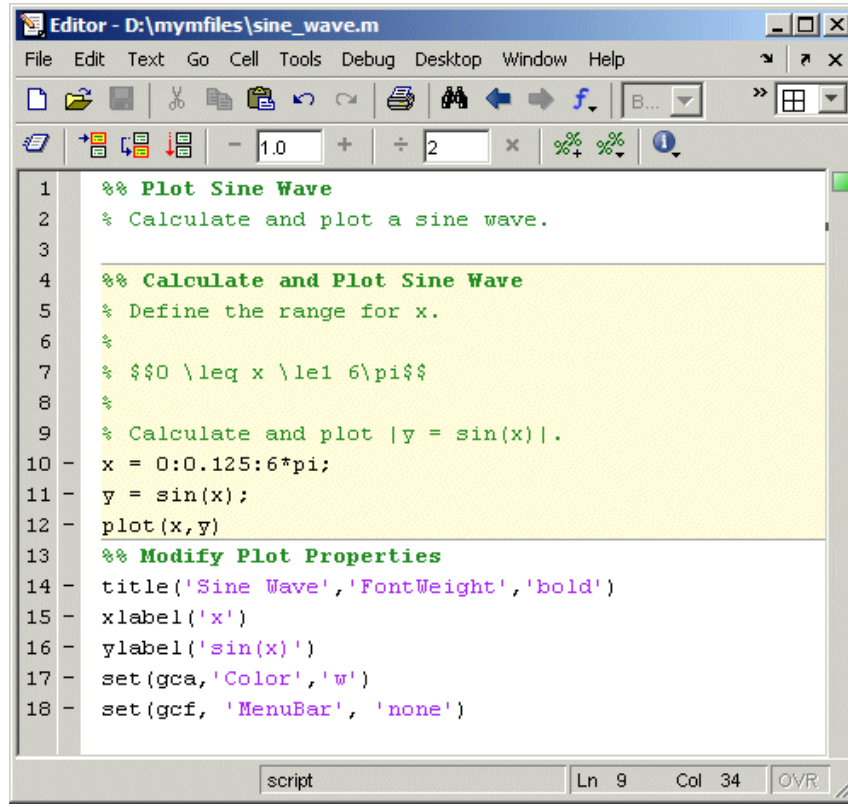
4 To reduce the size of the published figure, select **File > Preferences > Editor/Debugger > Publishing Images**. In the **Preferences** dialog box, for **Resize image**, select **Restrict height to** and enter `200`. Click **OK** to close the dialog box.

5 Select **File > Save and Publish to HTML**.

The HTML file displays in the MATLAB Web Browser, as shown at the start of this example, “Example of Publishing with Text Markup” on page 8-6.

6 By default, MATLAB stores the HTML document, `sine_wave.html`, and the associated image files in `d:/mymfiles/html` for this example.

The file `sine_wave.m` now appears as shown in the following illustration.


Editor - D:\mymfiles\sine_wave.m
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
2
%% Plot Sine Wave
% Calculate and plot a sine wave.
%% Calculate and Plot Sine Wave
% Define the range for x.
%
% $$0 \leq x \le1 6\pi$$
%
% Calculate and plot |y = sin(x)|.
x = 0:0.125:6*pi;
y = sin(x);
plot(x,y)
%% Modify Plot Properties
title('Sine Wave','FontWeight','bold')
xlabel('x')
ylabel('sin(x)')
set(gca,'Color','w')
set(gcf, 'MenuBar', 'none')
script
Ln 9 Col 34
OVR

Marking Up Text in Cells for Publishing

In this section...

Overview of Text Markup

When you publish an M-file and results, you can mark up the file using cell features—this adds and formats comments in the published file. You can include the markup as you write the basic code, mark up the file after you've written the code, or do both. The markup applies to any of the available publishing options: HTML, XML, LaTeX, Word, and PowerPoint.

Any cell features you use for evaluating and improving your code are used for publishing purposes as well. The "Example of Publishing Without Text Markup" on page 8-4 shows how the cells used for improving an M-file appear when the M-file is published. You might want to change the existing cells for publishing purposes, but note that this changes the cells for evaluation purposes as well. For example, to have text markup and formatted comments in the output document, the comments must appear at the start of a cell, before any code.

Mark up comment text in one of two ways:

- Use **Cell > Insert Text Markup** menu items to format the code, which automatically inserts the markup symbols for you. This is not available for all markup options.

- Type the markup symbols directly in the code. Note that what you type is the same as the code that results if you instead use the equivalent menu item.

The following tables describe each markup option and how to use it. The tables refer to “Example of Publishing with Text Markup” on page 8-6:

Text Markup for Cell Breaks, Headings, and Formatted Comments

Cell breaks, headings, and formatted comments are the structural elements that control the overall output format of the published document.

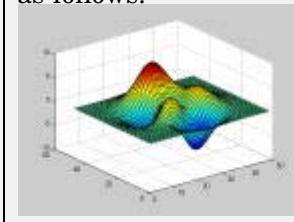
Format	How to Produce Format	Resulting Code	Published Results
Overall document heading and introductory text	**1** Position the cursor anywhere in the editor. **2** Select **Cell > Insert Text Markup > Document Title and Introduction.** **3** Replace **`DOCUMENT TITLE`** in the resulting code with your desired cell heading. **4** Replace `INTRODUCTORY TEXT` in the resulting code with text that introduces the M-file. See step in the example.	**`%% DOCUMENT TITLE`** `% INTRODUCTORY TEXT` In the example, the overall heading is **`%% Plot Sine Wave`**	Document title is formatted as a top-level heading (`h1` in HTML), using a large size, bold font. Introductory text appears as formatted text.
Section title, formatted comments, and a cell break	**1** Position the cursor where you want to insert a new cell. **2** Select **Cell > Insert Text Markup > Section Title with Cell Break.** **3** Replace **`SECTION TITLE`** with your desired title. **4** Replace `DESCRIPTIVE TEXT` with text that describes the cell.	`%%` **`SECTION TITLE`** `% DESCRIPTIVE TEXT` In the example, the section titles are **`%% Calculate and Plot Sine Wave`** **`%% Modify Plot Properties`** Note that descriptive text must appear before the first line of executable code in a cell.	Formatted as a heading (`h2` in HTML), using a medium size, bold font. Descriptive text appears as formatted text in the published output.

Text Markup for Indented Text, Lists, and Graphics

Indented text, lists, and graphics are types of block styles. Block styles control the appearance of large sections of text within the final published document. A block is a series of comment lines within a cell's descriptive text that starts and ends with a blank comment line, or the beginning or end of the descriptive text.

Format	How to Produce Format	Resulting Code and Explanation	Published Results

Indented text	**1** Position the cursor before the line where you want to add indented text.
	
	2 Select **Cell > Insert Text Markup > Preformatted Text**.
	
	3 Replace the resulting code with your desired text, including tabs and spaces. Be careful, however, to keep the percent signs (%) at the beginning of each line.
		`%  PREFORMATTED`
		`% TEXT`
		
		More than one space at the start of a block distinguishes it as preformatted.
			The indents, spacing, and line breaks in the M-file are preserved in the output:
			`  PREFORMATTED`
			`TEXT`

Format	How to Produce Format	Resulting Code and Explanation	Published Results
Image	**1** Position the cursor before the line where you want to add a graphic. **2** Select **Cell > Insert Text Markup > Image**. **3** Replace the sample text inserted, `FILENAME.PNG`, with filename of the graphic you want to insert. Keep the percent sign (%) and angle brackets (<< >>) . The image file must be in the same directory as the output file, or you must specify a relative path to the image file, from the output file.	`% <<FILENAME.PNG>>`	If you replace `FILENAME.PNG` with `surfpeaks.jpg`, the published results appear as follows: 

Format	How to Produce Format	Resulting Code and Explanation	Published Results

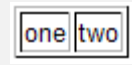
Bulleted list	**1** Position the cursor before the line where you want to add a bulleted list.
	2 Select **Cell > Insert Text Markup > Bulleted List**.
	3 Replace the sample text inserted, `ITEM 1` and `ITEM 2`, with your desired text. Be careful, however, to keep the percent signs (%) and asterisk (*) at the beginning of each line.
		`% * ITEM1`
		`% * ITEM2`
		The asterisk (`*`) at the start of a line indicates it is a bulleted list item.
			• `ITEM1`
			• `ITEM2`

Numbered list	**1** Position the cursor before the line where you want to add a numbered list.
	2 Select **Cell > Insert Text Markup > Numbered List**.
	3 Replace the inserted text , `ITEM 1` and `ITEM 2`, with your desired text. Do not, however, replace the percent sign (%) and number sign (#) that appear at the beginning of each line.
		`% # ITEM1`
		`% # ITEM2`
		The number sign (#) at the start of a line indicates it is a numbered list item.
			1 `ITEM1`
			2 `ITEM2`

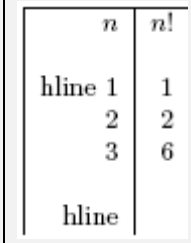
Text Markup for HTML, LaTeX, and TeX Equation Output Types

You can specify the output type for a published M-file as HTML, LaTeX or TeX equation. When you publish the M-file, use the **File > Publish to** menu options accordingly. For details, see “Publishing M-Files Using Cells” on page 8-24.

Note When you markup text for the HTML or LaTex output type, that text is published only when the specified output type matches the markup type. For example, if you add HTML markup, but then select **File > Publish To > LaTeX**, the text enclosed within the HTML markup is not published. Similarly, if you add LaTeX markup, but then select **File > Publish To > HTML** or click the Publish to HTML button (), the text enclosed within the LaTeX markup is not published.

Format	How to Produce Format	Resulting Code	Published Results
HTML markup	1 Position the cursor before the line where you want to add HTML markup. 2 Select **Cell > Insert Text Markup > HTML Markup**. 3 Replace the inserted HTML markup (as shown in the next column) with your desired text. Be careful, however, to keep the percent signs (%) at the beginning of each line.	`% <table border=1><tr> <td>one</td><td>two </td></tr></table>% </html>`	

Format	How to Produce Format	Resulting Code	Published Results

LaTeX markup	**1** Position the cursor before the line where you want to add LaTeX markup.
	2 Select **Cell > Insert Text Markup > LaTeX Markup**.
	3 Replace the inserted LaTeX markup (as shown in the next column) with your desired text. Be careful, however, to keep the percent signs (%) at the beginning of each line.
		`% <latex>`
		`%`
		`\begin{tabular}{|r|r}`
		`% \hline $n$&$n!$\\`
		`\\hline 1&1\\ 2&2\\`
		`3&6\\ \\hline`
		`% \end{tabular}`
		`% </latex>`
			

TeX Equations and symbols	**1** Position the cursor before the line where you want to add an equation or symbols.
	2 Select **Cell > Insert Text Markup > TeX Equation**.
	3 Replace the inserted text inserted, `e^{\pi i} + 1 = 0`, with your desired TeX equation. See step in the example.
		`% $$e^{\pi i} + 1 = 0$$`
		`%`
		The `$$` at the start and end of a block description distinguish the TeX equation.
		For a list of symbols you can display and the character sequence to create them, see the `String` property on MATLAB graphics reference page for `Text Properties`.
		In the example, line 7 includes a TeX equation:
		`% $$0\leq 6\pi$$`
			$e^{\pi i} + 1 = 0$

Text Markup for Bold, Italic, and Monospaced Text Formats

You can mark up selected strings in the M-file comments so that they appear in bold, italic or monospaced text formats when you publish the M-file.

Format	How to Produce Format	Resulting Code	Published Results
Bold text	Follow these steps to bold existing text: **1** Within a comment, select text that you want to be bold. **2** Select **Cell > Insert Text Markup > Bold Text**. To insert sample text, that you will replace with your desired text, follow these steps: **1** Select **Cell > Insert Text Markup > Bold Text**. **2** Replace the inserted text with the text that you want to be bold.	`% *BOLD TEXT*`	**BOLD TEXT**

Format	How to Produce Format	Resulting Code	Published Results
Italic text	Follow these steps to bold existing text: **1** Within a comment, select text that you want to be italic. **2** Select **Cell > Insert Text Markup > Italic Text**. To insert sample text, that you will replace with your desired text, follow these steps: **1** Select **Cell > Insert Text Markup > Italic Text**. **2** Replace the inserted text with the text you want to be italic.	`% _ITALIC TEXT_`	*ITALIC TEXT*

Monospaced text	Follow these steps to bold existing text:
	1 Within a comment, select text that you want to be monospaced.
	2 Select **Cell > Insert Text Markup > Monospaced Text**.
	To insert sample text, that you will replace with your desired text, follow these steps:
	1 Select **Cell > Insert Text Markup > Monospaced Text**.
	2 Replace the inserted text with the text you want to be monospaced.
		`% |MONOSPACED TEXT|`
		In the example, monospaced text is added in line 9:
		`% Calculate and plot |y=sin(x)|.`
			`MONOSPACED TEXT`

Text Markup for Inline Links

When you specify hypertext links within an M-file, when you publish the document the hypertext links become active links to a URL on the web.

Format	How to Produce Format	Resulting Code	Published Results
URL as hyperlinked text	**1** Within a comment, position the cursor where you want to insert the hypertext link. **2** Select **Cell > Insert Text Markup > Hyperlinked Text**. The Editor/Debugger inserts the following code: `<http://www.mathworks .com The MathWorks>` **3** Replace www.mathworks.com with your desired URL. **4** Delete the string, The MathWorks.	`% <http: //www.mathworks .com>>`	http://www.mathworks.com
Hyperlinked text without a printed URL	**1** Within a comment, position the cursor where you want to insert the hypertext link. **2** Select **Cell > Insert Text Markup > Hyperlinked Text**. The Editor/Debugger inserts the following code: <http://www.mathworks .com The MathWorks> **3** Replace www.mathworks.com with your desired URL. **4** Replace The MathWorks with the text that you want to appear as the hyperlinked text.	`% <http: //www.mathworks .com The MathWorks>>`	The MathWorks

Publishing M-Files Using Cells

In this section...

How to Publish an M-File

When you publish an M-file that contains cells and text markup, MATLAB produces an output document consisting of the M-file code, comments, and results.

1. When you publish an M-file function, MATLAB does not run the code or include results from running the code in the output document—for M-file functions, clear the **Evaluate code** preference prior to publishing.

2. After adding cells and text markup to an M-file, select **File > Publish To** and select an output format from those listed in the menu: **HTML**, **XML**, **LaTeX**, **Word**, or **PowerPoint**. If the M-file contains unsaved changes, the menu item becomes **Save and Publish To**.

   You can also publish to the default output format specified in **Preferences** using the Publish button in the Editor/Debugger toolbar.

MATLAB displays the published document in the appropriate tool for the selected output format:

- HTML displays in the MATLAB Web Browser.
- XML displays in the MATLAB Editor/Debugger.
- LaTeX displays in the MATLAB Editor/Debugger.
- Word displays in Microsoft Word.
- PowerPoint displays in Microsoft PowerPoint.

Note Publishing to Microsoft Word and to PowerPoint features are available only on Windows systems that have the applications installed. Supported Word and PowerPoint versions are 2000, 2002, 2003, and 2007.

The published file contains the formatted comments, code with syntax highlighting and a gray background to distinguish it from results, and results for each cell. It also contains a **Contents** heading at the top of the file with a bulleted list of links to the named cells in the rest of the document.

When code in a cell creates or modifies a figure, the published file includes an image of the figure. If the code in a given cell modifies a figure more than once, the published file includes only one image of the figure, that being the last version of the figure.

When publishing to HTML, the M-file code is included at the end of published HTML file as comments. Use the `grabcode` function to extract the code from the HTML file.

Function Alternative

From the Command Window, run the `publish` function to run the M-file and publish the results. See the `publish` function reference page for options you can set.

About Published M-Files

Published Filenames and Locations

MATLAB names the published file the same as the M-file that produced it, adding the relevant extension for the selected output format: `.html`, `.xml`, `.tex`, `.doc`, or `.ppt`. MATLAB stores this output file, along with supporting files such as images of figure windows, in the `html` subdirectory under the directory containing the M-file you published.

For example, when you publish `d:/mymfiles/sine_wave.m` to HTML, MATLAB creates a directory `d:/mymfiles/html` that includes the published document `sine_wave.html`. Any figure windows produced by running the M-file appear as image files in the directory, for example, `sine_wave_img.png`.

TeX equations are image files as well; in the example, the equation file is `sine_wave_eq_eq####.png`. MATLAB creates a thumbnail file for the document, `sine_wave_img_thumbnail.png` in the example, if that preference is selected — see in the online documentation.

Publishing Code that Displays Hyperlinks in Command Window

If the M-file you publish contains statements that display hyperlinks in the MATLAB Command Window, the published document shows the code rather than the hyperlinks.

For example

```
disp('<a href="http://www.mathworks.com">Link to MathWorks</a>')
```

displays

Link to MathWorks

in the Command Window. You can click the link to go to the MathWorks Web site. When that `disp` statement is in an M-file you publish, the hyperlink tag and the text between it, that is,

```
<a href="http://www.mathworks.com">Link to MathWorks</a>
```

rather than the link, appears in the published document.

Similar results occur if you include

```
help matlab_functioname
```

in an M-file.

Modifying Published Output Using Preferences

Use preferences to control execution, output, and options related to images created during publishing. For details about these preferences, click the **Help** button in the **Preferences** dialog box for those panes.

Notebook for Publishing to Word

In this section...

Note Notebook is available only on Windows systems that have Microsoft Word installed. For supported versions of Word, see "Configuring Notebook" on page 8-53.

Using Notebook to Create an M-book

Notebook enables you to access the numeric computation and visualization software of MATLAB from within the word processing environment, Microsoft Word. Using Notebook, you can create a document, called an *M-book*, that contains text, MATLAB commands, and the output from MATLAB commands.

You can think of an M-book as a record of an interactive MATLAB session annotated with text, or as a document embedded with live MATLAB commands and output. Notebook is useful for creating electronic or printed records of MATLAB sessions, class notes, textbooks or technical reports.

See Also Publishing Using Cells

As an alternative to Notebook, consider publishing using cells. For more information, see "Publishing to HTML, XML, LaTeX, Word, and PowerPoint Using Cells" on page 8-2.

Creating or Opening an M-Book

Creating an M-Book from MATLAB

To create a new M-book from within MATLAB, type

```
notebook
```

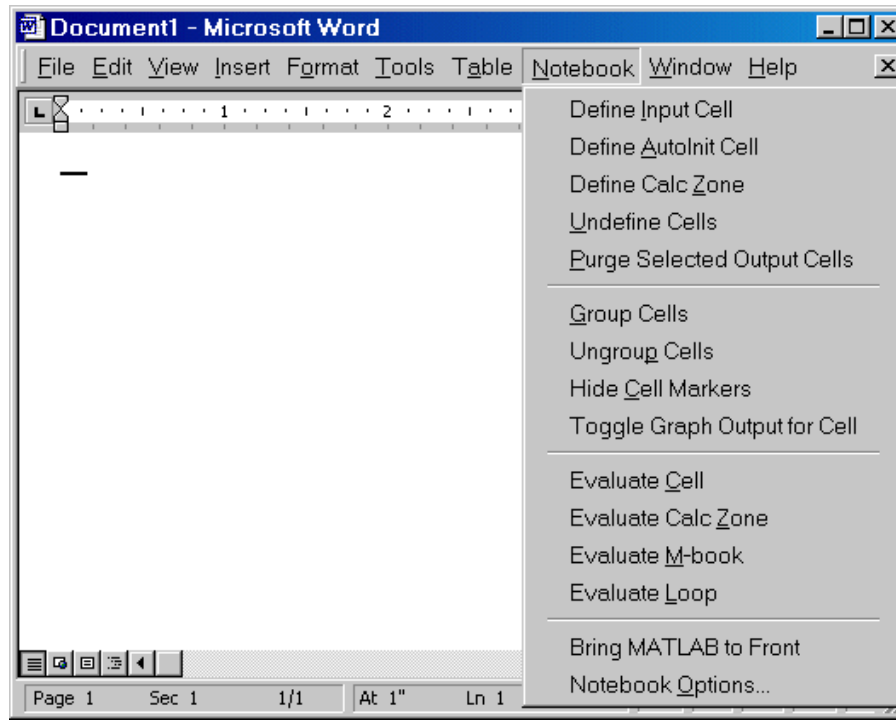
in the Command Window. If you are running Notebook for the first time, you might need to configure it. See “Configuring Notebook” on page 8-53 for more information.

Notebook starts Microsoft Word on your system and creates a new M-book, called `Document1`.

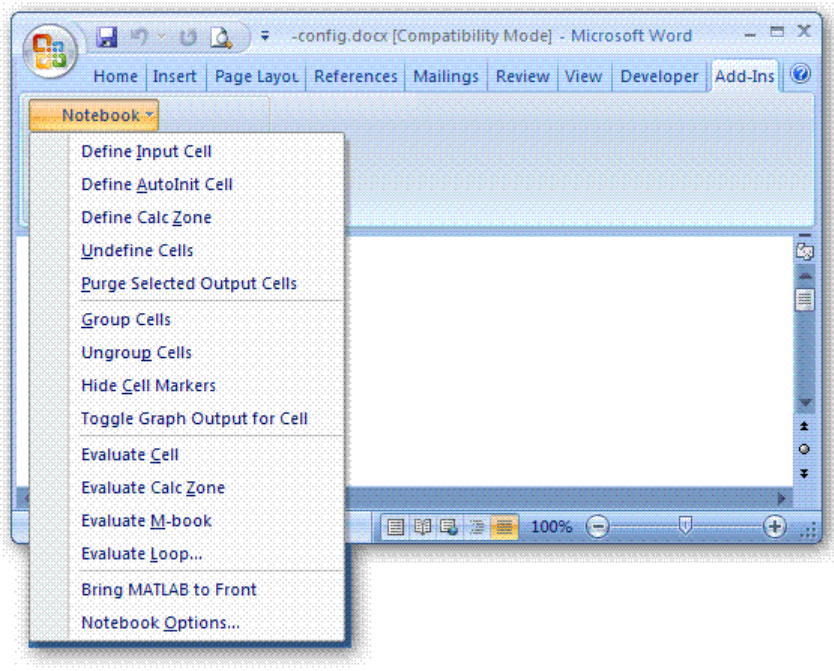
When Word is opening, if a dialog box appears asking you to enable or disable macros, choose to enable macros. Notebook defines Microsoft Word macros that enable MATLAB to interpret the different types of cells that hold MATLAB commands and their output. For more information on macro security, see “Configuring Notebook” on page 8-53.

Depending on the version of Word you are using, one of the following occurs:

- In Word 2000, 2002, and 2003, Notebook adds the **Notebook** menu to the Word menu bar, as shown in the following illustration. Use this menu to access Notebook features.



- In Word 2007, Notebook adds the **Notebook** menu to the Word **Add-Ins** tab, as shown in the following illustration. Use this menu to access Notebook features.

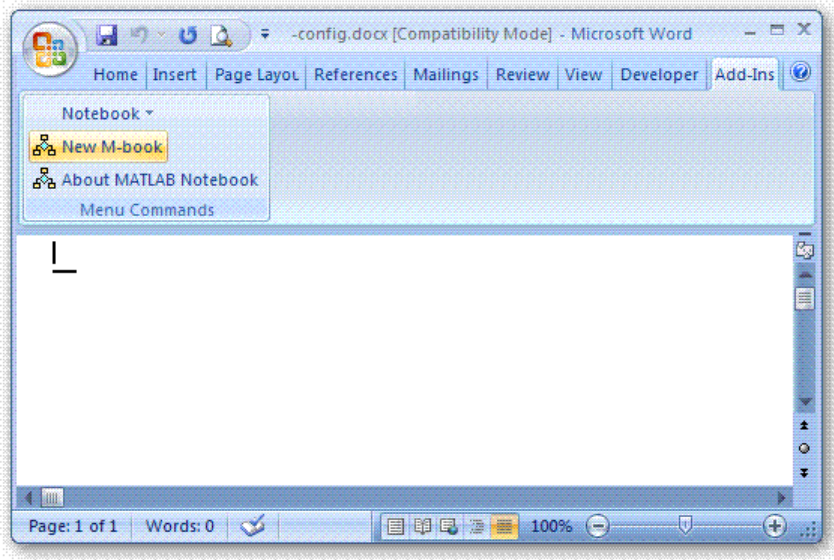


Microsoft product screen shot reprinted with permission from Microsoft Corporation.

Creating an M-Book While Running Notebook

With Notebook running, you can create a new M-book as follows:

- In Word 2000, 2002, and 2003, select **File > New M-book**
- In Word 2007, select **Add-Ins > New M-book**, as shown in the following figure:



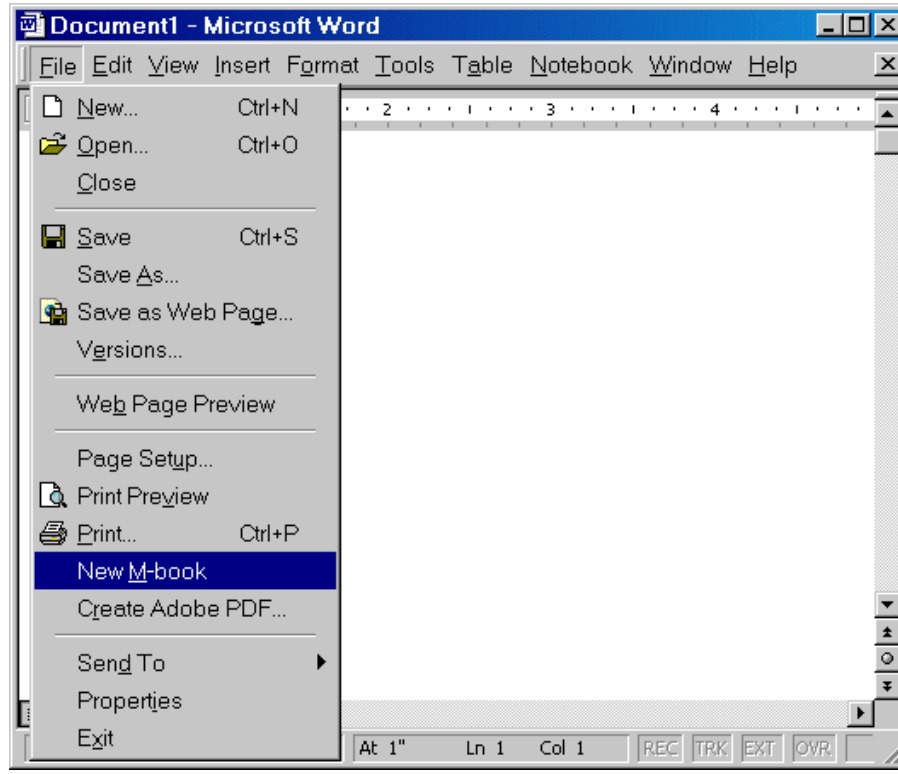
Microsoft product screen shot reprinted with permission from Microsoft Corporation.

Opening an Existing M-Book

You can use the `notebook` command to open an existing M-book

```
notebook filename
```

where `filename` is the M-book you want to open. Or you can double-click an M-book file in a Windows file management tool, such as Explorer.



When you double-click on an M-book, Microsoft Word opens the M-book and starts MATLAB if it is not already running. Notebook adds the **Notebook** menu to the Word menu bar and adds **New M-book** to the **File** menu.

Converting a Word Document to an M-Book

To convert a Word document to an M-book, follow these steps, depending on the version of Word you are using:

- Microsoft Word 2000, 2002, or 2003:

  1 Create a new M-book.

  2 From the **Insert** menu, select **File**.

3 Select the file you want to convert.

4 Click **OK**.

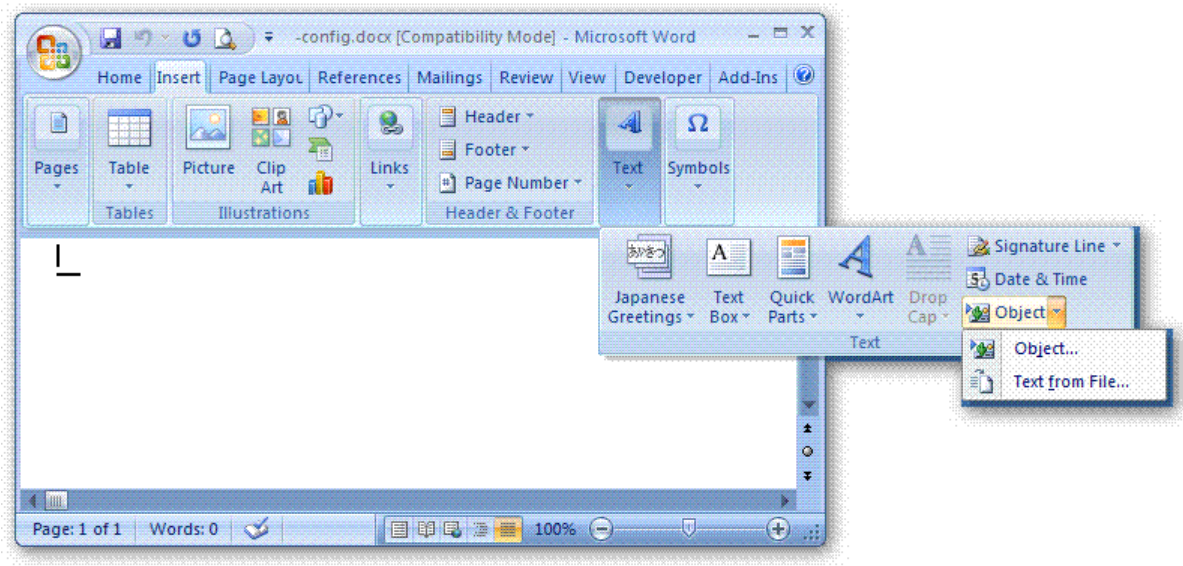
- Microsoft Word 2007:

1 Create a new M-book.

2 From the **Insert** tab, in the **Text** group, click the arrow next to **Object** and then click **Text from File**, as shown in the image that follows.

The Insert File dialog box opens.

3 In the Insert File dialog box, select the file that you want to convert, and then click **OK**.



Microsoft product screen shot reprinted with permission from Microsoft Corporation.

Entering MATLAB Commands in an M-Book

Note A good way to learn how to use Notebook is to open the sample M-book, `Readme.doc`, and try out the various techniques described in this section. You can find this file in the *`matlabroot`*`/notebook/pc` directory.

You enter MATLAB commands in an M-book the same way you enter text in any other Word document. For example, you can enter the following text in a Word document. The example uses text in Courier Font but you can use any font:

```
Here is a sample M-book.

a = magic(3)
```

To execute the MATLAB `magic` command in this document, you must

- Define the command as an input cell
- Evaluate the input cell

MATLAB displays the output of the command in the Word document in an output cell.

Protecting the Integrity of Your Workspace in M-Books

When you work on more than one M-book in a single word processing session, note that:

- Each M-book uses the same “copy” of MATLAB.
- All M-books share the same workspace.

If you use the same variable names in more than one M-book, data used in one M-book can be affected by another M-book. You can protect the integrity of your workspace by specifying the `clear` command as the first autoinit cell in the M-book.

Ensuring Data Consistency in M-Books

An M-book can be thought of as a sequential record of a MATLAB session. When executed in order, from the first MATLAB command to the last, the M-book accurately reflects the relationships among these commands.

If, however, you change an input cell or output cell as you refine your M-book, Notebook does not automatically recalculate input cells that depend on either the contents or the results of the changed cells. As a result, the M-book may contain inconsistent data.

When working on an M-book, you might find it useful to select **Evaluate M-book** periodically to ensure that your M-book data is consistent. You could also use calc zones to isolate related commands in a section of the M-book. You can then use **Evaluate Calc Zone** to execute only those input cells contained in the calc zone.

Debugging and Notebook

Do not use debugging functions or use the Editor/Debugger while evaluating cells with Notebook. Instead debug M-files from within MATLAB, and then after completing debugging, clear all the breakpoints and access the M-file via Notebook. If you debug while evaluating from Notebook, you might experience problems with MATLAB.

Defining MATLAB Commands as Input Cells for Notebook

In this section...

For information about evaluating the input cells you define, see "Evaluating MATLAB Commands with Notebook" on page 8-41.

Defining Commands as Input Cells for Notebook

To define a MATLAB command in a Word document as an input cell,

1. Type the command into the M-book as text. For example,

```
This is a sample M-book.

a = magic(3)
```

2. Position the cursor anywhere in the command and select **Notebook > Define Input Cell** or press **Alt+D**. If the command is embedded in a line of text, use the mouse to select it. Notebook defines the MATLAB command as an input cell:

```
This is a sample M-book.

[a = magic(3)]
```

Note how Notebook changes the character font of the text in the input cell to a bold, dark green color and encloses it within *cell markers*. Cell markers are bold, gray brackets. They differ from the brackets used to enclose matrices by their size and weight. For information about changing these default formats, see "Modifying Styles in the M-Book Template" on page 8-47.

Defining Cell Groups for Notebook

You can collect several input cells into a single input cell. This is called a *cell group*. Because all the output from a cell group appears in a single output cell that Notebook places immediately after the group, cell groups are useful when several MATLAB commands are needed, such as, to fully define a graphic.

For example, if you define all the MATLAB commands that produce a graphic as a cell group and then evaluate the cell group, Notebook generates a single graphic that includes all the graphic components defined in the commands. If instead you define all the MATLAB commands that generate the graphic as separate input cells, evaluating the cells generates multiple graphic output cells.

See “Evaluating Cell Groups with Notebook” on page 8-42 for information about evaluating a cell group. For information about ungrouping a cell group, see “Ungroup Cells” on page 8-61.

Creating a Cell Group for Notebook

To create a cell group,

1. Use the mouse to select the input cells that are to make up the group.
2. Select **Notebook > Group Cells** or press **Alt+G**.

Notebook converts the selected cells into a cell group and replaces cell markers with a single pair that surrounds the group:

```
This is a sample cell group.

[date
a = magic(3) ]
```

Note the following:

- A cell group cannot contain output cells. If the selection includes output cells, Notebook deletes them.
- A cell group cannot contain text. If the selection includes text, Notebook places the text after the cell group. However, if the text precedes the first input cell in the selection, Notebook leaves it where it is.

- If you select part or all of an output cell but not its input cell, Notebook includes the input cell in the cell group.

When you create a cell group, Notebook defines it as an input cell unless its first line is an autoinit cell, in which case Notebook defines the group as an autoinit cell.

Defining Autoinit Input Cells for Notebook

You can use *autoinit cells* to specify MATLAB commands to be automatically evaluated each time an M-book is opened. This is a quick and easy way to initialize the workspace. *Autoinit cells* are simply input cells with the following additional characteristics:

- Notebook evaluates the autoinit cells when it opens the M-book.
- Notebook displays the commands in autoinit cells using dark blue characters.

Autoinit cells are otherwise identical to input cells.

Creating an Autoinit Cell for Notebook

You can create an autoinit cell in two ways:

- Enter the MATLAB command as text, then convert the command to an autoinit cell by selecting **Notebook > Define AutoInit Cell**.
- If you already entered the MATLAB command as an input cell, you can convert the input cell to an autoinit cell. Either select the input cell or position the cursor in the cell, then select **Notebook > Define AutoInit Cell**.

See “Evaluating MATLAB Commands with Notebook” on page 8-41 for information about evaluating autoinit cells.

Defining Calc Zones for Notebook

You can partition an M-book into self-contained sections, called *calc zones*. A calc zone is a contiguous block of text, input cells, and output cells. Notebook inserts Microsoft Word section breaks before and after the section to define

the calc zone. The section break indicators include bold, gray brackets to distinguish them from standard Word section breaks.

You can use calc zones to prepare problem sets, making each problem a separate calc zone that can be created and tested on its own. An M-book can contain any number of calc zones.

Note Using calc zones does not affect the scope of the variables in an M-book. Variables used in one calc zone are accessible to all calc zones.

Creating a Calc Zone

After you create the text and cells you want to include in the calc zone, you define the calc zone by following these steps:

1. Select the input cells and text to be included in the calc zone.
2. Select **Notebook > Define Calc Zone**.

Note You must select an input cell and its output cell in their entirety to include them in the calc zone.

See “Evaluating a Calc Zone with Notebook” on page 8-44 for information about evaluating a calc zone.

Converting an Input Cell to Text with Notebook

To convert an input cell (or an autoinit cell or a cell group) to text,

1. Select the input cell with the mouse or position the cursor in the input cell.
2. Select **Notebook > Undefine Cells** or press **Alt+U**.

When Notebook converts the cell to text, it reformats the cell contents according to the Microsoft Word Normal style. For more information about M-book styles, see “Modifying Styles in the M-Book Template” on page

8-47. When you convert an input cell to text, Notebook also converts the corresponding output cell to text.

Evaluating MATLAB Commands with Notebook

In this section...

Evaluating Input Commands with Notebook

After you define a MATLAB command as an input cell, or as an autoinit cell, you can evaluate it in your M-book. Use the following steps to define and evaluate a MATLAB command:

1. Type the command into the M-book as text. For example:

   ```
   This is a sample M-book

   a = magic(3)
   ```

2. Position the cursor anywhere in the command. If the command is embedded in a line of text, use the mouse to select it. Then select **Notebook > Define Input Cell** or press **Alt+D**.

   Notebook defines the MATLAB command as an input cell. For example:

   ```
   This is a sample M-book

   [a = magic(3)]
   ```

3 Specify the input cell to be evaluated by selecting it with the mouse or by placing the cursor in it. Then select **Notebook > Evaluate Cell** or press **Ctrl+Enter**.

   Notebook evaluates the input cell and displays the results in a output cell immediately following the input cell. If there is already an output cell, Notebook replaces its contents, wherever it is in the M-book. For example:

```
This is a sample M-book.

[a = magic(3) ]

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

The text in the output cell is blue and is enclosed within cell markers. Cell markers are bold, gray brackets. They differ from the brackets used to enclose matrices by their size and weight. Error messages appear in red. For information about changing these default formats, see “Modifying Styles in the M-Book Template” on page 8-47.

Evaluating Cell Groups with Notebook

You evaluate a cell group the same way you evaluate an input cell (because a cell group is an input cell):

1 Position the cursor anywhere in the cell or in its output cell.

2 Select **Notebook > Evaluate Cell** or press **Ctrl+Enter**.

For information about creating a cell group, see “Defining Cell Groups for Notebook” on page 8-37.

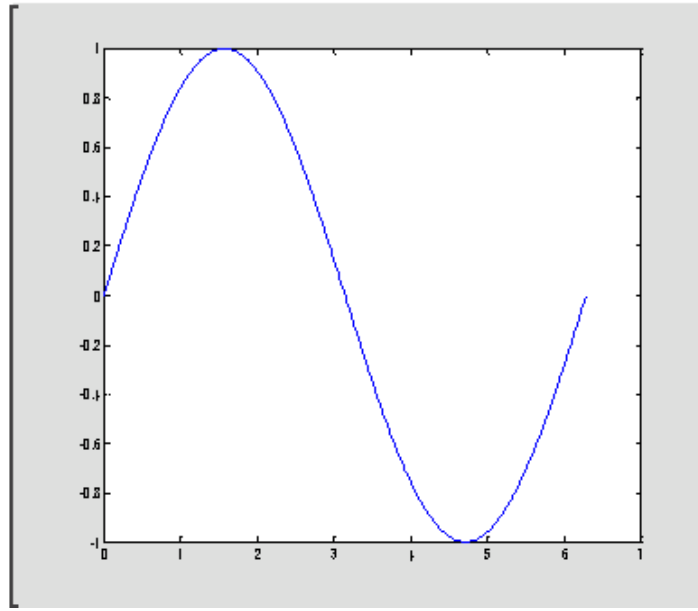
When MATLAB evaluates a cell group, the output for all commands in the group appears in a single output cell. By default, Notebook places the output cell immediately after the cell group the first time the cell group is evaluated. If you evaluate a cell group with an existing output cell, Notebook places the results in the output cell wherever it is located in the M-book.

Note Text or numeric output always comes first, regardless of the order of the commands in the group.

The illustration shows a cell group and the figure created when you evaluate the cell group.

This is a sample M-book with a cell group.

```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y)
```



Evaluating a Range of Input Cells with Notebook

To evaluate more than one MATLAB command contained in different but contiguous input cells,

1 Select the range of cells that includes the input cells you want to evaluate. You can include text that surrounds input cells in your selection.

2 Select **Notebook > Evaluate Cell** or press **Ctrl+Enter**.

Notebook evaluates each input cell in the selection, inserting new output cells or replacing existing ones.

Evaluating a Calc Zone with Notebook

To evaluate a calc zone,

1 Position the cursor anywhere in the calc zone.

2 Select **Notebook > Evaluate Calc Zone** or press **Alt+Enter**.

For information about creating a calc zone, see "Defining Calc Zones for Notebook" on page 8-38.

By default, Notebook places the output cell immediately after the calc zone the first time the calc zone is evaluated. If you evaluate a calc zone with an existing output cell, Notebook places the results in the output cell wherever it is located in the M-book.

Evaluating an Entire M-Book

To evaluate the entire M-book, either select **Notebook > Evaluate M-book** or press **Alt+R**.

Notebook begins at the top of the M-book regardless of the cursor position and evaluates each input cell in the M-book. As it evaluates the M-book, Notebook inserts new output cells or replaces existing output cells.

Controlling Execution of Multiple Commands

When you evaluate an entire M-book, and an error occurs, evaluation continues. If you want to stop evaluation if an error occurs, follow this procedure:

1 Select **Notebook > Notebook Options**.

The **Notebook Options** dialog box opens.

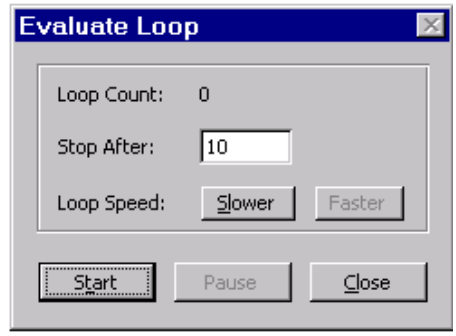
2 Select the **Stop evaluating on error** check box and click **OK**.

Using a Loop to Evaluate Input Cells Repeatedly with Notebook

To evaluate a sequence of MATLAB commands repeatedly,

1 Use the mouse to select the input cells, including any text or output cells located between them.

2 Select **Notebook > Evaluate Loop** or press **Alt+L**. Notebook displays the **Evaluate Loop** dialog box.



3 Enter the number of times you want MATLAB to evaluate the selected commands in the **Stop After** field, then click **Start**. The button changes to **Stop**. Notebook begins evaluating the commands and indicates the number of completed iterations in the **Loop Count** field.

You can increase or decrease the delay at the end of each iteration by clicking **Slower** or **Faster**. Slower increases the delay. Faster decreases the delay.

To suspend evaluation of the commands, click **Pause**. The button changes to **Resume**. Click **Resume** to continue evaluation.

To stop processing the commands, click **Stop**. To close the **Evaluate Loop** dialog box, click **Close**.

Converting Output Cells to Text with Notebook

You can convert an output cell to text by undefining cells. If the output is numeric or textual, Notebook removes the cell markers and converts the cell contents to text according to the Microsoft Word Normal style. If the output is graphical, Notebook removes the cell markers and dissociates the graphic from its input cell, but does not alter its contents.

Note Undefining an output cell does not affect the associated input cell.

To undefine an output cell,

1. Select the output cell you want to undefine.
2. Select **Notebook > Undefine Cells** or press **Alt+U**.

Deleting Output Cells with Notebook

To delete output cells,

1. Select an output cell, using the mouse, or place the cursor in the output cell.
2. Select **Notebook > Purge Selected Output Cells** or press **Alt+P**.

If you select a range of cells, Notebook deletes all the output cells in the selected range, but any associate input cells remain intact.

Printing and Formatting an M-Book

In this section...

Printing an M-Book

You can print all or part of an M-book by doing one of the following, depending on the version of Microsoft Word you are using:

- In Microsoft Word 2000, 2002, 2003 — Select **File > Print**.
- In Microsoft Word 2007 — Select **Microsoft Office Button > Print**

Word follows these rules when printing M-book cells and graphics:

- Cell markers are not printed.
- Input cells, autoinit cells, and output cells (including error messages) are printed according to their defined styles. If you prefer to print these cells using black type instead of colors or shades of gray, you can modify the styles.

Modifying Styles in the M-Book Template

You can control the appearance of the text in your M-book by modifying the predefined styles stored in the M-book template, `m-book.dot`. These styles control the appearance of text and cells. By default, M-books use the Word Normal style for all other text.

For example, if you print an M-book on a color printer, input cells appear dark green, output and autoinit cells appear dark blue, and error messages appear red. If you print the M-book on a grayscale printer, these cells appear as

shades of gray. To print these cells using black type, you need to modify the color of the Input, Output, AutoInit, and Error styles in the M-book template.

The table below describes the default styles used by Notebook. If you modify styles, you can use the information in the tables below to help you return the styles to their original settings. For general information about using styles in Word documents, see the Word documentation.

Style	Font	Size	Weight	Color
Normal	Times New Roman	10 points	N/A	Black
AutoInit	Courier New	10 points	Bold	Dark blue
Error	Courier New	10 points	Bold	Red
Input	Courier New	10 points	Bold	Dark green
Output	Courier New	10 points	N/A	Blue

When you change a style, Word applies the change to all characters in the M-book that use that style and gives you the option to change the template. Be cautious about making changes to the template. If you choose to apply the changes to the template, you will affect all new M-books you create using the template. See the Word documentation for more information.

Choosing Loose or Compact Format for Notebook

You can specify whether a blank line appears between the input and output cells by selecting the loose or compact format:

1. Select **Notebook > Notebook Options**.
2. In the **Notebook Options** dialog box, select either **Loose** or **Compact**. Loose format adds an empty line. Compact format does not.
3. Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for output generated *after* you click **OK**. To affect existing input or output cells, you must reevaluate the cells.

Controlling Numeric Output Format for Notebook

To change how Notebook displays numeric output,

1. Select **Notebook > Notebook Options**.
2. In the **Notebook Options** dialog box, select a format from the **Numeric Format** list. These settings correspond to the choices available with the MATLAB `format` command.
3. Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for output generated *after* you click **OK**. To affect existing input or output cells, you must reevaluate the cells.

Controlling Graphic Output for Notebook

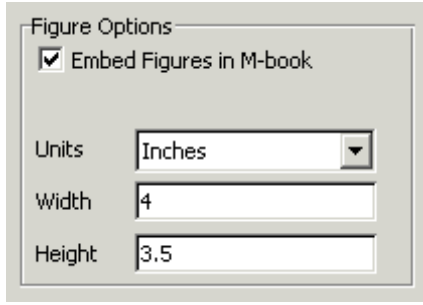
This section describes how to control several aspects of the graphic output produced by MATLAB commands in an M-book, including

Embedding Graphic Output in the M-Book

By default, graphic output is embedded in an M-book. To display graphic output in a separate figure window,

1 Select **Notebook > Notebook Options**.

2 In the **Notebook Options** dialog box, clear the **Embed Figures in M-book** check box.



3 Click **OK**.

Note Embedded figures do not include Handle Graphics objects generated by the `uicontrol` and `uimenu` functions.

Notebook determines whether to embed a figure in the M-book by examining the value of the figure object's `Visible` property. If the value of the property is `off`, Notebook embeds the figure. If the value of this property is `on`, all graphic output is directed to the current figure window.

Suppressing Graphic Output for Individual Input Cells in Notebook

If an input or autoinit cell generates figure output that you want to suppress,

1 Place the cursor in the input cell.

2 Select **Notebook > Toggle Graph Output for Cell**.

Notebook suppresses graphic output from the cell, inserting the string `(no graph)` after the input cell.

To allow graphic output for a cell, repeat the procedure. Notebook removes the `(no graph)` marker and allows graphic output from the cell.

Note **Toggle Graph Output for Cell** overrides the **Embed Figures in M-book** option, if that option is set.

Sizing Graphic Output in Notebook

To set the default size of embedded graphics in an M-book,

1. Select **Notebook > Notebook Options**.
2. In the **Notebook Options** dialog box, use the **Units**, **Width** and **Height** fields to set the size of graphics generated by the M-book.
3. Click **OK**.

Note Changes you make using the **Notebook Options** dialog box take effect for graphic output generated *after* you click **OK**. To affect existing input or output cells, you must reevaluate the cells.

You change the size of an existing embedded figure by selecting the figure, clicking the left mouse button anywhere in the figure, and dragging the resize handles of the figure. If you resize an embedded figure using its resize handles and then regenerate the figure, its size reverts to its original size.

Cropping Graphic Output in Notebook

To crop an embedded figure to cut off areas you do not want to show,

1. Select the graphic by clicking the left mouse button anywhere in the figure.
2. Hold down the **Shift** key.
3. Drag a sizing handle toward the center of the graphic.

Adding White Space Around Graphic Output in Notebook

You can add white space around an embedded figure by moving the boundaries of a graphic outward. Select the graphic, then hold down the **Shift** key and drag a sizing handle away from the graphic.

Configuring Notebook

After you install Notebook but before you begin using it, you must configure it. (Notebook is installed as part of the MATLAB installation process on Windows platforms. For more information, see the MATLAB installation documentation for your platform.)

Before configuring Notebook, you must specify that Word can use the Notebook macros.

- In Word 2000, 2002, and 2003 do either of the following:
  - Set the macro security level to medium: in Word, select **Tools > Macros > Security**, and in the resulting dialog box, choose **Medium**.
  - After starting Notebook, when Word first opens, a security warning dialog box appears. In the dialog box, select **Always trust macros from this source**. This allows you to use Notebook, but still maintain a high security level for other macros you use in Word.
- In Word 2007, follow the Word help instructions in the topic entitled "Enable or disable macros in Office documents."

To configure Notebook, type the following in the MATLAB Command Window:

```
notebook ('-setup')
```

MATLAB accesses the Windows system registry to locate Microsoft Word and the Word templates directory, and to identify the version of Word. MATLAB then copies Notebook's `m-book.dot` template to the Word templates directory. MATLAB Notebook supports Word versions 2000, 2002, 2003, and 2007.

When Notebook setup successfully finishes, MATLAB displays the message

```
Setup complete
```

Notebook Feature Reference

In this section...

This section provides reference information about each of the Notebook features, listed alphabetically. To use these features, select them from the **Notebook** menu in Microsoft Word. (In Word 2007, the **Notebook** menu is on the **Add-Ins** tab.)

Bring MATLAB to Front

Bring MATLAB to Front brings the MATLAB Command Window to the foreground.

Define Autoinit Cell

Define AutoInit Cell creates an autoinit cell by converting the current paragraph, selected text, or input cell. An autoinit cell is an input cell that is automatically evaluated whenever you open an M-book.

Result

If you select this feature while the cursor is in a paragraph of text, Notebook converts the entire paragraph to an autoinit cell. If you select this feature while text is selected, Notebook converts the text to an autoinit cell. If you select this feature while the cursor is in an input cell, Notebook converts the input cell to an autoinit cell.

Format

Notebook formats the autoinit cell using the AutoInit style, defined as bold, dark blue, 10-point Courier New.

See Also

For more information about autoinit cells, see “Defining Autoinit Input Cells for Notebook” on page 8-38.

Define Calc Zone

Define Calc Zone defines the selected text, input cells, and output cells as a calc zone. A calc zone is a contiguous block of related text, input cells, and output cells that describes a specific operation or problem.

Result

Notebook defines a calc zone as a Word document section, placing section breaks before and after the calc zone. However, Word does not display section breaks at the beginning or end of a document.

See Also

For information about evaluating calc zones, see “Evaluating a Calc Zone with Notebook” on page 8-44. For more information about document sections, see the Microsoft Word documentation.

Define Input Cell

Define Input Cell creates an input cell by converting the current paragraph, selected text, or autoinit cell. An input cell contains a MATLAB command.

Result

If you select this feature while the cursor is in a paragraph of text, Notebook converts the entire paragraph to an input cell. If you select this feature while text is selected, Notebook converts the text to an input cell. If you select this feature while the cursor is in an autoinit cell, Notebook converts the autoinit cell to an input cell.

Format

Notebook encloses the text in cell markers and formats the cell using the Input style, defined as bold, dark green, 10-point Courier New.

See Also

For more information about creating input cells, see “Defining MATLAB Commands as Input Cells for Notebook” on page 8-36. For information about evaluating input cells, see “Evaluating MATLAB Commands with Notebook” on page 8-41.

Evaluate Calc Zone

Evaluate Calc Zone sends the input cells in the current calc zone to MATLAB to be evaluated. The current calc zone is the Word section that contains the cursor.

Result

As Notebook evaluates each input cell, it generates an output cell. When you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book.

See Also

For more information, see “Evaluating a Calc Zone with Notebook” on page 8-44.

Evaluate Cell

Evaluate Cell sends the current input cell or cell group to MATLAB to be evaluated. An input cell contains a MATLAB command. A cell group is a single, multiline input cell that contains more than one MATLAB command. Notebook displays the output or an error message in an output cell.

Result

If you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book. If you evaluate a cell group, all output for the cell appears in a single output cell.

An input cell or cell group is the current input cell or cell group if

- The cursor is in the input cell or cell group.
- The cursor is at the end of the line that contains the closing cell marker for the input cell or cell group.
- The cursor is in the output cell for the input cell or cell group.
- The input cell or cell group is selected.

Note Evaluating a cell that involves a lengthy operation may cause a time-out. If this happens, Word displays a time-out message and asks whether you want to continue waiting for a response or terminate the request. If you choose to continue, Word resets the time-out value and continues waiting for a response. Word sets the time-out value; you cannot change it.

See Also

For more information, see “Evaluating MATLAB Commands with Notebook” on page 8-41. For information about evaluating the entire M-book, see “Evaluating an Entire M-Book” on page 8-44.

Evaluate Loop

Evaluate Loop evaluates the selected input cells repeatedly.

For more information, see “Using a Loop to Evaluate Input Cells Repeatedly with Notebook” on page 8-45.

Evaluate M-Book

Evaluate M-book evaluates the entire M-book, sending all input cells to MATLAB to be evaluated. Notebook begins at the top of the M-book regardless of the cursor position.

Result

As Notebook evaluates each input cell, it generates an output cell. When you evaluate an input cell for which there is no output cell, Notebook places the output cell immediately after the input cell that generated it. If you evaluate an input cell for which there is an output cell, Notebook replaces the results in the output cell wherever it is in the M-book.

See Also

For more information, see “Evaluating an Entire M-Book” on page 8-44.

Group Cells

Group Cells converts the input cells in the selection into a single multiline input cell called a cell group. You evaluate a cell group using **Evaluate Cell**. When you evaluate a cell group, all of its output follows the group and appears in a single output cell.

Result

If you include text in the selection, Notebook moves it after the cell group. However, if text precedes the first input cell in the group, the text will remain before the group.

If you include output cells in the selection, Notebook deletes them. If you select all or part of an output cell before selecting this feature, Notebook includes its input cell in the cell group.

If the first line in the cell group is an autoinit cell, the entire group acts as a sequence of autoinit cells. Otherwise, the group acts as a sequence of input cells. You can convert an entire cell group to an autoinit cell by using **Define AutoInit Cell**.

See Also

For more information, see "Defining Cell Groups for Notebook" on page 8-37. For information about converting a cell group to individual input cells, see the description of the "Ungroup Cells" on page 8-61.

Hide Cell Markers

Hide Cell Markers hides cell markers in the M-book.

When you select this feature, it changes to **Show Cell Markers**.

Note Notebook does not print cell markers whether you choose to hide them or show them on the screen.

Notebook Options

Notebook Options allows you to examine and modify display options for numeric and graphic output.

See Also

See "Printing and Formatting an M-Book" on page 8-47 for more information.

Purge Selected Output Cells

Purge Selected Output Cells deletes all output cells from the current selection.

See Also

For more information, see “Deleting Output Cells with Notebook” on page 8-46.

Toggle Graph Output for Cell

Toggle Graph Output for Cell suppresses or allows graphic output from an input cell.

If an input or autoinit cell generates figure output that you want to suppress, place the cursor in the input cell and choose this feature. The string `(no graph)` will be placed after the input cell to indicate that graph output for that cell will be suppressed.

To allow graphic output for that cell, place the cursor inside the input cell and choose **Toggle Graph Output for Cell** again. The `(no graph)` marker will be removed. This feature overrides the **Embed Figures in M-book** option, if that option is set in the **Notebook Options** dialog box.

See Also

See “Embedding Graphic Output in the M-Book” on page 8-49 and “Suppressing Graphic Output for Individual Input Cells in Notebook” on page 8-50 for more information.

Undefine Cells

Undefine Cells converts the selected cells to text. If no cells are selected but the cursor is in a cell, Notebook undefines that cell. Notebook removes the cell markers and reformats the cell according to the Normal style.

If you undefine an input cell, Notebook automatically undefines its output cell. However, if you undefine an output cell, Notebook does not undefine its input cell. If you undefine an output cell containing an embedded graphic, the graphic remains in the M-book but is no longer associated with an input cell.

See Also

For information about the Normal style, see “Modifying Styles in the M-Book Template” on page 8-47. For information about deleting output cells, see the description of the “Purge Selected Output Cells” on page 8-60.

Ungroup Cells

Ungroup Cells converts the current cell group into a sequence of individual input cells or autoinit cells. If the cell group is an input cell, Notebook converts the cell group to input cells. If the cell group is an autoinit cell, Notebook converts the cell group to autoinit cells. Notebook deletes the output cell for the cell group.

A cell group is the current cell group if

- The cursor is in the cell group.
- The cursor is at the end of a line that contains the closing cell marker for the cell group.
- The cursor is in the output cell for the cell group.
- The cell group is selected.

See Also

For information about creating cell groups, see the description of the “Defining Cell Groups for Notebook” on page 8-37.

Source Control Interface

The source control interface provides access to your source control system from MATLAB. Source control systems, also known as version control, revision control, configuration management, and file management systems, are platform dependent — the topics for the Windows platforms appear first, followed by the topics for the UNIX platforms.

Troubleshooting Source Control Problems on Windows (p. 9-24)	Solutions to some common source control problems.
Source Control Interface on UNIX (p. 9-26)	Overview of the ways you can use the source control interface on UNIX platforms.
Specifying the Source Control System on UNIX (p. 9-27)	Specify the source control system using MATLAB, list the currently selected source control system using the `cmopts` function, set a view and check out a directory with ClearCase.
Checking Files Into the Source Control System on UNIX (p. 9-30)	Check in files using the Current Directory browser, the Editor/Debugger, Simulink, Stateflow, or the `checkin` function.
Checking Files Out of the Source Control System on UNIX (p. 9-33)	Check files out using the Current Directory browser, the Editor/Debugger, Simulink, Stateflow, or the `checkout` function.
Undoing the Checkout on UNIX (p. 9-36)	Undoing a checkout using the Current Directory browser, the Editor/Debugger, Simulink, Stateflow, or the `undocheckout` function

Source Control Interface on Windows

If you use source control systems to manage your files, you can interface with the systems to perform source control actions from within MATLAB, Simulink, and Stateflow®. Use menu items in MATLAB, Simulink, or Stateflow, or run functions in the MATLAB Command Window to interface with your source control systems.

The source control interface on Windows works with any source control system that conforms to the Microsoft Common Source Control standard, Version 1.1. If your source control system does not conform to the standard, use a Microsoft Source Code Control API wrapper product for your source control system so that you can interface with it from MATLAB, Simulink, and Stateflow.

Perform most source control interface actions from the Current Directory browser. You can also perform many of these actions for a single file from the MATLAB Editor/Debugger, a Simulink model window, or a Stateflow chart window — for more information, see “Performing Source Control Actions from the Editor/Debugger, Simulink, or Stateflow on Windows” on page 9-23. Another way to access many of the source control actions is with the `verctrl` function.

Setting Up the Source Control Interface on Windows

In this section...

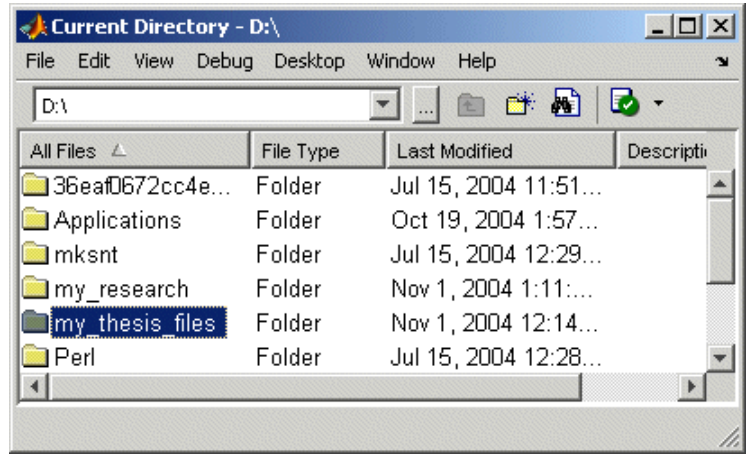
Create Projects in Source Control System

In your source control system, create the projects that your directories and files will be associated with.

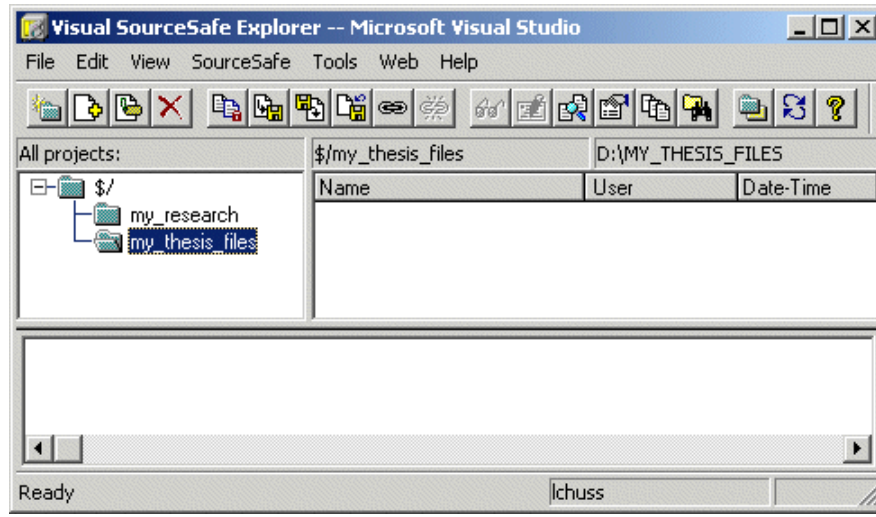
All files in a directory must belong to the same source control project. Be sure the working directory for the project in the source control system specifies the correct pathname to the directory on disk.

Example of Creating Source Control Project

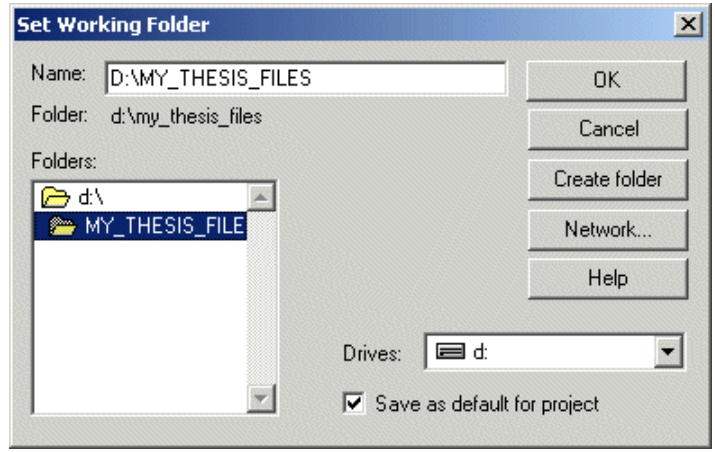
This example uses the project `my_thesis_files` in Microsoft Visual SourceSafe. This illustration of the Current Directory browser shows the pathname to the directory on disk, `D:\my_thesis_files`.



The following illustration shows the example project in the source control system.



To set the working directory in Microsoft Visual SourceSafe for this example, select `my_thesis_files`, right-click, select **Set Working Folder** from the context menu, and specify `D:\my_thesis_files` in the resulting dialog box.

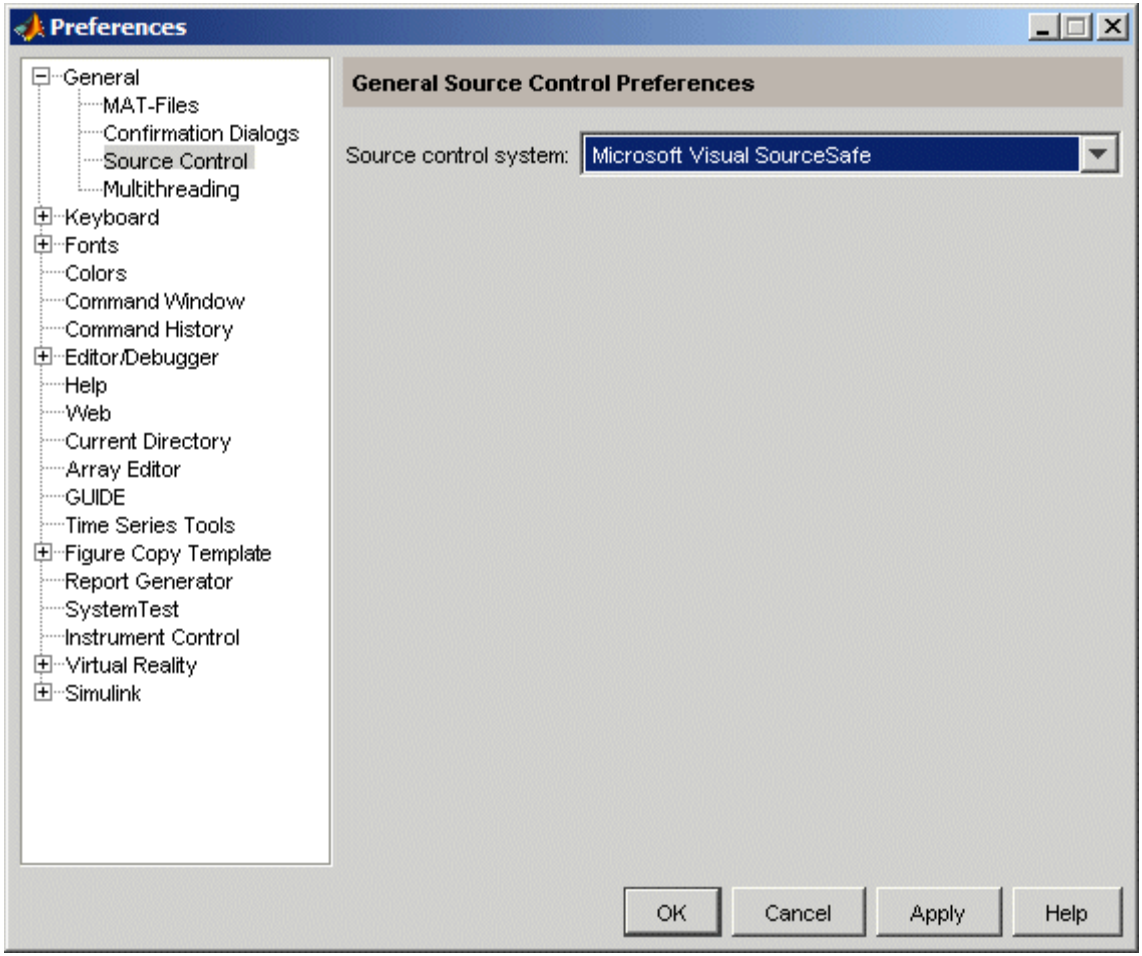


Specify Source Control System in MATLAB

In MATLAB, specify the source control system you want to access. Select **File > Preferences > General > Source Control**.

The currently selected system is shown in the Preferences dialog box. The list includes all installed source control systems that support the Microsoft Common Source Control standard.

Select the source control system you want to interface with and click **OK**.



MATLAB remembers preferences between sessions, so you only need to perform this action again when you want to access a different source control system.

Function Alternative

A function alternative to select a source control system is not available, but you can list all available source control systems using `verctrl` with the `all_systems` argument. Use `cmopts` to display the name of the currently selected source control system.

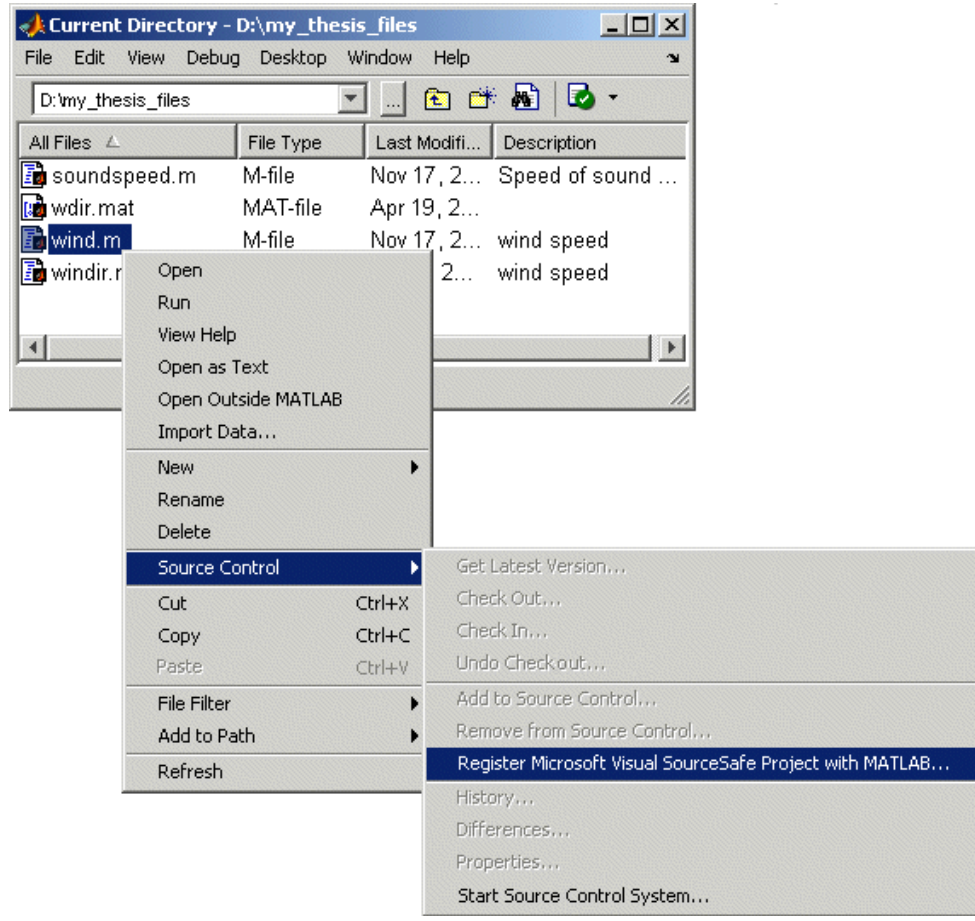
Register Source Control Project with MATLAB

Register a source control system project with a directory in MATLAB, that is, associate a source control system project with a directory and all files in that directory. Do this only one time for any file in the directory, which registers all files in that directory:

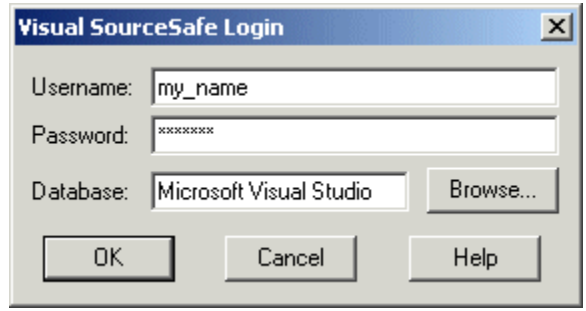
1. In the MATLAB Current Directory browser, select a file that is in the directory you want to associate with a project in your source control system. For example, select `D:\my_thesis_files\wind.m`. This will associate all files in the `my_thesis_files` directory.

2. Right-click, and from the context menu, select **Source Control > Register Name_of_Source_Control_System Project with MATLAB**. The **Name_of_Source_Control_System** is the source control system you selected using preferences as described in “Specify Source Control System in MATLAB” on page 9-6.

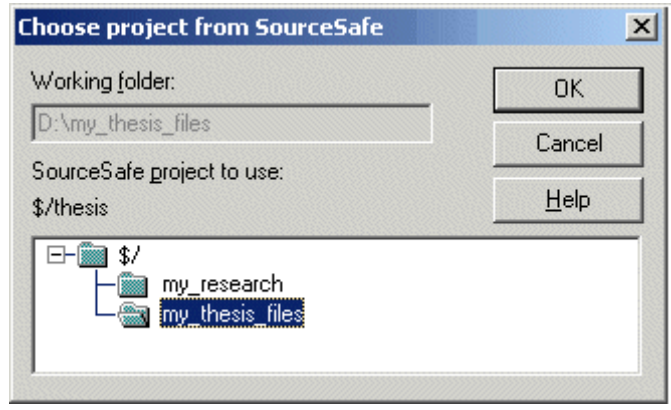
The following example shows Microsoft Visual SourceSafe.



3 In the resulting **Name_of_Source_Control_System Login** dialog box, provide the username and password you use to access your source control system, and click **OK**.



4 In the resulting **Choose project from Name_of_Source_Control_System** dialog box, select the source control system project to associate with the directory and click **OK**. This example shows `my_thesis_files`.



The selected file, its directory, and all files in the directory, are associated with the source control system project you selected. For the example, MATLAB associates all files in `D:\my_thesis_files` with the source control project `my_thesis_files`.

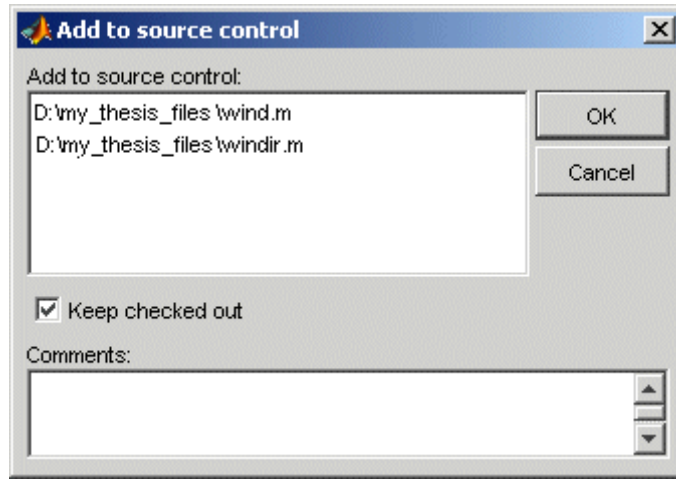
Add Files to Source Control

Add files to the source control system. Do this only once for each file:

1 In the Current Directory browser, select files you want to add to the source control system.

2 Right-click, and from the context menu, select **Source Control > Add to Source Control**.

3 The resulting **Add to source control** dialog box lists files you selected to add. You can add text in the **Comments** field. If you expect to use the files soon, select the **Keep checked out** check box (which is selected by default). Click **OK**.



If you try to add an unsaved file, the file is automatically saved upon adding.

Function Alternative

The function alternative is `verctrl` with the `add` argument.

Checking Files Into and Out of Source Control from MATLAB on Windows

In this section...

Before checking files into and out of your source control system from MATLAB, be sure to set up your system for use with MATLAB as described in “Setting Up the Source Control Interface on Windows” on page 9-4.

Check Files Into Source Control

After creating or modifying files in MATLAB or related products, check the files into the source control system by performing these steps:

1. In the Current Directory browser, select the files to check in. A file can be open or closed when you check it in, but it must be saved, that is, it cannot contain unsaved changes.
2. Right-click, and from the context menu, select **Source Control > Check In**.
3. In the resulting **Check in file(s)** dialog box, you can add text in the **Comments** field. If you want to continue working on the files, select the check box **Keep checked out**. Click **OK**.

If a file contains unsaved changes when you try to check it in, you will be prompted to save the changes to complete the checkin. If you did not keep the file checked out and you keep the file open, note that it is a read-only version.

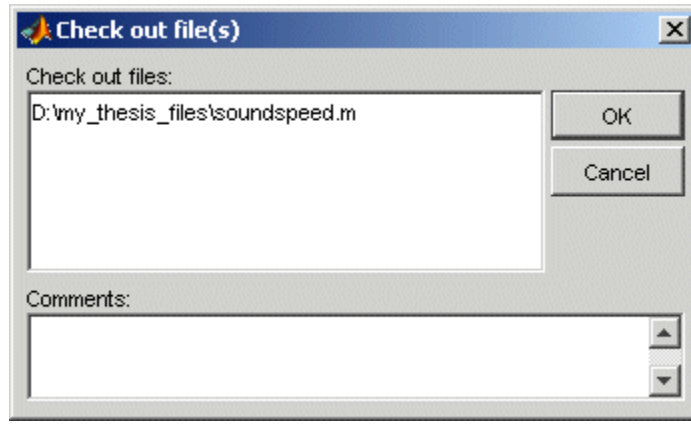
Function Alternative

The function alternative is `verctrl` with the `checkin` argument.

Check Files Out of Source Control

From MATLAB, to check out the files you want to modify, perform these steps:

1. In the Current Directory browser, select the files to check out.
2. Right-click, and from the context menu, select **Source Control > Check Out**.
3. The resulting **Check out file(s)** dialog box lists files you selected to check out. Enter comment text in the **Comments** field, which appears if your source control system supports comments on checkout. Click **OK**.



After checking out a file, make changes to it in MATLAB or another product, and save the file. For example, edit an M-file in the Editor/Debugger.

If you try to change a file without first having checked it out, the file is read-only, as seen in the title bar, and you will not be able to save any changes. This protects you from accidentally overwriting the source control version of the file.

If you end the MATLAB session, the file remains checked out. You can check in the file from within MATLAB during a later session, or directly from your source control system.

Function Alternative

The function alternative is `verctrl` with the `checkout` argument.

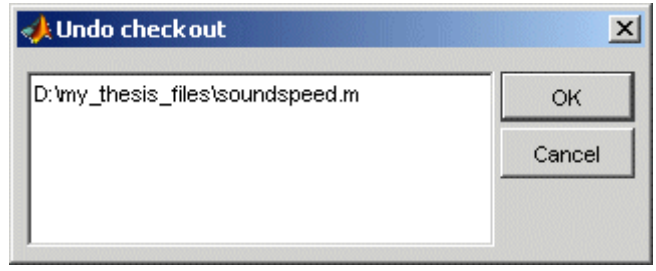
Undoing the Checkout

You can undo the checkout for files. The files remain checked in, and do not have any of the changes you made since you last checked them out. To save any changes you have made since checking out a particular file select **File > Save As**, and supply a different filename before you undo the checkout.

To undo a checkout, follow these steps:

1. In the MATLAB **Current Directory** browser, select the files for which you want to undo the checkout.
2. Right-click, and from the context menu, select **Source Control > Undo Checkout**.

   The MATLAB **Undo checkout** dialog box opens, listing the files you selected.



3. Click **OK**.

Function Alternative

The function alternative is `verctrl` with the `undocheckout` argument.

Additional Source Control Actions on Windows

In this section...

Getting the Latest Version of Files for Viewing or Compiling

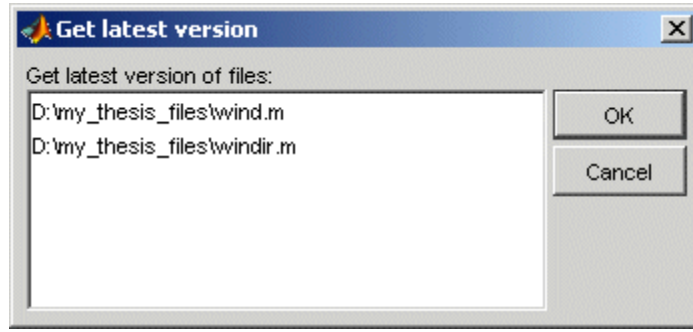
You can get the latest version of a file from the source control system for viewing or running. Getting a file differs from checking it out. When you get a file, it is write protected so you cannot edit it, but when you check out a file, you can edit it.

To get the latest version, follow these steps:

1 In the MATLAB **Current Directory** browser, select the directories or files that you want to get. If you select files, you cannot also select directories.

2 Right-click, and from the context menu, select **Source Control > Get Latest Version**.

The MATLAB **Get latest version** dialog box opens, listing the files or directories you selected.



3 Click **OK**.

You can now open the file to view it, run the file, or check out the file for editing.

Function Alternative

The function alternative is `verctrl` with the `get` argument.

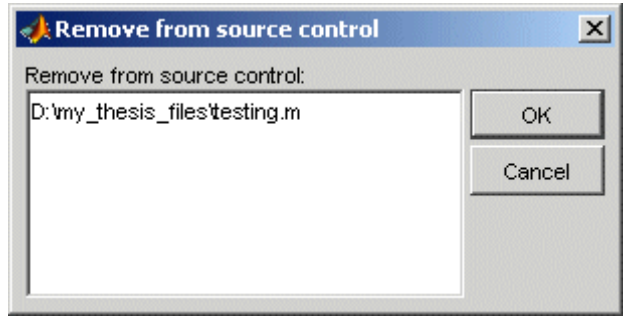
Removing Files from the Source Control System

To remove files from the source control system, follow these steps:

1 In the MATLAB **Current Directory** browser, select the files you want to remove.

2 Right-click, and from the context menu, select **Source Control > Remove from Source Control**.

The MATLAB **Remove from source control** dialog box opens, listing the files you selected.



3 Click **OK**.

Function Alternative

The function alternative is `verctrl` with the `remove` argument.

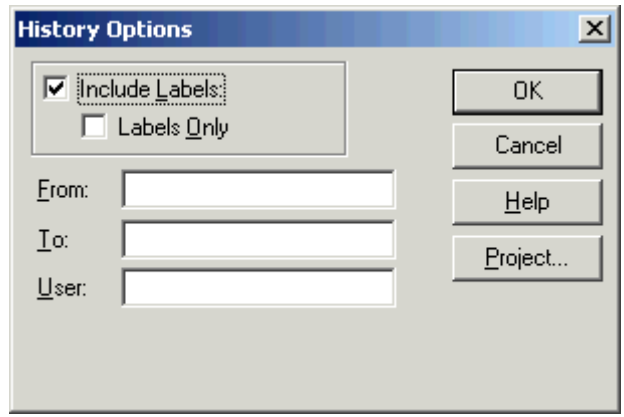
Showing File History

To show the history of a file in the source control system, follow these steps:

1 In the MATLAB **Current Directory** browser, select the file for which you want to view the history.

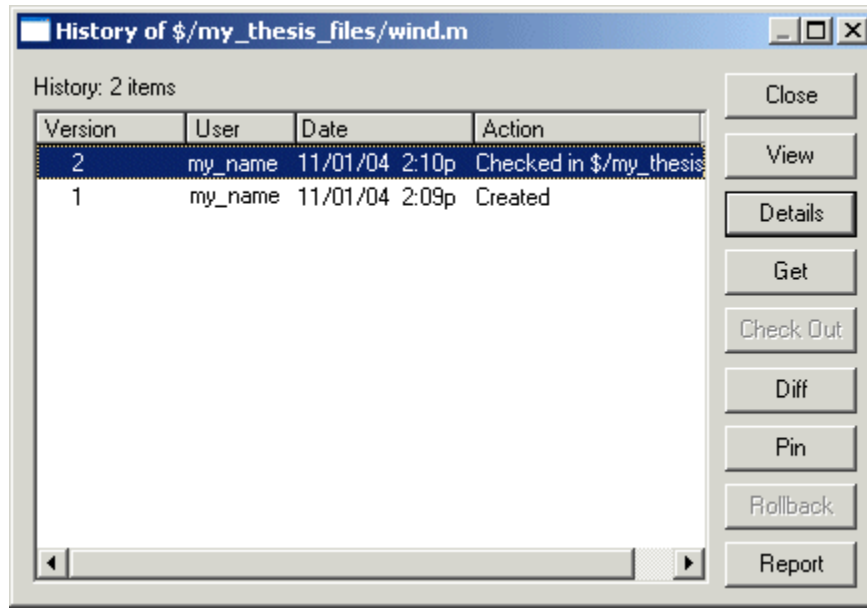
2 Right-click, and from the context menu, select **Source Control > History**.

A dialog box, which is specific to your source control system, opens. For Microsoft Visual SourceSafe, the **History Options** dialog box opens, as shown in the following example illustration.



3 Complete the dialog box to specify the range of history you want for the selected file and click **OK**. For example, enter `my_name` for **User**.

The history presented depends on your source control system. For Microsoft Visual SourceSafe, the **History** dialog box opens for that file, showing the file's history in the source control system.



Function Alternative

The function alternative is `verctrl` with the `history` argument.

Comparing the Working Copy of a File to the Latest Version in Source Control

You can compare the current working copy of a file with the latest checked-in version of the file in the source control system. This highlights the differences between the two files, showing the changes you made since you checked out the file.

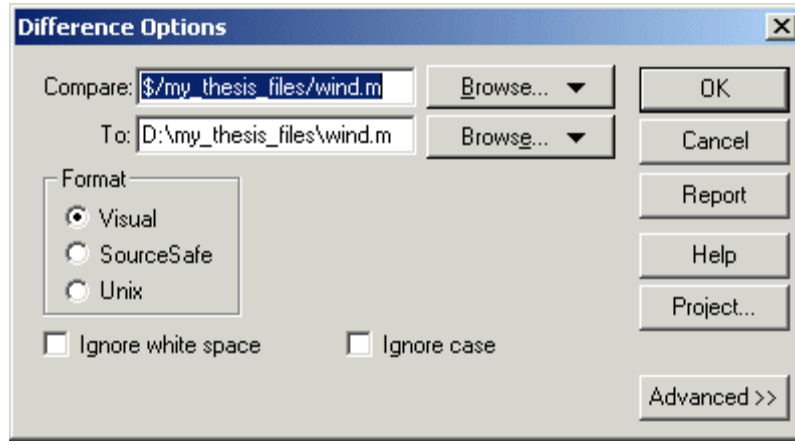
To view the differences, follow these steps:

1. In the MATLAB **Current Directory** browser, select the file for which you want to view differences. This is a file that has been checked out and edited.

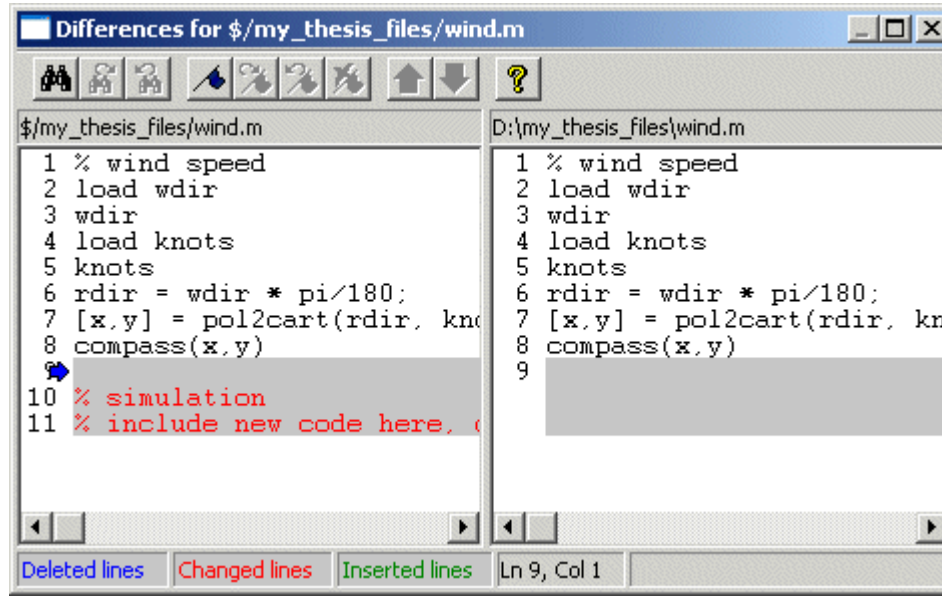
2 Right-click, and from the context menu, select **Source Control > Differences**.

A dialog box, which is specific to your source control system, opens. For Microsoft Visual SourceSafe, the **Difference Options** dialog box opens.

3 Review the default entries in the dialog box, make any needed changes, and click **OK**. The following example is for Microsoft Visual Source Safe.



The method of presenting differences depends on your source control system. For Microsoft Visual SourceSafe, the **Differences for** dialog box opens. This highlights the differences between the working copy of the file and the latest checked-in version of the file.



Function Alternative

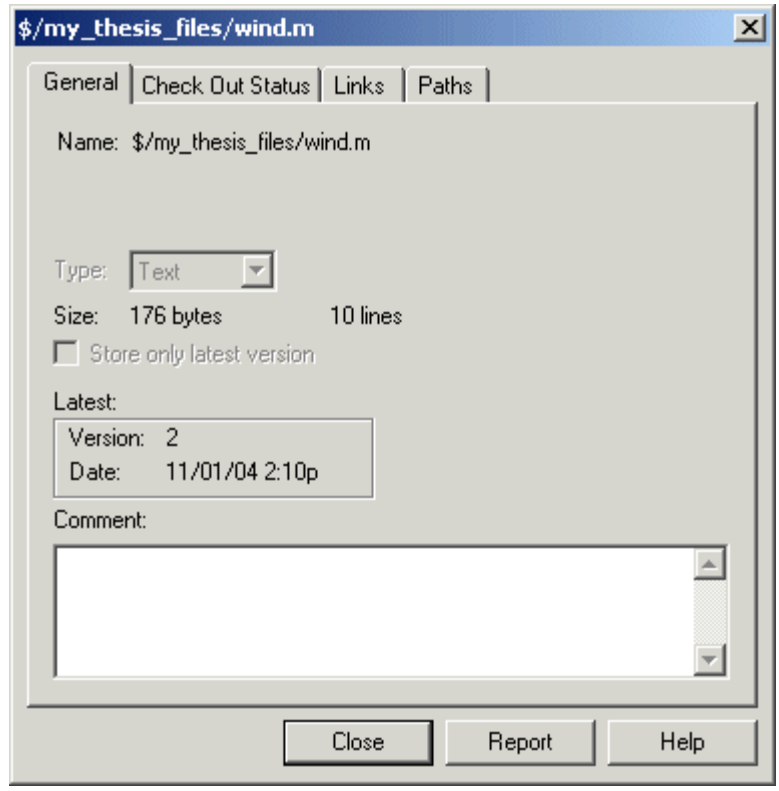
The function alternative is `verctrl` with the `showdiff` or `isdiff` argument.

Viewing Source Control Properties of a File

To view the source control properties of a file, follow these steps:

1. In the MATLAB **Current Directory** browser, select the file for which you want to view properties.
2. Right-click, and from the context menu, select **Source Control > Properties**.

A dialog box, which is specific to your source control system, opens. The following example shows the Microsoft Visual SourceSafe properties dialog box.



Function Alternative

The function alternative is `verctrl` with the `properties` argument.

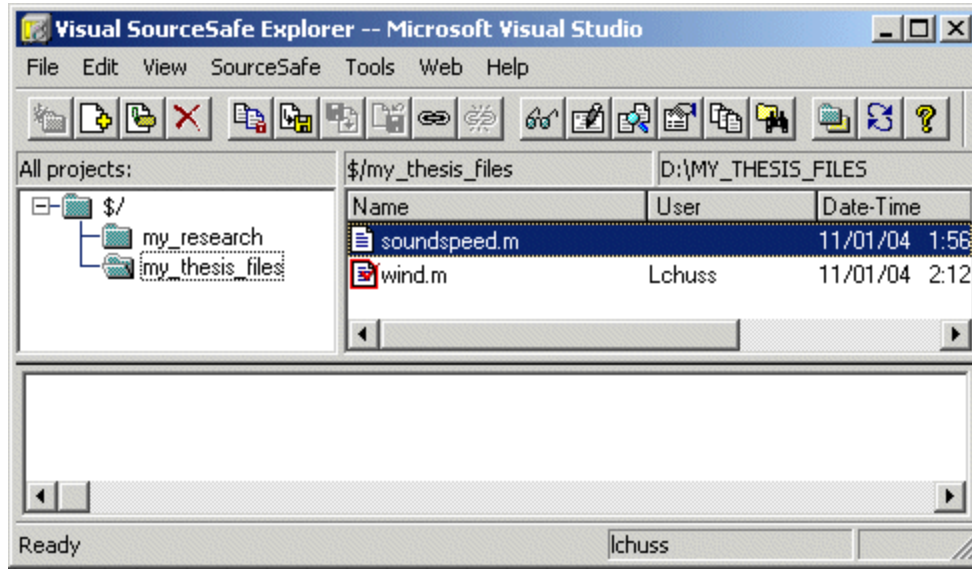
Starting the Source Control System

All the MATLAB source control actions automatically start the source control system to perform the action, if the source control system is not already open. If you want to start the source control system from MATLAB without performing a specific action source control action,

1 Right-click any directory or file in the MATLAB Current Directory browser

2 From the context menu, select **Source Control > Start Source Control System**.

The interface to your source control system opens, showing the source control project associated with the current directory in MATLAB. The following example shows the Microsoft **Visual SourceSafe Explorer** interface.



Function Alternative

The function alternative is `verctrl` with the `runscc` argument.

Performing Source Control Actions from the Editor/Debugger, Simulink, or Stateflow on Windows

You can create or open a file in the Editor/Debugger, Simulink, or Stateflow and perform most source control actions from their **File > Source Control** menus, rather than from the Current Directory browser as described in previous sections. Following are some differences in the source control interface process when you use the Editor/Debugger, Simulink, or Stateflow:

- You can perform actions on only one file at time.
- Some of the dialog boxes have a different icon in the title bar. For example, the **Check out file(s)** dialog box uses an M-file Editor/Debugger document icon instead of the MATLAB icon.
- You cannot add a new (`Untitled`) file, but must instead first save the file.
- You cannot register projects from Simulink or Stateflow. Instead, register a project using the Current Directory browser, as described in “Register Source Control Project with MATLAB” on page 9-7.

Troubleshooting Source Control Problems on Windows

In this section...

Source Control Error: Provider Not Present or Not Installed Properly

In some cases, MATLAB recognizes your source control system but you cannot use source control features for MATLAB. Specifically, when you select **File > Preferences > General > Source Control**, or run `cmopts`, MATLAB lists your source control system, but you cannot perform any source control actions. Only the **File > Source Control > Start Source Control System** menu item is available, and when you select it, MATLAB displays this error:

```
Source control provider is not present or not installed properly.
```

Often, this error occurs because a registry key that MATLAB requires from the source control application is not present. Make sure this registry key is present:

```
HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider\
InstalledSCCProviders
```

The registry key refers to another registry key that is similar to

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\SourceSafe\SccServerPath
```

This registry key has a path to a DLL-file in the file system. Make sure the DLL-file exists in that location. If you are not familiar with registry keys, ask your system administrator for help.

If this does not solve the problem and you use Microsoft Visual SourceSafe, try running a client setup for your source control application. When SourceSafe is

installed on a server for a group to use, each machine client can run a setup but is not required to do so. However, some applications that interface with SourceSafe, including MATLAB, require you to run the client setup. Run the client setup, which should resolve the problem.

If the problem persists, access source control outside of MATLAB.

Restriction Against @ Character

Some source control systems, such as Perforce and Synergy, reserve the `@` character. Perforce, for example, uses it as a revision specifier. Therefore, you might experience problems if you use these source control systems with MATLAB files and directories that include the `@` character in the directory or filename.

You might be able to work around this restriction by quoting nonstandard characters in filenames, such as with an escape sequence, which some source control systems allow. Consult your source control system documentation or technical support resources for a workaround.

Add to Source Control Is the Only Action Available

To use source control features for a file in Simulink or Stateflow, the file's source control project must first be registered with MATLAB. When a file's source control project is *not* registered with MATLAB, all **File > Source Control** menu items are disabled except **Add to Source Control**. You can select **Add to Source**, which registers the project with MATLAB, or you can register the project using the Current Directory browser, as described in "Register Source Control Project with MATLAB" on page 9-7. You can then perform source control actions for all files in that project (directory).

More Solutions for Source Control Problems

The latest solutions for problems interfacing MATLAB with a source control system appear on the MathWorks Web page for support at `http://www.mathworks.com/support/`. Search Solutions and Technical Notes for "source control."

Source Control Interface on UNIX

If you use a source control system to manage your files, you can check M-files and Simulink models, and Stateflow charts into and out of the source control system from within MATLAB, Simulink, and Stateflow.

The source control interface supports four popular source control systems, as well as a custom option:

- ClearCase from IBM Rational
- Concurrent Version System (CVS)
- ChangeMan from Serena (also used for PVCS from Merant)
- Revision Control System (RCS)
- Custom option — Allows you to build your own interface if you use a different source control system. For details, see the reference page for `customverctrl`.

Perform source control interface actions for a single file using menu items in the MATLAB Editor/Debugger, a Simulink model window, or a Stateflow chart window. To perform source control actions on multiple files, use the Current Directory browser. Alternatively, run source control functions in the Command Window, which provide some options not supported with the menu items.

Specifying the Source Control System on UNIX

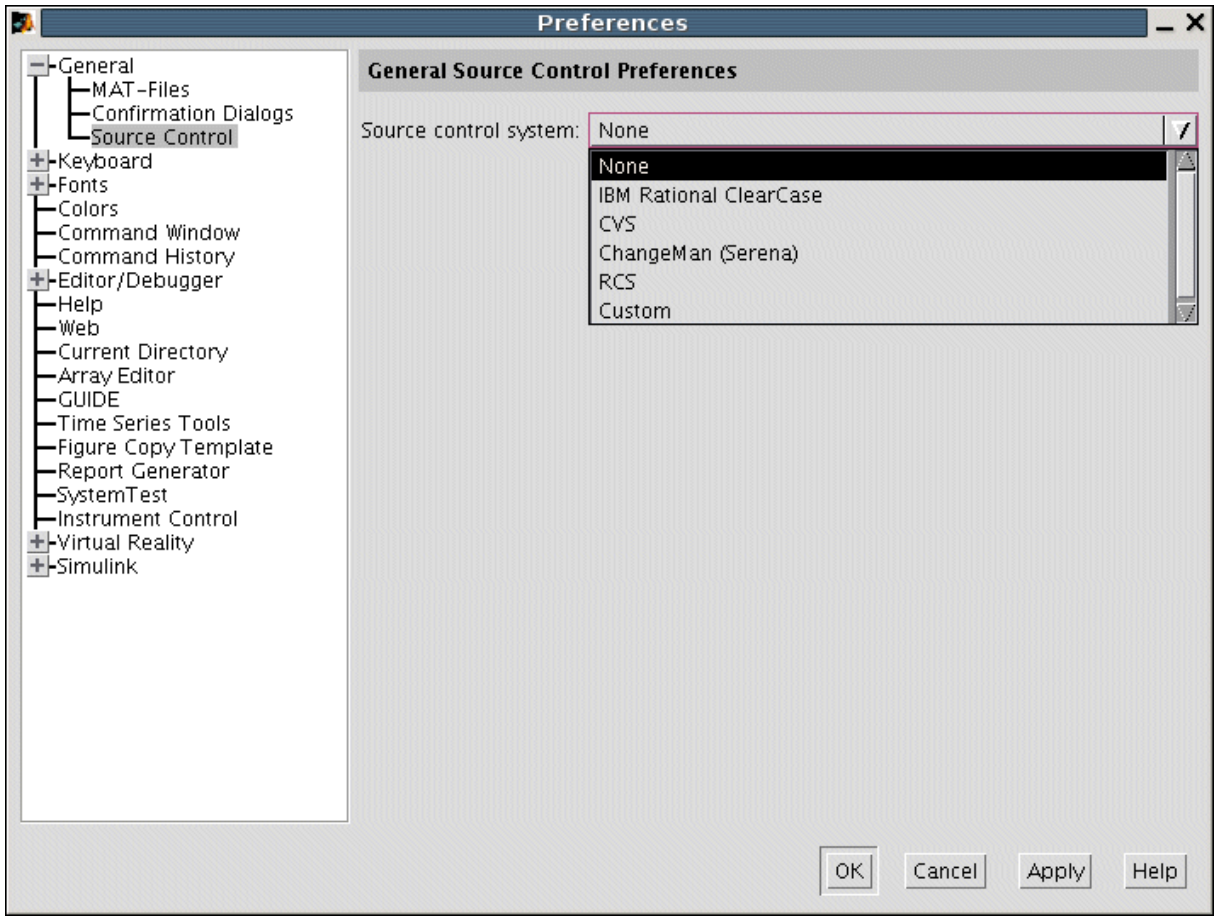
In this section...

MATLAB Alternative

In MATLAB, specify the source control system you want to access. Select **File > Preferences > General > Source Control**.

The currently selected system is shown in the Preferences dialog box. The default selection is `None`.

Select the source control system you want to interface with and click **OK**.



MATLAB remembers preferences between sessions, so you only need to perform this action when you want to access a different source control system.

Function Alternative

A function alternative to select a source control system is not available, but you can list the currently selected source control system by running `cmopts`.

Setting a View and Checking Out a Directory with ClearCase on UNIX

If you use ClearCase on a UNIX platform, perform the following from ClearCase:

1. Set a view.
2. Check out the directory that contains files you want to save, check in, or check out.

You can now use the MATLAB, Simulink, or Stateflow source control interfaces to ClearCase.

Checking Files Into the Source Control System on UNIX

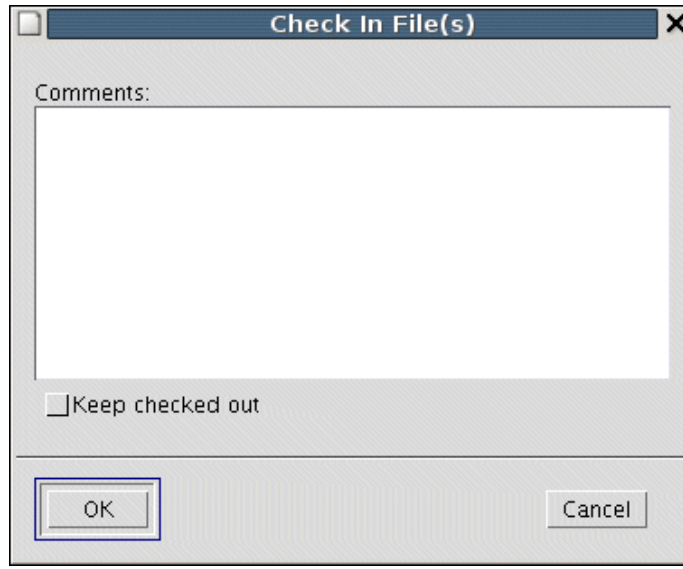
In this section...

Checking In One or More Files Using the Current Directory Browser

1. From the Current Directory browser, select the file or files to check in. A file can be open or closed when you check it in, but it must be saved, that is, it cannot contain unsaved changes.

2. Right-click, and from the context menu, select **Source Control > Check In**.

3. In the resulting **Check in file(s)** dialog box, you can add text in the **Comments** field. If you want to continue working on the files, select the check box **Keep checked out**. Click **OK**.



The files are checked into the source control system. If any file contains unsaved changes when you try to check it in, you will be prompted to and must then save the changes to complete the checkin.

An error appears in the Command Window if a file is already checked in.

If you did not keep a file checked out and you keep that file open, note that it is a read-only version.

Checking In One File Using the Editor/Debugger, Simulink, or Stateflow

1. From the Editor/Debugger, Simulink, or Stateflow, with the file open and saved, select **File > Source Control > Check In**.

2. In the resulting **Check in file(s)** dialog box, you can add text in the **Comments** field. If you want to continue working on the files, select the check box **Keep checked out**. Click **OK**.

Function Alternative

Use `checkin` to check files into the source control system. The files can be open or closed when you use `checkin`. The `checkin` function takes this form:

```
checkin({'file1','file2', ...},'comments','comment_text',...
'option','value')
```

For `filen`, use the complete path and include the file extension. You must supply the `comments` argument and a comments string with `checkin`.

Use the `option` argument to

- Check in a file and keep it checked out — set the `lock` option value to `on`.
- Check in a file even though it has not changed since the previous check in — set the `force` option value to `on`.

The `comments` argument and the `lock` and `force` options apply to all files checked in.

Example Using checkin Function

To check in the file `clock.m` with the comment `Adjustment for leap year`, type

```
checkin('\myserver\mymfiles\clock.m','comments', ...
'Adjustment for leap year')
```

For other examples, see the reference page for `checkin`.

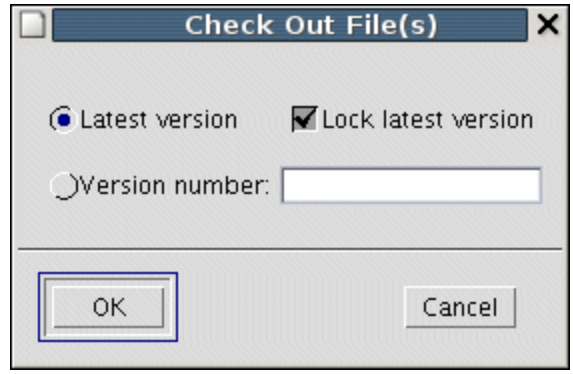
Checking Files Out of the Source Control System on UNIX

In this section...

Checking Out One or More Files Using the Current Directory Browser

1 In the Current Directory browser, select the file or files to check out.

2 Right-click, and from the context menu, select **Source Control > Check Out**. The **Check out file(s)** dialog box opens.



3 Complete the dialog box:

a To check out the versions that were most recently checked in, select the **Latest version** option.

b To check out a specific version of the files, select the **Version number** option and type the version number in the field.

c To prevent others from checking out the files while you have them checked out, select **Lock latest version**. To check out read-only versions of the file, clear **Lock latest version**.

4 Click **OK**.

An error appears in the Command Window if a file is already checked out.

After checking out files, make changes to them in MATLAB or another product, and save the files. For example, edit an M-file in the Editor/Debugger.

If you try to change a file without first having checked it out, the file is read-only, as seen in the title bar, and you will not be able to save any changes. This protects you from accidentally overwriting the source control version of the file.

If you end the MATLAB session, the file or files remain checked out. You can check in files from within MATLAB during a later session, or directly from your source control system.

Checking Out a Single File Using the Editor/Debugger, Simulink, or Stateflow

1 Open the M-file, Simulink model, or Stateflow chart you want to check out. The title bar indicates the file is read-only.

2 Select **File > Source Control > Check Out**. The **Check out file(s)** dialog box opens.

3 Complete the dialog box as described in step of "Checking Out One or More Files Using the Current Directory Browser" on page 9-33, and click **OK**.

Function Alternative

Use `checkout` to check out a file from the source control system. You can check out multiple files at once and specify checkout options. The `checkout` function takes this form:

```
checkout({'file1','file2', ...},'option','value')
```

For `filen`, use the complete path and include the file extension.

Use the `option` argument to

- Check out a read-only version of the file — set the `lock` option value to `off`.
- Check out the file even if you already have it checked out — set the `force` option value to `on`.
- Check out a specific version of the file — use the `revision` option, and assign the version number to the `value` argument.

The options apply to all files being checked out. The files can be open or closed when you use `checkout`.

Example Using checkout Function—Check Out a Specific Version of a File

To check out the 1.1 version of the file `clock.m`, type

```
checkout('\myserver\mymfiles\clock.m','revision','1.1')
```

For other examples, see the reference page for `checkout`.

Undoing the Checkout on UNIX

In this section...

Impact of Undoing a File Checkout

When you undo the checkout for a file, the file remains checked in, and does not have any of the changes you made since you checked it out. To save any changes you have made since checking out a file, select **File > Save As**, and supply a different filename before you undo the checkout. Undo the checkout using the Current Directory browser for one or more files. For only one file, you can also use the Editor/Debugger, Simulink, or Stateflow.

Undoing the Checkout for One or More Files Using the Current Directory Browser

1. In the MATLAB **Current Directory** browser, select the file or files for which you want to undo the checkout.
2. Right-click, and from the context menu, select **Source Control > Undo Checkout**. MATLAB undoes the checkout.

An error appears in the Command Window if the file is not checked out.

Undoing the Checkout for a Single File Using the Editor/Debugger, Simulink, or Stateflow

1. Open the M-file, Simulink model, or Stateflow chart for which you want to undo the checkout.

2 Select **File > Source Control > Undo Checkout**. MATLAB undoes the checkout.

Function Alternative

The `undocheckout` function takes this form:

```
undocheckout({'file1','file2', ...})
```

Use the complete path for `filen` and include the file extension. For example, to undo the checkout for the files `clock.m` and `calendar.m`, type

```
undocheckout({'\myserver\mymfiles\clock.m',...
'\myserver\mymfiles\calendar.m'})
```

Index

C

D

E

F

G

H

I

J

K

L

M

N

Q

R

S

T

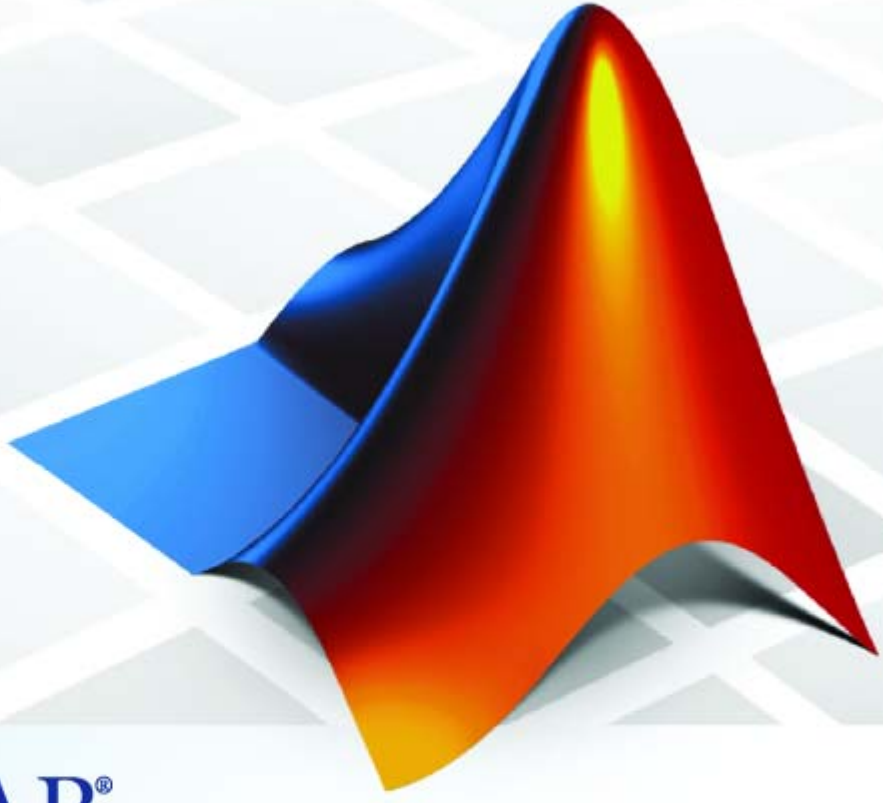
U

V

W

Y

MATLAB® 7
External Interfaces



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB External Interfaces

Revision History

December 1996	First printing	New for MATLAB 5 (release 8)
July 1997	Online only	Revised for MATLAB 5.1 (Release 9)
January 1998	Second printing	Revised for MATLAB 5.2 (Release 10)
October 1998	Third printing	Revised for MATLAB 5.3 (Release 11)
November 2000	Fourth printing	Revised and renamed for MATLAB 6.0 (Release 12)
June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	Online only	Revised for MATLAB 6.5 (Release 13)
January 2003	Online only	Revised for MATLAB 6.5.1 (Release 13SP1)
June 2004	Online only	Revised for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b)

Contents

1 Importing and Exporting Data

2 MATLAB Interface to Generic DLLs

3 Calling C and Fortran Programs from MATLAB

4 Creating C Language MEX-Files

5 Creating Fortran MEX-Files

6 Calling MATLAB from C and Fortran Programs

7 Calling Java from MATLAB

8 COM Support in MATLAB (Windows Only)

9 Web Services in MATLAB

10 Serial Port I/O

Examples

A

Index

1

Importing and Exporting Data

Using MAT-Files

In this section...

Introduction

MAT-files, the data file format MATLAB uses for saving data to your disk, provide a convenient mechanism for moving MATLAB data between platforms and for importing and exporting data to stand-alone MATLAB applications.

To simplify your use of MAT-files in applications outside of MATLAB, we have developed a library of access routines with a `mat` prefix that you can use in your own C or Fortran programs to read and write MAT-files. Programs that access MAT-files also use the `mx` prefixed API (application program interface) routines discussed in Chapter 4, "Creating C Language MEX-Files" and Chapter 5, "Creating Fortran MEX-Files".

Importing Data to MATLAB

The best method for importing data into MATLAB depends on how much data there is, whether the data is already in machine-readable form, and what format the data is in. Here are some choices; select the one that best meets your needs.

- **Enter the data at the MATLAB command prompt.**

  For small amounts of data, less than 10-15 elements, type the data directly into MATLAB using brackets [ ]. This method is awkward for large amounts of data because you can't edit your input.

- **Create data in an M-file.**

  With a text editor, create an M-file to enter data as an explicit list of elements. This method is useful when the data isn't already in computer-readable format and must be typed in. Use the editor to change the data or correct mistakes, then rerun the M-file to reenter the data.

- **Load data from an ASCII flat file.**

  A *flat file* stores data in ASCII format, with fixed-length rows terminated by new lines (carriage returns) and with spaces separating the numbers. Edit ASCII flat files using a text editor and use the `load` command to read them directly into MATLAB. MATLAB creates a variable with the same name as the filename.

- **Read data using MATLAB I/O functions.**

  Use `fopen`, `fread`, and other low-level MATLAB I/O functions to read data. This method allows you to load data files from applications that have their own file formats.

- **Write a MEX-file to read the data.**

  This is the method of choice if subroutines are available for reading data files from other applications. See the section, "Introducing MEX-Files" on page 3-2, for more information.

- **Write a program to translate your data.**

  Write programs in C or Fortran to translate your data into MAT-file format. Use the `load` command to read the MAT-file into MATLAB. Refer to the section, "Reading and Writing MAT-Files" on page 1-5, for more information.

Exporting Data from MATLAB

There are several methods for exporting MATLAB data.

- **Create a diary file.**

  For small matrices, use the `diary` command to create a diary file, a log of keyboard input and the resulting text output. You can use a text editor to modify the file. The diary file displays the variables and includes the MATLAB commands used during the session, which can be used in documents and reports.

- **Use the save command.**

  Save data in ASCII format using the `save` command with the `-ascii` option. For example,

  ```
  A = rand(4,3);
  save temp.dat A -ascii
  ```

  creates an ASCII file called `temp.dat`, which may look something like this:

  ```
  1.3889088e-001  2.7218792e-001  4.4509643e-001
  2.0276522e-001  1.9881427e-001  9.3181458e-001
  1.9872174e-001  1.5273927e-002  4.6599434e-001
  6.0379248e-001  7.4678568e-001  4.1864947e-001
  ```

  The `-ascii` option supports two-dimensional double and character arrays only. Multidimensional arrays, cell arrays, and structures are not supported.

- **Use MATLAB I/O functions.**

  Write the data in a special format using `fopen`, `fwrite`, and other low-level file I/O functions. This method is useful for writing data files in file formats required by other applications. See the section, "Using Low-Level File I/O Functions", for more information.

- **Create a MEX-file to write the data.**

  This is the method of choice if subroutines are available for writing data files in the form needed by other applications. See the section, "Introducing MEX-Files" on page 3-2, for more information.

- **Translate data from a MAT-file.**

  Write data into a MAT-file using the `save` command, then write a program in C or Fortran to translate the MAT-file into your own special format. See the section, "Reading and Writing MAT-Files" on page 1-5, for more information.

Exchanging Data Files Between Platforms

You can work with MATLAB on different computer systems and send MATLAB applications to users on other systems. MATLAB applications consist of M-files containing functions and scripts, and MAT-files containing binary data.

Both types of files can be transported directly between machines: M-files because they are platform independent and MAT-files because they contain a machine signature in the file header. MATLAB checks the signature when it loads a file and, if a signature indicates that a file is foreign, performs the necessary conversion.

Using MATLAB across different machine architectures requires a facility for exchanging both binary and ASCII data between the machines. Examples of this type of facility include FTP, NFS, and Kermit. When using these programs, be careful to transmit MAT-files in *binary file mode* and M-files in *ASCII file mode*. Failure to set these modes correctly corrupts the data.

Reading and Writing MAT-Files

Use the MATLAB `save` command to save MATLAB arrays currently in memory to a binary file called a MAT-file. MAT-files have the extension `.mat`. The `load` command reads MATLAB arrays from a MAT-file on disk back into the MATLAB workspace.

A MAT-file contains one or more of the data types supported in MATLAB 5 or later, including strings, matrices, multidimensional arrays, structures, and cell arrays. MATLAB writes the data sequentially onto disk in a continuous byte stream.

MAT-File Interface Library

The MAT-file interface library contains routines for reading and writing MAT-files. You can call these routines from your own C and Fortran programs. Use these routines, rather than attempt to write your own code, to perform these operations, since using the library insulates your applications from future changes to the MAT-file structure.

Functions in the MAT-file library begin with the three-letter prefix `mat`. These tables list and describe the MAT-functions.

C MAT-File Routines

MAT-Function	Purpose
`matOpen`	Open a MAT-file.
`matClose`	Close a MAT-file.
`matGetDir`	Get a list of MATLAB arrays from a MAT-file.
`matGetFp`	Get an ANSI C file pointer to a MAT-file.
`matGetVariable`	Read a MATLAB array from a MAT-file.
`matPutVariable`	Write a MATLAB array to a MAT-file.
`matGetNextVariable`	Read the next MATLAB array from a MAT-file.
`matDeleteVariable`	Remove a MATLAB array from a MAT-file.
`matPutVariableAsGlobal`	Put a MATLAB array into a MAT-file such that the `load` command places it into the global workspace.
`matGetVariableInfo`	Load a MATLAB array header from a MAT-file (no data).
`matGetNextVariableInfo`	Load the next MATLAB array header from a MAT-file (no data).

Fortran MAT-File Routines

MAT-Function	Purpose
`matOpen`	Open a MAT-file.
`matClose`	Close a MAT-file.
`matGetDir`	Get a list of MATLAB arrays from a MAT-file.
`matGetVariable`	Get a named MATLAB array from a MAT-file.
`matGetVariableInfo`	Get header for named MATLAB array from a MAT-file.

Fortran MAT-File Routines (Continued)

MAT-Function	Purpose
`matPutVariable`	Put a MATLAB array into a MAT-file.
`matPutVariableAsGlobal`	Put a MATLAB array into a MAT-file.
`matGetNextVariable`	Get the next sequential MATLAB array from a MAT-file.
`matGetNextVariableInfo`	Get header for next sequential MATLAB array from a MAT-file.
`matDeleteVariable`	Remove a MATLAB array from a MAT-file.

Writing Character Data

By default, MATLAB writes character data to MAT-files using Unicode character encoding. To override this setting and use your system's default encoding instead, do one of the following:

- From the MATLAB command line or a MATLAB function, save your data to the MAT-file using the command `save -nounicode`.
- From a C mex file, open the MAT-file you will write the data to using the command `matOpen -wL`.

See the individual reference pages for these functions for more information.

You can also set a save preference for all MATLAB sessions. See MAT-Files Preferences in the "General Preferences for MATLAB" section of the Desktop Tools and Development Environment documentation for more information.

ASCII Data Formats

When writing character data using Unicode character encoding (the default), MATLAB checks if the data is 7-bit ASCII. If it is, MATLAB writes the 7-bit ASCII character data to the MAT-file using 8 bits per character (UTF-8 format), thus minimizing the size of the resulting file. Any character data that is not 7-bit ASCII is written in 16-bit Unicode form (UTF-16). This algorithm operates on a per-string basis.

Note Level 4 MAT-files support only ASCII character data. If you have non-ASCII character data, writing a Level 4 MAT-file is not supported. In the event that a Level 4 MAT-file is created with such character data, it is unlikely that the original representation of the characters will be preserved.

Converting Character Data

Writing character data to MAT-files using Unicode character encoding enables you to share data with users that have systems with a different default system character encoding scheme, without character data loss or corruption. Although conversion between Unicode and other encoding forms is often lossless, there are scenarios in which round-trip conversions can result in loss of data. The following guidelines may reduce your chances of data loss or corruption.

In order to prevent loss or corruption of character data, all users sharing the data must have at least one of the following in common:

- They exchange Unicode-based MAT-files, and use a version of MATLAB that supports these files.
- Their computer systems all use the same default encoding, and all character data in the MAT-file was written using the `-nounicode` option

For example, if one user on a Japanese language operating system writes ASCII data having more than 7 bits per character to a MAT-file, another user running MATLAB version 6.5 on an English language operating system will be unable to read the data accurately. However, if both have MATLAB version 7, the information can be shared without corruption or loss of data.

Finding Associated Files

A collection of files associated with reading and writing MAT-files is located on your disk. The following table, MAT-Function Subdirectories, lists the path to the required subdirectories for importing and exporting data using MAT-functions. The term *`matlabroot`* refers to the root directory of your MATLAB installation.

MAT-Function Subdirectories

Platform	Contents	Directories
Windows	Include files	*matlabroot*\extern\include
	Libraries	*matlabroot*\bin\win32 or *matlabroot*\bin\win64
	Examples	*matlabroot*\extern\examples\eng_mat
UNIX	Include files	*matlabroot*/extern/include
	Libraries	*matlabroot*/bin/$arch
	Examples	*matlabroot*/extern/examples/eng_mat

Include Files

The `include` subdirectory holds header files containing function declarations with prototypes for the routines that you can access in the API Library. These files are the same for both Windows and UNIX. Included in the subdirectory are

- The `matrix.h` header file that defines MATLAB array access and creation methods
- The `mat.h` header file that defines MAT-file access and creation methods

Libraries

The libraries subdirectory, that contains shared (dynamically linkable) libraries for linking your programs, is platform dependent.

Shared Libraries on Windows. The `bin` subdirectory contains the shared libraries for linking your programs:

- The `libmat.dll` library of MAT-file routines (C and Fortran)
- The `libmx.dll` library of array access and creation routines

Shared Libraries on UNIX. The `bin/$arch` subdirectory, where `$arch` is your machine's architecture, contains the shared libraries for linking your programs. For example, on Solaris, the subdirectory is `bin/sol64`:

- The `libmat.so` library of MAT-file routines (C and Fortran)
- The `libmx.so` library of array access and creation routines

Example Files

The `examples/eng_mat` subdirectory contains C and Fortran source code for a number of example files that demonstrate how to use the MAT-file routines. The source code files are the same for both Windows and UNIX.

C and Fortran Examples

Library	Description
`matcreat.c`	C program that demonstrates how to use the library routines to create a MAT-file that can be loaded into MATLAB
`matdgns.c`	C program that demonstrates how to use the library routines to read and diagnose a MAT-file
`matdemo1.F`	Fortran program that demonstrates how to call the MATLAB MAT-file functions from a Fortran program
`matdemo2.F`	Fortran program that demonstrates how to use the library routines to read the MAT-file created by `matdemo1.F` and describe its contents

For more information about MATLAB API files and directories, see "Additional Information" on page 3-43.

Examples of MAT-Files

In this section...

Creating a MAT-File in C

This program, `matcreat.c`, illustrates how to use the library routines to create a MAT-file that can be loaded into MATLAB. The program also demonstrates how to check the return values of MAT-function calls for read or write failures. To see the code, you can open the file in MATLAB Editor.

To produce an executable version of this program, compile the file and link it with the appropriate library. Details of how to compile and link MAT-file programs on various platforms are discussed in the section, "Compiling and Linking MAT-File Programs" on page 1-15.

Once you have compiled and linked your MAT-file program, you can run the stand-alone application you have just produced. This program creates `mattest.mat`, a MAT-file that can be loaded into MATLAB. To run the application, depending on your platform, either double-click its icon or enter `matcreat` at the system prompt:

```
matcreat
Creating file mattest.mat...
```

To verify the MAT-file was created, at the MATLAB prompt type

```
whos -file mattest.mat
  Name               Size         Bytes  Class

  GlobalDouble       3x3             72  double array (global)
  LocalDouble        3x3             72  double array
  LocalString        1x43            86  char array
```

```
Grand total is 61 elements using 230 bytes
```

Reading a MAT-File in C

This program, `matdgns.c`, illustrates how to use the library routines to read and diagnose a MAT-file. To see the code, you can open the file in MATLAB Editor.

After compiling and linking this program, you can view its results:

```
matdgns mattest.mat
Reading file mattest.mat...

Directory of mattest.mat:
GlobalDouble
LocalString
LocalDouble

Examining the header for each variable:
According to its header, array GlobalDouble has 2 dimensions
  and was a global variable when saved
According to its header, array LocalString has 2 dimensions
  and was a local variable when saved
According to its header, array LocalDouble has 2 dimensions
  and was a local variable when saved

Reading in the actual array contents:
According to its contents, array GlobalDouble has 2 dimensions
  and was a global variable when saved
According to its contents, array LocalString has 2 dimensions
  and was a local variable when saved
According to its contents, array LocalDouble has 2 dimensions
  and was a local variable when saved
Done
```

Creating a MAT-File in Fortran

This program, `matdemo1.F`, creates the MAT-file, `matdemo.mat`. To see the code, you can open the file in MATLAB Editor.

Once you have compiled and linked your MAT-file program, you can run the stand-alone application you have just produced. This program creates a MAT-file, `matdemo.mat`, that can be loaded into MATLAB. To run the application, depending on your platform, either double-click its icon or enter `matdemo1` at the system prompt:

```
matdemo1
Creating MAT-file matdemo.mat ...
Done creating MAT-file
```

To verify that the MAT-file has been created, at the MATLAB prompt enter

```
whos -file matdemo.mat
Name          Size         Bytes  Class

Numeric       3x3             72  double array
String        1x33            66  char array

Grand total is 42 elements using 138 bytes
```

Note For an example of a Windows stand-alone program (not MAT-file specific), see `engwindemo.c` in the *`matlabroot`*`\extern\examples\eng_mat` directory.

Reading a MAT-File in Fortran

This program, `matdemo2.F`, illustrates how to use the library routines to read the MAT-file created by `matdemo1.F` and describe its contents. To see the code, you can open the file in MATLAB Editor.

After compiling and linking this program, you can view its results:

```
matdemo2
Directory of Mat-file:
String
Numeric
Getting full array contents:
  1
Retrieved String
```

```
  With size   1-by-  33
  3
Retrieved Numeric
  With size   3-by-  3
```

Compiling and Linking MAT-File Programs

In this section...

Masking Floating Point Exceptions

Certain mathematical operations can result in nonfinite values. For example, division by zero results in the nonfinite IEEE value, `inf`. A floating-point exception occurs when such an operation is performed. Because MATLAB uses an IEEE model that supports nonfinite values such as `inf` and `NaN`, MATLAB disables, or *masks*, floating-point exceptions.

Some compilers do not mask floating-point exceptions by default. This causes MAT-file applications built with such compilers to terminate when a floating-point exception occurs. Consequently, you need to take special precautions when using these compilers to mask floating-point exceptions so that your MAT-file application performs properly.

Note MATLAB-based applications should never get floating-point exceptions. If you do get a floating-point exception, verify that any third-party libraries that you link against do not enable floating-point exception handling.

The only compiler and platform on which you need to mask floating-point exceptions is the Borland C++ compiler on Windows.

Borland C++ Compiler on Windows

To mask floating-point exceptions when using the Borland C++ compiler on the Windows platform include the following at the beginning of your `main()` or `WinMain()` function, before calling any MATLAB API functions:

```
#include <float.h>
   .
   .
   .
_control87(MCW_EM,MCW_EM);
   .
   .
   .
```

Compiling and Linking on UNIX

Under UNIX at run-time, you must tell the system where the API shared libraries reside. These sections provide the necessary UNIX commands, depending on your shell and system architecture.

Setting Run-Time Library Path

Set the library path as follows for the C and Bourne shells. Replace the terms *envvar* and *pathspec* with the appropriate values from the table below.

In the C shell, set the library path using the command

```
setenv envvar pathspec
```

In the Bourne shell, use

```
envvar = pathspec:envvar
export envvar
```

Operating System	*envvar*	*pathspec*
Linux	LD_LIBRARY_PATH	*matlabroot*/bin/glnx86: *matlabroot*/sys/os/glnx86
64-bit Linux	LD_LIBRARY_PATH	*matlabroot*/bin/glnxa64: *matlabroot*/sys/os/glnxa64
64-bit Solaris SPARC	LD_LIBRARY_PATH	*matlabroot*/bin/sol64: *matlabroot*/sys/os/sol64

Operating System	*envvar*	*pathspec*
Macintosh (PPC)	DYLD_LIBRARY_PATH	*matlabroot*/bin/mac: *matlabroot*/sys/os/mac
Macintosh (Intel)	DYLD_LIBRARY_PATH	*matlabroot*/bin/maci: *matlabroot*/sys/os/maci

For example, for the C shell on a Solaris system use:

```
setenv LD_LIBRARY_PATH
matlabroot/bin/sol64:matlabroot/sys/os/sol64
```

and for the Bourne shell:

```
LD_LIBRARY_PATH=matlabroot/bin/sol64:matlabroot/sys/os/sol64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

You can place these commands in a startup script such as `~/.cshrc` for C shell or `~/.profile` for Bourne shell.

Using the Options File

MATLAB provides an options file, `matopts.sh`, that lets you use the `mex` script to easily compile and link MAT-file applications. For example, to compile and link the `matcreat.c` example, first copy the file

```
matlabroot/extern/examples/eng_mat/matcreat.c
```

(where *matlabroot* is the MATLAB root directory) to a writable directory, then use the following command to build it:

```
mex -f matlabroot/bin/matopts.sh matcreat.c
```

If you need to modify the options file for your particular compiler or platform, use the `-v` switch to view the current compiler and linker settings and then make the appropriate changes in a local copy of the `matopts.sh` file.

Compiling and Linking on Windows

To compile and link Fortran or C MAT-file programs, use the `mex` script with a MAT options file. The MAT options files reside in *`matlabroot`*`\bin\win32\mexopts` or *`matlabroot`*`\bin\win64\mexopts` and are named `*engmatopts.bat`, where `*` represents the compiler type and version.

For example, to compile and link the stand-alone MAT application `matcreat.c` using MSVC Version 7.1 on Windows, first copy the file

```
matlabroot\extern\examples\eng_mat\matcreat.c
```

(where *`matlabroot`* is the MATLAB root directory) to a directory that is writable, and then use the following command to build it:

```
mex -f matlabroot\bin\win32\mexopts\msvc71engmatopts.bat matcreat.c
```

If you need to modify the options file for your particular compiler, use the `-v` switch to view the current compiler and linker settings and then make the appropriate changes in a local copy of the options file.

Required Files from Third-Party Sources

MATLAB requires the following data and library files for building any MAT-file application. You must also distribute these files along with any MAT-file application that you deploy to another system.

Third-Party Data Files

When building a MAT-file application on your system or deploying a MAT-file application to some other system, make sure that the appropriate Unicode data file is installed in the *`matlabroot`*`/bin/$ARCH` directory. MATLAB uses this file to support Unicode.

For systems that order bytes in a big-endian manner, use `icudt32b.dat`.

For systems that order bytes in a little-endian manner, use `icudt32l.dat`.

Third-Party Libraries

When building a MAT-file application on your system or deploying a MAT-file application to some other system, make sure that the appropriate libraries are installed in the *matlabroot*/bin/$ARCH directory:

On Windows	On UNIX	On Intel-based Macintosh
libmat.dll	libmat.so	libmat.dylib
libmx.dll	libmx.so	libmx.dylib

In addition to these libraries, you must also have all third-party library files that libmat depends on. MATLAB uses these additional libraries to support Unicode character encoding and data compression in MAT-files. These library files must reside in the same directory as libmx.

You can determine what most of these libraries are using the platform-specific methods described below.

On Linux or Solaris Systems

Type the following command:

```
ldd -d libmat.so
```

On Macintosh Systems

Type the following command:

```
otool -L libmat.dylib
```

On Windows Systems

Download the Dependency Walker utility from the following Web site:

```
http://www.dependencywalker.com/
```

and then drag and drop the file *matlabroot*/bin/win32/libmat.dll or *matlabroot*/bin/win64/libmat.dll into Depends window. (*matlabroot* represents the MATLAB root directory).

Working Directly with Unicode

If you need to manipulate Unicode text directly in your application, the latest version of International Components for Unicode (ICU) is freely available online from the IBM Corporation Web site at

```
http://icu.sourceforge.net/download
```

2

MATLAB Interface to Generic DLLs

A shared library is a collection of functions that are available for use by one or more applications running on a system. MATLAB supports dynamic linking of external libraries on 32-bit MS-Windows systems and on 32-bit Linux systems.

You precompile the library into a dynamic link library file (`.dll`) on Windows, a shared object file (`.so`) on UNIX and Linux, or a dynamic shared library (`.dylib`) on Intel-based Macintosh. At run-time, the library is loaded into memory and made accessible to all applications. The MATLAB Interface to Generic DLLs enables you to interact with functions in dynamic link libraries directly from MATLAB.

This chapter covers the following topics:

Overview (p. 2-3)	Describes how to call functions in external, shared libraries (`.dll`, `.so`, and `.dylib` files) from MATLAB.
Loading and Unloading the Library (p. 2-4)	Describes functions to use in loading the library into MATLAB memory and later releasing that memory.
Getting Information About the Library (p. 2-6)	Shows several ways of obtaining information about the functions contained in a library.
Invoking Library Functions (p. 2-9)	Tells you how to make a call to any function in the library.

Passing Arguments (p. 2-10)	Explains how to construct MATLAB arguments that are compatible with the argument types found in the library functions.
Data Conversion (p. 2-15)	Describes how to convert MATLAB data to C data types when you need to do the conversion manually.

Overview

C programs built into external, shared libraries are accessed by MATLAB through a command-line interface. This interface gives you the ability to load an external library into MATLAB memory space and then access any of the functions defined therein. Although data types differ between the two language environments, in most cases you can pass MATLAB types to the C functions without having to do the work of conversion. MATLAB does this for you.

Note The MATLAB Generic Shared Library interface does not support library functions that have function pointer inputs because there is no way to write a MATLAB function that is compatible with a C function pointer.

This interface also supports libraries containing functions programmed in languages other than C, provided that the functions have a C interface. For example, it is possible to call a Visual Basic 6.0 DLL using `loadlibrary`; however, you must create a C header file for the library. The ActiveX interface might be simpler to use for this reason. See “Introducing MATLAB COM Integration” on page 8-3 for more information on ActiveX.

Loading and Unloading the Library

In this section...

Using a Shared Library

To give MATLAB access to external functions in a shared library, you must first load the library into memory. Once loaded, you can request information about any of the functions in the library and call them directly from MATLAB. When the library is no longer needed, unload it from memory to conserve memory usage.

Loading the Library

To load a shared library into MATLAB, use the `loadlibrary` function. The syntax for `loadlibrary` is

```
loadlibrary('shrlib', 'hfile')
```

where `shrlib` is the filename for the shared library file, and `hfile` is the filename for the header file that contains the function prototypes. See the reference page for `loadlibrary` for variations in the syntax that you can use and information on library file extensions.

Note The header file provides signatures for the functions in the library and is a required argument for `loadlibrary`.

As an example, you can use `loadlibrary` to load the `libmx` library that defines the MATLAB `mx` routines. The first statement below forms the directory specification for the `matrix.h` header file for the `mx` routines. The second loads the library from `libmx` (note the file extension is platform dependent), also specifying the header file:

```
hfile = [matlabroot '\extern\include\matrix.h'];
loadlibrary('libmx', hfile)
```

There are also several optional arguments that you can use with `loadlibrary`. See the `loadlibrary` reference page for more information.

Unloading the Library

To unload the library and free up the memory that it occupied, use the `unloadlibrary` function. For example:

```
unloadlibrary libmx
```

Getting Information About the Library

In this section...

Introduction

You can use these functions to get information on the functions available in a library that you have loaded:

```
libfunctions('libname')
libfunctionsview('libname')
```

The main difference is that `libfunctions` displays the information in the MATLAB Command Window (and you can assign its output to a variable), and `libfunctionsview` displays the information as a graphical display in a new window.

Listing Functions

To see what functions are available in the `libmx` library, use `libfunctions`, specifying the library filename as the only argument. Note that you can use the MATLAB command syntax (with no parentheses or quotes required) when specifying no output variables:

```
libfunctions libmx

Functions in library libmx:

mxAddField               mxGetFieldNumber  mxIsLogicalScalarTrue
mxArrayToString          mxGetImagData     mxIsNaN
mxCalcSingleSubscript    mxGetInf          mxIsNumeric
mxCalloc                 mxGetIr           mxIsObject
mxClearScalarDoubleFlag  mxGetJc           mxIsOpaque
mxCreateCellArray        mxGetLogicals     mxIsScalarDoubleFlagSet
       .                        .                 .
```

```
        .                   .                   .
        .                   .                   .
```

To list the functions along with their signatures, use the `-full` switch with `libfunctions`. This shows the MATLAB syntax for calling functions written in C. The data types used in the argument lists and return values match MATLAB types, not C types. See the section "Data Conversion" on page 2-15 for more information on these data types.

```
libfunctions libmx -full

Functions in library libmx:

[int32, MATLAB array, cstring] mxAddField(MATLAB array, cstring)
[cstring, MATLAB array] mxArrayToString(MATLAB array)
[int32, MATLAB array, int32Ptr] mxCalcSingleSubscript(MATLAB array, int32, int32Ptr)
lib.pointer mxCalloc(uint32, uint32)
MATLAB array mxClearScalarDoubleFlag(MATLAB array)
[MATLAB array, int32Ptr] mxCreateCellArray(int32, int32Ptr)
MATLAB array mxCreateCellMatrix(int32, int32)
        .
        .
        .
```

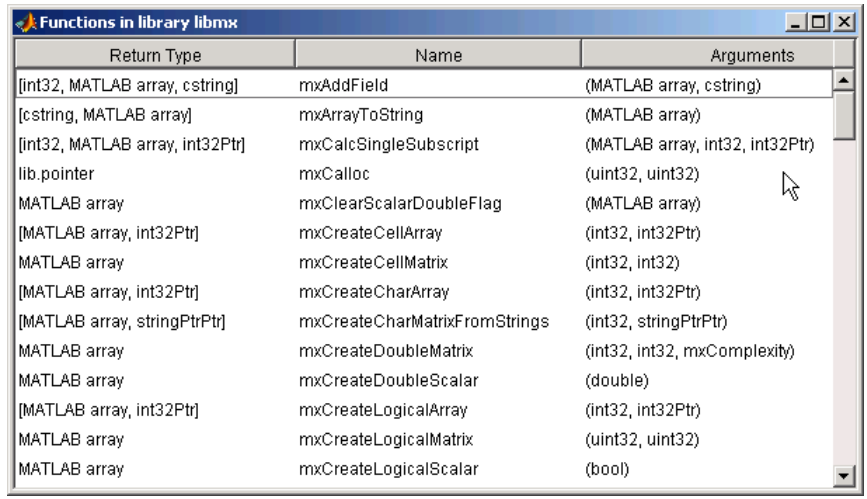
Viewing Functions in a GUI Interface

The `libfunctionsview` function creates a new window that displays all of the functions defined in a specific library. For each method, the following information is shown.

Heading	Description
Return Type	Data types that the method returns
Name	Function name
Arguments	Valid data types for input arguments
Inherited From	Not relevant for shared library functions

The following command opens the window shown below for the `libmx` library:

```
libfunctionsview libmx
```



As was true for the `libfunctions` function, the data types displayed here are MATLAB types. See the section “Data Conversion” on page 2-15 for more information on these data types.

Invoking Library Functions

Once a shared library has been loaded into MATLAB, use the `calllib` function to call any of the functions from that library. Specify the library name, function name, and any arguments that get passed to the function:

```
calllib('libname', 'funcname', arg1, ..., argN)
```

This example calls functions from the `libmx` library to test the value stored in `y`.

```
hfile = [matlabroot '\extern\include\matrix.h'];
loadlibrary('libmx', hfile)

y = rand(4, 7, 2);

calllib('libmx', 'mxGetNumberOfElements', y)
ans =
    56

calllib('libmx', 'mxGetClassID', y)
ans =
   mxDOUBLE_CLASS
```

See the section “Passing Arguments” on page 2-10 for information on how to define the argument types.

Passing Arguments

In this section...

Displaying MATLAB Syntax for Library Functions

A sample external library called `shrlibsample` is supplied with MATLAB. The library file for the `shrlibsample` library resides in the directory `extern\examples\shrlib`. MATLAB selects the appropriate version for your platform. The `mexext` function returns the file extension that is used on your platform. See the `mex` function for a list of all extensions used by MATLAB.

To use the `shrlibsample` library, you first need to either add this directory to your MATLAB path with the command

```
addpath([matlabroot '\extern\examples\shrlib'])
```

or make this your current working directory with the command,

```
cd([matlabroot '\extern\examples\shrlib'])
```

The following example loads the `shrlibsample` library and displays the MATLAB syntax for calling functions that come with the library:

```
loadlibrary shrlibsample shrlibsample.h
libfunctions shrlibsample -full

Functions in library shrlibsample:

[double, doublePtr] addDoubleRef(double, doublePtr, double)
double addMixedTypes(int16, int32, double)
[double, c_structPtr] addStructByRef(c_structPtr)
double addStructFields(c_struct)
```

```
c_structPtrPtr allocateStruct(c_structPtrPtr)
voidPtr deallocateStruct(voidPtr)
doublePtr multDoubleArray(doublePtr, int32)
[lib.pointer, doublePtr] multDoubleRef(doublePtr)
int16Ptr multiplyShort(int16Ptr, int32)
cstring readEnum(Enum1)
[cstring, cstring] stringToUpper(cstring)
```

While these functions are all written in C, `libfunctions` with the `full` option displays the MATLAB syntax for calling the C functions.

See "Primitive Types" on page 2-15 for a table of extended MATLAB data types (e.g., `doublePtr`).

General Rules for Passing Arguments

There are some important things to note about the input and output arguments shown in the function listing of the previous section:

- Many of the arguments (like `int32`, `double`) are very similar to their C counterparts. In these cases, you need only to pass in the MATLAB data types shown for these arguments.
- Some arguments in C (like `**double`, or predefined structures), are quite different from standard MATLAB data types. In these cases, you usually have the option of either passing a standard MATLAB type and letting MATLAB convert it for you, or converting the data yourself using MATLAB functions like `libstruct` and `libpointer`. See the next section on "Data Conversion" on page 2-15.
- C input arguments are often passed by reference. Although MATLAB does not support passing by reference, you can create MATLAB arguments that are compatible with C references. In the listing shown above, these are the arguments with names ending in `Ptr` and `PtrPtr`. See "Creating References" on page 2-26.
- C functions often return data in input arguments passed by reference. MATLAB creates additional output arguments to return these values. Note that in the listing in the previous section, all input arguments ending in `Ptr` or `PtrPtr` are also listed as outputs.

General Guidelines for Passing Arguments

- Nonscalar arguments must be declared as passed by reference in the library functions.
- If the library function uses single subscript indexing to reference a two-dimensional matrix, keep in mind that C programs process matrices row by row while MATLAB processes matrices by column. To get C behavior from the function, transpose the input matrix before calling the function, and then transpose the function output.
- When passing an array having more than two dimensions, the shape of the array might be altered by MATLAB. To ensure that the array retains its shape, store the size of the array before making the call, and then use this same size to reshape the output array to the correct dimensions:

```
vs = size(vin)              % Store the original dimensions
vs =
     2     5     2

vout = calllib('shrlibsample','multDoubleArray', vin, 20);

size(vout)                  % Dimensions have been altered
ans =
     2    10

vout = reshape(vout, vs);   % Restore the array to 2-by-5-by-2

size(vout)
ans =
     2     5     2
```

- Use an empty array, `[]`, to pass a `NULL` parameter to a library function that supports optional input arguments. This is valid only when the argument is declared as a `Ptr` or `PtrPtr` as shown by `libfunctions` or `libfunctionsview`.

Passing References

Many functions in external libraries use arguments that are passed by reference. To enable you to interact with these functions, MATLAB passes what is called a *pointer object* to these arguments. This should not be confused

with “passing by reference” in the typical sense of the term. See “Creating References” on page 2-26 for more information.

Passing a NULL Pointer

You can create a `NULL` pointer to pass to library functions in the following ways:

- Pass a `0` as the argument.
- Use the `libpointer` function:

```
p = libpointer; % no arguments

p = libpointer('string') % string argument

p = libpointer('stringPtr') % pointer to a string argument
```

- Use the `libstruct` function:

```
p = libstruct; % no arguments
```

Using C++ Libraries

The `loadlibrary` function cannot load C++ libraries unless you define the function prototypes as `extern "C"` in the library header file. For example, the following function prototype from the file `mex.h` shows the syntax to use for each function:

```
#ifdef __cplusplus
extern "C" {
#endif
void mexFunction(
    int           nlhs,
    mxArray       *plhs[],
    int           nrhs,
    const mxArray *prhs[]
);
#ifdef __cplusplus
}
#endif
```

Another approach to using C++ libraries is to generate a prototype M-file that contain aliases for the mangled C++ function names. Use the original (premangled) function names as the aliases for the C++ functions. Generate the M-file with the `mfilename` option of the `loadlibrary` function and then determine which functions in the library you want to make available by defining aliases for these functions.

Data Conversion

In this section...

When to Convert Manually

Under most conditions, data passed to and from external library functions is automatically converted by MATLAB to the data type expected by the external function. However, you may choose, at times, to convert some of your argument data manually. Circumstances under which you might find this advantageous are

- When you pass the same piece of data to a series of library functions, it probably makes more sense to convert it once manually at the start rather than having MATLAB convert it automatically on every call. This saves time on unnecessary copy and conversion operations.
- When you pass large structures, you can save memory by creating MATLAB structures that match the shape of the C structures used in the external function instead of using generic MATLAB structures. The `libstruct` function creates a MATLAB structure modeled from a C structure taken from the library. See "Structures" on page 2-20 for more information.
- When an argument to an external function uses more than one level of referencing (e.g., `double **`), you must pass a reference that you have constructed using the `libpointer` function rather than relying on MATLAB to convert the data type automatically.

Primitive Types

All standard scalar C data types are supported by the shared library interface. These are shown in the two tables below along with their equivalent MATLAB

types. MATLAB uses the type from the right column for arguments having the C type shown in the left column.

The second table shows *extended* MATLAB types in the right column. These are instances of the MATLAB `lib.pointer` class rather than standard MATLAB data types. See "Creating References" on page 2-26 for information on the `lib.pointer` class.

MATLAB Primitive Types

C Type (on a 32-bit computer)	Equivalent MATLAB Type
`char`, `byte`	`int8`
`unsigned char`, `byte`	`uint8`
`short`	`int16`
`unsigned short`	`uint16`
`int`, `long`	`int32`
`unsigned int`, `unsigned long`	`uint32`
`float`	`single`
`double`	`double`
`char *`	`cstring (1xn char array)`
`*char[]`	cell array of strings

MATLAB Extended Types

C Type (on a 32-bit computer)	Equivalent MATLAB Type
`integer pointer types (int *)`	`(u)int(`*`size`*`)Ptr`
Null-terminated string passed by value	`cstring`
Null-terminated string passed by reference (from a `libpointer` only)	`stringPtr`
Array of pointers to strings (or one `**char`)	`stringPtrPtr`

MATLAB Extended Types (Continued)

C Type (on a 32-bit computer)	Equivalent MATLAB Type
Matrix of signed bytes	`int8Ptr`
`float *`	`singlePtr`
`double *`	`doublePtr`
`mxArray *`	`MATLAB array`
`void *`	`voidPtr`
`void **`	`voidPtrPtr`
`type` `**`	Same as *`type`*`Ptr` with an added `Ptr` (e.g., `double **` is `doublePtrPtr`)

Converting to Other Primitive Types

For primitive types, MATLAB automatically converts any argument to the data type expected by the external function. This means that you can pass a `double` to a function that expects to receive a `byte` (8-bit integer) and MATLAB does the conversion for you.

For example, the C function shown here takes arguments that are of types `short`, `int`, and `double`:

```
double addMixedTypes(short x, int y, double z)
{
   return (x + y + z);
}
```

You can simply pass all of the arguments as type `double` from MATLAB. MATLAB determines what type of data is expected for each argument and performs the appropriate conversions:

```
calllib('shrlibsample', 'addMixedTypes', 127, 33000, pi)
ans =
  3.3130e+004
```

Converting to a Reference

MATLAB also automatically converts an argument passed by value into an argument passed by reference when the external function prototype defines the argument as a reference. So a MATLAB `double` argument passed to a function that expects `double *` is converted to a `double` reference by MATLAB.

`addDoubleRef` is a C function that takes an argument of type `double *`:

```
double addDoubleRef(double x, double *y, double z)
{
    return (x + *y + z);
}
```

Call the function with three arguments of type `double`, and MATLAB handles the conversion:

```
calllib('shrlibsample', 'addDoubleRef', 1.78, 5.42, 13.3)
ans =
   20.5000
```

Strings

For arguments that require `char *`, you can pass a MATLAB string (i.e., character array).

For example, the following C function takes a `char *` input argument:

```
char* stringToUpper(char *input) {
   char *p = input;

   if (p != NULL)
      while (*p!=0)
         *p++ = toupper(*p);
   return input;
}
```

`libfunctions` shows that you can use a MATLAB `cstring` for this input.

```
libfunctions shrlibsample -full
          .
```

```
        .
    [cstring, cstring] stringToUpper(cstring)
```

Create a MATLAB character array, `str`, and pass it as the input argument:

```
str = 'This was a Mixed Case string';
calllib('shrlibsample', 'stringToUpper', str)
ans =
   THIS WAS A MIXED CASE STRING
```

Note Although the input argument that MATLAB passes to `stringToUpper` resembles a reference to type `char`, it is not a true reference data type. That is, it does not contain the address of the MATLAB character array, `str`. So, when the function executes, it returns the correct result but does not modify the value in `str`. If you now examine `str`, you find that its original value is unchanged:

```
str

str =

  This was a Mixed Case string
```

Enumerated Types

For arguments defined as C enumerated types, you can pass either the enumeration string or its integer equivalent.

The `readEnum` function from the `shrlibsample` library returns the enumeration string that matches the argument passed in. Here is the `Enum1` definition and the `readEnum` function in C:

```
enum Enum1 {en1 = 1, en2, en4 = 4} TEnum1;

char* readEnum(TEnum1 val) {
   switch (val) {
   case 1 :return "You chose en1";
   case 2: return "You chose en2";
   case 4: return "You chose en4";
   default : return "enum not defined";
   }
}
```

In MATLAB, you can express an enumerated type as either the enumeration string or its equivalent numeric value. The `TEnum1` definition above declares enumeration `en4` to be equal to `4`. Call `readEnum` first with a string:

```
calllib('shrlibsample', 'readEnum', 'en4')
ans =
   You chose en4
```

Now call it with the equivalent numeric argument, `4`:

```
calllib('shrlibsample', 'readEnum', 4)
ans =
   You chose en4
```

Structures

For library functions that take structure arguments, you need to pass structures that have field names that are the same as those in the structure definitions in the library. To determine the names and also the data types of structure fields, you can:

- Consult the documentation that was provided with the library.
- Look for the structure definition in the header file that you used to load the library into MATLAB.

When you create and initialize the structure, you do not necessarily have to match the data types of numeric fields. MATLAB converts to the correct numeric type for you when you make the call using the `calllib` function.

Finding Field Names From MATLAB

You can also determine the field names of an externally defined structure from within MATLAB using the following procedure:

1. Use `libfunctionsview` to display the signatures for all functions in the library. `libfunctionsview` shows the names of the structures used by each function. For example, when you type

   ```
   libfunctionsview shrlibsample
   ```

   MATLAB opens a new window displaying function signatures for the `shrlibsample` library. The line showing the `addStructFields` function reads:

   ```
   double addStructFields (c_struct)
   ```

2. If the function you are using takes a structure argument, get the structure type from the `libfunctionsview` display, and invoke the `libstruct` function on that type. `libstruct` returns an object that is modeled on the structure as defined in the library:

   ```
   s = libstruct('c_struct');
   ```

3. Get the names of the structure fields from the object returned by `libstruct`:

   ```
   get(s)
       p1: 0
       p2: 0
       p3: 0
   ```

4. Initialize the fields to the values you want to pass to the library function and make the call using `calllib`:

   ```
   s.p1 = 476;   s.p2 = -299;   s.p3 = 1000;
   calllib('shrlibsample','addStructFields',s);
   ```

Specifying Structure Field Names

Here are a few general guidelines that apply to structures passed to external library functions:

- These structures can contain fewer fields than defined in the library structure. MATLAB sets any undefined or uninitialized fields to zero.
- Any structure field name that you use must match a field name in the structure definition. Structure names are case sensitive.
- Structures cannot contain additional fields that are not in the library structure definition.

Passing a MATLAB Structure

As with other data types, when an external function takes a structure argument (such as a C structure), you can pass a MATLAB structure to the function in its place. Structure field names must match field names defined in the library, but data types for numeric fields do not have to match. MATLAB converts each numeric field of the MATLAB structure to the correct data type.

Example of Passing a MATLAB Structure. The sample shared library, `shrlibsample`, defines the following C structure and function:

```
struct c_struct {
    double p1;
    short p2;
    long  p3;
};

double addStructFields(struct c_struct st)
{
    double t = st.p1 + st.p2 + st.p3;
    return t;
}
```

The following code passes a MATLAB structure, `sm`, with three `double` fields to `addStructFields`. MATLAB converts the fields to the `double`, `short`, and `long` data types defined in the C structure, `c_struct`.

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;

calllib('shrlibsample', 'addStructFields', sm)
ans =
        1177
```

Passing a libstruct Object

When you pass a structure to an external function, MATLAB makes sure that the structure being passed matches the library's definition for that structure type. It must contain all the necessary fields defined for that type and each field must be of the expected data type. For any fields that are missing in the structure being passed, MATLAB creates an empty field of that name and initializes its value to zero. For any fields that have a data type that doesn't match the structure definition, MATLAB converts the field to the expected type.

When working with small structures, it is efficient enough to have MATLAB do this work for you. You can pass the original MATLAB structure with `calllib` and let MATLAB handle the conversions automatically. However, when working with repeated calls that pass one or more large structures, it may be to your advantage to convert the structure manually before making any calls to external functions. In this way, you save processing time by converting the structure data only once at the start rather than at each function call. You can also save memory if the fields of the converted structure take up less space than the original MATLAB structure.

Preconverting a MATLAB Struct with libstruct. You can convert a MATLAB structure to a C-like structure derived from a specific type definition in the library in one step. Call the `libstruct` function, passing in the name of the structure type from the library, and the original structure from MATLAB. The syntax for `libstruct` is

```
s = libstruct('structtype', mlstruct)
```

The value `s` returned by this function is called a *libstruct object*. Although it is truly a MATLAB object, it handles much like a MATLAB structure. The fields of this new "structure" are derived from the external structure type specified by `structtype` in the syntax above.

For example, to convert a MATLAB structure, `sm`, to a `libstruct` object, `sc`, that is derived from the `c_struct` structure type, use

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
sc = libstruct('c_struct', sm);
```

The original structure, `sm`, has fields that are all of type `double`. The object, `sc`, returned from the `libstruct` call has fields that match the `c_struct` structure type. These fields are `double`, `short`, and `long`.

Note You can only use `libstruct` on scalar structures.

Creating an Empty libstruct Object. You can also create an empty libstruct object by calling `libstruct` with only the `structtype` argument. This constructs an object with all the required fields and with each field initialized to zero.

```
s = libstruct('structtype')
```

libstruct Requirements for Structures. When converting a MATLAB structure to a libstruct object, the structure to be converted must adhere to the same guidelines that were documented for MATLAB structures passed directly to external functions. See “Specifying Structure Field Names” on page 2-21.

Using the Structure as an Object

The value returned by `libstruct` is not a true MATLAB structure. It is actually an instance of a class called `lib.c_struct`, as seen by examining the output of `whos`:

```
whos sc
  Name      Size                    Bytes  Class

sc         1x1                           lib.c_struct
sm         1x1                       396  struct array

Grand total is 7 elements using 396 bytes
```

Determining the Size of a lib.c_struct Object. You can use the `lib.c_struct` class method `structsize` to obtain the size of a `lib.c_struct` object:

```
sc.structsize
ans =
  16
```

Accessing lib.c_struct Fields. The fields of this structure are implemented as properties of the `lib.c_struct` class. You can read and modify any of these fields using the MATLAB object-oriented functions, `set` and `get`:

```
sc = libstruct('c_struct');

set(sc, 'p1', 100, 'p2', 150, 'p3', 200);

get(sc)
    p1: 100
    p2: 150
    p3: 200
```

You can also read and modify the fields by simply treating them like any other MATLAB structure fields:

```
sc.p1 = 23;

sc.p1
ans =
   23
```

Example of Passing a libstruct Object

Repeat the `addStructFields` example, this time converting the structure to one of type `c_struct` before calling the function:

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;
sc = libstruct('c_struct', sm);

get(sc)
    p1: 476
    p2: -299
```

```
    p3: 1000
```

Now call the function, passing the structure `sc`:

```
calllib('shrlibsample', 'addStructFields', sc)
ans =
    1177
```

Note When passing manually converted structures, the structure passed must be of the same type as that used by the external function. For example, you cannot pass a structure of type `records` to a function that expects type `c_struct`.

Creating References

You can pass most arguments to an external function by value, even when the prototype for that function declares the argument to be a reference. The `calllib` function uses the header file to determine how to convert the function arguments.

There are times, however, when it is useful to pass MATLAB arguments that are the equivalent of C references:

- You want to modify the data in the input arguments.
- You are passing large amounts of data, and you don't want MATLAB to make copies of the data.
- The library is going to store and use the pointer for a period of time so it is better to give the M-code control over the lifetime of the pointer object.

In the cases above, you should use `libpointer` to construct a pointer object of a specified type (for structures use `libstruct`).

Constructing a Reference with the libpointer Function

To construct a reference, use the function `libpointer` with this syntax:

```
p = libpointer('type', 'value')
```

To give an example, create a pointer `pv` to an `int16` value. In the first argument to `libpointer`, enter the `type` of pointer you are creating. The type is always the data type (`int16`, in this case) suffixed by the letters `Ptr`:

```
v = int16(485);
pv = libpointer('int16Ptr', v);
```

The value returned, `pv`, is actually an instance of a MATLAB class called `lib.pointer`. The `lib.pointer` class has the properties `Value` and `DataType`. You can read and modify these properties with the MATLAB `get` and `set` functions:

```
get(pv)
       Value: 485
    DataType: 'int16Ptr'
```

The `lib.pointer` class also has two methods, `setdatatype` and `reshape`, that are described in the next section, "Obtaining the Function's Return Values" on page 2-28:

```
methods(pv)

Methods for class lib.pointer:
   disp plus reshape setdatatype
```

Creating a Reference to a Primitive Type

Here is a simple example that illustrates how to construct and pass a pointer to type `double`, and how to interpret the output data. The function `multDoubleRef` takes one input that is a reference to a `double` and returns the same.

Here is the C function:

```
double *multDoubleRef(double *x)
{
     *x *= 5;
     return x;
}
```

Construct a reference, xp, to input data, x, and verify its contents:

```
x = 15;
xp = libpointer('doublePtr', x);

get(xp)
       Value: 15
    DataType: 'doublePtr'
```

Now call the function and check the results:

```
calllib('shrlibsample', 'multDoubleRef', xp);

get(xp, 'Value')
ans =
   75
```

Note It is important to note that reference xp is not a true pointer as it would be in a language like C. That is, even though it was constructed as a reference to x, it does not contain the address of x. So, when the function executes, it modifies the Value property of xp, but it does not modify the value in x. If you now examine x, you find that its original value is unchanged:

```
x

x =

     15
```

Obtaining the Function's Return Values. In the last example, the result of the function called from MATLAB could be obtained by examining the modified input reference. But this function also returns data in its output arguments that may be useful.

The MATLAB prototype for this function (returned by `libfunctions -full`) indicates that MATLAB returns two outputs. The first is an object of class `lib.pointer`; the second is the `Value` property of the `doublePtr` input argument:

```
libfunctions shrlibsample -full
   [lib.pointer, doublePtr] multDoubleRef(doublePtr)
```

Run the example once more, but this time check the output values returned:

```
x = 15;
xp = libpointer('doublePtr', x);

[xobj, xval] = calllib('shrlibsample', 'multDoubleRef', xp)
xobj =
   lib.pointer
xval =
    75
```

Like the input reference argument, the first output, `xobj`, is an object of class `lib.pointer`. You can examine this output, but first you need to initialize its data type and size as these factors are undefined when returned by the function. Use the `setdatatype` method defined by class `lib.pointer` to set the data type to `doublePtr` and the size to `1`-by-`1`. Once initialized, you can examine outputs.

The first output is `xobj`:

```
setdatatype(xobj, 'doublePtr', 1, 1)

get(xobj)
ans =
       Value: 75
    DataType: 'doublePtr'
```

The second output, `xval`, is a `double` copied from the `Value` of input `xp`.

Creating a Reference by Offsetting from an Existing libpointer. You can use the plus operator (`+`) to create a new pointer that is offset from an existing pointer by a scalar numeric value. Note that this operation applies only to pointer of numeric data types. For example, suppose you create a `libpointer` to the vector `x`:

```
x = 1:10;
xp = libpointer('doublePtr',x);
xp.Value
ans =

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

You can now use the plus operator to create a new `libpointer` that is offset from the `xp`:

```
xp2 = xp+4;
xp2.Value

ans =

     5     6     7     8     9    10
```

Note that the new pointer (`xp2` in this example) is valid only as long as the original pointer exists.

Creating a Structure Reference

Creating a reference argument to a structure is not much different than using a reference to a primitive type. The function shown here takes a reference to a structure of type `c_struct` as its only input. It returns an output argument that is the sum of all fields in the structure. It also modifies the fields of the input argument:

```
double addStructByRef(struct c_struct *st)
{
    double t = st->p1 + st->p2 + st->p3;
    st->p1 = 5.5;
    st->p2 = 1234;
    st->p3 = 12345678;
    return t;
 }
```

Passing the Structure Itself. Although this function expects to receive a structure reference input, it is easier to pass the structure itself and let MATLAB do the conversion to a reference. This example passes a MATLAB structure, `sm`, to the function `addStructByRef`. MATLAB returns the correct value in the output, `x`, but does not modify the contents of the input, `sm`, since `sm` is not a reference:

```
sm.p1 = 476;   sm.p2 = -299;   sm.p3 = 1000;

x = calllib('shrlibsample', 'addStructByRef', sm)
x =
        1177
```

Passing a Structure Reference. The second part of this example passes the structure by reference. This time, the function receives a pointer to the structure and is then able to modify the structure fields.

```
sp = libpointer('c_struct', sm);
calllib('shrlibsample', 'addStructByRef', sp)
ans =
        1177

get(sp, 'Value')
ans =
    p1: 5.5000
    p2: 1234
    p3: 12345678
```

Passing a Pointer to the First Element of an Array

In cases where a function defines an input argument that is a pointer to the first element of a data array, the `calllib` function automatically passes an argument that is a pointer of the correct type to the first element of data in the MATLAB vector or matrix. For example, the following C function `sum` requires an argument that is a pointer to the first element of an array of shorts (`int16`).

Suppose you have the following variables defined in MATLAB:

```
Data = 1:100;
lp = libpointer(('int16Ptr',Data);
shortData = int16(Data);
```

The signature of the C function `sum` is:

```
int sum(int size, short* data);
```

All of the following statements work correctly and give the same answer:

```
summed_data = calllib('libname','sum',100,Data);
summed_data = calllib('libname','sum',100,shortData);
summed_data = calllib('libname','sum',100,lp);
```

Note that the `size` and `data` arguments must match. That is, length of the data vector must be equal to the specified size. For example:

```
% sum last 50 elements
summed_data = calllib('libname','sum',50,Data(50:100));
```

Creating a Void Pointer to a String

Suppose you want to create a `voidPtr` that points to a string as an input argument. In C, characters are represented as unsigned eight-bit integers. Therefore, you must first cast the string to this MATLAB type before creating a variable of type `voidPtr`.

You can create a variable of the correct type and value using `libpointer` as follows:

```
str = 'string variable';
vp = libpointer('voidPtr',[uint8(str) 0]);
```

To obtain the character string from the pointer, use

```
char(vp.Value)
ans =
string variable
```

Confirm the type of the pointer by accessing its `DataType` property:

```
vp.DataType
ans =
voidPtr
```

You can call a function that takes a `voidPtr` to a string as an input argument using the following syntax because MATLAB automatically converts an argument passed by value into an argument passed by reference when the external function prototype defines the argument as a reference:

```
func_name(uint8(str))
```

Note that while MATLAB converts the argument from a value to a reference, it must be of the correct type.

Memory Allocation for an External Library

In general, a valid memory address is passed each time you pass a MATLAB variable to a library function. You need to explicitly allocate memory only if the library provides a memory allocation function that you are required to use.

When to Use libpointer. You should use a `libpointer` object in cases where the library is going to store the pointer and access the buffer over a period of time. In these cases, you need to ensure that MATLAB has control over the lifetime of the buffer and to prevent copies of the data from being made. The following pseudo code is an example of asynchronous data acquisition that shows how to use `libpointer` in this type of situation.

Suppose an external library has the following functions:

```
AcquireData(int points, short *buffer)
IsAquistionDone(void)
```

First, create a pointer to a buffer of 1024 points:

```
BufferSize = 1024;
pBuffer = libpointer('int16Ptr',1:BufferSize);
```

Then, begin acquiring data and wait in a loop until it is done:

```
calllib('lib_name','AcquireData,BufferSize,pbuffer);
while (~calllib('lib_name','IsAcquisitionDone')
 pause(0.1)
end
```

The following statement reads the data in the buffer:

```
result = pBuffer.Value;
```

When the library is done with the buffer, `clear` the MATLAB variable:

```
clear pBuffer
```

Reference Pointers

Arguments that have more than one level of referencing (e.g., `uint16 **`) are referred to here as reference pointers. In MATLAB, these argument types are named with the suffix `PtrPtr` (for example, `uint16PtrPtr`). See the output of `libfunctionsview` or `methods -full` for examples of this type.

When calling a function that takes a reference pointer argument, you can use a reference argument instead and MATLAB converts it to the reference pointer. For example, the external `allocateStruct` function expects a `c_structPtrPtr` argument:

```
libfunctions shrlibsample -full
   c_structPtrPtr allocateStruct(c_structPtrPtr)
```

Here is the C function:

```
void allocateStruct(struct c_struct **val)
{
    *val=(struct c_struct*) malloc(sizeof(struct c_struct));
    (*val)->p1 = 12.4;
    (*val)->p2 = 222;
    (*val)->p3 = 333333;
}
```

Although the prototype says that a `c_structPtrPtr` is required, you can use a `c_structPtr` and let MATLAB do the second level of conversion. Create a reference to an empty structure argument and pass it to `allocateStruct`:

```
sp = libpointer('c_structPtr');
calllib('shrlibsample', 'allocateStruct', sp)

get(sp)
ans =
       Value: [1x1 struct]
    DataType: 'c_structPtr'

get(sp, 'Value')
ans =
    p1: 12.4000
    p2: 222
    p3: 333333
```

When you are done, return the memory that you had allocated:

```
calllib('shrlibsample', 'deallocateStruct', sp)
```

3

Calling C and Fortran Programs from MATLAB

Although MATLAB is a complete, self-contained environment for programming and manipulating data, it is often useful to interact with data and programs external to the MATLAB environment. MATLAB provides an interface to external programs written in the C and Fortran languages.

Introducing MEX-Files

In this section...

What are MEX-Files

You can call your own C or Fortran subroutines from MATLAB as if they were built-in functions. MATLAB callable C and Fortran programs are referred to as MEX-files. MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute.

MEX-files have several applications:

- Large pre-existing C and Fortran programs can be called from MATLAB without having to be rewritten as M-files.
- Bottleneck computations that do not run fast enough in MATLAB can be recoded in C or Fortran for efficiency.

MEX-files are not appropriate for all applications. MATLAB is a high-productivity system whose specialty is eliminating time-consuming, low-level programming in compiled languages like Fortran or C. In general, most programming should be done in MATLAB. Don't use the MEX facility unless your application requires it.

MATLAB supports MEX-files created in C++, with some limitations. For more information, see "Creating C++ MEX-Files" on page 4-9.

Using MEX-Files

MEX-files are subroutines produced from C, C++, or Fortran source code. They behave just like M-files and built-in functions. While M-files have a platform-independent extension, `.m`, MATLAB identifies MEX-files by

platform-specific extensions. This table lists the platform-specific extensions for MEX-files.

MEX-File Extensions

Platform	MEX-File Extension
Linux (32-bit)	`mexglx`
Linux x86-64	`mexa64`
Macintosh (PPC)	`mexmac`
Macintosh (Intel)	`mexmaci`
64-bit Solaris SPARC	`mexs64`
Windows (32-bit)	`mexw32`
Windows x64	`mexw64`

You call MEX-files exactly as you call any M-function. For example, on a Windows platform, there is a MEX-file called `histc.mexw32` in one of the MATLAB toolbox directories (*`matlabroot`*`\toolbox\matlab\datafun`) that performs a histogram count. The file `histc.m` contains the help text documentation. When you call `histc` from MATLAB, the dispatcher looks through the list of directories on the MATLAB search path. It scans each directory looking for the first occurrence of a file named `histc` with either the corresponding filename extension from the table or `.m`. When it finds one, it loads the file and executes it. MEX-files take precedence over M-files when like-named files exist in the same directory. However, help text documentation still reads from the `.m` file.

You cannot use a MEX-file on a platform if it was compiled on a different platform. You must recompile the source code on the platform for which you wish to use the MEX-file.

MEX-File Placement

For MATLAB to be able to execute your C or Fortran functions, you must either put the compiled MEX-files containing those functions in a directory on the MATLAB path, or run MATLAB in the directory in which they reside.

Functions in the current working directory are found before functions on the MATLAB path.

Type `path` to see what directories are currently included in your path. You can add new directories to the path either by using the `addpath` function, or by selecting **File > SetPath** to edit the path.

If you are using a Windows operating system and any of your MEX-files are on a network drive, be aware that file servers do not always report directory and file changes correctly. If you change any MEX-files that are on a network drive and you find that MATLAB is not using your latest changes, you can force MATLAB to look for the correct version of the file by changing directories away from and then back to the directory in which the files reside.

The Distinction Between mx and mex Prefixes

Routines in the API that are prefixed with `mx` allow you to create, access, manipulate, and destroy `mxArrays`. Routines prefixed with `mex` perform operations back in the MATLAB environment.

mx Routines

The array access and creation library provides a set of array access and creation routines for manipulating MATLAB arrays. These subroutines, which are fully documented in the online API reference pages, always start with the prefix `mx`. For example, `mxGetPi` retrieves the pointer to the imaginary data inside the array.

Although most of the routines in the array access and creation library let you manipulate the MATLAB array, there are two exceptions—the IEEE routines and memory management routines. For example, `mxGetNaN` returns a `double`, not an `mxArray`.

mex Routines

Routines that begin with the `mex` prefix perform operations back in the MATLAB environment. For example, the `mexEvalString` routine evaluates a string in the MATLAB workspace.

Note `mex` routines are only available in MEX-functions.

MATLAB Data

In this section...

The MATLAB Array

The MATLAB language works with only a single object type: the MATLAB array. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, structures, and objects, are stored as MATLAB arrays. In C, the MATLAB array is declared to be of type `mxArray`. The `mxArray` structure contains, among other things:

- Its type
- Its dimensions
- The data associated with this array
- If numeric, whether the variable is real or complex
- If sparse, its indices and nonzero maximum elements
- If a structure or object, the number of fields and field names

Data Storage

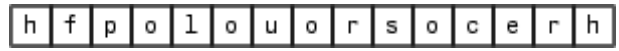
All MATLAB data is stored columnwise, which is how Fortran stores matrices. MATLAB uses this convention because it was originally written in Fortran. For example, given the matrix

```
a=['house'; 'floor'; 'porch']
a =
   house
   floor
   porch
```

its dimensions are

```
size(a)
ans =
     3     5
```

and its data is stored as



Data Types in MATLAB

Complex Double-Precision Matrices

The most common data type in MATLAB is the complex double-precision, nonsparse matrix. These matrices are of type `double` and have dimensions `m`-by-`n`, where `m` is the number of rows and `n` is the number of columns. The data is stored as two vectors of double-precision numbers—one contains the real data and one contains the imaginary data. The pointers to this data are referred to as `pr` (pointer to real data) and `pi` (pointer to imaginary data), respectively. A real-only, double-precision matrix is one whose `pi` is `NULL`.

Numeric Matrices

MATLAB also supports other types of numeric matrices. These are single-precision floating-point and 8-, 16-, and 32-bit integers, both signed and unsigned. The data is stored in two vectors in the same manner as double-precision matrices.

Logical Matrices

The logical data type represents a logical `true` or `false` state using the numbers `1` and `0`, respectively. Certain MATLAB functions and operators return logical `1` or logical `0` to indicate whether a certain condition was found to be true or not. For example, the statement `(5 * 10) > 40` returns a logical 1 value.

MATLAB Strings

MATLAB strings are of type `char` and are stored the same way as unsigned 16-bit integers except there is no imaginary data component. Unlike C, MATLAB strings are not null terminated.

Cell Arrays

Cell arrays are a collection of MATLAB arrays where each `mxArray` is referred to as a cell. This allows MATLAB arrays of different types to be stored together. Cell arrays are stored in a similar manner to numeric matrices, except the data portion contains a single vector of pointers to `mxArrays`. Members of this vector are called cells. Each cell can be of any supported data type, even another cell array.

Structures

A 1-by-1 structure is stored in the same manner as a 1-by-`n` cell array where `n` is the number of fields in the structure. Members of the data vector are called fields. Each field is associated with a name stored in the `mxArray`.

Objects

Objects are stored and accessed the same way as structures. In MATLAB, objects are named structures with registered methods. Outside MATLAB, an object is a structure that contains storage for an additional classname that identifies the name of the object.

Multidimensional Arrays

MATLAB arrays of any type can be multidimensional. A vector of integers is stored where each element is the size of the corresponding dimension. The storage of the data is the same as matrices.

Empty Arrays

MATLAB arrays of any type can be empty. An empty `mxArray` is one with at least one dimension equal to zero. For example, a double-precision `mxArray` of type `double`, where `m` and `n` equal 0 and `pr` is `NULL`, is an empty array.

Sparse Matrices

Sparse matrices have a different storage convention from that of full matrices in MATLAB. The parameters `pr` and `pi` are still arrays of double-precision numbers, but these arrays contain only nonzero data elements. There are three additional parameters: `nzmax`, `ir`, and `jc`.

- `nzmax` is an integer that contains the length of `ir`, `pr`, and, if it exists, `pi`. It is the maximum possible number of nonzero elements in the sparse matrix.
- `ir` points to an integer array of length `nzmax` containing the row indices of the corresponding elements in `pr` and `pi`.
- `jc` points to an integer array of length `n+1`, where `n` is the number of columns in the sparse matrix. The `jc` array contains column index information. If the `j`th column of the sparse matrix has any nonzero elements, `jc[j]` is the index in `ir` and `pr` (and `pi` if it exists) of the first nonzero element in the `j`th column, and `jc[j+1] - 1` is the index of the last nonzero element in that column. For the `j`th column of the sparse matrix, `jc[j]` is the total number of nonzero elements in all preceding columns. The last element of the `jc` array, `jc[n]`, is equal to `nnz`, the number of nonzero elements in the entire sparse matrix. If `nnz` is less than `nzmax`, more nonzero entries can be inserted into the array without allocating additional storage.

Using Data Types

You can write MEX-files, MAT-file applications, and engine applications in C that accept any data type supported by MATLAB. In Fortran, only the creation of double-precision n-by-m arrays and strings are supported. You can treat C and Fortran MEX-files, once compiled, exactly like M-functions.

Caution MATLAB does not check the validity of MATLAB data structures created in C or Fortran using one of the `mx` functions (e.g., `mxCreateStructArray`). Using invalid syntax to create a MATLAB data structure can result in unexpected behavior in your C or Fortran program.

The explore Example

There is an example MEX-file included with MATLAB, called `explore`, that identifies the data type of an input variable. The source file for this example is in the *`matlabroot`*`/extern/examples/mex` directory, where *`matlabroot`* represents the top-level directory where MATLAB is installed on your system.

Note In platform-independent discussions that refer to directory paths, this book uses the UNIX convention. For example, a general reference to the `mex` directory is *`matlabroot`*`/extern/examples/mex`.

For example, typing

```
cd([matlabroot '/extern/examples/mex']);
x = 2;
explore(x);
```

produces this result

```
------------------------------------------------
Name: prhs[0]
Dimensions: 1x1
Class Name: double
------------------------------------------------
 (1,1) = 2
```

`explore` accepts any data type. Try using `explore` with these examples.

```
explore([1 2 3 4 5])
explore 1 2 3 4 5
explore({1 2 3 4 5})
explore(int8([1 2 3 4 5]))
explore {1 2 3 4 5}
explore(sparse(eye(5)))
explore(struct('name', 'Joe Jones', 'ext', 7332))
explore(1, 2, 3, 4, 5)
```

Building MEX-Files

In this section...

Compiler Requirements

Your installed version of MATLAB contains all the tools you need to work with the API. MATLAB includes a C compiler for the PC called Lcc, but does not include a Fortran compiler. If you choose to use your own C compiler, it must be an ANSI C compiler. Also, if you are working on a Microsoft Windows platform, your compiler must be able to create 32-bit windows dynamically linked libraries (DLL files).

MATLAB supports many compilers and provides preconfigured files, called options files, designed specifically for these compilers. The purpose of supporting this large collection of compilers is to provide you with the flexibility to use the tool of your choice. However, in many cases, you simply can use the provided Lcc compiler with your C code to produce your applications.

The MathWorks maintains a list of compilers supported by MATLAB at the following location on the Web:

```
http://www.mathworks.com/support/tech-notes/1600/1601.shtml
```

Note The MathWorks provides an option called `-setup` for the `mex` script that lets you easily choose or switch your compiler.

The following sections contain configuration information for creating MEX-files on UNIX and Windows systems. More detailed information about the `mex` script is provided in "Custom Building MEX-Files" on page 3-19.

In addition, there is a section on "Troubleshooting" on page 3-32, if you are having difficulties creating MEX-files.

Testing Your Configuration on UNIX

The quickest way to check if your system is set up properly to create MEX-files is by trying the actual process. There is C source code for an example, `yprime.c`, and its Fortran counterpart, `yprimef.F` and `yprimefg.F`, included in the *`matlabroot`*`/extern/examples/mex` directory, where *`matlabroot`* represents the top-level directory where MATLAB is installed on your system.

To compile and link the example source files, `yprime.c` or `yprimef.F` and `yprimefg.F`, on UNIX, you must first copy the file(s) to a local directory, and then change directory (`cd`) to that local directory.

At the MATLAB prompt, type

```
mex yprime.c
```

This uses the system compiler to create the MEX-file called `yprime` with the appropriate extension for your system.

You can now call `yprime` as if it were an M-function:

```
yprime(1,1:4)
ans =
     2.0000   8.9685   4.0000    -1.0947
```

To try the Fortran version of the sample program with your Fortran compiler, at the MATLAB prompt, type

```
mex yprimef.F yprimefg.F
```

In addition to running the `mex` script from the MATLAB prompt, you can also run the script from the system prompt.

Selecting a Compiler

To change your default compiler, you select a different options file. You can do this anytime by using the command

```
mex -setup
```

```
    Using the 'mex -setup' command selects an options file that is
    placed in ~/matlab and used by default for 'mex'. An options
    file in the current working directory or specified on the
    command line overrides the default options file in ~/matlab.

    Options files control which compiler to use, the compiler and
    link command options, and the runtime libraries to link
    against.

    To override the default options file, use the 'mex -f' command
    (see 'mex -help' for more information).

The options files available for mex are:

  1: matlabroot/bin/gccopts.sh :
         Template Options file for building gcc MEXfiles

  2: matlabroot/bin/mexopts.sh :
         Template Options file for building MEXfiles using the
          system ANSI compiler

Enter the number of the options file to use as your default
options file:
```

Select the proper options file for your system by entering its number and pressing **Enter**. If an options file doesn't exist in your MATLAB directory, the system displays a message stating that the options file is being copied to your user-specific `matlab` directory. If an options file already exists in your `matlab` directory, the system prompts you to overwrite it.

Note The `-setup` option creates a user-specific `matlab` directory in your individual home directory and copies the appropriate options file to the directory. (If the directory already exists, a new one is not created.) This `matlab` directory is used for your individual options files only; each user can have his or her own default options files (other MATLAB products may place options files in this directory). Do not confuse these user-specific `matlab` directories with the system `matlab` directory, where MATLAB is installed. To see the name of this directory on your machine, use the MATLAB command `prefdir`.

Using the setup option resets your default compiler so that the new compiler is used every time you use the `mex` script.

Testing Your Configuration on Windows

Before you can create MEX-files on the Windows platform, you must configure the default options file, `mexopts.bat`, for your compiler. The `-setup` option provides an easy way for you to configure the default options file. To configure or change the options file at anytime, run

```
mex -setup
```

from either the MATLAB or DOS command prompt.

Lcc Compiler

MATLAB includes a C compiler called Lcc that you can use to create C MEX-files. Help on using the Lcc compiler is available in a help file that ships with MATLAB. To view this file, type in the MATLAB command window

```
!matlabroot\sys\lcc\bin\wedit.hlp
```

replacing the term *`matlabroot`* with the path to the directory in which MATLAB is installed on your system. (Type `matlabroot` in the Command Window to get the path for this directory.)

Selecting a Compiler

The `mex` script uses the Lcc compiler by default if you do not have a C or C++ compiler of your own already installed on your system and you try to compile a C MEX-file. If you need to compile Fortran programs, you must supply your own supported Fortran compiler.

The `mex` script uses the filename extension to determine the type of compiler to use for creating your MEX-files. For example,

```
mex test1.F
```

uses your Fortran compiler and

```
mex test2.c
```

uses your C compiler.

On Systems without a Compiler. If you do not have your own C or C++ compiler on your system, the `mex` script automatically configures itself for the included Lcc compiler. So, to create a C MEX-file on these systems, you can simply enter

```
mex filename.c
```

This simple method of creating MEX-files works for the majority of users.

If using the included Lcc compiler satisfies your needs, you can skip ahead in this section to "Building the MEX-File on Windows" on page 3-16.

On Systems with a Compiler. On systems where there is a C, C++, or Fortran compiler, you can select which compiler you want to use. Once you choose your compiler, that compiler becomes your default compiler and you no longer have to select one when you compile MEX-files. To select a compiler or change to existing default compiler, use `mex -setup`.

This example shows the process of setting your default compiler to the Microsoft Visual C++ Version 6.0 compiler:

```
mex -setup

Please choose your compiler for building external interface (MEX)
```

```
files.

Would you like mex to locate installed compilers [y]/n? n

Select a compiler:
[1] Compaq Visual Fortran version 6.6
[2] Lcc C version 2.4
[3] Microsoft Visual C/C++ version 6.0

[0] None

Compiler: 3

Your machine has a Microsoft Visual C/C++ compiler located at
D:\Applications\Microsoft Visual Studio. Do you want to use this
compiler [y]/n? y

Please verify your choices:

Compiler: Microsoft Visual C/C++ 6.0
Location: C:\Program Files\Microsoft Visual Studio

Are these correct?([y]/n): y

The default options file:
"C:\WINNT\Profiles\username\ApplicationData\MathWorks\MATLAB\R1
3\mexopts.bat" is being updated from ...
```

If the specified compiler cannot be located, you are given the message:

```
The default location for compiler-name is directory-name,
but that directory does not exist on this machine.
Use directory-name anyway [y]/n?
```

Using the `setup` option sets your default compiler so that the new compiler is used every time you use the `mex` script.

Building the MEX-File on Windows

There is example C source code, `yprime.c`, and its Fortran counterpart, `yprimef.F` and `yprimefg.F`, included in the

matlabroot\extern\examples\mex directory, where *matlabroot* represents the top-level directory where MATLAB is installed on your system.

To compile and link the example source file on Windows, at the MATLAB prompt, type

```
cd([matlabroot '\extern\examples\mex'])
mex yprime.c
```

This should create the MEX-file called `yprime` with the `.mexw32` extension, which corresponds to the 32-bit Windows platform.

You can now call `yprime` as if it were an M-function:

```
yprime(1,1:4)
ans =
     2.0000    8.9685    4.0000    -1.0947
```

To try the Fortran version of the sample program with your Fortran compiler, switch to your Fortran compiler using `mex -setup`. Then, at the MATLAB prompt, type

```
cd([matlabroot '\extern\examples\mex'])
mex yprimef.F yprimefg.F
```

In addition to running the `mex` script from the MATLAB prompt, you can also run the script from the system prompt.

Specifying an Options File

You can use the `-f` option to specify an options file on either UNIX or Windows. To use the `-f` option, at the MATLAB prompt type

```
mex filename -f <optionsfile>
```

and specify the name of the options file along with its pathname.

There are several situations when it may be necessary to specify an options file every time you use the `mex` script. These include

- *(Windows and UNIX)* You want to use a different compiler (and not use the `-setup` option), or you want to compile MAT or engine stand-alone programs.
- *(UNIX)* You do not want to use the system C compiler.

Preconfigured Options Files

MATLAB includes some preconfigured options files that you can use with particular compilers. The options files are located in the directory *`matlabroot`*`\bin\win32\mexopts` on Windows, *`matlabroot`*`\bin\win64\mexopts` on Windows x64, and *`matlabroot`*`/bin` on UNIX, where *`matlabroot`* stands for the MATLAB root directory as returned by the `matlabroot` command. On Windows and Windows x64, the options files are named `*.bat`, where `*` stands for the compiler type and version. On UNIX, the options file is named `*opts.sh`, where `*` stands for `mex` or a specific compiler.

For a list of all the compilers supported by MATLAB, see the following location on the Web:

```
http://www.mathworks.com/support/tech-notes/1600/1601.shtml
```

Note The next section, "Custom Building MEX-Files" on page 3-19, contains specific information on how to modify options files for particular systems.

Custom Building MEX-Files

In this section...

Who Should Read this Chapter

In general, the defaults that come with MATLAB should be sufficient for building most MEX-files. Following are reasons that you might need more detailed information:

- You want to use an Integrated Development Environment (IDE), rather than the provided script, to build MEX-files.
- You want to create a new options file, for example, to use a compiler that is not directly supported.
- You want to exercise more control over the build process than the script uses.

The script, in general, uses two stages (or three, for Microsoft Windows) to build MEX-files. These are the compile stage and the link stage. In between these two stages, Windows compilers must perform some additional steps to prepare for linking (the prelink stage).

MEX Script Switches

The `mex` script has a set of switches (also called **options**) that you can use to modify the link and compile stages. The MEX Script Switches table lists the available switches and their uses. Each switch is available on both UNIX and Windows unless otherwise noted.

For customizing the build process, you should modify the options file, which contains the compiler-specific flags corresponding to the general compile, prelink, and link steps required on your system. The options file consists of a series of variable assignments; each variable represents a different logical piece of the build process.

MEX Script Switches

Switch	Function
`@<rsp_file>`	(Windows only) Include the contents of the text file `<rsp_file>` as command-line arguments to `mex`.
`-<arch>`	Build an output file for architecture `<arch>`. To determine the value for `<arch>`, type `computer('arch')` at the MATLAB Command Prompt on the target machine. Valid values for `<arch>` depend on the architecture of the build platform.
`-ada <sfcn.ads>`	Use this option to compile a Simulink® S-function written in Ada, where `<sfcn.ads>` is the Package Specification for the S-function. When this option is specified, only the `-v` (verbose) and `-g` (debug) options are relevant. All other options are ignored. For examples and information on supported compilers and other requirements, see `README` in the `simulink/ada/examples` directory.
`-argcheck`	(C functions only) Add argument checking. This adds code so arguments passed incorrectly to MATLAB API functions cause assertion failures.
`-c`	Compile only. Creates an object file, but not a MEX-file.
`-compatibleArrayDims`	Build a MEX-file using the MATLAB Version 7.2 array-handling API, which limits arrays to 2^31-1 elements. This option is the default. (See also the `-largeArrayDims` option.)

MEX Script Switches (Continued)

Switch	Function
`-cxx`	(UNIX only) Use the C++ linker to link the MEX-file if the first source file is in C and there are one or more C++ source or object files. This option overrides the assumption that the first source file in the list determines which linker to use.
`-D<name>`	Define a symbol name to the C preprocessor. Equivalent to a `#define <name>` directive in the source.
`-D<name>=<value>`	Define a symbol name and value to the C preprocessor. Equivalent to a `#define <name> <value>` directive in the source.
`-f <optionsfile>`	Specify location and name of options file to use. Overrides the `mex` default-options-file search mechanism.
`-fortran`	(UNIX only) Specify that the gateway routine is in Fortran. This option overrides the assumption that the first source file in the list determines which linker to use.
`-g`	Create a MEX-file containing additional symbolic information for use in debugging. This option disables the `mex` default behavior of optimizing built object code (see the `-O` option).
`-h[elp]`	Print help for `mex`.
`-I<pathname>`	Add `<pathname>` to the list of directories to search for `#include` files.
`-inline`	Inline matrix accessor functions (`mx*`). The generated MEX-function may not be compatible with future versions of MATLAB.
`-l<name>`	Link with object library. On Windows, `<name>` expands to `<name>.lib` or `lib<name>.lib` and on UNIX to `lib<name>.so` or `lib<name>.dylib`.

MEX Script Switches (Continued)

Switch	Function
`-L<directory>`	Add `<directory>` to the list of directories to search for libraries specified with the `-l` option. On UNIX systems, you must also set the run-time library path, as explained in "Setting Run-Time Library Path" on page 1-16.
`-largeArrayDims`	Build a MEX-file using the MATLAB large-array-handling API. This API can handle arrays with more than 2^31–1 elements when compiled on 64-bit platforms. (See also the `-compatibleArrayDims` option.)
`-n`	No execute mode. Print any commands that `mex` would otherwise have executed, but do not actually execute any of them.
`-O`	Optimize the object code. Optimization is enabled by default and by including this option on the command line. If the `-g` option appears without the `-O` option, optimization is disabled.
`-outdir <dirname>`	Place all output files in directory `<dirname>`.
`-output <resultname>`	Create MEX-file named `<resultname>`. The appropriate MEX-file extension is automatically appended. Overrides the default MEX-file naming mechanism.
`-setup`	Interactively specify the compiler options file to use as the default for future invocations of `mex` by placing it in the user profile directory (returned by the `prefdir` command). When this option is specified, no other command-line input is accepted.
`-U<name>`	Remove any initial definition of the C preprocessor symbol `<name>`. (Inverse of the `-D` option.)

MEX Script Switches (Continued)

Switch	Function
`-v`	Verbose mode. Print the values for important internal variables after the options file is processed and all command-line arguments are considered. Prints each compile step and final link step fully evaluated.
`<name>=<value>`	Supplement or override an options file variable for variable `<name>`. This option is processed after the options file is processed and all command line arguments are considered.

Default Options File on UNIX

The default MEX options file provided with MATLAB is located in *`matlabroot`*`/bin`. The `mex` script searches for an options file called `mexopts.sh` in the following order:

- The current directory
- The directory specified by *`matlabroot`*`/bin`
- The directory returned by the `prefdir` function

`mex` uses the first occurrence of the options file it finds. If no options file is found, `mex` displays an error message. You can directly specify the name of the options file using the `-f` switch.

On UNIX, the options file is written in the Bourne shell script language.

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file in `fullfile(`*`matlabroot`*`, 'bin', 'mexopts.sh')`, or you can invoke the `mex` script in verbose mode (`-v`). Verbose mode prints the exact compiler options, prelink commands (if appropriate), and linker options used in the build process for each compiler. “Custom Building on UNIX” on page 3-24 gives an overview of the high-level build process.

Default Options File on Windows

The default MEX options file is placed in your `user profile` directory after you configure your system by running `mex -setup`. The `mex` script searches for an options file called `mexopts.bat` in the following order:

- The current directory
- The user profile directory (returned by the `prefdir` function)

`mex` uses the first occurrence of the options file it finds. If no options file is found, `mex` searches your machine for a supported C compiler and automatically configures itself to use that compiler. Also, during the configuration process, it copies the compiler's default options file to the `user profile` directory. If multiple compilers are found, you are prompted to select one.

On Windows, the options file is written in the Perl script language.

For specific information on the default settings for the MATLAB supported compilers, you can examine the options file, `mexopts.bat`, or you can invoke the `mex` script in verbose mode (`-v`). Verbose mode prints the exact compiler options, prelink commands, if appropriate, and linker options used in the build process for each compiler. "Custom Building on Windows" on page 3-27 gives an overview of the high-level build process.

The User Profile Directory

The Windows `user profile` directory is a directory that contains user-specific information such as desktop appearance, recently used files, and **Start** menu items. The `mex` and `mbuild` utilities store their respective options files, `mexopts.bat` and `compopts.bat`, which are created during the `-setup` process, in a subdirectory of your `user profile` directory, named `Application Data\MathWorks\MATLAB`.

Custom Building on UNIX

On UNIX systems, there are two stages in MEX-file building: compiling and linking.

Compile Stage

The compile stage must

- Add *matlabroot*/extern/include to the list of directories in which to find header files (-I*matlabroot*/extern/include).
- Define the preprocessor macro MATLAB_MEX_FILE (-DMATLAB_MEX_FILE).
- (C MEX-files only) Compile the source file, which contains version information for the MEX-file, *matlabroot*/extern/src/mexversion.c.

Link Stage

The link stage must

- Instruct the linker to build a shared library.
- If you link with your own libraries, set the run-time library path, which is explained in "Setting Run-Time Library Path" on page 1-16.
- Link all objects from compiled source files (including mexversion.c).
- (Fortran MEX-files only) Link in the precompiled versioning source file, *matlabroot*/extern/lib/$Arch/version4.o.
- Export the symbols mexFunction and mexVersion (these symbols represent functions called by MATLAB).

For Fortran MEX-files, the symbols are all lowercase and may have appended underscores. For specific information, invoke the mex script in verbose mode and examine the output.

Build Options

For customizing the build process, you should modify the options file. The options file contains the compiler-specific flags corresponding to the general steps outlined above. The options file consists of a series of variable assignments. Each variable represents a different logical piece of the build process. The options files provided with MATLAB are located in *matlabroot*/bin. The section "Default Options File on UNIX" on page 3-23, describes how the mex script looks for an options file.

To aid in providing flexibility, there are two sets of options in the options file that you can turn on and off with switches to the `mex` script. These sets of options correspond to building in *debug mode* and building in *optimization mode*. They are represented by the variables `DEBUGFLAGS` and `OPTIMFLAGS`, respectively, one pair for each *driver* that is invoked (`CDEBUGFLAGS` for the C compiler, `FDEBUGFLAGS` for the Fortran compiler, and `LDDEBUGFLAGS` for the linker; similarly for the `OPTIMFLAGS`):

- If you build in optimization mode (the default), the `mex` script includes the `OPTIMFLAGS` options in the compile and link stages.
- If you build in debug mode, the `mex` script includes the `DEBUGFLAGS` options in the compile and link stages. It does not include the `OPTIMFLAGS` options.
- You can include both sets of options by specifying both the optimization and debugging flags to the `mex` script (`-O` and `-g`, respectively).

Aside from these special variables, the `mex` options file defines the executable invoked for each of the three modes (C compile, Fortran compile, link) and the flags for each stage. You also can provide explicit lists of libraries that must be linked in to all MEX-files containing source files of each language.

The variable summary follows.

Variable	C Compiler	Fortran Compiler	Linker
Executable	`CC`	`FC`	`LD`
Flags	`CFLAGS`	`FFLAGS`	`LDFLAGS`
Optimization	`COPTIMFLAGS`	`FOPTIMFLAGS`	`LDOPTIMFLAGS`
Debugging	`CDEBUGFLAGS`	`FDEBUGFLAGS`	`LDDEBUGFLAGS`
Additional libraries	`CLIBS`	`FLIBS`	(none)

For specifics on the default settings for these variables, you can

- Examine the options file in *`matlabroot`*`/bin/mexopts.sh` (or the options file you are using), or

- Invoke the mex script in verbose mode.

Custom Building on Windows

There are three stages to MEX-file building for both C and Fortran on Windows: compiling, prelinking, and linking.

Compile Stage

For the compile stage, a mex options file must

- Set up paths to the compiler using the COMPILER (e.g., Watcom), PATH, INCLUDE, and LIB environment variables. If your compiler always has the environment variables set (e.g., in AUTOEXEC.BAT), you can comment them out in the options file.
- Define the name of the compiler, using the COMPILER environment variable, if needed.
- Define the compiler switches in the COMPFLAGS environment variable:
  - The switch to create a DLL is required for MEX-files.
  - For stand-alone programs, the switch to create an exe is required.
  - The -c switch (compile only; do not link) is recommended.
  - The switch to specify 8-byte alignment.
  - You can use any other switch specific to the environment.
- Define preprocessor macro, with -D, MATLAB_MEX_FILE is required.
- Set up optimizer switches and/or debug switches using OPTIMFLAGS and DEBUGFLAGS.
  - If you build in optimization mode (the default), the mex script includes the OPTIMFLAGS option in the compile stage.
  - If you build in debug mode, the mex script includes the DEBUGFLAGS options in the compile stage. It does not include the OPTIMFLAGS option.
  - You can include both sets of options by specifying both the optimization and debugging flags to the mex script (OPTIMFLAGS and DEBUGFLAGS, respectively).

Prelink Stage

The prelink stage dynamically creates import libraries to import the required function into the MEX, MAT, or engine file:

- All MEX-files link against `libmex.dll` (MEX library).
- MAT stand-alone programs link against `libmx.dll` (array access library) and `libmat.dll` (MAT-functions).
- Engine stand-alone programs link against `libmx.dll` (array access library) and `libeng.dll` for engine functions.

MATLAB and each DLL have corresponding `.def` files of the same names located in the *`matlabroot`*`\extern\include` directory.

Link Stage

For the link stage, a `mex` options file must

- Define the name of the linker in the `LINKER` environment variable.
- Define the `LINKFLAGS` environment variable that must contain
  - The switch to create a DLL for MEX-files, or the switch to create an `exe` for stand-alone programs.
  - Export of the entry point to the MEX-file as `mexFunction` for C or `MEXFUNCTION@16` for Fortran.
  - The import library (or libraries) created in the `PRELINK_CMDS` stage.
  - You can use any other link switch specific to the compiler.
- Set up the linking optimization and debugging switches `LINKOPTIMFLAGS` and `LINKDEBUGFLAGS`. Use the same conditions described in the “Compile Stage” on page 3-27.
- Define the link-file identifier in the `LINK_FILE` environment variable, if necessary. For example, Watcom uses `file` to identify that the name following is a file and not a command.
- Define the link-library identifier in the `LINK_LIB` environment variable, if necessary. For example, Watcom uses `library` to identify the name following is a library and not a command.

- Optionally, set up an output identifier and name with the output switch in the `NAME_OUTPUT` environment variable. The environment variable `MEX_NAME` contains the name of the first program in the command line. This must be set for `-output` to work. If this environment is not set, the compiler default is to use the name of the first program in the command line. Even if this is set, you can override it by specifying the `mex -output` switch.

Linking DLL Files to MEX-Files

To link a DLL to a MEX-file, list the DLL's `.lib` file on the command line.

Versioning MEX-Files

The `mex` script can build your MEX-file with a resource file that contains versioning and other essential information. The resource file is called `mexversion.rc` and resides in the `extern\include` directory. To support versioning, there are two new commands in the options files, `RC_COMPILER` and `RC_LINKER`, to provide the resource compiler and linker commands. It is assumed that

- If a compiler command is given, the compiled resource links to the MEX-file using the standard link command.
- If a linker command is given, the resource file links to the MEX-file after it is built using that command.

Compiling MEX-Files with the Microsoft Visual C++ IDE

Note This section provides information on how to compile MEX-files in the Microsoft Visual C++ (MSVC) IDE. It is not totally inclusive. This section assumes that you know how to use the IDE. If you need more information on using the MSVC IDE, refer to the corresponding Microsoft documentation.

To build MEX-files with the Microsoft Visual C++ integrated development environment:

1. Create a project and insert your MEX source files.

2. Add `mexversion.rc` from the MATLAB `include` directory, `matlab\extern\include`, to the project.

3. Create a `.def` file to export the MEX entry point. On the **Project** menu, click **Add New Item** and select **Module-Definition File (.def)**. For example:

```
LIBRARY MYFILE
EXPORTS mexFunction          <-- for a C MEX-file
   or
EXPORTS _MEXFUNCTION@16       <-- for a Fortran MEX-file
```

4. On the **Project** menu, click **Properties** for the project to open the property pages.

5. Under C/C++ General properties, add the MATLAB `include` directory, `matlab\extern\include`, as an additional include directory.

6. Under C/C++ Preprocessor properties, add `MATLAB_MEX_FILE` as a preprocessor definition.

7. Under Linker General properties, change the output file extension to `.mexw32` if you are building for a 32–bit platform or `.mexw64` if you are building for a 64–bit platform.

8. Locate the `.lib` files for the compiler you are using under *`matlabroot`*`\extern\lib\win32\microsoft` or *`matlabroot`*`\extern\lib\win64\microsoft`. Under Linker Input properties, add `libmx.lib`, `libmex.lib`, and `libmat.lib` as additional dependencies.

9. Under **Linker Input properties**, add the module definition (`.def`) file you created.

10. Under **Linker Debugging properties**, if you intend to debug the MEX-file using the IDE, specify that the build should generate debugging information. For more information about debugging, see “Debugging on Windows” on page 4-46.

If you are using a compiler other than the Microsoft Visual C/C++ compiler, the process for building MEX files is similar to that described above. In

step 4, locate the `.lib` files for the compiler you are using in a subdirectory of *`matlabroot`*`\extern\lib\win32` or *`matlabroot`*`\extern\lib\win64`. For example, if you are using the Borland C/C++ compiler, look in *`matlabroot`*`\extern\lib\win32\borland`.

Troubleshooting

In this section...

Configuration Issues

This section focuses on common problems that might occur when creating MEX-files.

Search Path Problem on Windows

Under Windows, if you move the MATLAB executable without reinstalling MATLAB, you may need to modify `mex.bat` to point to the new MATLAB location.

MATLAB Path Names Containing Spaces on Windows

If you have problems building MEX-files on Windows and there is a space in any of the directory names within the MATLAB path, either reinstall MATLAB into a path name that contains no spaces or rename the directory that contains the space. For example, if you install MATLAB under the `Program Files` directory, you may have difficulty building MEX-files with certain C compilers.

DLL Files Not on Path on Windows

MATLAB fails to load MEX-files if it cannot find all `.dll` files referenced by the MEX-file; the `.dll` files must be on the DOS path or in the same directory as the MEX-file. This is also true for third-party `.dll` files.

When this happens, MATLAB displays an error message of the following form:

```
??? Invalid MEX-file <mexfilename>:
The specified module could not be found.
```

On Windows, the third-party product Dependency Walker can be used to diagnose this problem. Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions that are exported by that module, and which of those functions are actually being called by other modules. You can download the Dependency Walker utility from the following Web site:

```
http://www.dependencywalker.com/
```

See the Technical Support solution 1-2RQL4L for information on using the Dependency Walker:

```
http://www.mathworks.com/support/solutions/data/1-2RQL4L.html
```

Internal Error When Using mex -setup (PC)

Some antivirus software packages may conflict with the `mex -setup` process or other `mex` commands. If you get an error message of the following form in response to a mex command,

```
mex.bat: internal error in sub get_compiler_info(): don't
recognize <string>
```

then you need to disable your antivirus software temporarily and reenter the command. After you have successfully run the `mex` script, you can reenable your antivirus software.

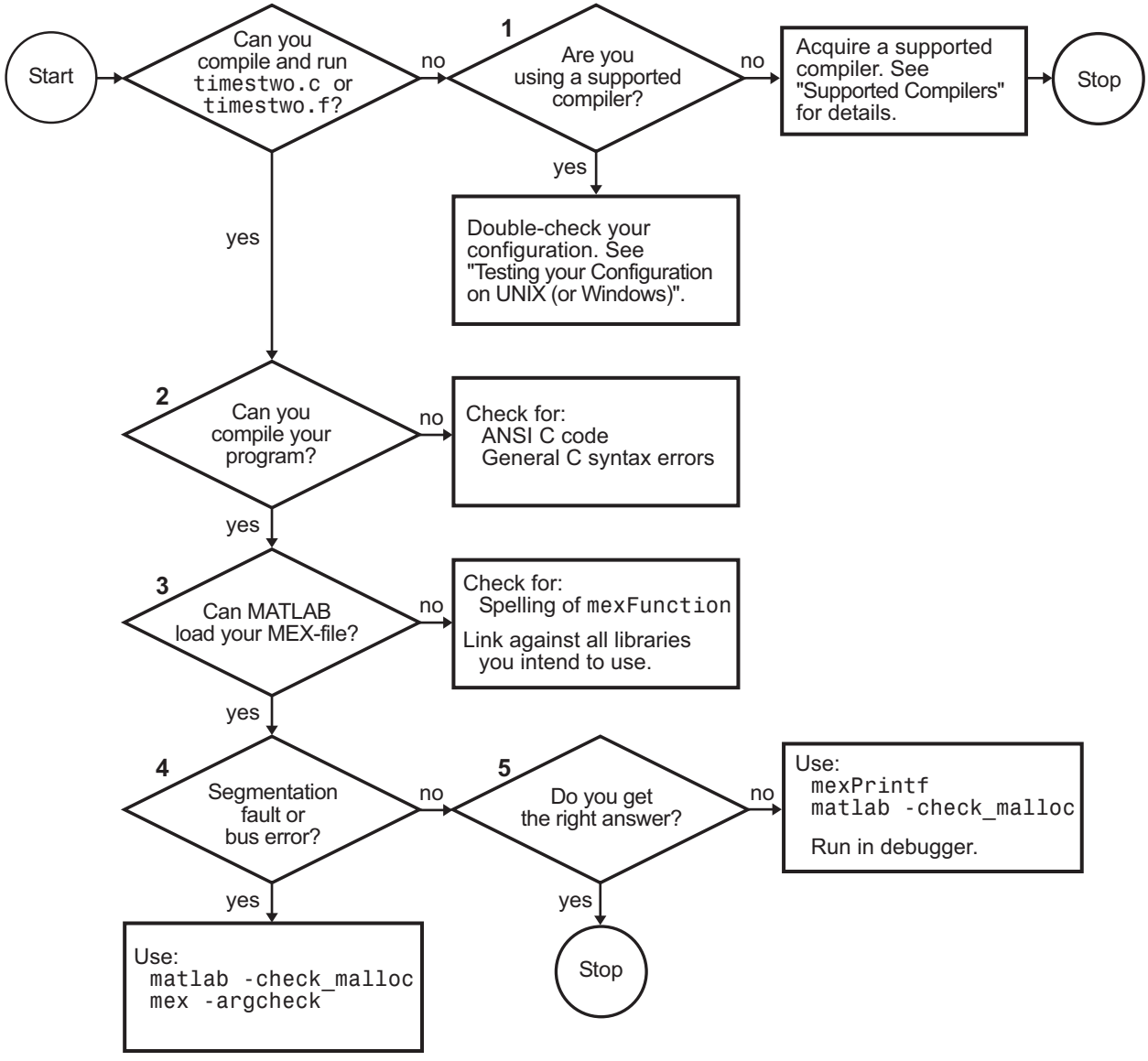
Alternatively, you can open a separate MS-DOS window and enter the `mex` command from that window.

General Configuration Problem

Make sure you followed the configuration steps for your platform described in this chapter. Also, refer to “Custom Building MEX-Files” on page 3-19 for additional information.

Understanding MEX-File Problems

This section contains information regarding common problems that occur when creating MEX-files. Use the following figure to help isolate these problems.



Troubleshooting MEX-File Creation Problems

Problems 1 through 5 refer to the corresponding numbered sections of the previous flowchart. For additional suggestions on resolving MEX-file build problems, access The MathWorks Technical Support Web site at:

```
http://www.mathworks.com/support
```

Problem 1 — Compiling a MathWorks Program Fails

The most common configuration problem in creating C MEX-files on UNIX involves using a non-ANSI C compiler, or failing to pass to the compiler a flag that tells it to compile ANSI C code.

A reliable way of knowing if you have this type of configuration problem is if the header files supplied by The MathWorks generate a string of syntax errors when you try to compile your code. See “Building MEX-Files” on page 3-11 for information on selecting the appropriate options file or, if necessary, obtain an ANSI C compiler.

Problem 2 — Compiling Your Own Program Fails

Mixing ANSI and non-ANSI C code can generate a string of syntax errors. The MathWorks provides header and source files that are ANSI C compliant. Therefore, your C code must also be ANSI compliant.

Other common problems that can occur in any C program are neglecting to include all necessary header files, or neglecting to link against all required libraries.

Make sure you are using a MATLAB supported compiler. See “Compiler Requirements” on page 3-11 for this information. Additional information can be found in “Compiler and Platform-Specific Issues” on page 3-37.

Problem 3 — MEX-File Load Errors

If you receive an error of the form

```
Unable to load mex file:
??? Invalid MEX-file
```

MATLAB does not recognize your MEX-file.

MATLAB loads MEX-files by looking for the gateway routine, `mexFunction`. If you misspell the function name, MATLAB cannot load your MEX-file and generates an error message. On Windows, check that you are exporting `mexFunction` correctly.

On some platforms, if you fail to link against required libraries, you may get an error when MATLAB loads your MEX-file rather than when you compile your MEX-file. In such cases, a system error message referring to *unresolved symbols* or *unresolved references* appears. Be sure to link against the library that defines the function in question.

On Windows, MATLAB fails to load MEX-files if it cannot find all `.dll` files referenced by the MEX-file; the `.dll` files must be on the path or in the same directory as the MEX-file. This is also true for third-party `.dll` files. See "DLL Files Not on Path on Windows" on page 3-32 for information to diagnose this problem.

Problem 4 — Segmentation Fault or Bus Error

If your MEX-file causes a segmentation violation or bus error, it means the MEX-file has attempted to access protected, read-only, or unallocated memory. Since this is such a general category of programming errors, such problems are sometimes difficult to track down.

Segmentation violations do not always occur at the same point as the logical errors that cause them. If a program writes data to an unintended section of memory, an error may not occur until the program reads and interprets the corrupted data. Consequently, a segmentation violation or bus error can occur after the MEX-file finishes executing.

MATLAB provides three features to help you troubleshoot problems of this nature. Listed in order of simplicity, they are as follows:

- Recompile your MEX-file with argument checking (C MEX-files only). You can add a layer of error checking to your MEX-file by recompiling with the `mex` script flag `-argcheck`. This warns you about invalid arguments to both MATLAB MEX-file (`mex`) and matrix access (`mx`) API functions.

  Although your MEX-file will not run as efficiently as it can, this switch detects errors such as passing `null` pointers to API functions.

- Run MATLAB with the `-check_malloc` option. The MATLAB startup flag, `-check_malloc`, indicates that MATLAB should maintain additional memory-checking information. When memory is freed, MATLAB checks to make sure that memory just before and just after this memory remains unwritten and that the memory has not been previously freed.

  If an error occurs, MATLAB reports the size of the allocated memory block. Using this information, you can track down where in your code this memory was allocated, and proceed accordingly.

  Although using this flag prevents MATLAB from running as efficiently as it can, it detects errors such as writing past the end of a dimensioned array, or freeing previously freed memory.

- Run MATLAB within a debugging environment. This process is already described in the chapters on creating C and Fortran MEX-files, respectively.

Problem 5 - Program Generates Incorrect Results

If your program generates the wrong answer(s), there are several possible causes. First, there could be an error in the computational logic. Second, the program could be reading from an uninitialized section of memory. For example, reading the 11th element of a 10-element vector yields unpredictable results.

Another cause of generating a wrong answer could be overwriting valid data due to memory mishandling. For example, writing to the 15th element of a 10-element vector might overwrite data in the adjacent variable in memory. This case can be handled in a similar manner as segmentation violations, as described in Problem 4.

In all of these cases, you can use `mexPrintf` to examine data values at intermediate stages or run MATLAB within a debugger to exploit all the tools the debugger provides.

Compiler and Platform-Specific Issues

This section describes situations specific to particular compilers and platforms.

Using MEX-Files from Other Sources

If you obtain a MEX-file from another source, be sure the file was compiled for the same platform on which you want to run it. See “What are MEX-Files” on page 3-2 for platform-specific information.

When you try to run a MEX-file from a version of MATLAB that is different from the version that created the MEX-file, MATLAB displays an error message of the following form:

```
??? Invalid MEX-file <mexfilename>:
The specified module could not be found.
```

Linux gcc Compiler Version Error

For information concerning a gcc compiler version error on Linux, see the Technical Support solution 1-2H64MF at:

```
http://www.mathworks.com/support/solutions/data/1-2H64MF.html
```

Fortran MEX-Files Compiler Errors

When you try to compile a Fortran MEX-file using a free source form format, MATLAB displays an error message of the following form:

```
Illegal character in statement label field
```

`mex` supports the fixed source form. The difference between free and fixed source forms is explained in the Fortran Language Reference Manual Source Forms topic. The URL for this topic is:

```
http://h21007.www2.hp.com/portal/download/files/unprot/Fortran/
docs/lrm/lrm0015.htm#source_formatmenu?&Record=383697&STASH=7
```

The URL for the Fortran Language Reference Manual is:

```
http://h21007.www2.hp.com/portal/download/files/unprot/Fortran/
docs/lrm/dflrm.htm
```

MEX-Files Created in Watcom IDE

If you use the Watcom IDE to create MEX-files and get unresolved references to API functions when linking against our libraries, check the argument-passing convention. The Watcom IDE uses a default switch that passes parameters in registers. MATLAB requires that you pass parameters on the stack.

Memory Management Compatibility Issues

MATLAB implicitly calls `mxDestroyArray`, the `mxArray` destructor, at the end of a MEX-file's execution on any `mxArrays` that are not returned in the left-hand side list (`plhs[]`). We recommend you review code in your MEX-files to avoid using these functions in the following situations. For additional information, see "Memory Management" on page 4-29 in Creating C Language MEX-Files.

Improperly Destroying an mxArray

You cannot use `mxFree` to destroy an `mxArray`.

Example. In the following example, `mxFree` does not destroy the array object. This operation frees the structure header associated with the array, but MATLAB stills operates as if the array object needs to be destroyed. Thus MATLAB tries to destroy the array object, and in the process, attempts to free its structure header again:

```
mxArray *temp = mxCreateDoubleMatrix(1,1,mxREAL);
      ...
mxFree(temp);  /* INCORRECT */
```

Solution. Call `mxDestroyArray` instead:

```
mxDestroyArray(temp);  /* CORRECT */
```

Incorrectly Constructing a Cell or Structure mxArray

You cannot call `mxSetCell` or `mxSetField` variants with `prhs[]` as the member array.

Example. In the following example, when the MEX-file returns, MATLAB destroys the entire cell array. Since this includes the members of the cell, this implicitly destroys the MEX-file's input arguments. This can cause several strange results, generally having to do with the corruption of the caller's workspace, if the right-hand side argument used is a temporary array (i.e., a literal or the result of an expression):

```
myfunction('hello')
/* myfunction is the name of your MEX-file and your code */
/* contains the following:    */

    mxArray *temp = mxCreateCellMatrix(1,1);
      ...
    mxSetCell(temp, 0, prhs[0]);  /* INCORRECT */
```

Solution. Make a copy of the right-hand side argument with `mxDuplicateArray` and use that copy as the argument to `mxSetCell` (or `mxSetField` variants). For example:

```
mxSetCell(temp, 0, mxDuplicateArray(prhs[0]));  /* CORRECT */
```

Creating a Temporary mxArray with Improper Data

You cannot call `mxDestroyArray` on an `mxArray` whose data was not allocated by an API routine.

Example. If you call `mxSetPr`, `mxSetPi`, `mxSetData`, or `mxSetImagData`, specifying memory that was not allocated by `mxCalloc`, `mxMalloc`, or `mxRealloc` as the intended data block (second argument), then when the MEX-file returns, MATLAB attempts to free the pointers to real data and imaginary data (if any). Thus MATLAB attempts to free memory, in this example, from the program stack.

```
mxArray *temp = mxCreateDoubleMatrix(0,0,mxREAL);
double data[5] = {1,2,3,4,5};
      ...
mxSetM(temp,1); mxSetN(temp,5); mxSetPr(temp, data);
/* INCORRECT */
```

Solution. Rather than use `mxSetPr` to set the data pointer, instead, create the `mxArray` with the right size and use `memcpy` to copy the stack data into the buffer returned by `mxGetPr`:

```
mxArray *temp = mxCreateDoubleMatrix(1,5,mxREAL);
double data[5] = {1,2,3,4,5};
      ...
memcpy(mxGetPr(temp), data, 5*sizeof(double)); /* CORRECT */
```

Potential Memory Leaks

Prior to Version 5.2, if you created an `mxArray` using one of the API creation routines and then you overwrote the pointer to the data using `mxSetPr`, MATLAB still freed the original memory. This is no longer the case.

For example,

```
pr = mxCalloc(5*5, sizeof(double));
... <load data into pr>
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
mxSetPr(plhs[0], pr);  /* INCORRECT */
```

will now leak 5*5*8 bytes of memory, where 8 bytes is the size of a `double`.

You can avoid that memory leak by changing the code

```
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
pr = mxGetPr(plhs[0]);
... <load data into pr>
```

or alternatively

```
pr = mxCalloc(5*5, sizeof(double));
... <load data into pr>
plhs[0] = mxCreateDoubleMatrix(5,5,mxREAL);
mxFree(mxGetPr(plhs[0]));
mxSetPr(plhs[0], pr);
```

Note that the first solution is more efficient.

Similar memory leaks can also occur when using `mxSetPi`, `mxSetData`, `mxSetImagData`, `mxSetIr`, or `mxSetJc`. You can avoid memory leaks by changing the code as described in this section.

MEX-Files Should Destroy Their Own Temporary Arrays

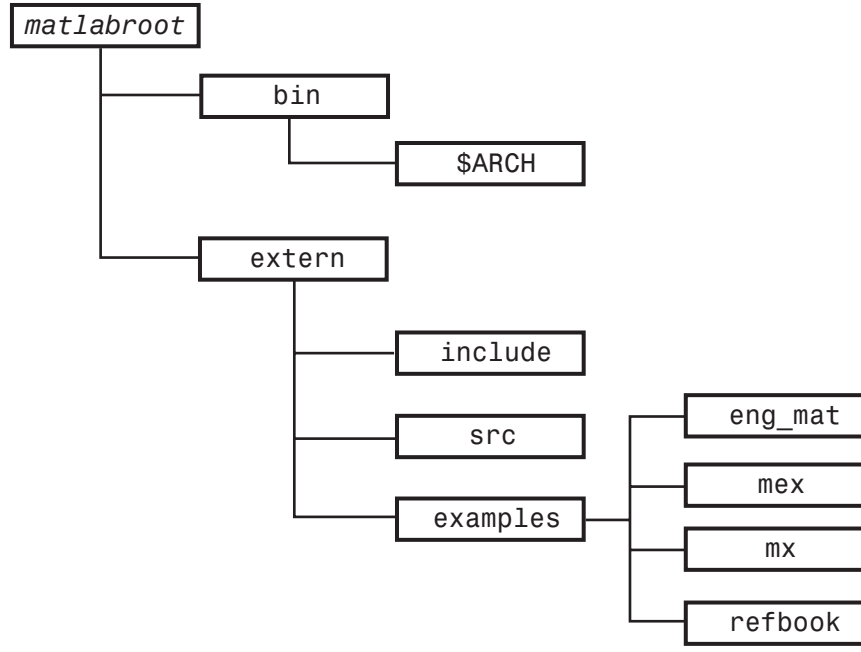
In general, we recommend that MEX-files destroy their own temporary arrays and clean up their own temporary memory. All `mxArrays` except those returned in the left-hand side list and those returned by `mexGetVariablePtr` may be safely destroyed. This approach is consistent with other MATLAB API applications (i.e., MAT-file applications, engine applications, and MATLAB Compiler generated applications, which do not have any automatic cleanup mechanism.)

Additional Information

In this section...

Files and Directories - UNIX Systems

This section describes the directory organization and purpose of the files associated with the MATLAB API on UNIX systems.



matlabroot/bin

The *matlabroot*/bin directory contains two files that are relevant for the MATLAB API:

mex
: UNIX shell script that creates MEX-files from C or Fortran MEX-file source code.

matlab
: UNIX shell script that initializes your environment and then invokes the MATLAB interpreter.

This directory also contains the preconfigured options files that the mex script uses with particular compilers. See "Preconfigured Options Files" on page 3-18 for more information.

matlabroot/bin/$ARCH

The *matlabroot*/bin/*$ARCH* directory contains libraries, where *$ARCH* specifies a particular UNIX platform. On some UNIX platforms, this directory contains two versions of this library. Library filenames ending with .so or .dylib are shared libraries.

matlabroot/extern/include

The *matlabroot*/extern/include directory contains the header files for developing C and C++ applications that interface with MATLAB.

The relevant header files for the MATLAB API are

engine.h
: Header file for MATLAB engine programs. Contains function prototypes for engine routines.

mat.h
: Header file for programs accessing MAT-files. Contains function prototypes for mat routines.

matrix.h
: Header file containing a definition of the mxArray structure and function prototypes for matrix access routines.

`mex.h`
: Header file for building MEX-files. Contains function prototypes for `mex` routines.

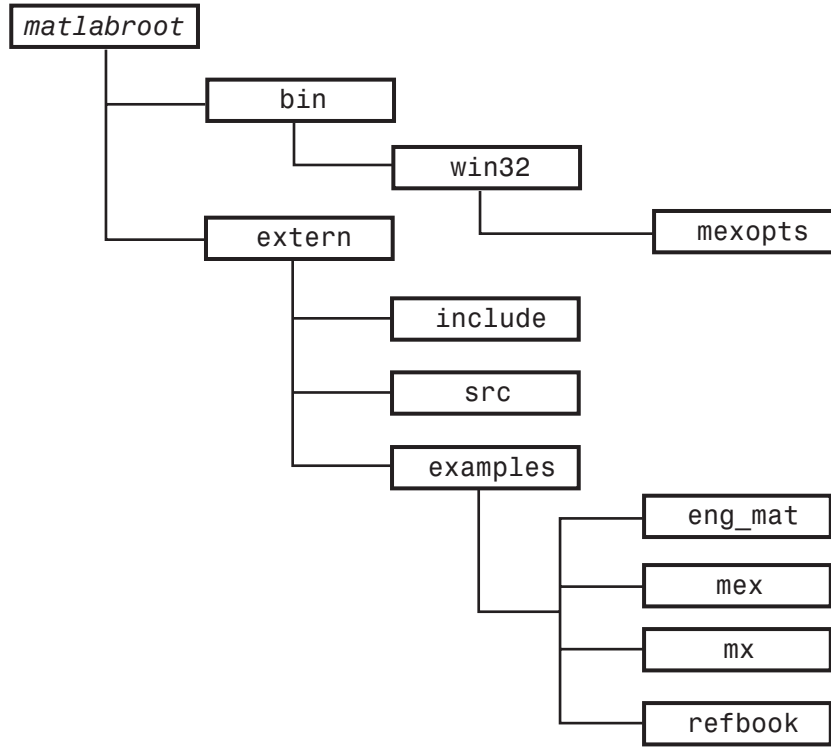
matlabroot/extern/src

The *`matlabroot`*`/extern/src` directory contains those C source files that are necessary to support certain MEX-file features such as argument checking and versioning.

Files and Directories — Windows Systems

This section describes the directory organization and purpose of the files associated with the MATLAB API on Microsoft Windows systems.

The following figure illustrates the directories in which the MATLAB API files are located. In the illustration, *matlabroot* symbolizes the top-level directory where MATLAB is installed on your system.



matlabroot\bin

The *matlabroot*\bin directory contains the mex.bat batch file that builds C and Fortran files into MEX-files. Also, this directory contains mex.pl, which is a Perl script used by mex.bat.

matlabroot\bin\win32\mexopts or matlabroot\bin\win64\mexopts

The *matlabroot*\bin\win32\mexopts or *matlabroot*\bin\win64\mexopts directory contains the preconfigured options files that the mex script uses

with particular compilers. See “Preconfigured Options Files” on page 3-18 for more information.

matlabroot\extern\include

The *matlabroot*\extern\include directory contains the header files for developing C and C++ applications that interface with MATLAB.

The relevant header files for the MATLAB API (MEX-files, engine, and MAT-files) are

`engine.h`
: Header file for MATLAB engine programs. Contains function prototypes for engine routines.

`mat.h`
: Header file for programs accessing MAT-files. Contains function prototypes for `mat` routines.

`matrix.h`
: Header file containing a definition of the `mxArray` structure and function prototypes for matrix access routines.

`mex.h`
: Header file for building MEX-files. Contains function prototypes for `mex` routines.

`_*.def`
: Files used by Borland compiler.

`*.def`
: Files used by MSVC and Microsoft Fortran compilers.

`mexversion.rc`
: Resource file for inserting versioning information into MEX-files.

matlabroot\extern\src

The *matlabroot*\extern\src directory contains files that are used for debugging MEX-files.

Examples

This book uses many examples to show how to write C and Fortran MEX-files.

Examples from the Text

The `refbook` subdirectory in the `extern/examples` directory contains the MEX-file examples (C and Fortran) that are used in this topic.

MEX Reference Examples

The `mex` subdirectory of `/extern/examples` directory contains MEX-file examples. It includes the examples described in the online C and Fortran API Reference for "MEX-Files" (the functions beginning with the `mex` prefix).

MX Examples

The `mx` subdirectory of `extern/examples` contains examples for using the array access functions. Although you can use these functions in stand-alone programs, most of these are MEX-file examples. The exception is `mxSetAllocFcns.c`, since this function is available only to stand-alone programs.

Engine and MAT Examples

The `eng_mat` subdirectory in the `extern/examples` directory contains the MEX-file examples (C and Fortran) for using the MATLAB engine facility, as well as examples for reading and writing MATLAB data files (MAT-files). These examples are all stand-alone programs.

Technical Support

The MathWorks provides additional Technical Support through its Web site. A few of the services provided are as follows:

- Solution Search Engine

This knowledge base on our Web site includes thousands of solutions and links to Technical Notes and is updated several times each week.

```
http://www.mathworks.com/support/
```

- Technical Notes

Technical notes are written by our Technical Support staff to address commonly asked questions.

```
http://www.mathworks.com/support/tech-notes/list_all.shtml
```

4

Creating C Language MEX-Files

This chapter describes how to write MEX-files in the C programming language. It discusses the MEX-file itself, how these C language files interact with MATLAB, how to pass and manipulate arguments of different data types, how to debug your MEX-file programs, and several other, more advanced topics.

C MEX-Files

In this section...

The Components of a C MEX-File

MEX-files are built by using the `mex` function. `mex` compiles and links source files into a shared library called a MEX-file, which you can run in MATLAB. Once compiled, you treat MEX-files exactly like MATLAB M-files and built-in functions.

The MEX-file consists of:

- A "Gateway Routine" on page 4-2 that interfaces C and MATLAB data.
- A "Computational Routine" on page 4-4 written in C that performs the computations you want implemented in the MEX-file.
- "Preprocessor Macros" on page 4-5 for building platform-independent code.

Gateway Routine

The *gateway routine* is the entry point to the MEX-file shared library. It is through this routine that MATLAB accesses the rest of the routines in your MEX-files. Use the following guideline to create a gateway routine:

- "Naming the Gateway Routine" on page 4-3
- "Required Parameters" on page 4-3
- "Creating and Using Source Files" on page 4-4
- "Using MATLAB Libraries" on page 4-4

A C MEX-file gateway routine looks like this:

```
void mexFunction(
    int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
{
   /* more C code ... */
}
```

Naming the Gateway Routine

The name of the gateway routine must be `mexFunction`.

Required Parameters

A gateway routine must contain the parameters `prhs`, `nrhs`, `plhs`, and `nlhs` which are described in the following table.

Parameter	Description
`prhs`	An array of right-hand input arguments.
`plhs`	An array of left-hand output arguments.
`nrhs`	The number of right-hand arguments, or the size of the `prhs` array.
`nlhs`	The number of left-hand arguments, or the size of the `plhs` array.

Both `prhs` and `plhs` are declared as type `mxArray *`, which means they point to MATLAB arrays. They are vectors that contain pointers to the arguments of the MEX-file.

You can think of the name `prhs` as representing the "parameters, right-hand side," that is, the input parameters. Likewise, `plhs` represents the "parameters, left-hand side," or output parameters.

Creating and Using Source Files

It is good practice to write the gateway routine to call a "Computational Routine" on page 4-4; however, this is not required. The computational code can be part of the gateway routine. If you use both gateway and computational routines, they can be combined in one source file or in separate files. If you use separate files, the gateway routine must be the first source file listed in the `mex` command.

The name of the file containing your gateway routine is important, as explained in "Naming the MEX-File" on page 4-4.

Using MATLAB Libraries

The MATLAB C and Fortran API Reference describes functions you can use in your gateway and computational routines that interact with MATLAB programs and the data in the MATLAB workspace. The `mx` prefixed functions provide access methods for manipulating MATLAB arrays. The `mex` prefixed functions perform operations in the MATLAB environment.

Required Header Files

To use the functions in the C and Fortran Reference library you must include the `mex` header, which declares the entry point and interface routines. Put this statement in your source file:

```
#include "mex.h"
```

Naming the MEX-File

The MEX-file name, and hence the name of the function you use in MATLAB, is the name of the source file containing your gateway routine.

The file extension of the MEX-file is platform-dependent. The `mexext` function returns the extension for the current machine.

Computational Routine

The *computational routine* contains the code for performing the computations you want implemented in the MEX-file. Computations can be numerical computations as well as inputting and outputting data. The gateway calls the computational routine as a subroutine.

The programming requirements described in "Creating and Using Source Files" on page 4-4, "Using MATLAB Libraries" on page 4-4, and "Required Header Files" on page 4-4 may also apply to your computational routine.

Preprocessor Macros

The MATLAB *preprocessor macros* `mwSize` and `mwIndex` are used in the `mx` and `mex` functions for cross-platform flexibility. `mwSize` represents size values, such as array dimensions and number of elements. `mwIndex` represents index values, such as indices into arrays.

Data Flow in MEX-Files

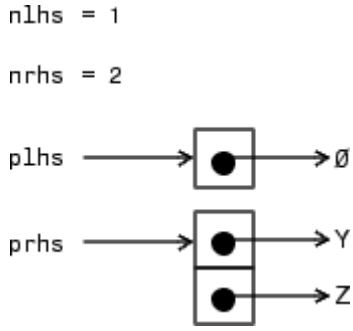
The following examples illustrate data flow in MEX-files:

Showing Data Input and Output

Suppose your MEX-file `myFunction` has 2 input arguments and 1 output argument. The MATLAB syntax is `[X] = myFunction(Y, Z)`. To call `myFunction` from MATLAB, type:

```
X = myFunction(Y, Z);
```

The MATLAB interpreter calls `mexFunction`, the gateway routine to `myFunction`, with the following arguments:



Your input is `prhs`, a 2-element C array (`nrhs = 2`). The first element is a pointer to an `mxArray` named `Y` and the second element is a pointer to an `mxArray` named `Z`.

Your output is `plhs`, a 1-element C array (`nlhs = 1`) where the single element is a `null` pointer. The parameter `plhs` points at nothing because the output `X` is not created until the subroutine executes.

The gateway routine creates the output array and sets a pointer to it in `plhs[0]`. If `plhs[0]` is left unassigned and you assign an output value to the function when you call it, MATLAB generates an error stating that no output was assigned.

Note It is possible to return an output value even if `nlhs = 0`. This corresponds to returning the result in the `ans` variable.

Gateway Routine Data Flow Diagram

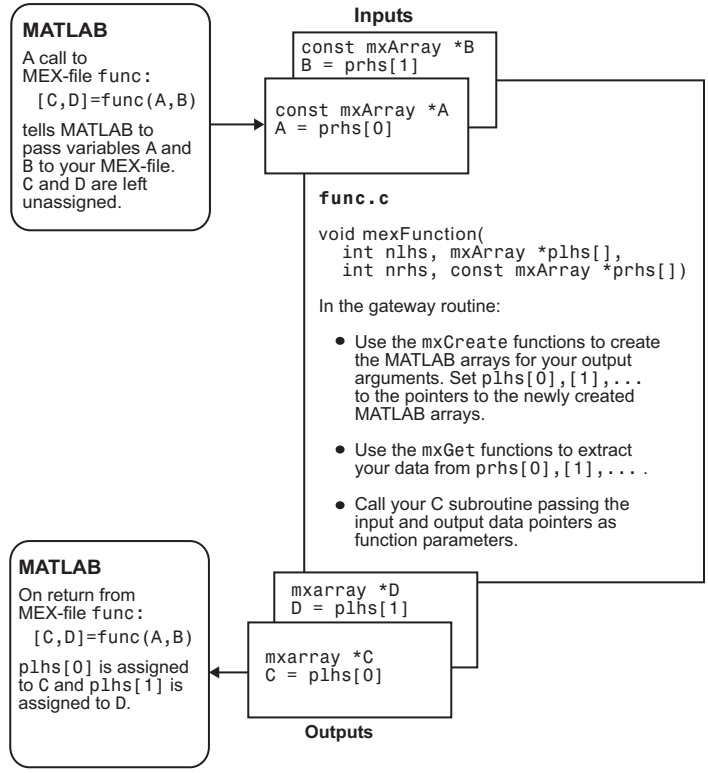
The following MEX Cycle diagram shows how inputs enter a MEX-file, what functions the gateway routine performs, and how outputs return to MATLAB.

In this example, the syntax of the MEX-file `func` is `[C, D] = func(A,B)`. In the figure, a call to `func` tells MATLAB to pass variables `A` and `B` to your MEX-file. `C` and `D` are left unassigned.

The gateway routine `func.c` uses the `mxCreate*` functions to create the MATLAB arrays for your output arguments. It sets `plhs[0]` and `plhs[1]`

to the pointers to the newly created MATLAB arrays. It uses the `mxGet*` functions to extract your data from your input arguments `prhs[0]` and `prhs[0]`. Finally, it calls your computational routine, passing the input and output data pointers as function parameters.

On return to MATLAB, `plhs[0]` is assigned to `C` and `plhs[1]` is assigned to `D`.



C MEX Cycle

MATLAB Example yprime.c

Let's look at an example, `yprime.c`, found in your *`matlabroot`*`/extern/examples/mex/` directory. ("Building MEX-Files" on page 3-11 explains how to create the MEX-file.) It's calling syntax is `[YP] = YPRIME(T,Y)`, where `T` is an integer and `Y` is a vector with 4 elements. For `T=1` and `Y=1:4`, when you type

```
yprime(T,Y)
```

MATLAB displays:

```
ans =
     2.0000    8.9685    4.0000    -1.0947
```

The gateway routine should validate the input arguments. This step includes checking the number, type, and size of the input arrays as well as examining the number of output arrays. If the inputs are not valid, call `mexErrMsgTxt`. For example,

`mexErrMsgTxt`

```
/* Check for proper number of arguments */
if (nrhs != 2) {
 mexErrMsgTxt("Two input arguments required.");
} else if (nlhs > 1) {
 mexErrMsgTxt("Too many output arguments.");
}

/* Check the dimensions of Y.  Y can be 4 X 1 or 1 X 4. */
m = mxGetM(Y_IN);
n = mxGetN(Y_IN);
if (!mxIsDouble(Y_IN) || mxIsComplex(Y_IN) ||
   (MAX(m,n) != 4) || (MIN(m,n) != 1)) {
 mexErrMsgTxt("YPRIME requires that Y be a 4 x 1 vector.");
}
```

To create MATLAB arrays, call any of the `mxCreate*` functions, like `mxCreateDoubleMatrix`, `mxCreateSparse`, or `mxCreateString`. If it needs them, the gateway routine can call `mxCalloc` to allocate temporary work arrays for the computational routine. In this example,

```
/* Create a matrix for the return argument */
plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
```

In the gateway routine, you access the data in `mxArray` and manipulate it in your computational subroutine. For example, the expression `mxGetPr(prhs[0])` returns a pointer of type `double *` to the real data in the

mxArray pointed to by prhs[0]. You can then use this pointer like any other pointer of type double * in C. For example,

```
/* Assign pointers to the various parameters */
yp = mxGetPr(plhs[0]);
```

In this example, a computational routine, yprime, performs the calculations:

```
/* Do the actual computations in a subroutine */
yprime(yp,t,y);
```

After calling your computational routine from the gateway, you can set a pointer of type mxArray to the data it returns. MATLAB recognizes the output from your computational routine as the output from the MEX-file.

When a MEX-file completes its task, it returns control to MATLAB. Any MATLAB arrays that are created by the MEX-file but are not returned to MATLAB through the left-hand side arguments are automatically destroyed.

Creating C++ MEX-Files

All C++ language standards are supported in MEX-files.

This section discusses specific C++ language issues to consider when creating and using MEX-files.

Creating Your C++ Source File

The C++ source code for the examples provided by MATLAB use the .cpp file extension. The extension .cpp is unambiguous and generally recognized by C++ compilers. Other possible extensions include .C, .cc, and .cxx.

For information on using C++ features, see Technical Note 1605, MEX-files Guide, at http://www.mathworks.com/support/tech-notes/1600/1605.html. Look for the sections under the “C++ Mex-files” heading.

Compiling and Linking

You can run a C++ MEX-file only on systems with the same version of MATLAB that the file was compiled on.

Use `mex setup` to select a C++ compiler, then type

```
mex filename.cpp
```

You can use command-line options, as shown in the “MEX Script Switches” on page 3-19 table.

Your link command must have all of the necessary DLL files that the MEX-function is dependent upon. To help you check for dependent files, see the Troubleshooting topic “DLL Files Not on Path on Windows” on page 3-32.

Examples

The examples “Using C++ Features in MEX-Files” on page 4-21 and “File Handling with C++” on page 4-22 illustrate the use of C++ by walking through source code examples available in your MATLAB directory.

Examples of C MEX-Files

In this section...

Introduction

The MATLAB API provides a full set of routines that handle the various data types supported by MATLAB. For each data type there is a specific set of functions that you can use for data manipulation. The first example discusses the simple case of doubling a scalar. After that, the examples discuss how to pass in, manipulate, and pass back various data types, and how to handle multiple inputs and outputs. Finally, the sections discuss passing and manipulating various MATLAB data types.

Note Source code for most examples in this chapter is available in the *matlabroot*`/extern/examples/refbook` directory of your MATLAB installation. *matlabroot* is your MATLAB root directory, the value returned by the `matlabroot` function.

A First Example — Passing a Scalar

Let's look at a simple example of C code and its MEX-file equivalent. Here is a C computational function that takes a scalar and doubles it.

```
#include <math.h>
void timestwo(double y[], double x[])
{
  y[0] = 2.0*x[0];
  return;
}
```

To see the same function written in the MEX-file format (`timestwo.c`), open the file in MATLAB Editor.

In C, function argument checking is done at compile time. In MATLAB, you can pass any number or type of arguments to your M-function, which is responsible for argument checking. This is also true for MEX-files. Your program must safely handle any number of input or output arguments of any supported type.

To compile and link this example source file at the MATLAB prompt, type

```
mex timestwo.c
```

This carries out the necessary steps to create the MEX-file called `timestwo` with an extension corresponding to the platform on which you're running. You can now call `timestwo` as if it were an M-function.

```
x = 2;
y = timestwo(x)
y =
     4
```

You can create and compile MEX-files in MATLAB or at your operating system's prompt. MATLAB uses `mex.m`, an M-file version of the `mex` script, and your operating system uses `mex.bat` on Windows and `mex.sh` on UNIX. In either case, typing

```
mex filename
```

at the prompt produces a compiled version of your MEX-file.

In the above example, scalars are viewed as 1-by-1 matrices. Alternatively, you can use a special API function called `mxGetScalar` that returns the values of scalars instead of pointers to copies of scalar variables (`timestwoalt.c`). To see the alternative code (error checking has been omitted for brevity), open the file in MATLAB Editor.

This example passes the input scalar `x` by value into the `timestwo_alt` subroutine, but passes the output scalar `y` by reference.

Passing Strings

Any MATLAB data type can be passed to and from MEX-files. The example `revord.c` accepts a string and returns the characters in reverse order. To see the example, open the file in MATLAB Editor.

In this example, the API function `mxCalloc` replaces `calloc`, the standard C function for dynamic memory allocation. `mxCalloc` allocates dynamic memory using the MATLAB memory manager and initializes it to zero. You must use `mxCalloc` in any situation where C would require the use of `calloc`. The same is true for `mxMalloc` and `mxRealloc`; use `mxMalloc` in any situation where C would require the use of `malloc` and use `mxRealloc` where C would require `realloc`.

Note MATLAB automatically frees up memory allocated with the `mx` allocation routines (`mxCalloc`, `mxMalloc`, `mxRealloc`) upon exiting your MEX-file. If you don't want this to happen, use the API function `mexMakeMemoryPersistent`.

The gateway routine `mexFunction` allocates memory for the input and output strings. Since these are C strings, they need to be one greater than the number of elements in the MATLAB string. Next the MATLAB string is copied to the input string. Both the input and output strings are passed to the computational subroutine (`revord`), which loads the output in reverse order. Note that the output buffer is a valid null-terminated C string because `mxCalloc` initializes the memory to 0. The API function `mxCreateString` then creates a MATLAB string from the C string, `output_buf`. Finally, `plhs[0]`, the left-hand side return argument to MATLAB, is set to the MATLAB array you just created.

By isolating variables of type `mxArray` from the computational subroutine, you can avoid having to make significant changes to your original C code.

In this example, typing

```
x = 'hello world';
y = revord(x)
```

produces

```
The string to convert is 'hello world'.
y =
dlrow olleh
```

Passing Two or More Inputs or Outputs

The `plhs[]` and `prhs[]` parameters are vectors that contain pointers to each left-hand side (output) variable and each right-hand side (input) variable, respectively. Accordingly, `plhs[0]` contains a pointer to the first left-hand side argument, `plhs[1]` contains a pointer to the second left-hand side argument, and so on. Likewise, `prhs[0]` contains a pointer to the first right-hand side argument, `prhs[1]` points to the second, and so on.

This example, `xtimesy`, multiplies an input scalar by an input scalar or matrix and outputs a matrix. For example, using `xtimesy` with two scalars gives

```
x = 7;
y = 7;
z = xtimesy(x,y)

z =
     49
```

Using `xtimesy` with a scalar and a matrix gives

```
x = 9;
y = ones(3);
z = xtimesy(x,y)

z =
     9     9     9
     9     9     9
     9     9     9
```

To see the corresponding MEX-file C code `xtimesy.c`, open the file in MATLAB Editor.

As this example shows, creating MEX-file gateways that handle multiple inputs and outputs is straightforward. All you need to do is keep track of which indices of the vectors `prhs` and `plhs` correspond to the input and output arguments of your function. In the example above, the input variable `x` corresponds to `prhs[0]` and the input variable `y` to `prhs[1]`.

Note that `mxGetScalar` returns the value of `x` rather than a pointer to `x`. This is just an alternative way of handling scalars. You could treat `x` as a 1-by-1 matrix and use `mxGetPr` to return a pointer to `x`.

Passing Structures and Cell Arrays

Passing structures and cell arrays into MEX-files is just like passing any other data types, except the data itself is of type `mxArray`. In practice, this means that `mxGetField` (for structures) and `mxGetCell` (for cell arrays) return pointers of type `mxArray`. You can then treat the pointers like any other pointers of type `mxArray`, but if you want to pass the data contained in the `mxArray` to a C routine, you must use an API function such as `mxGetData` to access it.

This example takes an m-by-n structure matrix as input and returns a new 1-by-1 structure that contains these fields:

- String input generates an m-by-n cell array

- Numeric input (noncomplex, scalar values) generates an m-by-n vector of numbers with the same class ID as the input, for example, `int`, `double`, and so on.

To see the program `phonebook.c`, open the file in MATLAB Editor.

To see how this program works, enter this structure.

```
friends(1).name = 'Jordan Robert';
friends(1).phone = 3386;
friends(2).name = 'Mary Smith';
friends(2).phone = 3912;
friends(3).name = 'Stacy Flora';
friends(3).phone = 3238;
friends(4).name = 'Harry Alpert';
friends(4).phone = 3077;
```

The results of this input are

```
phonebook(friends)

ans =
     name: {1x4 cell  }
    phone: [3386 3912 3238 3077]
```

Prompting User for Input

Because MATLAB does not use `stdin` and `stdout`, C functions like `scanf` and `printf` cannot be used to prompt users for input. The following example shows how to use `mexCallMATLAB` with the `input` function to get a number from the user.

```
#include "mex.h"
#include "string.h"
void mexFunction( int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[] )
{
  mxArray   *new_number, *str;
  double out;

  str = mxCreateString("Enter extension:  ");
```

```
  mexCallMATLAB(1,&new_number,1,&str,"input");
  out = mxGetScalar(new_number);
  mexPrintf("You entered: %.0f ", out);
  mxFree(new_number);
  mxFree(str);
  return;
 }
```

Handling Complex Data

Complex data from MATLAB is separated into real and imaginary parts. The MATLAB API provides two functions, `mxGetPr` and `mxGetPi`, that return pointers (of type `double *`) to the real and imaginary parts of your data.

This example, `convec.c`, takes two complex row vectors and convolves them. To see the example, open the file in MATLAB Editor.

Entering these numbers at the MATLAB prompt

```
x = [3.000 - 1.000i, 4.000 + 2.000i, 7.000 - 3.000i];
y = [8.000 - 6.000i, 12.000 + 16.000i, 40.000 - 42.000i];
```

and invoking the new MEX-file

```
z = convec(x,y)
```

results in

```
z =
   1.0e+02 *

Columns 1 through 4

0.1800 - 0.2600i 0.9600 + 0.2800i 1.3200 - 1.4400i 3.7600 - 0.1200i

Column 5

1.5400 - 4.1400i
```

which agrees with the results that the built-in MATLAB function `conv.m` produces.

Handling 8-,16-, and 32-Bit Data

You can create and manipulate signed and unsigned 8-, 16-, and 32-bit data from within your MEX-files. The MATLAB API provides a set of functions that support these data types. The API function `mxCreateNumericArray` constructs an unpopulated N-dimensional numeric array with a specified data size. Refer to the entry for `mxClassID` in the online reference pages for a discussion of how the MATLAB API represents these data types.

Once you have created an unpopulated MATLAB array of a specified data type, you can access the data using `mxGetData` and `mxGetImagData`. These two functions return pointers to the real and imaginary data. You can perform arithmetic on data of 8-, 16- or 32-bit precision in MEX-files and return the result to MATLAB, which will recognize the correct data class.

The example, `doubleelement.c`, constructs a 2-by-2 matrix with unsigned 16-bit integers, doubles each element, and returns both matrices to MATLAB. To see the example, open the file in MATLAB Editor.

At the MATLAB prompt, entering

```
doubleelement
```

produces

```
ans =
     2     6
     4     8
```

The output of this function is a 2-by-2 matrix populated with unsigned 16-bit integers.

Manipulating Multidimensional Numerical Arrays

You can manipulate multidimensional numerical arrays by using `mxGetData` and `mxGetImagData` to return pointers to the real and imaginary parts of the data stored in the original multidimensional array. The example, `findnz.c`, takes an N-dimensional array of doubles and returns the indices for the nonzero elements in the array. To see the example, open the file in MATLAB Editor.

Entering a sample matrix at the MATLAB prompt gives

```
matrix = [ 3 0 9 0; 0 8 2 4; 0 9 2 4; 3 0 9 3; 9 9 2 0]
matrix =
     3     0     9     0
     0     8     2     4
     0     9     2     4
     3     0     9     3
     9     9     2     0
```

This example determines the position of all nonzero elements in the matrix. Running the MEX-file on this matrix produces

```
nz = findnz(matrix)
nz =
     1     1
     4     1
     5     1
     2     2
     3     2
     5     2
     1     3
     2     3
     3     3
     4     3
     5     3
     2     4
     3     4
     4     4
```

Handling Sparse Arrays

The MATLAB API provides a set of functions that allow you to create and manipulate sparse arrays from within your MEX-files. These API routines access and manipulate `ir` and `jc`, two of the parameters associated with sparse arrays. For more information on how MATLAB stores sparse arrays, see “The MATLAB Array” on page 3-6.

The example, `fulltosparse.c`, illustrates how to populate a sparse matrix. To see the example, open the file in MATLAB Editor.

At the MATLAB prompt, entering

```
full = eye(5)
full =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

creates a full, 5-by-5 identity matrix. Using `fulltosparse` on the full matrix produces the corresponding sparse matrix.

```
spar = fulltosparse(full)
spar =
   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

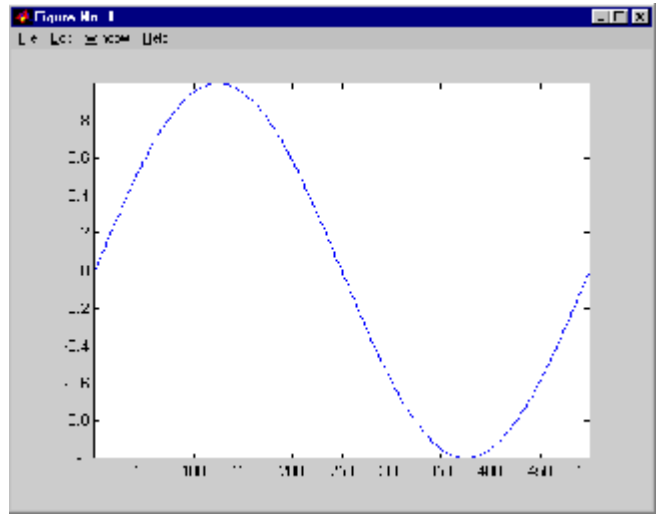
Calling Functions from C MEX-Files

It is possible to call MATLAB functions, operators, M-files, and other MEX-files from within your C source code by using the API function `mexCallMATLAB`. The example, `sincall.c`, creates an `mxArray`, passes various pointers to a subfunction to acquire data, and calls `mexCallMATLAB` to calculate the `sine` function and plot the results. To see the example, open the file in MATLAB Editor.

Running this example

```
sincall
```

displays the results



Note It is possible to generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. See the example below.

The following example creates an M-file that returns two variables but only assigns one of them a value.

```
function [a,b] = foo[c]
a = 2*c;
```

MATLAB displays the following warning message.

```
Warning: One or more output arguments not assigned during call to
'foo'.
```

If you then call `foo` using `mexCallMATLAB`, the unassigned output variable is now type `mxUNKNOWN_CLASS`.

Using C++ Features in MEX-Files

This example, `mexcpp.cpp`, illustrates how to use C++ code with your C language MEX-file. It makes use of member functions, constructors,

destructors, and the `iostream` include file. To see the example, open the file in MATLAB Editor.

The calling syntax is `mexcpp(num1, num2)`.

The routine defines a class, `MyData`, with member functions `display` and `set_data`, and variables `v1` and `v2`. It constructs an object `d` of class `MyData` and displays the initialized values of `v1` and `v2`. It then sets `v1` and `v2` to your input, `num1` and `num2`, and displays the new values. Finally, cleanup of the object is done using the `delete` operator.

File Handling with C++

This example, `mexatexit.cpp`, illustrates C++ file handling features. To see the C++ code, open the C++ file in MATLAB Editor. To compare it with a C code example `mexatexit.c`, open the C file in MATLAB Editor.

C Example

The C code example registers the `mexAtExit` function to perform cleanup tasks (close the data file) when the MEX-file clears. This example prints a message on the screen (using `mexPrintf`) when performing file operations `fopen`, `fprintf`, and `fclose`.

To build the `mexatexit.c` MEX-file, type:

```
mex mexatexit.c
```

If you type

```
x = 'my input string';
mexatexit(x)
```

MATLAB displays

```
Opening file matlab.data.
Writing data to file.
```

To clear the MEX-file, type

```
clear mexatexit
```

MATLAB displays

```
Closing file matlab.data.
```

You can see the contents of `matlab.data` by typing

```
type matlab.data
```

MATLAB displays

```
my input string
```

C++ Example

The C++ example does not use the `mexAtExit` function. The file open and close functions are handled in a `fileresource` class. The destructor for this class (which closes the data file) is automatically called when the MEX-file clears. This example also prints a message on the screen when performing operations on the data file. However, in this case, the only C file operation performed is the write operation, `fprintf`.

To build the `mexatexit.cpp` MEX-file, make sure you have selected a C++ compiler, then type:

```
mex mexatexit.cpp
```

If you type

```
z = 'for the C++ MEX-file';
mexatexit(x)
mexatexit(z)
clear mexatexit
```

MATLAB displays

```
Writing data to file.
Writing data to file.
```

To see the contents of `matlab.data`, type

```
type matlab.data
```

MATLAB displays

```
my input string
for the C++ MEX-file
```

Advanced Topics

In this section...

Help Files

Because the MATLAB interpreter chooses the MEX-file when both an M-file and a MEX-file with the same name are encountered in the same directory, it is possible to use M-files for documenting the behavior of your MEX-files. The MATLAB `help` command automatically finds and displays the appropriate M-file when help is requested and the interpreter finds and executes the corresponding MEX-file when the function is invoked.

Linking Multiple Files

It is possible to combine several object files and to use object file libraries when building MEX-files. To do so, simply list the additional files with their full extension, separated by spaces. For example, on the PC

```
mex circle.c square.obj rectangle.c shapes.lib
```

is a legal command that operates on the `.c`, `.obj`, and `.lib` files to create a MEX-file called `circle.mexw32`, where `mexw32` is the extension corresponding to the MEX-file type on 32-bit Windows. The name of the resulting MEX-file is taken from the first file in the list.

You may find it useful to use a software development tool like `MAKE` to manage MEX-file projects involving multiple source files. Simply create a `MAKEFILE` that contains a rule for producing object files from each of your source files

and then invoke `mex` to combine your object files into a MEX-file. This way you can ensure that your source files are recompiled only when necessary.

Workspace for MEX-File Functions

Unlike M-file functions, MEX-file functions do not have their own variable workspace. MEX-file functions operate in the caller's workspace.

`mexEvalString` evaluates the string in the caller's workspace. In addition, you can use the `mexGetVariable` and `mexPutVariable` routines to get and put variables into the caller's workspace.

Handling Large mxArrays

MEX-files built on 64-bit platforms can handle 64-bit mxArrays. These large data arrays can have up to 2^{48}–1 elements. The maximum number of elements a sparse mxArray can have is 2^{48}-2.

Using the following instructions creates platform-independent MEX-files as well.

Your system configuration can impact the performance of MATLAB. The 64-bit processor requirement enables you to create the mxArray and access data in it. However, your system's memory, in particular the size of RAM and virtual memory, determine the speed at which MATLAB processes the mxArray. The more memory available, the faster the processing.

The amount of RAM also limits the amount of data you can process at one time in MATLAB. For guidance on memory issues, see "Memory Allocation in MATLAB". Memory management within MEX-files can have special considerations, as described in "Memory Management" on page 4-29.

Using the 64-Bit API

To work with 64-bit mxArrays, your source code must comply with the 64-bit API, which consists of the functions in the following table.

`mxCalcSingleSubscript`	`mxCreateCellMatrix`
`mxCalloc`	`mxCreateCharArray`

mxCopyCharacterToPtr	mxCreateCharMatrixFromStrings
mxCopyComplex16ToPtr	mxCreateDoubleMatrix
mxCopyComplex8ToPtr	mxCreateLogicalArray
mxCopyInteger1ToPtr	mxCreateLogicalMatrix
mxCopyInteger2ToPtr	mxCreateNumericArray
mxCopyInteger4ToPtr	mxCreateNumericMatrix
mxCopyPtrToCharacter	mxCreateSparse
mxCopyPtrToComplex16	mxCreateSparseLogicalMatrix
mxCopyPtrToComplex8	mxCreateSparseLogicalMatrix
mxCopyPtrToInteger1	mxCreateStructMatrix
mxCopyPtrToInteger2	mxGetCell
mxCopyPtrToInteger4	mxGetElementSize
mxCopyPtrToPtrArray	mxGetField
mxCopyPtrToReal4	mxGetFieldByNumber
mxCopyPtrToReal8	mxGetIr
mxCopyReal4ToPtr	mxGetJc
mxCopyReal8ToPtr	mxGetM
mxCopyReal8ToPtr	mxGetN
mxCopyReal4ToPtr	mxGetNumberOfDimensions
mxCreateCellArray	mxGetNumberOfElements

Functions in this API use the `mwIndex` and `mwSize` types. For information about using these macros, see "Required Header Files" on page 4-4.

Building the MEX-File

Use the `mex` command option, `-largeArrayDims`, with the 64-bit API.

Example

The example, `arraySize.c` in *`matlabroot`*`/extern/examples/mex`, illustrates memory requirements of large mxArrays. To see the example, open the file in MATLAB Editor.

This function requires one positive scalar numeric input, which it uses to create a square matrix. It checks the size of the input to make sure your system can theoretically create a matrix of this size. If the input is valid, it displays the size of the mxArray in kilobytes.

To build this MEX-file, type:

```
mex -largeArrayDims arraySize.c
```

To run the MEX-file, type:

```
arraySize(2^10)
```

If your system has enough available memory, MATLAB displays:

```
Dimensions: 1024 x 1024
Size of array in kilobytes: 1024
```

If your system does not have enough memory to create the array, MATLAB displays an `Out of memory` error.

You can experiment with this function to test the performance and limits of handling large arrays on your system.

Caution Using Negative Values

When using the 64-bit API, `mwSize` and `mwIndex` are equivalent to `size_t` in C or `INTEGER*8` in Fortran. These types are unsigned, unlike `int` and `INTEGER*4`, which are the types used in the 32-bit API. Be careful not to pass any negative values to functions that take `mwSize` or `mwIndex` arguments. Do not cast negative `int` or `INTEGER*4` values to `mwSize` or `mwIndex`; the returned value can not be predicted. Instead, change your code to avoid using negative values.

Building Cross-Platform Applications

If you develop cross-platform applications (programs that can run on both 32- and 64-bit architectures), you must pay attention to the upper limit of values you use for `mwSize` and `mwIndex`. The 32-bit application reads these values and assigns them to variables declared as `int` in C or `INTEGER*4` in Fortran. Be careful to avoid assigning a large `mwSize` or `mwIndex` value to an `int`, `INTEGER*4`, or other variable that might be too small.

Memory Management

Memory management within MEX-files is not unlike memory management for regular C or Fortran applications. However, there are special considerations because the MEX-file must exist within the context of a larger application, i.e., MATLAB itself.

Automatic Cleanup of Temporary Arrays

When a MEX-file returns to MATLAB, it gives to MATLAB the results of its computations in the form of the left-hand side arguments — the `mxArrays` contained within the `plhs[]` list. Any `mxArrays` created by the MEX-file that are not in this list are automatically destroyed. In addition, any memory allocated with `mxCalloc`, `mxMalloc`, or `mxRealloc` during the MEX-file's execution is automatically freed.

In general, we recommend that MEX-files destroy their own temporary arrays and free their own dynamically allocated memory. It is more efficient for the MEX-file to perform this cleanup than to rely on the automatic mechanism. However, there are several circumstances in which the MEX-file does not reach its normal return statement.

The normal return is not reached if:

- A call to `mexErrMsgTxt` occurs.
- A call to `mexCallMATLAB` occurs and the function being called creates an error. (A MEX-file can trap such errors by using `mexSetTrapFlag`, but not all MEX-files necessarily need to trap errors.)
- The user interrupts the MEX-file's execution using **Ctrl+C**.

- The MEX-file runs out of memory. When this happens, the MATLAB out-of-memory handler immediately terminates the MEX-file.

A careful MEX-file programmer can ensure safe cleanup of all temporary arrays and memory before returning in the first two cases, but not in the last two cases. In the last two cases, the automatic cleanup mechanism is necessary to prevent memory leaks.

Persistent Arrays

You can exempt an array, or a piece of memory, from the MATLAB automatic cleanup by calling `mexMakeArrayPersistent` or `mexMakeMemoryPersistent`. However, if a MEX-file creates such persistent objects, there is a danger that a memory leak could occur if the MEX-file is cleared before the persistent object is properly destroyed. In order to prevent this from happening, a MEX-file that creates persistent objects should register a function, using `mexAtExit`, which disposes of the objects. (You can use a `mexAtExit` function to dispose of other resources as well; for example, you can use `mexAtExit` to close an open file.)

For example, here is a simple MEX-file that creates a persistent array and properly disposes of it.

```
#include "mex.h"

static int initialized = 0;
static mxArray *persistent_array_ptr = NULL;

void cleanup(void) {
    mexPrintf("MEX-file is terminating, destroying array\n");
    mxDestroyArray(persistent_array_ptr);
}

void mexFunction(int nlhs,
    mxArray *plhs[],
    int nrhs,
    const mxArray *prhs[])
{
  if (!initialized) {
    mexPrintf("MEX-file initializing, creating array\n");
```

```
    /* Create persistent array and register its cleanup. */
    persistent_array_ptr = mxCreateDoubleMatrix(1, 1, mxREAL);
    mexMakeArrayPersistent(persistent_array_ptr);
    mexAtExit(cleanup);
    initialized = 1;

    /* Set the data of the array to some interesting value. */
    *mxGetPr(persistent_array_ptr) = 1.0;
  } else {
    mexPrintf("MEX-file executing; value of first array
        element is %g\n", *mxGetPr(persistent_array_ptr));
  }
}
```

Hybrid Arrays

Functions such as `mxSetPr`, `mxSetData`, and `mxSetCell` allow the direct placement of memory pieces into an `mxArray`. `mxDestroyArray` destroys these pieces along with the entire array. Because of this, it is possible to create an array that cannot be destroyed, i.e., an array on which it is not safe to call `mxDestroyArray`. Such an array is called a *hybrid* array, because it contains both destroyable and nondestroyable components.

For example, it is not legal to call `mxFree` (or the ANSI `free()` function, for that matter) on automatic variables. Therefore, in the following code fragment, `pArray` is a hybrid array.

```
mxArray *pArray = mxCreateDoubleMatrix(0, 0, mxREAL);
double data[10];

mxSetPr(pArray, data);
mxSetM(pArray, 1);
mxSetN(pArray, 10);
```

Another example of a hybrid array is a cell array or structure, one of whose children is a read-only array (an array with the `const` qualifier, such as one of the inputs to the MEX-file). The array cannot be destroyed because the input to the MEX-file would also be destroyed.

Because hybrid arrays cannot be destroyed, they cannot be cleaned up by the automatic mechanism outlined in “Automatic Cleanup of Temporary Arrays” on page 4-29. As described in that section, the automatic cleanup mechanism is the only way to destroy temporary arrays in case of a user interrupt. Therefore, *temporary hybrid arrays are illegal* and can cause your MEX-file to crash. Although persistent hybrid arrays are viable, it is best to avoid using them whenever possible.

Large File I/O

MATLAB supports the use of 64-bit file I/O operations in your MEX-file programs. This enables you to read and write data to files that are up to and greater than 2 GB (2^{31-1} bytes) in size. Note that some operating systems or compilers might not support files larger than 2 GB.

This section covers the following topics on large file I/O:

Prerequisites to Using 64-Bit I/O

This section describes the components you need to use 64-bit file I/O in your MEX-file programs:

Header File. Header file `io64.h` defines many of the types and functions required for 64-bit file I/O. The statement to include this file must be the *first* `#include` statement in your source file and must also precede any system header include statements:

```
#include "io64.h"
#include "mex.h"
        .
        .
        .
```

Type Declarations. Use the following types to declare variables used in 64-bit file I/O.

MEX Type	Description	POSIX
`fpos_T`	Declares a 64-bit `int` type for `setFilePos()` and `getFilePos()`. Defined in `io64.h`.	`fpos_t`
`int64_T, uint64_T`	Declares 64-bit signed and unsigned integer types. Defined in `tmwtypes.h`.	`long, long`
`structStat`	Declares a structure to hold the size of a file. Defined in `io64.h`.	`struct stat`
`FMT64`	Used in `mexPrintf` to specify length within a format specifier such as `%d`. See example in the section "Printing Formatted Messages" on page 4-36. `FMT64` is defined in `tmwtypes.h`.	`%lld`
`LL, LLU`	Suffixes for literal `int` constant 64-bit values (C Standard ISO/IEC 9899:1999(E) Section 6.4.4.1). Used only on UNIX.	`LL, LLU`

Functions. Here are the functions you need for 64-bit file I/O. All are defined in the header file `io64.h`.

Function	Description	POSIX
`fileno()`	Gets a file descriptor from a file pointer	`fileno()`
`fopen()`	Opens the file and obtains the file pointer	`fopen()`
`getFileFstat()`	Gets the file size of a given file pointer	`fstat()`
`getFilePos()`	Gets the file position for the next I/O	`fgetpos()`
`getFileStat()`	Gets the file size of a given filename	`stat()`
`setFilePos()`	Sets the file position for the next I/O	`fsetpos()`

Specifying Constant Literal Values

To assign signed and unsigned 64-bit integer literal values, use type definitions `int64_T` and `uint64_T`.

On UNIX, to assign a literal value to an integer variable where the value to be assigned is greater than $2^{31}-1$ signed, you must suffix the value with `LL`. If the value is greater than $2^{32}-1$ unsigned, then use `LLU` as the suffix. These suffixes apply only to UNIX systems and are considered invalid on Windows systems.

Note The `LL` and `LLU` suffixes are not required for hardcoded (literal) values less than `2 G (2 `^{31}`-1)`, even if they are assigned to a 64-bit `int` type.

The following example declares a 64-bit integer variable initialized with a large literal `int` value, and two 64-bit integer variables:

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs,
```

```
                  const mxArray *prhs[])
{
#if defined(_MSC_VER) || defined(__BORLANDC__)     /* Windows */
   int64_T large_offset_example = 9000222000;
#else                                              /* UNIX    */
   int64_T large_offset_example = 9000222000LL;
#endif

int64_T offset   = 0;
int64_T position = 0;
```

Opening a File

To open a file for reading or writing, use the C `fopen` function as you normally would. As long as you have included `io64.h` at the start of your program, `fopen` works correctly for large files. No changes at all are required for `fread`, `fwrite`, `fprintf`, `fscanf`, and `fclose`.

To open an existing file for read and update in binary mode,

```
fp = fopen(filename, "r+b");
if (NULL == fp)
   {
   /* File does not exist. Create new file for writing
    * in binary mode.
    */
   fp = fopen(filename, "wb");
   if (NULL == fp)
      {
      sprintf(str, "Failed to open/create test file '%s'",
              filename);
      mexErrMsgTxt(str);
      return;
      }
   else
      {
      mexPrintf("New test file '%s' created\n",filename);
      }
   }
else mexPrintf("Existing test file '%s' opened\n",filename);
```

Printing Formatted Messages

You cannot print 64-bit integers using the `%d` conversion specifier. Instead, use `FMT64` to specify the appropriate format for your platform. `FMT64` is defined in the header file `tmwtypes.h`. The following example shows how to print a message showing the size of a large file:

```
int64_T large_offset_example = 9000222000LL;

mexPrintf("Example large file size: %" FMT64 "d bytes.\n",
           large_offset_example);
```

Replacing fseek and ftell with 64-Bit Functions

The ANSI C `fseek` and `ftell` functions are not 64-bit file I/O capable on most platforms. The functions `setFilePos` and `getFilePos`, however, are defined as the corresponding POSIX `fsetpos` and `fgetpos`, (or `fsetpos64` and `fgetpos64`), as required by your platform/OS. These functions are 64-bit file I/O capable on all platforms.

The following example shows how to use `setFilePos` instead of `fseek`, and `getFilePos` instead of `ftell`. It uses `getFileFstat` to find the size of the file, and then uses `setFilePos` to seek to the end of the file to prepare for adding data at the end of the file.

Note Although the `offset` parameter to `setFilePos` and `getFilePos` is really a pointer to a signed 64-bit integer, `int64_T`, it must be cast to an `fpos_T*`. The `fpos_T` type is defined in `io64.h` as the appropriate `fpos64_t` or `fpos_t`, as required by your platform/OS.

```
getFileFstat(fileno(fp), &statbuf);
fileSize = statbuf.st_size;
offset = fileSize;

setFilePos(fp, (fpos_T*) &offset);
getFilePos(fp, (fpos_T*) &position );
```

Unlike `fseek`, `setFilePos` supports only absolute seeking relative to the beginning of the file. If you want to do a relative seek, first call `getFileFstat`

to obtain the file size, and then convert the relative offset to an absolute offset that you can pass to `setFilePos`.

Determining the Size of an Open File

Getting the size of an open file involves two steps:

1. Refresh the record of the file size stored in memory using `getFilePos` and `setFilePos`.
2. Retrieve the size of the file using `getFileFstat`.

Refreshing the File Size Record. Before attempting to retrieve the size of an open file, you should first refresh the record of the file size residing in memory. If you skip this step on a file that is opened for writing, the file size returned might be incorrect or 0.

To refresh the file size record, seek to any offset in the file using `setFilePos`. If you do not want to change the position of the file pointer, you can seek to the current position in the file. This example obtains the current offset from the start of the file and then seeks to the current position to update the file size without moving the file pointer:

```
getFilePos( fp, (fpos_T*) &position);
setFilePos( fp, (fpos_T*) &position);
```

Getting the File Size. The `getFileFstat` function takes a file descriptor input argument (that you can obtain from the file pointer of the open file using `fileno`), and returns the size of that file in bytes in the `st_size` field of a `structStat` structure:

```
structStat statbuf;
int64_T fileSize = 0;

if (0 == getFileFstat(fileno(fp), &statbuf))
   {
   fileSize = statbuf.st_size;
   mexPrintf("File size is %" FMT64 "d bytes\n", fileSize);
   }
```

Determining the Size of a Closed File

The `getFileStat` function takes the filename of a closed file as an input argument, and returns the size of the file in bytes in the `st_size` field of a `structStat` structure:

```
structStat statbuf;
int64_T fileSize = 0;

if (0 == getFileStat(filename, &statbuf))
   {
   fileSize = statbuf.st_size;
   mexPrintf("File size is %" FMT64 "d bytes\n", fileSize);
   }
```

Using LAPACK and BLAS Functions

LAPACK is a large, multiauthor Fortran subroutine library that MATLAB uses for numerical linear algebra. BLAS, which stands for Basic Linear Algebra Subroutines, is used by MATLAB to speed up matrix multiplication and the LAPACK routines themselves. The functions provided by LAPACK and BLAS can also be called directly from within your C MEX-files.

This section explains how to write and build MEX-files that call LAPACK and BLAS functions. It provides information on

Specifying the Function Name

When calling an LAPACK or BLAS function, some platforms require an underscore character following the function name in the call statement.

On the PC platform use the function name alone, with no trailing underscore. For example, to call the LAPACK `dgemm` function, use

```
dgemm(arg1, arg2, ..., argn);
```

On the LINUX, Solaris, and Macintosh platforms, add the underscore after the function name. For example, to call `dgemm` on any of these platforms, use

```
dgemm_(arg1, arg2, ..., argn);
```

Calling LAPACK and BLAS Functions from C

Since the LAPACK and BLAS functions are written in Fortran, arguments passed to and from these functions must be passed by reference. The following example calls `dgemm`, passing all arguments by reference. An ampersand (&) precedes each argument unless that argument is already a reference.

```
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[])
{
  double *A, *B, *C, one = 1.0, zero = 0.0;
  int m,n,p;
  char *chn = "N";

  A = mxGetPr(prhs[0]);
  B = mxGetPr(prhs[1]);
  m = mxGetM(prhs[0]);
  p = mxGetN(prhs[0]);
  n = mxGetN(prhs[1]);

  if (p != mxGetM(prhs[1])) {mexErrMsgTxt
    ("Inner dimensions of matrix multiply do not match");
  }

  plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
```

```
  C = mxGetPr(plhs[0]);

  /* Pass all arguments to Fortran by reference */
  dgemm(chn, chn, &m, &n, &p, &one, A, &m, B, &p, &zero, C, &m);
}
```

Handling Complex Numbers

MATLAB stores complex numbers differently than Fortran. MATLAB stores the real and imaginary parts of a complex number in separate, equal length vectors, `pr` and `pi`. Fortran stores the same number in one location with the real and imaginary parts interleaved.

As a result, complex variables exchanged between MATLAB and the Fortran functions in LAPACK and BLAS are incompatible. MATLAB provides conversion routines that change the storage format of complex numbers to address this incompatibility.

Input Arguments. For all complex variables passed as input arguments to a Fortran function, you need to convert the storage of the MATLAB variable to be compatible with the Fortran function. Use the `mat2fort` function for this. See the example that follows.

Output Arguments. For all complex variables passed as output arguments to a Fortran function, you need to do the following:

1. When allocating storage for the complex variable, allocate a real variable with twice as much space as you would for a MATLAB variable of the same size. You need to do this because the returned variable uses the Fortran format, which takes twice the space. See the allocation of `zout` in the example that follows.

2. Once the variable is returned to MATLAB, convert its storage so that it is compatible with MATLAB. Use the `fort2mat` function for this.

Example — Passing Complex Variables. The example below shows how to call an LAPACK function from MATLAB, passing complex `prhs[0]` as input and receiving complex `plhs[0]` as output. Temporary variables `zin` and `zout` are used to hold `prhs[0]` and `plhs[0]` in Fortran format.

```
#include "mex.h"
```

```
#include "fort.h"      /* defines mat2fort and fort2mat */

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, mxArray
*prhs[])
{
  int lda, n;
  double *zin, *zout;
  lda = mxGetM(prhs[0]);
  n = mxGetN(prhs[0]);

  /* Convert input to Fortran format */
  zin = mat2fort(prhs[0], lda, n);

  /* Allocate a real, complex, lda-by-n variable to store output
*/
  zout = mxCalloc(2*lda*n, sizeof(double));

  /* Call complex LAPACK function */
  zlapack_function(zin, &lda, &n, zout);

  /* Convert output to MATLAB format */
  plhs[0] = fort2mat(zout, lda, lda, n);

  /* Free intermediate Fortran format arrays */
  mxFree(zin);
  mxFree(zout);
}
```

Preserving Input Values from Modification

Many LAPACK and BLAS functions modify the values of arguments passed in to them. It is advisable to make a copy of arguments that can be modified prior to passing them to the function. For complex inputs, this point is moot since the `mat2fort` version of the input is a new piece of memory, but for real data this is not the case.

The following example calls an LAPACK function that modifies the first input argument. The code in this example makes a copy of `prhs[0]`, and then passes the copy to the LAPACK function to preserve the contents of `prhs[0]`.

```
/* lapack_function modifies A so make a copy of the input */
m = mxGetM(prhs[0]);
n = mxGetN(prhs[0]);
A = mxCalloc(m*n, sizeof(double));

/* Copy mxGetPr(prhs[0]) into A */
temp = mxGetPr(prhs[0]);
for (k = 0; k < m*n; k++) {
   A[k] = temp[k];
   }

/* lapack_function does not modify B
/* so it is OK to use the input
directly */
B = mxGetPr(prhs[1]);
lapack_function(A, B);       /* modifies A but not B */

/* Free A when you are done with it */
mxFree(A);
```

Building the C MEX-File

The examples in this section show how to compile and link a C MEX file, `myCmexFile.c`, on the platforms supported by MATLAB.

Building on the PC. If you build your C MEX-file on a PC platform, you need to explicitly specify a library file to link with.

On the PC, use this command if you are using the Lcc compiler that ships with MATLAB:

```
mex myCmexFile.c
matlabroot\extern\lib\win32\lcc\libmwlapack.lib
matlabroot\extern\lib\win32\lcc\libmwblas.lib
```

Or, use this command if you are using Microsoft Visual C++ as your C compiler:

```
mex myCmexFile.c
matlabroot\extern\lib\win32\microsoft\libmwlapack.lib
matlabroot\extern\lib\win32\microsoft\libmwblas.lib
```

or

```
mex myCmexFile.c
matlabroot\extern\lib\win64\microsoft\libmwlapack.lib
matlabroot\extern\lib\win64\microsoft\libmwblas.lib
```

Building on Other Platforms. On all other platforms, you can build your MEX-file as you would any other C MEX-file. For example,

```
mex myCmexFile.c
```

MEX-Files That Perform Complex Number Conversion. MATLAB supplies the files `fort.c` and `fort.h`, which provide routines for conversion between MATLAB and FORTRAN complex data structures. These files define the `mat2fort` and `fort2mat` routines mentioned previously under "Handling Complex Numbers" on page 4-40.

If your program uses these routines, you need to:

1. Include the `fort.h` file in your program using `#include "fort.h"`. See "Example — Passing Complex Variables" on page 4-40.
2. Build the `fort.c` file with your program. Specify the path, *`matlabroot`*`/extern/examples/refbook` for both `fort.c` and `fort.h` in the build command.

On the PC, use either one of the following:

1.

```
mex myCmexFile.c matlabroot/extern/examples/refbook/fort.c
-Imatlabroot/extern/examples/refbook
matlabroot/extern/lib/win32/microsoft/libmwlapack.lib
matlabroot/extern/lib/win32/microsoft/libmwblas.lib
```

2.

```
mex myCmexFile.c matlabroot/extern/examples/refbook/fort.c
-Imatlabroot/extern/examples/refbook
matlabroot/extern/lib/win32/lcc/libmwlapack.lib
matlabroot/extern/lib/win32/lcc/libmwblas.lib
```

For all other platforms, use

```
mex myCmexFile.c matlabroot/extern/examples/refbook/fort.c
-Imatlabroot/extern/examples/refbook
```

Example — Symmetric Indefinite Factorization Using LAPACK

The directory *matlabroot*/extern/examples/refbook contains an example C MEX-file that calls two LAPACK functions. There are two versions of this file:

- utdu_slv.c - calls functions zhesvx and dsysvx, and thus is compatible with the PC platform.
- utdu_slv_.c - calls functions zhesvx_ and dsysvx_, and thus is compatible with the LINUX, Solaris, and Macintosh platforms.

Calling LAPACK and BLAS Functions from Fortran

You can make calls to the LAPACK and BLAS functions used by MATLAB from your Fortran MEX files. The following is an example program that takes two matrices and multiplies them by calling the LAPACK routine, dgemm:

```
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
integer plhs(*), prhs(*)
integer nlhs, nrhs
integer mxcreatedoublematrix, mxgetpr
integer mxgetm, mxgetn
integer m, n, p
integer A, B, C
double precision one, zero, ar, br
character ch1, ch2

ch1 = 'N'
ch2 = 'N'
one = 1.0
zero = 0.0

A = mxgetpr(prhs(1))
B = mxgetpr(prhs(2))
m = mxgetm(prhs(1))
p = mxgetn(prhs(1))
```

```
      n = mxgetn(prhs(2))

      plhs(1) = mxcreatedoublematrix(m, n, 0.0)
      C = mxgetpr(plhs(1))
      call mxcopyptrtoreal8(A, ar, 1)
      call mxcopyptrtoreal8(B, br, 1)

      call dgemm(ch1, ch2, m, n, p, one, %val(A), m,
     +           %val(B), p, zero, %val(C), m)

      return
      end
```

Building the Fortran MEX-File

The examples in this section show how to compile and link a Fortran MEX file, `myFortranmexFile.F`, on the platforms supported by MATLAB.

Building on the PC. On the PC, using Visual Fortran, link against the libraries `libdflapack.lib` and `libdfblas.lib`:

```
mex -v myFortranMexFile.F
matlabroot/extern/lib/win32/microsoft/libdflapack.lib
matlabroot/extern/lib/win32/microsoft/libdfblas.lib
```

Building on Other UNIX Platforms. On the UNIX platforms, create the MEX file as follows:

```
mex -v myFortranMexFile.F
```

Debugging C Language MEX-Files

In this section...

Notes on Debugging

The examples show how to debug `yprime.c`, found in your *`matlabroot`*`/extern/examples/mex/` directory.

MEX-files built with the `-g` option do not execute on other computers because they rely on files that are not distributed with MATLAB. Refer to the "Calling C and Fortran Programs from MATLAB" topic "Troubleshooting" on page 3-32 for additional information on isolating problems with MEX-files.

Debugging on Windows

The Microsoft Visual Studio development environment provides complete source code debugging, including the ability to set breakpoints, examine variables, and step through the source code line-by-line.

For information on debugging MEX-files compiled with other MATLAB supported compilers, see Technical Note 1605, MEX-files Guide, at `http://www.mathworks.com/support/tech-notes/1600/1605.html`.

Microsoft Visual Studio 2005

This section describes how to debug using the default compiler, that is, the compiler used to build MATLAB.

1 Select the Microsoft Visual C++ 2005 compiler. At the MATLAB prompt, type

```
mex -setup
```

Type `y` to locate installed compilers, and then type the number corresponding to this compiler.

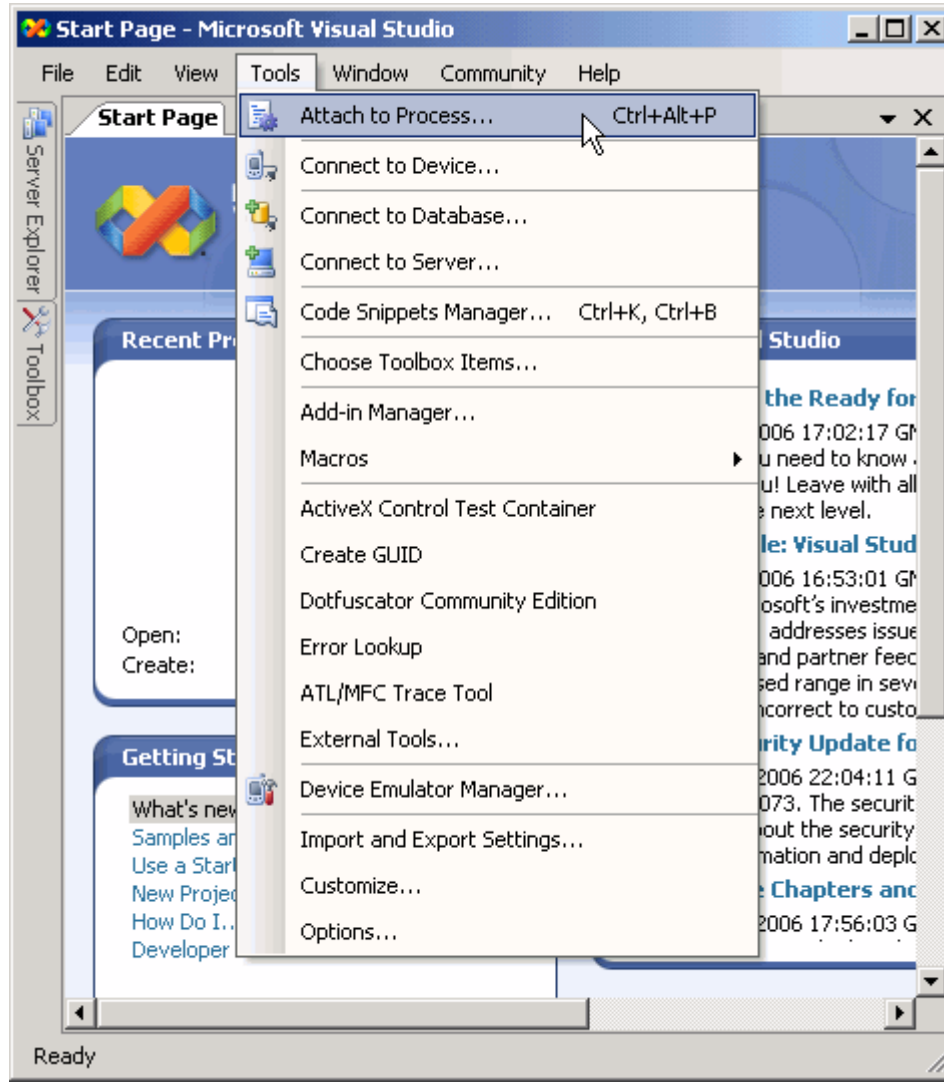
2 Next, compile the MEX-file with the `-g` option, which builds the file with debugging symbols included. For example

```
mex -g yprime.c
```

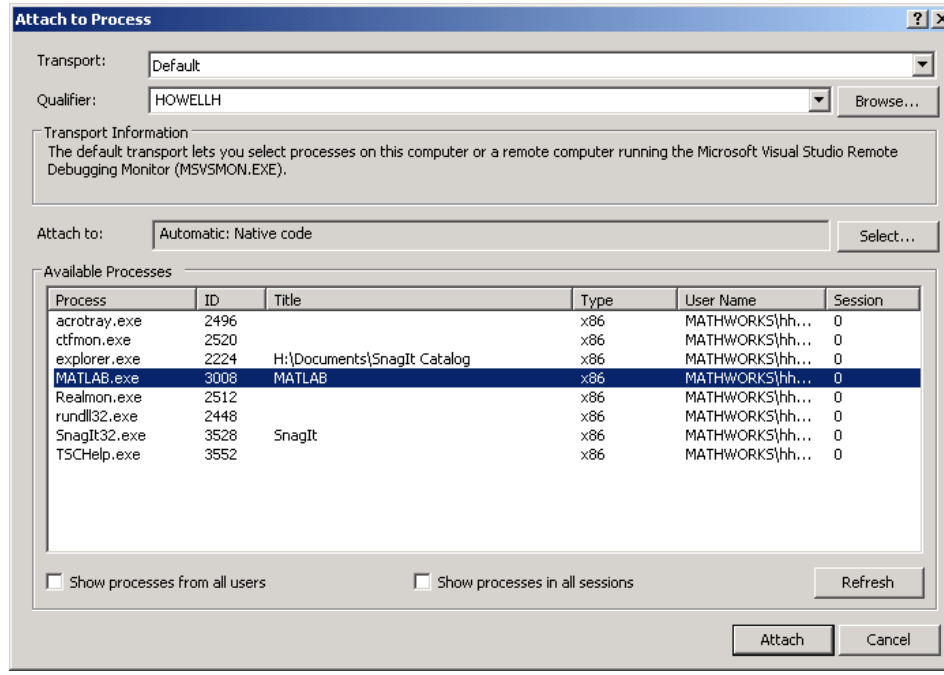
On a 32–bit platform, this command creates the executable file `yprime.mexw32`.

3 Start Visual Studio. Do not exit your MATLAB session.

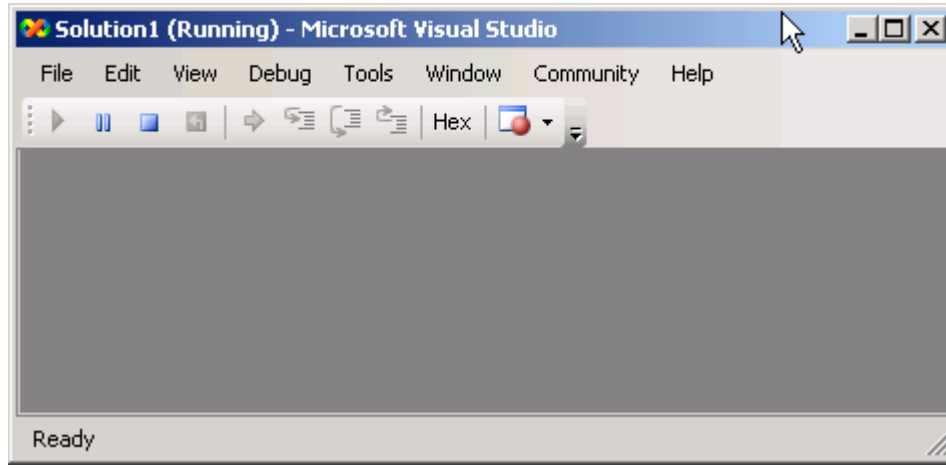
4 From the Visual Studio **Tools** menu, select **Attach to Process...**



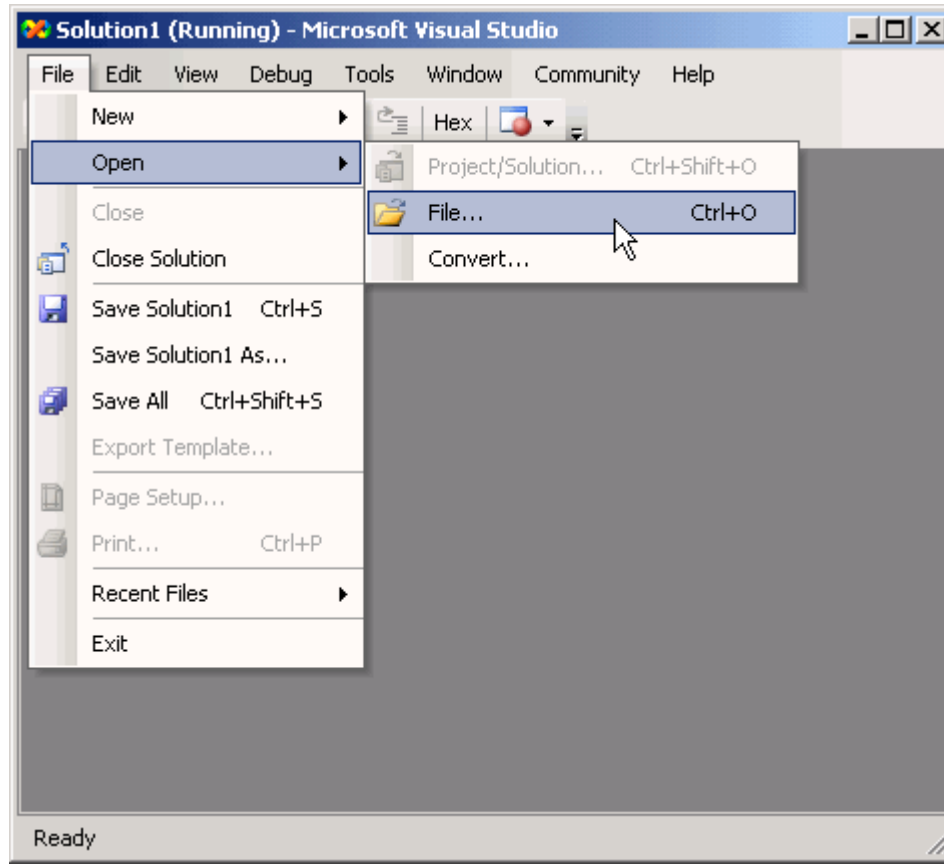
5 In the Attach to Process dialog box, select the MATLAB process and click **Attach**.



Visual Studio loads data then displays an empty code pane.



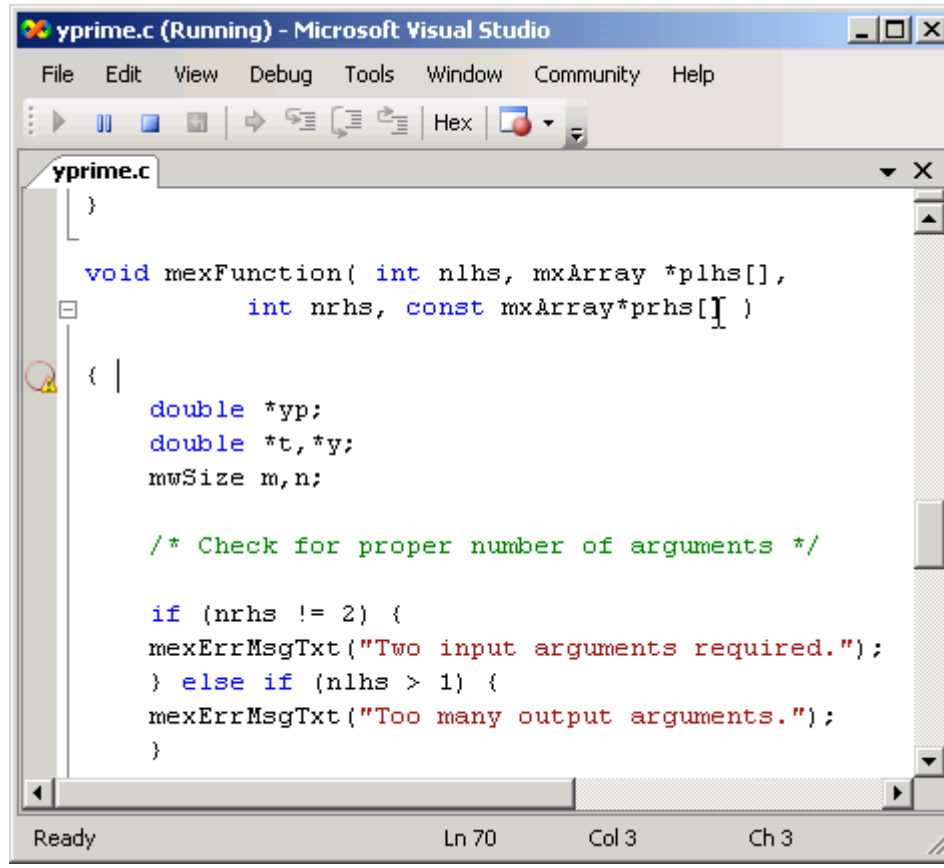
6 Open the source file `yprime.c` by selecting **File > Open > File**. `yprime.c` is found in the *`matlabroot`*`/extern/examples/mex/` directory.



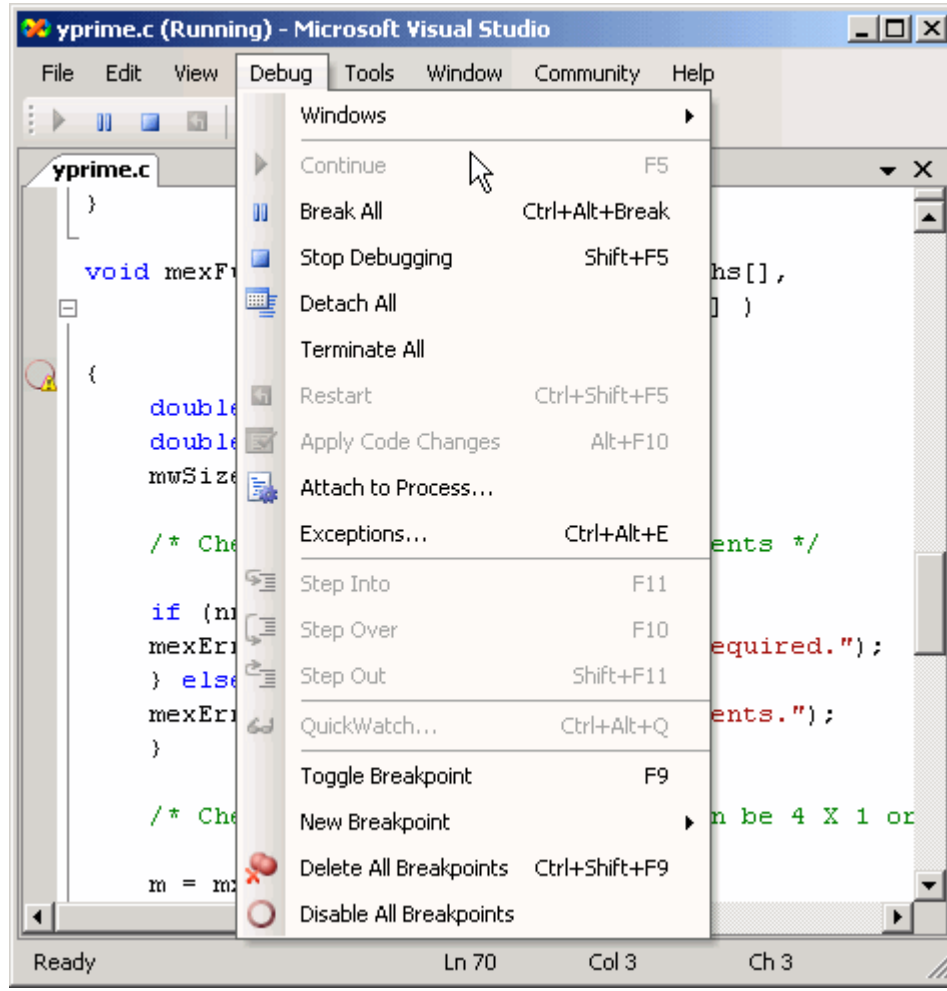
7 Set a breakpoint by right-clicking the desired line of code and following **Breakpoint > Insert Breakpoint** on the context menu. It is often

convenient to set a breakpoint at `mexFunction` in order to stop at the beginning of the gateway routine.

If you have not yet run the executable file, ignore any "`!`" icon that appears with the breakpoint next to the line of code.



Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.



8 Run the MEX-file in MATLAB. After typing

```
yprime(1,1:4)
```

`yprime.c` is opened in the Visual Studio debugger at the first breakpoint.

9 If you select **Debug > Continue**, MATLAB displays

```
ans =

    2.0000    8.9685    4.0000   -1.0947
```

For more information on how to debug in the Visual Studio environment, see the Microsoft documentation.

Debugging on Linux

The GNU Debugger gdb, available on Linux, provides complete source code debugging, including the ability to set breakpoints, examine variables, and step through the source code line-by-line.

For information on debugging MEX-files compiled with other MATLAB supported compilers, see Technical Note 1605, MEX-files Guide, at `http://www.mathworks.com/support/tech-notes/1600/1605.html`.

GNU Debugger gdb

In this procedure, the MATLAB command prompt >> is shown in front of MATLAB commands, and `linux>` represents a Linux prompt; your system may show a different prompt. The debugger prompt is `<gdb>`.

To debug with gdb,

1 Compile the MEX-file with the `-g` option, which builds the file with debugging symbols included. For this example, at the Linux prompt, type

```
linux> mex -g yprime.c
```

On a Linux 32–bit platform, this command creates the executable file `yprime.mexglx`.

2 At the Linux prompt, start the gdb debugger using the `matlab` function `-D` option:

```
linux> matlab -Dgdb
```

3 Start MATLAB without the Java virtual machine (JVM) by using the `-nojvm` startup flag:

```
<gdb> run -nojvm
```

4 In MATLAB, enable debugging with the `dbmex` function and run your MEX-file:

```
>> dbmex on
>> yprime(1,1:4)
```

5 At this point, you are ready to start debugging.

It is often convenient to set a breakpoint at `mexFunction` so you stop at the beginning of the gateway routine.

```
<gdb> break mexFunction
<gdb> continue
```

6 Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

To proceed from a breakpoint, type

```
<gdb> continue
```

7 After stopping at the last breakpoint, type

```
<gdb> continue
```

`yprime` finishes and MATLAB displays

```
ans =

    2.0000    8.9685    4.0000   -1.0947
```

8 From the MATLAB prompt you can return control to the debugger by typing

```
>> dbmex stop
```

Or, if you are finished running MATLAB, type

```
>> quit
```

9 When you are finished with the debugger, type

```
<gdb> quit
```

You return to the Linux prompt.

Refer to the documentation provided with your debugger for more information on its use.

5

Creating Fortran MEX-Files

Fortran MEX-Files

In this section...

The Components of a Fortran MEX-File

MEX-files are built by using the `mex` function. `mex` compiles and links source files into a shared library called a MEX-file, which you can run in MATLAB. Once compiled, you treat MEX-files exactly like MATLAB M-files and built-in functions.

The MEX-file consists of:

- A "Gateway Routine" on page 5-2 that interfaces Fortran and MATLAB data.
- A "Computational Routine" on page 5-5 that performs the computations you want implemented in the MEX-file.
- "Preprocessor Macros" on page 5-5 for building platform-independent code.

Gateway Routine

The *gateway routine* is the entry point to the MEX-file shared library. It is through this routine that MATLAB accesses the rest of the routines in your MEX-files. Use the following guideline to create a gateway routine:

- "Naming the Gateway Routine" on page 5-3
- "Required Parameters" on page 5-3
- "Creating and Using Source Files" on page 5-4

A Fortran MEX-file gateway routine looks like this:

```
C     The gateway routine.
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      integer nlhs, nrhs
      mwpointer plhs(*), prhs(*)
```

Note Fortran is not case sensitive. This document uses mixed-case function names for ease of reading.

Naming the Gateway Routine

The name of the gateway routine must be `mexFunction`.

Required Parameters

A gateway routine must contain the parameters `prhs`, `nrhs`, `plhs`, and `nlhs` which are described in the following table.

Parameter	Description
`prhs`	An array of right-hand input arguments.
`plhs`	An array of left-hand output arguments.
`nrhs`	The number of right-hand arguments, or the size of the `prhs` array.
`nlhs`	The number of left-hand arguments, or the size of the `plhs` array.

Both `prhs` and `plhs` are declared as type `mxArray *`, which means they point to MATLAB arrays. They are vectors that contain pointers to the arguments of the MEX-file.

You can think of the name `prhs` as representing the "parameters, right-hand side," that is, the input parameters. Likewise, `plhs` represents the "parameters, left-hand side," or output parameters.

Creating and Using Source Files

It is good practice to write the gateway routine to call a "Computational Routine" on page 4-4; however, this is not required. The computational code can be part of the gateway routine. If you use both gateway and computational routines, they can be combined in one source file or in separate files. If you use separate files, the gateway routine must be the first source file listed in the `mex` command.

The name of the file containing your gateway routine is important, as explained in "Naming the MEX-File" on page 5-5.

Name your Fortran source file with an uppercase `.F` file extension.

The Difference Between .f and .F Files. Fortran compilers assume source files using a lowercase `.f` file extension have been preprocessed. On most platforms, `mex` makes sure the file is preprocessed regardless of the file extension. However, on Macintosh systems, `mex` cannot force preprocessing. Use an uppercase `.F` file extension to ensure your Fortran MEX-file is platform independent.

Using MATLAB Libraries

The MATLAB C and Fortran API Reference describes functions you can use in your gateway and computational routines that interact with MATLAB programs and the data in the MATLAB workspace. The `mx` prefixed functions provide access methods for manipulating MATLAB arrays. The `mex` prefixed functions perform operations in the MATLAB environment.

Required Header Files

To use the functions in the C and Fortran Reference library you must include the `mex` header, which declares the entry point and interface routines. Put this statement in your source file:

```
#include "mex.h"
```

In addition, Fortran MEX-files require the `fintrf.h` header file, which is used by the `mwPointer` preprocessor macro. Put this statement in your Fortran source file:

```
#include "fintrf.h"
```

Naming the MEX-File

The MEX-file name, and hence the name of the function you use in MATLAB, is the name of the source file containing your gateway routine.

The file extension of the MEX-file is platform-dependent. The `mexext` function returns the extension for the current machine.

Computational Routine

The *computational routine* contains the code for performing the computations you want implemented in the MEX-file. Computations can be numerical computations as well as inputting and outputting data. The gateway calls the computational routine as a subroutine.

The programming requirements described in "Creating and Using Source Files" on page 4-4, "Using MATLAB Libraries" on page 4-4, and "Required Header Files" on page 4-4 may also apply to your computational routine.

Preprocessor Macros

The MATLAB *preprocessor macros* `mwSize` and `mwIndex` are used in the `mx` and `mex` functions for cross-platform flexibility. `mwSize` represents size values, such as array dimensions and number of elements. `mwIndex` represents index values, such as indices into arrays.

MATLAB has an additional preprocessor macro for Fortran files, `mwPointer`. MATLAB uses a unique data type, the `mxArray`. Because there is no way to create a new data type in Fortran, MATLAB passes a special identifier, created by the `mwPointer` preprocessor macro, to a Fortran program. This is how you get information about an `mxArray` in a native Fortran data type. For example, you can find out the size of the `mxArray`, determine whether or not it is a string, and look at the contents of the array. Use `mwPointer` to build platform-independent code.

The Fortran preprocessor converts `mwPointer` to `integer*4` when building MEX-files on 32-bit platforms and to `integer*8` when building on 64-bit platforms.

Note Declaring a pointer to be the incorrect size may cause your program to crash.

Using the Fortran %val Construct

The Fortran `%val(arg)` construct specifies that an argument, *arg*, is to be passed by value, instead of by reference. The `%val` construct is supported by most, but not all, Fortran compilers. Compaq Visual Fortran *does* support the construct.

If your compiler does not support the `%val` construct, you must copy the array values into a temporary true Fortran array using the `mxCopy*` routines (e.g., `mxCopyPtrToReal8`).

A %val Construct Example

If your compiler supports the `%val` construct, you can use routines that point directly to the data (i.e., the pointer returned by `mxGetPr` or `mxGetPi`). You can use `%val` to pass this pointer's contents to a subroutine, where it is declared as a Fortran double-precision matrix.

For example, consider a gateway routine that calls its computational routine, `yprime`, by

```
call yprime(%val(yp), %val(t), %val(y))
```

If your Fortran compiler does not support the `%val` construct, you would replace the call to the computational subroutine with

```
C Copy array pointers to local arrays.
       call mxCopyPtrToReal8(t, tr, 1)
       call mxCopyPtrToReal8(y, yr, 4)
C
C Call the computational subroutine.
       call yprime(ypr, tr, yr)
```

```
C
C Copy local array to output array pointer.
       call mxCopyReal8ToPtr(ypr, yp, 4)
```

You must also add the following declaration line to the top of the gateway routine:

```
real*8 ypr(4), tr, yr(4)
```

Note that if you use `mxCopyPtrToReal8` or any of the other `mxCopy*` routines, the size of the arrays declared in the Fortran gateway routine must be greater than or equal to the size of the inputs to the MEX-file coming in from MATLAB. Otherwise, `mxCopyPtrToReal8` does not work correctly.

Data Flow in MEX-Files

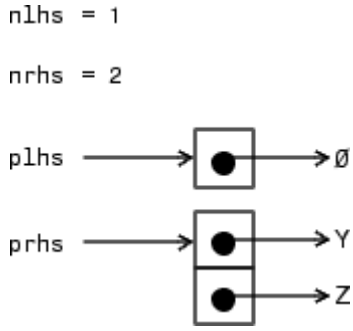
The following examples illustrate data flow in MEX-files:

Showing Data Input and Output

Suppose your MEX-file `myFunction` has 2 input arguments and 1 output argument. The MATLAB syntax is `[X] = myFunction(Y, Z)`. To call `myFunction` from MATLAB, type:

```
X = myFunction(Y, Z);
```

The MATLAB interpreter calls `mexFunction`, the gateway routine to `myFunction`, with the following arguments:



Your input is `prhs`, a 2-element C array (`nrhs = 2`). The first element is a pointer to an `mxArray` named `Y` and the second element is a pointer to an `mxArray` named `Z`.

Your output is `plhs`, a 1-element C array (`nlhs = 1`) where the single element is a `null` pointer. The parameter `plhs` points at nothing because the output `X` is not created until the subroutine executes.

The gateway routine creates the output array and sets a pointer to it in `plhs[0]`. If `plhs[0]` is left unassigned and you assign an output value to the function when you call it, MATLAB generates an error stating that no output was assigned.

Note It is possible to return an output value even if `nlhs = 0`. This corresponds to returning the result in the `ans` variable.

Gateway Routine Data Flow Diagram

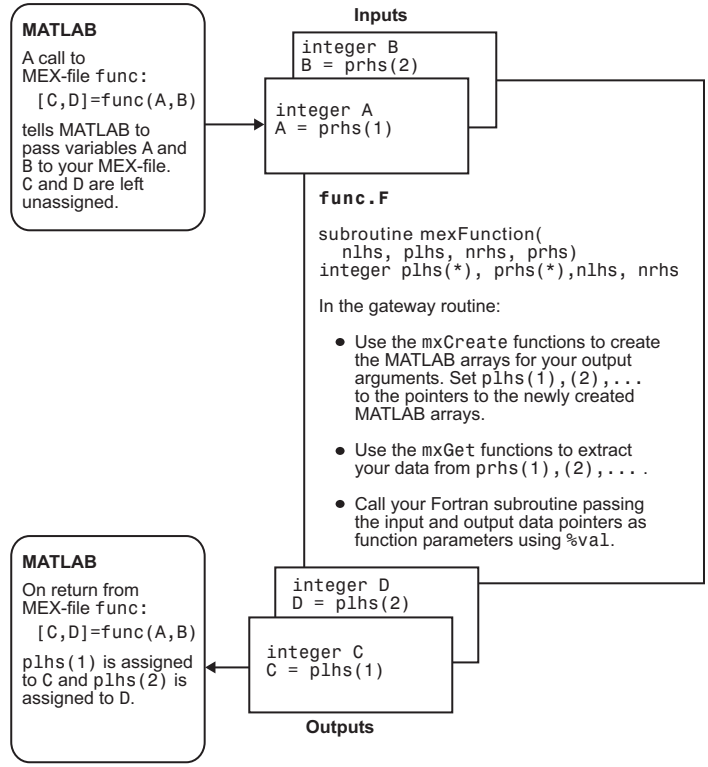
The following MEX Cycle diagram shows how inputs enter a MEX-file, what functions the gateway routine performs, and how outputs return to MATLAB.

In this example, the syntax of the MEX-file `func` is `[C, D] = func(A,B)`. In the figure, a call to `func` tells MATLAB to pass variables `A` and `B` to your MEX-file. `C` and `D` are left unassigned.

The gateway routine `func.F` uses the `mxCreate*` functions to create the MATLAB arrays for your output arguments. It sets `plhs[0]` and `plhs[1]`

to the pointers to the newly created MATLAB arrays. It uses the `mxGet*` functions to extract your data from your input arguments `prhs[0]` and `prhs[0]`. Finally, it calls your computational routine, passing the input and output data pointers as function parameters.

On return to MATLAB, `plhs[0]` is assigned to `C` and `plhs[1]` is assigned to `D`.



Fortran MEX Cycle

MATLAB Example yprime.F

Let's look at an example, `yprime.F`, found in your *matlabroot*`/extern/examples/mex/` directory. ("Building MEX-Files" on page 3-11 explains how to create the MEX-file.) It's calling syntax is `[YP] = YPRIME(T,Y)`, where `T` is an integer and `Y` is a vector with 4 elements. For `T=1` and `Y=1:4`, when you type

```
yprime(T,Y)
```

MATLAB displays:

```
ans =
     2.0000    8.9685    4.0000    -1.0947
```

The gateway routine should validate the input arguments. This step includes checking the number, type, and size of the input arrays as well as examining the number of output arrays. If the inputs are not valid, call `mexErrMsgTxt`. For example,

```
C
C CHECK FOR PROPER NUMBER OF ARGUMENTS
C
      IF (NRHS .NE. 2) THEN
        CALL MEXERRMSGTXT('YPRIME requires two input arguments')
      ELSEIF (NLHS .GT. 1) THEN
        CALL MEXERRMSGTXT('YPRIME requires one output argument')
      ENDIF
C
C CHECK THE DIMENSIONS OF Y.  IT CAN BE 4 X 1 OR 1 X 4.
C
      M = MXGETM(PRHS(2))
      N = MXGETN(PRHS(2))
C
      IF ((MAX(M,N) .NE. 4) .OR. (MIN(M,N) .NE. 1)) THEN
        CALL MEXERRMSGTXT('YPRIME requires that Y be a 4 x 1 vector')
      ENDIF
```

To create MATLAB arrays, call any of the `mxCreate*` functions, like `mxCreateDoubleMatrix`, `mxCreateSparse`, or `mxCreateString`. If it needs them, the gateway routine can call `mxCalloc` to allocate temporary work arrays for the computational routine. In this example,

```
C
C CREATE A MATRIX FOR RETURN ARGUMENT
C
       PLHS(1) = MXCREATEDOUBLEMATRIX(M,N,0)
```

In the gateway routine, you access the data in `mxArray` and manipulate it in your computational subroutine. For example, the expression `mxGetPr(prhs[0])` returns a pointer of type `double *` to the real data in the `mxArray` pointed to by `prhs[0]`. You can then use this pointer like any other pointer of type `double *` in Fortran. For example,

```
C
C ASSIGN POINTERS TO THE VARIOUS PARAMETERS
C
      YPP = MXGETPR(PLHS(1))
C
      TP = MXGETPR(PRHS(1))
      YP = MXGETPR(PRHS(2))
C
C COPY RIGHT HAND ARGUMENTS TO LOCAL ARRAYS OR VARIABLES
      NEL = 1
      CALL MXCOPYPTRTOREAL8(TP, RTP, NEL)
      NEL = 4
      CALL MXCOPYPTRTOREAL8(YP, RYP, NEL)
```

In this example, a computational routine, `yprime`, performs the calculations:

```
C
C DO THE ACTUAL COMPUTATIONS IN A SUBROUTINE
C       CREATED ARRAYS.
C
      CALL YPRIME(RYPP,RTP,RYP)
```

After calling your computational routine from the gateway, you can set a pointer of type `mxArray` to the data it returns. MATLAB recognizes the output from your computational routine as the output from the MEX-file.

```
C
C COPY OUTPUT WHICH IS STORED IN LOCAL ARRAY TO MATRIX OUTPUT
      NEL = 4
      CALL MXCOPYREAL8TOPTR(RYPP, YPP, NEL)
```

When a MEX-file completes its task, it returns control to MATLAB. Any MATLAB arrays that are created by the MEX-file but are not returned to MATLAB through the left-hand side arguments are automatically destroyed.

Examples of Fortran MEX-Files

In this section...

Introduction

The MATLAB API provides a set of Fortran routines that handle double-precision data and strings in MATLAB. For each data type, there is a specific set of functions that you can use for data manipulation.

Note Source code for the examples in this chapter are located in the *matlabroot*/extern/examples/refbook directory of your MATLAB installation.

A First Example — Passing a Scalar

Let's look at a simple example of Fortran code and its MEX-file equivalent. Here is a Fortran computational routine that takes a scalar and doubles it:

```
      subroutine timestwo(y, x)
      real*8 x, y
C
      y = 2.0 * x
```

```
      return
      end
```

To see the same function written in the MEX-file format (`timestwo.F`), open the file in MATLAB Editor.

To compile and link this example, at the MATLAB prompt type

```
mex timestwo.F
```

This command creates the MEX-file called `timestwo` with an extension corresponding to the machine type on which you're running. You can now call `timestwo` as if it were an M-function:

```
x = 2;
y = timestwo(x)
y =
     4
```

Passing Strings

Passing strings from MATLAB to a Fortran MEX-file is straightforward. The program `revord.F` accepts a string and returns the characters in reverse order. To see the example `revord.F`, open the file in MATLAB Editor.

After checking for the correct number of inputs, the gateway routine `mexFunction` verifies that the input was a row vector string. It then finds the size of the string and places the string into a Fortran character array. Note that in the case of character strings, it is not necessary to copy the data into a Fortran character array using `mxCopyPtrToCharacter`. In fact, `mxCopyPtrToCharacter` works only with MAT-files. For more information, see "Using MAT-Files" on page 1-2.

For an input string

```
x = 'hello world';
```

typing

```
y = revord(x)
```

produces

```
y =

dlrow olleh
```

Passing Arrays of Strings

Passing arrays of strings adds a complication to the example “Passing Strings” on page 5-13. Because MATLAB stores elements of a matrix by column instead of by row, the size of the string array must be correctly defined in the Fortran MEX-file. The key point is that the row and column sizes as defined in MATLAB must be reversed in the Fortran MEX-file. Consequently, when returning to MATLAB, the output matrix must be transposed.

This example places a string array/character matrix into MATLAB as output arguments rather than placing it directly into the workspace. At the MATLAB prompt, type

```
passstr;
```

which creates the 5-by-15 `mystring` matrix. You need to do some further manipulation. The original string matrix is 5-by-15. Because of the way MATLAB reads and orients elements in matrices, the size of the matrix must be defined as `M=15` and `N=5` in the MEX-file. After the matrix is put into MATLAB, the matrix must be transposed. The program `passstr.F` illustrates how to pass a character matrix. To see the code `passstr.F`, open the file in MATLAB Editor.

Typing

```
passstr
```

at the MATLAB prompt produces this result:

```
ans =

MATLAB
The Scientific
Computing
Environment
   by TMW, Inc.
```

Passing Matrices

In MATLAB, you can pass matrices into and out of MEX-files written in Fortran. You can manipulate the MATLAB arrays by using `mxGetPr` and `mxGetPi` to assign pointers to the real and imaginary parts of the data stored in the MATLAB arrays. You can create new MATLAB arrays from within your MEX-file by using `mxCreateDoubleMatrix`.

The example `matsq.F` takes a real 2-by-3 matrix and squares each element. To see the source code, open the file in MATLAB Editor.

After performing error checking to ensure that the correct number of inputs and outputs was assigned to the gateway subroutine and to verify the input was in fact a numeric matrix, `matsq.F` creates a matrix. The matrix is copied to a Fortran matrix using `mxCopyPtrToReal8`. Now the computational subroutine can be called, and the return argument is placed into `y_pr`, the pointer to the output, using `mxCopyReal8ToPtr`.

For a 2-by-3 real matrix

```
x = [1 2 3; 4 5 6];
```

typing

```
y = matsq(x)
```

produces this result:

```
y =
     1     4     9
    16    25    36
```

Passing Two or More Inputs or Outputs

The `plhs` and `prhs` parameters (see "The Components of a Fortran MEX-File" on page 5-2) are vectors containing pointers to the left-hand side (output) variables and right-hand side (input) variables. `plhs(1)` contains a pointer to the first left-hand side argument, `plhs(2)` contains a pointer to the second left-hand side argument, and so on. Likewise, `prhs(1)` contains a pointer to the first right-hand side argument, `prhs(2)` points to the second, and so on.

The example `xtimesy.F` multiplies an input scalar times an input scalar or matrix. To see the source code, open the file in MATLAB Editor.

As this example shows, creating MEX-file gateways that handle multiple inputs and outputs is straightforward. All you need to do is keep track of which indices of the vectors `prhs` and `plhs` correspond to which input and output arguments of your function. In this example, the input variable `x` corresponds to `prhs(1)` and the input variable `y` to `prhs(2)`.

For an input scalar `x` and a real 3-by-3 matrix,

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

typing

```
z = xtimesy(x, y)
```

yields this result:

```
z =
     3     3     3
     3     3     3
     3     3     3
```

Handling Complex Data

MATLAB stores complex double-precision data as two vectors of numbers—one vector contains the real data and the other contains the imaginary data. The functions `mxCopyPtrToComplex16` and `mxCopyComplex16ToPtr` copy MATLAB data to a native `complex*16` Fortran array.

The example `convec.F` takes two complex vectors (of length 3) and convolves them. To see the source code, open the file in MATLAB Editor.

Entering the following at the MATLAB prompt

```
x = [3 - 1i, 4 + 2i, 7 - 3i]

x =

   3.0000 - 1.0000i   4.0000 + 2.0000i   7.0000 - 3.0000i
y = [8 - 6i, 12 + 16i, 40 - 42i]

y =

   8.0000 - 6.0000i  12.0000 +16.0000i  40.0000 -42.0000i
```

and calling the new MEX-file

```
z = convec(x, y)
```

results in

```
z =

   1.0e+02 *

  Columns 1 through 4

   0.1800 - 0.2600i   0.9600 + 0.2800i   1.3200 - 1.4400i
   3.7600 - 0.1200i

  Column 5

   1.5400 - 4.1400i
```

which agrees with the results the built-in MATLAB function `conv.m` produces.

Dynamically Allocating Memory

To allocate memory dynamically in a Fortran MEX-file, use `%val`. (See "Using the Fortran %val Construct" on page 5-6.) The example `dblmat.F` takes an input matrix of real data and doubles each of its elements. To see the source code, open the file in MATLAB Editor.

`compute.F` is the subroutine `dblmat` calls to double the input matrix. (Open the file in MATLAB Editor.)

For the 2-by-3 matrix

```
x = [1 2 3; 4 5 6];
```

typing

```
y = dblmat(x)
```

yields

```
y =
     2     4     6
     8    10    12
```

Note The `dblmat.F` example, as well as `fulltosparse.F` and `sincall.F`, are split into two parts, the gateway and the computational subroutine, because of restrictions in some compilers.

Handling Sparse Matrices

MATLAB provides a set of functions that allow you to create and manipulate sparse matrices. There are special parameters associated with sparse matrices, namely `ir`, `jc`, and `nzmax`. For information on how to use these parameters and how MATLAB stores sparse matrices in general, see “Sparse Matrices” on page 3-9.

Note Sparse array indexing is zero-based, not one-based.

The `fulltosparse.F` example illustrates how to populate a sparse matrix. To see the source code, open the file in MATLAB Editor.

`loadsparse.F` is the subroutine `fulltosparse` calls to fill the `mxArray` with the sparse data. (Open the file in MATLAB Editor.)

At the MATLAB prompt, typing

```
full = eye(5)
full =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

creates a full, 5-by-5 identity matrix. Using `fulltosparse` on the full matrix produces the corresponding sparse matrix:

```
spar = fulltosparse(full)
spar =
   (1,1)        1
   (2,2)        1
   (3,3)        1
   (4,4)        1
   (5,5)        1
```

Calling Functions from Fortran MEX-Files

You can call MATLAB functions, operators, M-files, and even other MEX-files from within your Fortran source code by using the API function `mexCallMATLAB`. The `sincall.F` example creates an `mxArray`, passes various pointers to a subfunction to acquire data, and calls `mexCallMATLAB` to calculate the sine function and plot the results. To see the source code, open the file in MATLAB Editor.

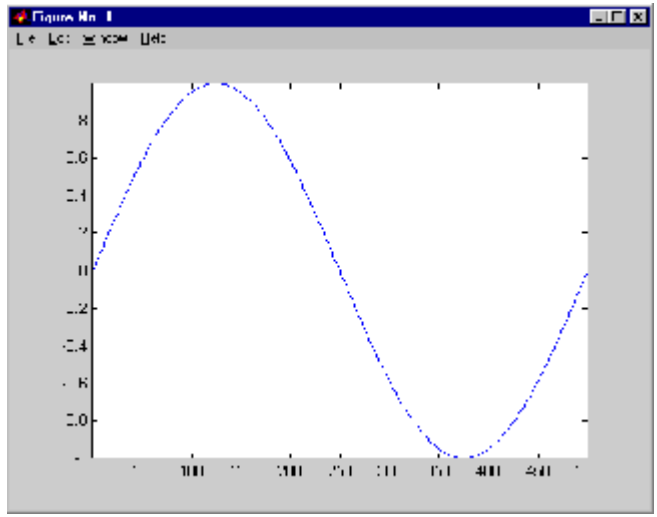
`fill.F` is the subroutine `sincall` calls to fill the `mxArray` with data. (Open the file in MATLAB Editor.)

It is possible to use `mexCallMATLAB` (or any other API routine) from within your computational Fortran subroutine. Note that you can only call most MATLAB functions with double-precision data. M-functions that perform computations, such as `eig`, do not work correctly with data that is not double precision.

Running this example

```
sincall
```

displays the results



Note You can generate an object of type `mxUNKNOWN_CLASS` using `mexCallMATLAB`. See the following example.

This example creates an M-file that returns two variables but only assigns one of them a value:

```
function [a,b]=foo[c]
a=2*c;
```

MATLAB displays the following warning:

```
Warning: One or more output arguments not assigned during call to
'foo'.
```

If you then call `foo` using `mexCallMATLAB`, the unassigned output variable is now of type `mxUNKNOWN_CLASS`.

Advanced Topics

In this section...

Help Files

Because the MATLAB interpreter chooses the MEX-file when both an M-file and a MEX-file with the same name are encountered in the same directory, it is possible to use M-files for documenting the behavior of your MEX-files. The MATLAB `help` command automatically finds and displays the appropriate M-file when help is requested and the interpreter finds and executes the corresponding MEX-file when the function is actually invoked.

Linking Multiple Files

You can combine several source files when building MEX-files. For example,

```
mex circle.F square.o rectangle.F shapes.o
```

is a legal command that operates on the `.F` and `.o` files to create a MEX-file called `circle.`*`ext`*, where *`ext`* is the extension corresponding to the MEX-file type. The name of the resulting MEX-file is taken from the first file in the list.

You may find it useful to use a software development tool like `MAKE` to manage MEX-file projects involving multiple source files. Simply create a `MAKEFILE` that contains a rule for producing object files from each of your source files and then invoke `mex` to combine your object files into a MEX-file. This way you can ensure that your source files are recompiled only when necessary.

Note On Linux, you must use the `-fortran` switch to the `mex` script if you are linking Fortran objects.

Workspace for MEX-File Functions

Unlike M-file functions, MEX-file functions do not have their own variable workspace. `mexEvalString` evaluates the string in the caller's workspace. In addition, you can use the `mexGetVariable` and `mexPutVariable` routines to get and put variables into the caller's workspace.

Handling Large mxArrays

MEX-files built on 64-bit platforms can handle 64-bit mxArrays. These large data arrays can have up to 2^{48}–1 elements. The maximum number of elements a sparse mxArray can have is 2^{48}-2.

Using the following instructions creates platform-independent MEX-files as well.

Your system configuration can impact the performance of MATLAB. The 64-bit processor requirement enables you to create the mxArray and access data in it. However, your system's memory, in particular the size of RAM and virtual memory, determine the speed at which MATLAB processes the mxArray. The more memory available, the faster the processing.

The amount of RAM also limits the amount of data you can process at one time in MATLAB. For guidance on memory issues, see "Memory Allocation in MATLAB". Memory management within MEX-files can have special considerations, as described in "Memory Management" on page 4-29.

Using the 64-Bit API

To work with 64-bit mxArrays, your source code must comply with the 64-bit API, which consists of the functions in the following table.

`mxCalcSingleSubscript`	`mxCreateCellMatrix`
`mxCalloc`	`mxCreateCharArray`

mxCopyCharacterToPtr	mxCreateCharMatrixFromStrings
mxCopyComplex16ToPtr	mxCreateDoubleMatrix
mxCopyComplex8ToPtr	mxCreateLogicalArray
mxCopyInteger1ToPtr	mxCreateLogicalMatrix
mxCopyInteger2ToPtr	mxCreateNumericArray
mxCopyInteger4ToPtr	mxCreateNumericMatrix
mxCopyPtrToCharacter	mxCreateSparse
mxCopyPtrToComplex16	mxCreateSparseLogicalMatrix
mxCopyPtrToComplex8	mxCreateSparseLogicalMatrix
mxCopyPtrToInteger1	mxCreateStructMatrix
mxCopyPtrToInteger2	mxGetCell
mxCopyPtrToInteger4	mxGetElementSize
mxCopyPtrToPtrArray	mxGetField
mxCopyPtrToReal4	mxGetFieldByNumber
mxCopyPtrToReal8	mxGetIr
mxCopyReal4ToPtr	mxGetJc
mxCopyReal8ToPtr	mxGetM
mxCopyReal8ToPtr	mxGetN
mxCopyReal4ToPtr	mxGetNumberOfDimensions
mxCreateCellArray	mxGetNumberOfElements

Functions in this API use the `mwIndex`, `mwSize`, and `mwPointer` preprocessor macros. For information about using these macros, see "Required Header Files" on page 5-4.

Building the MEX-File

Use the `mex` command option, `-largeArrayDims`, with the 64-bit API.

Caution Using Negative Values

When using the 64-bit API, `mwSize` and `mwIndex` are equivalent to `size_t` in C or `INTEGER*8` in Fortran. These types are unsigned, unlike `int` and `INTEGER*4`, which are the types used in the 32-bit API. Be careful not to pass any negative values to functions that take `mwSize` or `mwIndex` arguments. Do not cast negative `int` or `INTEGER*4` values to `mwSize` or `mwIndex`; the returned value can not be predicted. Instead, change your code to avoid using negative values.

Building Cross-Platform Applications

If you develop cross-platform applications (programs that can run on both 32- and 64-bit architectures), you must pay attention to the upper limit of values you use for `mwSize` and `mwIndex`. The 32-bit application reads these values and assigns them to variables declared as `int` in C or `INTEGER*4` in Fortran. Be careful to avoid assigning a large `mwSize` or `mwIndex` value to an `int`, `INTEGER*4`, or other variable that might be too small.

Memory Management

MATLAB now implicitly destroys (by calling `mxDestroyArray`) any arrays created by a MEX-file that are not returned in the left-hand side list (`plhs()`). Consequently, any misconstructed arrays left over at the end of a MEX-file's execution have the potential to cause memory errors.

In general, we recommend that MEX-files destroy their own temporary arrays and clean up their own temporary memory. For additional information on memory management techniques, see the sections "Memory Management" on page 4-29 and "Memory Management Compatibility Issues" on page 3-39.

Debugging Fortran Language MEX-Files

In this section...

Notes on Debugging

The examples show how to debug `timestwo.F`, found in your *`matlabroot`*`/extern/examples/refbook/` directory.

MEX-files built with the `-g` option do not execute on other computers because they rely on files that are not distributed with MATLAB. Refer to the "Calling C and Fortran Programs from MATLAB" topic "Troubleshooting" on page 3-32 for additional information on isolating problems with MEX-files.

Debugging on Windows

For MEX-files compiled with any version of the Intel Visual Fortran compiler, you can use the debugging tools found in your version of Microsoft Visual Studio. Refer to the "Creating C Language MEX-Files" topic "Debugging on Windows" on page 4-46 for instructions on using this debugger.

For information on debugging MEX-files compiled with other MATLAB supported compilers, see Technical Note 1605, MEX-files Guide, at http://www.mathworks.com/support/tech-notes/1600/1605.html.

Debugging on Linux

The MATLAB supported Fortran compiler g95 has a `-g` option for building MEX-files with debug information. Such files can be used with gdb, the GNU Debugger. This section describes using gdb.

For information on debugging MEX-files compiled with other MATLAB supported compilers, see Technical Note 1605, MEX-files Guide, at http://www.mathworks.com/support/tech-notes/1600/1605.html.

GNU Debugger gdb

In this example, the MATLAB command prompt >> is shown in front of MATLAB commands, and `linux>` represents a Linux prompt; your system may show a different prompt. The debugger prompt is `<gdb>`.

1. To compile the MEX-file, type

   ```
   linux> mex -g timestwo.F
   ```

   On a Linux 32–bit platform, this command creates the executable file `timestwo.mexglx`.

2. At the Linux prompt, start the gdb debugger using the `matlab -D` option:

   ```
   linux> matlab -Dgdb
   ```

3. Start MATLAB without the Java virtual machine (JVM) by using the `-nojvm` startup flag:

   ```
   <gdb> run -nojvm
   ```

4. In MATLAB, enable debugging with the `dbmex` function and run your MEX-file:

   ```
   >> dbmex on
   >> y = timestwo(4)
   ```

5. At this point, you are ready to start debugging.

   It is often convenient to set a breakpoint at `mexFunction` so you stop at the beginning of the gateway routine.

Note The function name may be slightly altered by the compiler (e.g., it may have an underscore appended). To determine how this symbol appears in a given MEX-file, use the Linux command `nm`. For example,

```
linux> nm timestwo.mexglx | grep -i mexfunction
```

Linux responds with something like

```
0000091c T mexfunction_
```

Use `mexfunction_` in the breakpoint statement. Be sure to use the correct case.

```
<gdb> break mexfunction_
<gdb> continue
```

6 Once you hit one of your breakpoints, you can make full use of any commands the debugger provides to examine variables, display memory, or inspect registers.

To proceed from a breakpoint, type `continue`.

```
<gdb> continue
```

7 After stopping at the last breakpoint, type

```
<gdb> continue
```

`timestwo` finishes and MATLAB displays

```
y =

     8
```

8 From the MATLAB prompt you can return control to the debugger by typing

```
>> dbmex stop
```

Or, if you are finished running MATLAB, type

```
>> quit
```

9 When you are finished with the debugger, type

```
<gdb> quit
```

You return to the Linux prompt.

Refer to the documentation provided with your debugger for more information on its use.

6

Calling MATLAB from C and Fortran Programs

The MATLAB engine library is a set of routines that allows you to call MATLAB from your own programs, thereby employing MATLAB as a computation engine. MATLAB engine programs are C or Fortran programs that communicate with a separate MATLAB process via pipes, on UNIX, and through a Component Object Model (COM) interface, on Windows. There is a library of functions provided with MATLAB that allows you to start and end the MATLAB process, send data to and from MATLAB, and send commands to be processed in MATLAB.

Using the MATLAB Engine

In this section...

Introduction

Some of the things you can do with the MATLAB engine are

- Call a math routine, for example, to invert an array or to compute an FFT from your own program. When employed in this manner, MATLAB is a powerful and programmable mathematical subroutine library.
- Build an entire system for a specific task, for example, radar signature analysis or gas chromatography, where the front end (GUI) is programmed in C and the back end (analysis) is programmed in MATLAB, thereby shortening development time.

The MATLAB engine operates by running in the background as a separate process from your own program. This offers several advantages:

- On UNIX, the MATLAB engine can run on your machine, or on any other UNIX machine on your network, including machines of a different architecture. This allows you to implement a user interface on your workstation and perform the computations on a faster machine located elsewhere on your network. The description of the `engOpen` function offers further information.
- Instead of requiring that all of MATLAB be linked to your program (a substantial amount of code), only a small engine communication library is needed.

Note To run MATLAB engine on the UNIX platform, you must have the C shell `csh` installed at `/bin/csh`.

The Engine Library

The engine library contains the following routines for controlling the MATLAB computation engine. Their names all begin with the three-letter prefix `eng`. These tables list all the available engine functions and their purposes.

C Engine Routines

Function	Purpose
`engOpen`	Start up MATLAB engine
`engClose`	Shut down MATLAB engine
`engGetVariable`	Get a MATLAB array from the MATLAB engine
`engPutVariable`	Send a MATLAB array to the MATLAB engine
`engEvalString`	Execute a MATLAB command
`engOutputBuffer`	Create a buffer to store MATLAB text output
`engOpenSingleUse`	Start a MATLAB engine session for single, nonshared use
`engGetVisible`	Determine visibility of MATLAB engine session
`engSetVisible`	Show or hide MATLAB engine session

Fortran Engine Routines

Function	Purpose
`engOpen`	Start up MATLAB engine
`engClose`	Shut down MATLAB engine
`engGetVariable`	Get a MATLAB array from the MATLAB engine
`engPutVariable`	Send a MATLAB array to the MATLAB engine

Fortran Engine Routines (Continued)

Function	Purpose
`engEvalString`	Execute a MATLAB command
`engOutputBuffer`	Create a buffer to store MATLAB text output

The MATLAB engine also uses the `mx`–prefixed API routines discussed in Chapter 4, “Creating C Language MEX-Files” and Chapter 5, “Creating Fortran MEX-Files”.

Communicating with MATLAB

On UNIX, the engine library communicates with the MATLAB engine using pipes, and, if needed, `rsh` for remote execution. On Microsoft Windows, the engine library communicates with MATLAB using a Component Object Model (COM) interface. Chapter 8, “COM Support in MATLAB (Windows Only)” contains a detailed description of COM.

GUI-Intensive Applications

If you have graphical user interface (GUI) intensive applications that execute a lot of callbacks through the MATLAB engine, you should force these callbacks to be evaluated in the context of the base workspace. Use `evalin` to specify that the base workspace be used in evaluating the callback expression, as follows:

```
engEvalString(ep, "evalin('base', expression)")
```

Specifying the base workspace in this manner ensures MATLAB processes the callback correctly and returns results for that call.

This does not apply to computational applications that do not execute callbacks.

Examples of Calling Engine Functions

In this section...

Overview

It is important to understand the sequence of steps you must follow when using the engine functions. For example, before using `engPutVariable`, you must create the matrix and populate it.

After reviewing these examples, follow the instructions in "Compiling and Linking MATLAB Engine Programs" on page 6-10 to build the application and test it. By building and running the application, you ensure that your system is properly configured for engine applications.

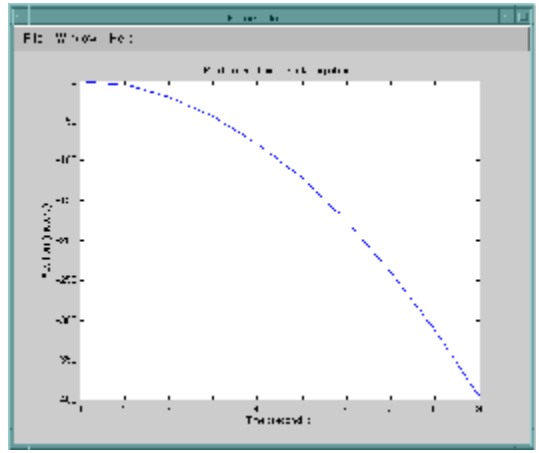
Calling MATLAB from a C Application

This program, `engdemo.c`, illustrates how to call the engine functions from a stand-alone C program. For the Windows version of this program, see `engwindemo.c` in the *`matlabroot`*`\extern\examples\eng_mat` directory. *`matlabroot`* is the MATLAB root directory. engine examples, like the MAT-file examples, are located in the `eng_mat` directory:

To see `engdemo.c`, open the file in MATLAB Editor.

To see the Windows version `engwindemo.c`, open the file.

The first part of this program launches MATLAB and sends it data. MATLAB then analyzes the data and plots the results.



The program then continues with

```
Press Return to continue
```

Pressing **Return** continues the program:

```
Done for Part I.
Enter a MATLAB command to evaluate.  This command should
create a variable X.  This program will then determine
what kind of variable you created.
For example: X = 1:5
```

Entering `X = 17.5` continues the program execution.

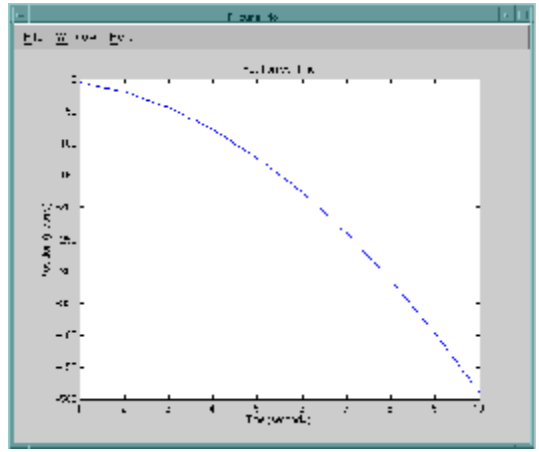
```
X = 17.5

X =

   17.5000


Retrieving X...
X is class double
Done!
```

Finally, the program frees memory, closes the MATLAB engine, and exits.

Calling MATLAB from a Fortran Application

The program, `fengdemo.F`, illustrates how to call the engine functions from a stand-alone Fortran program. To see the code, open the file in MATLAB Editor.

Executing this program launches MATLAB, sends it data, and plots the results.



The program continues with

```
Type 0 <return> to Exit
Type 1 <return> to continue
```

Entering `1` at the prompt continues the program execution:

```
1
 MATLAB computed the following distances:
   time(s)  distance(m)
   1.00     -4.90
   2.00     -19.6
   3.00     -44.1
   4.00     -78.4
   5.00     -123.
   6.00     -176.
   7.00     -240.
```

```
8.00     -314.
9.00     -397.
10.0     -490.
```

Finally, the program frees memory, closes the MATLAB engine, and exits.

Attaching to an Existing MATLAB Session

You can make a MATLAB engine program attach to a MATLAB session that is already running by starting the MATLAB session with `/Automation` in the command line. When you make a call to `engOpen`, it then connects to this existing session. You should only call `engOpen` once, because any `engOpen` calls now connect to this one MATLAB session.

The `/Automation` option also causes the command window to be minimized. You must open it manually.

Note For more information on the `/Automation` command-line argument, see "Additional Automation Server Information" on page 8-122. For information about the Component Object Model interfaces used by MATLAB, see "Introducing MATLAB COM Integration" on page 8-3.

For example,

1. Shut down any MATLAB sessions.
2. From the **Start** button on the Windows menu bar, click **Run**.
3. In the **Open** field, type

   ```
   d:\matlab\bin\win32\matlab.exe /Automation
   ```

   or

   ```
   d:\matlab\bin\win64\matlab.exe /Automation
   ```

   where `d:\matlab\bin\win32` or `d:\matlab\bin\win64` represents the path to the MATLAB executable.

4 Click **OK**. This starts MATLAB.

5 In MATLAB, change directories to *matlabroot*`/extern/examples/eng_mat`.

6 Compile the `engwindemo.c` example.

7 Run the `engwindemo` program by typing at the MATLAB prompt

```
!engwindemo
```

This does not start another MATLAB session, but rather uses the MATLAB session that is already open.

Note On the UNIX platform, you cannot make a MATLAB engine program use a MATLAB session that is already running.

Compiling and Linking MATLAB Engine Programs

In this section...

Step 1 — Write Your Application

Write your application in C or Fortran using any of the MATLAB engine routines to perform computations in MATLAB. See "Using the MATLAB Engine" on page 6-2 and "Examples of Calling Engine Functions" on page 6-5 for help.

Note If you plan to build with the Borland C++ compiler on Windows, read "Masking Floating-Point Exceptions" on page 6-18 for information on floating-point exceptions.

Step 2 — Check Required Libraries and Files

MATLAB requires the following files for building any engine application:

- "Third-Party Libraries" on page 6-11
- "Library Files Required by libeng" on page 6-11
- "Unicode Data Files" on page 6-12

Third-Party Libraries

Verify that the required libraries are installed. Use the following table to identify the path and library filename. Replace *`libfile`* with each of these filenames:

```
libeng
libmx
```

Operating System	Library Path and Filename
Linux	*`matlabroot`*`/bin/glnx86/`*`libfile`*`.so`
64-bit Linux	*`matlabroot`*`/bin/glnxa64/`*`libfile`*`.so`
64-bit Solaris SPARC	*`matlabroot`*`/bin/sol64/`*`libfile`*`.so`
Macintosh (PPC)	*`matlabroot`*`/bin/mac/`*`libfile`*`.dylib`
Macintosh (Intel)	*`matlabroot`*`/bin/maci/`*`libfile`*`.dylib`
Windows	*`matlabroot`*`\bin\win32\`*`libfile`*`.dll`
Windows x64	*`matlabroot`*`\bin\win64\`*`libfile`*`.dll`

Library Files Required by libeng

The `libeng` library requires additional third-party library files. MATLAB uses these libraries to support Unicode character encoding and data compression in MAT-files.

These library files must reside in the same directory as `libmx`. You can determine what most of these libraries are using the platform-specific commands shown here.

Operating System	Library Path and Filename
All Linux, Solaris	`ldd -d libeng.so`
All Macintosh	`otool -L libeng.dylib`
Windows	See instructions below.

On Windows, download the Dependency Walker utility from the following Web site:

```
http://www.dependencywalker.com/
```

Drag and drop the `libeng.dll` file into the Depends window.

Unicode Data Files

Verify that the appropriate Unicode data file is installed. Systems that order bytes in a big-endian manner use `icudt32b.dat`, and those that have little-endian ordering use `icudt32l.dat`.

Operating System	Unicode File Path and Filename
Linux	*matlabroot*`/bin/glnx86/icudt32l.dat`
64-bit Linux	*matlabroot*`/bin/glnxa64/icudt32l.dat`
64-bit Solaris SPARC	*matlabroot*`/bin/sol64/icudt32b.dat`
Macintosh (PPC)	*matlabroot*`/bin/mac/icudt32b.dat`
Macintosh (Intel)	*matlabroot*`/bin/maci/icudt32b.dat`
Windows	*matlabroot*`\bin\win32\icudt32l.dat`
Windows x64	*matlabroot*`\bin\win64\icudt32l.dat`

Note If you need to manipulate Unicode text directly in your application, the latest version of International Components for Unicode (ICU) is available online from the IBM Corporation Web site at `http://icu.sourceforge.net/download`.

Step 3 — Build the Application

Use the `mex` script to compile and link engine programs. `mex` has a set of switches you can use to modify the compile and link stages. The table MEX Script Switches on page 3-20 lists the available switches and their uses.

MEX Options File

MATLAB supplies an options file to facilitate building MEX applications. This file contains compiler-specific flags that correspond to the general compile, prelink, and link steps required on your system. If you want to customize the build process, you can modify this file.

Different options files are provided for UNIX and Windows.

Operating System	Default Options File
UNIX	*matlabroot*/bin/engopts.sh
Windows	*matlabroot*\bin\win32\mexopts*engmatopts.bat
Windows x64	*matlabroot*\bin\win64\mexopts*engmatopts.bat

On Windows systems, the options file depends on which compiler you use. The name of the options file is prefixed with a string representing the compiler and compiler version it is used with.

For example, to locate the options file on a Windows 32-bit system, type:

```
dir([matlabroot '\bin\win32\mexopts\*engmatopts.bat'])
```

If you need to modify the options file for your particular compiler, use the `mex` command with the `-v` switch to view the current compiler and linker settings, and then make the appropriate changes in the options file.

Build the Application

To build your engine application, use the `mex` script with the options filename and the name of your MEX-file.

On UNIX Systems. Enter the following command, where *`mexfilename`* is the name of your C or Fortran program. Enclose *`mexfilename`* in single quotation marks.

```
mex('-f', [matlabroot '/bin/engopts.sh'], mexfilename);
```

Alternatively, copy the options file to your current working directory, and then enter the following command:

```
mex -f engopts.sh mexfilename
```

On Windows Systems. Enter the following command, where *mexfilename* is the name of your C or Fortran program. Enclose *mexfilename* in single quotation marks. This example uses the Lcc compiler. Be sure to use the appropriate options file for your compiler.

```
mex('-f', [matlabroot ...
   '\bin\win32\mexopts\lccengmatopts.bat'], mexfilename);
```

Alternatively, copy the options file to your current working directory, and then enter the following command:

```
mex -f lccengmatopts.bat mexfilename
```

Step 4 — Set Run-Time Library Path

At run-time, you need to tell the system where the API shared libraries reside.

On UNIX Systems

Set the library path as follows for the C and Bourne shells. In the commands shown, replace the terms *envvar* and *pathspec* with the appropriate values from the table that follows.

In the C shell, the command to set the library path is

```
setenv envvar pathspec
```

In the Bourne shell, use

```
envvar = pathspec:envvar export envvar
```

Operating System	*envvar*	*pathspec*
Linux	LD_LIBRARY_PATH	*matlabroot*/bin/glnx86: *matlabroot*/sys/os/glnx86
64-bit Linux	LD_LIBRARY_PATH	*matlabroot*/bin/glnxa64: *matlabroot*/sys/os/glnxa64
64-bit Solaris SPARC	LD_LIBRARY_PATH	*matlabroot*/bin/sol64: *matlabroot*/sys/os/sol64
Macintosh (PPC)	DYLD_LIBRARY_PATH	*matlabroot*/bin/mac: *matlabroot*/sys/os/mac
Macintosh (Intel)	DYLD_LIBRARY_PATH	*matlabroot*/bin/maci: *matlabroot*/sys/os/maci

Here is an example for a Solaris system for the C shell:

```
setenv LD_LIBRARY_PATH matlabroot/bin/sol64:matlabroot/sys/os/sol64
```

and for the Bourne shell:

```
LD_LIBRARY_PATH=matlabroot/bin/sol64:matlabroot/sys/os/sol64:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH
```

Place these commands in a startup script such as `~/.cshrc` for the C shell or `~/.profile` for the Bourne shell.

On Windows Systems

Set the `Path` environment variable to the path string returned by MATLAB in response to the following expression:

```
[matlabroot '\bin\win32']
```

or

```
[matlabroot '\bin\win64']
```

To set an environment variable in Windows, select **Start > Settings > Control Panel > System**. Windows opens the System

Properties dialog box. Click the **Advanced** tab, and then the **Environment Variables** button.

In the **System variables** panel scroll down until you find the `Path` variable. Click this variable to highlight it, and then click the **Edit** button to open the Edit System Variable dialog box. At the end of the path string, enter a semicolon and then the path string returned by evaluating the expression shown above in MATLAB. Click **OK** in the Edit System Variable dialog box, and in all remaining dialog boxes.

Step 5 — (Windows Only) Register MATLAB as a COM Server

To run this program on Windows, you need to have MATLAB registered as a COM server on your system. This registration is part of the MATLAB installation and should have already been done for you as part of the installation. If, for some reason, the registration was not done or did not complete successfully, you may see the following error displayed when you try to run this example:

```
Can't start MATLAB engine
```

If you see this error, manually register MATLAB as a server by entering the following commands in a DOS command window:

```
cd matlabroot\bin\win32
matlab /regserver
```

or

```
cd matlabroot\bin\win64
matlab /regserver
```

Step 6 — Test the Program

Test your application in MATLAB by typing

```
!engwindemo
```

Example — Building an Engine Application on Windows

MATLAB provides a demonstration program written in C that you can use to verify the build process on your computer. The demo file for Windows systems is `engwindemo.c`.

Copy the C language MEX-file `engwindemo.c` to your current working directory:

```
demofile = [matlabroot '\extern\examples\eng_mat\engwindemo.c'];
copyfile(demofile, '.');
```

Look in the `\bin\win32\mexopts` directory for the appropriate options file for the Lcc compiler. Use the following commands to build the executable file using this compiler:

```
optsfile = [matlabroot '\bin\win32\mexopts\lccengmatopts.bat'];
mex('-f', optsfile, 'engwindemo.c');
```

Verify that the build worked by looking in your current working directory for the file `engwindemo.exe`:

```
dir engwindemo.exe
```

To run the demo from MATLAB, make sure your current working directory is set to the one in which you built the executable file, and then type

```
!engwindemo
```

Borland Compilers on Windows

When using a Borland compiler to build a Windows application such as `engwindemo`, you must modify the appropriate `bcc*engmatopts.bat` options file. For information about making these changes, see *`matlabroot`*`\bin\win32\mexopts\README.borland`.

Example — Building an Engine Application on UNIX

MATLAB provides a demonstration program written in C that you can use to verify the build process on your computer. The demo file for UNIX systems is called `engdemo.c`.

Copy the C language MEX-file `engdemo.c` to your current working directory:

```
demofile = [matlabroot '/extern/examples/eng_mat/engdemo.c'];
copyfile(demofile, '.');
```

Build the executable file using the ANSI compiler for engine stand-alone programs and the options file `engopts.sh`:

```
optsfile = [matlabroot '/bin/engopts.sh'];
mex('-f', optsfile, 'engdemo.c');
```

Verify that the build worked by looking in your current working directory for the file `engdemo`:

```
dir engdemo
```

To run the demo in MATLAB, make sure your current working directory is set to the one in which you built the executable file, and then type

```
!engdemo
```

Masking Floating-Point Exceptions

Read this section if you plan to build with the Borland C++ compiler on Windows. It explains how to avoid program termination when a floating-point exception occurs.

Certain mathematical operations can result in nonfinite values. For example, division by zero results in the nonfinite IEEE value `inf`. A floating-point exception occurs when such an operation is performed. Because MATLAB uses an IEEE model that supports nonfinite values such as `inf` and `NaN`, MATLAB disables, or *masks*, floating-point exceptions.

Some compilers do not mask floating-point exceptions by default. This causes engine programs built with such compilers to terminate when a floating-point exception occurs. Take special precautions when using these compilers to mask floating-point exceptions so that your engine application performs properly.

Note MATLAB based applications should never get floating-point exceptions. If you do get a floating-point exception, verify that any third-party libraries that you link against do not enable floating-point exception handling.

The only compiler and platform on which you need to mask floating-point exceptions is the Borland C++ compiler on Windows.

Borland C++ Compiler on Windows

To mask floating-point exceptions when using the Borland C++ compiler on the Windows platform, you must add some code to your program. Include the following at the beginning of your `main()` or `WinMain()` function, before any calls to MATLAB API functions.

```
#include <float.h>
 .
 .
 .
_control87(MCW_EM,MCW_EM);
 .
 .
 .
```

7

Calling Java from MATLAB

Example — Communicating Through a Serial Port (p. 7-76)	Create a SerialPort object and configure the port using methods provided by that class
Example — Creating and Using a Phone Book (p. 7-82)	Create a phone book using a data dictionary that operates using key/value pairs in a hash table

Using Java from MATLAB: An Overview

In this section...

Java Interface Is Integral to MATLAB

Every installation of MATLAB includes a Java Virtual Machine (JVM), so that you can use the Java interpreter via MATLAB commands, and you can create and run programs that create and access Java objects. For information on MATLAB installation, see the MATLAB installation documentation for your platform.

Benefits of the MATLAB Java Interface

The MATLAB Java interface enables you to:

- Access Java API (application programming interface) class packages that support essential activities such as I/O and networking. For example, the `URL` class provides convenient access to resources on the Internet.
- Access third-party Java classes
- Easily construct Java objects in MATLAB
- Call Java object methods, using either Java or MATLAB syntax
- Pass data between MATLAB variables and Java objects

Who Should Use the MATLAB Java Interface

The MATLAB Java interface is intended for all MATLAB users who want to take advantage of the special capabilities of the Java programming language.

For example:

- You need to access, from MATLAB, the capabilities of available Java classes.
- You are familiar with object-oriented programming in Java or in another language, such as C++.
- You are familiar with MATLAB object-oriented classes, or with MATLAB MEX-files.

To Learn More About Java Programming

For a complete description of the Java language and for guidance in object-oriented software design and programming, you'll need to consult outside resources. For example, these recently published books may be helpful:

- *Java in a Nutshell* (Fourth Edition), by David Flanagan
- *Teach Yourself Java in 21 Days*, by Lemay and Perkins

Another place to find information is the JavaSoft Web site.

```
http://www.javasoft.com
```

For other suggestions on object-oriented programming resources, see:

- *Object-Oriented Software Construction*, by Bertrand Meyer
- *Object-Oriented Analysis and Design with Applications*, by Grady Booch, Robert A. Maksimchuk, Michael W. Engel, and Alan Brown

Platform Support for the Java Virtual Machine

To find out which version of the Java Virtual Machine (JVM) is being used by MATLAB on your platform, type the following at the MATLAB prompt.

```
version -java
```

Using a Different Version of the Java JVM

MATLAB ships with one specific version of the Java Virtual Machine (JVM) and uses this version by default with the MATLAB interface to Java. This section describes how to download and select a version other than the default.

Note MATLAB is only fully supported on the JVM that it ships with. Some components might not work properly under a different version of the JVM. For example, calling functions in a dynamically linked library that was created with a different JVM than that used by MATLAB might cause a segmentation violation error.

To change the JVM version that MATLAB uses, follow these steps:

1. "Download the JVM Version You Want to Use" on page 7-5.
2. "Locate the Root of the Run-time Path for this Version" on page 7-5.
3. "Set the MATLAB_JAVA Environment Variable to this Path" on page 7-6.

When you have enabled a different version of the JVM, you can verify that MATLAB is using this version by entering the `version -java` command documented in the previous section.

Download the JVM Version You Want to Use

You can download the Java Virtual Machine from the Web site `http://java.sun.com/j2se/downloads.html`.

If you are using Linux, go to the Web site `http://www.blackdown.org/java-linux/mirrors.html`, and choose the version required by your processor.

Locate the Root of the Run-time Path for this Version

To get MATLAB to use the version you have just downloaded, you must first find the root of the run-time path for this JVM, and then set the `MATLAB_JAVA` environment variable to that path. To locate the JVM run-time path, find the directory in the Java installation tree that is one level up from the directory containing the file `rt.jar`. This may be a subdirectory of the main JDK install directory. (If you cannot find `rt.jar`, look for the file `classes.zip`.)

For example, if the JDK is installed in `D:\jdk1.2.1` on Windows and the `rt.jar` file is in `D:\jdk1.2.1\jre\lib`, set `MATLAB_JAVA` to the directory one level up from that: `D:\jdk1.2.1\jre`.

On UNIX, if the JDE is installed in `/usr/openv/java/jre/lib` and the `rt.jar` is in `/usr/openv/java/jre/lib`, set `MATLAB_JAVA` to the path `/usr/openv/java/jre`.

Set the MATLAB_JAVA Environment Variable to this Path

The way you set or modify the value of the `MATLAB_JAVA` variable depends on which platform you are running MATLAB on.

Windows 2000/XP. To set `MATLAB_JAVA` on Windows 2000 or Windows XP:

1. Click **Settings** in the **Start** Menu.
2. Choose **Control Panel**.
3. Click **System**.
4. Choose the **Advanced** tab and then click the **Environment Variables** button.
5. You now can set (or add) the `MATLAB_JAVA` system environment variable to the path of your JVM.

UNIX/Linux. To set `MATLAB_JAVA` on UNIX or Linux systems, use the `setenv` command, as shown here:

```
setenv MATLAB_JAVA <path to JVM>
```

Bringing Java Classes and Methods into MATLAB

In this section...

Introduction

You can draw from an extensive collection of existing Java classes or create your own class definitions to use with MATLAB. This section explains how to go about finding the class definitions that you need or how to create classes of your own design. Once you have the classes you need, defined in either individual `.class` files, packages, or Java Archive files, you can make them available in the MATLAB environment. This section also describes how to specify the native method libraries used by Java.

Sources of Java Classes

Following are the three main Java class sources that you can use in MATLAB:

- Java built-in classes — general-purpose class packages, such as `java.awt`, included in the Java language. See your Java language documentation for descriptions of these packages.
- Third-party classes — packages of special-purpose Java classes.
- User-defined classes — Java classes or subclasses of existing classes that you define. You need to use a Java development environment to do this, as explained in the following section.

Defining New Java Classes

To define new Java classes and subclasses of existing classes, you must use a Java development environment external to MATLAB. See Technical Note 1601 `http://www.mathworks.com/support/tech-notes/1600/1601.html` for information on supported versions of the JDK. You can download the development kit from the Sun Microsystems Web site, (`http://java.sun.com/j2se/`). The Sun site also provides documentation for the Java language and classes that you need for development.

After you create class definitions in `.java` files, use your Java compiler to produce `.class` files from them. The next step is to make the class definitions in those `.class` files available for you to use in MATLAB.

The Java Class Path

MATLAB loads Java class definitions from files that are on the *Java class path*. The class path is a series of file and directory specifications that MATLAB uses to locate class definitions. When loading a particular Java class, MATLAB searches files and directories in the order they occur on the class path until a file is found that contains that class definition. The first definition that is found ends the search.

The Java class path consists of two segments: the *static path* and *dynamic path*. The static path is loaded at the start of each MATLAB session and cannot be changed without restarting MATLAB. The dynamic path can be loaded and modified at any time during a session using MATLAB functions. MATLAB always searches the static path before the dynamic path.

Note Java classes on the static path should not have dependencies on classes on the dynamic path.

You can view these two path segments using the `javaclasspath` function:

```
javaclasspath

       STATIC JAVA PATH

   D:\SysO\Java\util.jar
   D:\SysO\Java\widgets.jar
   D:\SysO\Java\beans.jar
              .
              .

       DYNAMIC JAVA PATH

   C:\Work\Java\ClassFiles
   C:\Work\Java\mywidgets.jar
              .
              .
```

You probably want to use both the static and dynamic paths:

- Put the Java class definitions that are more stable on the static class path. Classes defined on the static path load somewhat faster than those on the dynamic path.
- Put the Java class definitions that you are likely to modify on the dynamic class path. You can make changes to the class definitions on this path without restarting MATLAB.

The Static Path

MATLAB loads the static Java class path from the file `classpath.txt` at the start of each session. The static path offers better Java class loading performance than the dynamic path. However, to modify the static path you need to edit the file `classpath.txt` and then restart MATLAB.

Finding and Editing classpath.txt. The default `classpath.txt` file resides in the `toolbox\local` subdirectory of your MATLAB root directory:

```
[matlabroot '\toolbox\local\classpath.txt']
ans =
   \\sys07\matlab\toolbox\local\classpath.txt
```

To make changes in the static path that affect all users who share this same MATLAB root directory, edit this file in `toolbox\local`. If you want to make changes that do not affect anyone else, copy `classpath.txt` to your own startup directory and edit the file there. When MATLAB starts up, it looks for `classpath.txt` first in your startup directory, and then in the default location. It uses the file it finds first.

To see which `classpath.txt` file is currently being used by your MATLAB environment, use the `which` function:

```
which classpath.txt
```

To edit either the default file or the copy in your own directory, enter the following command in MATLAB:

```
edit classpath.txt
```

Note MATLAB reads `classpath.txt` only upon startup. If you edit `classpath.txt` or change your `.class` files while MATLAB is running, you must restart MATLAB to put those changes into effect.

Special Symbols in classpath.txt. You can designate special tokens or macros in the `classpath.txt` file using a leading dollar sign, (e.g., `$matlabroot` or `$jre_home`). However, this can cause problems if you use this sign in any of your class directory paths. For example, the following path string does not correctly represent the path to a directory named `hello$world`:

```
d:\applications\hello$world
```

You must use two consecutive dollar signs in `classpath.txt` to represent a single `$` character. So, to correctly specify the directory path shown above, you need to use the following text:

```
d:\applications\hello$$world
```

The Dynamic Path

The dynamic Java class path can be loaded at any time during a MATLAB session using the `javaclasspath` function. You can define the dynamic path (using `javaclasspath`), modify the path (using `javaaddpath` and `javarmpath`), and refresh the Java class definitions for all classes on the dynamic path (using `clear` with the `java` keyword) without restarting MATLAB. See the Java function reference pages for more information on how to use these functions.

Although the dynamic path offers more flexibility in changing the path, Java classes on the dynamic path may load more slowly than those on the static path.

Making Java Classes Available to MATLAB

To make your third-party and user-defined Java classes available in MATLAB, place them on either the static or dynamic Java class path, as described in the previous section, "The Java Class Path" on page 7-8.

- For classes you want on the static path, edit the `classpath.txt` file.
- For classes you want on the dynamic path, use either the `javaclasspath` or `javaaddpath` function.

Making Individual (Unpackaged) Classes Available

To make individual classes (classes that are not part of a package) available in MATLAB, specify the full path to the directory you want to use for the `.class` file(s).

For example, to make available your compiled Java classes in the file `d:\work\javaclasses\test.class`, add the following entry to the static or dynamic class path:

```
d:\work\javaclasses
```

To put this directory on the static class path, add the above line to the default copy (in `toolbox\local`) or your own local copy of `classpath.txt`. See "Finding and Editing classpath.txt" on page 7-9.

To put this on the dynamic class path, use the following command:

```
javaaddpath d:\work\javaclasses
```

Making Entire Packages Available

To access one or more classes belonging to a package, you need to make the entire package available to MATLAB. To do this, specify the full path to the *parent directory of the highest level directory* of the package path. This directory is the first component in the package name.

For example, if your Java class package `com.mw.tbx.ini` has its classes in directory `d:\work\com\mw\tbx\ini`, add the following directory to your static or dynamic class path:

```
d:\work
```

Making Classes in a JAR File Available

You can use the `jar` (Java Archive) tool to create a JAR file, containing multiple Java classes and packages in a compressed ZIP format. For information on `jar` and JAR files, consult your Java development documentation or the JavaSoft Web site http://www.javasoft.com. See also "To Learn More About Java Programming" on page 7-4.

To make the contents of a JAR file available for use in MATLAB, specify the full path, *including full filename*, for the JAR file.

Note The `classpath.txt` requirement for JAR files is different than that for `.class` files and packages, for which you do not specify any filename.

For example, to make available the JAR file `e:\java\classes\utilpkg.jar`, add the following file specification to your static or dynamic class path:

```
e:\java\classes\utilpkg.jar
```

Loading Java Class Definitions

Normally, MATLAB loads a Java class automatically when your code first uses it, (for example, when you call its constructor). However, there is one exception you should be aware of.

When you use the `which` function on methods defined by Java classes, the function only acts on the classes currently loaded into the MATLAB working environment. In contrast, `which` always operates on MATLAB classes, whether or not they are loaded.

Determining Which Classes Are Loaded

At any time during a MATLAB session, you can obtain a listing of all the Java classes that are currently loaded. To do so, use the `inmem` function as follows:

```
[M,X,J] = inmem
```

This function returns the list of Java classes in the output argument `J`. (It also returns in `M` the names of all currently loaded M-files, and in `X` the names of all currently loaded MEX-files.)

Here's a sample of output from the `inmem` function:

```
[m,x,j] = inmem;
j =
 'java.awt.Frame'
 'com.mathworks.ide.desktop.MLDesktop'
```

Simplifying Java Class Names

Your MATLAB commands can refer to any Java class by its fully qualified name, which includes its package name. For example, the following are fully qualified names:

- `java.lang.String`
- `java.util.Enumeration`

A fully qualified name can be rather long, making commands and functions, such as constructors, cumbersome to edit and to read. You can refer to classes by the class name alone (without a package name) if you first import the fully qualified name into MATLAB.

The `import` command has the following forms:

```
import pkg_name.*               % Import all classes in package
import pkg_name1.* pkg_name2.*  % Import multiple packages
import class_name               % Import one class
import                          % Display current import list
L = import                      % Return current import list
```

MATLAB adds all classes that you import to a list called the *import list*. You can see what classes are on that list by typing `import`, without any arguments. Your code can refer to any class on the list by class name alone.

When called from a function, `import` adds the specified classes to the import list in effect for that function. When invoked at the command prompt, `import` uses the base import list for your MATLAB environment.

For example, suppose a function contains the following statements:

```
import java.lang.String
import java.util.* java.awt.*
import java.util.Enumeration
```

Code that follows the `import` statements above can now refer to the `String`, `Frame`, and `Enumeration` classes without using the package names.

```
str = String('hello');     % Create java.lang.String object
frm = Frame;               % Create java.awt.Frame object
methods Enumeration        % List java.util.Enumeration methods
```

To `clear` the list of imported Java classes, invoke the command

```
clear import
```

Locating Native Method Libraries

Java classes can dynamically load native methods using the Java method `java.lang.System.loadLibrary("LibFile")`. In order for the JVM to locate the specified library file, the directory containing it must be on the Java Library Path. This path is established when MATLAB launches the JVM at startup, and is based on the contents of the file

```
matlabroot/toolbox/local/librarypath.txt
```

(where *matlabroot* is the MATLAB root directory returned by the MATLAB command `matlabroot`).

You can augment the search path for native method libraries by editing the `librarypath.txt` file. Follow these guidelines when editing this file:

- Specify each new directory on a line by itself.
- Specify only the directory names, not the names of the DLL files. The `loadLibrary` call does this for you.
- To simplify the specification of directories in cross-platform environments, use any of these macros: `$matlabroot`, `$arch`, and `$jre_home`.

Java Classes Contained in a JAR File

You can access Java classes that are contained in a JAR file once you have added the JAR file to either the static or dynamic Java class path. See "The Java Class Path" on page 7-8 for more information on how MATLAB uses the Java class path.

For example, suppose you have a file, `myArchive.jar` in a directory called `work` in your MATLAB root directory. You can construct the path to this file using the `matlabroot` command:

```
[matlabroot '/work/myArchive.jar']
```

Add the JAR file to your dynamic Java class path using the `javaaddpath` function (`fullfile` adds the platform-correct directory separators):

```
javaaddpath(fullfile(matlabroot,'work','myArchive.jar'))
```

You can now call the public methods in the JAR file.

Creating and Using Java Objects

In this section...

Overview

In MATLAB, you create a Java object by calling one of the constructors of that class. You then use commands and programming statements to perform operations on these objects. You can also save your Java objects to a MAT-file and, in subsequent sessions, reload them into MATLAB.

Constructing Java Objects

You construct Java objects in MATLAB by calling the Java class constructor, which has the same name as the class. For example, the following constructor creates a `Frame` object with the title `'Frame A'` and the other properties with their default values.

```
frame = java.awt.Frame('Frame A');
```

Displaying the new object `frame` shows the following.

```
frame =
java.awt.Frame[frame0,0,0,0x0,invalid,hidden,layout=
java.awt.BorderLayout,title=Frame A,resizable,normal]
```

All of the programming examples in this chapter contain Java object constructors. For example, the code in the Example — Reading a URL creates a `java.net.URL` object with the constructor

```
url = java.net.URL(...
     'http://archive.ncsa.uiuc.edu/demoweb/')
```

Using the javaObject Function

Under certain circumstances, you may need to use the `javaObject` function to construct a Java object. The following syntax invokes the Java constructor for class, `class_name`, with the argument list that matches `x1,...,xn`, and returns a new object, `J`.

```
J = javaObject('class_name',x1,...,xn);
```

For example, to construct and return a Java object of class `java.lang.String`, you use

```
strObj = javaObject('java.lang.String','hello');
```

Using the `javaObject` function enables you to:

- Use classes that have names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify the class for an object at run-time, for example, as input from an application user

The default MATLAB constructor syntax requires that no segment of the input class name be longer than `namelengthmax` characters. (A *class name segment* is any portion of the class name before, between, or after a dot. For example, there are three segments in class, `java.lang.String`.) Any class name segment that exceeds `namelengthmax` characters is truncated by MATLAB. In the rare case where you need to use a class name of this length, you must use `javaObject` to instantiate the class.

The `javaObject` function also allows you to specify the Java class for the object being constructed at run-time. In this situation, you call `javaObject` with a string variable in place of the class name argument.

```
class  = 'java.lang.String';
text = 'hello';
strObj = javaObject(class, text);
```

In the usual case, when the class to instantiate is known at development time, it is more convenient to use the MATLAB constructor syntax. For example, to create a `java.lang.String` object, you would use

```
strObj = java.lang.String('hello');
```

Note Typically, you do not need to use `javaObject`. The default MATLAB syntax for instantiating a Java class is somewhat simpler and is preferable for most applications. Use `javaObject` primarily for the two cases described above.

Java Objects Are References in MATLAB

In MATLAB, Java objects are *references* and do not adhere to MATLAB copy-on-assignment and pass-by-value rules. For example,

```
origFrame = java.awt.Frame;
setSize(origFrame, 800, 400);
newFrameRef = origFrame;
```

In the third statement above, the variable `newFrameRef` is a second reference to `origFrame`, not a copy of the object. In any code following the example above, any change to the object at `newFrameRef` also changes the object at `origFrame`. This effect occurs whether the object is changed by MATLAB code, or by Java code.

The following example shows that `origFrame` and `newFrameRef` are both references to the same entity. When the size of the frame is changed via one reference (`newFrameRef`), the change is reflected through the other reference (`origFrame`), as well.

```
setSize(newFrameRef, 1000, 800);

getSize(origFrame)
ans =
java.awt.Dimension[width=1000,height=800]
```

Concatenating Java Objects

You can concatenate Java objects in the same way that you concatenate native MATLAB data types. You use either the `cat` function or the [ ] operators to tell MATLAB to assemble the enclosed objects into a single object.

Concatenating Objects of the Same Class

If all of the objects being operated on are of the same Java class, the concatenation of those objects produces an array of objects from the same class.

In the following example, the `cat` function concatenates two objects of the class `java.awt.Point`. The class of the result is also `java.awt.Point`.

```
point1 = java.awt.Point(24,127);
point2 = java.awt.Point(114,29);

cat(1, point1, point2)
ans =
java.awt.Point[]:
    [1x1 java.awt.Point]
    [1x1 java.awt.Point]
```

Concatenating Objects of Unlike Classes

When you concatenate objects of unlike classes, MATLAB finds one class from which all of the input objects inherit, and makes the output an instance of this class. MATLAB selects the lowest common parent in the Java class hierarchy as the output class.

For example, concatenating objects of `java.lang.Byte`, `java.lang.Integer`, and `java.lang.Double` yields an object of `java.lang.Number`, since this is the common parent to the three input classes.

```
byte = java.lang.Byte(127);
integer = java.lang.Integer(52);
double = java.lang.Double(7.8);

[byte; integer; double]

ans =
java.lang.Number[]:
```

```
    [   127]
    [    52]
    [7.8000]
```

If there is no common, lower level parent, then the resultant class is `java.lang.Object`, which is the root of the entire Java class hierarchy.

```
byte = java.lang.Byte(127);
point = java.awt.Point(24,127);

[byte; point]

ans =
java.lang.Object[]:
    [               127]
    [1x1 java.awt.Point]
```

Saving and Loading Java Objects to MAT-Files

Use the MATLAB `save` function to save a Java object to a MAT-file. Use the `load` function to load it back into MATLAB from that MAT-file. To save a Java object to a MAT-file, and to load the object from the MAT-file, make sure that the object and its class meet all of the following criteria:

- The class implements the Serializable interface (part of the Java API), either directly or by inheriting it from a parent class. Any embedded or otherwise referenced objects must also implement Serializable.
- The definition of the class is not changed between saving and loading the object. Any change to the data fields or methods of a class prevents the loading (deserialization) of an object that was constructed with the old class definition.
- Either the class does not have any transient data fields, or the values in transient data fields of the object to be saved are not significant. Values in transient data fields are never saved with the object.

If you define your own Java classes, or subclasses of existing classes, you can follow the criteria above to enable objects of the class to be saved and loaded in MATLAB. For details on defining classes to support serialization, consult your Java development documentation. (See also “To Learn More About Java Programming” on page 7-4.)

Finding the Public Data Fields of an Object

To list the public fields that belong to a Java object, use the `fieldnames` function, which takes either of these forms.

```
names = fieldnames(obj)
names = fieldnames(obj,'-full')
```

Calling `fieldnames` without `'-full'` returns the names of all the data fields (including inherited) on the object. With the `'-full'` qualifier, `fieldnames` returns the full description of the data fields defined for the object, including type, attributes, and inheritance information.

Suppose, for example, that you constructed a `Frame` object with:

```
frame = java.awt.Frame;
```

To obtain the full description of the data fields on `frame`, you could use the command

```
fieldnames(frame,'-full')
```

Sample output from this command follows:

```
ans =
'static final int WIDTH
    % Inherited from java.awt.image.ImageObserver'
'static final int HEIGHT
    % Inherited from java.awt.image.ImageObserver'
[1x74 char]
'static final int SOMEBITS
    % Inherited from java.awt.image.ImageObserver'
'static final int FRAMEBITS
    % Inherited from java.awt.image.ImageObserver'
'static final int ALLBITS
    % Inherited from java.awt.image.ImageObserver'
'static final int ERROR
    % Inherited from java.awt.image.ImageObserver'
'static final int ABORT
    % Inherited from java.awt.image.ImageObserver'
'static final float TOP_ALIGNMENT
    % Inherited from java.awt.Component'
```

```
'static final float CENTER_ALIGNMENT
    % Inherited from java.awt.Component'
'static final float BOTTOM_ALIGNMENT
    % Inherited from java.awt.Component'
'static final float LEFT_ALIGNMENT
    % Inherited from java.awt.Component'
'static final float RIGHT_ALIGNMENT
    % Inherited from java.awt.Component'
.
.
.
```

Accessing Private and Public Data

Java API classes provide accessor methods you can use to read from and, where allowed, to modify *private* data fields. These are sometimes referred to as *get* and *set* methods, respectively.

Some Java classes have *public* data fields, which your code can read or modify directly. To access these fields, use the syntax `object.field`.

Examples

The `java.awt.Frame` class provides an example of access to both private and public data fields. This class has the read accessor method `getSize`, which returns a `java.awt.Dimension` object. The `Dimension` object has data fields `height` and `width`, which are public and therefore directly accessible. The following example shows MATLAB commands accessing this data.

```
frame = java.awt.Frame;
frameDim = getSize(frame);
height = frameDim.height;
frameDim.width = 42;
```

The programming examples in this chapter also contain calls to data field accessors. For instance, the sample code for “Example — Finding an Internet Protocol Address” on page 7-74 uses calls to accessors on a `java.net.InetAddress` object.

```
hostname = address.getHostName;
ipaddress = address.getHostAddress;
```

Accessing Data from a Static Field

In Java, a static data field is a field that applies to an entire class of objects. Static fields are most commonly accessed in relation to the class name itself in Java. For example, the code below accesses the `WIDTH` field of the `Frame` class by referring to it in relation to the package and class names, `java.awt.Frame`, rather than an object instance.

```
width = java.awt.Frame.WIDTH;
```

In MATLAB, you can use that same syntax. Or you can refer to the `WIDTH` field in relation to an instance of the class. The following example creates an instance of `java.awt.Frame` called `frameObj`, and then accesses the `WIDTH` field using the name `frameObj` rather than the package and class names.

```
frame = java.awt.Frame('Frame A');

width = frame.WIDTH
width =
    1
```

Assigning to a Static Field

You can assign values to static Java fields by using a static `set` method of the class, or by making the assignment in reference to an instance of the class. For more information, see the previous section, “Accessing Data from a Static Field” on page 7-23. You can assign `value` to the field `staticFieldName` in the example below by referring to this field in reference to an instance of the class.

```
objectName = java.className;
objectName.staticFieldName = value;
```

Note MATLAB does not allow assignment to static fields using the class name itself.

Determining the Class of an Object

To find the class of a Java object, use the query form of the MATLAB function, `class`. After execution of the following example, `frameClass` contains the name of the package and class that Java object `frame` instantiates.

```
frameClass = class(frame)
frameClass =
java.awt.Frame
```

Because this form of `class` also works on MATLAB objects, it does not, in itself, tell you whether it is a Java class. To determine the type of class, use the `isjava` function, which has the form

```
x = isjava(obj)
```

`isjava` returns `1` if `obj` is Java, and `0` if it is not.

```
isjava(frame)
ans =
    1
```

To find out whether or not an object is an instance of a specified class, use the `isa` function, which has the form

```
x = isa(obj, 'class_name')
```

`isa` returns `1` if `obj` is an instance of the class named `'class_name'`, and `0` if it is not. Note that `'class_name'` can be a MATLAB built-in or user-defined class, as well as a Java class.

```
isa(frame, 'java.awt.Frame')
ans =
    1
```

Invoking Methods on Java Objects

In this section...

Using Java and MATLAB Calling Syntax

To call methods on Java objects, you can use the Java syntax

```
object.method(arg1,...,argn)
```

In the following example, `frame` is the `java.awt.Frame` object created above, and `getTitle` and `setTitle` are methods of that object.

```
frame.setTitle('Sample Frame')

title = frame.getTitle
title =
Sample Frame
```

Alternatively, you can call Java object (nonstatic) methods with the MATLAB syntax

```
method(object, arg1,...,argn)
```

With MATLAB syntax, the `java.awt.Frame` example above becomes

```
setTitle(frame, 'Sample Frame')

title = getTitle(frame)
title =
Sample Frame
```

All of the programming examples in this chapter contain invocations of Java object methods. For example, the code for "Example — Reading a URL" on page 7-71 contains a call, using MATLAB syntax, to the `openStream` method on a `java.net.URL` object, `url`.

```
is = openStream(url)
```

In another example, the code for "Example — Creating and Using a Phone Book" on page 7-82 contains a call, using Java syntax, to the `load` method on a `java.utils.Properties` object, `pb_htable`.

```
pb_htable.load(FIS);
```

Using the javaMethod Function on Nonstatic Methods

Under certain circumstances, you may need to use the `javaMethod` function to call a Java method. The following syntax invokes the method, `method_name`, on Java object `J` with the argument list that matches `x1,...,xn`. This returns the value `X`.

```
X = javaMethod('method_name',J,x1,...,xn);
```

For example, to call the `startsWith` method on a `java.lang.String` object passing one argument, use

```
gAddress = java.lang.String('Four score and seven years ago');
str = java.lang.String('Four score');

javaMethod('startsWith', gAddress, str)
ans =
     1
```

Using the `javaMethod` function enables you to

- Use methods that have names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify the method you want to invoke at run time, for example, as input from an application user.

The only way to invoke a method whose name is longer than `namelengthmax` characters is to use `javaMethod`. The Java and MATLAB calling syntax does not accept method names of this length.

With `javaMethod`, you can also specify the method to be invoked at run time. In this situation, your code calls `javaMethod` with a string variable in place of the `method_name` argument. When you use `javaMethod` to invoke a static method, you can also use a string variable in place of the class name argument.

Note Typically, you do not need to use `javaMethod`. The default MATLAB syntax for invoking a Java method is somewhat simpler and is preferable for most applications. Use `javaMethod` primarily for the two cases described above.

Invoking Static Methods on Java Classes

To invoke a `static` method on a Java class, use the Java invocation syntax

```
class.method(arg1,...,argn)
```

For example, call the `isNaN` static method on the `java.lang.Double` class.

```
java.lang.Double.isNaN(2.2)
```

Alternatively, you can apply static method names to instances of a class. In this example, the `isNaN` static method is referenced in relation to the `dblObject` instance of the `java.lang.Double` class.

```
dblObject = java.lang.Double(2.2);

dblObject.isNaN
ans =
     0
```

Several of the programming examples in this chapter contain examples of static method invocation. For example, the code in the Example — Communicating Through a Serial Port contains a call to static method `getPortIdentifier` on Java class `javax.comm.CommPortIdentifier`.

```
commPort =
javax.comm.CommPortIdentifier.getPortIdentifier('COM1');
```

Using the javaMethod Function on Static Methods

The `javaMethod` function was introduced in section "Using the javaMethod Function on Nonstatic Methods" on page 7-26. You can also use this function to call static methods.

The following syntax invokes the static method, `method_name`, in class, `class_name`, with the argument list that matches `x1,...,xn`. This returns the value `X`.

```
X = javaMethod('method_name','class_name',x1,...,xn);
```

For example, to call the static `isNaN` method of the `java.lang.Double` class on a double value of 2.2, you use

```
javaMethod('isNaN','java.lang.Double',2.2);
```

Using the `javaMethod` function to call static methods enables you to:

- Use methods that have names that exceed the maximum length of a MATLAB identifier. (Call the `namelengthmax` function to obtain the maximum identifier length.)
- Specify method and class names at run-time, for example, as input from an application user.

Obtaining Information About Methods

MATLAB offers several functions to help obtain information related to the Java methods you are working with. You can request a list of all of the methods that are implemented by any class. The list may be accompanied by other method information such as argument types and exceptions. You can also request a listing of every Java class that you loaded into MATLAB that implements a specified method.

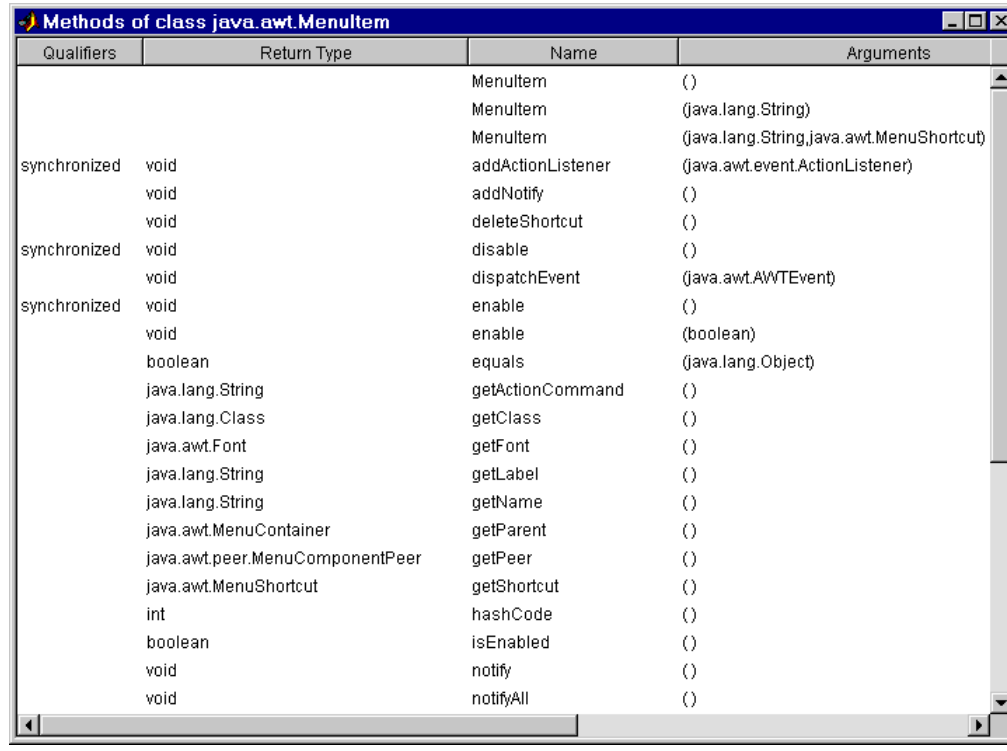
Methodsview: Displaying a Listing of Java Methods

If you want to know what methods are implemented by a particular Java (or MATLAB) class, use the `methodsview` function in MATLAB. Specify the class name (along with its package name, for Java classes) in the command line. If you have imported the package that defines this class, then the class name alone suffices.

The following command lists information on all methods in the `java.awt.MenuItem` class.

```
methodsview java.awt.MenuItem
```

A new window appears, listing one row of information for each method in the class. This is what the `methodsview` display looks like. The field names shown at the top of the window are described following the figure.



Each row in the window displays up to six fields of information describing the method. The table below lists the fields displayed in the `methodsview` window along with a description and examples of each field type.

Fields Displayed in the Methodsview Window

Field Name	Description	Examples
Qualifiers	Method type qualifiers	abstract, synchronized
Return Type	Data type returned by the method	`void`, `java.lang.String`
Name	Method name	`addActionListener`, `dispatchEvent`
Arguments	Arguments passed to method	`boolean`, `java.lang.Object`
Other	Other relevant information	throws `java.io.IOException`
Parent	Parent of the specified class	`java.awt.MenuComponent`

Using the Methods Function on Java Classes

In addition to `methodsview`, the MATLAB `methods` function, which returns information on methods of MATLAB classes, also works on Java classes. You can use any of the following forms of this command.

```
methods class_name
methods class_name -full
n = methods('class_name')
n = methods('class_name','-full')
```

Use `methods` without the `'-full'` qualifier to return the names of all the methods (including inherited methods) of the class. Names of overloaded methods are listed only once.

With the `'-full'` qualifier, `methods` returns a listing of the method names (including inherited methods) along with attributes, argument lists, and inheritance information on each. Each overloaded method is listed separately.

For example, display a full description of all methods of the `java.awt.Dimension` object.

```
methods java.awt.Dimension -full

Methods for class java.awt.Dimension:
Dimension()
Dimension(java.awt.Dimension)
Dimension(int,int)
java.lang.Class getClass()  % Inherited from java.lang.Object
int hashCode()  % Inherited from java.lang.Object
boolean equals(java.lang.Object)
java.lang.String toString()
void notify()  % Inherited from java.lang.Object
void notifyAll()  % Inherited from java.lang.Object
void wait(long) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait(long,int) throws java.lang.InterruptedException
   % Inherited from java.lang.Object
void wait() throws java.lang.InterruptedException
   % Inherited from java.lang.Object
java.awt.Dimension getSize()
void setSize(java.awt.Dimension)
void setSize(int,int)
```

Determining What Classes Define a Method

You can use the `which` function to display the fully qualified name (package and class name) of a method implemented by a *loaded* Java class. With the `-all` qualifier, the `which` function finds all classes with a method of the name specified.

Suppose, for example, that you want to find the package and class name for the `concat` method, with the `String` class currently loaded. Use the command

```
which concat
java.lang.String.concat  % String method
```

If the `java.lang.String` class has not been loaded, the same `which` command would give the output

```
which concat
concat not found.
```

If you use `which -all` for the method `equals`, with the `String` and `java.awt.Frame` classes loaded, you see the following display.

```
which -all equals
java.lang.String.equals                      % String method
java.awt.Frame.equals                        % Frame method
com.mathworks.ide.desktop.MLDesktop.equals   % MLDesktop method
```

The `which` function operates differently on Java classes than it does on MATLAB classes. MATLAB classes are always displayed by `which`, whether or not they are loaded. This is not true for Java classes. You can find out which Java classes are currently loaded by using the command `[m,x,j]=inmem`, described in "Determining Which Classes Are Loaded" on page 7-13.

For a description of how Java classes are loaded, see "Making Java Classes Available to MATLAB" on page 7-11.

Java Methods That Affect MATLAB Commands

MATLAB commands that operate on Java objects and arrays make use of the methods that are implemented within, or inherited by, these objects' classes. There are some MATLAB commands that you can alter somewhat in behavior by changing the Java methods that they rely on.

Changing the Effect of disp and display

You can use the `disp` function to display the value of a variable or an expression in MATLAB. Terminating a command line without a semicolon also calls the `disp` function. You can also use `disp` to display a Java object in MATLAB.

When `disp` operates on a Java object, MATLAB formats the output using the `toString` method of the class to which the object belongs. If the class does not implement this method, then an inherited `toString` method is used. If no intermediate ancestor classes define this method, it uses the `toString` method defined by the `java.lang.Object` class. You can override inherited `toString` methods in classes that you create by implementing such a method

within your class definition. In this way, you can change the way MATLAB displays information regarding the objects of the class.

Changing the Effect of isequal

The MATLAB `isequal` function compares two or more arrays for equality in type, size, and contents. This function can also be used to test Java objects for equality.

When you compare two Java objects using `isequal`, MATLAB performs the comparison using the Java method, `equals`. MATLAB first determines the class of the objects specified in the command, and then uses the `equals` method implemented by that class. If it is not implemented in this class, then an inherited `equals` method is used. This is the `equals` method defined by the `java.lang.Object` class if no intermediate ancestor classes define this method.

You can override inherited `equals` methods in classes that you create by implementing such a method within your class definition. In this way, you can change the way MATLAB performs comparison of the members of this class.

Changing the Effect of double and char

You can also define your own Java methods `toDouble` and `toChar` to change the output of the MATLAB `double` and `char` functions. For more information, see the sections entitled “Converting to the MATLAB double Data Type” on page 7-66 and “Converting to the MATLAB char Data Type” on page 7-67.

How MATLAB Handles Undefined Methods

If your MATLAB command invokes a nonexistent method on a Java object, MATLAB looks for a function with the same name. If MATLAB finds a function of that name, it attempts to invoke it. If MATLAB does not find a function with that name, it displays a message stating that it cannot find a method by that name for the class.

For example, MATLAB has a function named `size`, and the Java API `java.awt.Frame` class also has a `size` method. If you call `size` on a `Frame` object, the `size` method defined by `java.awt.Frame` is executed. However,

if you call `size` on an object of `java.lang.String`, MATLAB does not find a `size` method for this class. It executes the MATLAB `size` function instead.

```
string = java.lang.String('hello');

size(string)
ans =
     1     1
```

Note When you define a Java class for use in MATLAB, avoid giving any of its methods the same name as a MATLAB function.

How MATLAB Handles Java Exceptions

If invoking a Java method or constructor throws an exception, MATLAB catches the exception and transforms it into a MATLAB error. MATLAB puts the text of the Java error message into its own error message. Receiving an error from a Java method or constructor has the same appearance as receiving an error from an M-file.

Method Execution in MATLAB

When calling a `main` method from MATLAB, the method returns as soon as it executes its last statement, even if the method creates a thread that is still executing. In other environments, the `main` method does not return until the thread completes execution.

You, therefore, need to be cautious when calling `main` methods from MATLAB, particularly `main` methods that launch GUIs. `main` methods are usually written assuming they are the entry point to application code. When called from MATLAB this is not the case, and the fact that other Java GUI code might be already running can lead to problems.

Working with Java Arrays

In this section...

Introduction

You can pass singular Java objects to and from methods or you may pass them in an array, providing the method expects them in that form. This array must either be a Java array (returned from another method call or created within MATLAB) or, under certain circumstances, a MATLAB cell array. This section describes how to create and manipulate Java arrays in MATLAB. Later sections will describe how to use MATLAB cell arrays in calls to Java methods.

Note The term *dimension* here refers more to the number of subscripts required to address the elements of an array than to its length, width, and height characteristics. For example, a 5-by-1 array is referred to as being one-dimensional, as its individual elements can be indexed into using only one array subscript.

How MATLAB Represents the Java Array

The term *java array* refers to any array of Java objects returned from a call to a Java class constructor or method. You may also construct a Java array within MATLAB using the `javaArray` function. The structure of a Java array is significantly different from that of a MATLAB matrix or array. MATLAB *hides* these differences whenever possible, allowing you to operate on the

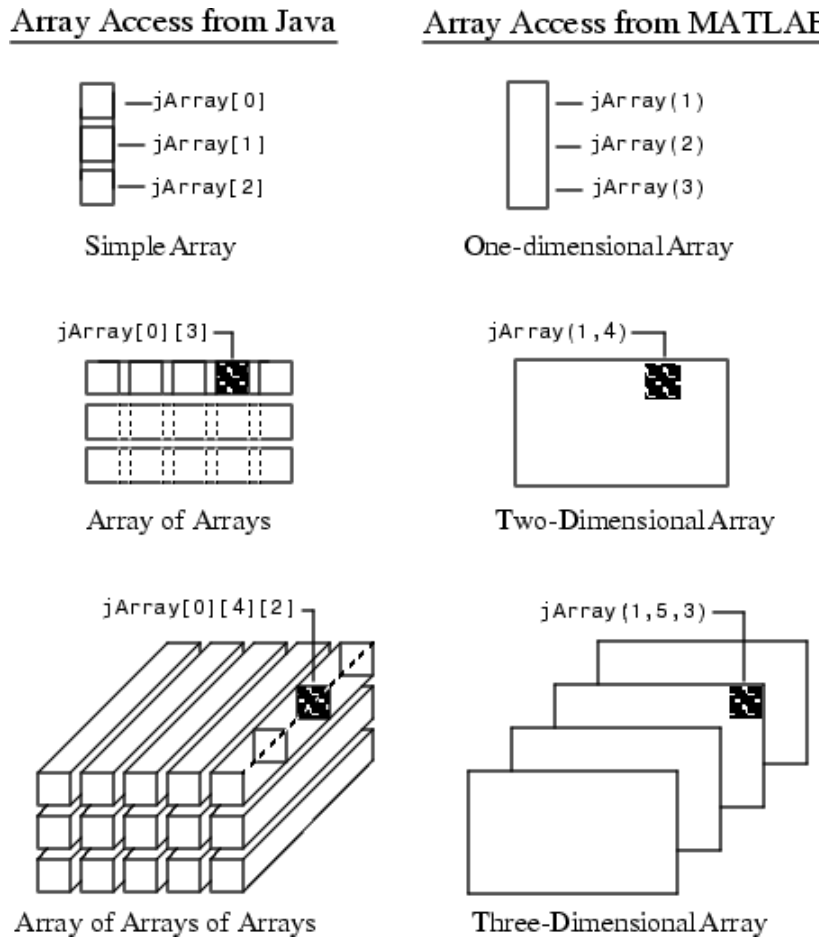
arrays using the usual MATLAB command syntax. Just the same, it may be helpful to keep the following differences in mind as you work with Java arrays.

Representing More Than One Dimension

An array in the Java language is strictly a one-dimensional structure because it is measured only in length. If you want to work with a two-dimensional array, you can create an equivalent structure using an array of arrays. To add further dimensions, you add more levels to the array, making it an array of arrays of arrays, and so on. You may want to use such multilevel arrays when working in MATLAB as it is a matrix and array-based programming language.

MATLAB makes it easy for you to work with multilevel Java arrays by treating them like the matrices and multidimensional arrays that are a part of the language itself. You access elements of an array of arrays using the same MATLAB syntax that you use if you are handling a matrix. If you add more levels to the array, MATLAB can access and operate on the structure as if it is a multidimensional MATLAB array.

The left side of the following figure shows Java arrays of one, two, and three dimensions. To the right of each is the way the same array is represented to you in MATLAB. Note that single-dimension arrays are represented as a column vector.



Array Indexing

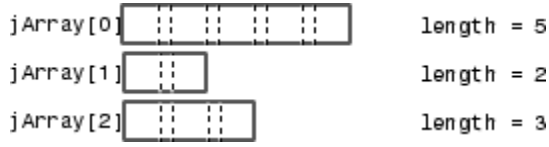
Java array indexing is different than MATLAB array indexing. Java array indices are zero-based, MATLAB array indices are one-based. In Java programming, you access the elements of array `y` of length `N` using `y[0]`

through `y[N-1]`. When working with this array in MATLAB, you access these same elements using the MATLAB indexing style of `y(1)` through `y(N)`. Thus, if you have a Java array of 10 elements, the seventh element is obtained using `y(7)`, and not `y[6]` as you use when writing a program in Java.

The Shape of the Java Array

A Java array can be different from a MATLAB array in its overall *shape*. A two-dimensional MATLAB array maintains a rectangular shape, as each row is of equal length and each column of equal height. The Java counterpart of this, an array of arrays, does not necessarily hold to this rectangular form. Each individual lower level array may have a different length.

Such an array structure is pictured below. This is an array of three underlying arrays of different lengths. The term *ragged* is commonly used to describe this arrangement of array elements as the array ends do not match up evenly. When a Java method returns an array with this type of structure, it is stored in a cell array by MATLAB.

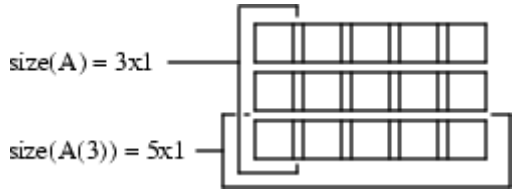


Interpreting the Size of a Java Array

When the MATLAB `size` function is applied to a simple Java array, the number of rows returned is the length of the Java array and the number of columns is always 1.

Determining the size of a Java array of arrays is not so simple. The potentially ragged shape of an array returned from Java makes it impossible to size the array in the same way as for a rectangular matrix. In a ragged Java array, there is no one value that represents the size of the lower level arrays.

When the `size` function is applied to a Java array of arrays, the resulting value describes the top level of the specified array. For the Java array



`size(A)` returns the dimensions of the highest array level of A. The highest level of the array has a size of 3-by-1.

```
size(A)
ans =
     3     1
```

To find the size of a lower level array, say the five-element array in row 3, refer to the row explicitly.

```
size(A(3))
ans =
     5     1
```

You can specify a dimension in the `size` command using the following syntax. However, you will probably find this useful only for sizing the first dimension, `dim=1`, as this will be the only non-unary dimension.

```
m = size(X,dim)

size(A, 1)
ans =
     3
```

Interpreting the Number of Dimensions of a Java Arrays

For Java arrays, whether they are simple one-level arrays or multilevel, the MATLAB `ndims` function always returns a value of 2 to indicate the number of dimensions in the array. This is a measure of the number of dimensions in the top-level array, which always equals 2.

Creating an Array of Objects Within MATLAB

To call a Java method that has one or more arguments defined as an array of Java objects, you must, under most circumstances, pass your objects in a Java array. You can construct an array of objects in a call to a Java method or constructor. Or you can create the array within MATLAB.

The MATLAB `javaArray` function lets you create a Java array structure that can be handled in MATLAB as a single multidimensional array. You specify the number and size of the array dimensions along with the class of objects you intend to store in it. Using the one-dimensional Java array as its primary building block, MATLAB then builds an array structure that satisfies the dimensions requested in the `javaArray` command.

Using the javaArray Function

To create a Java object array, use the MATLAB `javaArray` function, which has the following syntax:

```
A = javaArray('element_class', m, n, p, ...)
```

The first argument is the `'element_class'` string, which names the class of the elements in the array. You must specify the fully qualified name (package and class name). The remaining arguments (`m, n, p, ...`) are the number of elements in each dimension of the array.

An array that you create with `javaArray` is equivalent to the array that you create with the Java code.

```
A = new element_class[m][n][p]...;
```

The following command builds a Java array of four lower level arrays, each capable of holding five objects of the `java.lang.Double` class. (You are more likely to use primitive types of `double` than instances of the `java.lang.Double` class, but in this context, it affords us a simple example.)

```
dblArray = javaArray('java.lang.Double', 4, 5);
```

The `javaArray` function does not deposit any values into the array elements that it creates. You must do this separately. The following MATLAB code stores objects of the `java.lang.Double` type in the Java array `dblArray` that was just created.

```
for m = 1:4
    for n = 1:5
    dblArray(m,n) = java.lang.Double((m*10) + n);
    end
end

dblArray
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

Another Way to Create a Java Array

You can also create an array of Java objects using syntax that is more typical to MATLAB. For example, the following syntax creates a 4-by-5 MATLAB array of type double and assigns zero to each element of the array.

```
matlabArray(4,5) = 0;
```

You use similar syntax to create a Java array in MATLAB, except that you must specify the Java class name. The value being assigned, `0` in this example, is stored in the final element of the array, javaArray`(4,5)`. All other elements of the array receive the empty matrix.

```
javaArray(4,5) = java.lang.Double(0)
javaArray =
java.lang.Double[][]:
     []     []     []     []     []
     []     []     []     []     []
     []     []     []     []     []
     []     []     []     []    [0]
```

Note You cannot change the dimensions of an existing Java array as you can with a MATLAB array. The same restriction exists when working with Java arrays in the Java language. See the example below.

This example first creates a scalar MATLAB array, and then successfully modifies it to be two-dimensional.

```
matlabArray = 0;
matlabArray(4,5) = 0

matlabArray =

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

When you try this with a Java array, you get an error. Similarly, you cannot create an array of Java arrays from a Java array, and so forth.

```
javaArray = java.lang.Double(0);
javaArray(4,5) = java.lang.Double(0);
??? Index exceeds Java array dimensions.
```

Accessing Elements of a Java Array

You can access elements of a Java object array by using the MATLAB array indexing syntax, `A(row,col)`. For example, to access the element of array `dblArray` located at row 3, column 4, use

```
row3_col4 = dblArray(3,4)
row3_col4 =
34.0
```

In Java, this is `dblArray[2][3]`.

You can also use MATLAB array indexing syntax to access an element in an object's data field. Suppose that `myMenuObj` is an instance of a window menu class. This user-supplied class has a data field, `menuItemArray`, which is a Java array of `java.awt.menuItem`. To get element `3` of this array, use the following command.

```
currentItem = myMenuObj.menuItemArray(3)
```

Using Single Subscript Indexing to Access Arrays

Elements of a MATLAB matrix are most commonly referenced using both row and column subscripts. For example, you use `x(3,4)` to reference the array element at the intersection of row 3 and column 4. Sometimes it is more advantageous to use just a single subscript. MATLAB provides this capability (see the section on "Linear Indexing" in MATLAB Programming).

Indexing into a MATLAB matrix using a single subscript references one element of the matrix. Using the MATLAB matrix shown here, `matlabArray(3)` returns a single element of the matrix.

```
matlabArray = [11 12 13 14 15; 21 22 23 24 25; ...
               31 32 33 34 35; 41 42 43 44 45]
matlabArray =
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
    41    42    43    44    45

matlabArray(3)
ans =
    31
```

Indexing this way into a Java array of arrays references an entire subarray of the overall structure. Using the `dblArray` Java array, that looks the same as `matlabArray` shown above, `dblArray(3)` returns the 5-by-1 array that makes up the entire third row.

```
row3 = dblArray(3)
row3 =
java.lang.Double[]:
    [31]
    [32]
    [33]
    [34]
    [35]
```

This is a useful feature of MATLAB because it allows you to specify an entire array from a larger array structure, and then manipulate it as an object.

Using the Colon Operator

Use of the MATLAB `colon` operator (`:`) is supported in subscripting Java array references. This operator works just the same as when referencing the contents of a MATLAB array. Using the Java array of `java.lang.Double` objects shown here, the statement `dblArray(2,2:4)` refers to a portion of the lower level array, `dblArray(2)`. A new array, `row2Array`, is created from the elements in columns 2 through 4.

```
dblArray
dblArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]    [15]
    [21]    [22]    [23]    [24]    [25]
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]

row2Array = dblArray(2,2:4)
row2Array =
java.lang.Double[]:
    [22]
    [23]
    [24]
```

You also can use the colon operator in single-subscript indexing, as covered in “Using Single Subscript Indexing to Access Arrays” on page 7-43. By making your subscript a colon rather than a number, you can convert an array of arrays into one linear array. The following example converts the 4-by-5 array `dblArray` into a 20-by-1 linear array.

```
linearArray = dblArray(:)
linearArray =
java.lang.Double[]:
    [11]
    [12]
    [13]
    [14]
    [15]
    [21]
    [22]
     .
     .
     .
```

This works the same way on an N-dimensional Java array structure. Using the colon operator as a single subscripted index into the array produces a linear array composed of all of the elements of the original array.

Note Java and MATLAB arrays are stored differently in memory. This is reflected in the order they are given in a linear array. Java array elements are stored in an order that matches the *rows* of the matrix, (`11`, `12`, `13`, `...` in the array shown above). MATLAB array elements are stored in an order that matches the *columns*, (`11`, `21`, `31`, `...`).

Using END in a Subscript

You can use the `end` keyword in the first subscript of an access statement. The first subscript references the top-level array in a multilevel Java array structure.

Note Using `end` on lower level arrays is not valid due to the potentially ragged nature of these arrays (see “The Shape of the Java Array” on page 7-38). In this case, there is no consistent end value to be derived.

The following example displays data from the third to the last row of Java array `dblArray`.

```
last2rows = dblArray(3:end, :)
last2rows =
java.lang.Double[][]:
    [31]    [32]    [33]    [34]    [35]
    [41]    [42]    [43]    [44]    [45]
```

Assigning to a Java Array

You assign values to objects in a Java array in essentially the same way as you do in a MATLAB array. Although Java and MATLAB arrays are structured quite differently, you use the same command syntax to specify which elements you want to assign to. See "Introduction" on page 7-35 for more information on Java and MATLAB array differences.

The following example deposits the value `300` in the `dblArray` element at row 3, column 2. In Java, this is `dblArray[2][1]`.

```
dblArray(3,2) = java.lang.Double(300)
dblArray =
java.lang.Double[][]:
    [11]    [ 12]    [13]    [14]    [15]
    [21]    [ 22]    [23]    [24]    [25]
    [31]    [300]    [33]    [34]    [35]
    [41]    [ 42]    [43]    [44]    [45]
```

You use the same syntax to assign to an element in an object's data field. Continuing with the `myMenuObj` example shown in "Accessing Elements of a Java Array" on page 7-42, you assign to the third menu item in `menuItemArray` as follows.

```
myMenuObj.menuItemArray(3) = java.lang.String('Save As...');
```

Using Single Subscript Indexing for Array Assignment

You can use a single-array subscript to index into a Java array structure that has more than one dimension. Refer to "Using Single Subscript Indexing to Access Arrays" on page 7-43 for a description of this feature as used with Java arrays.

You can use single-subscript indexing to assign values to an array as well. The example below assigns a one-dimensional Java array, `onedimArray`, to a row of a two-dimensional Java array, `dblArray`. Start out by creating the one-dimensional array.

```
onedimArray = javaArray('java.lang.Double', 5);
for k = 1:5
    onedimArray(k) = java.lang.Double(100 * k);
    end
```

Since `dblArray(3)` refers to the 5-by-1 array displayed in the third row of `dblArray`, you can assign the entire, similarly dimensioned, 5-by-1 `onedimArray` to it.

```
dblArray(3) = onedimArray
dblArray =
java.lang.Double[][]:
    [ 11]    [ 12]    [ 13]    [ 14]    [ 15]
    [ 21]    [ 22]    [ 23]    [ 24]    [ 25]
    [100]    [200]    [300]    [400]    [500]
    [ 41]    [ 42]    [ 43]    [ 44]    [ 45]
```

Assigning to a Linear Array

You can assign a value to *every* element of a multidimensional Java array by treating the array structure as if it were a single linear array. This entails replacing the single, numerical subscript with the MATLAB colon operator. If you start with the `dblArray` array, you can initialize the contents of every object in the two-dimensional array with the following statement.

```
dblArray(:) = java.lang.Double(0)
dblArray =
java.lang.Double[][]:
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
    [0]    [0]    [0]    [0]    [0]
```

You can use the MATLAB colon operator as you would when working with MATLAB arrays. The statements below assign given values to each of the four rows in the Java array, `dblArray`. Remember that each row actually represents a separate Java array in itself.

```
dblArray(1,:) = java.lang.Double(125);
dblArray(2,:) = java.lang.Double(250);
dblArray(3,:) = java.lang.Double(375);
dblArray(4,:) = java.lang.Double(500)
dblArray =
java.lang.Double[][]:
    [125]    [125]    [125]    [125]    [125]
    [250]    [250]    [250]    [250]    [250]
    [375]    [375]    [375]    [375]    [375]
    [500]    [500]    [500]    [500]    [500]
```

Assigning the Empty Matrix

When working with MATLAB arrays, you can assign the empty matrix, (i.e., the 0-by-0 array denoted by []) to an element of the array. For Java arrays, you can also assign [] to array elements. This stores the `NULL` value, rather than a 0-by-0 array, in the Java array element.

Subscripted Deletion

When you assign the empty matrix value to an entire row or column of a MATLAB array, you find that MATLAB actually removes the affected row or column from the array. In the example below, the empty matrix is assigned to all elements of the fourth column in the MATLAB matrix, `matlabArray`. Thus, the fourth column is completely eliminated from the matrix. This changes its dimensions from 4-by-5 to 4-by-4.

```
matlabArray = [11 12 13 14 15; 21 22 23 24 25; ...
               31 32 33 34 35; 41 42 43 44 45]
matlabArray =
    11    12    13    14    15
    21    22    23    24    25
    31    32    33    34    35
    41    42    43    44    45

matlabArray(:,4) = []
matlabArray =
    11    12    13    15
    21    22    23    25
    31    32    33    35
    41    42    43    45
```

You can assign the empty matrix to a Java array, but the effect is different. The next example shows that, when the same operation is performed on a Java array, the structure is not collapsed; it maintains its 4-by-5 dimensions.

```
dblArray(:,4) = []
dblArray =
java.lang.Double[][]:
    [125]    [125]    [125]     []    [125]
    [250]    [250]    [250]     []    [250]
    [375]    [375]    [375]     []    [375]
    [500]    [500]    [500]     []    [500]
```

The `dblArray` data structure is actually an array of five-element arrays of `java.lang.Double` objects. The empty array assignment placed the `NULL` value in the fourth element of each of the lower level arrays.

Concatenating Java Arrays

You can concatenate arrays of Java objects in the same way as arrays of other data types. Java objects, however, can only be catenated along the first or second axis. To understand how scalar Java objects are concatenated in MATLAB, see “Concatenating Java Objects” on page 7-19.

Use either the `cat` function or the square bracket (`[]`) operators. This example horizontally concatenates two Java arrays: `d1` and `d2`.

```
% Construct a 2-by-3 array of java.lang.Double.
d1 = javaArray('java.lang.Double',2,3);
for m = 1:3     for n = 1:3
d1(m,n) = java.lang.Double(n*2 + m-1);
end;            end;

d1
d1 =
java.lang.Double[][]:
    [2]    [4]    [6]
    [3]    [5]    [7]
    [4]    [6]    [8]

% Construct a 2-by-2 array of java.lang.Double.
d2 = javaArray('java.lang.Double',2,2);
for m = 1:3     for n = 1:2
d2(m,n) = java.lang.Double((n+3)*2 + m-1);
end;            end;

d2
d2 =
java.lang.Double[][]:
    [ 8]    [10]
    [ 9]    [11]
    [10]    [12]

% Concatenate the two along the second dimension.
d3 = cat(2,d1,d2)
d3 =
java.lang.Double[][]:
    [2]    [4]    [6]    [ 8]    [10]
    [3]    [5]    [7]    [ 9]    [11]
    [4]    [6]    [8]    [10]    [12]
```

Creating a New Array Reference

Because Java arrays in MATLAB are *references*, assigning an array variable to another variable results in a second reference to the array.

Consider the following example where two separate array variables reference a common array. The original array, `origArray`, is created and initialized.

The statement `newArrayRef = origArray` creates a copy of this array variable. Changes made to the array referred to by `newArrayRef` also show up in the original array.

```
origArray = javaArray('java.lang.Double', 3, 4);
for m = 1:3
   for n = 1:4
      origArray(m,n) = java.lang.Double((m * 10) + n);
   end
end

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]

%  ----- Make a copy of the array reference -----
newArrayRef = origArray;
newArrayRef(3,:) = java.lang.Double(0);

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [ 0]    [ 0]    [ 0]    [ 0]
```

Creating a Copy of a Java Array

You can create an entirely new array from an existing Java array by indexing into the array to describe a block of elements, (or subarray), and assigning this subarray to a variable. The assignment copies the values in the original array to the corresponding cells of the new array.

As with the example in section “Creating a New Array Reference” on page 7-50, an original array is created and initialized. But, this time, a copy is made of the array contents rather than copying the array reference. Changes made using the reference to the new array do not affect the original.

```
origArray = javaArray('java.lang.Double', 3, 4);
for m = 1:3
   for n = 1:4
      origArray(m,n) = java.lang.Double((m * 10) + n);
   end
end

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]

%  ----- Make a copy of the array contents -----
newArray = origArray(:,:);
newArray(3,:) = java.lang.Double(0);

origArray
origArray =
java.lang.Double[][]:
    [11]    [12]    [13]    [14]
    [21]    [22]    [23]    [24]
    [31]    [32]    [33]    [34]
```

Passing Data to a Java Method

In this section...

Introduction

When you make a call from MATLAB to Java code, any MATLAB data types you pass in the call are converted to data types native to the Java language. MATLAB performs this conversion on each argument that is passed, except for those arguments that are already Java objects. This section describes the conversion that is performed on specific MATLAB data types and, at the end, also takes a look at how argument types affect calls made to overloaded methods.

If data is to be returned by the method being called, MATLAB receives this data and converts it to the appropriate MATLAB format wherever necessary. This process is covered in "Handling Data Returned from a Java Method" on page 7-64.

Conversion of MATLAB Argument Data

MATLAB data, passed as arguments to Java methods, are converted by MATLAB into data types that best represent the data to the Java language. The table below shows all of the MATLAB base types for passed arguments and the Java base types defined for input arguments. Each row shows a MATLAB type followed by the possible Java argument matches, from left to right in order of closeness of the match. The MATLAB types (except cell arrays) can all be scalar (1-by-1) arrays or matrices. All of the Java types can be scalar values or arrays.

Conversion of MATLAB Types to Java Types

MATLAB Argument	Closest Type (7)	Java Input Argument (Scalar or Array)					Least Close Type (1)
logical	boolean	byte	short	int	long	float	double
double	double	float	long	int	short	byte	boolean
single	float	double	N/A	N/A	N/A	N/A	N/A
char	String	char	N/A	N/A	N/A	N/A	N/A
uint8	byte	short	int	long	float	double	N/A
uint16	short	int	long	float	double	N/A	N/A
uint32	int	long	float	double	N/A	N/A	N/A
int8	byte	short	int	long	float	double	N/A
int16	short	int	long	float	double	N/A	N/A
int32	int	long	float	double	N/A	N/A	N/A
cell array of strings	array of String	N/A	N/A	N/A	N/A	N/A	N/A
Java object	Object	N/A	N/A	N/A	N/A	N/A	N/A
cell array of object	array of Object	N/A	N/A	N/A	N/A	N/A	N/A
MATLAB object	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Data type conversion of arguments passed to Java code are discussed in the following three categories. MATLAB handles each category differently.

- "Passing Built-In Data Types" on page 7-55
- "Passing String Arguments" on page 7-56
- "Passing Java Objects" on page 7-57

Passing Built-In Data Types

Java has eight data types that are intrinsic to the language and are not represented as Java objects. These are often referred to as *built-in*, or *elemental*, data types and they include `boolean`, `byte`, `short`, `long`, `int`, `double`, `float`, and `char`. MATLAB converts its own data types to these Java built-in types according to the table, Conversion of MATLAB Types to Java Types on page 7-54. Built-in types are in the first 10 rows of the table.

When a Java method you are calling expects one of these data types, you can pass it the type of MATLAB argument shown in the left-most column of the table. If the method takes an array of one of these types, you can pass a MATLAB array of the data type. MATLAB converts the data type of the argument to the type assigned in the method declaration.

The MATLAB code shown below creates a top-level window frame and sets its dimensions. The call to `setBounds` passes four MATLAB scalars of the `double` type to the inherited Java Frame method, `setBounds`, that takes four arguments of the `int` type. MATLAB converts each 64-bit double data type to a 32-bit integer prior to making the call. Shown here is the `setBounds` method declaration followed by the MATLAB code that calls the method.

```
public void setBounds(int x, int y, int width, int height)

frame=java.awt.Frame;
frame.setBounds(200,200,800,400);
frame.setVisible(1);
```

Passing Built-In Types in an Array

To call a Java method with an argument defined as an *array* of a built-in type, you can create and pass a MATLAB matrix with a compatible base type. The following code defines a polygon by sending four `x` and `y` coordinates to the Polygon constructor. Two 1-by-4 MATLAB arrays of `double` are passed to `java.awt.Polygon`, which expects `integer` arrays in the first two arguments. Shown here is the Java method declaration followed by MATLAB code that calls the method, and then verifies the set coordinates.

```
public Polygon(int xpoints[], int ypoints[], int npoints)

poly = java.awt.Polygon([14 42 98 124], [55 12 -2 62], 4);
[poly.xpoints poly.ypoints]       % Verify the coordinates
ans =
14     55
42     12
98    -2
124    62
```

MATLAB Arrays Are Passed by Value

Since MATLAB arrays are passed by value, any changes that a Java method makes to them are not visible to your MATLAB code. If you need to access changes that a Java method makes to an array, then, rather than passing a MATLAB array, you should create and pass a Java array, which is a reference. For a description of using Java arrays in MATLAB, see "Working with Java Arrays" on page 7-35.

Note Generally, it is preferable to have methods return data that has been modified using the return argument mechanism as opposed to passing a reference to that data in an argument list.

Passing String Arguments

To call a Java method that has an argument defined as an object of class `java.lang.String`, you can pass either a `String` object that was returned from an earlier Java call or a MATLAB 1-by-n character array. If you pass the character array, MATLAB converts the array to a Java object of `java.lang.String` for you.

For a programming example, see "Example — Reading a URL" on page 7-71. This shows a MATLAB character array that holds a URL being passed to the Java `URL` class constructor. The constructor, shown below, expects a Java `String` argument.

```
public URL(String spec) throws MalformedURLException
```

In the MATLAB call to this constructor, a character array specifying the URL is passed. MATLAB converts this array to a Java String object prior to calling the constructor.

```
url = java.net.URL(...
     'http://archive.ncsa.uiuc.edu/demoweb/')
```

Passing Strings in an Array

When the method you are calling expects an argument of an array of type `String`, you can create such an array by packaging the strings together in a MATLAB cell array. The strings can be of varying lengths since you are storing them in different cells of the array. As part of the method call, MATLAB converts the cell array to a Java array of `String` objects.

In the following example, the `echoPrompts` method of a user-written class accepts a string array argument that MATLAB converted from its original format as a cell array of strings. The parameter list in the Java method appears as follows:

```
public String[] echoPrompts(String s[])
```

You create the input argument by storing both strings in a MATLAB cell array. MATLAB converts this structure to a Java array of `String`.

```
myaccount.echoPrompts({'Username: ','Password: '})
ans =
'Username: '
'Password: '
```

Passing Java Objects

When calling a method that has an argument belonging to a particular Java class, you must pass an object that is an instance of that class. In the example below, the `add` method belonging to the `java.awt.Menu` class requires, as an argument, an object of the `java.awt.MenuItem` class. The method declaration for this is

```
public MenuItem add(MenuItem mi)
```

The example operates on the frame created in the previous example in “Passing Built-In Data Types” on page 7-55. The second, third, and fourth

lines of code shown here add items to a menu to be attached to the existing window frame. In each of these calls to `menu1.add`, an object that is an instance of the `java.awt.MenuItem` Java class is passed.

```
menu1 = java.awt.Menu('File Options');
menu1.add(java.awt.MenuItem('New'));
menu1.add(java.awt.MenuItem('Open'));
menu1.add(java.awt.MenuItem('Save'));

menuBar=java.awt.MenuBar;
menuBar.add(menu1);
frame.setMenuBar(menuBar);
```

Handling Objects of Class java.lang.Object

A special case exists when the method being called takes an argument of the `java.lang.Object` class. Since this class is the root of the Java class hierarchy, you can pass objects of any class in the argument. The following hash table example passes objects belonging to different classes to a common method, `put`, which expects an argument of `java.lang.Object`. The method declaration for `put` is

```
public synchronized Object put(Object key, Object value)
```

The following MATLAB code passes objects of different types (`boolean`, `float`, and `string`) to the `put` method.

```
hTable = java.util.Hashtable;
hTable.put(0, java.lang.Boolean('TRUE'));
hTable.put(1, java.lang.Float(41.287));
hTable.put(2, java.lang.String('test string'));

hTable                  % Verify hash table contents
hTable =
{1.0=41.287, 2.0=test string, 0.0=true}
```

When passing arguments to a method that takes `java.lang.Object`, it is not necessary to specify the class name for objects of a built-in data type. Line 3, in the example above, specifies that `41.287` is an instance of class `java.lang.Float`. You can omit this and simply say, `41.287`, as shown in the

following example. Thus, MATLAB creates each object for you, choosing the closest matching Java object representation for each argument.

The three calls to `put` from the preceding example can be rewritten as

```
hTable.put(0, 1);
hTable.put(1, 41.287);
hTable.put(2, 'test string');
```

Passing Objects in an Array

The only types of object arrays that you can pass to Java methods are Java arrays and MATLAB cell arrays. MATLAB automatically converts the cell array elements to `java.lang.Object` class objects. Note that in order for a cell array to be passed from MATLAB, the corresponding argument in the Java method signature must specify `java.lang.Object` or an array of `java.lang.Object`.

If the objects are already in a Java array, either an array returned from a Java constructor or constructed in MATLAB by the `javaArray` function, then you simply pass it as the argument to the method being called. No conversion is done by MATLAB, because the argument is already a Java array.

The following example shows the `mapPoints` method of a user-written class accepting an array of `java.awt.Point` objects. The declaration for this method is

```
public Object mapPoints(java.awt.Point p[])
```

The MATLAB code shown below creates a 4-by-1 array containing four Java Point objects. When the array is passed to the `mapPoints` method, no conversion is necessary because the `javaArray` function created a Java array of `java.awt.Point` objects.

```
pointObj = javaArray('java.awt.Point',4);
pointObj(1) = java.awt.Point(25,143);
pointObj(2) = java.awt.Point(31,147);
pointObj(3) = java.awt.Point(49,151);
pointObj(4) = java.awt.Point(52,176);

testData.mapPoints(pointObj);
```

Handling a Cell Array of Java Objects

You create a cell array of Java objects by using the MATLAB syntax `{a1,a2,...}`. You index into a cell array of Java objects in the usual way, with the syntax `a{m,n,...}`.

The following example creates a cell array of two `Frame` objects, `frame1` and `frame2`, and assigns it to variable `frameArray`.

```
frame1 = java.awt.Frame('Frame A');
frame2 = java.awt.Frame('Frame B');

frameArray = {frame1, frame2}
frameArray =
[1x1 java.awt.Frame]    [1x1 java.awt.Frame]
```

The next statement assigns element `{1,2}` of the cell array `frameArray` to variable `f`.

```
f = frameArray {1,2}
f =
java.awt.Frame[frame2,0,0,0x0,invalid,hidden,layout =
java.awt.BorderLayout,resizable,title=Frame B]
```

Other Data Conversion Topics

There are several remaining items of interest regarding the way MATLAB converts its data to a compatible Java type. This includes how MATLAB matches array dimensions, and how it handles empty matrices and empty strings.

How Array Dimensions Affect Conversion

The term *dimension*, as used in this section, refers more to the number of subscripts required to address the elements of an array than to its length, width, and height characteristics. For example, a 5-by-1 array is referred to as having one dimension, because its individual elements can be indexed into using only one array subscript.

In converting MATLAB to Java arrays, MATLAB handles dimension in a special manner. For a MATLAB array, dimension can be considered as the number of nonsingleton dimensions in the array. For example, a 10-by-1 array

has dimension 1, and a 1-by-1 array has dimension 0. In Java, dimension is determined solely by the number of nested arrays. For example, `double[][]` has dimension 2, and `double` has dimension 0.

If the Java array's number of dimensions exactly matches the MATLAB array's number of dimensions `n`, the conversion results in a Java array with `n` dimensions. If the Java array has fewer than `n` dimensions, the conversion drops singleton dimensions, starting with the first one, until the number of remaining dimensions matches the number of dimensions in the Java array.

Empty Matrices and Nulls

The empty matrix is compatible with any method argument for which `NULL` is a legal value in Java. The empty string (`''`) in MATLAB translates into an empty (not `NULL`) `String` object in Java.

Passing Data to Overloaded Methods

When you invoke an overloaded method on a Java object, MATLAB determines which method to invoke by comparing the arguments your call passes to the arguments defined for the methods. Note that in this discussion, the term *method* includes constructors. When it determines the method to call, MATLAB converts the calling arguments to Java method types according to Java conversion rules, except for conversions involving objects or cell arrays. See "Passing Objects in an Array" on page 7-59.

How MATLAB Determines the Method to Call

When your MATLAB function calls a Java method, MATLAB

1. Checks to make sure that the object (or class, for a static method) has a method by that name.
2. Determines whether the invocation passes the same number of arguments of at least one method with that name.
3. Makes sure that each passed argument can be converted to the Java type defined for the method.

If all of the preceding conditions are satisfied, MATLAB calls the method.

In a call to an overloaded method, if there is more than one candidate, MATLAB selects the one with arguments that best fit the calling arguments. First, MATLAB rejects all methods that have any argument types that are incompatible with the passed arguments (for example, if the method has a `double` argument and the passed argument is a `char`).

Among the remaining methods, MATLAB selects the one with the highest fitness value, which is the sum of the fitness values of all its arguments. The fitness value for each argument is the fitness of the base type minus the difference between the MATLAB array dimension and the Java array dimension. (Array dimensionality is explained in "How Array Dimensions Affect Conversion" on page 7-60.) If two methods have the same fitness, the first one defined in the Java class is chosen.

Example — Calling an Overloaded Method

Suppose a function constructs a `java.io.OutputStreamWriter` object, `osw`, and then invokes a method on the object.

```
osw.write('Test data', 0, 9);
```

MATLAB finds that the class `java.io.OutputStreamWriter` defines three `write` methods.

```
public void write(int c);
public void write(char[] cbuf, int off, int len);
public void write(String str, int off, int len);
```

MATLAB rejects the first `write` method, because it takes only one argument. Then, MATLAB assesses the fitness of the remaining two `write` methods. These differ only in their first argument, as explained below.

In the first of these two `write` methods, the first argument is defined with base type, `char`. The table, Conversion of MATLAB Types to Java Types on page 7-54, shows that for the type of the calling argument (MATLAB `char`), Java type, `char`, has a value of 6. There is no difference between the dimension of the calling argument and the Java argument. So the fitness value for the first argument is 6.

In the other `write` method, the first argument has Java type `String`, which has a fitness value of 7. The dimension of the Java argument is 0, so the

difference between it and the calling argument dimension is 1. Therefore, the fitness value for the first argument is 6.

Because the fitness value of those two `write` methods is equal, MATLAB calls the one listed first in the class definition, with `char[]` first argument.

Handling Data Returned from a Java Method

In this section...

Introduction

In many cases, data returned from Java is incompatible with the data types operated on within MATLAB. When this is the case, MATLAB converts the returned value to a data type native to the MATLAB language. This section describes the conversion performed on the various data types that can be returned from a call to a Java method.

Conversion of Java Return Data

The following table lists Java return types and the resulting MATLAB types. For some Java base return types, MATLAB treats scalar and array returns differently, as described following the table.

Conversion of Java Types to MATLAB Types

Java Return Type	If Scalar Return, Resulting MATLAB Type	If Array Return, Resulting MATLAB Type
`boolean`	`logical`	`logical`
`byte`	`double`	`int8`
`short`	`double`	`int16`
`int`	`double`	`int32`
`long`	`double`	`double`
`float`	`double`	`single`

Conversion of Java Types to MATLAB Types (Continued)

Java Return Type	If Scalar Return, Resulting MATLAB Type	If Array Return, Resulting MATLAB Type
`double`	`double`	`double`
`char`	`char`	`char`

Built-In Data Types

Java *built-in* data types are described in "Passing Built-In Data Types" on page 7-55. This data type includes `boolean`, `byte`, `short`, `long`, `int`, `double`, `float`, and `char`. When the value returned from a method call is one of these types, MATLAB converts it according to the table Conversion of Java Types to MATLAB Types on page 7-64.

A single numeric or `boolean` value converts to a 1-by-1 matrix of `double`, which is convenient for use in MATLAB. An array of a numeric or `boolean` return values converts to an array of the closest base type to minimize the required storage space. Array conversions are listed in the right-hand column of the table.

A return value of Java type `char` converts to a 1-by-1 matrix of `char`. An array of Java `char` converts to a MATLAB array of that type.

Java Objects

When a method call returns Java objects, MATLAB leaves them in their original form. They remain as Java objects so you can continue to use them to interact with other Java methods.

The only exception to this is when the method returns data of type `java.lang.Object`. This class is the root of the Java class hierarchy and is frequently used as a catchall for objects and arrays of various types. When the method being called returns a value of the `Object` class, MATLAB converts its value according to the table Conversion of Java Types to MATLAB Types on page 7-64. That is, numeric and `boolean` objects such as `java.lang.Integer` or `java.lang.Boolean` convert to a 1-by-1 MATLAB matrix of `double`.

Object arrays of these types convert to the MATLAB data types listed in the right-hand column of the table. Other object types are not converted.

Converting Objects to MATLAB Data Types

With the exception of objects of class `Object`, MATLAB does not convert Java objects returned from method calls to a native MATLAB data type. If you want to convert Java object data to a form more readily usable in MATLAB, there are a few MATLAB functions that enable you to do this. These are described in the following sections.

Converting to the MATLAB double Data Type

Using the `double` function in MATLAB, you can convert any Java object or array of objects to the MATLAB `double` data type. The action taken by the `double` function depends on the class of the object you specify:

- If the object is an instance of a numeric class (`java.lang.Number` or one of the classes that inherit from that class), MATLAB uses a preset conversion algorithm to convert the object to a MATLAB `double`.
- If the object is not an instance of a numeric class, MATLAB checks the class definition to see if it implements a method called `toDouble`. MATLAB uses `toDouble` to perform its conversion of Java objects to the MATLAB `double` data type. If such a method is implemented for this class, MATLAB executes it to perform the conversion.
- If you are using a class of your own design, you can write your own `toDouble` method to perform conversions on objects of that class to a MATLAB `double`. This enables you to specify your own means of data type conversion for objects belonging to your own classes.

Note If the class of the specified object is not `java.lang.Number`, does not inherit from that `java.lang.Number`, and does not implement a `toDouble` method, then an attempt to convert the object using the `double` function results in a MATLAB error.

The syntax for the `double` command is as follows, where `object` is a Java object or Java array of objects:

```
double(object);
```

Converting to the MATLAB char Data Type

With the MATLAB `char` function, you can convert `java.lang.String` objects and arrays to MATLAB data types. A single `java.lang.String` object converts to a MATLAB character array. An array of `java.lang.String` objects converts to a MATLAB cell array, with each cell holding a character array.

If the object specified in the `char` command is not an instance of the `java.lang.String` class, MATLAB checks its class to see if it implements a method named `toChar`. If this is the case, MATLAB executes the `toChar` method of the class to perform the conversion. If you write your own class definitions, you can make use of this feature by writing a `toChar` method that performs the conversion according to your own needs.

Note If the class of the specified object is not `java.lang.String` and it does not implement a `toChar` method, an attempt to convert the object using the `char` function results in a MATLAB error.

The syntax for the `char` command is as follows, where `object` is a Java object or Java array of objects:

```
char(object);
```

Converting to a MATLAB Structure

Java objects are similar to the MATLAB `structure` in that many of an object's characteristics are accessible via field names defined within the object. You may want to convert a Java object into a MATLAB `structure` to facilitate the handling of its data in MATLAB. Use the MATLAB `struct` function to do this.

The syntax for the `struct` command is as follows, where `object` is a Java object or a Java array of objects:

```
struct(object);
```

The following example converts a `java.awt.Polygon` object into a MATLAB `structure`. You can access the fields of the object directly using MATLAB `structure` operations. The last line indexes into the array, `pstruct.xpoints`, to deposit a new value into the third array element.

```
polygon = java.awt.Polygon([14 42 98 124], [55 12 -2 62], 4);

pstruct = struct(polygon)
pstruct =
    npoints: 4
    xpoints: [4x1 int32]
    ypoints: [4x1 int32]

pstruct.xpoints
ans =
    14
    42
    98
   124

pstruct.xpoints(3) = 101;
```

Converting to a MATLAB Cell Array

Use the `cell` function to convert a Java array or Java object into a MATLAB cell array. Elements of the resulting cell array are of the MATLAB type (if any) closest to the Java array elements or Java object.

The syntax for the `cell` command is as follows, where `object` is a Java object or a Java array of objects.

```
cell(object);
```

The code in the following example creates a MATLAB cell array in which each cell holds an array of a different data type. The `cell` command used in the first line converts each type of object array into a cell array.

```
import java.lang.* java.awt.*;

% Create a Java array of double
dblArray = javaArray('java.lang.Double', 1, 10);
```

```
for m = 1:10
   dblArray(1, m) = Double(m * 7);
end

% Create a Java array of points
ptArray = javaArray('java.awt.Point', 3);
ptArray(1) = Point(7.1, 22);
ptArray(2) = Point(5.2, 35);
ptArray(3) = Point(3.1, 49);

% Create a Java array of strings
strArray = javaArray('java.lang.String', 2, 2);
strArray(1,1) = String('one');    strArray(1,2) = String('two');
strArray(2,1) = String('three');  strArray(2,2) = String('four');

% Convert each to cell arrays
cellArray = {cell(dblArray), cell(ptArray), cell(strArray)}
cellArray =
    {1x10 cell}    {3x1 cell}    {2x2 cell}

cellArray{1,1}       % Array of type double
ans =

   [7]   [14]   [21]   [28]   [35]   [42]   [49]   [56]   [63]   [70]

cellArray{1,2}       % Array of type Java.awt.Point

ans =
F
    [1x1 java.awt.Point]
    [1x1 java.awt.Point]
    [1x1 java.awt.Point]

cellArray{1,3}       % Array of type char array

ans =
    'one'      'two'
    'three'    'four'
```

Introduction to Programming Examples

Each example contains the following sections:

- Overview — Describes what the example does and how it uses the Java interface to accomplish it. Highlighted are the most important Java objects that are constructed and used in the example code.
- Description — provides a detailed description of all code in the example. For longer functions, the description is divided into functional sections that focus on a few statements.
- Running the Example — Shows a sample of the output from execution of the example code.

The example descriptions concentrate on the Java-related functions. For information on other MATLAB programming constructs, operators, and functions used in the examples, see the applicable sections in the MATLAB documentation.

Example — Reading a URL

In this section...

Overview

This program, `URLdemo`, opens a connection to a Web site specified by a URL (Uniform Resource Locator) for the purpose of reading text from a file at that site.

`URLdemo` constructs an object of the Java API class, `java.net.URL`, which enables convenient handling of URLs. Then, it calls a method on the URL object, to open a connection.

To read and display the lines of text at the site, `URLdemo` uses classes from the Java I/O package `java.io`. It creates an `InputStreamReader` object, and then uses that object to construct a `BufferedReader` object. Finally, it calls a method on the `BufferedReader` object to read the specified number of lines from the site.

Description of URLdemo

The major tasks performed by `URLdemo` are:

1. Construct a URL object.

   The example first calls a constructor on `java.net.URL` and assigns the resulting object to variable `url`. The `URL` constructor takes a single argument, the name of the URL to be accessed, as a string. The constructor checks whether the input URL has a valid form.

   ```
   url = java.net.URL(...
   'http://www.mathworks.com/support/tech-notes/1100/1109.shtml')
   ```

2. Open a connection to the URL.

The second statement of the example calls the method, `openStream`, on the `URL` object `url`, to establish a connection with the Web site named by the object. The method returns an `InputStream` object to variable, `is`, for reading bytes from the site.

```
is = openStream(url);
```

3 Set up a buffered stream reader.

The next two lines create a buffered stream reader for characters. The `java.io.InputStreamReader` constructor is called with the input stream `is`, to return to variable `isr` an object that can read characters. Then, the `java.io.BufferedReader` constructor is called with `isr`, to return a `BufferedReader` object to variable `br`. A buffered reader provides for efficient reading of characters, arrays, and lines.

```
isr = java.io.InputStreamReader(is);
br = java.io.BufferedReader(isr);
```

4 Read and display lines of text.

The final statements read the initial lines of HTML text from the site, displaying only the first 4 lines that contain meaningful text. Within the MATLAB `for` statements, the `BufferedReader` method `readLine` reads each line of text (terminated by a `return` and/or line feed character) from the site.

```
for k = 1:288           % Skip initial HTML formatting lines
   s = readLine(br);
end

for k = 1:4             % Read the first 4 lines of text
   s = readLine(br);
   disp(s)
end
```

Running the Example

When you run this example, you see output similar to the following. (Note that the line breaks were changed to fit the output in the documentation).

```
<p>This technical note provides an introduction to vectorization techniques.
```

```
In order to understand some of the possible techniques, an introduction to MATLAB
referencing is provided. Then several vectorization examples are discussed.</p>
<p>This technical note examines how to identify situations where vectorized techniques
would yield a quicker or cleaner algorithm. Vectorization is ofen a smooth process;
however, in many application-specific cases, it can be difficult to construct a vectorized
routine. Understanding the tools and
```

Example — Finding an Internet Protocol Address

In this section...

Overview

The `resolveip` function returns either the name or address of an IP (internet protocol) host. If you pass `resolveip` a host name, it returns the IP address. If you pass `resolveip` an IP address, it returns the host name. The function uses the Java API class `java.net.InetAddress`, which enables you to find an IP address for a host name, or the host name for a given IP address, without making DNS calls.

`resolveip` calls a `static` method on the `InetAddress` class to obtain an `InetAddress` object. Then, it calls accessor methods on the `InetAddress` object to get the host name and IP address for the input argument. It displays either the host name or the IP address, depending on the program input argument.

Description of resolveip

The major tasks performed by `resolveip` are:

1. Create an InetAddress object.

   Instead of constructors, the `java.net.InetAddress` class has `static` methods that return an instance of the class. The `try` statement calls one of those methods, `getByName`, passing the `input` argument that the user has passed to `resolveip`. The input argument can be either a host name or an IP address. If `getByName` fails, the `catch` statement displays an `error` message.

```
function resolveip(input)
try
 address = java.net.InetAddress.getByName(input);
```

```
catch
 error(sprintf('Unknown host %s.', input));
end
```

2 Retrieve the host name and IP address.

The example uses calls to the `getHostName` and `getHostAddress` accessor functions on the `java.net.InetAddress` object, to obtain the host name and IP address, respectively. These two functions return objects of class `java.lang.String`; use the `char` function to convert them to character arrays.

```
hostname = char(address.getHostName);
ipaddress = char(address.getHostAddress);
```

3 Display the host name or IP address.

The example uses the MATLAB `strcmp` function to compare the `input` argument to the resolved IP address. If it matches, MATLAB displays the host name for the Internet address. If the input does not match, MATLAB displays the IP address.

```
if strcmp(input,ipaddress)
 disp(sprintf('Host name of %s is %s', input, hostname));
else
 disp(sprintf('IP address of %s is %s', input, ipaddress));
end;
```

Running the Example

Here is an example of calling the `resolveip` function with a host name.

```
resolveip ('www.mathworks.com')
IP address of www.mathworks.com is 144.212.100.10
```

Here is a call to the function with an IP address.

```
resolveip ('144.212.100.10')
Host name of 144.212.100.10 is www.mathworks.com
```

Example — Communicating Through a Serial Port

In this section...

Overview

The `serialexample` program uses classes of the Java API `javax.comm` package, which support access to communications ports.

After defining port configuration variables, `serialexample` constructs a `javax.comm.CommPortIdentifier` object to manage the serial communications port. The program calls the `open` method on that object to return an object of the `javax.comm.SerialPort` class, which describes the low-level interface to the `COM1` serial port, assumed to be connected to a Tektronix oscilloscope. (The example can be run without an oscilloscope.) The `serialexample` program then calls several methods on the `SerialPort` object to configure the serial port.

The `serialexample` program uses the I/O package `java.io` to write to and read from the serial port. It calls a `static` method to return an `OutputStream` object for the serial port. It then passes that object to the constructor for `java.io.OutputStreamWriter`. It calls the `write` method on the `OutputStreamWriter` object to write a command to the serial port, which sets the contrast on the oscilloscope. It calls `write` again to write a command that checks the contrast. It then constructs an object of the `java.io.InputStreamWriter` class to read from the serial port.

It calls another `static` method on the SerialPort object to return an `OutputStream` object for the serial port. It calls a method on that object to get the number of bytes to read from the port. It passes the InputStream object to the constructor for `java.io.OutputStreamWriter`. Then, if there is data to read, it calls the `read` method on the `OutputStreamWriter` object to read the contrast data returned by the oscilloscope.

Note MATLAB also provides built-in serial port support, described in Chapter 10, “Serial Port I/O”.

Setting Up the Java Environment

Before beginning to run this example, follow the procedure described here to set up your Java environment:

1. Download the Java class `javax.comm` to a local directory. You can download this class from

   ```
   http://java.sun.com/products/javacomm/downloads/index.html
   ```

2. Add the Java class to your Java class path in MATLAB. See “Finding and Editing classpath.txt” on page 7-9.

   For example, if you downloaded the package to *`matlabroot`*`/work/javax`, where *`matlabroot`* is your MATLAB root directory, you need to add the following entry to `classpath.txt`:

   ```
   $matlabroot/work/javax/commapi/comm.jar
   ```

3. Copy the file `win32com.dll` from the `commapi` directory into

   `matlabroot``\sys\java\jre\win32\jre1.5.0\bin`

4. Copy the file `comm.jar` from the `commapi` directory into

   `matlabroot``\sys\java\jre\win32\jre1.5.0\lib\ext`

5. Copy the file `javax.comm.properties` from the `commapi` directory into

   `matlabroot``\sys\java\jre\win32\jre1.5.0\lib`

6. Exit, then restart MATLAB.

Description of Serial Example

The major tasks performed by `serialexample` are:

1 Define variables for serial port configuration and output.

   The first five statements define variables for configuring the serial port. The first statement defines the baud rate to be 9600, the second defines number of data bits to be 8, and the third defines the number of stop bits to be 1. The fourth statement defines parity to be off, and the fifth statement defines flow control (handshaking) to be off.

2 Create a CommPortIdentifier object.

   Instead of constructors, the `javax.comm.CommPortIdentifier` class has `static` methods that return an instance of the class. The example calls one of these, `getPortIdentifier`, to return a `CommPortIdentifier` object for port `COM1`.

```
commPort = ...
javax.comm.CommPortIdentifier.getPortIdentifier('COM1');
```

3 Open the serial port.

   The example opens the serial port by calling `open` on the `CommPortIdentifier` object `commPort`. The `open` call returns a `SerialPort` object, assigning it to `serialPort`. The first argument to `open` is the name (owner) for the port, the second argument is the name for the port, and the third argument is the number of milliseconds to wait for the open.

```
serialPort = open(commPort, 'serial', 1000);
```

4 Configure the serial port.

   The next three statements call configuration methods on the `SerialPort` object `serialPort`. The first statement calls `setSerialPortParams` to set the baud rate, data bits, stop bits, and parity. The next two statements call `setFlowControlMode` to set the flow control, and then `enableReceiveTimeout` to set the timeout for receiving data.

```
setSerialPortParams(serialPort, SerialPort_BAUD_9600,...
SerialPort_DATABITS_8, SerialPort_STOPBITS_1,...
SerialPort_PARITY_NONE);
setFlowControlMode(serialPort, SerialPort_FLOWCTRL_NONE);
enableReceiveTimeout(serialPort, 1000);
```

5 Set up an output stream writer.

The example then calls a constructor to create and open a `java.io.OutputStreamWriter` object. The constructor call passes the `java.io.OutputStream` object, returned by a call to the `getOutputStream` method `serialPort`, and assigns the `OutputStreamWriter` object to `out`.

```
out = java.io.OutputStreamWriter(getOutputStream(serialPort));
```

6 Write data to serial port and close output stream.

The example writes a string to the serial port, by calling `write` on the object `out`. The string is formed by concatenating (with MATLAB [ ] syntax) a command to set the oscilloscope's contrast to 45, with the command terminator that is required by the instrument. The next statement calls `flush` on `out` to flush the output stream.

```
write(out, ['Display:Contrast 45' terminator]);
flush(out);
```

Then, the example again calls `write` on `out` to send another string to the serial port. This string is a query command, to determine the oscilloscope's contrast setting, concatenated with the command terminator. The example then calls `close` on the output stream.

```
write(out, ['Display:Contrast?' terminator]);
close(out);
```

7 Open an input stream and determine number of bytes to read.

To read the data expected from the oscilloscope in response to the contrast query, the example opens an input stream by calling the `static` method, `InputStream.getInputStream`, to obtain an `InputStream` object for the serial port. Then, the example calls the method `available` on the `InputStream` object, `in`, and assigns the returned number of bytes to `numAvail`.

```
in = getInputStream(serialPort);
numAvail = available(in);
```

8 Create an input stream reader for the serial port.

The example then calls a `java.io.InputStreamReader` constructor, with the `InputStream` object, `in`, and assigns the new object to `reader`.

```
reader = java.io.InputStreamReader(in);
```

9 Read data from serial port and close reader.

The example reads from the serial port, by calling the `read` method on the `InputStreamReader` object reader for each available byte. The `read` statement uses MATLAB array concatenation to add each newly read byte to the array of bytes already read. After reading the data, the example calls `close` on `reader` to close the input stream reader.

```
result = [];
for k = 1:numAvail
 result = [result read(reader)];
end
close(reader);
```

10 Close the serial port.

The example closes the serial port, by calling `close` on the `serialPort` object.

```
close(serialPort);
```

11 Convert input argument to a MATLAB character array.

The last statement of the example uses the MATLAB function, `char`, to convert the array input bytes (integers) to an array of characters:

```
result = char(result);
```

Running the serialexample Program

The value of `result` depends upon whether your system's COM1 port is cabled to an oscilloscope. If you have run the example with an oscilloscope, you see the result of reading the serial port.

```
result =
45
```

If you run the example without an oscilloscope attached, there is no data to read. In that case, you see an empty character array.

```
result =
''
```

Example — Creating and Using a Phone Book

In this section...

Overview

The example's main function, `phonebook`, can be called either with no arguments, or with one argument, which is the key of an entry that exists in the phone book. The function first determines the directory to use for the phone book file.

If no phone book file exists, it creates one by constructing a `java.io.FileOutputStream` object, and then closing the output stream. Next, it creates a data dictionary by constructing an object of the Java API class, `java.util.Properties`, which is a subclass of `java.util.Hashtable` for storing key/value pairs in a hash table. For the `phonebook` program, the key is a name, and the value is one or more telephone numbers.

The `phonebook` function creates and opens an input stream for reading by constructing a `java.io.FileInputStream` object. It calls `load` on that object to load the hash table contents, if it exists. If the user passed the key to an entry to look up, it looks up the entry by calling `pb_lookup`, which finds and displays it. Then, the `phonebook` function returns.

If phonebook was called without the name argument, it then displays a textual menu of the available phone book actions:

- Look up an entry
- Add an entry
- Remove an entry
- Change the phone number(s) in an entry
- List all entries

The menu also has a selection to exit the program. The function uses MATLAB functions to display the menu and to input the user selection.

The phonebook function iterates accepting user selections and performing the requested phone book action until the user selects the menu entry to exit. The phonebook function then opens an output stream for the file by constructing a java.io.FileOutputStream object. It calls save on the object to write the current data dictionary to the phone book file. It finally closes the output stream and returns.

Description of Function phonebook

The major tasks performed by phonebook are:

1. Determine the data directory and full filename.

   The first statement assigns the phone book filename, 'myphonebook', to the variable pbname. If the phonebook program is running on a PC, it calls the java.lang.System static method getProperty to find the directory to use for the data dictionary. This is set to the user's current working directory. Otherwise, it uses MATLAB function getenv to determine the directory, using the system variable HOME, which you can define beforehand to anything you like. It then assigns to pbname the full pathname, consisting of the data directory and filename 'myphonebook'.

```
function phonebook(varargin)
pbname = 'myphonebook'; % name of data dictionary
if ispc
   datadir = char(java.lang.System.getProperty('user.dir'));
```

```
else
   datadir = getenv('HOME');
end;
pbname = fullfile(datadir, pbname);
```

2 If needed, create a file output stream.

If the `phonebook` file does not already exist, `phonebook` asks the user whether to create a new one. If the user answers `y`, `phonebook` creates a new phone book by constructing a `FileOutputStream` object. In the `try` clause of a `try-catch` block, the argument `pbname` passed to the `FileOutputStream` constructor is the full name of the file that the constructor creates and opens. The next statement closes the file by calling `close` on the `FileOutputStream` object `FOS`. If the output stream constructor fails, the `catch` statement prints a message and terminates the program.

```
if ~exist(pbname)
   disp(sprintf('Data file %s does not exist.', pbname));
   r = input('Create a new phone book (y/n)?','s');
   if r == 'y',
      try
         FOS = java.io.FileOutputStream(pbname);
         FOS.close
      catch
         error(sprintf('Failed to create %s', pbname));
      end;
   else
      return;
   end;
end;
```

3 Create a hash table.

The example constructs a `java.util.Properties` object to serve as the hash table for the data dictionary.

```
pb_htable = java.util.Properties;
```

4 Create a file input stream.

In a `try` block, the example invokes a `FileInputStream` constructor with the name of the phone book file, assigning the object to `FIS`. If the call fails, the `catch` statement displays an `error` message and terminates the program.

```
try
   FIS = java.io.FileInputStream(pbname);
catch
   error(sprintf('Failed to open %s for reading.', pbname));
   end;
```

5 Load the phone book keys and close the file input stream.

The example calls `load` on the `FileInputStream` object `FIS`, to load the phone book keys and their values (if any) into the hash table. It then closes the file input stream.

```
pb_htable.load(FIS);
FIS.close;
```

6 Display the Action menu and get the user's selection.

Within a `while` loop, several `disp` statements display a menu of actions that the user can perform on the phone book. Then, an `input` statement requests the user's typed selection.

```
while 1
   disp ' '
   disp ' Phonebook Menu:'
   disp ' '
   disp ' 1. Look up a phone number'
   disp ' 2. Add an entry to the phone book'
   disp ' 3. Remove an entry from the phone book'
   disp ' 4. Change the contents of an entry in the phone book'
   disp ' 5. Display entire contents of the phone book'
   disp ' 6. Exit this program'
   disp ' '
   s = input('Please type the number for a menu selection: ','s');
```

7 Invoke the function to perform a phone book action

Still within the `while` loop, a `switch` statement provides a case to handle each user selection. Each of the first five cases invokes the function to perform a phone book action.

Case 1 prompts for a name that is a key to an entry. It calls `isempty` to determine whether the user has entered a name. If a name has not been entered, it calls `disp` to display an error message. If a name has been input, it passes it to `pb_lookup`. The `pb_lookup` routine looks up the entry and, if it finds it, displays the entry contents.

```
switch s
   case '1',
      name = input('Enter the name to look up: ','s');
      if isempty(name)
         disp 'No name entered'
      else
         pb_lookup(pb_htable, name);
      end;
```

Case 2 calls `pb_add`, which prompts the user for a new entry and then adds it to the phone book.

```
    case '2',
       pb_add(pb_htable);
```

Case 3 uses `input` to prompt for the name of an entry to remove. If a name has not been entered, it calls `disp` to display an error message. If a name has been entered, it passes it to `pb_remove`.

```
    case '3',
       name=input('Enter the name of the entry to remove: ', 's');
       if isempty(name)
          disp 'No name entered'
       else
          pb_remove(pb_htable, name);
       end;
```

Case 4 uses `input` to prompt for the name of an entry to change. If a name has not been entered, it calls `disp` to display an error message. If a name has been entered, it passes it to `pb_change`.

```
    case '4',
```

```
        name=input('Enter the name of the entry to change: ', 's');
        if isempty(name)
           disp 'No name entered'
        else
           pb_change(pb_htable, name);
        end;
```

Case 5 calls `pb_listall` to display all entries.

```
     case '5',
        pb_listall(pb_htable);
```

8 Exit by creating an output stream and saving the phone book.

If the user has selected case `6` to exit the program, a `try` statement calls the constructor for a `FileOuputStream` object, passing it the name of the phone book. If the constructor fails, the `catch` statement displays an error message.

If the object is created, the next statement saves the phone book data by calling `save` on the `Properties` object `pb_htable`, passing the `FileOutputStream` object `FOS` and a descriptive header string. It then calls `close` on the `FileOutputStream` object, and returns.

```
     case '6',
        try
           FOS = java.io.FileOutputStream(pbname);
        catch
           error(sprintf('Failed to open %s for writing.',...
                         pbname));
        end;
        pb_htable.save(FOS,'Data file for phonebook program');
        FOS.close;
        return;
     otherwise
        disp 'That selection is not on the menu.'
     end;
  end;
```

Description of Function pb_lookup

Arguments passed to `pb_lookup` are the `Properties` object `pb_htable` and the `name` key for the requested entry. The `pb_lookup` function first calls `get` on `pb_htable` with the `name` key, on which support function `pb_keyfilter` is called to change spaces to underscores. The `get` method returns the entry (or null, if the entry is not found) to variable `entry`. Note that `get` takes an argument of type `java.lang.Object` and also returns an argument of that type. In this invocation, the key passed to `get` and the entry returned from it are actually character arrays.

`pb_lookup` then calls `isempty` to determine whether `entry` is `null`. If it is, it uses `disp` to display an message stating that the name was not found. If `entry` is not `null`, it calls `pb_display` to display the entry.

```
function pb_lookup(pb_htable,name)
entry = pb_htable.get(pb_keyfilter(name));
if isempty(entry),
   disp(sprintf('The name %s is not in the phone book',name));
else
   pb_display(entry);
end
```

Description of Function pb_add

1 Input the entry to add.

The `pb_add` function takes one argument, the `Properties` object `pb_htable`. `pb_add` uses `disp` to prompt for an entry. Using the up arrow (`^`) character as a line delimiter, `input` inputs a name to the variable `entry`. Then, within a while loop, it uses input to get another line of the entry into variable `line`. If the line is empty, indicating that the user has finished the entry, the code breaks out of the while loop. If the line is not empty, the `else` statement appends line to entry and then appends the line delimiter. At the end, the `strcmp` checks the possibility that no input was entered and, if that is the case, returns.

```
function pb_add(pb_htable)
disp 'Type the name for the new entry, followed by Enter.'
disp 'Then, type the phone number(s), one per line.'
disp 'To complete the entry, type an extra Enter.'
```

```
name = input(':: ','s');
entry=[name '^'];
while 1
   line = input(':: ','s');
   if isempty(line)
      break;
   else
      entry=[entry line '^'];
   end;
end;

if strcmp(entry, '^')
   disp 'No name entered'
   return;
end;
```

2 Add the entry to the phone book.

After the input has completed, `pb_add` calls `put` on `pb_htable` with the hash key `name` (on which `pb_keyfilter` is called to change spaces to underscores) and `entry`. It then displays a message that the entry has been added.

```
pb_htable.put(pb_keyfilter(name),entry);
disp ' '
disp(sprintf('%s has been added to the phone book.', name));
```

Description of Function pb_remove

1 Look for the key in the phone book.

Arguments passed to `pb_remove` are the `Properties` object `pb_htable` and the `name` key for the entry to remove. The `pb_remove` function calls `containsKey` on `pb_htable` with the `name` key, on which support function `pb_keyfilter` is called to change spaces to underscores. If `name` is not in the phone book, `disp` displays a message and the function returns.

```
function pb_remove(pb_htable,name)
if ~pb_htable.containsKey(pb_keyfilter(name))
   disp(sprintf('The name %s is not in the phone book',name))
   return
end;
```

2 Ask for confirmation and if given, remove the key.

If the key is in the hash table, pb_remove asks for user confirmation. If the user confirms the removal by entering y, pb_remove calls remove on pb_htable with the (filtered) name key, and displays a message that the entry has been removed. If the user enters n, the removal is not performed and disp displays a message that the removal has not been performed.

```
r = input(sprintf('Remove entry %s (y/n)? ',name), 's');
if r == 'y'
   pb_htable.remove(pb_keyfilter(name));
   disp(sprintf('%s has been removed from the phone book',name))
else
   disp(sprintf('%s has not been removed',name))
end;
```

Description of Function pb_change

1 Find the entry to change, and confirm.

Arguments passed to pb_change are the Properties object pb_htable and the name key for the requested entry. The pb_change function calls get on pb_htable with the name key, on which pb_keyfilter is called to change spaces to underscores. The get method returns the entry (or null, if the entry is not found) to variable entry. pb_change calls isempty to determine whether the entry is empty. If the entry is empty, pb_change displays a message that the name is added to the phone book, and allows the user to enter the phone number(s) for the entry.

If the entry is found, in the else clause, pb_change calls pb_display to display the entry. It then uses input to ask the user to confirm the replacement. If the user enters anything other than y, the function returns.

```
function pb_change(pb_htable,name)
entry = pb_htable.get(pb_keyfilter(name));
if isempty(entry)
   disp(sprintf('The name %s is not in the phone book', name));
   return;
else
   pb_display(entry);
   r = input('Replace phone numbers in this entry (y/n)? ','s');
```

```
    if r ~= 'y'
       return;
    end;
end;
```

2 Input new phone number(s) and change the phone book entry.

`pb_change` uses `disp` to display a prompt for new phone number(s). Then, `pb_change` inputs data into variable `entry`, with the same statements described in "Description of Function pb_lookup" on page 7-88.

Then, to replace the existing entry with the new one, `pb_change` calls `put` on `pb_htable` with the (filtered) key `name` and the new entry. It then displays a message that the entry has been changed.

```
disp 'Type in the new phone number(s), one per line.'
disp 'To complete the entry, type an extra Enter.'
disp(sprintf(':: %s', name));
entry=[name '^'];
while 1
   line = input(':: ','s');
   if isempty(line)
      break;
   else
      entry=[entry line '^'];
   end;
end;
pb_htable.put(pb_keyfilter(name),entry);
disp ' '
disp(sprintf('The entry for %s has been changed', name));
```

Description of Function pb_listall

The `pb_listall` function takes one argument, the `Properties` object `pb_htable`. The function calls `propertyNames` on the `pb_htable` object to return to `enum` a `java.util.Enumeration` object, which supports convenient enumeration of all the keys. In a `while` loop, `pb_listall` calls `hasMoreElements` on `enum`, and if it returns true, `pb_listall` calls `nextElement` on `enum` to return the next key. It then calls `pb_display` to display the key and entry, which it retrieves by calling `get` on `pb_htable` with the key.

```
function pb_listall(pb_htable)
enum = pb_htable.propertyNames;
while enum.hasMoreElements
   key = enum.nextElement;
   pb_display(pb_htable.get(key));
end;
```

Description of Function pb_display

The `pb_display` function takes an argument `entry`, which is a phone book entry. After displaying a horizontal line, `pb_display` calls MATLAB function `strtok` to extract the first line of the entry, up to the line delimiter (`^`), into `t` and the remainder into `r`. Then, within a `while` loop that terminates when `t` is empty, it displays the current line in `t`. Then it calls `strtok` to extract the next line from `r`, into `t`. When all lines have been displayed, `pb_display` indicates the end of the entry by displaying another horizontal line.

```
function pb_display(entry)
disp ' '
disp '-------------------------'
[t,r] = strtok(entry,'^');
while ~isempty(t)
   disp(sprintf(' %s',t));
   [t,r] = strtok(r,'^');
end;
disp '-------------------------'
```

Description of Function pb_keyfilter

The `pb_keyfilter` function takes an argument `key`, which is a name used as a key in the hash table, and either filters it for storage or unfilters it for display. The filter, which replaces each space in the key with an underscore (_), makes the key usable with the methods of `java.util.Properties`.

```
function out = pb_keyfilter(key)
if ~isempty(findstr(key,' '))
   out = strrep(key,' ','_');
else
   out = strrep(key,'_',' ');
end;
```

Running the phonebook Program

In this sample run, a user invokes `phonebook` with no arguments. The user selects menu action `5`, which displays the two entries currently in the phone book (all entries are fictitious). Then, the user selects `2`, to add an entry. After adding the entry, the user again selects `5`, which displays the new entry along with the other two entries.

```
Phonebook Menu:

     1. Look up a phone number
     2. Add an entry to the phone book
     3. Remove an entry from the phone book
     4. Change the contents of an entry in the phone book
     5. Display entire contents of the phone book
     6. Exit this program

Please type the number for a menu selection: 5

-------------------------
 Sylvia Woodland
 (508) 111-3456
-------------------------

-------------------------
 Russell Reddy
 (617) 999-8765
-------------------------

Phonebook Menu:

     1. Look up a phone number
     2. Add an entry to the phone book
     3. Remove an entry from the phone book
     4. Change the contents of an entry in the phone book
     5. Display entire contents of the phone book
     6. Exit this program

Please type the number for a menu selection: 2

Type the name for the new entry, followed by Enter.
```

```
Then, type the phone number(s), one per line.
To complete the entry, type an extra Enter.
:: BriteLites Books
:: (781) 777-6868
::

BriteLites Books has been added to the phone book.

Phonebook Menu:

     1. Look up a phone number
     2. Add an entry to the phone book
     3. Remove an entry from the phone book
     4. Change the contents of an entry in the phone book
     5. Display entire contents of the phone book
     6. Exit this program

Please type the number for a menu selection: 5

-------------------------
 BriteLites Books
 (781) 777-6868
-------------------------

-------------------------
 Sylvia Woodland
 (508) 111-3456
-------------------------

-------------------------
 Russell Reddy
 (617) 999-8765
-------------------------
```

8

COM Support in MATLAB (Windows Only)

The Microsoft Component Object Model, or COM, is a set of object-oriented technologies and tools that enable software developers to integrate application-specific components from different vendors into their own application solution.

With COM, MATLAB can include ActiveX controls or OLE server processes, or you can configure MATLAB as a computational server controlled by your client application programs.

COM support in MATLAB is available only on the Microsoft Windows platform.

MATLAB Automation Server Functions and Properties (p. 8-116)	How to use properties and methods in a MATLAB Automation server
Additional Automation Server Information (p. 8-122)	Starting the MATLAB server, shared and dedicated servers, using MATLAB as a DCOM server
Examples of a MATLAB Automation Server (p. 8-125)	Examples that show how to access a MATLAB Automation server from Visual Basic .NET and C#

Introducing MATLAB COM Integration

In this section...

What is COM?

The *Component Object Model* (*COM*) provides a framework for integrating reusable, binary software components into an application. Because components are implemented with compiled code, the source code can be written in any of the many programming languages that support COM. Upgrades to applications are simplified, as components can simply be swapped without the need to recompile the entire application. In addition, a component's location is transparent to the application, so components can be relocated to a separate process or even a remote system without having to modify the application.

Using COM, developers and end users can select application-specific components produced by different vendors and integrate them into a complete application solution. For example, a single application might require database access, mathematical analysis, and presentation-quality business graphs. Using COM, a developer can choose a database-access component by one vendor, a business graph component by another, and integrate these into a mathematical analysis package produced by yet a third.

MATLAB supports COM integration on the Microsoft Windows platform.

Concepts and Terminology

While the ideas behind COM technology are straightforward, the terminology is not. The meaning of COM terms has changed over time and few concise definitions exit. Here are some terms that you should be familiar with before

reading this chapter. These are not comprehensive definitions. For a complete description of COM, you'll need to consult outside resources.

COM Objects, Clients, and Servers

A COM *object* is a software component that conforms to Microsoft's Component Object Model. COM enforces encapsulation of the object, preventing direct access of its data and implementation. COM objects expose "Interfaces" on page 8-4, which consist of properties, methods and events.

A COM *client* is a program that makes use of COM objects. COM objects that expose functionality for use are called COM *servers*. COM servers can be in-process or out-of-process. An example of an out-of-process server is Microsoft Excel. These configurations are described in "In-Process and Out-of-Process Servers" on page 8-5.

An *ActiveX control* is a type of in-process COM server that requires a control container. ActiveX controls typically have a user interface. An example is the Microsoft Calendar control. A control container is an application capable of hosting ActiveX controls. A MATLAB figure window or a Simulink model are examples of control containers.

MATLAB can be used as either a COM client or COM server.

Interfaces

The functionality of a component is defined by one or more interfaces. To use a COM component, you must learn about its interfaces, and the methods, properties, and events implemented by the component. The component vendor provides this information.

There are two standard COM interfaces:

- `IUnknown` — An interface required by all COM components. All other COM interfaces are derived from `IUnknown`.
- `IDispatch` — An interface that exposes objects, methods and properties to applications that support Automation.

COM Server Types

There are three types of COM servers:

- Automation — A server that supports the OLE Automation standard. Automation servers are based on the `IDispatch` interface. Automation servers can be accessed by clients of all types, including scripting clients.
- Custom — A server that implements an interface directly derived from `IUnknown`. Custom servers are preferred when faster client access is critical.
- Dual — A server that implements a combination of Automation and Custom interfaces.

Programmatic Identifiers

To create an instance of a COM object, you use its programmatic identifier, or *ProgID*. The ProgID is a unique string defined by the component vendor to identify the COM object. You obtain a ProgID from the vendor's documentation.

MATLAB's ProgIDs are

- `Matlab.Application` — Starts a command window Automation server with the version of MATLAB that was most recently used as an Automation server (might not be the latest installed version of MATLAB).
- `Matlab.Autoserver` — Starts a command window Automation server using the most recent version of MATLAB.
- `Matlab.Desktop.Application` — Starts the full desktop MATLAB as an Automation server using the most recent version of MATLAB.

In-Process and Out-of-Process Servers

You can configure a server three ways. MATLAB supports all of these configurations.

In-Process Server. An in-process server is a component implemented as a dynamic link library (DLL) or ActiveX control that runs in the same process as the client application, sharing the same address space. Communication between client and server is relatively fast and simple.

Local Out-of-Process Server. A local out-of-process server is a component implemented as an executable (`EXE`) file that runs in a separate process from the client application. The client and server processes are on the same computer system. This configuration is somewhat slower due to the overhead required when transferring data across process boundaries.

Remote Out-of Process Server. This is a type of out-of-process server; however, the client and server processes are on different systems and communicate over a network. Network communications, in addition to the overhead required for data transfer, can make this configuration slower than the local out-of-process configuration. This configuration runs only on systems that support the *Distributed Component Object Model* (*DCOM*).

The MATLAB COM Client

Using MATLAB as a COM client provides two techniques for developing programs in MATLAB:

- You can include COM components in your MATLAB application (for example, a spreadsheet).
- You can access existing applications that expose objects via Automation.

In a typical scenario, MATLAB creates ActiveX controls in figure windows, which are manipulated by MATLAB through the controls' properties, methods, and events. This is useful because there exists a wide variety of graphical user interface components implemented as ActiveX controls. For example, Internet Explorer exposes objects that you can include in a figure to display an HTML file. There also are treeviews, spreadsheets, and calendars available from a variety of sources.

MATLAB COM clients can access applications that support Automation, such as Excel. In this case, MATLAB creates an Automation server in which to run the application and returns a handle to the primary interface for the object created.

Information about creating and using COM controls and server objects in MATLAB can be found in "MATLAB COM Client Support" on page 8-38.

The MATLAB COM Automation Server

Automation provides an infrastructure whereby applications called automation controllers can access and manipulate (i.e. set properties of or call methods on) shared automation objects that are exported by other applications, called Automation servers. Any Windows program that can be configured as an Automation controller can control MATLAB.

For example, using Visual Basic, you can run a MATLAB demo in a Microsoft PowerPoint presentation. In this case, PowerPoint is the controller and MATLAB is the server.

Information for creating and connecting to a MATLAB Automation server running MATLAB can be found in "MATLAB COM Automation Server Support" on page 8-112.

Registering Controls and Servers

Before using COM objects, you must register their controls and servers. Most are registered by default. However, if you get a new `.ocx`, `.dll`, or other object file for the control or server, you must register the file manually in the Windows registry.

Use the DOS `regsvr32` command to register your file. From the DOS prompt, use the `cd` function to go to the directory where the object file is located. If your object file is an `.ocx` file, type

```
regsvr32 filename.ocx
```

For example, to register the MATLAB control `mwsamp2.ocx`, type

```
cd matlabroot\toolbox\matlab\winfun\win32
regsvr32 mwsamp2.ocx
```

If you encounter problems with this procedure, please consult a Windows manual or contact your local system administrator.

Verifying the Registration

Here are several ways to verify that a control or server is registered. These examples use the MATLAB `mwsamp` control. Refer to your Microsoft product documentation for information about using Microsoft Visual Studio or the Registry Editor.

- Go to the Microsoft Visual Studio .NET 2003 Tools menu and execute the ActiveX control test container. Click **Edit**, insert a new control, and select `MwSamp Control`. If you are able to insert the control without any problems, the control is successfully registered. Note that this method only works on controls.
- Open the Registry Editor by typing `regedit` at the DOS prompt. Search for your control or server object by selecting **Find** from the **Edit** menu. It will likely be in the following structure:

  ```
  HKEY_CLASSES_ROOT/progid
  ```

- Open OLEViewer from the Microsoft Visual Studio .NET 2003 Tools menu. Look in the following structure for your `Control` object:

  ```
  Object Classes : Grouped by Component Category : Control :
  Your_Control_Object_Name (i.e. Object Classes : Grouped by
  Component Category : Control : Mwsamp Control)
  ```

Getting Started with COM

In this section...

Introduction

A COM client is a program that manipulates COM objects. These objects can run in the MATLAB application or can be part of another application that exposes its objects as a programmatic interface to the application.

This section provides examples that show how to use MATLAB as a COM client.

Note You can also access MATLAB as an Automation server from other applications, such as Visual Basic. For information on this technique, see "MATLAB COM Automation Server Support" on page 8-112.

Basic COM Functions

To start using COM objects, you need to create the object and get information about it. This section covers the following topics:

- "Creating an Instance of a COM Object" on page 8-10
- "Getting Information About a Particular COM Control" on page 8-10
- "Getting an Object's ProgID" on page 8-11
- "Registering a Custom Control" on page 8-11

Creating an Instance of a COM Object

Two MATLAB functions enable you to create COM objects:

- `actxcontrol` — Creates an instance of a control in a MATLAB figure.
- `actxserver` — Creates and manipulates objects from MATLAB that are exposed in an application that supports Automation.

Each function returns a *handle* to the object's main interface, which you use to access the object's methods, properties, and events, and any other interfaces it provides.

Getting Information About a Particular COM Control

In general, you can determine what you can do with an object using the `methods`, `get`, and `events` functions.

Information about Methods. To list the methods supported by the object *`handle`*, type

```
handle.methods
```

Information about Properties. To list the properties of the object *`handle`*, type

```
get(handle)
```

To see the value of the property *`PropertyName`*, type

```
get(handle,'PropertyName')
```

Use `set` to change a property value.

Information about Events. To list the events supported by the object *`handle`*, type

```
handle.events
```

For more information on calling syntax, see "Getting Interfaces to the Object" on page 8-50 and "Invoking Commands on a COM Object" on page 8-53. For more information on events, see "Control and Server Events" on page 8-75.

Getting an Object's ProgID

To get the programmatic identifier (`ProgID`) of a COM control that is already registered on your computer, use the `actxcontrollist` command. You can also use the ActiveX Control Selector, displayed with the command `actxcontrolselect`. This interface lets you see instances of the controls installed on your computer.

For more information on using these commands, see "Creating an ActiveX Control" on page 8-39.

Registering a Custom Control

If your MATLAB program uses a custom control (e.g., one that you have created especially for your application), you must register it with the Windows operating system before you can use it. You can do this from your MATLAB program by issuing an operating system command:

```
!regsvr32 /s filename.ocx
```

where *`filename`* is the name of the file containing the control. Using this command in your program enables you to provide custom-made controls that you make available to other users by registering the control on their computer when they run your MATLAB program. You might also want to supply versions of a Microsoft ActiveX control to ensure that all users have the same version.

For more information about registration, see "Registering Controls and Servers" on page 8-7.

Overview of MATLAB COM Client Examples

The following examples illustrate various techniques for using MATLAB as an COM client. Some of the examples use ActiveX controls, which is a specific type of COM object. For a description, see "COM Objects, Clients, and Servers" on page 8-4.

- "Example — Using Internet Explorer in a MATLAB Figure" on page 8-12 — This example uses the ActiveX control exposed by Internet Explorer to add an HTML viewer to a MATLAB Figure, which also contains an axes object for plotting. As the user clicks various graphics objects that are

displayed in the figure (including the figure itself), the documentation of the object's properties is displayed in the viewer.

- "Example — Grid ActiveX Control in a Figure" on page 8-17 — This example puts a spreadsheet-like grid control in a figure and uses the control's mouse-down event to trigger the acquisition of data from the grid and plot the data in the axes.
- "Example — Reading Data from Excel" on page 8-24 — This MATLAB GUI reads data programmatically from an Excel spreadsheet. By running an Automation server, MATLAB can access the objects exposed by Excel, which provides a variety of interfaces to the application.

Example — Using Internet Explorer in a MATLAB Figure

This example uses the ActiveX control `Shell.Explorer`, which is exposed by Microsoft's Internet Explore application, to include an HTML viewer in a MATLAB figure. The figure's window button down function is then used to select a graphics object when the user clicks the graph and load the object's property documentation into the HTML viewer.

Techniques Demonstrated

- Using Internet Explore from a MATLAB ActiveX client program.
- Defining a window button down function that displays HTML property documentation for whatever object the user clicks.
- Defining a resize function for the figure that also resizes the ActiveX object container.

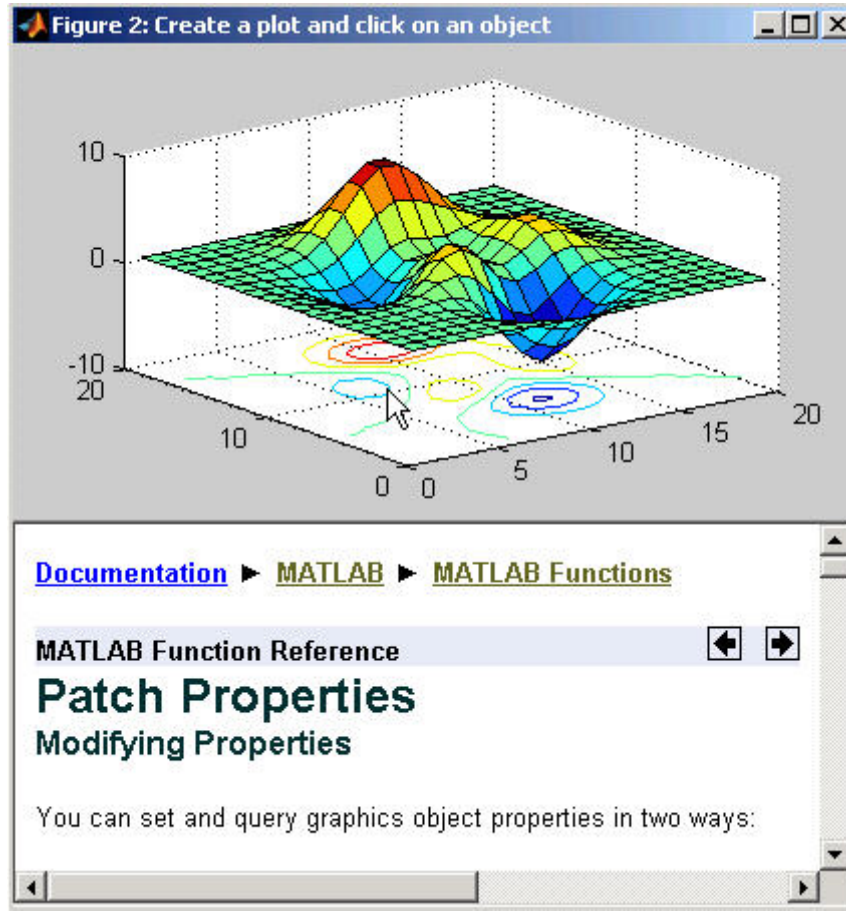
Using the Figure to Access Properties

This example creates a larger than normal figure window that contains an axes object and an HTML viewer on the lower part of the figure window. By default, the viewer displays the URL `http://www.mathworks.com`. When you issue a plotting command, such as

```
surfc(peaks(20))
```

the graph displays in the axes.

Click anywhere in the graph to see the property documentation for the selected object.



Complete Code Listing

You can open the M-file that implements this example in the MATLAB editor or you can run this example with the following links:

- Open M-file in editor
- Run this example

Creating the Figure

This example defines the figure size based on the default figure size and adds space for the ActiveX control. Here is the code to define the figure:

```
dfpos = get(0,'DefaultFigurePosition');
hfig = figure('Position',dfpos([1 2 3 4]).*[.8 .2 1 1.65],...
   'Menu','none','Name','Create a plot and click on an object',...
   'ResizeFcn',@reSize,...
   'WindowButtonDownFcn',@wbdf,...
   'Renderer','Opengl',...
   'DeleteFcn',@figDelete);
```

Note that the figure also defines a resize function and a window button down function by assigning function handles to the `ResizeFcn` and `WindowButtonDownFcn` properties. The callback functions `reSize` and `wbdf` are defined as nested functions in the same M-file.

The figure's `delete` function (called when the figure is closed) provides a mechanism to delete the control.

Calculating the ActiveX Object Container Size

The MATLAB `actxcontrol` function creates the ActiveX control inside the specified figure and returns the control's handle. You need to supply the following information:

- Control's programmatic identifier (use `actxcontrollist` to find it)
- Location and size of the control container in the figure (pixels) [left bottom width height]
- Handle of the figure that contains the control

```
conSize = calcSize;                                % Calculate the container size
hExp = actxcontrol('Shell.Explorer.2',conSize,hfig);  % Create the control
Navigate(hExp,'http://www.mathworks.com/');          % Specify content of html viewer
```

The nested function, `calcSize` calculates the size of the object container based on the current size of the figure. `calcSize` is also used by the figure `resize` function, which is described in the next section.

```
function conSize = calcSize
fp = get(hfig,'Position');              % Get current figure size
conSize = [0 0 1 .45].*fp([3 4 3 4]);   % Calculate container size
end % calcSize
```

Automatic Resize

In MATLAB, you can change the size of a figure and the axes automatically resize to fit the new size. This example implements similar resizing behavior for the ActiveX object container within the figure using the object's `move` method. This method enables you to change both size and location of the ActiveX object container (i.e., it is equivalent to setting the figure `Position` property).

When you resize the figure window, MATLAB automatically calls the function assigned to the figure's `ResizeFcn` property. This example implements the nested function `reSize` for the figure `resize` function.

ResizeFcn at Figure Creation. The `resize` function first determines if the ActiveX object exists because MATLAB calls the figure resize function when the figure is first created. Since the ActiveX object has not been created at this point, the `resize` function simply returns.

When the Figure Is Resized. When you change the size of the figure, the `resize` function executes and does the following:

- Calls the `calcSize` function to calculate a new size for the control container based on the new figure size.
- Calls the control's `move` method to apply the new size to the control.

Figure ResizeFcn.

```
function reSize(src,evnt)
if ~exist('hExp','var')
   return
end
conSize = calcSize;
move(hExp,conSize);
end % reSize
```

Selecting Graphics Objects

This example uses the figure `WindowButtonDownFcn` property to define a callback function that handles mouse click events within the figure. When you click the left mouse button while the cursor is over the figure, MATLAB executes the `WindowButtonDownFcn` callback on the mouse down event.

The callback determines which object was clicked by querying the figure `CurrentObject` property, which contains the handle of the graphics object most recently clicked. Once you have the object's handle, you can determine its type and then load the appropriate HTML page into the `Shell.Explorer` control.

The nested function `wbdf` implements the callback. Once it determines the type of the selected object, it uses the control `Navigate` method to display the documentation for the object type.

Figure WindowButtonDownFcn.

```
function wbdf(src,evnt)
cobj = get(hfig,'CurrentObject');
if isempty(cobj)
   disp('Click somewhere else')
   return
end
pth = 'http://www.mathworks.com/access/helpdesk/help/techdoc/ref/';
typ = get(cobj,'Type');
switch typ
   case ('figure')
      Navigate(hExp,[pth,'figure_props.html']);
   case ('axes')
      Navigate(hExp,[pth,'axes_props.html']);
   case ('line')
      Navigate(hExp,[pth,'line_props.html']);
   case ('image')
      Navigate(hExp,[pth,'image_props.html']);
   case ('patch')
      Navigate(hExp,[pth,'patch_props.html']);
   case ('surface')
      Navigate(hExp,[pth,'surface_props.html']);
   case ('text')
```

```
      Navigate(hExp,[pth,'text_props.html']);
   case ('hggroup')
      Navigate(hExp,[pth,'hggroupproperties.html']);
   otherwise % Display property browser
      Navigate(hExp,[pth(1:end-4),'infotool/hgprop/doc_frame.html']);
end
end % wbdf
```

Closing the Figure

This example uses the figure `delete` function (`DeleteFcn` property) to delete the ActiveX object before closing the figure. MATLAB calls the figure `delete` function before deleting the figure, which enables the function to perform any clean up needed before closing the figure. The figure `delete` function calls the control's `delete` method.

```
function figDelete(src,evnt)
   hExp.delete;
end
```

Example — Grid ActiveX Control in a Figure

This example adds a simple spreadsheet ActiveX control to a figure, which also contains an axes object for plotting the data displayed by the control. Clicking a column in the spreadsheet causes the data in that column to be plotted. Clicking down and dragging the mouse across multiple columns plots all columns touched.

Techniques Demonstrated

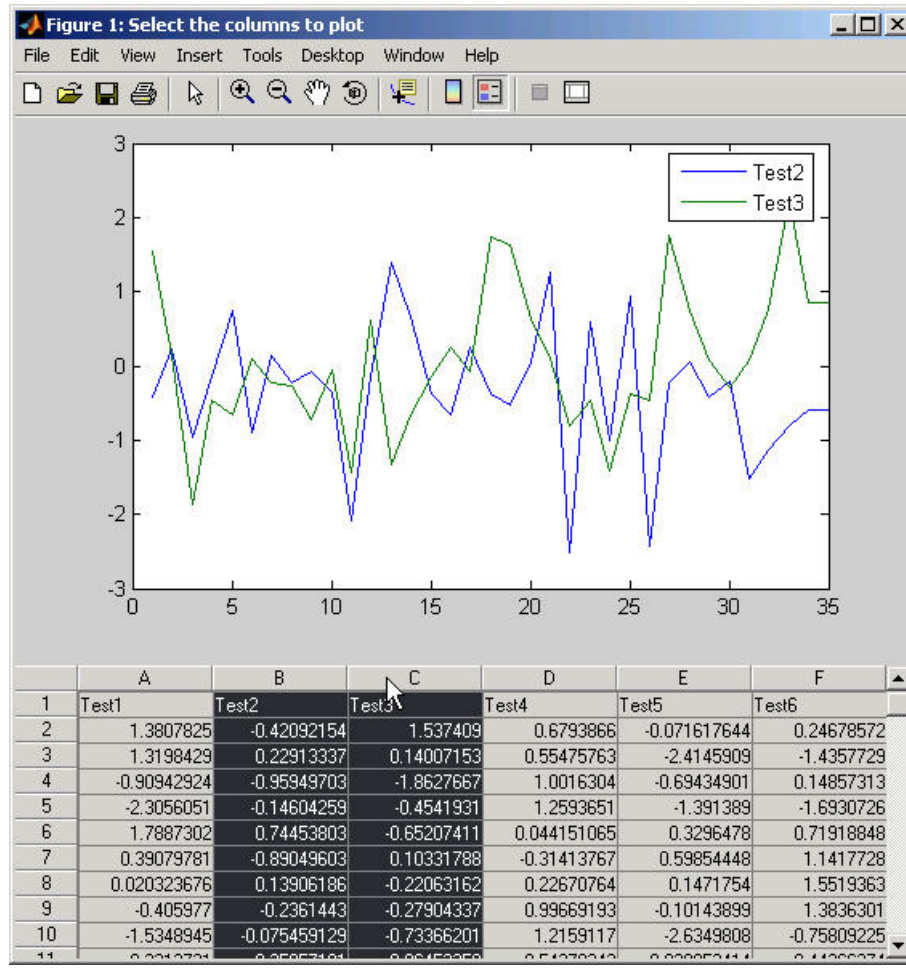
- Registering a control for use on your system.
- Writing a handler for one of the control's events and using the event to execute MATLAB plotting commands.
- Writing a `resize` function for the figure that manages the control's size as users resize the figure.

Using the Control

This example assumes that your data samples are organized in columns and that the first cell in each column is a title, which is used by the legend. See

"Complete Code Listing" on page 8-19 for an example of how to load data into the control.

Once the data is loaded, click the column to plot the data. The following picture shows a graph of the results of `Test2` and `Test3` created by selecting column `B` and dragging and releasing on column `C`.



	A	B	C	D	E	F
1	Test1	Test2	Test3	Test4	Test5	Test6
2	1.3807825	-0.42092154	1.537409	0.6793866	-0.071617644	0.24678572
3	1.3198429	0.22913337	0.14007153	0.55475763	-2.4145909	-1.4357729
4	-0.90942924	-0.95949703	-1.8627667	1.0016304	-0.69434901	0.14857313
5	-2.3056051	-0.14604259	-0.4541931	1.2593651	-1.391389	-1.6930726
6	1.7887302	0.74453803	-0.65207411	0.044151065	0.3296478	0.71918848
7	0.39079781	-0.89049603	0.10331788	-0.31413767	0.59854448	1.1417728
8	0.020323676	0.13906186	-0.22063162	0.22670764	0.1471754	1.5519363
9	-0.405977	-0.2361443	-0.27904337	0.99669193	-0.10143899	1.3836301
10	-1.5348945	-0.075459129	-0.73366201	1.2159117	-2.6349808	-0.75809225

Complete Code Listing

You can open the M-file used to implement this example in the MATLAB editor:

- Open M-file in editor.

Preparing to Use the Control

The ActiveX control used in this example is typical of those downloadable from the Internet. Once you have downloaded the files you need, register the control on your system using the DOS command `regsvr32`. In a command prompt, enter a command of the following form:

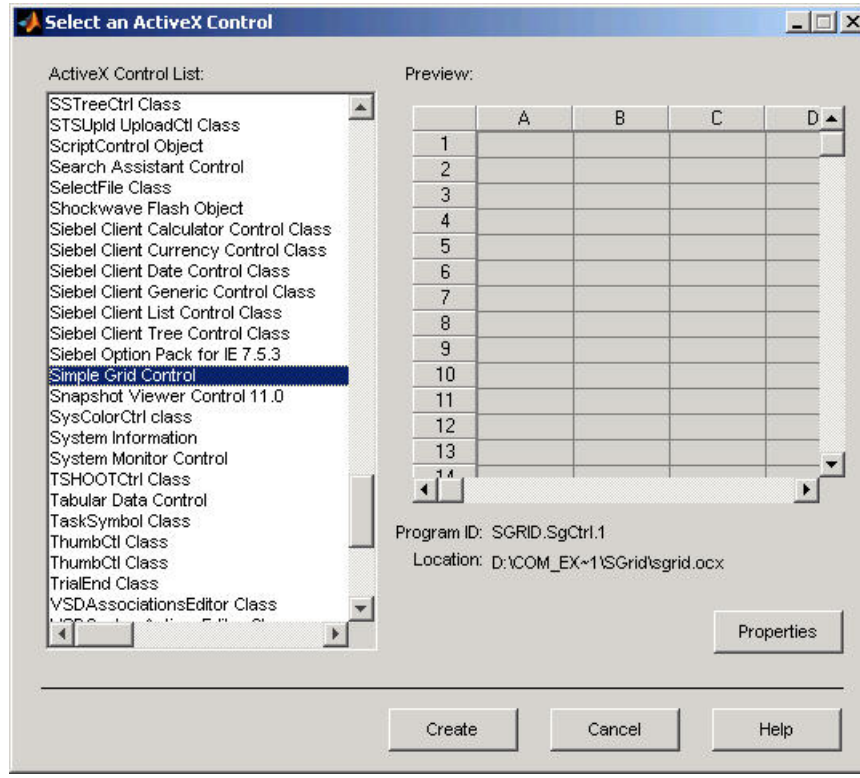
```
regsvr32 sgrid.ocx
```

From MATLAB, use the following command:

```
system 'regsvr32 sgrid.ocx'
```

See the section “Registering Controls and Servers” on page 8-7 for more information.

Finding the Control's ProgID. Once you have installed and registered the control, you can obtain its programmatic identifier using the ActiveX Control Selector dialog. To display this dialog box, use the MATLAB `actxcontrolselect` command. Locate the control in the list and the selector displays the control and the ProgID.



Creating a Figure to Contain the Control

This example creates a figure that contains an axes object and the grid control. The first step is to determine the size of the figure and then create the figure and axes. This example uses the default figure and axes size (obtained from the respective `Position` properties) to calculate a new size and location for each object.

```
dfpos = get(0,'DefaultFigurePosition');
dapos = get(0,'DefaultAxesPosition');
hfig = figure('Position',dfpos([1 2 3 4]).*[1 .8 1 1.25],...
   'Name','Select the columns to plot',...
   'Renderer','ZBuffer',...
   'ResizeFcn',{@reSize dfpos(3)});
hax = axes('Position',dapos([1 2 3 4]).*[1 4 1 .65]);
```

The above code moves the figure down from the top of the screen (multiply second element in position vector by `.8`) and increases the height of the figure (multiply fourth element in position vector by `1.25`). Axes are created and sized in a similar way.

Creating an Instance of the Control

Use the MATLAB `actxcontrol` function to create an instance of the control in a figure window. This function creates a container for the control and enables you to specify the size of this container, which usually defines the size of the control. See “Managing Figure Resize” on page 8-23 for a specific example.

Specifying the Size and Location. The control size and location in the figure is calculated by a nested function `calcSize`. This function is used to calculate both the initial size of the control container and the size resulting from resize of the figure. It gets the figure's current position (i.e., size and location) and scales the four-element vector so that the control container is

- Positioned at the lower-left corner of the figure.
- Equal to the figure in width.
- Has a height that is `.35` times the figure's height.

The value returned is of the correct form to be passed to the `actxcontrol` function and the control's `move` method.

```
function conSize = calcSize
   fp = get(hfig,'Position');
   conSize = fp([3 4 3 4]).*[0 0 1 .35];
end % conSize
```

Creating the Control. Creating the control entails the following steps:

- Calculating the container size
- Instantiating the control in the figure
- Setting the number of rows and columns to match the size of the data array
- Specifying the width of the columns

```
conSize = calcSize;
hgrid = actxcontrol('SGRID.SgCtrl.1',conSize,hfig);
hgrid.NRows = size(dat,1);
hgrid.NColumns = size(dat,2);
colwth = 4350; hdwth = hgrid.HdrWidth;
SetColWidth(hgrid,0,sz(2)-1,colwth,1)
```

Using Mouse-Click Event to Plot Data

This example uses the control's `Click` event to implement interactive plotting. When a user clicks the control, MATLAB executes a function that plots the data in the column where the mouse click occurred. Users can also select multiple columns by clicking down and dragging the cursor over more than one column.

Registering the Event. You need to register events with MATLAB so that when the event occurs (a mouse click in this case), MATLAB responds by executing the event handler function. Register the event with the `registerevent` method:

```
hgrid.registerevent({'Click',@click_event});
```

Pass the event name (`Click`) and a function handle for the event handler function inside a cell array.

Defining the Event Handler. The event handler function `click_event` uses the control's `GetSelection` method to determine what columns and rows have been selected by the mouse click. This function plots the data in the selected columns as lines, one line per column.

It is possible to click down on a column and drag the mouse to select multiple columns before releasing the mouse. In this case, each column is plotted because the event is not fired until the mouse button is released (which reflects the way the author chose to implement the control). The MATLAB

`legend` uses the column number stored in the variable `cols` to label the individual plotted lines. You must add one to `cols` because the control counts the columns starting from zero.

Note that you implement event handlers to accept a variable number of arguments (`varargin`) .

```
function click_event(varargin)
[row1,col1,row2,col2] = hgrid.GetSelection(1,1,1,1,1);
ncols = (col2-col1)+1;
cols = [col1:col2];
   for n = 1:ncols
      hgrid.Col = cols(n);
      for ii = 1:sz(1)
         hgrid.Row = ii;
         plot_data(ii,n) = hgrid.Number;
      end
   end
hgrid.SetSelection(row1,col1,row2,col2);
plot(plot_data)
legend(labels(cols+1))
end % click_event
```

Managing Figure Resize

The size and location of a MATLAB axes object is defined in units that are normalized to the figure that contains it. Therefore, when you resize the figure, the axes automatically resize proportionally. When a figure contains objects that are not contained in axes, you are responsible for defining a function that manages the resize process.

The figure `ResizeFcn` property references a function that executes whenever the figure is resized and also when the figure is first created. This example creates a resize function that manages resizing grid control by doing the following:

- Disables control updates while changes are being made to improve performance (use the `hDisplay` property).
- Calculates a new size for the control container based on the new figure size (`calcSize` function).

- Applies the new size to the control container using its `move` method.
- Scales the column widths of the grid proportional to the change in width of the figure (`SetColWidth` method).
- Refreshes the display of the control, showing the new size.

```
function reSize(src,evnt,dfp)
% Return if control does not exist (figure creation)
if ~exist('hgrid','var')
   return
end
% Resize container
hgrid.bDisplay = 0;
conSize = calcSize;
move(hgrid,conSize);
% Resize columns
scl = conSize(3)/dfp;
ncolwth = scl*colwth;
nhdrwth = hdwth*(scl);
hgrid.HdrWidth = nhdrwth;
SetColWidth(hgrid,0,sz(2)-1,ncolwth,2)
hgrid.Refresh;
end % reSize
```

Closing the Figure

This example uses the figure delete function (`DeleteFcn` property) to delete the ActiveX object before closing the figure. MATLAB calls the figure delete function before deleting the figure, which enables the function to perform any clean up needed before closing the figure. The figure delete function calls the control's `delete` method.

```
function figDelete(src,evnt)
   hgrid.delete;
end
```

Example — Reading Data from Excel

This example creates a graphical interface to access the data in a Microsoft Excel file. To enable the communication between MATLAB and Excel, this example creates an ActiveX object in an Automation server running Excel.

MATLAB then accesses the data in the Excel file through the interfaces provided by the Excel Automation server.

Techniques Demonstrated

This example shows how to use the following techniques:

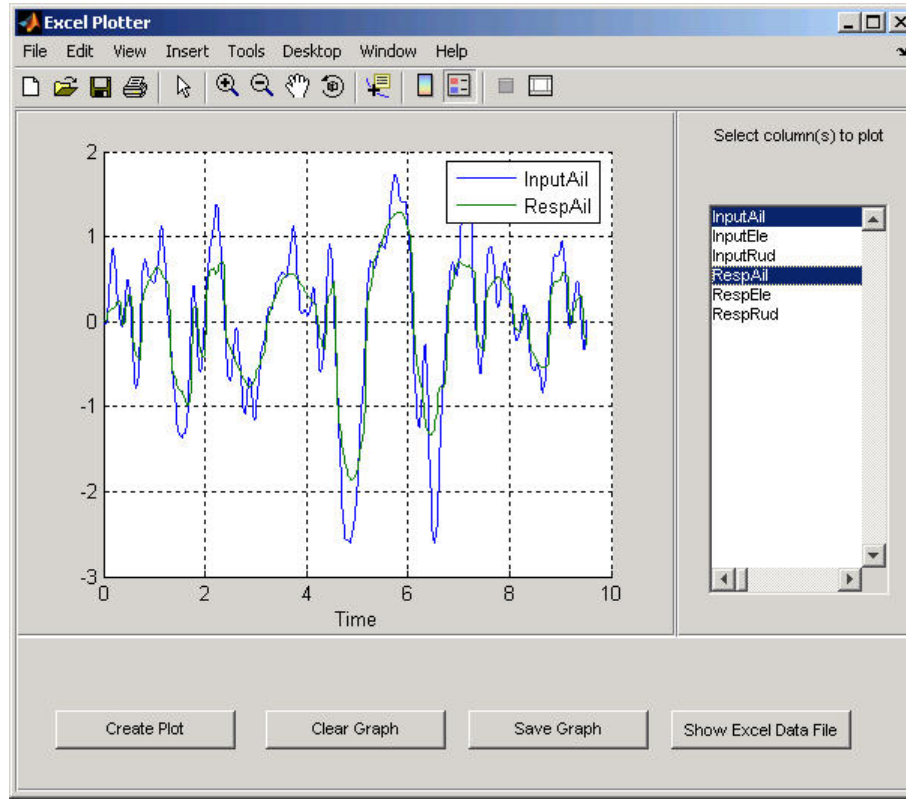
- Use of an Automation server to access another application from MATLAB
- Ways to manipulate Excel data into types used in the GUI and plotting.
- Implementing a GUI that enables plotting of selected columns of the Excel spreadsheet.
- Inserting a MATLAB figure into an Excel file.

Using the GUI

To use the GUI, select any items in the list box and click the **Create Plot** button. The sample data provided with this example contain three input and three associated response data sets, all of which are plotted vs. the first column in the Excel file, which is the time data.

You can view the Excel data file by clicking the **Show Excel Data File** button, and you can save an image of the graph in a different Excel file by clicking **Save Graph** button. Note that the **Save Graph** option creates a temporary PNG file in your working directory.

The following picture shows the GUI with an input/response pair selected in the list box and plotted in the axes.



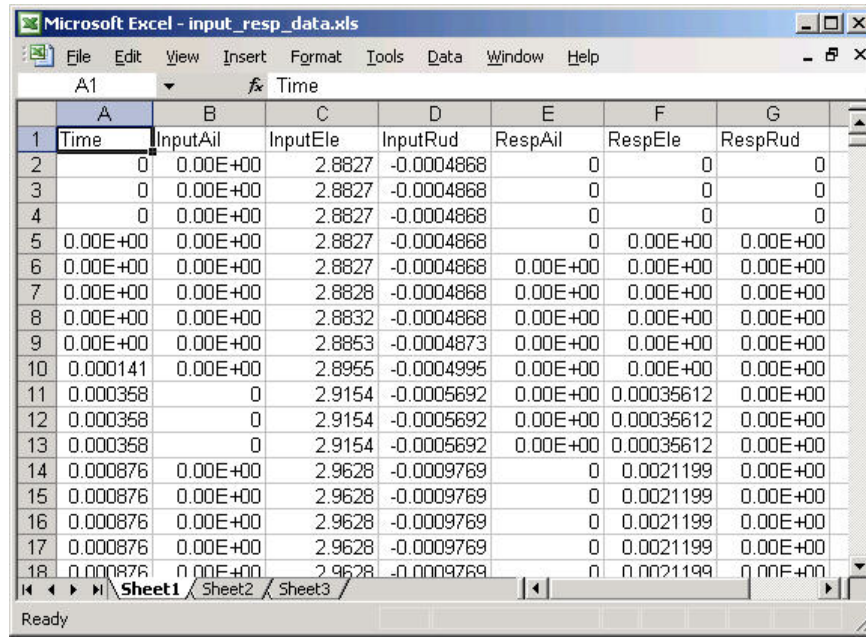
Complete Code Listing

You can open the M-file used to implement this example in the MATLAB editor or run this example:

- Open M-file in editor.
- Run this example.

Excel Spreadsheet Format

This example assumes a particular organization of the Excel spreadsheet, as shown in the following picture.



	A	B	C	D	E	F	G
1	Time	InputAil	InputEle	InputRud	RespAil	RespEle	RespRud
2	0	0.00E+00	2.8827	-0.0004868	0	0	0
3	0	0.00E+00	2.8827	-0.0004868	0	0	0
4	0	0.00E+00	2.8827	-0.0004868	0	0	0
5	0.00E+00	0.00E+00	2.8827	-0.0004868	0	0.00E+00	0.00E+00
6	0.00E+00	0.00E+00	2.8827	-0.0004868	0.00E+00	0.00E+00	0.00E+00
7	0.00E+00	0.00E+00	2.8828	-0.0004868	0.00E+00	0.00E+00	0.00E+00
8	0.00E+00	0.00E+00	2.8832	-0.0004868	0.00E+00	0.00E+00	0.00E+00
9	0.00E+00	0.00E+00	2.8853	-0.0004873	0.00E+00	0.00E+00	0.00E+00
10	0.000141	0.00E+00	2.8955	-0.0004995	0.00E+00	0.00E+00	0.00E+00
11	0.000358	0	2.9154	-0.0005692	0.00E+00	0.00035612	0.00E+00
12	0.000358	0	2.9154	-0.0005692	0.00E+00	0.00035612	0.00E+00
13	0.000358	0	2.9154	-0.0005692	0.00E+00	0.00035612	0.00E+00
14	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00
15	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00
16	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00
17	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00
18	0.000876	0.00E+00	2.9628	-0.0009769	0	0.0021199	0.00E+00

The format of the Excel file is as follows:

- The first element in each column is a text string that identifies the data contain in the column. These strings are extracted and used to populate the list box.
- The first column (Time) is used for the *x*-axis of all plots of the remaining data.
- All rows in each column are read into MATLAB.

Excel Automation Server

The first step in accessing Excel from MATLAB is to run the Excel application in an Automation server process using the `actxserver` function and the Excel program ID, `excel.application`.

```
exl = actxserver('excel.application');
```

The ActiveX object that is returned provides access to a number of interfaces supported by Excel. Use the workbook interface to open the Excel file containing the data.

```
exlWkbk = exl.Workbooks;
exlFile = exlWkbk.Open([docroot '/techdoc/matlab_external/examples/input_resp_data.xls']);
```

Use the workbook's sheet interface to access the data from a range object, which stores a reference to a range of data from the specified sheet. This example accesses all the data in column `A`, first cell to column `G`, last cell:

```
exlSheet1 = exlFile.Sheets.Item('Sheet1');
robj = exlSheet1.Columns.End(4);       % Find the end of the column
numrows = robj.row;                    % And determine what row it is
dat_range = ['A1:G' num2str(numrows)]; % Read to the last row
rngObj = exlSheet1.Range(dat_range);
```

At this point, the entire data set from the Excel file's `sheet1` is accessed via the range object interface. This object returns the data in a MATLAB cell array, which can contain both numeric and character data:

```
exlData = rngObj.Value;
```

Manipulating the Data in MATLAB

Now that the data is in a cell array, you can use MATLAB functions to extract and reshape parts of the data into the forms needed to use in the GUI and pass to the plot function.

The following code performs two operations:

- Extracts numeric data from the cell array (indexing with curly braces), concatenates the individual doubles returned by the indexing operation (square brackets), and reshapes it into an array that arranges the data in columns.
- Extracts the string in the first cell in each column of Excel sheet and stores them in a cell array, which is used to generate the items in the list box.

```
for ii = 1:size(exlData,2)
   matData(:,ii) = reshape([exlData{2:end,ii}],size(exlData(2:end,ii)));
   lBoxList{ii} = [exlData{1,ii}];
end
```

The Plotter GUI

This example uses a GUI that enables you to select from a list of input and response data from a list box. All data is plotted as a function of time (which is, therefore, not a choice in the list box) and you can continue to add more data to the graph. Each data plot added to the graph causes the legend to expand.

Additional implementation details include:

- A legend that updates as you add data to a graph
- A clear button that enables you to clear all graphs from the axes
- A save button that saves the graph as a PNG file and adds it to another Excel file
- A toggle button that shows or hides the Excel file being accessed
- The figure delete function (`DeleteFcn` property), which MATLAB calls when the figure is closed, is used to terminate the Automation server process.

Selecting and Plotting Data. When you click the **Create Plot** button, its callback function queries the list box to determine what items are selected and plots each data vs. time. The legend is updated to display any new data while maintaining the legend for the existing data.

```
function plotButtonCallback(src,evnt)
iSelected = get(listBox,'Value');
grid(a,'on');hold all
for p = 1:length(iSelected)
   switch iSelected(p)
      case 1
         plot(a,tme,matData(:,2))
      case 2
         plot(a,tme,matData(:,3))
      case 3
```

```
            plot(a,tme,matData(:,4))
         case 4
            plot(a,tme,matData(:,5))
         case 5
            plot(a,tme,matData(:,6))
         case 6
            plot(a,tme,matData(:,7))
         otherwise
            disp('Select data to plot')
      end
   end
   [b,c,g,lbs] = legend([lbs lBoxList(iSelected+1)]);
   end % plotButtonCallback
```

Clearing the Axes. The plotter is designed to continually add graphs as the user selects data from the list box. The **Clear Graph** button clears and resets the axes and clears the variable used to store the labels of the plot data (used by legend).

```
%% Callback for clear button
function clearButtonCallback(src,evt)
   cla(a,'reset')
   lbs = '';
end % clearButtonCallback
```

Display or Hide Excel File. The MATLAB program has access to the properties of the Excel application running in the Automation server. By setting the `Visible` property to `1` or `0`, this callback controls the visibility of the Excel file.

```
%% Display or hide Excel file
function dispButtonCallback(src,evt)
   exl.visible = get(src,'Value');
end % dispButtonCallback
```

Close Figure and Terminate Excel Automation Process. Since the Excel Automation server runs in a separate process from MATLAB, you must terminate this process explicitly. There is no reason to keep this process running after the GUI has been closed, so this example uses the figure's delete function to terminate the Excel process with the `Quit` method. Also, terminate the second Excel process used for saving the graph. See "Inserting MATLAB Graphs Into Excel" on page 8-31 for information on this second process.

```
%% Terminate Excel processes
function deleteFig(src,evt)
   exlWkbk.Close
   exlWkbk2.Close
   exl.Quit
   exl2.Quit
end % deleteFig
```

Inserting MATLAB Graphs Into Excel

You can save the graph created with this GUI in an Excel file. (This example uses a separate Excel Automation server process for this purpose.) The callback for the **Save Graph** push button creates the image and adds it to an Excel file:

- Both the axes and legend are copied to an invisible figure configured to print the graph as you see it on the screen (figure `PaperPositionMode` property is set to `auto`).
- The `print` command creates the PNG image.
- Use the `Shapes` interface to insert the image in the Excel workbook.

The server and interfaces are instanced during GUI initialization phase:

```
exl2 = actxserver('excel.application');
exlWkbk2 = exl2.Workbooks;
wb = invoke(exlWkbk2,'Add');
graphSheet = invoke(wb.Sheets,'Add');
Shapes = graphSheet.Shapes;
```

The following code implements the **Save Graph** button callback:

```
function saveButtonCallback(src,evt)
   tempfig = figure('Visible','off','PaperPositionMode','auto');
   tempfigfile = [tempname '.png'];
   ah = findobj(f,'type','axes');
   copyobj(ah,tempfig) % Copy both graph axes and legend axes
   print(tempfig,'-dpng',tempfigfile);
   Shapes.AddPicture(tempfigfile,0,1,50,18,300,235);
   exl2.visible = 1;
end
```

Supported Client/Server Configurations

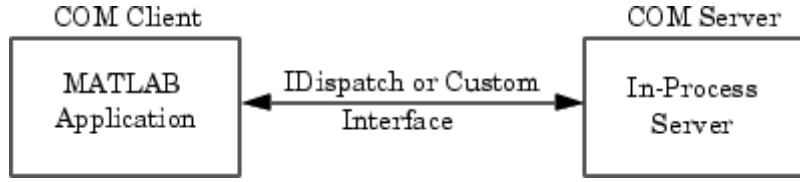
In this section...

Introduction

You can configure MATLAB to either control or be controlled by other COM components. When MATLAB controls another component, MATLAB is the client, and the other component is the server. When another component controls MATLAB, MATLAB is the server.

MATLAB Client and In-Process Server

The following diagram shows how the MATLAB client interacts with an "In-Process Server" on page 8-6.



The server exposes its properties and methods through the IDispatch (Automation) interface or a Custom interface, depending on which interfaces the component implements. For information on accessing interfaces, see "Getting Interfaces to the Object" on page 8-50 .

ActiveX Controls

An ActiveX control is an object with some type of graphical user interface (GUI). When MATLAB constructs an ActiveX control, it places the control's GUI in a MATLAB figure window. Click the various options available in the user interface (e.g., making a particular menu selection) to trigger *events* that get communicated from the control in the server to the client MATLAB application. The client decides what to do about each event and responds accordingly.

MATLAB comes with a sample ActiveX control called `mwsamp`. This control draws a circle on the screen and displays some text. You can use this control to try out MATLAB COM features. For more information, see "MATLAB Sample Control" on page 8-105.

DLL Servers

Any COM component that has been implemented as a dynamic link library (DLL) is also instantiated in an in-process server. That is, it is created in the same process as the MATLAB client application. When MATLAB uses a DLL server, it runs in a separate window rather than a MATLAB figure window.

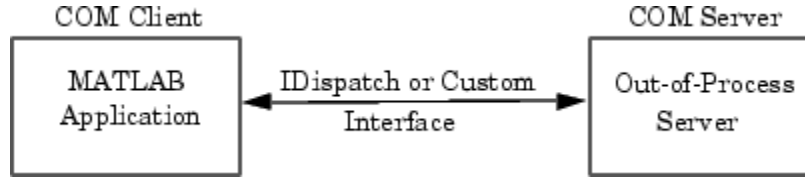
MATLAB responds to events generated by a DLL server in the same way as events from an ActiveX control.

For More Information

To learn more about working with MATLAB as a client, see "MATLAB COM Client Support" on page 8-38 and "Additional COM Client Information" on page 8-110.

MATLAB Client and Out-of-Process Server

In this configuration, a MATLAB client application interacts with a component that has been implemented as a "Local Out-of-Process Server" on page 8-6. Examples of out-of-process servers are Microsoft Excel and Microsoft Word.



As with in-process servers, this server exposes its properties and methods through the IDispatch (Automation) interface or a Custom interface, depending on which interfaces the component implements. For information on accessing interfaces, see “Getting Interfaces to the Object” on page 8-50.

Since the client and server run in separate processes, you have the option of creating the server on any system on the same network as the client.

If the component provides a user interface, its window is separate from the client application.

MATLAB responds to events generated by an out-of-process server in the same way as events from an ActiveX control.

For More Information

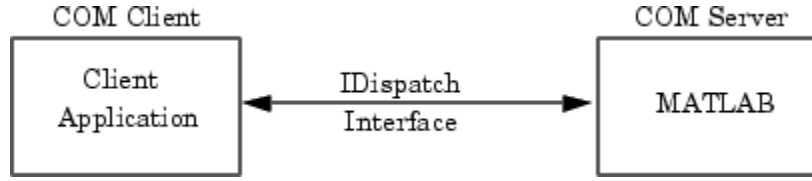
To learn more about working with MATLAB as a client, see “MATLAB COM Client Support” on page 8-38 and “Additional COM Client Information” on page 8-110.

COM Implementations Supported by MATLAB

MATLAB only supports COM implementations that are compatible with the Microsoft Active Template Library (ATL) API. In general, your COM object should be able to be contained in an ATL host window in order to work with MATLAB.

Client Application and MATLAB Automation Server

MATLAB operates as the Automation server in this configuration. It can be created and controlled by any Windows program that can be an *Automation controller*. Some examples of Automation controllers are Microsoft Excel, Microsoft Access, Microsoft Project, and many Visual Basic and Visual C++ programs.



MATLAB Automation server capabilities include the ability to execute commands in the MATLAB workspace, and to get and put matrices directly from and into the workspace. You can start a MATLAB server to run in either a shared or dedicated mode. You also have the option of running it on a local or remote system.

To create the MATLAB server from an external application program, use the appropriate function from that language to instantiate the server. (For example, use the `CreateObject` function in Visual Basic.) For the programmatic identifier, specify `matlab.application`. To run MATLAB as a dedicated server, use the `matlab.application.single` programmatic identifier. See “Shared and Dedicated Servers” on page 8-113 for more information.

The function that creates the MATLAB server also returns a handle to the properties and methods available in the server through the IDispatch interface. See “MATLAB Automation Server Functions and Properties” on page 8-116 for descriptions of these methods.

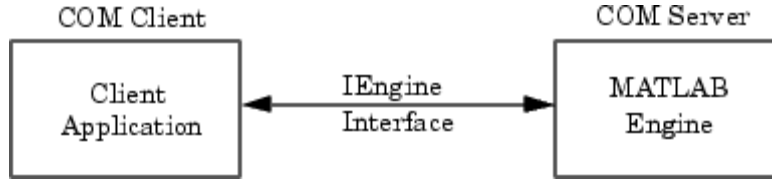
Note Because VBScript client programs require an Automation interface to communicate with servers, this is the only configuration that supports a VBScript client.

For More Information

To learn more about working with MATLAB Automation servers, see “MATLAB COM Automation Server Support” on page 8-112 and “Additional Automation Server Information” on page 8-122.

Client Application and MATLAB Engine Server

MATLAB provides a faster custom interface called IEngine for client applications written in C, C++, or Fortran. MATLAB uses IEngine to communicate between the client application and the MATLAB engine running as a COM server.



MATLAB provides a library of functions that let you to start and end the server process, and to send commands to be processed by MATLAB. See "MATLAB Engine" in the C and Fortran API Reference for more information.

For More Information

To learn more about the MATLAB engine and the functions provided in its C and Fortran libraries, see Chapter 6, "Calling MATLAB from C and Fortran Programs".

MATLAB COM Client Support

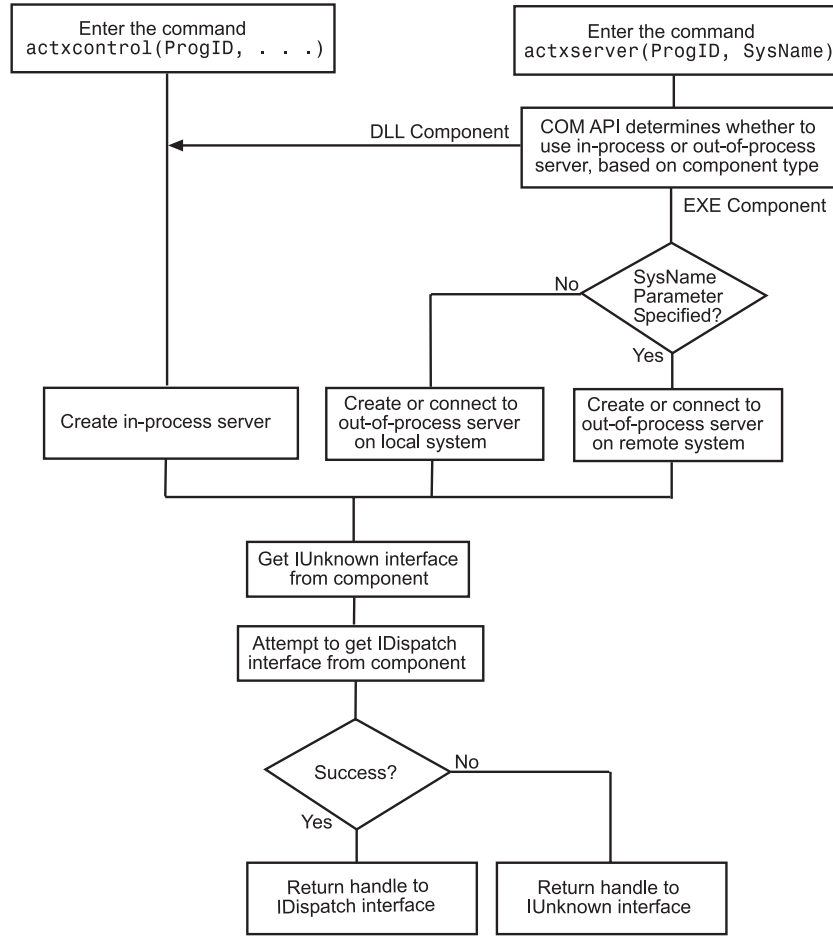
In this section...

Creating the Server Process — An Overview

MATLAB provides two functions to create a COM object:

- `actxcontrol` — Creates an ActiveX control in a MATLAB figure window.
- `actxserver` — Creates an in-process server for a dynamic link library (DLL) component or an out-of-process server for an executable (EXE) component.

The diagram below shows the basic steps in creating the server process. For more information on how MATLAB establishes interfaces to the resultant COM object, see “Getting Interfaces to the Object” on page 8-50.



Creating an ActiveX Control

You can create an ActiveX control from the MATLAB client using either a graphical user interface or the `actxcontrol` function directly from the command line. Either of these methods creates an instance of the control in

the MATLAB client process and returns a handle to the primary interface to the COM object. Through this interface, you can access the object's public properties or methods . You can also establish additional interfaces to the object, including interfaces that use IDispatch, and any custom interfaces that may exist.

This section describes how to create the control and how to position its physical representation in the MATLAB figure window:

Finding Out What Controls Are Installed

The `actxcontrollist` function shows you what COM controls are currently installed on your system. Type

```
list = actxcontrollist
```

and MATLAB returns a cell array listing each control, including its name, programmatic identifier (ProgID), and filename.

This example shows information that might be returned for several controls (your results might be different):

```
list = actxcontrollist;
s=sprintf(' Name = %s\n ProgID = %s\n File = %s\n', list{114:115,:})
```

MATLAB displays:

```
s =
 Name = OleInstall Class
 ProgID = Outlook Express Mime Editor
 File = OlePrn.OleInstall.1
 Name = OutlookExpress.MimeEdit.1
```

```
ProgID = C:\WINNT\System32\oleprn.dll
File = C:\WINNT\System32\inetcomm.dll
```

Finding a Particular Control

If you know the name of a control, you can find it in the list and display its ProgID and the path of the directory containing it using a few MATLAB commands. For example the `Mwsamp2` control is used in some of the examples in this manual. You can find it with the following code:

```
list = actxcontrollist;
for ii = 1:length(list)
   if ~isempty(findstr('Mwsamp2',[list{ii,:}]))
      s = sprintf(' Name = %s\n ProgID = %s\n File = %s\n', ...
             list{ii,:})
   end
end
```

The formatted output contained in `s` is displayed:

```
s =
   Name = Mwsamp2 Control
   ProgID = MWSAMP.MwsampCtrl.2
   File =
   D:\Apps\MATLAB\R2006a\toolbox\matlab\winfun\win32\mwsamp2.ocx
```

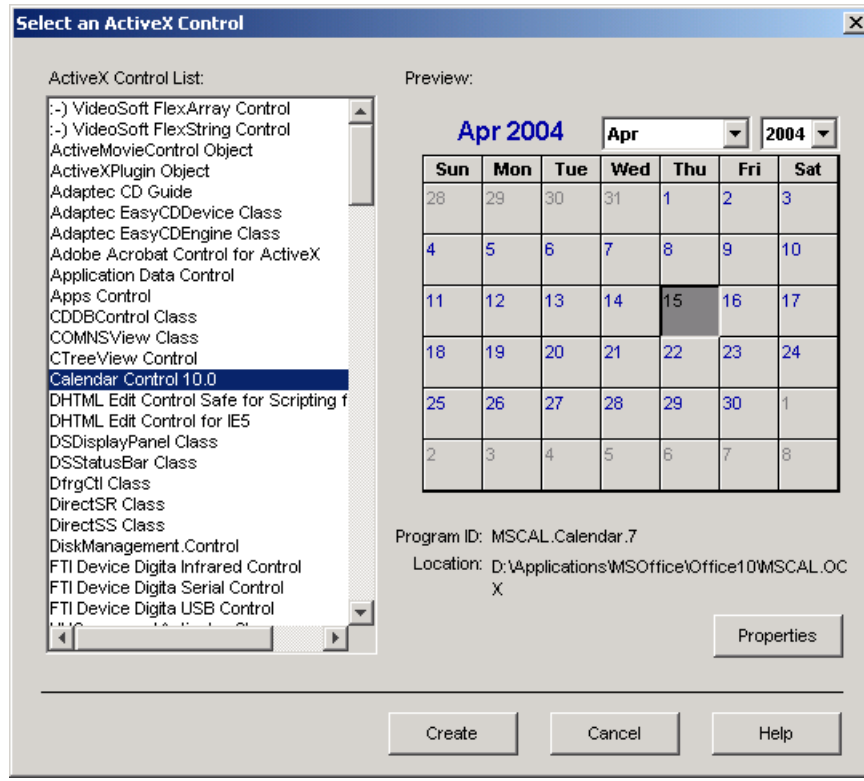
The location of this file might be different on your system.

Creating Control Objects Using a Graphical Interface

The simplest way to create a control object is to use the `actxcontrolselect` function. This function displays a graphical user interface (GUI) listing all controls installed on your system. When you select an item from the list and click the **Create** button, MATLAB creates the control and returns a handle to it. Type

```
h = actxcontrolselect
```

MATLAB displays:



The interface has a selection panel on the left and a preview panel on the right. Click one of the control names in the selection panel to see a preview of the control. (For controls that do not have a preview, the preview panel

is blank.) If MATLAB cannot create the control, an error message appears in the preview panel.

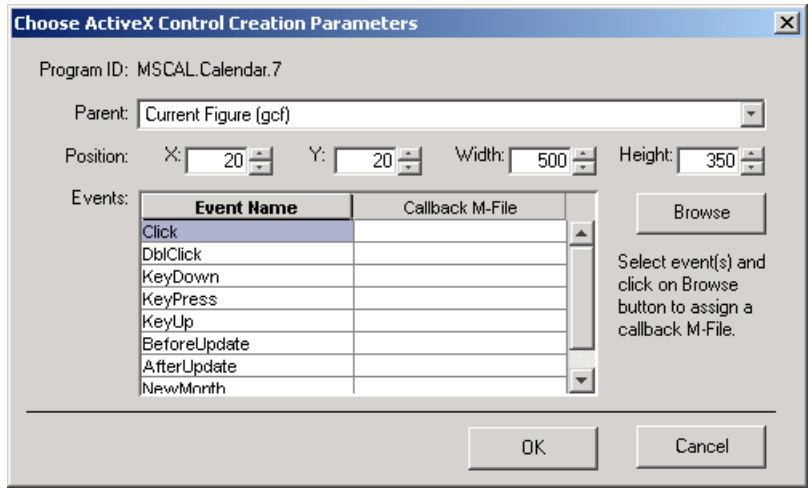
Setting Properties with actxcontrolselect. Click the **Properties** button on the `actxcontrolselect` window to change property values when creating the control. You can select which figure window to put the control in (**Parent** field), where to position it in the window (**X** and **Y** fields), and what size to make the control (**Width** and **Height**).

You can register events you want the control to respond to in this window. (See "Control and Server Events" on page 8-75 for an explanation of event registration.) Register an event and the callback routine to handle that event by entering the name of the routine to the right of the event under **Callback M-File**.

You can select callback routines by browsing for their M-files. Click a name in the **Event Names** column and then click the **Browse** button. To assign a callback routine to more than one event, first press the **Ctrl** key and click individual event names, or drag the mouse over consecutive event names, then click **Browse** to select the callback routine.

MATLAB only responds to registered events, so if you do not specify a **Callback M-File**, the event is ignored.

For example, at the `actxcontrolselect` window, select **Calendar Control 10.0** and click **Properties** to see the window shown below. Enter a **Width** of `500` and a **Height** of `350` to change the default size for the control. Click **OK** in this window, and click **Create** in the `actxcontrolselect` window to create the Calendar control.



You can also set control parameters using the `actxcontrol` function. One parameter you can set with `actxcontrol` but not with `actxcontrolselect` is the name of an initialization file. When you specify this filename, MATLAB sets the initial state of the control to that of a previously saved control.

Information Returned by actxcontrolselect. `actxcontrolselect` creates an object that is an instance of the MATLAB COM class. The function returns up to two arguments: a handle for the object `h`, and a 1-by-3 cell array `info` containing information about the control.

```
[h, info] = actxcontrolselect;
```

The cell array `info` shows the name, ProgID, and filename for the control.

If you select Calendar Control 9.0 and then click **Create**, MATLAB displays:

```
h =
   COM.mscal.calendar.7
```

```
info =
    [1x20 char]    'MSCAL.Calendar.7'    [1x41 char]
```

Expand the `info` cell array to show the control name, ProgID, and filename.

```
info{:}
ans =
   Calendar Control 9.0
ans =
   MSCAL.Calendar.7
ans =
   D:\Applications\MSOffice\Office\MSCAL.OCX
```

Creating Control Objects from the Command Line

If you already know which control you want and you know its ProgID, you can bypass the GUI by using the `actxcontrol` function to create it.

The only required input when calling the function is the ProgID. However, as with `actxcontrolselect`, you can supply additional inputs that enable you to select which figure window to put the control in, where to position it in the window, and what size to make it. You can also register any events you want the control to respond to, or set the initial state of the control by reading that state from a file. See the reference page on `actxcontrol` for a full explanation of its input arguments.

`actxcontrol` returns a handle to the primary interface to the object. Use this handle to reference the object in other COM function calls. You can also use the handle to obtain additional interfaces to the object. For more information on using interfaces, see “Getting Interfaces to the Object” on page 8-50.

This example creates a control to run a Microsoft Calendar application. Position the control in figure window `fig3`, at a `[0 0] x-y` offset from the bottom left of the window, and make it `300` by `400` pixels in size:

```
fig3 = figure('position', [50 50 600 500]);
h = actxcontrol('mscal.calendar', [0 0 300 400], fig3)
```

MATLAB displays:

```
h =
    COM.mscal.calendar
```

Repositioning the Control in a Figure Window

Once a control has been created, you can change its shape and position in the window with the `move` function.

Observe what happens to the object created in the last section when you specify new origin coordinates (`70`, `120`) and new width and height dimensions of `400` and `350`:

```
h.move([70 120 400 350]);
```

Using Microsoft Forms 2.0 Controls

You may encounter problems when creating or using Microsoft Forms 2.0 controls in MATLAB. Forms 2.0 controls are designed for use with applications enabled by Visual Basic for Applications (VBA). Microsoft Office is one such application.

To work around these problems, use the replacement controls listed below, or consult article 236458 in the Microsoft Knowledge Base for further information:

```
http://support.microsoft.com/default.aspx?kbid=236458
```

Affected Controls. You may see this behavior with any of the following Forms 2.0 controls:

- `Forms.TextBox.1`
- `Forms.CheckBox.1`

- `Forms.CommandButton.1`
- `Forms.Image.1`
- `Forms.OptionButton.1`
- `Forms.ScrollBar.1`
- `Forms.SpinButton.1`
- `Forms.TabStrip.1`
- `Forms.ToggleButton.1`

Replacement Controls. The following replacements are recommended by Microsoft:

Old	New
`Forms.TextBox.1`	`RICHTEXT.RichtextCtrl.1`
`Forms.CheckBox.1`	`vidtc3.Checkbox`
`Forms.CommandButton.1`	`MSComCtl2.FlatScrollBar.2`
`Forms.TabStrip.1`	`COMCTL.TabStrip.1`

Deploying ActiveX Controls Requiring Run-Time Licenses

When you deploy an ActiveX control that requires a run-time license, you must include a license key, which the control reads at run-time. If the key matches the control's own version of the license key, an instance of the control is created. Use the following procedure to deploy a run-time-licensed control with MATLAB.

Create an M-File to Build the Control

First, create an M-file to build the control. This M-file must contain two elements:

- The pragma `%#function actxlicense`. This pragma causes MATLAB Compiler to embed a function named `actxlicense` into the stand-alone executable file you build.

- A call to `actxcontrol` to create the control.

Place this M-file in a directory that is not part of the MATLAB code tree.

Here is an example M-file.

```
function buildcontrol
%#function actxlicense
h=actxcontrol('MFCCONTROL2.MFCControl2Ctrl.1',[10 10 200 200]);
```

Build the Control and the License M-File

Change to the directory where you placed the M-file you created to build the control. Call the function you defined in the M-file. When it executes this function, MATLAB determines whether the control requires a run-time license. If it does, MATLAB creates another M-file, named `actxlicense.m`, in the current working directory. The `actxlicense` function defined in this file provides the license key to MATLAB at run-time.

Build the Executable

Next, call MATLAB Compiler to create the stand-alone executable from the file you created to build the control. The executable contains both the function that builds the control and the `actxlicense` function.

```
mcc -m buildcontrol
```

Deploy the Files

Finally, distribute `buildcontrol.exe`, `buildcontrol.ctf`, and the control (`.ocx` or `.dll`).

Instantiating a DLL Component

To create a server for a component implemented as a dynamic link library (DLL), use the `actxserver` function. MATLAB creates an instance of the component in the same process that contains the client application.

The syntax for `actxserver`, when used with a DLL component, is

```
actxserver(ProgID)
```

where `ProgID` is the programmatic identifier for the component.

`actxserver` returns a handle to the primary interface to the object. Use this handle to reference the object in other COM function calls. You can also use the handle to obtain additional interfaces to the object. For more information on using interfaces, see “Getting Interfaces to the Object” on page 8-50.

Unlike ActiveX controls, any user interface displayed by the server appears in a separate window.

Instantiating an EXE Component

You can use the `actxserver` function to create a server for a component implemented as an executable (EXE). In this case, MATLAB instantiates the component in an out-of-process server.

The syntax for `actxserver` is

```
actxserver(ProgID, sysname)
```

where `ProgID` is the programmatic identifier for the component, and `sysname` is an optional argument used in configuring a distributed COM (DCOM) system.

`actxserver` returns a handle to the primary interface to the COM object. Use this handle to reference the object in other COM function calls. You can also use the handle to obtain additional interfaces to the object. For more information on using interfaces, see “Getting Interfaces to the Object” on page 8-50.

Any user interface displayed by the server appears in a separate window.

This example creates a COM server application running Excel. The handle is assigned to `h`:

```
h = actxserver('excel.application')
```

MATLAB displays:

```
h =
    COM.excel.application
```

MATLAB can programmatically connect to an instance of a COM Automation server application that is already running on your computer. Use the `actxGetRunningServer` function to get a reference to such an application.

The syntax for `actxGetRunningServer` is

```
actxGetRunningServer(ProgID)
```

where `ProgID` is the programmatic identifier for the component.

This example gets a reference to Excel, which must already be running on your system. The returned handle is assigned to `h`:

```
h = actxGetRunningServer('excel.application')
```

MATLAB displays:

```
h =
    COM.excel.application
```

Getting Interfaces to the Object

The COM component you are working with can provide different types of interfaces for accessing the object's public properties and methods:

- The IUnknown and IDispatch interfaces
- One or more custom interfaces

IUnknown and IDispatch

When you invoke the `actxserver` or `actxcontrol` function, MATLAB creates the server and returns a handle to the server interface as a means of accessing its properties and methods. MATLAB uses the following process to determine which handle to return:

1. MATLAB first gets a handle to the IUnknown interface from the component. All COM components are required to implement this interface.
2. MATLAB then attempts to get the IDispatch interface. If IDispatch is implemented, MATLAB returns a handle to this interface. If IDispatch is not implemented, MATLAB returns the handle to IUnknown.

Additional Interfaces. Components often provide additional interfaces, based on IDispatch, that are implemented as properties. Like any other property, you obtain these interfaces using the MATLAB `get` function.

For example, a Microsoft Excel component contains numerous interfaces. To list these interfaces, along with other Excel properties, use the MATLAB `get` function without any arguments. For example, type:

```
h = actxserver('excel.application');
h.get
```

MATLAB displays information similar to:

```
    Application: [1x1 Interface.Microsoft_Excel_9.0_
Object_Library._Application]
        Creator: 'xlCreatorCode'
         Parent: [1x1 Interface.Microsoft_Excel_9.0_
Object_Library._Application]
     ActiveCell: []
    ActiveChart: [1x50 char]
                .
                .
```

In the following example, `h` is a handle to a specific interface and `Workbooks` is the name of the interface. Type

```
w = h.Workbooks
```

MATLAB displays:

```
w =
   Interface.Microsoft_Excel_9.0_Object_Library.Workbooks
```

The information displayed depends on the version of Excel you have on your system.

Custom Interfaces

The following two client/server configurations support a component's custom interface:

- "Introduction" on page 8-33

• "MATLAB Client and Out-of-Process Server" on page 8-34

Once you have created the server, you can query the server component to see if any custom interfaces are implemented. Use the `interfaces` function to return a list of all available custom interfaces. This list is returned in a cell array of strings.

For example, for a component with the ProgID `mytestenv.calculator`, type

```
h = actxserver('mytestenv.calculator')
```

MATLAB displays:

```
h =
   COM.mytestenv.calculator
```

Type

```
customlist = h.interfaces
```

MATLAB displays:

```
customlist =
   ICalc1
   ICalc2
   ICalc3
```

To get a handle to a particular custom interface, use the `invoke` function, specifying the handle returned by `actxcontrol` or `actxserver`, and also the name of the custom interface:

```
c1 = h.invoke('ICalc1')
c1 =
   Interface.Calc_1.0_Type_Library.ICalc_Interface
```

Use this handle with the COM client functions to access the properties and methods of the object through the selected custom interface.

For example, to list the properties available through the `ICalc1` interface, use

```
c1.get
     background: 'Blue'
         height: 10
          width: 0
```

To list the methods, use

```
c1.invoke
   Add = double Add(handle, double, double)
   Divide = double Divide(handle, double, double)
   Multiply = double Multiply(handle, double, double)
   Subtract = double Subtract(handle, double, double)
```

Add and multiply numbers using the `Add` and `Multiply` methods of the custom object `c1`:

```
sum = c1.Add(4, 7)
sum =
    11

prod = c1.Multiply(4, 7)
prod =
    28
```

Invoking Commands on a COM Object

This section covers the following topics:

Dot Syntax

When invoking either MATLAB COM functions or methods belonging to COM objects, the simplest syntax to use is *dot syntax*. Specify the object name, the

dot (.), and the name of the function or method you are calling. Enclose any input arguments in parentheses after the function name. Specify output arguments to the left of the equal sign:

```
outputvalue = object.function(arg1, arg2, ...)
```

An Example of Calling Syntax

To work with the example that follows, first create an ActiveX control called mwsamp. (The mwsamp control is built into MATLAB to enable you to work with the examples shown in the COM documentation. The control displays a circle and text label that you can manipulate from MATLAB.)

Call actxcontrol to create the mwsamp control. This function returns a handle h that you need to work further with the control.

```
h = actxcontrol('mwsamp.mwsampctrl.2', [200 120 200 200]);
```

Once you have a handle to an object, you can invoke MATLAB functions on the object by referencing it through the handle.

For example, you can create a custom property, called Position, using the addproperty function. See “Custom Properties” on page 8-74.

```
h.addproperty('Position');
```

An alternative syntax for the same operation is

```
addproperty(h, 'Position');
```

Specifying Property, Method, and Event Names

You can specify the names of properties and methods using the simple notation

```
handle.propertyname
handle.methodname
```

For example, the mwsamp object has a property called Radius that represents the radius of the circle it draws, and a method called Redraw that redraws the circle. You can get the circle's radius by typing

```
h.Radius
```

You can redraw the circle with

```
h.Redraw
```

More information is provided on this in the sections "Implicit Syntax for Calling get, set, and invoke" on page 8-55 and "Exceptions to Using Implicit Syntax" on page 8-56. Here are a few specific rules regarding how to express property, method, and event names.

Property Names. You can abbreviate the names of properties, as long as you include enough characters in the name to distinguish it from another property. Property names are also case insensitive.

These two statements produce the same result:

```
x = h.Radius
x = h.r
```

Method Names. Method names cannot be abbreviated and are case sensitive.

Event Names. Event names are always specified as quoted strings in arguments to a function. Event names cannot be abbreviated and they are not case sensitive.

These statements produce the same result:

```
h.registerevent({'MouseDown' 'mymoused'});
h.registerevent({'MOUSEdown' 'myMOUSEd'});
```

Implicit Syntax for Calling get, set, and invoke

When calling `get`, `set`, or `invoke` on a COM object, MATLAB provides a simpler syntax that doesn't require an explicit function call. You can use this shortened syntax in all but a few cases (see "Exceptions to Using Implicit Syntax" on page 8-56).

Continue with the `mwsamp` control created in the last section and represented by handle `h`. To get the value of `Radius` property and assign it to variable `x`, use the syntax shown here. MATLAB still makes the call to `get`, but this shortened syntax is somewhat easier to enter:

```
x = h.Radius
x =
    20
```

The same shortened syntax applies when calling the `set` and `invoke` functions. Compare these two ways of setting a new radius value for the circle and invoking the `Redraw` method of `mwsamp` to display the circle in its enlarged size. The commands on the left call `set` and `invoke` explicitly. The commands on the right make implicit calls:

```
h.set('Radius', 40);              h.Radius = 40;
h.invoke('Redraw');               h.Redraw;
```

Exceptions to Using Implicit Syntax

There are some conditions under which you must explicitly call `get`, `set`, and `invoke`:

- When the property or method is not public
- When accessing a property that takes arguments
- When operating on a vector of objects

Nonpublic properties and methods. If the property or method you want to access is not provided as a public property or method of the object class, or if it is not in the type library for the control or server, you must call `get`, `set`, or `invoke` explicitly. For example, the `Visible` property of an Internet Explorer application is not public and must be accessed using `get` and `set`:

```
h = actxserver('internetexplorer.application');

% This syntax is invalid because 'Visible' is not public.
v = h.Visible
??? No appropriate method or public field Visible for class
   COM.internetexplorer.application.

% You must call the get function explicitly.
v = h.get('Visible')
v =
     1
```

```
% The same holds true when setting nonpublic properties.
h.set('Visible', 1);
```

Public properties and methods are those that are listed in response to the following commands on COM object `h`:

```
publicproperties = h.get
publicmethods = h.invoke
```

Accessing Properties That Take Arguments. Some COM objects have properties that behave somewhat like methods in that they accept input arguments. This is explained fully in the section "Properties That Take Arguments" on page 8-68. In order to get or set the value of such a property, you must make an explicit call to the `get` or `set` function, as shown here. In this example, `A1` and `B2` are arguments that specify which `Range` interface to return on the `get` operation:

```
eActivesheetRange = e.Activesheet.get('Range', 'A1', 'B2')
eActivesheetRange =
   Interface.Microsoft_Excel_5.0_Object_Library.Range
```

Operating on a Vector of Objects. If you operate on a vector of objects, (see "Get and Set on a Vector of Objects" on page 8-69), you must call `get` and `set` explicitly to access properties.

This example creates a vector of handles to two Microsoft Calendar objects. It then modifies the `Day` property of both objects in one operation by invoking `set` on the vector. Explicit calls to `get` and `set` are required:

```
h1 = actxcontrol('mscal.calendar', [0 200 250 200]);
h2 = actxcontrol('mscal.calendar', [250 200 250 200]);
H = [h1 h2];

H.set('Day', 23)
H.get('Day')
ans =
    [23]
    [23]
```

This applies only to `get` and `set`. You cannot invoke a method on more than one COM object at a time, even if you call `invoke` explicitly.

Identifying Objects and Interfaces

You can get some additional information about a control or server using the following functions.

Function	Description
`class`	Return the class of an object
`isa`	Determine if an object is of a given MATLAB class
`iscom`	Determine if the input is a COM or ActiveX object
`isinterface`	Determine if the input is a COM interface

This example creates a COM object in an Automation server running Excel, giving it the handle `h`, and a `Workbooks` interface to the object, with handle `w`.

```
h = actxserver('excel.application');
w = h.Workbooks;
```

Use the `iscom` function to see if variable `h` is a handle to a COM or ActiveX object:

```
h.iscom
ans =
     1
```

Use the `isa` function to test variable `h` against a known class name:

```
h.isa('COM.excel.application')
ans =
     1
```

Use `isinterface` to see if variable `w` is a handle to a COM interface:

```
w.isinterface
ans =
     1
```

Use the `class` function to find out the class of variable `w`:

```
w.class
ans =
```

```
Interface.Microsoft_Excel_9.0_Object_Library.Workbooks
```

Invoking Methods

This section covers the following topics:

Functions for Working with Methods

Use the following MATLAB functions to find out what methods a COM object has and to invoke any of these methods on the object.

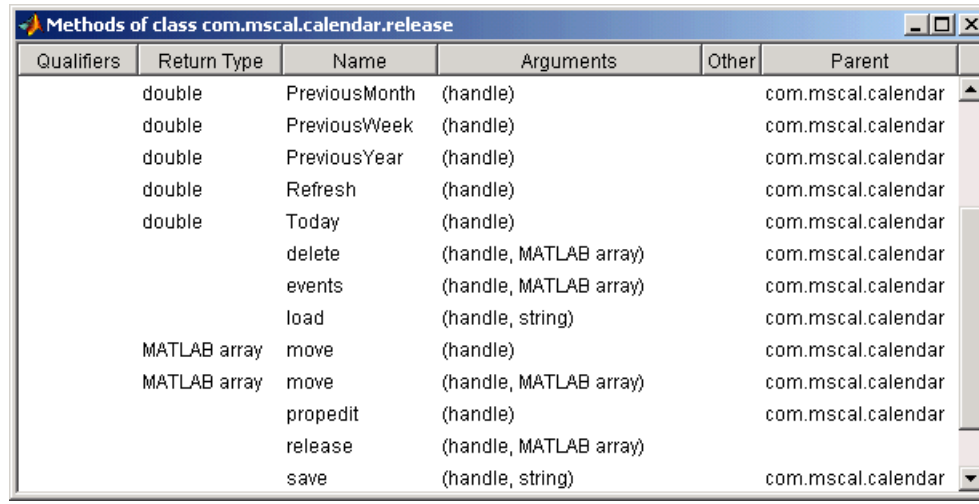
Function	Description
`invoke`	Invoke a method or display a list of methods and types
`ismethod`	Determine if an item is a method of a COM object
`methods`	List all method names for the control or server
`methodsview`	GUI interface to list information on all methods and types

Listing the Methods of a Class or Object

You can see what methods are supported by a control or server object either in a graphical display using the `methodsview` function, or in a returned cell array using the `methods` function.

Using methodsview. The `methodsview` function opens a new window with an easy-to-read display of all methods supported by the specified control or server object along with several related fields of information. Type the following to bring up a window such as the one shown below:

```
cal = actxcontrol('mscal.calendar', [0 0 400 400]);
cal.methodsview
```



Qualifiers	Return Type	Name	Arguments	Other	Parent
	double	PreviousMonth	(handle)		com.mscal.calendar
	double	PreviousWeek	(handle)		com.mscal.calendar
	double	PreviousYear	(handle)		com.mscal.calendar
	double	Refresh	(handle)		com.mscal.calendar
	double	Today	(handle)		com.mscal.calendar
		delete	(handle, MATLAB array)		com.mscal.calendar
		events	(handle, MATLAB array)		com.mscal.calendar
		load	(handle, string)		com.mscal.calendar
	MATLAB array	move	(handle)		com.mscal.calendar
	MATLAB array	move	(handle, MATLAB array)		com.mscal.calendar
		propedit	(handle)		com.mscal.calendar
		release	(handle, MATLAB array)		
		save	(handle, string)		com.mscal.calendar

Methods that return `void` show no **Return Type** in the display.

Using methods. The `methods` function returns in a cell array the names of all methods supported by the specified control or server object. This includes MATLAB COM functions that you can use on the object.

When you include the `-full` switch in the command, MATLAB also specifies the input and output arguments for each method:

```
cal.methods('-full')

Methods for class COM.mscal.calendar:

release(handle, MATLAB array)
delete(handle, MATLAB array)
MATLAB array events(handle, MATLAB array)
```

```
          .
          .
HRESULT Refresh(handle)
HRESULT Today(handle)
HRESULT AboutBox(handle)
```

The `invoke` function, when called with only a handle argument, returns a similar output.

Invoking Methods on an Object

To execute, or *invoke*, any method on an object, use either the MATLAB `invoke` function, or the somewhat simpler method name syntax.

Using invoke. The `invoke` function executes the specified method on an object. You can use either of the following syntaxes with `invoke`:

```
v = invoke(handle, 'methodname', 'arg1', 'arg2', ...);
v = handle.invoke('methodname', 'arg1', 'arg2', ...);
```

See the output of `methodsview` for a method to see what data types to use for input and output arguments.

The following example reads today's date and then advances it by five years by invoking the `NextYear` method in a loop.

To get today's date, type

```
cal = actxcontrol('mscal.calendar', [0 0 400 400]);
cal.Value
ans =
   11/5/2001
```

Call the NextYear method to advance the date, and verify the results:

```
for k = 1:5
   cal.invoke('NextYear');
end

cal.Value
ans =
   11/5/2006
```

Using the Method Name. Instead of using invoke, you can just use the name of the method to call it. The syntax for calling by method name is

```
v = handle.methodname('arg1', 'arg2', ...);
```

or

```
v = methodname(handle, 'arg1', 'arg2', ...);
```

Continuing the example shown in the last section, return to the original data by going back five years.

```
for k = 1:5
   cal.PreviousYear;
end

cal.Value
ans =
   11/5/2001
```

Specifying Enumerated Parameters

Enumeration is a way of representing a somewhat cryptic symbolic value by using a more descriptive name that makes it clear what the value stands for. For example, a program that takes atomic numbers of elements as input is easier to work with if the program accepts element names as input rather than requiring you to recall and pass atomic numbers for each element. You can pass the word 'arsenic' as an enumeration for the value 33.

MATLAB supports enumeration for parameters passed to methods under the condition that the type library in use reports the parameter as ENUM, and only as ENUM.

Note MATLAB does not support enumeration for any parameter that the type library reports as both `ENUM` and `Optional`.

The last line of this example passes an enumerated value (`'xlLocationAsObject'`) to the `Location` method of a Microsoft Excel `Chart` object. You have the choice of passing the enumeration or its numeric equivalent:

```
e = actxserver('Excel.Application');

% Insert a new workbook.
Workbook = e.Workbooks.Add;
e.Visible = 1;
Sheets = e.ActiveWorkBook.Sheets;

% Get a handle to the active sheet.
Activesheet = e.Activesheet;

%Add a Chart
Charts = Workbook.Charts;
Chart = Charts.Add;

% Set chart type to be a line plot.
Chart.ChartType = 'xlXYScatterLines'
C1 = Chart.Location('xlLocationAsObject', 'Sheet1');
```

When you are dealing with only three numeric values, it is not that difficult to remember the meaning of each. But with programs that require a large number of such values, enumeration becomes more important.

Optional Input Arguments

When calling a method that takes optional input arguments, you can skip any optional argument by specifying an empty array (`[]`) in its place. The syntax for calling a method with second argument (`arg2`) not specified is as follows:

```
handle.methodname(arg1, [], arg3);
```

The example below invokes the `Add` method of an Excel object. This method adds new sheets to an Excel workbook. The `Add` method takes up to four optional input arguments:

- `Before` — The sheet before which to add the new sheet
- `After` — The sheet after which to add the new sheet
- `Count` — The total number of sheets to add
- `Type` — The type of sheet to add

The following code creates a workbook with the default number of worksheets, and inserts an additional sheet after Sheet 1. To do this, you invoke `Add` specifying only the second argument, `After`. You can omit the first argument, `Before`, by using `[]` in its place. This is done on the last line:

```
% Open an Excel Server.
e = actxserver('excel.application');

% Insert a new workbook.
e.Workbooks.Add;
e.Visible = 1;

% Get the Active Workbook with three sheets.
eSheets = e.ActiveWorkbook.Sheets;

% Add a new sheet after eSheet1.
eSheet1 = eSheets.Item(1);
eNewSheet = eSheets.Add([], eSheet1);
```

Returning Multiple Output Arguments

If you know that a server function supports multiple outputs, you can return any or all of those outputs to a MATLAB client. Specify the output arguments within brackets on the left side of the equation. This gives the MATLAB client code access to any values returned by the server function.

The syntax shown here shows a server function being called by the MATLAB client. The function's return value is shown as `retval`. The function's output arguments (`out1`, `out2`, `...`) follow this:

```
[retval out1 out2 ...] = handle.functionname(in1, in2, ...);
```

MATLAB makes use of the pass-by-reference capabilities in COM to implement this feature. Note that pass by reference is a COM feature. It is not available in MATLAB at this time.

Argument Callouts in Error Messages

When a MATLAB client sends a command with an invalid argument to a COM server application, the server sends back an error message similar to that shown here, identifying the invalid argument. Be careful when interpreting the argument callout in this type of message:

```
PutFullMatrix(handle, 'a', 'base', 7, [5 8]);
??? Error: Type mismatch, argument 3.
```

In the `PutFullMatrix` command shown above, the fourth argument, `7`, is invalid. (It is scalar and not the expected array data type.) However, the error message identifies the failing argument as `argument 3`.

This is because the COM server receives only the last four of the arguments shown in the MATLAB code. (The `handle` argument merely identifies the server. It does not get passed to the server). So the server sees `'a'` as the first argument, and the invalid argument, `7`, as the third.

As another example, submitting the same command with the `invoke` function makes the invalid argument fifth in the MATLAB client code. Yet the server still identifies it as `argument 3` because neither of the first two arguments is seen by the server:

```
invoke(handle, 'PutFullMatrix', 'a', 'base', 7, [5 8]);
??? Error: Type mismatch, argument 3.
```

Object Properties

This section covers the following topics describing how to set and get the value of a property, and how to create custom properties:

Functions for Working with Object Properties

Use these MATLAB functions to get, set, and modify the properties of a COM object or interface, or to add your own custom properties.

Function	Description
`addproperty`	Add a custom property to a COM object
`deleteproperty`	Remove a custom property from a COM object
`get`	List one or more properties and their values
`inspect`	Display graphical interface to list and modify property values
`isprop`	Determine if an item is a property of a COM object
`propedit`	Display the control's built-in property page
`set`	Set a property on an interface

Getting the Value of a Property

The `get` function returns information on one or more properties belonging to a COM object or interface. Use `get` with only the handle argument, and MATLAB returns a list of all properties for the object, and their values:

```
h = actxserver('excel.application');

h.get
    Application: [1x1 Interface.Microsoft_Excel_9.0_
Object_Library._Application]
        Creator: 'xlCreatorCode'
         Parent: [1x1 Interface.Microsoft_Excel_9.0_
Object_Library._Application]
```

```
      ActiveCell: []
     ActiveChart: [1x50 char]
                 .
                 .
```

To return the value of just one property, specify the object handle and property name using dot syntax:

```
company = h.OrganizationName
company =
   The MathWorks, Inc.
```

Property names are not case sensitive and may also be abbreviated, as long as you include enough letters in the name to make it unambiguous. You can use `'org'` in place of the full `'OrganizationName'` property name used above:

```
company = h.org
company =
   The MathWorks, Inc.
```

You can also use the `get` function, without dot syntax, for this same purpose:

```
filepath = h.get('DefaultFilePath')
filepath =
   H:\Documents
```

Getting Multiple Property Values. To get more than one property with just one command, you must use the `get` function, specifying the property names in a cell array of strings. This returns a cell array containing a column for each property value:

```
C = h.get({'prop1', 'prop2', ...});
```

For example, to get the `DefaultFilePath` and `UserName` property values for COM object `h`, use

```
h = actxserver('excel.application');

C = h.get({'DefaultFilePath', 'UserName'});
C{:}
ans =
   H:\Documents
```

```
ans =
   C. Coolidge
```

Setting the Value of a Property

The simplest way to set or modify the value of a property is to use an assignment statement like that shown in the second line below. This sets the value of the `DefaultFilePath` property for object `h` to '`C:\ExcelWork`':

```
h = actxserver('excel.application');
h.DefaultFilePath = 'C:\ExcelWork';
```

You can also use the `set` function, without dot syntax, for this same purpose. Specify both the property name and new value as input arguments to `set`:

```
h.set('DefaultFilePath', 'C:\ExcelWork');
```

Setting Multiple Property Values. To change more than one property with one command, you must use the `set` function:

```
h.set('prop1', 'value1', 'prop2', 'value2', ...);
```

For example, to set the `DefaultFilePath` and `UserName` fields of COM object `h`, use

```
h = actxserver('excel.application');
h.set('DefaultFilePath', 'C:\ExcelWork', ...
   'UserName', 'H. Hoover');
```

Properties That Take Arguments

Some COM objects have properties that behave somewhat like methods in that they accept input arguments. On a `get` or `set` operation, the value they end up getting or setting depends on the arguments you pass in.

The `Activesheet` interface of a Microsoft Excel application running as a COM server is one example. This interface has a property called `Range`, which is actually another interface. In order to `get` the correct `Range` interface, you must pass in specific range coordinates.

The first line of code shown here (taken from the example in "Using MATLAB as an Automation Client" on page 8-105) returns a specific `Range` interface.

Arguments `A1` and `B2` specify which rectangular region of the spreadsheet to get the interface for:

```
eActivesheetRange = e.Activesheet.get('Range', 'A1', 'B2')
eActivesheetRange =
   Interface.Microsoft_Excel_5.0_Object_Library.Range
```

To get or set this type of property, use the `get` or `set` function as shown above for the `Range` property. Enter the input arguments in the parentheses following the property name:

```
handle.get(propertyname, arg1, arg2, ...);
```

In some ways, MATLAB handles these properties internally as though they were actually methods. The most important difference is that you need to use `invoke`, not `get`, to view the property:

```
e.Activesheet.invoke
            :
   Range = handle Range(handle, Variant, Variant(Optional))
            :
```

Get and Set on a Vector of Objects

You can use the `get` and `set` functions on more than one object at a time by putting the object handles into a vector and then operating on the vector.

This example creates a vector of handles to four Microsoft Calendar objects. It then modifies the `Day` property of all the objects in one operation by invoking `set` on the vector:

```
h1 = actxcontrol('mscal.calendar', [0 200 250 200]);
h2 = actxcontrol('mscal.calendar', [250 200 250 200]);
h3 = actxcontrol('mscal.calendar', [0 0 250 200]);
h4 = actxcontrol('mscal.calendar', [250 0 250 200]);
H = [h1 h2 h3 h4];

H.set('Day', 23)
H.get('Day')
ans =
    [23]
    [23]
    [23]
    [23]
```

Note To get or set values for multiple objects, you must use the `get` and `set` functions explicitly. Syntax like `H.Day` is only supported for scalar objects.

Using Enumerated Values for Properties

Enumeration makes examining and changing properties easier because each possible value for the property is given a string to represent it. For example, one of the values for the `DefaultSaveFormat` property in an Excel application is `xlUnicodeText`. This is easier to remember than a numeric value like 57.

Finding All Enumerated Properties. The MATLAB COM `get` and `set` functions support enumerated values for properties for those applications that provide them. To see which properties use enumerated types, use the `set` function with the object handle argument:

```
h = actxserver('excel.application');

h.set
ans =
                    Creator: {'xlCreatorCode'}
        ConstrainNumeric: {}
    CopyObjectsWithCells: {}
                  Cursor: {4x1 cell}
            CutCopyMode: {2x1 cell}
                    .
                    .
```

MATLAB displays the properties that accept enumerated types as nonempty cell arrays. Properties for which there is a choice of settings are displayed as a multirow cell array, with one row per setting (see `Cursor` in the example above). Properties for which there is only one possible setting are displayed as a one row cell array (see `Creator`, above).

To display the current values of these properties, use `get` with just the object handle argument:

```
h.get
                    Creator: 'xlCreatorCode'
        ConstrainNumeric: 0
    CopyObjectsWithCells: 1
                  Cursor: 'xlDefault'
            CutCopyMode: ''
                    .
                    .
```

Setting an Enumerated Value. To list all possible enumerated values for a specific property, use `set` with the property name argument. The output is a cell array of strings, one string for each possible setting of the specified property:

```
h.set('Cursor')
ans =
    'xlIBeam'
    'xlDefault'
    'xlNorthwestArrow'
    'xlWait'
```

To set the value of a property, assign the enumerated value to the property name:

```
handle.property = 'enumeratedvalue';
```

You can also use the `set` function with the property name and enumerated value:

```
handle.set('property', 'enumeratedvalue');
```

You have a choice of using the enumeration or its equivalent numeric value. You can abbreviate the enumeration string, as in the third line below, as long as you use enough letters in the string to make it unambiguous. Enumeration strings are also case insensitive.

Make the Excel spreadsheet window visible and then change the cursor from the MATLAB client. To see how the cursor has changed, you need to click the spreadsheet window. Either of the following assignments to `h.Cursor` sets the cursor to the `Wait` (hourglass) type:

```
h.Visible = 1;

h.Cursor = 'xlWait'
h.Cursor = 'xlw'           % Abbreviated form of xlWait
```

Read the value of the `Cursor` property you have just set:

```
h.Cursor
ans =
   xlWait
```

Using the Property Inspector

MATLAB provides a GUI to display and modify properties. Open the Property Inspector using either of these two methods:

- Call the `inspect` function from the MATLAB command line.
- Double-click the object in the MATLAB Workspace browser.

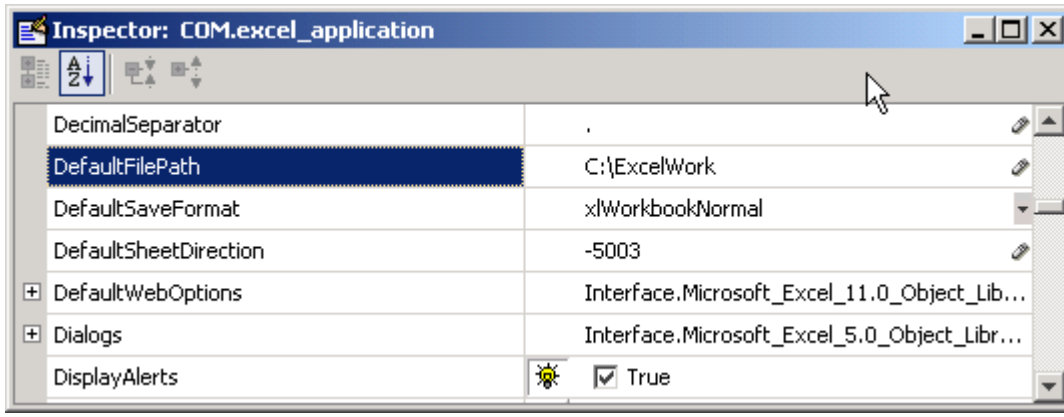
For example, create a server object running Microsoft Excel, then set the object's `DefaultFilePath` property to '`C:\ExcelWork`':

```
h = actxserver('excel.application');
h.DefaultFilePath = 'C:\ExcelWork';
```

Next call the `inspect` function to display a new window showing the server object's properties:

```
h.inspect
```

Scroll down until you see the `DefaultFilePath` property that you just changed. It should read `C:\ExcelWork`.



Using the Property Inspector, change `DefaultFilePath` once more, this time to `C:\MyWorkDirectory`. To do this, select the value at the right and type the new value.

Now go back to the MATLAB Command Window and confirm that the `DefaultFilePath` property has changed as expected.

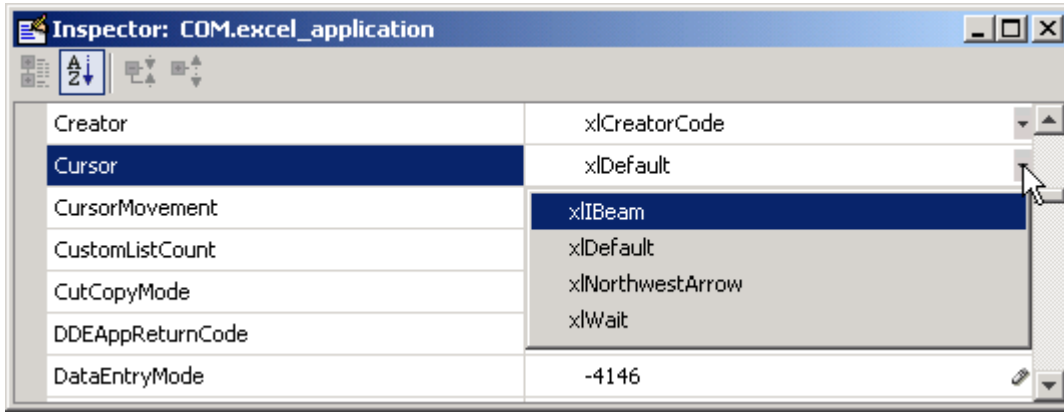
```
h.DefaultFilePath
```

MATLAB displays:

```
ans =

C:\MyWorkDirectory
```

Note If you modify properties at the MATLAB command line, you must refresh the Property Inspector window to see the change reflected there. Refresh the Property Inspector window by reinvoking `inspect` on the object.

Using the Property Inspector on Enumerated Values. A drop-down list button next to a property value indicates the property accepts enumerated values. Click anywhere in the field on the right to see the values. The following figure displays four enumerated values for the `Cursor` property. The current value is displayed in the field next to the property name.



To change a property's value using the Property Inspector, use the drop-down list to display the options for that property, and then click the desired value.

Custom Properties

You can attach your own properties to a control using the `addproperty` function. The syntax shown here creates a custom property for control, `h`:

```
h.addproperty('propertyName')
```

This example creates the `mwsamp2` control, adds a new property called `Position` to it, and assigns the value `[200 120]` to that property:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [200 120 200 200]);
h.addproperty('Position');
h.Position = [200 120];
```

Use `get` to list all properties of control, `h`. You see that the new `Position` property has been added:

```
h.get
ans =
     Label: 'Label'
    Radius: 20
  Position: [200 120]

h.Position
ans =
   200   120
```

To remove custom properties from a control, use `deleteproperty` with the following syntax:

```
h.deleteproperty('propertyName')
```

For example, delete the `Position` property that you just created, and use `get` to show that it no longer exists:

```
h.deleteproperty('Position');

h.get
     Label: 'Label'
    Radius: 20
```

Control and Server Events

An *event* is typically a user-initiated action that takes place in a server application which often requires a reaction from the client. For example, a user clicking the mouse at a particular location in a server interface window might require the client take some action in response. When an event is *fired*, the server communicates this occurrence to the client. If the client is *listening* for this particular type of event, it responds by executing a routine called an event handler.

The MATLAB COM client can subscribe to and handle the events fired by an ActiveX control or a COM server. Select which events you want the client to listen to by registering each event you want active along with the event handler to be used in responding to the event. When any registered event takes place in the control or server, the client is notified and responds by executing the appropriate handler routine. Event handlers in MATLAB are often implemented using M-files.

This section covers the following topics on registering and responding to events fired from an ActiveX control or a COM server:

Note MATLAB does not support interface events from a Custom server.

Functions for Working with Events

Use these MATLAB functions to register and unregister events, to list all events, or to list just registered events for a control or server.

Function	Description
`actxcontrol`	Create a COM control and optionally register those events you want the client to listen to
`eventlisteners`	Return a list of events attached to listeners
`events`	List all events, both registered and unregistered, a control or server can generate
`isevent`	Determine if an item is an event of a COM object

Function	Description
`registerevent`	Register an event handler with a control or server event
`unregisterallevents`	Unregister all events for a control or server
`unregisterevent`	Unregister an event handler with a control or server event

When using these functions, enter event names and event handler names as strings or in a cell array of strings. These names are case insensitive, but cannot be abbreviated.

Examples of Event Handlers

The following examples have implementations of event handlers:

- "Example — Grid ActiveX Control in a Figure" on page 8-17
- "Example — Reading Data from Excel" on page 8-24

Responding to Events from a COM Server

This section describes the basic steps you need to take in handling events fired by a COM control or server.

- "Identifying All Events" on page 8-77
- "Registering Those Events You Want to Respond To" on page 8-78
- "Identifying Registered Events" on page 8-78
- "Responding to Events As They Occur" on page 8-79
- "Unregistering Events You No Longer Want to Listen To" on page 8-79

Identifying All Events. Use the `events` function to list all events the control or server is capable of responding to. This function returns a structure array, where each field of the structure is the name of an event handler and the value of that field contains the signature for the handler routine. To invoke `events` on an object with handle `h`, type

```
S = h.events
```

Registering Those Events You Want to Respond To. Use the `registerevent` function to register those server events you want the client to respond to. There are two ways you can register events:

- If you have one function to handle all server events, you can register this common event handler using the syntax

```
h.registerevent('handler');
```

- If you have a separate event handler function for different events, use the syntax

```
h.registerevent({'event1' 'handler1'; 'event2' 'handler2';
...});
```

For ActiveX controls, you can register events at the time you create the control using the `actxcontrol` function.

- To register a common event handler function to respond to all events, use

```
h = actxcontrol('progid', position, figure, 'handler');
```

- To register a separate function to handle each type of event, use

```
h = actxcontrol('progid', position, figure, ...
   {'event1' 'handler1'; 'event2' 'handler2'; ...});
```

The MATLAB client responds only to events you have registered.

Identifying Registered Events. The `eventlisteners` function lists only those events that are currently registered. This function returns a cell array, with each row representing a registered event and the name of its event handler. For example, to invoke `eventlisteners` on an object with handle `h`, type

```
C = h.eventlisteners
```

Responding to Events As They Occur. Whenever a control or server fires an event that the client is listening to, the client responds to the event by invoking one or more event handlers that have been registered for that event. You can implement these routines in M-file programs that you write to handle events. Read more about event handlers in the section on "Writing Event Handlers" on page 8-87.

Unregistering Events You No Longer Want to Listen To. If you have registered events that you now want the client to ignore, you can unregister them at any time using the functions `unregisterevent` and `unregisterallevents` as follows:

- For a server with handle `h`, to unregister all events registered with a common event handling function `handler`, use

  ```
  h.unregisterevent('handler');
  ```

- To unregister individual events `eventN`, each registered with its own event handling function `handlerN`, use

  ```
  h.unregisterevent({'event1' 'handler1'; 'eventN' 'handlerN'});
  ```

- To unregister all events from the server regardless of which event handling function they are registered with, use

  ```
  h.unregisterallevents;
  ```

Responding to Events from an ActiveX Control

This section describes how to handle events fired by an ActiveX control. It uses a control called `mwsamp2` that ships with MATLAB. The event handler routines for `mwsamp2` are defined when you install MATLAB.

Tasks described in this section are

Creating a Control and Registering Events. The `actxcontrol` function not only creates the control object, but can be used to register specific events as well. The code shown here registers two events (`Click` and `MouseDown`) and two respective handler routines (`myclick` and `mymoused`) with the `mwsamp2` control.

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    {'Click' 'myclick'; 'MouseDown' 'mymoused'});
```

If, at some later time, you want to register additional events, use the `registerevent` function:

```
h.registerevent({'DblClick' 'my2click'});
```

You can view the event handler code written for the `mwsamp2` control in the section "Sample Event Handlers" on page 8-90.

Unregister the `DblClick` event before continuing with the example:

```
h.unregisterevent({'DblClick' 'my2click'});
```

Listing Control Events. At this point, only the `Click` and `MouseDown` events should be registered. To see all events that the control can fire, use the `events` function. This function returns a structure array, where each field of the structure is the name of an event handler and the value of that field contains the signature for the handler routine.

To list all events, whether registered or not, use

```
S = h.events
S =
         Click: 'void Click()'
      DblClick: 'void DblClick()'
     MouseDown: 'void MouseDown(int16 Button, int16 Shift,
                 Variant x, Variant y)'
    Event_Args: [1x101 char]

S.Event_Args
ans =
   void Event_Args(int16 typeshort, int32 typelong,
        double typedouble, string typestring, bool typebool)
```

To list only those events that are currently registered with the control, use the `eventlisteners` function. This function returns a cell array, with each row representing a registered event and the name of its event handler.

Use `eventlisteners` to list registered event names and their handler routines:

```
h.eventlisteners
ans =
    'click'        'myclick'
    'mousedown'    'mymoused'
```

Responding to Control Events. When MATLAB creates the `mwsamp2` control, it also displays a figure window showing a label and circle at the center. If you click on different positions in this window, you see a report in the MATLAB Command Window of the `X` and `Y` position of the mouse.

Each time you press the mouse button, the `MouseDown` event fires, calling the `mymoused` function. This function prints the position values for that event to the MATLAB Command Window:

```
The X position is:
ans =
    [122]
The Y position is:
ans =
```

```
    [63]
```

You also see the following line reported in response to the `Click` event:

```
Single click function
```

Double-clicking the mouse does nothing different, since the `DblClick` event has been unregistered.

Unregistering Control Events. When you unregister an event, the client discontinues listening for occurrences of that event. When the event fires, the client does not respond. If you unregister the `MouseDown` event, MATLAB no longer reports the `X` and `Y` position when you click in the window:

```
h.unregisterevent({'MouseDown' 'mymoused'});
```

Now, register the `DblClick` event, connecting it with handler function `my2click`:

```
h.registerevent({'DblClick', 'my2click'});
```

If you call `eventlisteners` again, the registered events are now `Click` and `DblClick`:

```
h.eventlisteners
ans =
    'click'       'myclick'
    'dblclick'    'my2click'
```

When you double-click the mouse button, MATLAB responds to both the `Click` and `DblClick` events by displaying the following in the MATLAB Command Window:

```
Single click function
Double click function
```

An easy way to unregister all events for a control is to use the `unregisterallevents` function. When there are no events registered, `eventlisteners` returns an empty cell array:

```
h.unregisterallevents
```

```
h.eventlisteners
ans =
     {}
```

Clicking the mouse in the control window now does nothing since there are no active events.

If you have events that are registered with a common event handling routine, such as `sampev.m` used in the example below, you can use `unregisterevent` to unregister all of these events in one operation. The example that follows first registers all events from the server with a common handling routine `sampev`. MATLAB now handles any type of event from this server by executing `sampev`:

```
h.registerevent('sampev');
```

Verify the registration by listing all event listeners for that server:

```
h.eventlisteners
ans =
    'click'        'sampev'
    'dblclick'     'sampev'
    'mousedown'    'sampev'
```

Now unregister all events for the server that use the `sampev` event handling routine:

```
h.unregisterevent('sampev');
h.eventlisteners
ans =
     {}
```

Responding to Events from an Automation Server

The next section shows how to handle events fired by an Automation server. The example creates a server running Internet Explorer, registers a common handler for all events, and then has you fire events by browsing to Web sites using the Internet Explorer application.

Tasks described in this section are

- "Creating an Event Handler" on page 8-84

Creating an Event Handler. This example registers all events with the same handler routine, `serverevents`. Since this example does not ship with MATLAB, you have to create the event handler routine yourself.

Create the file `serverevents.m`, inserting the following code. Make sure the file is in your current directory.

```
function serverevents(varargin)

% Display incoming event name
eventname = varargin{end}

% Display incoming event args
eventargs = varargin{end-1}
```

Creating a Server. Next, in your MATLAB session, use the following commands to create your Automation server application.

```
% Create a server running Internet Explorer.
h = actxserver('internetexplorer.application');

% Make the server application visible.
h.set('Visible', 1);
```

Listing Server Events. Use the `events` function to list all events the control or server is capable of responding to, and `eventlisteners` to list only those events that are currently registered.

```
h.events
```

MATLAB displays event information similar to:

```
        :
StatusTextChange = void StatusTextChange(string Text)
ProgressChange = void ProgressChange(int32 Progress,int32 ProgressMax)
CommandStateChange = void CommandStateChange(int32 Command,bool Enable)
        :
```

No events are registered at this time, so `eventlisteners` returns an empty cell array.

```
h.eventlisteners
```

MATLAB displays:

```
ans =
      {}
```

Registering Server Events. Now use your event handler `serverevents`.

```
h.registerevent('serverevents');
h.eventlisteners
```

MATLAB displays:

```
ans =
              :                    :
    'statustextchange'      'serverevents'
    'progresschange'        'serverevents'
    'commandstatechange'    'serverevents'
              :                    :
```

Responding to Server Events. At this point, all events have been registered. If any event fires, the common handler routine defined in `serverevents.m` executes to handle that event. Use the Internet Explorer application to browse your favorite Web site, or enter the following command in the MATLAB Command Window:

```
h.Navigate2('http://www.mathworks.com');
```

You should see a long series of events displayed in your client window.

Unregistering Server Events. Use the `unregisterevent` function to remove specific events from the registry. If the events were registered with a common handler, as in this example, specify the name of the common routine with each event that you want removed from the event registry for that object:

```
h.unregisterevent({'event1', 'commonhandler'; ...
   'event2', 'commonhandler', ...});
```

Continuing with this example, unregister the `progresschange` and `commandstatechange` events:

```
h.unregisterevent({'progresschange', 'serverevents'; ...
   'commandstatechange', 'serverevents'});
```

To unregister all events for an object, use `unregisterallevents`. The following commands unregister all the events that had been registered for the Internet Explorer application and then register a single event:

```
h.unregisterallevents;
h.registerevent({'TitleChange', 'serverevents'});
```

If you now browse with Internet Explorer, MATLAB only responds to the `TitleChange` event.

Closing the Application. It is always advisable to close a server application when you no longer intend to use it. To unregister all events and close the application, type:

```
h.unregisterallevents;
h.Quit;
h.delete;
```

Responding to Interface Events from an Automation Server

This example, demonstrating how to handle a COM interface event, shows how to set up an event in a Microsoft Excel's Workbook object and how to handle its `BeforeClose` event.

To create the event handler `OnBeforeCloseWorkbook`, create the file `OnBeforeCloseWorkbook.m`, inserting the following code. Make sure the file is in your current directory.

```
% Event handler for Excel workbook BeforeClose event
function OnBeforeCloseWorkbook(varargin)
    disp('BeforeClose event occured');
```

When you run the following commands

```
% Create Excel automation server instance
xl = actxserver('Excel.Application');
% Make it visible
xl.Visible = 1;

% Get collection of workbooks and add a new workbook
hWbks = xl.Workbooks;
hWorkbook = hWbks.Add;

% Register OnClose event
hWorkbook.registerevent({'BeforeClose' @OnBeforeCloseWorkbook});

%% Close the workbook. This will fire Close event and call OnClose handler
hWorkbook.Close
```

MATLAB displays:

```
BeforeClose event occured
```

Writing Event Handlers

This section covers the following topics on writing handler routines to respond to events fired from an ActiveX control or Automation server:

Overview of Event Handling

An event is fired when a control or server wants to notify its client that something of interest has occurred. For example, many controls trigger an event when the user clicks somewhere in the interface window of a control. In MATLAB, you can create and register your own M-file functions to respond to events when they occur. These functions serve as event handlers. You can create one handler function to handle all events or a separate handler for each type of event.

For controls, you can register handler functions either at the time you create the control (using `actxcontrol`), or at any time afterwards (using `registerevent`).

Both `actxcontrol` and `registerevent` use an event handler argument. The event handler argument can be either the name of a single callback routine or a cell array that associates specific events with their respective event handlers. Strings used in the event handler argument are not case sensitive.

For servers, you must use `registerevent` to register those events you want the client to listen to. For example, to register the `Click` and `DblClick` events, use

```
h.registerevent({'click' 'myclick'; 'dblclick' 'my2click'});
```

Use `events` to list all the events a COM object recognizes. For example, to list all events for the `mwsamp2` control, use

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

h.events
   Click = void Click()
   DblClick = void DblClick()
   MouseDown = void MouseDown(int16 Button, int16 Shift,
      Variant x, Variant y)
```

Arguments Passed to Event Handlers

When a registered event is triggered, MATLAB passes information from the event to its handler function, as shown in this table.

Arguments Passed by MATLAB

Arg. No.	Contents	Format
1	Object name	MATLAB COM class
2	Event ID	double
3	Start of Event Argument List	As passed by the control
end-2	End of Event Argument List (Argument N)	As passed by the control
end-1	Event Structure	structure
end	Event Name	char array

When writing an event handler function, use the Event Name argument to identify the source of the event. Get the arguments passed by the control from the Event Argument List (arguments 3 through end-2). All event handlers must accept a variable number of arguments:

```
function event (varargin)
if (varargin{end}) == 'MouseDown')      % Check the event name
    x_pos = varargin{5};              % Read 5th Event Argument
    y_pos = varargin{6};              % Read 6th Event Argument
end
```

Note The values passed vary with the particular event and control being used.

Event Structure

The second to last argument passed by MATLAB is the Event Structure, which has the following fields.

Fields of the Event Structure

Field Name	Description	Format
`Type`	Event Name	`char array`
`Source`	Control Name	MATLAB `COM` class
`EventID`	Event Identifier	`double`
`Event Arg Name 1`	Event Arg Value 1	As passed by the control
`Event Arg Name 2`	Event Arg Value 2	As passed by the control
etc.	Event Arg N	As passed by the control

For example, when the `MouseDown` event of the `mwsamp2` control is triggered, MATLAB passes this Event Structure to the registered event handler:

```
    Type: 'MouseDown'
  Source: [1x1 COM.mwsamp.mwsampctrl.2]
 EventID: -605
  Button: 1
   Shift: 0
       x: 27
       y: 24
```

Sample Event Handlers

Specify a single callback, `sampev`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
    gcf, 'sampev')
h =
    COM.mwsamp.mwsampctrl.2
```

Or specify several events using the cell array format:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f, ...
    {'Click' 'myclick'; 'DblClick' 'my2click'; ...
    'MouseDown' 'mymoused'});
```

The event handlers, `myclick.m`, `my2click.m`, and `mymoused.m`, are

```
function myclick(varargin)
disp('Single click function')

function my2click(varargin)
disp('Double click function')

function mymoused(varargin)
disp('You have reached the mouse down function')
disp('The X position is: ')
double(varargin{5})
disp('The Y position is: ')
double(varargin{6})
```

Alternatively, you can use the same event handler for all the events you want to monitor using the cell array pairs. Response time is better than using the callback style.

For example:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', ...
[0 0 200 200], f, {'Click' 'allevents'; ...
'DblClick' 'allevents'; 'MouseDown' 'allevents'})
```

where `allevents.m` is

```
function allevents(varargin)
if (strcmp(varargin{end-1}.Type, 'Click'))
   disp ('Single Click Event Fired')
elseif (strcmp(varargin{end-1}.Type, 'DblClick'))
   disp ('Double Click Event Fired')
elseif (strcmp(varargin{end-1}.Type, 'MouseDown'))
   disp ('Mousedown Event Fired')
end
```

Writing Event Handlers Using M-File Subfunctions

Instead of having to maintain a separate M-file for every event handler routine you write, you can consolidate some or all of these routines into a single M-file using M-file subfunctions.

This example shows three event handler routines, (`myclick`, `my2click`, and `mymoused`) implemented as subfunctions in the file `mycallbacks.m`. The call to `str2func` converts the input string to a function handle:

```
function a = mycallbacks(str)
a = str2func(str);

function myclick(varargin)
disp('Single click function')

function my2click(varargin)
disp('Double click function')

function mymoused(varargin)
disp('You have reached the mouse down function')
disp('The X position is: ')
double(varargin{5})
disp('The Y position is: ')
double(varargin{6})
```

To register one of these events, call `mycallbacks`, passing the name of the event handler:

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], ...
    gcf, 'sampev')
h.registerevent({'Click', mycallbacks('myclick')});
```

Saving Your Work

Use these MATLAB functions to save and restore the state of a COM control object.

Function	Description
`load`	Load and initialize a COM control object from a file
`save`	Write and serialize a COM control object to a file

Save the current state of a COM control to a file using the `save` function. The following example creates an `mwsamp2` control and saves its original state to the file `mwsample`:

```
f = figure('position', [100 200 200 200]);
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 200 200], f);
h.save('mwsample')
```

Now, alter the figure by changing its label and the radius of the circle:

```
h.Label = 'Circle';
h.Radius = 50;
h.Redraw;
```

Using the `load` function, you can restore the control to its original state:

```
h.load('mwsample');
h.get
ans =
     Label: 'Label'
    Radius: 20
```

Note The COM `save` and `load` functions are only supported for controls at this time.

Releasing COM Interfaces and Objects

Use these MATLAB functions to release or delete a COM object or interface.

Function	Description
`delete`	Delete a COM object or interface
`release`	Release a COM object or interface

When each interface is no longer needed, use the `release` function to release the interface and reclaim the memory used by it. When the entire control or server is no longer needed, use the `delete` function to delete it. Alternatively, you can use the `delete` function on any valid interface. All interfaces for that object are automatically released and the server or control itself is deleted.

Note In versions of MATLAB earlier than 6.5, failure to explicitly release interface handles or delete the control or server often results in a memory leak. This is true even if the variable representing the interface or COM object has been reassigned. In MATLAB 6.5 and later, the control or server, along with all interfaces to it, is destroyed on reassignment of the variable or when the variable representing a COM object or interface goes out of scope.

MATLAB automatically releases all interfaces for a control when the figure window that contains that control is deleted or closed. MATLAB also automatically releases all handles for an Automation server when MATLAB is shut down.

Identifying Objects

Use these MATLAB functions to get information about a COM object.

Function	Description
`class`	Return the class of a COM object
`isa`	Detect a COM object of a given class
`isevent`	Determine if an item is an event of a COM object
`ismethod`	Determine if an item is a method of a COM object
`isprop`	Determine if an item is a property of a COM object

Create a COM object, `h`, in an Automation server running Excel, and also a `Workbooks` interface, `w`, to the object:

```
h = actxserver('excel.application');
w = h.Workbooks;
```

To find out the class of variable `w`, use the `class` function:

```
w.class
ans =
   Interface.Microsoft_Excel_9.0_Object_Library.Workbooks
```

To test a variable against a known class name, use `isa`:

```
h.isa('COM.excel.application')
ans =
     1
```

To see if `UsableWidth` is a property of object `h`, use `isprop`:

```
h.isprop('UsableWidth')
ans =
     1
```

To see if `SaveWorkspace` is a method of object `h`, use `ismethod`. Method names are case sensitive and cannot be abbreviated:

```
h.ismethod('SaveWorkspace')
ans =
     1
```

Create the sample `mwsamp2` control that comes with MATLAB, and use `isevent` to see if `DblClick` is one of the events that this control recognizes:

```
f = figure ('position', [100 200 200 200]);
h = actxcontrol ('mwsamp.mwsampctrl.2', [0 0 200 200], f);

h.isevent('DblClick')
ans =
     1
```

Handling COM Data in MATLAB

This section covers the following topics:

• "Displaying MATLAB Syntax for COM Objects" on page 8-102

Passing Data to a COM Object

When you call a COM object from MATLAB, the MATLAB data types you pass in the call are converted to data types native to the COM object. MATLAB performs this conversion on each argument that is passed. This section describes the conversion.

MATLAB arguments are converted by MATLAB into data types that best represent the data to the COM object. The following table shows all of the MATLAB base types for passed arguments and the COM types defined for input arguments. Each row shows a MATLAB type followed by the possible COM argument matches.

MATLAB Argument	Closest Type	Allowed Types
handle	VT_DISPATCH VT_UNKNOWN	VT_DISPATCH VT_UNKNOWN
string	VT_BSTR	VT_LPWSTR VT_LPSTR VT_BSTR VT_FILETIME VT_ERROR VT_DECIMAL VT_CLSID VT_DATE
int16	VT_I2	VT_UINT VT_I2 VT_UI2
int32	VT_I4	VT_I4 VT_UI4 VT_INT
single	VT_R4	VT_R4
double	VT_R8	VT_R8 VT_CY (currency)

MATLAB Argument	Closest Type	Allowed Types
bool	VT_BOOL	VT_BOOL
char	VT_I1	VT_I1 VT_UI1

Variant Data. variant is any data type except a structure or a sparse array. (Refer to the Data Type Summary table in the MATLAB Programming documentation.)

When used as an input argument, MATLAB treats variant and variant(pointer) the same way.

MATLAB Argument	Closest Type	Allowed Types
variant	VT_VARIANT	VT_VARIANT VT_USERDEFINED VT_ARRAY
variant(pointer)	VT_VARIANT	VT_VARIANT | VT_BYREF

SAFEARRAY Data. When a COM method identifies a SAFEARRAY or SAFEARRAY(pointer), the MATLAB equivalent is a matrix.

MATLAB Argument	Closest Type	Allowed Types
SAFEARRAY	VT_SAFEARRAY	VT_SAFEARRAY
SAFEARRAY(pointer)	VT_SAFEARRAY	VT_SAFEARRAY | VT_BYREF

Handling Data from a COM Object

Data returned from a COM object is often incompatible with MATLAB data types. When this occurs, MATLAB converts the returned value to a data type native to the MATLAB language. This section describes the conversion performed on the various data types that can be returned from COM objects.

This table shows how MATLAB converts data from a COM object into MATLAB variables.

COM Return Type	MATLAB Representation
VT_DISPATCH VT_UNKNOWN	handle
VT_LPWSTR VT_LPSTR VT_BSTR VT_FILETIME VT_ERROR VT_DECIMAL VT_CLSID VT_DATE	string
VT_UINT VT_I2 VT_UI2	int16
VT_I4 VT_UI4 VT_INT	int32
VT_R4	single
VT_R8 VT_CY (currency)	double
VT_BOOL	bool
VT_I1 VT_UI1	char
VT_VARIANT VT_USERDEFINED VT_ARRAY	variant
VT_VARIANT | VT_BYREF	variant(pointer)
VT_SAFEARRAY	SAFEARRAY
VT_SAFEARRAY | VT_BYREF	SAFEARRAY(pointer)

Unsupported Data Types

The following data types are not supported in the MATLAB COM interface:

- VT_I8
- VT_UI8
- Structure
- Sparse array
- Unsigned integer
- Multidimensional SAFEARRAYs
- Write-only properties
- Enumerated types

Passing Data from MATLAB to ActiveX Objects

The tables also show the mapping of MATLAB data types to COM data types that you must use to pass data from MATLAB to an ActiveX object. Note that all other types result in the following warning:

```
"ActiveX - invalid argument type or value".
```

Passing SAFEARRAY from MATLAB to COM Object

The SAFEARRAY data type is a standard way to pass arrays between COM objects. This section explains how MATLAB passes SAFEARRAY data to a COM object.

- “Default Behavior in MATLAB” on page 8-99
- “Examples” on page 8-99
- “How to Pass a Single-Dimension SAFEARRAY” on page 8-101
- “Passing SAFEARRAY By Reference” on page 8-101

Default Behavior in MATLAB. MATLAB represents an m-by-n matrix as a two-dimensional SAFEARRAY, where the first dimension has m elements and the second dimension has n elements. MATLAB passes the SAFEARRAY by value.

Examples. The following examples use a COM object that expects a SAFEARRAY input parameter.

When MATLAB passes a 1-by-3 array

```
B = [2 3 4]
B =
     2     3     4
```

the object reads

```
No. of dimensions: 2
Dim: 1,   No. of elements: 1
Dim: 2,   No. of elements: 3
    Elements:
    2.0
    3.0
    4.0
```

When MATLAB passes a 3-by-1 array

```
C = [1;2;3]
C =
     1
     2
     3
```

the object reads

```
No. of dimensions: 2
Dim: 1,   No. of elements: 3
Dim: 2,   No. of elements: 1
Elements:
    1.0
    2.0
    3.0
```

When MATLAB passes a 2-by-4 array

```
D = [2 3 4 5;5 6 7 8]

D =
     2     3     4     5
     5     6     7     8
```

the object reads

```
No. of dimensions: 2
Dim: 1,   No. of elements: 2
Dim: 2,   No. of elements: 4
Elements:
    2.0
    3.0
    4.0
    5.0
    5.0
    6.0
    7.0
    8.0
```

How to Pass a Single-Dimension SAFEARRAY. For information about passing arguments as one-dimensional arrays to a COM object, see the Technical Support solution 1-SKYP9 at `http://www.mathworks.com/support/solutions/data/1-SKYP9.html?solution=1-SKY`

Passing SAFEARRAY By Reference. For information about passing arguments by reference to a COM object, see the Technical Support solution 1-SKYPY at `http://www.mathworks.com/support/solutions/data/1-SKYPY.html?solution=1-SKY`

Reading SAFEARRAY from a COM Object in MATLAB

This section explains how MATLAB reads `SAFEARRAY` data from a COM object.

A one-dimensional `SAFEARRAY` with `n` elements from a COM object is rendered as a 1-by-`n` matrix. For example, using methods from the MATLAB sample control `mwsamp`,

```
h=actxcontrol('mwsamp.mwsampctrl.1')
a = h.GetI4Vector
```

MATLAB displays:

```
a =
           1           2           3
```

A two-dimensional SAFEARRAY with n elements is rendered by MATLAB as a 2-by-n matrix, for example:

```
a = h.GetR8Array
```

MATLAB displays:

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

A three-dimensional SAFEARRAY with 2 elements is rendered as a 2-by-2-by-2 cell array, for example:

```
a = h.GetBSTRArray
```

MATLAB displays:

```
a(:,:,1) =

    '1 1 1'    '1 2 1'
    '2 1 1'    '2 2 1'

a(:,:,2) =

    '1 1 2'    '1 2 2'
    '2 1 2'    '2 2 2'
```

Displaying MATLAB Syntax for COM Objects

To determine which MATLAB data types to use when passing arguments to COM objects, use the invoke or methodsview functions. These functions list all of the methods found in an object, along with a specification of the data types required for each argument.

In the following example, a server called MyApp has a method TestMeth1 with the following syntax:

```
HRESULT TestMeth1 ([out, retval] double* dret);
```

This method has no input argument, and it returns a variable of type `double`. To display the MATLAB syntax for calling the method, type

```
h = actxserver('MyApp');
h.invoke
```

MATLAB displays:

```
ans =
   TestMeth1 = double TestMeth1 (handle)
```

The signature of `TestMeth1` is

```
double TestMeth1(handle)
```

MATLAB requires you to use an object handle as an input argument for every method, in addition to any input arguments required by the method itself.

Using the variable `var`, which is of type `double`, type

```
var = h.TestMeth1;
```

or

```
var = TestMeth1(h);
```

While the following syntax is correct, its use is discouraged:

```
var = invoke(h,'TestMeth1');
```

Now consider the server called `MyApp1` with the following methods:

```
HRESULT TestMeth1 ([out, retval] double* dret);
HRESULT TestMeth2 ([in] double* d, [out, retval] double* dret);
HRESULT TestMeth3 ([out] BSTR* sout,
                   [in, out] double* dinout,
                   [in, out] BSTR* sinout,
                   [in] short sh,
                   [out] long* ln,
                   [in, out] float* b1,
                   [out, retval] double* dret);
```

Type the commands:

```
h = actxserver('MyApp1');
h.invoke
```

MATLAB displays the list of methods

```
ans =
   TestMeth1 = double TestMeth1 (handle)
   TestMeth2 = double TestMeth1 (handle, double)
   TestMeth3 = [double, string, double, string, int32, single] ...
               TestMeth3(handle, double, string, int16, single)
```

`TestMeth2` requires an input argument `d` of type `double`, as well as returning a variable `dret` of type `double`. Some examples of calling `TestMeth2` are

```
var = h.TestMeth2(5);
```

or

```
var = TestMeth2(h, 5);
```

`TestMeth3` requires multiple input arguments, as indicated within the parentheses on the right side of the equals sign, and returns multiple output arguments, as indicated within the brackets on the left side of the equals sign.

```
[double, string, double, string, int32, single]  %output arguments
TestMeth3(handle, double, string, int16, single) %input arguments
```

The first input argument is the required `handle`, followed by four input arguments.

```
TestMeth3(handle, in1, in2, in3, in4)
```

The first output argument is the return value `retval`, followed by five output arguments.

```
[retval, out1, out2, out3, out4, out5]
```

This is how the arguments map into a MATLAB command:

```
[dret, sout, dinout, sinout, ln, b1] = TestMeth3(handle, ...
```

```
                                                    dinout, sinout, sh, b1)
```

where `dret` is `double`, `sout` is `string`, `dinout` is `double` and is both an input and an output argument, `sinout` is `string` (input and output argument), `ln` is `int32`, `b1` is `single` (input and output argument), `handle` is the handle to the object, and `sh` is `int16`.

Examples of MATLAB as an Automation Client

This section provides examples of using MATLAB as an Automation client with controls and servers:

MATLAB Sample Control

MATLAB ships with a simple example COM control that draws a circle on the screen, displays some text, and fires events when the user single- or double-clicks on the control. Create the control by running the `mwsamp.m` file in the directory, `winfun\comcli`, or type

```
h = actxcontrol('mwsamp.mwsampctrl.2', [0 0 300 300]);
```

This control is stored in the MATLAB `bin`, or executable, directory along with the control's *type library*. The type library is a binary file used by COM tools to decipher the control's capabilities. See the section "Writing Event Handlers" on page 8-87 for other examples that use the `mwsamp2` control.

Using MATLAB as an Automation Client

This example uses MATLAB as an Automation client and Microsoft Excel as the server. It provides a good overview of typical functions. In addition, it is a good example of using the Automation interface of another application:

```
% MATLAB Automation client example
%
```

```
% Open Excel, add workbook, change active worksheet,
% get/put array, save.

% First, open an Excel Server.
e = actxserver('excel.application');

% Insert a new workbook.
eWorkbook = e.Workbooks.Add;
e.Visible = 1;

% Make the first sheet active.
eSheets = e.ActiveWorkbook.Sheets;

eSheet1 = eSheets.get('Item', 1);
eSheet1.Activate;

% Put a MATLAB array into Excel.
A = [1 2; 3 4];
eActivesheetRange = e.Activesheet.get('Range', 'A1:B2');
eActivesheetRange.Value = A;

% Get back a range.
% It will be a cell array, since the cell range
% can contain different types of data.
eRange = e.Activesheet.get('Range', 'A1:B2');
B = eRange.Value;

% Convert to a double matrix. The cell array must contain only
% scalars.
B = reshape([B{:}], size(B));

% Now, save the workbook.
eWorkbook.SaveAs('myfile.xls');

% Avoid saving the workbook and being prompted to do so
eWorkbook.Saved = 1;
eWorkbook.Close;

% Quit Excel and delete the server.
e.Quit;
```

```
e.delete;
```

Note Make sure that you always close any workbooks that you add in Excel. This can prevent potential memory leaks.

Connecting to an Existing Excel Application

You can give MATLAB access to a file that is open by another application by creating a new COM server from the MATLAB client, and then opening the file through this server. This example shows how to do this for an Excel application that has a file `weekly_log.xls` open:

```
excelapp = actxserver('Excel.Application');
wkbk = excelapp.Workbooks;
wdata = wkbk.Open('d:\weatherlog\weekly_log.xls');
```

To see what methods are available, type

```
wdata.methods
   Methods for class Interface.Microsoft_Excel_10.0_
      Object_Library._Workbook:

   AcceptAllChanges     LinkInfo        ReloadAs
   Activate             LinkSources     RemoveUser
      :                    :               :
      :                    :               :
```

Access data from the spreadsheet by selecting a particular sheet (called `'Week 12'` in the example), selecting the range of values (the rectangular area defined by `D1` and `F6` here), and then reading from this range:

```
sheets = wdata.Sheets;
sheet12 = sheets.Item('Week 12');
range = sheet12.get('Range', 'D1', 'F6');
range.value

ans =
    'Temp.'      'Heat Index'    'Wind Chill'
    [78.4200]    [        32]    [        37]
```

```
    [69.7300]    [        27]    [        30]
    [77.6500]    [        17]    [        16]
    [74.2500]    [        -5]    [         0]
    [68.1900]    [        22]    [        35]

wkbk.Close;
excelapp.Quit;
```

Running a Macro in an Excel Server Application

In the example below, MATLAB runs Microsoft Excel in a COM server and invokes a macro that has been defined within the active Excel spreadsheet file. The macro, `init_last`, takes no input parameters and is called from the MATLAB client using the statement

```
handle.ExecuteExcel4Macro('!macroname()');
```

Start the example by opening the spreadsheet file and recording a macro. The macro used here simply sets all items in the last column to zero. Save your changes to the spreadsheet.

Next, in MATLAB, create a COM server running an Excel application, and open the spreadsheet:

```
h = actxserver('Excel.Application');
wkbk = h.Workbooks;
file = wkbk.Open('d:\weatherlog\weekly.xls');
```

Open the sheet that you want to change, and retrieve the current values in the range of interest:

```
sheets = file.Sheets;
sheet12 = sheets.Item('Week 12');
range = sheet12.get('Range', 'D1', 'F5');
range.Value
ans =
    [     78]    [        32]    [        37]
    [     69]    [        27]    [        30]
    [     77]    [        17]    [        16]
    [     74]    [        -5]    [        -1]
    [     68]    [        22]    [        35]
```

Now execute the macro, and verify that the values have changed as expected:

```
h.ExecuteExcel4Macro('!init_last()');
range.Value
ans =
    [     78]    [         32]    [          0]
    [     69]    [         27]    [          0]
    [     77]    [         17]    [          0]
    [     74]    [         -5]    [          0]
    [     68]    [         22]    [          0]
```

MATLAB COM Client Demo

MATLAB includes a demo illustrating the use of the COM Client with MATLAB. To run the demo, click the **Demos** tab in the MATLAB Help browser. Click to expand the folder called `External Interfaces` and select `Programming with COM`.

Additional COM Client Information

In this section...

Using COM Collections

COM *collections* are a way to support groups of related COM objects that can be iterated over. A collection is itself a special interface with a `Count` property (read only), which contains the number of items in the collection, and an `Item` method, which allows you to retrieve a single item from the collection.

The `Item` method is indexed, which means that it requires an argument that specifies which item in the collection is being requested. The data type of the index can be any data type that is appropriate for the particular collection and is specific to the control or server that supports the collection. Although integer indices are common, the index could just as easily be a string value. Often, the return value from the `Item` method is itself an interface. Like all interfaces, this interface should be released when you are finished with it.

This example iterates through the members of a collection. Each member of the collection is itself an interface (called `Plot` and represented by a MATLAB COM object called `hPlot`.) In particular, this example iterates through a collection of `Plot` interfaces, invokes the `Redraw` method for each interface, and then releases each interface:

```
hCollection = hControl.Plots;
for i = 1:hCollection.Count
   hPlot = hCollection.invoke('Item', i);
   hPlot.Redraw;
   hPlot.release;
end;
hCollection.release;
```

Using MATLAB as a DCOM Client

Distributed Component Object Model (DCOM) is a protocol that allows clients to use remote COM objects over a network. Additionally, MATLAB can be used as a DCOM client with remote Automation servers if the operating system on which MATLAB is running is DCOM enabled.

Note If you use MATLAB as a remote DCOM server, all MATLAB windows appears on the remote machine.

MATLAB COM Support Limitations

The following is a list of limitations of MATLAB COM support:

- MATLAB only supports indexed collections.
- COM controls are not printed with figure windows.
- “Unsupported Data Types” on page 8-98

MATLAB COM Automation Server Support

In this section...

Introduction

Automation is a COM protocol that allows one application (the *controller*) to control objects exported by another application (the *server*). MATLAB on Microsoft Windows supports COM Automation server capabilities. Any Windows program that can be configured as an Automation controller can control MATLAB. Some examples of applications that can be Automation controllers are Microsoft Excel, Microsoft Access, Microsoft Project, and many Visual Basic and Visual C++ programs.

Note If you plan to build your client application using C, C++, or Fortran, we recommend you use MATLAB Engine instead of an Automation server.

Creating the MATLAB Server

A controller needs a programmatic identifier (ProgID) to identify the server. MATLAB's ProgID is `matlab.application`.

Exactly how you create an Automation server depends on the controller you are using. Consult your controller's documentation for this information.

If your controller is a MATLAB application, you can create the Automation server using `actxserver`:

```
h = actxserver('matlab.application')
h =
   COM.matlab.application
```

This command automatically creates the Automation server. You can also create the server manually. See “Creating the Server Manually” on page 8-122.

Shared and Dedicated Servers

The MATLAB Automation server has two modes:

- Shared — One or more client applications connect to the same MATLAB server. The server is shared between all clients.
- Dedicated — Each client application creates its own dedicated MATLAB server.

If you use `matlab.application` as your ProgID, MATLAB creates a shared server. See “Specifying a Shared or Dedicated Server” on page 8-123.

Startup Directory

The Automation server starts up in the *`matlabroot`*`\bin\win32` directory. If this is not the startup directory, the newly created server does not run the MATLAB startup file (`startup.m`) and does not have access to files in that directory.

To access files in the startup directory, do one of the following:

- Set the server’s working directory to the startup directory (using `cd`) and add the startup directory to the server’s MATLAB path (using `addpath`).
- Include the path name to the startup directory when referencing those files.

Get the Status of a MATLAB Automation Server

Using the `enableservice` function you can learn the current state of a MATLAB Automation server. The function returns a logical value, where logical `1` (`true`) means MATLAB is an Automation server and logical `0` (`false`) means MATLAB is not an Automation server.

For example, if you type

```
enableservice('AutomationServer')
```

and MATLAB displays:

```
ans =

     1
```

then MATLAB is currently an Automation server.

Creating a MATLAB Automation Server from Visual Basic .NET

If you are using a Visual Basic client application to access a MATLAB Automation server, you can start the server using one of the following two methods:

- `New MLApp.MLApp`
- `CreateObject`

The first method requires you to reference the MATLAB *type library* in your Visual Basic project. By using the Object Browser of your Visual Basic client application you can see what methods are available from a MATLAB Automation server. Use the following procedure to reference the MATLAB Application Type Library:

1. Select the **Project** menu.
2. Select **Reference** from the subsequent menu.
3. Check the box next to the **MATLAB Application Type Library**.
4. Click **OK**.

Start the server with the following code:

```
Matlab = New MLApp.MLApp
```

View MATLAB automation methods from the Visual Basic Object Browser under the Library called `MLAPP`.

Alternatively, use the following code to start the server:

```
MatLab = CreateObject("Matlab.Application")
```

Connecting to an Existing MATLAB Server

It is not always necessary to create a new instance of a MATLAB server whenever your application needs some task done in MATLAB. Clients can connect to an existing MATLAB Automation server using the `actxGetRunningServer` function or by using a command similar to the Visual Basic .NET `GetObject` command.

Using Visual Basic .NET

The Visual Basic .NET command shown here returns a handle `h` to the MATLAB server application:

```
h = GetObject(, "matlab.application")
```

Note It is important to use the syntax shown above to connect to an existing MATLAB Automation server. Omit the first argument, and make sure the second argument is as shown.

The following Visual Basic .NET example connects to an existing MATLAB server, then executes a plot command in the server. If you do not already have a MATLAB server running, create one following the instructions in “Creating a MATLAB Automation Server from Visual Basic .NET” on page 8-114.

```
Dim h As Object
h = GetObject(, "matlab.application")

' Handle h should be valid now. Test it by calling Execute.
h.Execute ("plot([0 18], [7 23])")
```

MATLAB Automation Server Functions and Properties

In this section...

Introduction

As an Automation server, MATLAB provides functions and properties to enable an Automation controller to manipulate data in the MATLAB workspace. MATLAB can be both a controller and a server. The examples in this section use a MATLAB M-file as the client application. "Examples of a MATLAB Automation Server" on page 8-125 shows you how to access MATLAB from other applications.

This section explains how to call functions in the MATLAB Automation server and how to use properties that affect the server. These are shown in the following tables and are described in individual function reference pages.

For a complete list of these functions, see "Component Object Model and ActiveX" in the MATLAB Function Reference documentation.

Executing Commands in the MATLAB Server

The client program can execute commands in the MATLAB server using these functions.

Function	Description
Execute	Execute MATLAB command in server
Feval	Evaluate MATLAB command in server

Using Execute

Use Execute when you want the MATLAB server to execute a command that can be expressed in a single string:

```
h = actxserver('matlab.application');

h.PutWorkspaceData('A', 'base', rand(6))
h.Execute('A(4:6,:) = [];'); % remove rows 4-6
B = h.GetWorkspaceData('A', 'base')
B =
    0.6208    0.2344    0.6273    0.3716    0.7764    0.7036
    0.7313    0.5488    0.6991    0.4253    0.4893    0.4850
    0.1939    0.9316    0.3972    0.5947    0.1859    0.1146
```

Using Feval

Use Feval when you want the server to execute commands that you cannot express in a single string. The following example uses variables defined in the client, rows and cols, to modify the server.

This is a continuation of the example above:

```
rows = 6;   cols = 3;
h.Feval('reshape', 0, 'A=', rows, cols);
```

MATLAB interprets A in the expression 'A=' as a server variable name.

The reshape operation in the statement above does not make an assignment to the server variable A; it is equivalent to the following MATLAB statement:

```
reshape(A,6,3)
```

which returns a result, but does not assign the new array. If you get the variable A from the server, it is unchanged:

```
B = h.GetWorkspaceData('A', 'base')
B =
    0.6208    0.2344    0.6273    0.3716    0.7764    0.7036
    0.7313    0.5488    0.6991    0.4253    0.4893    0.4850
    0.1939    0.9316    0.3972    0.5947    0.1859    0.1146
```

Use the `Feval` method returned value to get the result of this type of operation. For example, the following statement reshapes the server-side array `A` and returns the result of this MATLAB operation in the client-side variable `a`.

```
a = h.Feval('reshape', 1, 'A=', rows, cols);
```

The `Feval` method returns a cell array:

```
a{:}
ans =
    0.6208    0.6273    0.7764
    0.7313    0.6991    0.4893
    0.1939    0.3972    0.1859
    0.2344    0.3716    0.7036
    0.5488    0.4253    0.4850
    0.9316    0.5947    0.1146
```

Date Data Type

When you need to pass a `VT_DATE` type input to a Visual Basic program or an ActiveX control method, you can use the MATLAB class `COM.date`. For example:

```
d = COM.date(2005,12,21,15,30,05);
get(d)
     Value: 7.3267e+005
    String: '12/21/2005 3:30:05 PM'
```

You can use `now` to set the `Value` property to a date number:

```
d.Value = now;
```

Exchanging Data with the Server

MATLAB provides functions to read and write data to any workspace of a MATLAB server. In these commands, pass the name of the variable to read or write, and the name of the workspace holding that data.

Function	Description
`GetCharArray`	Get character array from server
`GetFullMatrix`	Get matrix from server
`GetWorkspaceData`	Get any type of data from server
`PutCharArray`	Store character array in server
`PutFullMatrix`	Store matrix in server
`PutWorkspaceData`	Store any type of data in server

The `Get/PutCharArray` functions read and write string values to the MATLAB server.

The `Get/PutFullMatrix` functions pass data as a `SAFEARRAY` data type. You can use these functions with any client that supports the `SAFEARRAY` type. This includes MATLAB and Visual Basic clients.

The `Get/PutWorkspaceData` functions pass data as a `variant` data type. Use these functions with any client that supports the `variant` type. These functions are especially useful for VBScript clients because VBScript does not support the `SAFEARRAY` data type.

In this example, write a string to variable `str` in the base workspace of the MATLAB server and read it back to the client:

```
h = actxserver('matlab.application');
h.PutCharArray('str', 'base', ...
   'He jests at scars that never felt a wound.');

S = h.GetCharArray('str', 'base')
S =
  He jests at scars that never felt a wound.
```

Controlling the Server Window

These functions enable you to make the server window visible or to minimize it.

Function	Description
`MaximizeCommandWindow`	Display server window on Windows desktop
`MinimizeCommandWindow`	Minimize size of server window

In this example, create a COM server running MATLAB and minimize it:

```
h = actxserver('matlab.application');
h.MinimizeCommandWindow;
```

Terminating the Server Process

When you are finished with the MATLAB server, use these functions to quit the MATLAB session and terminate the server process.

Function	Description
`Quit`	Quit the MATLAB session
`delete`	Terminate MATLAB server process

To quit MATLAB, type

```
h.Quit;
```

To terminate the server process, type

```
h.delete;
```

Client-Specific Information

This section provides information specific to MATLAB and Visual Basic .NET clients only.

For MATLAB Clients

To see a summary of all functions along with the required syntax, use the `invoke` function as follows:

```
handle = actxserver('matlab.application');
handle.invoke
```

For Visual Basic .NET Clients

Data types for the arguments and return values of the server functions are expressed as Automation data types, which are language-independent types defined by the Automation protocol.

For example, `BSTR` is a wide-character string type defined as an Automation type, and is the same data format used by Visual Basic to store strings. Any COM-compliant controller should support these data types, although the details of how you declare and manipulate these are controller specific.

Using the Visible Property

You have the option of making MATLAB visible or not by setting the `Visible` property. When visible, MATLAB appears on the desktop, enabling the user to interact with it. This might be useful for such purposes as debugging. When not visible, the MATLAB window does not appear, thus perhaps making for a cleaner interface and also preventing any interaction with the application.

By default, the `Visible` property is enabled, or set to `1`:

```
h = actxserver('matlab.application');
h.Visible
ans =
     1
```

You can change the `Visible` property by setting it to `0` (invisible) or `1` (visible). The following command removes the server application window from the desktop:

```
h.Visible = 0;
h.Visible
ans =
     0
```

Additional Automation Server Information

In this section...

Creating the Server Manually

An Automation server is created automatically by Windows when a controller application first establishes a server connection. Alternatively, you may choose to create the server manually, prior to starting any of the client processes.

To manually create a MATLAB server, use the `/Automation` switch in the MATLAB startup command. You can do this from the DOS command line by typing

```
matlab /Automation
```

Alternatively, you can add this switch every time you run MATLAB, as follows:

1. Right-click the MATLAB shortcut icon

   and select **Properties** from the context menu. The Properties dialog box for `matlab.exe` opens to the **Shortcut** tab.

2. In the **Target** field, add `/Automation` to the end of the target path for `matlab.exe`. Be sure to include a space between the file name and the symbol `/`. For example:

   ```
   "C:\Program Files\MATLAB\R2006a\bin\win32\MATLAB.exe /Automation"
   ```

Note When Windows automatically creates a MATLAB server, it too uses the `/Automation` switch. In this way, MATLAB servers are differentiated from other MATLAB sessions. This protects controllers from interfering with any interactive MATLAB sessions that may be running.

Specifying a Shared or Dedicated Server

You can start the MATLAB Automation server in one of two modes – shared or dedicated. A dedicated server is dedicated to a single client; a shared server is shared by multiple clients. The mode is determined by the programmatic identifier (ProgID) used by the client to start MATLAB.

Starting a Shared Server

The ProgID, `matlab.application`, specifies the default mode, which is shared. You can also use the version-specific ProgID, `matlab.application.N.M`, where `N` is the major version and `M` is the minor version of your MATLAB. For example, use `N = 7` and `M = 4` for MATLAB 7.4.

Once MATLAB is started as a shared server, all clients that request a connection to MATLAB using the shared server ProgID connect to the already running instance of MATLAB. In other words, there is never more than one instance of a shared server running, since it is shared by all clients that use the shared server ProgID.

Starting a Dedicated Server

To specify a dedicated server, use the ProgID, `matlab.application.single`, (or the version-specific ProgID, `matlab.application.single.N.M`).

Each client that requests a connection to MATLAB using a dedicated ProgID creates a separate instance of MATLAB; it also requests the server not be shared with any other client. Therefore, there can be several instances of a dedicated server running simultaneously, since the dedicated server is not shared by multiple clients.

Using MATLAB as a DCOM Server

Distributed Component Object Model (DCOM) is a protocol that allows COM connections to be established over a network. If you are using a version of Windows that supports DCOM and a controller that supports DCOM, you can use the controller to start a MATLAB server on a remote machine.

To do this, DCOM must be configured properly, and MATLAB must be installed on each machine that is used as a client or server. (Even though the client machine may not be running MATLAB in such a configuration, the

client machine must have a MATLAB installation because certain MATLAB components are required to establish the remote connection.) Consult the DCOM documentation for how to configure DCOM for your environment.

Examples of a MATLAB Automation Server

In this section...

Example — Running an M-File from Visual Basic .NET

This example calls a user-defined M-file function named `solve_bvp` from a Visual Basic client application through a COM interface. It also plots a graph in a new MATLAB window and performs a simple computation:

```
Dim MatLab As Object
Dim Result As String
Dim MReal(1, 3) As Double
Dim MImag(1, 3) As Double

MatLab = CreateObject("Matlab.Application")

'Calling m-file from VB
'Assuming solve_bvp exists at specified location
Result = MatLab.Execute("cd d:\matlab\work\bvp")
Result = MatLab.Execute("solve_bvp")

'Executing other MATLAB commands
Result = MatLab.Execute("surf(peaks)")
Result = MatLab.Execute("a = [1 2 3 4; 5 6 7 8]")
Result = MatLab.Execute("b = a + a ")
'Bring matrix b into VB program
MatLab.GetFullMatrix("b", "base", MReal, MImag)
```

Example — Viewing Methods from a Visual Basic .NET Client

You can find out what methods are available from a MATLAB Automation server using the Object Browser of your Visual Basic client application. To do this, follow this procedure in the client application to reference the MATLAB Application Type Library:

1. Select the **Project** menu.
2. Select **Reference** from the subsequent menu.
3. Check the box next to the **MATLAB Application Type Library**.
4. Click **OK**.

This enables you to view MATLAB Automation methods from the Visual Basic Object Browser under the Library called `MLAPP`. You can also see a list of MATLAB Automation methods when you use the term `Matlab` followed by a period. For example:

```
Dim Matlab As MLApp.MLApp
Private Sub View_Methods()
Matlab = New MLApp.MLApp
'The next line shows a list of MATLAB Automation methods
Matlab.
End Sub
```

Example — Calling MATLAB from a Web Application

This example shows you how to create a Web page that uses MATLAB as an Automation server. For another example using ASP.NET, see Technical Support solution 1–3JJZWN at `http://www.mathworks.com/support/solutions/data/1-3JJZWN.html?solution=1-3J`

You can invoke MATLAB as an Automation server from any language that supports COM, so for Web applications, you can use VBScript and JavaScript. While this example is simple, it illustrates techniques for passing commands to MATLAB and writing data to and retrieving data from the MATLAB workspace. See “Exchanging Data with the Server” on page 8-119 for related functions.

VBScript and HTML forms are combined in this example to create an interface that enables the user to select a MATLAB plot type from a pull-down menu, click a button, and create the plot in a MATLAB figure window. To accomplish this, the HTML file contains code that:

- Starts MATLAB as an Automation server via a VBScript.
- When users click a button on the HTML page, a VBScript executes that:
  - **a** Determines the type of plot selected.
  - **b** Forms a command string to create the type of plot selected.
  - **c** Forms a string describing the type of plot selected, which passes to the MATLAB base workspace in a variable.
  - **d** Executes the MATLAB command.
  - **e** Retrieves the descriptive string from the MATLAB workspace.
  - **f** Updates the text box on the HTML page.

Here is the HTML used to create this example:

```
<HTML>
<HEAD>
<TITLE>Example of calling MATLAB from VBScript</TITLE>
</HEAD>
<BODY>
<FONT FACE = "Arial, Helvetica, Geneva" SIZE = "+1" COLOR = "maroon">
Example of calling MATLAB from VBScript
</FONT>
<FONT FACE = "Arial, Helvetica, Geneva" SIZE = "-1">

<!-- %%%%%%%%%%%%%%%%%%%% BEGIN SCRIPT %%%%%%%%%%%%%%%%%%%% -->
<SCRIPT LANGUAGE="VBScript">
<!-- Invoke MATLAB as a COM Automation server upon loading page
' Initialize global variables
Dim MatLab 'COM Automation server variable
Dim MLcmd 'string to send to MATLAB for execution
' Invoke COM Automation server
Set MatLab = CreateObject("Matlab.Application")
' End initialization script -->
</SCRIPT>
```

```
<!-- %%%%%%%%%%%%%%%%%%%% END SCRIPT %%%%%%%%%%%%%%%%%%%% -->
<!-- Create form to contain controls -->
<FORM NAME="Form">
<!-- Create pulldown menu to select which plot to view -->
<P>Select type of plot:
<SELECT NAME=plot_choice>
 <OPTION SELECTED VALUE=first>Line</OPTION>
 <OPTION VALUE=second>Peaks</OPTION>
 <OPTION VALUE=third>Logo</OPTION>
</SELECT>
<!-- Create button to create plot and fill text area -->
<P>Create figure:
<INPUT TYPE="button" NAME="plot_but" VALUE="Plot">

<!-- %%%%%%%%%%%%%%%%%%%% BEGIN SCRIPT %%%%%%%%%%%%%%%%%%%% -->
<SCRIPT FOR="plot_but" EVENT="onClick" LANGUAGE="VBScript">
<!-- Start script
Dim plot_choice
Dim text_str   'string to display in text area
Dim form_var 'form object variable
Set form_var = Document.Form
plot_choice = form_var.plot_choice.value
' Condition MATLAB command to execute based on plot choice
If plot_choice = "first" Then
   MLcmd = "figure; plot(1:10);"
   text_str = "Simple line plot of 1 to 10"
   Call MatLab.PutCharArray("text","base",text_str)
Elseif plot_choice = "second" Then
   MLcmd = "figure; mesh(peaks);"
   text_str = "Mesh plot of peaks"
   Call MatLab.PutCharArray("text","base",text_str)
Elseif plot_choice = "third" Then
   MLcmd = "figure; logo;"
   text_str = "MATLAB logo"
   Call MatLab.PutCharArray("text","base",text_str)
End If
' Execute command in MATLAB
MatLab.execute(MLcmd)
' Get variable from MATLAB into VBScript
```

```
Call MatLab.GetWorkspaceData("text","base","text_str")
' Update text area
form_var.plottext.value = text_str
' End script -->
</SCRIPT>

<!-- %%%%%%%%%%%%%%%%%%%% END SCRIPT %%%%%%%%%%%%%%%%%%%% -->
<!-- Create text area to show text -->
<P><TEXTAREA NAME="plottext" ROWS="1" COLS="50"
CONTENTEDITABLE="false"></TEXTAREA>
</FONT>
</FORM>
</BODY>
</HTML>
```

Example — Calling MATLAB from a C# Client

This example creates data in the client C# program and passes it to MATLAB. The matrix (containing complex data) is then passed back to the C# program.

The reference to the MATLAB Type Library for C# is:

```
MLApp.MLAppClass matlab = new MLApp.MLAppClass();
```

Here is the complete example:

```
using System;
namespace ConsoleApplication4
{
class Class1
{
[STAThread]
static void Main(string[] args)
{
MLApp.MLAppClass matlab = new MLApp.MLAppClass();

System.Array pr = new double[4];
pr.SetValue(11,0);
pr.SetValue(12,1);
pr.SetValue(13,2);
pr.SetValue(14,3);
```

```
System.Array pi = new double[4];
pi.SetValue(1,0);
pi.SetValue(2,1);
pi.SetValue(3,2);
pi.SetValue(4,3);

matlab.PutFullMatrix("a", "base", pr, pi);

System.Array prresult = new double[4];
System.Array piresult = new double[4];

matlab.GetFullMatrix("a", "base", ref prresult, ref piresult);
}
}
}
```

9

Web Services in MATLAB

What Are Web Services in MATLAB?

In this section...

Introduction

The term *Web service* encompasses a set of XML-based technologies for making remote procedure calls over a network. The network can be a local intranet within an organization or a remote server on the other side of the globe. In short, Web services let applications running on disparate operating systems and development platforms communicate with each other.

MATLAB acts as a Web service client by sending requests to a server and handling the responses. MATLAB implements the following Web service technologies:

- Simple Object Access Protocol (SOAP)
- Web Services Description Language (WSDL)

SOAP defines a standard for making XML-based exchanges between clients and servers. The client initiates the client/server interaction, which usually takes place over HTTP. When the server receives the request, which includes the operation to be performed and any necessary parameters, it sends back a response.

Web Service Examples

The following example shows a simple HTTP-based SOAP request for retrieving the local temperature by zip code:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
    <ns1:getTemp
        xmlns:ns1="urn:xmethods-Temperature-Demo"
        soapenv:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">
      <zipcode xsi:type="xsd:string">94041</zipcode>
    </ns1:getTemp>
  </soapenv:Body>
</soapenv:Envelope>
```

The SOAP protocol defines an envelope, and inside the envelope, defines a message body. Also, inside the message body, the SOAP method `getTempRequest` is specified, as well as the `zipcode` parameter.

In the response sent by the server, notice that the SOAP message structure is similar:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soapenv:Body>
    <ns1:getTempResponse
        xmlns:ns1="urn:xmethods-Temperature-Demo"
        soapenv:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:float">68.0</return>
    </ns1:getTempResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

In the code, SOAP defines the envelope and message body as well as the response (`return`).

Most SOAP implementations use WSDL, an XML-based language, to describe and locate available services. The following example shows the message and

service definitions of the WSDL file for the temperature service from the previous examples:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="TemperatureService"
    targetNamespace=
      "http://www.xmethods.net/sd/TemperatureService.wsdl"
    xmlns:tns=
      "http://www.xmethods.net/sd/TemperatureService.wsdl"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="getTempRequest">
   <part name="zipcode" type="xsd:string"/>
  </message>
  <message name="getTempResponse">
    <part name="return" type="xsd:float"/>
  </message>
  <portType name="TemperaturePortType">
    <operation name="getTemp">
      <input message="tns:getTempRequest"/>
      <output message="tns:getTempResponse"/>
    </operation>
  </portType>
  <binding name="TemperatureBinding"
      type="tns:TemperaturePortType">
    <soap:binding style="rpc"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getTemp">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="encoded"
            namespace="urn:xmethods-Temperature-Demo"
            encodingStyle=
              "http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
            namespace="urn:xmethods-Temperature-Demo"
            encodingStyle=
```

```
            "http://schemas.xmlsoap.org/soap/encoding/"/>
        </output>
      </operation>
    </binding>
    <service name="TemperatureService">
      <documentation>Returns current temperature in a given U.S.
        zipcode</documentation>
      <port name="TemperaturePort"
          binding="tns:TemperatureBinding">
        <soap:address
          location=
          "http://services.xmethods.net:80/soap/servlet/rpcrouter"/>
      </port>
    </service>
  </definitions>
```

The code defines the request and response message actions (`getTempRequest` and `getTempResponse`) and the service name (`TemperatureService`).

Understanding Data Type Conversions

Using SOAP data types, MATLAB automatically converts XML data types to native MATLAB data types and vice versa. The following table contains the XML data type with the corresponding MATLAB data type.

XML Data Type	MATLAB Data Type
`string`	`char` array
`boolean`	`logical` scalar
`decimal`	`double` scalar
`float`	`double` scalar
`double`	`double` scalar
`duration`	`double` scalar
`time`	`double` scalar
`date`	`double` scalar
`gYearMonth`	`char` array

XML Data Type	MATLAB Data Type
gYear	char array
gMonthDay	char array
hexbinary	double array
base64Binary	double array
anyURI	char array
QName	char array

Finding More Information About Web Services

To learn more about SOAP, see the following resources:

- World Wide Web Consortium (W3C) SOAP specification
- Apache Axis Web Services
- W3 Schools SOAP Tutorial

To learn more about WSDL, see the following resources:

- W3C WSDL specification
- WSDL4J Project
- W3 Schools WSDL Tutorial

To find publicly available Web services and for more information about popular development platforms for Web services, see the following resources:

- XMethods
- Java Web Services
- Microsoft Developer Network—Web Services

Using Web Services in MATLAB

In this section...

Getting Started

In MATLAB, use the `createClassFromWsdl` function to call Web service methods. The function creates a MATLAB class based on the methods of the Web service application program interface (API).

Here is an example of the `createClassFromWsdl` function using a URL:

```
createClassFromWsdl('http://example.com/service.wsdl')
```

The example passes a URL to a WSDL file to the function. The following example uses a file path instead of a URL:

```
createClassFromWsdl('\myservicedirectory\service.wsdl')
```

The example passes a relative file path to the function. Keep in mind that the target file must contain WSDL.

Note To call remote Web services with MATLAB, you must have a working Internet connection.

Building a Simple Web Service

The following procedure walks you through the necessary steps to build a simple Web service. To begin, the procedure shows you how to find the Currency Exchange Rate Web service:

1. In a Web browser, go to the XMethods Web site at `http://www.xmethods.net/`.

2 In **XMethods Demo Services**, click the **Currency Exchange Rate** link near the bottom of the page.

3 On the Currency Exchange Rate Web page, find the WSDL URL, as well as links to analyze the WSDL. Click the **View RPC Profile** link.

In the RPC Profile page, find the available methods. In this case, the available method is `getRate`.

In addition to the method name, notice the input and output parameters and their data types. The output parameter returns a `float` data type. In "Understanding Data Type Conversions" on page 9-5, note that MATLAB converts `float` to a `double` scalar.

4 Enter the following code at the MATLAB command line to pass the WSDL URL to the function`createClassFromWsdl`, creating the `CurrencyExchangeService` class:

```
createClassFromWsdl(['http://www.xmethods.net/sd/2001' ...
                     '/CurrencyExchangeService.wsdl']);
ans =
    CurrencyExchangeService
```

Use the MATLAB `methods` function to view the methods associated with the `CurrencyExchangeService` class:

```
methods(CurrencyExchangeService)

   Methods for class CurrencyExchangeService:

   CurrencyExchangeService    getRate    display
```

5 In the current MATLAB directory, find the `@CurrencyExchangeService` folder. In the folder, you see the following files:

- `CurrencyExchangeService.m` — Contains the M-code for MATLAB object constructor
- `display.m` — Contains the M-code for a generic display method
- `getRate.m` — Contains the M-code for the `getRate` method

The `createClassFromWsdl` function automatically creates a file for each Web service method, a file for a generic display method, and a file for the Web service MATLAB object.

You can use the MATLAB `help` function to see the method signature, such as

```
help CurrencyExchangeService/getRate

getRate(obj,country1,country2)

      Input:
        country1 = (string)
        country2 = (string)

      Output:
        Result = (float)
```

6 Call the `getRate` method to obtain the exchange rate between two countries.

```
ces = CurrencyExchangeService;

getRate(ces, 'USA', 'France')
ans =
    5.1127

getRate(ces, 'Argentina', 'Chile')
ans =
    172.3052
```

To review, the `createClassFromWsdl` function performs the following actions:

- Fetches and parses the WSDL to determine the Web service API
- Creates a folder, such as `@CurrencyExchangeService`, in the current MATLAB directory
- Creates the necessary M files in the directory, such as `getRate.m`, `display.m`, and `CurrencyExchangeService.m`, based on the service API

For more information about object-oriented programming in MATLAB, see the MATLAB Programming documentation.

Building MATLAB Applications with Web Services

In this section...

Understanding Web Service Limitations

At the time of this writing, Web service technologies continue to evolve and change. The following list contains possible limitations to consider before building MATLAB applications with Web services:

- The majority of Web services are made available via HTTP. Like the Internet itself, quality of service cannot be guaranteed. Therefore, your application performance might suffer or might appear unreliable.
- Web services and the related technologies like WSDL and SOAP are relatively new. As with any new technology, established procedures and best practices are still being written.
- If you plan to call remote Web services, make sure you validate their accuracy and reliability. Also, Web services that are free today might not remain free in the future.

Programming with Web Services

Because the Internet is inherently unpredictable, make sure to take proper precautions in programming with Web services. One way to minimize the risk is to use common program control and error-handling routines.

Common programming techniques you might use include

- Try - Catch statements can catch errors that result from method calls as well as creating the MATLAB class from the WSDL. The following example shows a method call in a `try - catch` statement:

```
try
 r = getRate(CurrencyExchangeService, 'USA', 'France');
catch
 r = Nan;
 disp(lasterr);
end
```

- If statements can check that expressions or statements are true or false. The following example uses an `if` statement to cache the WSDL locally:

```
% Note: Code contains line breaks for formatting
wsdlUrl = ['http://www.xmethods.net/sd/2001' ...
           '/CurrencyExchangeService.wsdl'];
wsdlFile = 'CurrencyExchangeService.wsdl';
if ~(exist(wsdlFile,'file') == 2)
    urlwrite(wsdlUrl,wsdlFile);
end
```

- Error functions can be used to throw specific errors. The following example shows an `error` function used in an `try - catch` statement:

```
try
 r = getRate(CurrencyExchangeService, 'USA', 'France');
catch
 error('Could not return exchange rate');
end
```

For more information about program control and error-handling statements, see the MATLAB Programming documentation.

Simple M-File Example

The following M-file example provides a simple demonstration of programming with Web services. The script takes an array of country name strings and uses the Currency Exchange Rate Web service to return the rates of exchange between each pair of countries. It then runs the `max`, and `min` functions on the exchange rates:

```
% Note: Code contains line breaks for formatting
% Create a cell array with the country names:
```

```
C = { ...
   'USA', 'France'; ...
   'Argentina', 'Chile'; ...
   'Morocco', 'Portugal'; ...
   'Germany', 'Denmark'};

% Create empty array to contain exchange rate output
rates = {};

wsdlUrl = ['http://www.xmethods.net/sd/2001' ...
           '/CurrencyExchangeService.wsdl'];
wsdlFile = 'CurrencyExchangeService.wsdl';
if ~(exist(wsdlFile, 'file') == 2)
    urlwrite(wsdlUrl, wsdlFile);
end

% Catch errors during class creation
try
    % Create class from WSDL
    createClassFromWsdl(wsdlFile);
catch
    % Throw error
    error('Unable to create WSDL class');
end

ces = CurrencyExchangeService;

% Iterate through the country array
count = length(C);
for x = 1:count
    try
        % Call the getRate method
        r = getRate(ces, C{x,:});
    catch
        % Throw error
        r = NaN;
        warning(lasterr);
    end
    % Concatenate exchange rates to array
    rates = horzcat(rates, r);
```

```
end

for x = 1:count
   disp(['Exchange rate for ' C{x,1} ' and ' C{x,2} ':'])
   disp(rates{x})
end
disp(' ')

% Display highest rate
disp('Highest exchange rate:');
disp(max([rates{:}]));
disp(' ')

% Display lowest rate
disp('Lowest exchange rate:');
disp(min([rates{:}]));
```

Serial Port I/O

Disconnecting and Cleaning Up (p. 10-74)	Disconnect the serial port object from the device, and remove the object from memory and from the workspace
Property Reference (p. 10-76)	Properties grouped by category
Properties — Alphabetical List (p. 10-80)	

Introduction

In this section...

What Is the MATLAB Serial Port Interface?

The MATLAB serial port interface provides direct access to peripheral devices such as modems, printers, and scientific instruments that you connect to your computer's serial port. This interface is established through a serial port object. The serial port object supports functions and properties that allow you to

- Configure serial port communications
- Use serial port control pins
- Write and read data
- Use events and callbacks
- Record information to disk

Additional serial port functionality is available by using Instrument Control Toolbox. In addition to command-line access, Instrument Control Toolbox provides a graphical tool called the Test & Measurement Tool, which allows you to communicate with, configure, and transfer data with your serial device without writing code. The Test & Measurement Tool generates MATLAB code for your serial device that you can later reuse to communicate with your device or to develop GUI-based applications. Instrument Control Toolbox also includes additional serial I/O utility functions that facilitate object creation and configuration, instrument communication, and so on. Instrument Control Toolbox also lets you communicate with GPIB- or VISA-compatible instruments.

If you want to communicate with PC-compatible data acquisition hardware such as multifunction I/O boards, you need Data Acquisition Toolbox.

For more information about these products, visit the MathWorks Web site at `http://www.mathworks.com/products`.

Supported Serial Port Interface Standards

Over the years, several serial port interface standards have been developed. These standards include RS-232, RS-422, and RS-485 - all of which are supported by the MATLAB serial port object. Of these, the most widely used interface standard for connecting computers to peripheral devices is RS-232.

This guide assume you are using the RS-232 standard, which is discussed in "Overview of the Serial Port" on page 10-5. Refer to your computer and device documentation to see which interface standard you can use.

Supported Platforms

The MATLAB serial port interface is supported on Microsoft Windows 32-bit, Linux 32-bit, and Sun Solaris 64-bit platforms.

Using the Examples with Your Device

Many of the examples in this section reflect specific peripheral devices connected to a PC serial port — in particular a Tektronix TDS 210 two-channel oscilloscope connected to the COM1 port. Therefore, many of the string commands are specific to this instrument.

If your peripheral device is connected to a different serial port, or if it accepts different commands, modify the examples accordingly.

Overview of the Serial Port

In this section...

Introduction

For many serial port applications, you can communicate with your device without detailed knowledge of how the serial port works. If your application is straightforward, or if you are already familiar with the topics mentioned above, you might want to begin with "The Serial Port Session" on page 10-19 to see how to use your serial port device with MATLAB.

What Is Serial Communication?

Serial communication is the most common low-level protocol for communicating between two or more devices. Normally, one device is a computer, while the other device can be a modem, a printer, another computer, or a scientific instrument such as an oscilloscope or a function generator.

As the name suggests, the serial port sends and receives bytes of information in a serial fashion — one bit at a time. These bytes are transmitted using either a binary (numerical) format or a text format.

The Serial Port Interface Standard

The serial port interface for connecting two devices is specified by the TIA/EIA-232C standard published by the Telecommunications Industry Association.

The original serial port interface standard was given by RS-232, which stands for Recommended Standard number 232. The term *RS-232* is still in popular use, and is used in this guide when referring to a serial communication port that follows the TIA/EIA-232 standard. RS-232 defines these serial port characteristics:

- The maximum bit transfer rate and cable length
- The names, electrical characteristics, and functions of signals
- The mechanical connections and pin assignments

Primary communication is accomplished using three pins: the Transmit Data pin, the Receive Data pin, and the Ground pin. Other pins are available for data flow control, but are not required.

Other standards such as RS-485 define additional functionality such as higher bit transfer rates, longer cable lengths, and connections to as many as 256 devices.

Connecting Two Devices with a Serial Cable

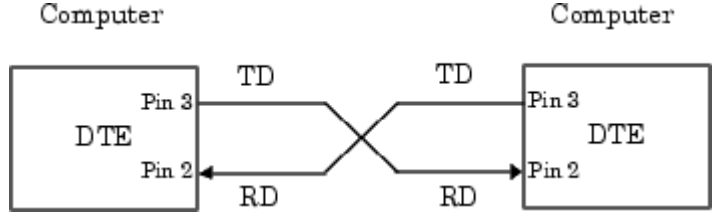
The RS-232 standard defines the two devices connected with a serial cable as the Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE). This terminology reflects the RS-232 origin as a standard for communication between a computer terminal and a modem.

Throughout this guide, your computer is considered a DTE, while peripheral devices such as modems and printers are considered DCE's. Many scientific instruments function as DTEs.

Because RS-232 mainly involves connecting a DTE to a DCE, the pin assignments are defined such that straight-through cabling is used, where pin 1 is connected to pin 1, pin 2 is connected to pin 2, and so on. A DTE to DCE serial connection using the transmit data (TD) pin and the receive data (RD) pin is shown below.

Refer to “Serial Port Signals and Pin Assignments” on page 10-7 for more information about serial port pins.

If you connect two DTEs or two DCEs using a straight serial cable, the TD pins on each device are connected to each other, and the RD pins on each device are connected to each other. Therefore, to connect two like devices, you must use a *null modem* cable. As shown below, null modem cables cross the transmit and receive lines in the cable.

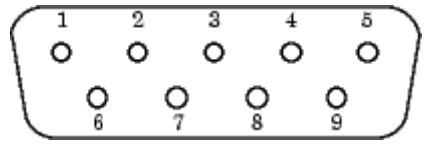


Note You can connect multiple RS-422 or RS-485 devices to a serial port. If you have an RS-232/RS-485 adaptor, you can use the MATLAB serial port object with these devices.

Serial Port Signals and Pin Assignments

Serial ports consist of two signal types: data signals and control signals. To support these signal types, as well as the signal ground, the RS-232 standard defines a 25-pin connection. However, most PCs and UNIX platforms use a 9-pin connection. In fact, only three pins are required for serial port communications: one for receiving data, one for transmitting data, and one for the signal ground.

The pin assignment scheme for a 9-pin male connector on a DTE is shown below.



The pins and signals associated with the 9-pin connector are described below. Refer to the RS-232 standard for a description of the signals and pin assignments used for a 25-pin connector.

Serial Port Pin and Signal Assignments

Pin	Label	Signal Name	Signal Type
1	CD	Carrier Detect	Control
2	RD	Received Data	Data
3	TD	Transmitted Data	Data
4	DTR	Data Terminal Ready	Control
5	GND	Signal Ground	Ground
6	DSR	Data Set Ready	Control
7	RTS	Request to Send	Control
8	CTS	Clear to Send	Control
9	RI	Ring Indicator	Control

The term *data set* is synonymous with *modem* or *device*, while the term *data terminal* is synonymous with *computer*.

Note The serial port pin and signal assignments are with respect to the DTE. For example, data is transmitted from the TD pin of the DTE to the RD pin of the DCE.

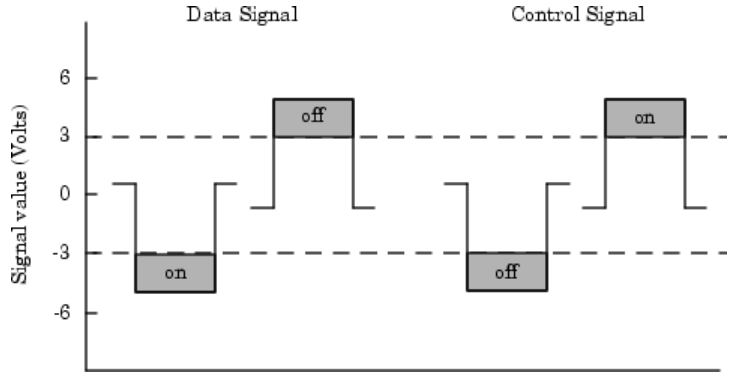
Signal States

Signals can be in either an *active* state or an *inactive* state. An active state corresponds to the binary value 1, while an inactive state corresponds to the binary value 0. An active signal state is often described as *logic 1*, *on*, *true*, or a *mark*. An inactive signal state is often described as *logic 0*, *off*, *false*, or a *space*.

For data signals, the *on* state occurs when the received signal voltage is more negative than -3 volts, while the *off* state occurs for voltages more positive than 3 volts. For control signals, the *on* state occurs when the received signal voltage is more positive than 3 volts, while the *off* state occurs for voltages more negative than -3 volts. The voltage between -3 volts and +3 volts is considered a transition region, and the signal state is undefined.

To bring the signal to the *on* state, the controlling device *unasserts* (or *lowers*) the value for data pins and *asserts* (or *raises*) the value for control pins. Conversely, to bring the signal to the *off* state, the controlling device asserts the value for data pins and unasserts the value for control pins.

The *on* and *off* states for a data signal and for a control signal are shown below.



The Data Pins

Most serial port devices support *full-duplex* communication meaning that they can send and receive data at the same time. Therefore, separate pins are used for transmitting and receiving data. For these devices, the TD, RD, and GND pins are used. However, some types of serial port devices support

only one-way or *half-duplex* communications. For these devices, only the TD and GND pins are used. This guide assumes that a full-duplex serial port is connected to your device.

The TD pin carries data transmitted by a DTE to a DCE. The RD pin carries data that is received by a DTE from a DCE.

The Control Pins

The control pins of a 9-pin serial port are used to determine the presence of connected devices and control the flow of data. The control pins include

- "The RTS and CTS Pins" on page 10-10
- "The DTR and DSR Pins" on page 10-10
- "The CD and RI Pins" on page 10-11

The RTS and CTS Pins. The RTS and CTS pins are used to signal whether the devices are ready to send or receive data. This type of data flow control—called *hardware handshaking*—is used to prevent data loss during transmission. When enabled for both the DTE and DCE, hardware handshaking using RTS and CTS follows these steps:

1. The DTE asserts the RTS pin to instruct the DCE that it is ready to receive data.
2. The DCE asserts the CTS pin indicating that it is clear to send data over the TD pin. If data can no longer be sent, the CTS pin is unasserted.
3. The data is transmitted to the DTE over the TD pin. If data can no longer be accepted, the RTS pin is unasserted by the DTE and the data transmission is stopped.

To enable hardware handshaking in MATLAB, refer to "Controlling the Flow of Data: Handshaking" on page 10-63.

The DTR and DSR Pins. Many devices use the DSR and DTR pins to signal if they are connected and powered. Signaling the presence of connected devices using DTR and DSR follows these steps:

1. The DTE asserts the DTR pin to request that the DCE connect to the communication line.
2. The DCE asserts the DSR pin to indicate it is connected.
3. DCE unasserts the DSR pin when it is disconnected from the communication line.

The DTR and DSR pins were originally designed to provide an alternative method of hardware handshaking. However, the RTS and CTS pins are usually used in this way, and not the DSR and DTR pins. Refer to your device documentation to determine its specific pin behavior.

The CD and RI Pins. The CD and RI pins are typically used to indicate the presence of certain signals during modem-modem connections.

CD is used by a modem to signal that it has made a connection with another modem, or has detected a carrier tone. CD is asserted when the DCE is receiving a signal of a suitable frequency. CD is unasserted if the DCE is not receiving a suitable signal.

RI is used to indicate the presence of an audible ringing signal. RI is asserted when the DCE is receiving a ringing signal. RI is unasserted when the DCE is not receiving a ringing signal (e.g., it is between rings).

Serial Data Format

The serial data format includes one start bit, between five and eight data bits, and one stop bit. A parity bit and an additional stop bit might be included in the format as well. The diagram below illustrates the serial data format.

The format for serial port data is often expressed using the following notation

number of data bits - parity type - number of stop bits

For example, 8-N-1 is interpreted as eight data bits, no parity bit, and one stop bit, while 7-E-2 is interpreted as seven data bits, even parity, and two stop bits.

The data bits are often referred to as a *character* because these bits usually represent an ASCII character. The remaining bits are called *framing bits* because they frame the data bits.

Bytes Versus Values

The collection of bits that comprise the serial data format is called a *byte*. At first, this term might seem inaccurate because a byte is 8 bits and the serial data format can range between 7 bits and 12 bits. However, when serial data is stored on your computer, the framing bits are stripped away, and only the data bits are retained. Moreover, eight data bits are always used regardless of the number of data bits specified for transmission, with the unused bits assigned a value of 0.

When reading or writing data, you might need to specify a *value*, which can consist of one or more bytes. For example, if you read one value from a device using the `int32` format, that value consists of four bytes. For more information about reading and writing values, refer to "Writing and Reading Data" on page 10-32.

Synchronous and Asynchronous Communication

The RS-232 standard supports two types of communication protocols: synchronous and asynchronous.

Using the synchronous protocol, all transmitted bits are synchronized to a common clock signal. The two devices initially synchronize themselves to each other, and continually send characters to stay synchronized. Even when actual data is not really being sent, a constant flow of bits allows each device to know where the other is at any given time. That is, each bit that is sent is either actual data or an idle character. Synchronous communications allows faster data transfer rates than asynchronous methods, because additional bits to mark the beginning and end of each data byte are not required.

Using the asynchronous protocol, each device uses its own internal clock, resulting in bytes that are transferred at arbitrary times. So, instead of using time as a way to synchronize the bits, the data format is used.

In particular, the data transmission is synchronized using the start bit of the word, while one or more stop bits indicate the end of the word. The requirement to send these additional bits causes asynchronous communications to be slightly slower than synchronous. However, it has the advantage that the processor does not have to deal with the additional idle characters. Most serial ports operate asynchronously.

Note When used in this guide, the terms *synchronous* and *asynchronous* refer to whether read or write operations block access to the MATLAB command line. Refer to "Controlling Access to the MATLAB Command Line" on page 10-33 for more information.

How Are the Bits Transmitted?

By definition, serial data is transmitted one bit at a time. The order in which the bits are transmitted is as follows

1. The start bit is transmitted with a value of 0.
2. The data bits are transmitted. The first data bit corresponds to the least significant bit (LSB), while the last data bit corresponds to the most significant bit (MSB).
3. The parity bit (if defined) is transmitted.
4. One or two stop bits are transmitted, each with a value of 1.

The number of bits transferred per second is given by the *baud rate*. The transferred bits include the start bit, the data bits, the parity bit (if defined), and the stop bits.

Start and Stop Bits

As described in "Synchronous and Asynchronous Communication" on page 10-12, most serial ports operate asynchronously. This means that the transmitted byte must be identified by start and stop bits. The start bit indicates when the data byte is about to begin; the stop bit(s) indicate(s) when the data byte has been transferred. The process of identifying bytes with the serial data format follows these steps:

1. When a serial port pin is idle (not transmitting data), it is in an *on* state.
2. When data is about to be transmitted, the serial port pin switches to an *off* state due to the start bit.
3. The serial port pin switches back to an *on* state due to the stop bit(s). This indicates the end of the byte.

Data Bits

The data bits transferred through a serial port might represent device commands, sensor readings, error messages, and so on. The data can be transferred as either binary data or ASCII data.

Most serial ports use between five and eight data bits. Binary data is typically transmitted as eight bits. Text-based data is transmitted as either seven bits or eight bits. If the data is based on the ASCII character set, a minimum of seven bits is required because there are 2^7 or 128 distinct characters. If an eighth bit is used, it must have a value of 0. If the data is based on the extended ASCII character set, eight bits must be used because there are 2^8 or 256 distinct characters.

The Parity Bit

The parity bit provides simple error (parity) checking for the transmitted data. The types of parity checking are shown below.

Parity Types

Parity Type	Description
Even	The data bits plus the parity bit result in an even number of 1s.
Mark	The parity bit is always 1.

Parity Types (Continued)

Parity Type	Description
Odd	The data bits plus the parity bit result in an odd number of 1s.
Space	The parity bit is always 0.

Mark and space parity checking are seldom used because they offer minimal error detection. You might choose to not use parity checking at all.

The parity checking process follows these steps:

1. The transmitting device sets the parity bit to 0 or to 1, depending on the data bit values and the type of parity-checking selected.
2. The receiving device checks if the parity bit is consistent with the transmitted data. If it is, the data bits are accepted. If it is not, an error is returned.

Note Parity checking can detect only 1-bit errors. Multiple-bit errors can appear as valid data.

For example, suppose the data bits 01110001 are transmitted to your computer. If even parity is selected, the parity bit is set to 0 by the transmitting device to produce an even number of 1s. If odd parity is selected, the parity bit is set to 1 by the transmitting device to produce an odd number of 1s.

Finding Serial Port Information for Your Platform

This section describes the ways to find serial port information for Windows and UNIX platforms.

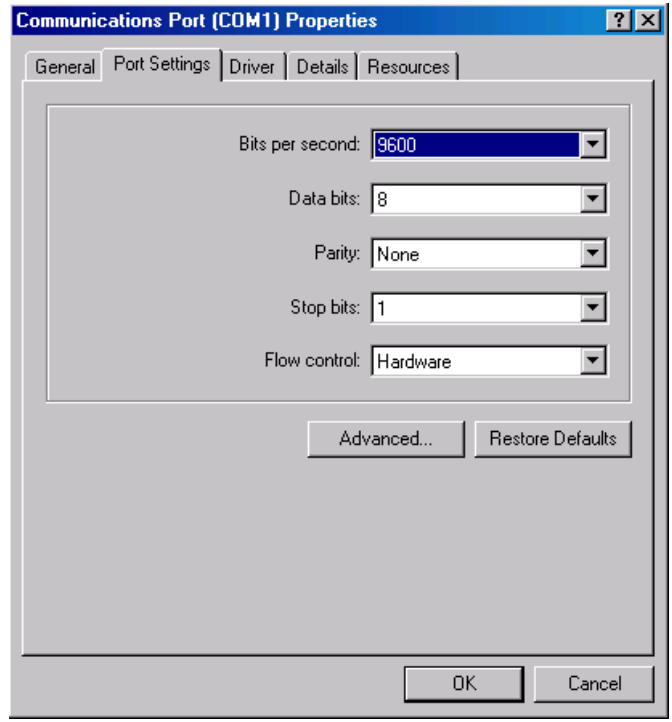
Note Your operating system provides default values for all serial port settings. However, these settings are overridden by your MATLAB code, and will have no effect on your serial port application.

Windows Platform

You can access serial port information through the **System Properties** dialog. To access this in Windows XP,

1. Right-click **My Computer** on the desktop, and select **Properties**.
2. In the **System Properties** dialog, click the **Hardware** tab.
3. Click **Device Manager**.
4. In the **Device Manager** dialog, expand the Ports node.
5. Double-click the `Communications Port (COM1)` node.
6. Select the **Port Settings** tab.

The resulting Ports dialog box is shown below.



UNIX Platform

To find serial port information for UNIX platforms, you need to know the serial port names. These names might vary between different operating systems.

On Linux, serial port devices are typically named `ttyS0`, `ttyS1`, etc. Use the `setserial` command to display or configure serial port information. For example, to display which ports are available

```
setserial -bg /dev/ttyS*
/dev/ttyS0 at 0x03f8 (irq = 4) is a 16550A
/dev/ttyS1 at 0x02f8 (irq = 3) is a 16550A
```

To display detailed information about `ttyS0`

```
setserial -ag /dev/ttyS0
/dev/ttyS0, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
        Baud_base: 115200, close_delay: 50, divisor: 0
        closing_wait: 3000, closing_wait2: infinte
        Flags: spd_normal skip_test session_lockout
```

Note If the `setserial -ag` command does not work, make sure that you have read and write permission for the port.

For all supported UNIX platforms, use the `stty` command to display or configure serial port information. For example, to display serial port properties for `ttyS0`, enter

```
stty -a < /dev/ttyS0
```

To configure the baud rate to 4800 bits per second, enter

```
stty speed 4800 < /dev/ttyS0 > /dev/ttyS0
```

Selected Bibliography

[1] TIA/EIA-232-F, *Interface Between Data Terminal Equipment and Data Circuit-Terminating Equipment Employing Serial Binary Data Interchange*.

[2] Jan Axelson, *Serial Port Complete*, Lakeview Research, Madison, WI, 1998.

[3] *Instrument Communication Handbook*, IOTech, Inc., Cleveland, OH, 1991.

[4] *TDS 200-Series Two Channel Digital Oscilloscope Programmer Manual,* Tektronix, Inc., Wilsonville, OR.

[5] *Courier High Speed Modems User's Manual*, U.S. Robotics, Inc., Skokie, IL, 1994.

Getting Started with Serial I/O

In this section...

Example: Getting Started

This example illustrates some basic serial port commands.

If you have a device connected to the serial port COM1 and configured for a baud rate of 4800, execute the following example.

```
s = serial('COM1');
set(s,'BaudRate',4800);
fopen(s);
fprintf(s,'*IDN?')
out = fscanf(s);
fclose(s)
delete(s)
clear s
```

The `*IDN?` command queries the device for identification information, which is returned to `out`. If your device does not support this command, or if it is connected to a different serial port, modify the above example accordingly.

Note `*IDN?` is one of the commands supported by the Standard Commands for Programmable Instruments (SCPI) language, which is used by many modern devices. Refer to your device documentation to see if it supports the SCPI language.

The Serial Port Session

This example describes the steps you use to perform any serial port task from beginning to end.

The serial port *session* comprises all the steps you are likely to take when communicating with a device connected to a serial port. These steps are:

1. Create a serial port object — Create a serial port object for a specific serial port using the `serial` creation function.

   Configure properties during object creation if necessary. In particular, you might want to configure properties associated with serial port communications such as the baud rate, the number of data bits, and so on.

2. Connect to the device — Connect the serial port object to the device using the `fopen` function.

   After the object is connected, alter the necessary device settings by configuring property values, read data, and write data.

3. Configure properties — To establish the desired serial port object behavior, assign values to properties using the `set` function or dot notation.

   In practice, you can configure many of the properties at any time including during, or just after, object creation. Conversely, depending on your device settings and the requirements of your serial port application, you might be able to accept the default property values and skip this step.

4. Write and read data — Write data to the device using the `fprintf` or `fwrite` function, and read data from the device using the `fgetl`, `fgets`, `fread`, `fscanf`, or `readasync` function.

   The serial port object behaves according to the previously configured or default property values.

5. Disconnect and clean up — When you no longer need the serial port object, disconnect it from the device using the `fclose` function, remove it from memory using the `delete` function, and remove it from the MATLAB workspace using the `clear` command.

The serial port session is reinforced in many of the serial port documentation examples. Refer to “Example: Getting Started” on page 10-19 to see a basic example that uses the steps shown above.

Configuring and Returning Properties

This example describes how you display serial port property names and property values, and how you assign values to properties.

You establish the desired serial port object behavior by configuring property values. You can display or configure property values using the `set` function, the `get` function, or dot notation.

Displaying Property Names and Property Values

Once the serial port object is created, use the `set` function to display all the configurable properties to the command line. Additionally, if a property has a finite set of string values, `set` also displays these values.

```
s = serial('COM1');
set(s)
    ByteOrder: [ {littleEndian} | bigEndian ]
    BytesAvailableFcn
    BytesAvailableFcnCount
    BytesAvailableFcnMode: [ {terminator} | byte ]
    ErrorFcn
    InputBufferSize
    Name
    OutputBufferSize
    OutputEmptyFcn
    RecordDetail: [ {compact} | verbose ]
    RecordMode: [ {overwrite} | append | index ]
    RecordName
    Tag
    Timeout
    TimerFcn
    TimerPeriod
    UserData

    SERIAL specific properties:
    BaudRate
    BreakInterruptFcn
    DataBits
    DataTerminalReady: [ {on} | off ]
    FlowControl: [ {none} | hardware | software ]
```

```
    Parity: [ {none} | odd | even | mark | space ]
    PinStatusFcn
    Port
    ReadAsyncMode: [ {continuous} | manual ]
    RequestToSend: [ {on} | off ]
    StopBits
    Terminator
```

Use the `get` function to display one or more properties and their current values to the command line. To display all properties and their current values

```
get(s)
    ByteOrder = littleEndian
    BytesAvailable = 0
    BytesAvailableFcn =
    BytesAvailableFcnCount = 48
    BytesAvailableFcnMode = terminator
    BytesToOutput = 0
    ErrorFcn =
    InputBufferSize = 512
    Name = Serial-COM1
    OutputBufferSize = 512
    OutputEmptyFcn =
    RecordDetail = compact
    RecordMode = overwrite
    RecordName = record.txt
    RecordStatus = off
    Status = closed
    Tag =
    Timeout = 10
    TimerFcn =
    TimerPeriod = 1
    TransferStatus = idle
    Type = serial
    UserData = []
    ValuesReceived = 0
    ValuesSent = 0

    SERIAL specific properties:
    BaudRate = 9600
```

```
    BreakInterruptFcn =
    DataBits = 8
    DataTerminalReady = on
    FlowControl = none
    Parity = none
    PinStatus = [1x1 struct]
    PinStatusFcn =
    Port = COM1
    ReadAsyncMode = continuous
    RequestToSend = on
    StopBits = 1
    Terminator = LF
```

To display the current value for one property, supply the property name to `get`.

```
get(s,'OutputBufferSize')
ans =
   512
```

To display the current values for multiple properties, include the property names as elements of a cell array.

```
get(s,{'Parity','TransferStatus'})
ans =
    'none'    'idle'
```

Use the dot notation to display a single property value.

```
s.Parity
ans =
none
```

Configuring Property Values

You can configure property values using the `set` function

```
set(s,'BaudRate',4800);
```

or the dot notation

```
s.BaudRate = 4800;
```

To configure values for multiple properties, supply multiple property name/property value pairs to `set`.

```
set(s,'DataBits',7,'Name','Test1-serial')
```

Note that you can configure only one property value at a time using the dot notation.

In practice, you can configure many of the properties at any time while the serial port object exists — including during object creation. However, some properties are not configurable while the object is connected to the device or when recording information to disk. Refer to “Property Reference” on page 10-76 for information about when a property is configurable.

Specifying Property Names

Serial port property names are presented using mixed case. While this makes property names easier to read, use any case you want when specifying property names. Additionally, you need use only enough letters to identify the property name uniquely, so you can abbreviate most property names. For example, you can configure the `BaudRate` property any of these ways:

```
set(s,'BaudRate',4800)
set(s,'baudrate',4800)
set(s,'BAUD',4800)
```

When you include property names in an M-file, you should use the full property name. This practice can prevent problems with future releases of MATLAB if a shortened name is no longer unique because of the addition of new properties.

Default Property Values

Whenever you do not explicitly define a value for a property, the default value is used. All configurable properties have default values.

Note Your operating system provides default values for all serial port settings such as the baud rate. However, these settings are overridden by your MATLAB code and have no effect on your serial port application.

If a property has a finite set of string values, the default value is enclosed by `{}`. For example, the default value for the `Parity` property is `none`.

```
set(s,'Parity')
[ {none} | odd | even | mark | space ]
```

You can find the default value for any property in the property reference pages.

Creating a Serial Port Object

In this section...

Overview of a Serial Port Object

The `serial` function requires the name of the serial port connected to your device as an input argument. Additionally, you can configure property values during object creation. For example, to create a serial port object associated with the serial port COM1, enter

```
s = serial('COM1');
```

The serial port object `s` now exists in the MATLAB workspace. You can display the class of `s` with the `whos` command.

```
whos s
  Name      Size         Bytes  Class

  s         1x1            512  serial object

Grand total is 11 elements using 512 bytes
```

Once the serial port object is created, the properties listed below are automatically assigned values. These general-purpose properties provide descriptive information about the serial port object based on the object type and the serial port.

Descriptive General Purpose Properties

Property Name	Description
`Name`	Specify a descriptive name for the serial port object

Descriptive General Purpose Properties (Continued)

Property Name	Description
Port	Indicate the platform-specific serial port name
Type	Indicate the object type

Display the values of these properties for s with the get function.

```
get(s,{'Name','Port','Type'})
ans =
    'Serial-COM1'    'COM1'    'serial'
```

Configuring Properties During Object Creation

You can configure serial port properties during object creation. serial accepts property names and property values in the same format as the set function. For example, you can specify property name/property value pairs.

```
s = serial('COM1','BaudRate',4800,'Parity','even');
```

If you specify an invalid property name, the object is not created. However, if you specify an invalid value for some properties (for example, BaudRate is set to 50), the object might be created but you are not informed of the invalid value until you connect the object to the device with the fopen function.

The Serial Port Object Display

The serial port object provides you with a convenient display that summarizes important configuration and state information. You can invoke the display summary these three ways:

- Type the serial port object variable name at the command line.
- Exclude the semicolon when creating a serial port object.
- Exclude the semicolon when configuring properties using the dot notation.

The display summary for the serial port object s is as follows:

```
Serial Port Object : Serial-COM1
```

```
Communication Settings
   Port:               COM1
   BaudRate:           9600
   Terminator:         'LF'

Communication State
   Status:             closed
   RecordStatus:       off

Read/Write State
   TransferStatus:     idle
   BytesAvailable:     0
   ValuesReceived:     0
   ValuesSent:         0
```

Creating an Array of Serial Port Objects

In MATLAB, you can create an array from existing variables by concatenating those variables together. The same is true for serial port objects. For example, suppose you create the serial port objects `s1` and `s2`.

```
s1 = serial('COM1');
s2 = serial('COM2');
```

You can now create a serial port object array consisting of `s1` and `s2` using the usual MATLAB syntax. To create the row array `x`, enter

```
x = [s1 s2]

Instrument Object Array

   Index:   Type:        Status:      Name:
   1        serial       closed       Serial-COM1
   2        serial       closed       Serial-COM2
```

To create the column array `y`, enter

```
y = [s1;s2];
```

Note that you cannot create a matrix of serial port objects. For example, you cannot create the matrix

```
z = [s1 s2;s1 s2];
??? Error using ==> serial/vertcat
Only a row or column vector of instrument objects can be created.
```

Depending on your application, you might want to pass an array of serial port objects to a function. For example, to configure the baud rate and parity for `s1` and `s2` using one call to `set`:

```
set(x,'BaudRate',19200,'Parity','even')
```

Refer to the Serial Port Devices functional reference to see which functions accept a serial port object array as an input.

Connecting to the Device

Before you can use the serial port object to write or read data, you must connect it to your device via the serial port specified in the `serial` function. You connect a serial port object to the device with the `fopen` function.

```
fopen(s)
```

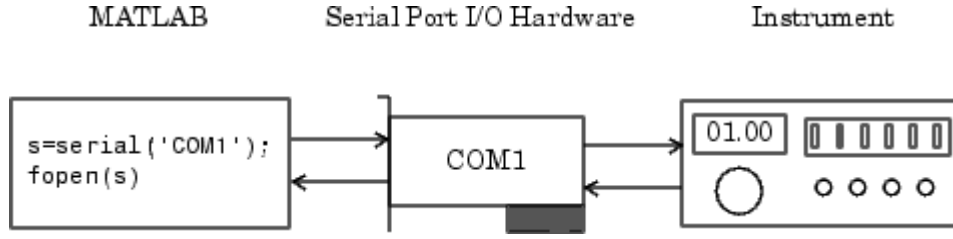
Some properties are read only while the serial port object is connected and must be configured before using `fopen`. Examples include the `InputBufferSize` and the `OutputBufferSize` properties. Refer to "Property Reference" on page 10-76 to determine when you can configure a property.

Note You can create any number of serial port objects, but you can connect only one serial port object per MATLAB session to a given serial port at a time. However, the serial port is not locked by MATLAB, so other applications or other instances of MATLAB can access the same serial port, which could result in a conflict, with unpredictable results.

You can examine the `Status` property to verify that the serial port object is connected to the device.

```
s.Status
ans =
open
```

As illustrated below, the connection between the serial port object and the device is complete; data is readable and writable.



Configuring Communication Settings

Before you can write or read data, both the serial port object and the device must have identical communication settings. Configuring serial port communications involves specifying values for properties that control the baud rate and the serial data format. The properties described below.

Communication Properties

Property Name	Description
`BaudRate`	Specify the rate at which bits are transmitted
`DataBits`	Specify the number of data bits to transmit
`Parity`	Specify the type of parity checking
`StopBits`	Specify the number of bits used to indicate the end of a byte
`Terminator`	Specify the terminator character

Note If the serial port object and the device communication settings are not identical, data is not readable or writable.

Refer to the device documentation for an explanation of its supported communication settings.

Writing and Reading Data

In this section...

Before You Begin

For many serial port applications, there are three important questions that you should consider when writing or reading data:

- Will the read or write function block access to the MATLAB command line?
- Is the data to be transferred binary (numerical) or text?
- Under what conditions will the read or write operation complete?

For write operations, these questions are answered in "Writing Data" on page 10-34. For read operations, these questions are answered in "Reading Data" on page 10-39.

Example — Introduction to Writing and Reading Data

Suppose you want to return identification information for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1. This requires writing the `*IDN?` command to the instrument using the `fprintf` function, and reading back the result of that command using the `fscanf` function.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
```

```
out = fscanf(s)
```

The resulting identification information is as follows.

```
out =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

End the serial port session.

```
fclose(s)
delete(s)
clear s
```

Controlling Access to the MATLAB Command Line

You control access to the MATLAB command line by specifying whether a read or write operation is *synchronous* or *asynchronous*.

A synchronous operation blocks access to the command line until the read or write function completes execution. An asynchronous operation does not block access to the command line, and you can issue additional commands while the read or write function executes in the background.

The terms *synchronous* and *asynchronous* are often used to describe how the serial port operates at the hardware level. The RS-232 standard supports an asynchronous communication protocol. Using this protocol, each device uses its own internal clock. The data transmission is synchronized using the start bit of the bytes, while one or more stop bits indicate the end of the byte. Refer to “Serial Data Format” on page 10-11 for more information on start bits and stop bits. The RS-232 standard also supports a synchronous mode where all transmitted bits are synchronized to a common clock signal.

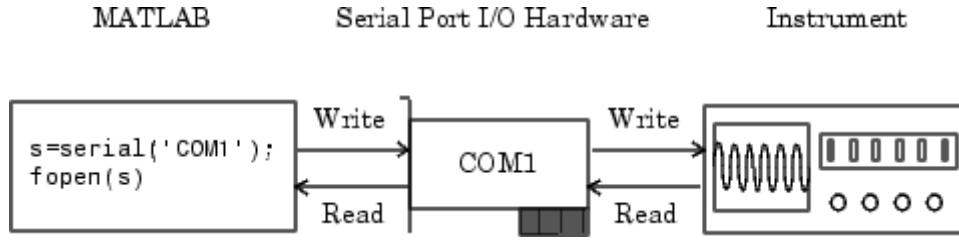
At the hardware level, most serial ports operate asynchronously. However, using the default behavior for many of the read and write functions, you can mimic the operation of a synchronous serial port.

Note When used in this guide, the terms *synchronous* and *asynchronous* refer to whether read or write operations block access to the MATLAB command-line. In other words, these terms describe how the software behaves, and not how the hardware behaves.

The two main advantages of writing or reading data asynchronously are:

- You can issue another command while the write or read function is executing.
- You can use all supported callback properties (see "Events and Callbacks" on page 10-51).

For example, because serial ports have separate read and write pins, you can simultaneously read and write data. This is illustrated below.



Writing Data

This section describes writing data to your serial port device in three parts:

- "The Output Buffer and Data Flow" on page 10-35 describes the flow of data from MATLAB to the device.
- "Writing Text Data" on page 10-37 describes how to write text data (string commands) to the device.
- "Writing Binary Data" on page 10-39 describes how to write binary (numerical) data to the device.

The functions associated with writing data are shown below.

Functions Associated with Writing Data

Function Name	Description
`fprintf`	Write text to the device
`fwrite`	Write binary data to the device
`stopasync`	Stop asynchronous read and write operations

The properties associated with writing data are shown below.

Properties Associated with Writing Data

Property Name	Description
`BytesToOutput`	Indicate the number of bytes currently in the output buffer
`OutputBufferSize`	Specify the size of the output buffer in bytes
`Timeout`	Specify the waiting time to complete a read or write operation
`TransferStatus`	Indicate if an asynchronous read or write operation is in progress
`ValuesSent`	Indicate the total number of values written to the device

The Output Buffer and Data Flow

The output buffer is computer memory allocated by the serial port object to store data that is to be written to the device. When writing data to your device, the data flow follows these two steps:

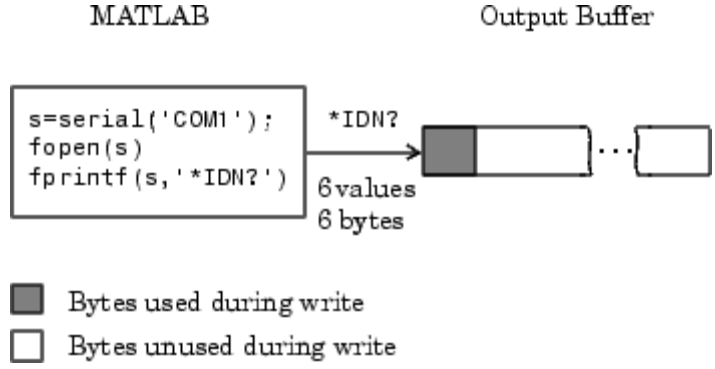
1. The data specified by the write function is sent to the output buffer.
2. The data in the output buffer is sent to the device.

The `OutputBufferSize` property specifies the maximum number of bytes that you can store in the output buffer. The `BytesToOutput` property indicates the number of bytes currently in the output buffer. The default values for these properties are shown below.

```
s = serial('COM1');
get(s,{'OutputBufferSize','BytesToOutput'})
ans =
    [512]    [0]
```

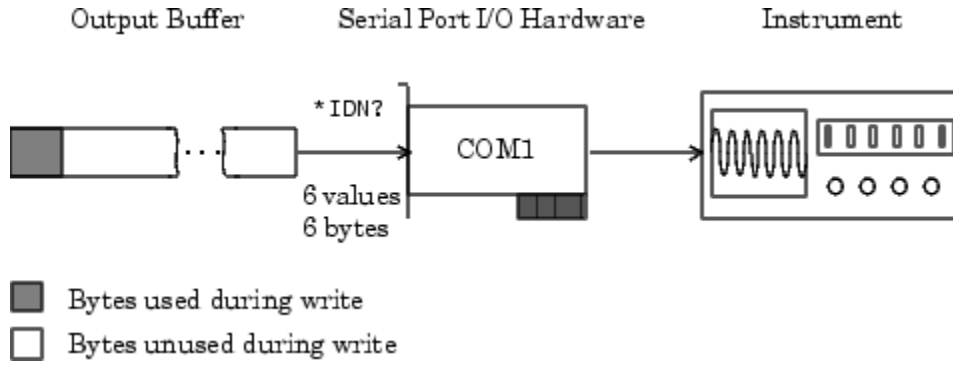
If you attempt to write more data than can fit in the output buffer, an error is returned and no data is written.

For example, suppose you write the string command `*IDN?` to the TDS 210 oscilloscope using the `fprintf` function. As shown below, the string is first written to the output buffer as six values.



The `*IDN?` command consists of six values because the terminator is automatically written. Moreover, the default data format for the `fprintf` function specifies that one value corresponds to one byte. For more information about bytes and values, refer to “Bytes Versus Values” on page 10-12. `fprintf` and the terminator are discussed in “Writing Text Data” on page 10-37.

As shown below, after the string is written to the output buffer, it is then written to the device via the serial port.



Writing Text Data

You use the `fprintf` function to write text data to the device. For many devices, writing text data means writing string commands that change device settings, prepare the device to return data or status information, and so on.

For example, the `Display:Contrast` command changes the display contrast of the oscilloscope.

```
s = serial('COM1');
fopen(s)
fprintf(s,'Display:Contrast 45')
```

By default, `fprintf` writes data using the `%s\n` format because many serial port devices accept only text-based commands. However, you can specify many other formats, as described in the `fprintf` reference pages.

You can verify the number of values sent to the device with the `ValuesSent` property.

```
s.ValuesSent
ans =
    20
```

Note that the `ValuesSent` property value includes the terminator because each occurrence of `\n` in the command sent to the device is replaced with the `Terminator` property value.

```
s.Terminator
```

```
ans =
LF
```

The default value of `Terminator` is the linefeed character. The terminator required by your device will be described in its documentation.

Synchronous Versus Asynchronous Write Operations. By default, `fprintf` operates synchronously and blocks the MATLAB command line until execution completes. To write text data asynchronously to the device, you must specify `async` as the last input argument to `fprintf`.

```
fprintf(s,'Display:Contrast 45','async')
```

Asynchronous operations do not block access to the MATLAB command line. Additionally, while an asynchronous write operation is in progress, you can:

- Execute an asynchronous read operation because serial ports have separate pins for reading and writing
- Make use of all supported callback properties

You can determine which asynchronous operations are in progress with the `TransferStatus` property. If no asynchronous operations are in progress, `TransferStatus` is `idle`.

```
s.TransferStatus
ans =
idle
```

Rules for Completing a Write Operation with fprintf. A synchronous or asynchronous write operation using `fprintf` completes when:

- The specified data is written.
- The time specified by the `Timeout` property passes.

Additionally, you can stop an asynchronous write operation with the `stopasync` function.

Writing Binary Data

You use the `fwrite` function to write binary data to the device. Writing binary data means writing numerical values. A typical application for writing binary data involves writing calibration data to an instrument such as an arbitrary waveform generator.

Note Some serial port devices accept only text-based commands. These commands might use the SCPI language or some other vendor-specific language. Therefore, you might need to use the `fprintf` function for all write operations.

By default, `fwrite` translates values using the `uchar` precision. However, you can specify many other precisions as described in the reference pages for this function.

By default, `fwrite` operates synchronously. To write binary data asynchronously to the device, you must specify `async` as the last input argument to `fwrite`. For more information about synchronous and asynchronous write operations, refer to the “Writing Text Data” on page 10-37. For a description of the rules used by `fwrite` to complete a write operation, refer to its reference pages.

Reading Data

This section describes reading data from your serial port device in three parts:

- “The Input Buffer and Data Flow” on page 10-40 describes the flow of data from the device to MATLAB.
- “Reading Text Data” on page 10-42 describes how to read from the device, and format the data as text.
- “Reading Binary Data” on page 10-44 describes how to read binary (numerical) data from the device.

The functions associated with reading data are shown below.

Functions Associated with Reading Data

Function Name	Description
`fgetl`	Read one line of text from the device and discard the terminator
`fgets`	Read one line of text from the device and include the terminator
`fread`	Read binary data from the device
`fscanf`	Read data from the device and format as text
`readasync`	Read data asynchronously from the device
`stopasync`	Stop asynchronous read and write operations

The properties associated with reading data are shown below.

Properties Associated with Reading Data

Property Name	Description
`BytesAvailable`	Indicate the number of bytes available in the input buffer
`InputBufferSize`	Specify the size of the input buffer in bytes
`ReadAsyncMode`	Specify whether an asynchronous read operation is continuous or manual
`Timeout`	Specify the waiting time to complete a read or write operation
`TransferStatus`	Indicate if an asynchronous read or write operation is in progress
`ValuesReceived`	Indicate the total number of values read from the device

The Input Buffer and Data Flow

The input buffer is computer memory allocated by the serial port object to store data that is to be read from the device. When reading data from your device, the data flow follows these two steps:

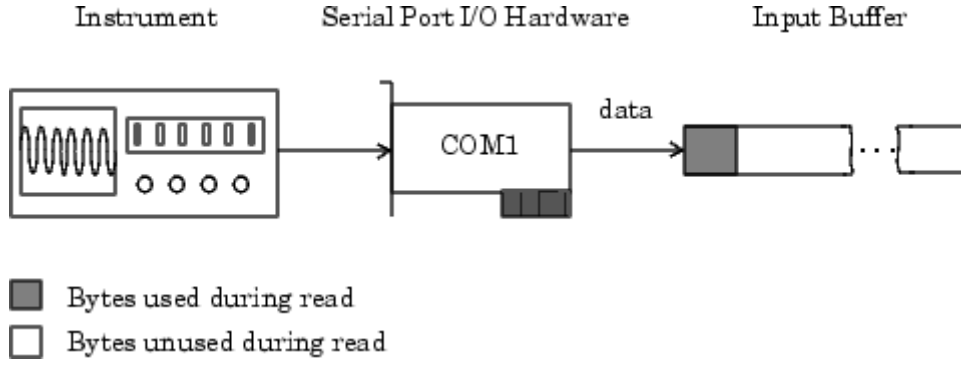
1. The data read from the device is stored in the input buffer.
2. The data in the input buffer is returned to the MATLAB variable specified by the read function.

The `InputBufferSize` property specifies the maximum number of bytes that you can store in the input buffer. The `BytesAvailable` property indicates the number of bytes currently available to be read from the input buffer. The default values for these properties are shown below.

```
s = serial('COM1');
get(s,{'InputBufferSize','BytesAvailable'})
ans =
    [512]    [0]
```

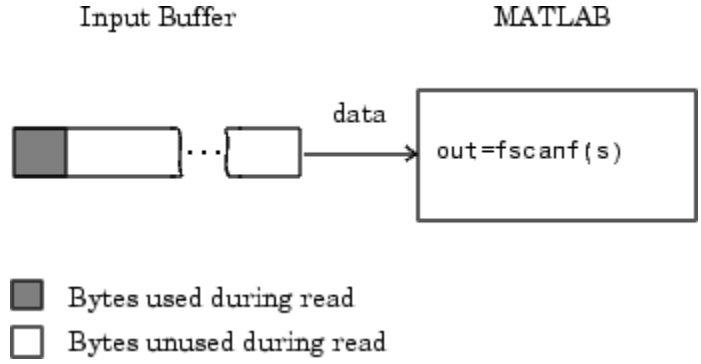
If you attempt to read more data than can fit in the input buffer, an error is returned and no data is read.

For example, suppose you use the `fscanf` function to read the text-based response of the `*IDN?` command previously written to the TDS 210 oscilloscope. As shown below, the text data is first read into to the input buffer via the serial port.



Note that for a given read operation, you might not know the number of bytes returned by the device. Therefore, you might need to preset the `InputBufferSize` property to a sufficiently large value before connecting the serial port object.

As shown below, after the data is stored in the input buffer, it is then transferred to the output variable specified by `fscanf`.



Reading Text Data

You use the `fgetl`, `fgets`, and `fscanf` functions to read data from the device, and format the data as text.

For example, suppose you want to return identification information for the oscilloscope. This requires writing the `*IDN?` command to the instrument, and then reading back the result of that command.

```
s = serial('COM1');
fopen(s)
fprintf(s,'*IDN?')
out = fscanf(s)
out =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

By default, `fscanf` reads data using the `%c` format because the data returned by many serial port devices is text based. However, you can specify many other formats as described in the `fscanf` reference pages.

You can verify the number of values read from the device - including the terminator — with the `ValuesReceived` property.

```
s.ValuesReceived
ans =
    56
```

Synchronous Versus Asynchronous Read Operations. You specify whether read operations are synchronous or asynchronous with the `ReadAsyncMode` property. You can configure `ReadAsyncMode` to `continuous` or `manual`.

If `ReadAsyncMode` is `continuous` (the default value), the serial port object continuously queries the device to determine if data is available to be read. If data is available, it is asynchronously stored in the input buffer. To transfer the data from the input buffer to MATLAB, use one of the synchronous (blocking) read functions such as `fgetl` or `fscanf`. If data is available in the input buffer, these functions return quickly.

```
s.ReadAsyncMode = 'continuous';
fprintf(s,'*IDN?')
s.BytesAvailable
ans =
    56
out = fscanf(s);
```

If `ReadAsyncMode` is `manual`, the serial port object does not continuously query the device to determine if data is available to be read. To read data asynchronously, use the `readasync` function. Then use one of the synchronous read functions to transfer data from the input buffer to MATLAB.

```
s.ReadAsyncMode = 'manual';
fprintf(s,'*IDN?')
s.BytesAvailable
ans =
    0
readasync(s)
s.BytesAvailable
ans =
    56
out = fscanf(s);
```

Asynchronous operations do not block access to the MATLAB command line. Additionally, while an asynchronous read operation is in progress, you can

- Execute an asynchronous write operation because serial ports have separate pins for reading and writing

- Make use of all supported callback properties

You can determine which asynchronous operations are in progress with the `TransferStatus` property. If no asynchronous operations are in progress, then `TransferStatus` is `idle`.

```
s.TransferStatus
ans =
idle
```

Rules for Completing a Read Operation with fscanf. A read operation with `fscanf` blocks access to the MATLAB command line until:

- The terminator specified by the `Terminator` property is read.
- The time specified by the `Timeout` property passes.
- The specified number of values specified is read.
- The input buffer is filled.

Reading Binary Data

You use the `fread` function to read binary data from the device. Reading binary data means that you return numerical values to MATLAB.

For example, suppose you want to return the cursor and display settings for the oscilloscope. This requires writing the `CURSOR?` and `DISPLAY?` commands to the instrument, and then reading back the results of those commands.

```
s = serial('COM1');
fopen(s)
fprintf(s,'CURSOR?')
fprintf(s,'DISPLAY?')
```

Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the device. You can verify the number of values read with the `BytesAvailable` property.

```
s.BytesAvailable
ans =
    69
```

You can return the data to MATLAB using any of the synchronous read functions. However, if you use `fgetl`, `fgets`, or `fscanf`, you must issue the function twice because there are two terminators stored in the input buffer. If you use `fread`, you can return all the data to MATLAB in one function call.

```
out = fread(s,69);
```

By default, `fread` returns numerical values in double precision arrays. However, you can specify many other precisions as described in the `fread` reference pages. You can convert the numerical data to text using the MATLAB `char` function.

```
val = char(out)'
val =
HBARS;CH1;SECONDS;-1.0E-3;1.0E-3;VOLTS;-6.56E-1;6.24E-1
YT;DOTS;0;45
```

For more information about synchronous and asynchronous read operations, refer to "Reading Text Data" on page 10-42. For a description of the rules used by `fread` to complete a read operation, refer to its reference pages.

Example — Writing and Reading Text Data

This example illustrates how to communicate with a serial port instrument by writing and reading text data.

The instrument is a Tektronix TDS 210 two-channel oscilloscope connected to the COM1 port. Therefore, many of the commands shown below are specific to this instrument. A sine wave is input into channel 2 of the oscilloscope, and your job is to measure the peak-to-peak voltage of the input signal.

1. Create a serial port object — Create the serial port object `s` associated with serial port COM1.

   ```
   s = serial('COM1');
   ```

2. Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

3 Write and read data — Write the `*IDN?` command to the instrument using `fprintf`, and then read back the result of the command using `fscanf`.

```
fprintf(s,'*IDN?')
idn = fscanf(s)
idn =
TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
```

You need to determine the measurement source. Possible measurement sources include channel 1 and channel 2 of the oscilloscope.

```
fprintf(s,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s)
source =
CH1
```

The scope is configured to return a measurement from channel 1. Because the input signal is connected to channel 2, you must configure the instrument to return a measurement from this channel.

```
fprintf(s,'MEASUREMENT:IMMED:SOURCE CH2')
fprintf(s,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s)
source =
CH2
```

You can now configure the scope to return the peak-to-peak voltage, and then request the value of this measurement.

```
fprintf(s,'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(s,'MEASUREMENT:MEAS1:VALUE?')
```

Transfer data from the input buffer to MATLAB using `fscanf`.

```
ptop = fscanf(s,'%g')
ptop =
2.0199999809E0
```

4 Disconnect and clean up — When you no longer need s disconnect it from the instrument and remove it from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

Example — Parsing Input Data Using strread

This example illustrates how to use the `strread` function to parse and format data that you read from a device. `strread` is particularly useful when you want to parse a string into one or more variables, where each variable has its own specified format.

The instrument is a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

1 Create a serial port object — Create the serial port object `s` associated with serial port COM1.

```
s = serial('COM1');
```

2 Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

3 Write and read data — Write the `RS232?` command to the instrument using `fprintf`, and then read back the result of the command using `fscanf`. `RS232?` queries the RS-232 settings and returns the baud rate, the software flow control setting, the hardware flow control setting, the parity type, and the terminator.

```
fprintf(s,'RS232?')
data = fscanf(s)
data =
9600;0;0;NONE;LF
```

Use the `strread` function to parse and format the `data` variable into five new variables.

```
[br,sfc,hfc,par,tm] = strread(data,'%d%d%d%s%s','delimiter',';')

br =
        9600
sfc =
     0
hfc =
     0
par =
    'NONE'
tm =
    'LF'
```

4 Disconnect and clean up — When you no longer need `s`, you should disconnect it from the instrument, and remove it from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

Example — Reading Binary Data

This example illustrates how you can download the TDS 210 oscilloscope screen display to MATLAB. The screen display data is transferred and saved to disk using the Windows bitmap format. This data provides a permanent record of your work, and is an easy way to document important signal and scope parameters.

Because the amount of data transferred is expected to be fairly large, it is asynchronously returned to the input buffer as soon as it is available from the instrument. This allows you to perform other tasks as the transfer progresses. Additionally, the scope is configured to its highest baud rate of 19,200.

1 Create a serial port object — Create the serial port object `s` associated with serial port COM1.

```
s = serial('COM1');
```

2 Configure property values — Configure the input buffer to accept a reasonably large number of bytes, and configure the baud rate to the highest value supported by the scope.

```
s.InputBufferSize = 50000;
s.BaudRate = 19200;
```

3 Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

4 Write and read data — Configure the scope to transfer the screen display as a bitmap.

```
fprintf(s,'HARDCOPY:PORT RS232')
fprintf(s,'HARDCOPY:FORMAT BMP')
fprintf(s,'HARDCOPY START')
```

Wait until all the data is sent to the input buffer, and then transfer the data to the MATLAB workspace as unsigned 8-bit integers.

```
out = fread(s,s.BytesAvailable,'uint8');
```

5 Disconnect and clean up — When you no longer need `s`, disconnect it from the instrument and remove it from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

Viewing the Bitmap Data

To view the bitmap data, follow these steps:

1 Open a disk file.

2 Write the data to the disk file.

3 Close the disk file.

4 Read the data into MATLAB using the `imread` function.

5 Scale and display the data using the `imagesc` function.

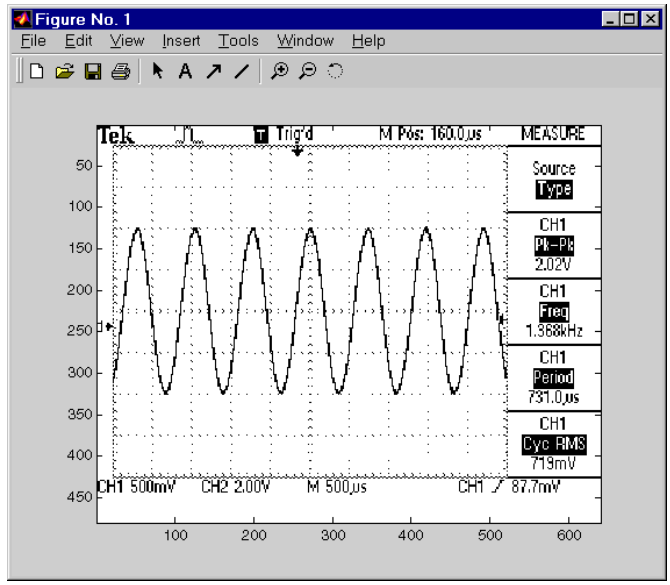
Note that the file I/O versions of the `fopen`, `fwrite`, and `fclose` functions are used.

```
fid = fopen('test1.bmp','w');
fwrite(fid,out,'uint8');
fclose(fid)
a = imread('test1.bmp','bmp');
imagesc(a)
```

Because the scope returns the screen display data using only two colors, an appropriate colormap is selected.

```
mymap = [0 0 0; 1 1 1];
colormap(mymap)
```

The resulting bitmap image is shown below.



Events and Callbacks

In this section...

Introduction

You can enhance the power and flexibility of your serial port application by using *events*. An event occurs after a condition is met and might result in one or more callbacks.

While the serial port object is connected to the device, you can use events to display a message, display data, analyze data, and so on. Callbacks are controlled through *callback properties* and *callback functions*. All event types have an associated callback property. Callback functions are M-file functions that you construct to suit your specific application needs.

You execute a callback when a particular event occurs by specifying the name of the M-file callback function as the value for the associated callback property.

Example — Introduction to Events and Callbacks

This example uses the M-file callback function `instrcallback` to display a message to the command line when a bytes-available event occurs. The event is generated when the terminator is read.

```
s = serial('COM1');
fopen(s)
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @instrcallback;
```

```
fprintf(s,'*IDN?')
out = fscanf(s);
```

The resulting display from `instrcallback` is shown below.

```
BytesAvailable event occurred at 17:01:29 for the object:
Serial-COM1.
```

End the serial port session.

```
fclose(s)
delete(s)
clear s
```

You can use the `type` command to display `instrcallback` at the command line.

Event Types and Callback Properties

The serial port event types and callback properties are described below.

This table consists of two columns and nine rows. In the first column (event type), the second item (bytes available) applies to rows 2 through 4. Also, in the first column the last item (timer) applies to rows 8 and 9.

Event Types and Callback Properties

Event Type	Associated Properties
Break interrupt	`BreakInterruptFcn`
Bytes available	`BytesAvailableFcn`
	`BytesAvailableFcnCount`
	`BytesAvailableFcnMode`
Error	`ErrorFcn`
Output empty	`OutputEmptyFcn`
Pin status	`PinStatusFcn`

Event Types and Callback Properties (Continued)

Event Type	Associated Properties
Timer	`TimerFcn`
	`TimerPeriod`

Break-Interrupt Event

A break-interrupt event is generated immediately after a break interrupt is generated by the serial port. The serial port generates a break interrupt when the received data has been in an inactive state longer than the transmission time for one character.

This event executes the callback function specified for the `BreakInterruptFcn` property. It can be generated for both synchronous and asynchronous read and write operations.

Bytes-Available Event

A bytes-available event is generated immediately after a predetermined number of bytes are available in the input buffer or a terminator is read, as determined by the `BytesAvailableFcnMode` property.

If `BytesAvailableFcnMode` is `byte`, the bytes-available event executes the callback function specified for the `BytesAvailableFcn` property every time the number of bytes specified by `BytesAvailableFcnCount` is stored in the input buffer. If `BytesAvailableFcnMode` is `terminator`, the callback function executes every time the character specified by the `Terminator` property is read.

This event can be generated only during an asynchronous read operation.

Error Event

An error event is generated immediately after an error occurs.

This event executes the callback function specified for the `ErrorFcn` property. It can be generated only during an asynchronous read or write operation.

An error event is generated when a time-out occurs. A time-out occurs if a read or write operation does not successfully complete within the time specified by the `Timeout` property. An error event is not generated for configuration errors such as setting an invalid property value.

Output-Empty Event

An output-empty event is generated immediately after the output buffer is empty.

This event executes the callback function specified for the `OutputEmptyFcn` property. It can be generated only during an asynchronous write operation.

Pin Status Event

A pin status event is generated immediately after the state (pin value) changes for the CD, CTS, DSR, or RI pins. Refer to "Serial Port Signals and Pin Assignments" on page 10-7 for a description of these pins.

This event executes the callback function specified for the `PinStatusFcn` property. It can be generated for both synchronous and asynchronous read and write operations.

Timer Event

A timer event is generated when the time specified by the `TimerPeriod` property passes. Time is measured relative to when the serial port object is connected to the device.

This event executes the callback function specified for the `TimerFcn` property. Note that some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value is too small.

Storing Event Information

You can store event information in a callback function or in a record file. Event information is stored in a callback function using two fields: `Type` and `Data`. The `Type` field contains the event type, while the `Data` field contains event-specific information. As described in "Creating and Executing Callback Functions" on page 10-57, these two fields are associated with a structure that you define in the callback function header. Refer to "Debugging: Recording

Information to Disk" on page 10-66 to learn about recording data and event information to a record file.

The event types and the values for the `Type` and `Data` fields are shown below.

The table consists of three columns and 15 rows. Items in the first column (event type) span several rows, as follows:

Break interrupt: rows 1 and 2

Bytes available: rows 3 and 4

Error: rows 5 through 7

Output empty: rows 8 and 9

Pin status: rows 10 through 13

Timer: rows 14 and 15

Event Information

Event Type	Field	Field Value
Break interrupt	`Type`	`BreakInterrupt`
	`Data.AbsTime`	day-month-year hour:minute:second
Bytes available	`Type`	`BytesAvailable`
	`Data.AbsTime`	day-month-year hour:minute:second
Error	`Type`	`Error`
	`Data.AbsTime`	day-month-year hour:minute:second
	`Data.Message`	An error string

Event Information (Continued)

Event Type	Field	Field Value
Output empty	`Type`	`OutputEmpty`
	`Data.AbsTime`	day-month-year hour:minute:second
Pin status	`Type`	`PinStatus`
	`Data.AbsTime`	day-month-year hour:minute:second
	`Data.Pin`	`CarrierDetect`, `ClearToSend`, `DataSetReady`, or `RingIndicator`
	`Data.PinValue`	`on` or `off`
Timer	`Type`	`Timer`
	`Data.AbsTime`	day-month-year hour:minute:second

The `Data` field values are described below.

The AbsTime Field

`AbsTime` is defined for all events, and indicates the absolute time the event occurred. The absolute time is returned using the `clock` format: day-month-year hour:minute:second

The Pin Field

`Pin` is used by the pin status event to indicate if the CD, CTS, DSR, or RI pins changed state. Refer to “Serial Port Signals and Pin Assignments” on page 10-7 for a description of these pins.

The PinValue Field

`PinValue` is used by the pin status event to indicate the state of the CD, CTS, DSR, or RI pins. Possible values are `on` or `off`.

The Message Field

`Message` is used by the error event to store the descriptive message that is generated when an error occurs.

Creating and Executing Callback Functions

You can specify the callback function to be executed when a specific event type occurs by including the name of the M-file as the value for the associated callback property. You can specify the callback function as a function handle or as a string cell array element. Function handles are described in the `function_handle` reference pages.

For example, to execute the callback function `mycallback` every time the terminator is read from your device

```
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @mycallback;
```

Alternatively, you can specify the callback function as a cell array.

```
s.BytesAvailableFcn = {'mycallback'};
```

M-file callback functions require at least two input arguments. The first argument is the serial port object. The second argument is a variable that captures the event information shown in the table, Event Information on page 10-55. This event information pertains only to the event that caused the callback function to execute. The function header for `mycallback` is shown below.

```
function mycallback(obj,event)
```

You pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array. For example, to pass the MATLAB variable `time` to `mycallback`

```
time = datestr(now,0);
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = {@mycallback,time};
```

Alternatively, you can specify the callback function as a string in the cell array.

```
s.BytesAvailableFcn = {'mycallback',time};
```

The corresponding function header is

```
function mycallback(obj,event,time)
```

If you pass additional parameters to the callback function, they must be included in the function header after the two required arguments.

Note You can also specify the callback function as a string. In this case, the callback is evaluated in the MATLAB workspace and no requirements are made on the input arguments of the callback function.

Enabling Callback Functions After They Error

If an error occurs while a callback function is executing the following occurs:

- The callback function is automatically disabled.
- A warning is displayed at the command line, indicating that the callback function is disabled.

If you want to enable the same callback function, set the callback property to the same value or disconnect the object with the `fclose` function. If you want to use a different callback function, the callback is enabled when you configure the callback property to the new value.

Example — Using Events and Callbacks

This example uses the M-file callback function `instrcallback` to display event-related information to the command line when a bytes-available event or an output-empty event occurs.

1. Create a serial port object — Create the serial port object `s` associated with serial port COM1.

   ```
   s = serial('COM1');
   ```

2. Connect to the device — Connect `s` to the Tektronix TDS 210 oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

3 Configure properties — Configure `s` to execute the callback function `instrcallback` when a bytes-available event or an output-empty event occurs. Because `instrcallback` requires the serial port object and event information to be passed as input arguments, the callback function is specified as a function handle.

```
s.BytesAvailableFcnMode = 'terminator';
s.BytesAvailableFcn = @instrcallback;
s.OutputEmptyFcn = @instrcallback;
```

4 Write and read data — Write the `RS232?` command asynchronously to the oscilloscope. This command queries the RS-232 settings and returns the baud rate, the software flow control setting, the hardware flow control setting, the parity type, and the terminator.

```
fprintf(s,'RS232?','async')
```

`instrcallback` is called after the `RS232?` command is sent, and when the terminator is read. The resulting displays are shown below.

```
OutputEmpty event occurred at 17:37:21 for the object:
Serial-COM1.

BytesAvailable event occurred at 17:37:21 for the object:
Serial-COM1.
```

Read the data from the input buffer.

```
out = fscanf(s)
out =
9600;0;0;NONE;LF
```

5 Disconnect and clean up — When you no longer need `s`, disconnect it from the instrument and remove it from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

Using Control Pins

In this section...

Properties of Serial Port Control Pins

As described in "Serial Port Signals and Pin Assignments" on page 10-7, 9-pin serial ports include six control pins. The properties associated with the serial port control pins are shown below.

Control Pin Properties

Property Name	Description
`DataTerminalReady`	Specify the state of the DTR pin
`FlowControl`	Specify the data flow control method to use
`PinStatus`	Indicate the state of the CD, CTS, DSR, and RI pins
`RequestToSend`	Specify the state of the RTS pin

Signaling the Presence of Connected Devices

DTEs and DCEs often use the CD, DSR, RI, and DTR pins to indicate whether a connection is established between serial port devices. Once the connection is established, you can begin to write or read data.

You can monitor the state of the CD, DSR, and RI pins with the `PinStatus` property. You can specify or monitor the state of the DTR pin with the `DataTerminalReady` property.

The following example illustrates how these pins are used when two modems are connected to each other.

Example — Connecting Two Modems

This example connects two modems to each other via the same computer, and illustrates how you can monitor the communication status for the computer-modem connections, and for the modem-modem connection. The first modem is connected to COM1, while the second modem is connected to COM2.

1. Create the serial port objects — After the modems are powered on, the serial port object `s1` is created for the first modem, and the serial port object `s2` is created for the second modem.

   ```
   s1 = serial('COM1');
   s2 = serial('COM2');
   ```

2. Connect to the devices — `s1` and `s2` are connected to the modems. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned to the input buffers as soon as it is available from the modems.

   ```
   fopen(s1)
   fopen(s2)
   ```

   Because the default `DataTerminalReady` property value is `on`, the computer (data terminal) is now ready to exchange data with the modems. You can verify that the modems (data sets) can communicate with the computer by examining the value of the Data Set Ready pin with the `PinStatus` property.

   ```
   s1.Pinstatus
   ans =
       CarrierDetect: 'off'
         ClearToSend: 'on'
        DataSetReady: 'on'
       RingIndicator: 'off'
   ```

   The value of the `DataSetReady` field is `on` because both modems were powered on before they were connected to the objects.

3. Configure properties — Both modems are configured for a baud rate of 2400 bits per second and a carriage return (CR) terminator.

```
s1.BaudRate = 2400;
s1.Terminator = 'CR';
s2.BaudRate = 2400;
s2.Terminator = 'CR';
```

4 Write and read data — Write the `atd` command to the first modem. This command puts the modem "off the hook," which is equivalent to manually lifting a phone receiver.

```
fprintf(s1,'atd')
```

Write the `ata` command to the second modem. This command puts the modem in "answer mode," which forces it to connect to the first modem.

```
fprintf(s2,'ata')
```

After the two modems negotiate their connection, verify the connection status by examining the value of the Carrier Detect pin using the `PinStatus` property.

```
s1.PinStatus
ans =
    CarrierDetect: 'on'
      ClearToSend: 'on'
     DataSetReady: 'on'
    RingIndicator: 'off'
```

Verify the modem-modem connection by reading the descriptive message returned by the second modem.

```
s2.BytesAvailable
ans =
    25
out = fread(s2,25);
char(out)'
ans =
ata
CONNECT 2400/NONE
```

Now break the connection between the two modems by configuring the `DataTerminalReady` property to `off`. You can verify that the modems are disconnected by examining the Carrier Detect pin value.

```
s1.DataTerminalReady = 'off';
s1.PinStatus
ans =
    CarrierDetect: 'off'
      ClearToSend: 'on'
     DataSetReady: 'on'
    RingIndicator: 'off'
```

5 Disconnect and clean up — Disconnect the objects from the modems and remove the objects from memory and from the MATLAB workspace.

```
fclose([s1 s2])
delete([s1 s2])
clear s1 s2
```

Controlling the Flow of Data: Handshaking

Data flow control or *handshaking* is a method used for communicating between a DCE and a DTE to prevent data loss during transmission. For example, suppose your computer can receive only a limited amount of data before it must be processed. As this limit is reached, a handshaking signal is transmitted to the DCE to stop sending data. When the computer can accept more data, another handshaking signal is transmitted to the DCE to resume sending data.

If supported by your device, you can control data flow using one of these methods:

- Hardware handshaking
- Software handshaking

Note Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, MATLAB does not support this behavior.

You can specify the data flow control method with the `FlowControl` property. If `FlowControl` is `hardware`, hardware handshaking is used to control data

flow. If `FlowControl` is `software`, software handshaking is used to control data flow. If `FlowControl` is `none`, no handshaking is used.

Hardware Handshaking

Hardware handshaking uses specific serial port pins to control data flow. In most cases, these are the RTS and CTS pins. Hardware handshaking using these pins is described in "The RTS and CTS Pins" on page 10-10.

If `FlowControl` is `hardware`, the RTS and CTS pins are automatically managed by the DTE and DCE. You can return the CTS pin value with the `PinStatus` property. Configure or return the RTS pin value with the `RequestToSend` property.

Note Some devices also use the DTR and DSR pins for handshaking. However, these pins are typically used to indicate that the system is ready for communication, and are not used to control data transmission. In MATLAB, hardware handshaking always uses the RTS and CTS pins.

If your device does not use hardware handshaking in the standard way, then you might need to manually configure the `RequestToSend` property. In this case, you should configure `FlowControl` to `none`. If `FlowControl` is `hardware`, then the `RequestToSend` value that you specify might not be honored. Refer to the device documentation to determine its specific pin behavior.

Software Handshaking

Software handshaking uses specific ASCII characters to control data flow. These characters, known as Xon and Xoff (or XON and XOFF), are described below.

Software Handshaking Characters

Character	Integer Value	Description
Xon	17	Resume data transmission
Xoff	19	Pause data transmission

When using software handshaking, the control characters are sent over the transmission line the same way as regular data. Therefore, only the TD, RD, and GND pins are needed.

The main disadvantage of software handshaking is that Xon or Xoff characters are not writable while numerical data is being written to the device. This is because numerical data might contain a 17 or 19, which makes it impossible to distinguish between the control characters and the data. However, you can write Xon or Xoff while data is being asynchronously read from the device because you are using both the TD and RD pins.

Example: Using Software Handshaking

Suppose you want to use software flow control with the example described in “Example — Reading Binary Data” on page 10-48. To do this, you must configure the oscilloscope and serial port object for software flow control.

```
fprintf(s,'RS232:SOFTF ON')
s.FlowControl = 'software';
```

To pause data transfer, write the numerical value `19` to the device.

```
fwrite(s,19)
```

To resume data transfer, write the numerical value `17` to the device.

```
fwrite(s,17)
```

Debugging: Recording Information to Disk

In this section...

Introduction

Recording information to disk provides a permanent record of your serial port session, and is an easy way to debug your application. While the serial port object is connected to the device, you can record the following information to a disk file:

- The number of values written to the device, the number of values read from the device, and the data type of the values
- Data written to the device, and data read from the device
- Event information

Recording Properties

You record information to a disk file with the `record` function. The following table shows the properties associated with recording information to disk.

Recording Properties

Property Name	Description
`RecordDetail`	Specify the amount of information saved to a record file
`RecordMode`	Specify whether data and event information is saved to one record file or to multiple record files

Recording Properties (Continued)

Property Name	Description
RecordName	Specify the name of the record file
RecordStatus	Indicate if data and event information are saved to a record file

Example: Introduction to Recording Information

This example records the number of values written to and read from the device, and stores the information to the file myfile.txt.

```
s = serial('COM1');
fopen(s)
s.RecordName = 'myfile.txt';
record(s)
fprintf(s,'*IDN?')
idn = fscanf(s);
fprintf(s,'RS232?')
rs232 = fscanf(s);
```

End the serial port session.

```
fclose(s)
delete(s)
clear s
```

You can use the type command to display myfile.txt at the command line.

Creating Multiple Record Files

When you initiate recording with the record function, the RecordMode property determines if a new record file is created or if new information is appended to an existing record file.

You can configure RecordMode to overwrite, append, or index. If RecordMode is overwrite, the record file is overwritten each time recording is initiated. If RecordMode is append, the new information is appended to the file specified by RecordName. If RecordMode is index, a different disk file is created each

time recording is initiated. The rules for specifying a record filename are discussed in the next section.

Specifying a Filename

You specify the name of the record file with the `RecordName` property. You can specify any value for `RecordName` — including a directory path — provided the filename is supported by your operating system. Additionally, if `RecordMode` is `index`, the filename follows these rules:

- Indexed filenames are identified by a number. This number precedes the filename extension and is increased by 1 for successive record files.
- If no number is specified as part of the initial filename, the first record file does not have a number associated with it. For example, if `RecordName` is `myfile.txt`, `myfile.txt` is the name of the first record file, `myfile01.txt` is the name of the second record file, and so on.
- `RecordName` is updated after the record file is closed.
- If the specified filename already exists, the existing file is overwritten.

The Record File Format

The record file is an ASCII file that contains a record of one or more serial port sessions. You specify the amount of information saved to a record file with the `RecordDetail` property.

`RecordDetail` can be `compact` or `verbose`. A compact record file contains the number of values written to the device, the number of values read from the device, the data type of the values, and event information. A verbose record file contains the preceding information as well as the data transferred to and from the device.

Binary data with precision given by `uchar`, `schar`, `(u)int8`, `(u)int16`, or `(u)int32` is recorded using hexadecimal format. For example, if the integer value 255 is read from the instrument as a 16-bit integer, the hexadecimal value 00FF is saved in the record file. Single- and double-precision floating-point numbers are recorded as decimal values using the `%g` format, and as hexadecimal values using the format specified by the IEEE Standard 754-1985 for Binary Floating-Point Arithmetic.

The IEEE floating-point format includes three components: the sign bit, the exponent field, and the significant field. Single-precision floating-point values consist of 32 bits. The value is given by

$$value = (-1)^{sign}(2^{exp-127})(1.significand)$$

Double-precision floating-point values consist of 64 bits. The value is given by

$$value = (-1)^{sign}(2^{exp-1023})(1.significand)$$

The floating-point format component, and the associated single-precision and double-precision bits are shown below.

Component	Single-Precision Bits	Double-Precision Bits
`sign`	1	1
`exp`	2–9	2–12
`significand`	10–32	13–64

Bit 1 is the left-most bit as stored in the record file.

Example: Recording Information to Disk

This example illustrates how to record information transferred between a serial port object and a Tektronix TDS 210 oscilloscope. Additionally, the structure of the resulting record file is presented.

1. Create the serial port object — Create the serial port object `s` associated with the serial port COM1.

```
s = serial('COM1');
```

2. Connect to the device — Connect `s` to the oscilloscope. Because the default value for the `ReadAsyncMode` property is `continuous`, data is asynchronously returned the input buffer as soon as it is available from the instrument.

```
fopen(s)
```

3 Configure property values — Configure `s` to record information to multiple disk files using the verbose format. Recording is then initiated with the first disk file defined as `WaveForm1.txt`.

```
s.RecordMode = 'index';
s.RecordDetail = 'verbose';
s.RecordName = 'WaveForm1.txt';
record(s)
```

4 Write and read data — The commands written to the instrument, and the data read from the instrument are recorded in the record file. Refer to "Example — Writing and Reading Text Data" on page 10-45 for an explanation of the oscilloscope commands.

```
fprintf(s,'*IDN?')
idn = fscanf(s);
fprintf(s,'MEASUREMENT:IMMED:SOURCE CH2')
fprintf(s,'MEASUREMENT:IMMED:SOURCE?')
source = fscanf(s);
```

Read the peak-to-peak voltage with the `fread` function. Note that the data returned by `fread` is recorded using hex format.

```
fprintf(s,'MEASUREMENT:MEAS1:TYPE PK2PK')
fprintf(s,'MEASUREMENT:MEAS1:VALUE?')
ptop = fread(s,s.BytesAvailable);
```

Convert the peak-to-peak voltage to a character array.

```
char(ptop)'
ans =
2.0199999809E0
```

The recording state is toggled from `on` to `off`. Because the `RecordMode` value is `index`, the record filename is automatically updated.

```
record(s)
s.RecordStatus
ans =
off
s.RecordName
ans =
```

```
WaveForm2.txt
```

5 Disconnect and clean up — When you no longer need s, disconnect it from the instrument, and remove it from memory and from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

The Record File Contents

The contents of the `WaveForm1.txt` record file are shown below. Because the `RecordDetail` property was `verbose`, the number of values, commands, and data were recorded. Note that data returned by the `fread` function is in hex format.

```
type WaveForm1.txt

Legend:
  * - An event occurred.
  > - A write operation occurred.
  < - A read operation occurred.
1      Recording on 22-Jan-2000 at 11:21:21.575. Binary data in...
2    > 6 ascii values.
       *IDN?
3    < 56 ascii values.
       TEKTRONIX,TDS 210,0,CF:91.1CT FV:v1.16 TDS2CM:CMV:v1.04
4    > 29 ascii values.
       MEASUREMENT:IMMED:SOURCE CH2
5    > 26 ascii values.
       MEASUREMENT:IMMED:SOURCE?
6    < 4 ascii values.
       CH2
7    > 27 ascii values.
       MEASUREMENT:MEAS1:TYPE PK2PK
8    > 25 ascii values.
       MEASUREMENT:MEAS1:VALUE?
9    < 15 uchar values.
       32 2e 30 31 39 39 39 39 39 38 30 39 45 30 0a
10     Recording off.
```

Saving and Loading

In this section...

Using save and load

You can save serial port objects to a MAT-file just as you would any workspace variable - using the `save` command. For example, suppose you create the serial port object `s` associated with the serial port COM1, configure several property values, and perform a write and read operation.

```
s = serial('COM1');
s.BaudRate = 19200;
s.Tag = 'My serial object';
fopen(s)
fprintf(s, '*IDN?')
out = fscanf(s);
```

To save the serial port object and the data read from the device to the MAT-file `myserial.mat`

```
save myserial s out
```

Note You can save data and event information as text to a disk file with the `record` function.

You can recreate `s` and `out` in the workspace using the `load` command.

```
load myserial
```

Values for read only properties are restored to their default values upon loading. For example, the `Status` property is restored to `closed`. Therefore, to use `s`, you must connect it to the device with the `fopen` function. To determine if a property is read only, examine its reference pages.

Using Serial Port Objects on Different Platforms

If you save a serial port object from one platform, and then load that object on a different platform having different serial port names, you need to modify the `Port` property value. For example, suppose you create the serial port object `s` associated with the serial port COM1 on a Windows platform. If you want to save `s` for eventual use on a Linux platform, configure `Port` to an appropriate value such as `ttyS0` after the object is loaded.

Disconnecting and Cleaning Up

In this section...

Disconnecting a Serial Port Object

When you no longer need to communicate with the device, disconnect it from the serial port object with the `fclose` function.

```
fclose(s)
```

Examine the `Status` property to verify that the serial port object and the device are disconnected.

```
s.Status
ans =
closed
```

After `fclose` is issued, the serial port associated with `s` is available. Now connect another serial port object to it using `fopen`.

Cleaning Up the MATLAB Environment

When the serial port object is no longer needed, remove it from memory with the `delete` function.

```
delete(s)
```

Before using `delete`, disconnect the serial port object from the device with the `fclose` function.

A deleted serial port object is *invalid*, which means that you cannot connect it to the device. In this case, remove the object from the MATLAB workspace. To remove serial port objects and other variables from the MATLAB workspace, use the `clear` command.

```
clear s
```

Use `clear` on a serial port object that is still connected to a device to remove the object from the workspace but leave it connected to the device. Restore cleared objects to MATLAB with the `instrfind` function.

Property Reference

In this section...

The Property Reference Page Format

Each serial port property description contains some or all of this information:

- The property name
- A description of the property
- The property characteristics, including:
  - Read only — The condition under which the property is read only

    A property can be read-only always, never, while the serial port object is open, or while the serial port object is recording. You can configure a property value using the `set` function or dot notation. You can return the current property value using the `get` function or dot notation.
  - Data type — the property data type

    This is the data type you use when specifying a property value.
- Valid property values including the default value

  When property values are given by a predefined list, the default value is usually indicated by `{}`.
- An example using the property
- Related properties and functions

Serial Port Object Properties

The serial port object properties are briefly described below, and organized into categories based on how they are used. Following this section the properties are listed alphabetically and described in detail.

Communications Properties	
BaudRate	Specify the rate at which bits are transmitted
DataBits	Specify the number of data bits to transmit
Parity	Specify the type of parity checking
StopBits	Specify the number of bits used to indicate the end of a byte
Terminator	Specify the terminator character

Write Properties	
BytesToOutput	Indicate the number of bytes currently in the output buffer
OutputBufferSize	Specify the size of the output buffer in bytes
Timeout	Specify the waiting time to complete a read or write operation
TransferStatus	Indicate if an asynchronous read or write operation is in progress
ValuesSent	Indicate the total number of values written to the device

Read Properties	
BytesAvailable	Indicate the number of bytes available in the input buffer
InputBufferSize	Specify the size of the input buffer in bytes
ReadAsyncMode	Specify whether an asynchronous read operation is continuous or manual
Timeout	Specify the waiting time to complete a read or write operation

Read Properties	
TransferStatus	Indicate if an asynchronous read or write operation is in progress
ValuesReceived	Indicate the total number of values read from the device

Callback Properties	
BreakInterruptFcn	Specify the M-file callback function to execute when a break-interrupt event occurs
BytesAvailableFcn	Specify the M-file callback function to execute when a specified number of bytes is available in the input buffer, or a terminator is read
BytesAvailableFcnCount	Specify the number of bytes that must be available in the input buffer to generate a bytes-available event
BytesAvailableFcnMode	Specify if the bytes-available event is generated after a specified number of bytes is available in the input buffer, or after a terminator is read
ErrorFcn	Specify the M-file callback function to execute when an error event occurs
OutputEmptyFcn	Specify the M-file callback function to execute when the output buffer is empty
PinStatusFcn	Specify the M-file callback function to execute when the CD, CTS, DSR, or RI pins change state
TimerFcn	Specify the M-file callback function to execute when a predefined period of time passes
TimerPeriod	Specify the period of time between timer events

Control Pin Properties	
DataTerminalReady	Specify the state of the DTR pin
FlowControl	Specify the data flow control method to use
PinStatus	Indicate the state of the CD, CTS, DSR, and RI pins
RequestToSend	Specify the state of the RTS pin

Recording Properties	
RecordDetail	Specify the amount of information saved to a record file
RecordMode	Specify whether data and event information are saved to one record file or to multiple record files
RecordName	Specify the name of the record file
RecordStatus	Indicate if data and event information are saved to a record file

General Purpose Properties	
ByteOrder	Specify the order in which the device stores bytes
Name	Specify a descriptive name for the serial port object
Port	Indicate the platform-specific serial port name
Status	Indicate if the serial port object is connected to the device
Tag	Specify a label to associate with a serial port object
Type	Indicate the object type
UserData	Specify data that you want to associate with a serial port object

Properties — Alphabetical List

Purpose

Specify the rate at which bits are transmitted

Description

You configure `BaudRate` as bits per second. The transferred bits include the start bit, the data bits, the parity bit (if used), and the stop bits. However, only the data bits are stored.

The baud rate is the rate at which information is transferred in a communication channel. In the serial port context, "9600 baud" means that the serial port is capable of transferring a maximum of 9600 bits per second. If the information unit is one baud (one bit), the bit rate and the baud rate are identical. If one baud is given as 10 bits, (for example, eight data bits plus two framing bits), the bit rate is still 9600 but the baud rate is 9600/10, or 960. You always configure `BaudRate` as bits per second. Therefore, in the above example, set `BaudRate` to 9600.

Note Both the computer and the peripheral device must be configured to the same baud rate before you can successfully read or write data.

Standard baud rates include 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200, 128000 and 256000 bits per second. To display the supported baud rates for the serial ports on your platform, refer to "Finding Serial Port Information for Your Platform" on page 10-16.

Characteristics

Read only	Never
Data type	Double

Values

The default value is `9600`.

See Also

Properties

`DataBits`, `Parity`, `StopBits`

BreakInterruptFcn

Purpose

Specify the M-file callback function to execute when a break-interrupt event occurs

Description

You configure `BreakInterruptFcn` to execute an M-file callback function when a break-interrupt event occurs. A break-interrupt event is generated by the serial port when the received data is in an off (space) state longer than the transmission time for one byte.

Note A break-interrupt event can be generated at any time during the serial port session.

If the `RecordStatus` property value is `on`, and a break-interrupt event occurs, the record file records this information:

- The event type as `BreakInterrupt`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Refer to "Creating and Executing Callback Functions" on page 10-57 to learn how to create a callback function.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

See Also

Functions

`record`

Properties

`RecordStatus`

Purpose

Specify the byte order of the device

Description

You configure `ByteOrder` to be `littleEndian` or `bigEndian`. If `ByteOrder` is `littleEndian`, the device stores the first byte in the first memory address. If `ByteOrder` is `bigEndian`, the device stores the last byte in the first memory address.

For example, suppose the hexadecimal value 4F52 is to be stored in device memory. Because this value consists of two bytes, 4F and 52, two memory locations are used. Using big-endian format, 4F is stored first in the lower storage address. Using little-endian format, 52 is stored first in the lower storage address.

Note You should configure `ByteOrder` to the appropriate value for your device before performing a read or write operation. Refer to your device documentation for information about the order in which it stores bytes.

Characteristics

Read only	Never
Data type	String

Values

`{littleEndian}`	The byte order of the device is little-endian.
`bigEndian`	The byte order of the device is big-endian.

See Also

Properties

`Status`

BytesAvailable

Purpose

Indicate the number of bytes available in the input buffer

Description

`BytesAvailable` indicates the number of bytes currently available to be read from the input buffer. The property value is continuously updated as the input buffer is filled, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesAvailable` only when reading data asynchronously. This is because when reading data synchronously, control is returned to the MATLAB command line only after the input buffer is empty. Therefore, the `BytesAvailable` value is always 0. Refer to "Reading Text Data" on page 10-42 to learn how to read data asynchronously.

The `BytesAvailable` value can range from zero to the size of the input buffer. Use the `InputBufferSize` property to specify the size of the input buffer. Use the `ValuesReceived` property to return the total number of values read.

Characteristics

Read only	Always
Data type	Double

Values

The default value is `0`.

See Also

Functions

`fopen`

Properties

`InputBufferSize`, `TransferStatus`, `ValuesReceived`

Purpose

Specify the M-file callback function to execute when a specified number of bytes is available in the input buffer, or a terminator is read

Description

You configure `BytesAvailableFcn` to execute an M-file callback function when a bytes-available event occurs. A bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available in the input buffer, or after a terminator is read, as determined by the `BytesAvailableFcnMode` property.

Note A bytes-available event can be generated only for asynchronous read operations.

If the `RecordStatus` property value is `on`, and a bytes-available event occurs, the record file records this information:

- The event type as `BytesAvailable`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Refer to "Creating and Executing Callback Functions" on page 10-57 to learn how to create a callback function.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

Example

Create the serial port object `s` for a Tektronix TDS 210 two-channel oscilloscope connected to the serial port COM1.

```
s = serial('COM1');
```

Configure s to execute the M-file callback function instrcallback when 40 bytes are available in the input buffer.

```
s.BytesAvailableFcnCount = 40;
s.BytesAvailableFcnMode = 'byte';
s.BytesAvailableFcn = @instrcallback;
```

Connect s to the oscilloscope.

```
fopen(s)
```

Write the *IDN? command, which instructs the scope to return identification information. Because the default value for the ReadAsyncMode property is continuous, data is read as soon as it is available from the instrument.

```
fprintf(s,'*IDN?')
```

The resulting output from instrcallback is shown below.

```
BytesAvailable event occurred at 18:33:35 for the object:
Serial-COM1.
```

56 bytes are read and instrcallback is called once. The resulting display is shown above.

```
s.BytesAvailable
ans =
    56
```

Suppose you remove 25 bytes from the input buffer and then issue the MEASUREMENT? command, which instructs the scope to return its measurement settings.

```
out = fscanf(s,'%c',25);
fprintf(s,'MEASUREMENT?')
```

The resulting output from instrcallback is shown below.

```
BytesAvailable event occurred at 18:33:48 for the object:
```

```
Serial-COM1.

BytesAvailable event occurred at 18:33:48 for the object:
Serial-COM1.
```

There are now 102 bytes in the input buffer, 31 of which are left over from the `*IDN?` command. `instrcallback` is called twice—once when 40 bytes are available and once when 80 bytes are available.

```
s.BytesAvailable
ans =
   102
```

See Also

Functions

`record`

Properties

`BytesAvailableFcnCount`, `BytesAvailableFcnMode`, `RecordStatus`, `Terminator`, `TransferStatus`

BytesAvailableFcnCount

Purpose

Specify the number of bytes that must be available in the input buffer to generate a bytes-available event

Description

You configure `BytesAvailableFcnCount` to the number of bytes that must be available in the input buffer before a bytes-available event is generated.

Use the `BytesAvailableFcnMode` property to specify whether the bytes-available event occurs after a certain number of bytes are available or after a terminator is read.

The bytes-available event executes the M-file callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnCount` only when the object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	Double

Values

The default value is `48`.

See Also

Functions

`fclose`

Properties

`BytesAvailableFcn`, `BytesAvailableFcnMode`, `Status`

Purpose

Specify if the bytes-available event is generated after a specified number of bytes is available in the input buffer, or after a terminator is read

Description

You can configure `BytesAvailableFcnMode` to be `terminator` or `byte`. If `BytesAvailableFcnMode` is `terminator`, a bytes-available event occurs when the terminator specified by the `Terminator` property is reached. If `BytesAvailableFcnMode` is `byte`, a bytes-available event occurs when the number of bytes specified by the `BytesAvailableFcnCount` property is available.

The bytes-available event executes the M-file callback function specified for the `BytesAvailableFcn` property.

You can configure `BytesAvailableFcnMode` only when the object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	String

Values

`{terminator}`	A bytes-available event is generated when the terminator is read.
`byte`	A bytes-available event is generated when the specified number of bytes are available.

See Also

Functions

`fclose`

Properties

`BytesAvailableFcn`, `BytesAvailableFcnCount`, `Status`, `Terminator`

BytesToOutput

Purpose

Indicate the number of bytes currently in the output buffer

Description

`BytesToOutput` indicates the number of bytes currently in the output buffer waiting to be written to the device. The property value is continuously updated as the output buffer is filled and emptied, and is set to `0` after the `fopen` function is issued.

You can make use of `BytesToOutput` only when writing data asynchronously. This is because when writing data synchronously, control is returned to the MATLAB command line only after the output buffer is empty. Therefore, the `BytesToOutput` value is always `0`. Refer to "Writing Text Data" on page 10-37 to learn how to write data asynchronously.

Use the `ValuesSent` property to return the total number of values written to the device.

Note If you attempt to write out more data than can fit in the output buffer, an error is returned and `BytesToOutput` is `0`. Specify the size of the output buffer with the `OutputBufferSize` property.

Characteristics

Read only	Always
Data type	Double

Values

The default value is `0`.

See Also

Functions

`fopen`

Properties

`OutputBufferSize`, `TransferStatus`, `ValuesSent`

Purpose

Specify the number of data bits to transmit

Description

You can configure `DataBits` to be `5`, `6`, `7`, or `8`. Data is transmitted as a series of five, six, seven, or eight bits with the least significant bit sent first. At least seven data bits are required to transmit ASCII characters. Eight bits are required to transmit binary data. Five and six bit data formats are used for specialized communications equipment.

Note Both the computer and the peripheral device must be configured to transmit the same number of data bits.

In addition to the data bits, the serial data format consists of a start bit, one or two stop bits, and possibly a parity bit. You specify the number of stop bits with the `StopBits` property, and the type of parity checking with the `Parity` property.

To display the supported number of data bits for the serial ports on your platform, refer to "Finding Serial Port Information for Your Platform" on page 10-16.

Characteristics

Read only	Never
Data type	Double

Values

`DataBits` can be `5`, `6`, `7`, or `8`. The default value is `8`.

See Also

Properties

`Parity`, `StopBits`

DataTerminalReady

Purpose

Specify the state of the DTR pin

Description

You can configure `DataTerminalReady` to be `on` or `off`. If `DataTerminalReady` is `on`, the Data Terminal Ready (DTR) pin is asserted. If `DataTerminalReady` is `off`, the DTR pin is unasserted.

In normal usage, the DTR and Data Set Ready (DSR) pins work together, and are used to signal if devices are connected and powered. However, there is nothing in the RS-232 standard that states the DTR pin must be used in any specific way. For example, DTR and DSR might be used for handshaking. You should refer to your device documentation to determine its specific pin behavior.

You can return the value of the DSR pin with the `PinStatus` property. Handshaking is described in "Controlling the Flow of Data: Handshaking" on page 10-63.

Characteristics

Read only	Never
Data type	String

Values

`{on}`	The DTR pin is asserted.
`off`	The DTR pin is unasserted.

See Also

Properties

`FlowControl`, `PinStatus`

Purpose

Specify the M-file callback function to execute when an error event occurs

Description

You configure `ErrorFcn` to execute an M-file callback function when an error event occurs.

Note An error event is generated only for asynchronous read and write operations.

An error event is generated when a time-out occurs. A time-out occurs if a read or write operation does not successfully complete within the time specified by the `Timeout` property. An error event is not generated for configuration errors such as setting an invalid property value.

If the `RecordStatus` property value is `on`, and an error event occurs, the record file records this information:

- The event type as `Error`
- The error message
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Refer to "Creating and Executing Callback Functions" on page 10-57 to learn how to create a callback function.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

See Also

Functions

`record`

Properties

`RecordStatus`, `Timeout`

Purpose

Specify the data flow control method to use

Description

You can configure `FlowControl` to be `none`, `hardware`, or `software`. If `FlowControl` is `none`, data flow control (handshaking) is not used. If `FlowControl` is `hardware`, hardware handshaking is used to control data flow. If `FlowControl` is `software`, software handshaking is used to control data flow.

Hardware handshaking typically utilizes the Request to Send (RTS) and Clear to Send (CTS) pins to control data flow. Software handshaking uses control characters (Xon and Xoff) to control data flow. Refer to "Controlling the Flow of Data: Handshaking" on page 10-63 for more information about handshaking.

You can return the value of the CTS pin with the `PinStatus` property. You can specify the value of the RTS pin with the `RequestToSend` property. However, if `FlowControl` is `hardware`, and you specify a value for `RequestToSend`, that value might not be honored.

Note Although you might be able to configure your device for both hardware handshaking and software handshaking at the same time, MATLAB does not support this behavior.

Characteristics

Read only	Never
Data type	String

Values

`{none}`	No flow control is used.
`hardware`	Hardware flow control is used.
`software`	Software flow control is used.

See Also

Properties

`PinStatus`, `RequestToSend`

Purpose

Specify the size of the input buffer in bytes

Description

You configure `InputBufferSize` as the total number of bytes that can be stored in the input buffer during a read operation.

A read operation is terminated if the amount of data stored in the input buffer equals the `InputBufferSize` value. You can read text data with the `fgetl`, `fgets`, or `fscanf` functions. You can read binary data with the `fread` function.

You can configure `InputBufferSize` only when the serial port object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

If you configure `InputBufferSize` while there is data in the input buffer, that data is flushed.

Characteristics

Read only	While open
Data type	Double

Values

The default value is `512`.

See Also

Functions

`fclose`, `fgetl`, `fgets`, `fopen`, `fread`, `fscanf`

Properties

`Status`

Name

Purpose

Specify a descriptive name for the serial port object

Description

You configure `Name` to be a descriptive name for the serial port object.

When you create a serial port object, a descriptive name is automatically generated and stored in `Name`. This name is given by concatenating the word “Serial” with the serial port specified in the `serial` function. However, you can change the value of `Name` at any time.

The serial port is given by the `Port` property. If you modify this property value, then `Name` is automatically updated to reflect that change.

Characteristics

Read only	Never
Data type	String

Values

`Name` is automatically defined when the serial port object is created.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');
```

`s` is automatically assigned a descriptive name.

```
s.Name
ans =
Serial-COM1
```

See Also

Functions

`serial`

Purpose

Control access to serial port object

Description

The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the serial port objects created by their applications. When an object's `ObjectVisibility` property is set to `off`, `instrfind` does not return or delete that object.

Objects that are not visible are still valid. If you have access to the object (for example, from within the M-file that creates it), you can set and get its properties and pass it to any function that operates on serial port objects.

Characteristics

Usage	Any serial port object
Read only	Never
Data type	String

Values

`{on}`	Object is visible to `instrfind`.
`off`	Object is not visible from the command line (except by `instrfindall`).

Examples

The following statement creates a serial port object with its `ObjectVisibility` property set to `off`:

```
s = serial('COM1','ObjectVisibility','off');
instrfind
ans =
     []
```

However, because the hidden object is in the workspace (`s`), you can access it.

```
get(s,'ObjectVisibility')
ans =
 off
```

See Also

Functions

`instrfind`, `instrfindall`

Purpose

Specify the size of the output buffer in bytes

Description

You configure `OutputBufferSize` as the total number of bytes that can be stored in the output buffer during a write operation.

An error occurs if the output buffer cannot hold all the data to be written. You write text data with the `fprintf` function. You write binary data with the `fwrite` function.

You can configure `OutputBufferSize` only when the serial port object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	Double

Values

The default value is `512`.

See Also

Functions

`fprintf`, `fwrite`

Properties

`Status`

OutputEmptyFcn

Purpose

Specify the M-file callback function to execute when the output buffer is empty

Description

You configure `OutputEmptyFcn` to execute an M-file callback function when an output-empty event occurs. An output-empty event is generated when the last byte is sent from the output buffer to the device.

Note An output-empty event can be generated only for asynchronous write operations.

If the `RecordStatus` property value is `on`, and an output-empty event occurs, the record file records this information:

- The event type as `OutputEmpty`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Refer to "Creating and Executing Callback Functions" on page 10-57 to learn how to create a callback function.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

See Also

Functions

`record`

Properties

`RecordStatus`

Purpose

Specify the type of parity checking

Description

You can configure `Parity` to be `none`, `odd`, `even`, `mark`, or `space`. If `Parity` is `none`, parity checking is not performed and the parity bit is not transmitted. If `Parity` is `odd`, the number of mark bits (1s) in the data is counted, and the parity bit is asserted or unasserted to obtain an odd number of mark bits. If `Parity` is `even`, the number of mark bits in the data is counted, and the parity bit is asserted or unasserted to obtain an even number of mark bits. If `Parity` is `mark`, the parity bit is asserted. If `Parity` is `space`, the parity bit is unasserted.

Parity checking can detect errors of one bit only. An error in two bits might cause the data to have a seemingly valid parity, when in fact it is incorrect. Refer to "The Parity Bit" on page 10-14 for more information about parity checking.

In addition to the parity bit, the serial data format consists of a start bit, between five and eight data bits, and one or two stop bits. You specify the number of data bits with the `DataBits` property, and the number of stop bits with the `StopBits` property.

Characteristics

Read only	Never
Data type	String

Values

`{none}`	No parity checking
`odd`	Odd parity checking
`even`	Even parity checking
`mark`	Mark parity checking
`space`	Space parity checking

See Also

Properties

`DataBits`, `StopBits`

PinStatus

Purpose

Indicate the state of the CD, CTS, DSR, and RI pins

Description

`PinStatus` is a structure array that contains the fields `CarrierDetect`, `ClearToSend`, `DataSetReady` and `RingIndicator`. These fields indicate the state of the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) and Ring Indicator (RI) pins, respectively. Refer to "Serial Port Signals and Pin Assignments" on page 10-7 for more information about these pins.

`PinStatus` can be `on` or `off` for any of these fields. A value of `on` indicates the associated pin is asserted. A value of `off` indicates the associated pin is unasserted. A pin status event occurs when any of these pins changes its state. A pin status event executes the M-file specified by `PinStatusFcn`.

In normal usage, the Data Terminal Ready (DTR) and DSR pins work together, while the Request to Send (RTS) and CTS pins work together. You can specify the state of the DTR pin with the `DataTerminalReady` property. You can specify the state of the RTS pin with the `RequestToSend` property.

Refer to "Example — Connecting Two Modems" on page 10-61 for an example that uses `PinStatus`.

Characteristics

Read only	Always
Data type	Structure

Values

`off`	The associated pin is asserted.
`on`	The associated pin is asserted.

The default value is device dependent.

See Also

Properties

`DataTerminalReady`, `PinStatusFcn`, `RequestToSend`

Purpose

Specify the M-file callback function to execute when the CD, CTS, DSR, or RI pins change state

Description

You configure `PinStatusFcn` to execute an M-file callback function when a pin status event occurs. A pin status event occurs when the Carrier Detect (CD), Clear to Send (CTS), Data Set Ready (DSR) or Ring Indicator (RI) pin changes state. A serial port pin changes state when it is asserted or unasserted. Information about the state of these pins is recorded in the `PinStatus` property.

Note A pin status event can be generated at any time during the serial port session.

If the `RecordStatus` property value is `on`, and a pin status event occurs, the record file records this information:

- The event type as `PinStatus`
- The pin that changed its state, and the pin state as either `on` or `off`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Refer to “Creating and Executing Callback Functions” on page 10-57 to learn how to create a callback function.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

See Also

Functions

`record`

Properties

PinStatus, RecordStatus

Purpose

Specify the platform-specific serial port name

Description

You configure `Port` to be the name of a serial port on your platform. `Port` specifies the physical port associated with the object and the device.

When you create a serial port object, `Port` is automatically assigned the port name specified for the `serial` function.

You can configure `Port` only when the object is disconnected from the device. You disconnect an object with the `fclose` function. A disconnected object has a `Status` property value of `closed`.

Characteristics

Read only	While open
Data type	String

Values

The `Port` value is determined when the serial port object is created.

Example

Suppose you create a serial port object associated with serial port COM1.

```
s = serial('COM1');
```

The value of the `Port` property is `COM1`.

```
s.Port
ans =
COM1
```

See Also

Functions

`fclose`, `serial`

Properties

`Name`, `Status`

ReadAsyncMode

Purpose

Specify whether an asynchronous read operation is continuous or manual

Description

You can configure `ReadAsyncMode` to be `continuous` or `manual`. If `ReadAsyncMode` is `continuous`, the serial port object continuously queries the device to determine if data is available to be read. If data is available, it is automatically read and stored in the input buffer. If issued, the `readasync` function is ignored.

If `ReadAsyncMode` is `manual`, the object does not query the device to determine if data is available to be read. Instead, you must manually issue the `readasync` function to perform an asynchronous read operation. Because `readasync` checks for the terminator, this function can be slow. To increase speed, configure `ReadAsyncMode` to `continuous`.

Note If the device is ready to transmit data, it will do so regardless of the `ReadAsyncMode` value. Therefore, if `ReadAsyncMode` is `manual` and a read operation is not in progress, data might be lost. To guarantee that all transmitted data is stored in the input buffer, you should configure `ReadAsyncMode` to `continuous`.

You can determine the amount of data available in the input buffer with the `BytesAvailable` property. For either `ReadAsyncMode` value, you can bring data into the MATLAB workspace with one of the synchronous read functions such as `fscanf`, `fgetl`, `fgets`, or `fread`.

Characteristics

Read only	Never
Data type	String

Values

`{continuous}`	Continuously query the device to determine if data is available to be read.
`manual`	Manually read data from the device using the `readasync` function.

See Also

Functions

`fgetl`, `fgets`, `fread`, `fscanf`, `readasync`

Properties

`BytesAvailable`, `InputBufferSize`

RecordDetail

Purpose

Specify the amount of information saved to a record file

Description

You can configure `RecordDetail` to be `compact` or `verbose`. If `RecordDetail` is `compact`, the number of values written to the device, the number of values read from the device, the data type of the values, and event information are saved to the record file. If `RecordDetail` is `verbose`, the data written to the device, and the data read from the device are also saved to the record file.

The event information saved to a record file is shown in the table, Event Information on page 10-55. The verbose record file structure is shown in “Example: Recording Information to Disk” on page 10-69.

Characteristics

Read only	Never
Data type	String

Values

`{compact}`	The number of values written to the device, the number of values read from the device, the data type of the values, and event information are saved to the record file.
`verbose`	The data written to the device, and the data read from the device are also saved to the record file.

See Also

Functions

`record`

Properties

`RecordMode`, `RecordName`, `RecordStatus`

Purpose

Specify whether data and event information are saved to one record file or to multiple record files

Description

You can configure `RecordMode` to be `overwrite`, `append`, or `index`. If `RecordMode` is `overwrite`, the record file is overwritten each time recording is initiated. If `RecordMode` is `append`, data is appended to the record file each time recording is initiated. If `RecordMode` is `index`, a different record file is created each time recording is initiated, each with an indexed filename.

You can configure `RecordMode` only when the object is not recording. You terminate recording with the `record` function. A object that is not recording has a `RecordStatus` property value of `off`.

You specify the record filename with the `RecordName` property. The indexed filename follows a prescribed set of rules. Refer to "Specifying a Filename" on page 10-68 for a description of these rules.

Characteristics

Read only	While recording
Data type	String

Values

`{overwrite}`	The record file is overwritten.
`append`	Data is appended to an existing record file.
`index`	A different record file is created, each with an indexed filename.

Example

Suppose you create the serial port object `s` associated with the serial port COM1.

```
s = serial('COM1');
fopen(s)
```

Specify the record filename with the `RecordName` property, configure `RecordMode` to `index`, and initiate recording.

```
s.RecordName = 'MyRecord.txt';
s.RecordMode = 'index';
record(s)
```

The record filename is automatically updated with an indexed filename after recording is turned off.

```
record(s,'off')
s.RecordName
ans =
MyRecord01.txt
```

Disconnect s from the peripheral device, remove s from memory, and remove s from the MATLAB workspace.

```
fclose(s)
delete(s)
clear s
```

See Also

Functions

`record`

Properties

`RecordDetail`, `RecordName`, `RecordStatus`

Purpose

Specify the name of the record file

Description

You configure `RecordName` to be the name of the record file. You can specify any value for `RecordName` - including a directory path - provided the filename is supported by your operating system.

MATLAB supports any filename supported by your operating system. However, if you access the file through MATLAB, you might need to specify the filename using single quotes. For example, suppose you name the record file `My Record.txt`. To type this file at the MATLAB command line, you must include the name in quotes.

```
type('My Record.txt')
```

You can specify whether data and event information are saved to one disk file or to multiple disk files with the `RecordMode` property. If `RecordMode` is `index`, the filename follows a prescribed set of rules. Refer to “Specifying a Filename” on page 10-68 for a description of these rules.

You can configure `RecordName` only when the object is not recording. You terminate recording with the `record` function. An object that is not recording has a `RecordStatus` property value of `off`.

Characteristics

Read only	While recording
Data type	String

Values

The default record filename is `record.txt`.

See Also

Functions

`record`

Properties

`RecordDetail`, `RecordMode`, `RecordStatus`

RecordStatus

Purpose

Indicate if data and event information are saved to a record file

Description

You can configure `RecordStatus` to be `off` or `on` with the `record` function. If `RecordStatus` is `off`, then data and event information are not saved to a record file. If `RecordStatus` is `on`, then data and event information are saved to the record file specified by `RecordName`.

Use the `record` function to initiate or complete recording. `RecordStatus` is automatically configured to reflect the recording state.

For more information about recording to a disk file, refer to "Debugging: Recording Information to Disk" on page 10-66.

Characteristics

Read only	Always
Data type	String

Values

`{off}`	Data and event information are not written to a record file.
`on`	Data and event information are written to a record file.

See Also

Functions

`record`

Properties

`RecordDetail`, `RecordMode`, `RecordName`

Purpose

Specify the state of the RTS pin

Description

You can configure `RequestToSend` to be `on` or `off`. If `RequestToSend` is `on`, the Request to Send (RTS) pin is asserted. If `RequestToSend` is `off`, the RTS pin is unasserted.

In normal usage, the RTS and Clear to Send (CTS) pins work together, and are used as standard handshaking pins for data transfer. In this case, RTS and CTS are automatically managed by the DTE and DCE. However, there is nothing in the RS-232 standard that requires the RTS pin must be used in any specific way. Therefore, if you manually configure the `RequestToSend` value, it is probably for nonstandard operations.

If your device does not use hardware handshaking in the standard way, and you need to manually configure `RequestToSend`, configure the `FlowControl` property to `none`. Otherwise, the `RequestToSend` value that you specify might not be honored. Refer to your device documentation to determine its specific pin behavior.

You can return the value of the CTS pin with the `PinStatus` property. Handshaking is described in "Controlling the Flow of Data: Handshaking" on page 10-63.

Characteristics

Read only	Never
Data type	String

Values

`{on}`	The RTS pin is asserted.
`off`	The RTS pin is unasserted.

See Also

Properties

`FlowControl`, `PinStatus`

Status

Purpose

Indicate if the serial port object is connected to the device

Description

`Status` can be `open` or `closed`. If `Status` is `closed`, the serial port object is not connected to the device. If `Status` is `open`, the serial port object is connected to the device.

Before you can write or read data, you must connect the serial port object to the device with the `fopen` function. Use the `fclose` function to disconnect a serial port object from the device.

Characteristics

Read only	Always
Data type	String

Values

`{closed}`	The serial port object is not connected to the device.
`open`	The serial port object is connected to the device.

See Also

Functions

`fclose`, `fopen`

Purpose

Specify the number of bits used to indicate the end of a byte

Description

You can configure `StopBits` to be `1`, `1.5`, or `2`. If `StopBits` is `1`, one stop bit is used to indicate the end of data transmission. If `StopBits` is `2`, two stop bits are used to indicate the end of data transmission. If `StopBits` is `1.5`, the stop bit is transferred for 150% of the normal time used to transfer one bit.

Note Both the computer and the peripheral device must be configured to transmit the same number of stop bits.

In addition to the stop bits, the serial data format consists of a start bit, between five and eight data bits, and possibly a parity bit. You specify the number of data bits with the `DataBits` property, and the type of parity checking with the `Parity` property.

Characteristics

Read only	Never
Data type	Double

Values

`{1}`	One stop bit is transmitted to indicate the end of a byte.
`1.5`	The stop bit is transferred for 150% of the normal time used to transfer one bit.
`2`	Two stop bits are transmitted to indicate the end of a byte.

See Also

Properties

`DataBits`, `Parity`

Tag

Purpose

Specify a label to associate with a serial port object

Description

You configure `Tag` to be a string value that uniquely identifies a serial port object.

`Tag` is particularly useful when constructing programs that would otherwise need to define the serial port object as a global variable, or pass the object as an argument between callback routines.

You can return the serial port object with the `instrfind` function by specifying the `Tag` property value.

Characteristics

Read only	Never
Data type	String

Values

The default value is an empty string.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');
fopen(s)
```

You can assign `s` a unique label using `Tag`.

```
set(s,'Tag','MySerialObj')
```

You can access `s` in the MATLAB workspace or in an M-file using the `instrfind` function and the `Tag` property value.

```
s1 = instrfind('Tag','MySerialObj');
```

See Also

Functions

`instrfind`

Purpose

Specify the terminator character

Description

You can configure `Terminator` to an integer value ranging from 0 to 127, which represents the ASCII code for the character, or you can configure `Terminator` to the ASCII character. For example, to configure `Terminator` to a carriage return, specify the value to be `CR` or `13`. To configure `Terminator` to a linefeed, specify the value to be `LF` or `10`. You can also set `Terminator` to `CR/LF` or `LF/CR`. If `Terminator` is `CR/LF`, the terminator is a carriage return followed by a line feed. If Terminator is `LF/CR`, the terminator is a linefeed followed by a carriage return. Note that there are no integer equivalents for these two values. Additionally, you can set `Terminator` to a 1-by-2 cell array. The first element of the cell is the read terminator and the second element of the cell array is the write terminator.

When performing a write operation using the `fprintf` function, all occurrences of `\n` are replaced with the `Terminator` property value. Note that `%s\n` is the default format for `fprintf`. A read operation with `fgetl`, `fgets`, or `fscanf` completes when the `Terminator` value is read. The terminator is ignored for binary operations.

You can also use the terminator to generate a bytes-available event when the `BytesAvailableFcnMode` is set to `terminator`.

Characteristics

Read only	Never
Data type	String

Values

An integer value ranging from 0 to 127, or the equivalent ASCII character. `CR/LF` and `LF/CR` are also accepted values. You specify different read and write terminators as a 1-by-2 cell array.

See Also

Functions

`fgetl`, `fgets`, `fprintf`, `fscanf`

Properties

`BytesAvailableFcnMode`

Purpose

Specify the waiting time to complete a read or write operation

Description

You configure `Timeout` to be the maximum time (in seconds) to wait to complete a read or write operation.

If a time-out occurs, the read or write operation aborts. Additionally, if a time-out occurs during an asynchronous read or write operation, then:

- An error event is generated.
- The M-file callback function specified for `ErrorFcn` is executed.

Characteristics

Read only	Never
Data type	Double

Values

The default value is `10` seconds.

See Also

Properties

`ErrorFcn`

TimerFcn

Purpose

Specify the M-file callback function to execute when a predefined period of time passes.

Description

You configure `TimerFcn` to execute an M-file callback function when a timer event occurs. A timer event occurs when the time specified by the `TimerPeriod` property passes. Time is measured relative to when the serial port object is connected to the device with `fopen`.

Note A timer event can be generated at any time during the serial port session.

If the `RecordStatus` property value is `on`, and a timer event occurs, the record file records this information:

- The event type as `Timer`
- The time the event occurred using the format day-month-year hour:minute:second:millisecond

Some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value is too small.

Refer to "Creating and Executing Callback Functions" on page 10-57 to learn how to create a callback function.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is an empty string.

See Also

Functions

`fopen`, `record`

Properties

RecordStatus, TimerPeriod

TimerPeriod

Purpose

Specify the period of time between timer events

Description

`TimerPeriod` specifies the time, in seconds, that must pass before the callback function specified for `TimerFcn` is called. Time is measured relative to when the serial port object is connected to the device with `fopen`.

Some timer events might not be processed if your system is significantly slowed or if the `TimerPeriod` value is too small.

Characteristics

Read only	Never
Data type	Callback function

Values

The default value is 1 second. The minimum value is 0.01 second.

See Also

Functions

`fopen`

Properties

`TimerFcn`

Purpose

Indicate if an asynchronous read or write operation is in progress

Description

`TransferStatus` can be `idle`, `read`, `write`, or `read&write`. If `TransferStatus` is `idle`, no asynchronous read or write operations are in progress. If `TransferStatus` is `read`, an asynchronous read operation is in progress. If `TransferStatus` is `write`, an asynchronous write operation is in progress. If `TransferStatus` is `read&write`, both an asynchronous read and an asynchronous write operation are in progress.

You can write data asynchronously using the `fprintf` or `fwrite` functions. You can read data asynchronously using the `readasync` function, or by configuring the `ReadAsyncMode` property to `continuous`. While `readasync` is executing, `TransferStatus` might indicate that data is being read even though data is not filling the input buffer. If `ReadAsyncMode` is `continuous`, `TransferStatus` indicates that data is being read only when data is actually filling the input buffer.

You can execute an asynchronous read and an asynchronous write operation simultaneously because serial ports have separate read and write pins. Refer to "Writing and Reading Data" on page 10-32 for more information about synchronous and asynchronous read and write operations.

Characteristics

Read only	Always
Data type	String

Values

`{idle}`	No asynchronous operations are in progress.
`read`	An asynchronous read operation is in progress.
`write`	An asynchronous write operation is in progress.
`read&write`	Asynchronous read and write operations are in progress.

See Also

Functions

`fprintf`, `fwrite`, `readasync`

Properties

`ReadAsyncMode`

Purpose

Indicate the object type

Description

`Type` indicates the type of the object. `Type` is automatically defined after the serial port object is created with the `serial` function. The `Type` value is always `serial`.

Characteristics

Read only	Always
Data type	String

Values

`Type` is always `serial`. This value is automatically defined when the serial port object is created.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');
```

The value of the `Type` property is `serial`, which is the object class.

```
s.Type
ans =
serial
```

You can also display the object class with the `whos` command.

```
Name      Size          Bytes  Class

  s         1x1             644  serial object

Grand total is 18 elements using 644 bytes
```

See Also

Functions

`serial`

UserData

Purpose

Specify data that you want to associate with a serial port object

Description

You configure `UserData` to store data that you want to associate with a serial port object. The object does not use this data directly, but you can access it using the `get` function or the dot notation.

Characteristics

Read only	Never
Data type	Any type

Values

The default value is an empty vector.

Example

Suppose you create the serial port object associated with the serial port COM1.

```
s = serial('COM1');
```

You can associate data with `s` by storing it in `UserData`.

```
coeff.a = 1.0;
coeff.b = -1.25;
s.UserData = coeff;
```

Purpose

Indicate the total number of values read from the device

Description

`ValuesReceived` indicates the total number of values read from the device. The value is updated after each successful read operation, and is set to `0` after the `fopen` function is issued. If the terminator is read from the device, then this value is reflected by `ValuesReceived`.

If you are reading data asynchronously, use the `BytesAvailable` property to return the number of bytes currently available in the input buffer.

When performing a read operation, the received data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes. Refer to "Bytes Versus Values" on page 10-12 for more information about bytes and values.

Characteristics

Read only	Always
Data type	Double

Values

The default value is `0`.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');
fopen(s)
```

If you write the `RS232?` command, and read back the response using `fscanf`, `ValuesReceived` is `17` because the instrument is configured to send the `LF` terminator.

```
fprintf(s,'RS232?')
out = fscanf(s)
out =
9600;0;0;NONE;LF
s.ValuesReceived
```

```
ans =
    17
```

See Also

Functions

`fopen`

Properties

`BytesAvailable`

Purpose

Indicate the total number of values written to the device

Description

`ValuesSent` indicates the total number of values written to the device. The value is updated after each successful write operation, and is set to `0` after the `fopen` function is issued. If you are writing the terminator, `ValuesSent` reflects this value.

If you are writing data asynchronously, use the `BytesToOutput` property to return the number of bytes currently in the output buffer.

When performing a write operation, the transmitted data is represented by values rather than bytes. A value consists of one or more bytes. For example, one `uint32` value consists of four bytes. Refer to "Bytes Versus Values" on page 10-12 for more information about bytes and values.

Characteristics

Read only	Always
Data type	Double

Values

The default value is `0`.

Example

Suppose you create a serial port object associated with the serial port COM1.

```
s = serial('COM1');
fopen(s)
```

If you write the `*IDN?` command using the `fprintf` function, `ValuesSent` is `6` because the default data format is `%s\n`, and the terminator was written.

```
fprintf(s,'*IDN?')
s.ValuesSent
ans =
    6
```

See Also

Functions

fopen

Properties

BytesToOutput

A

Examples

Use this list to find examples in the documentation.

Importing and Exporting Data

MATLAB Interface to Generic DLLs

Calling C and Fortran Programs from MATLAB

Creating C Language MEX-Files

Creating Fortran MEX-Files

Calling MATLAB from C and Fortran Programs

Calling Java from MATLAB

COM Support

Serial Port I/O

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

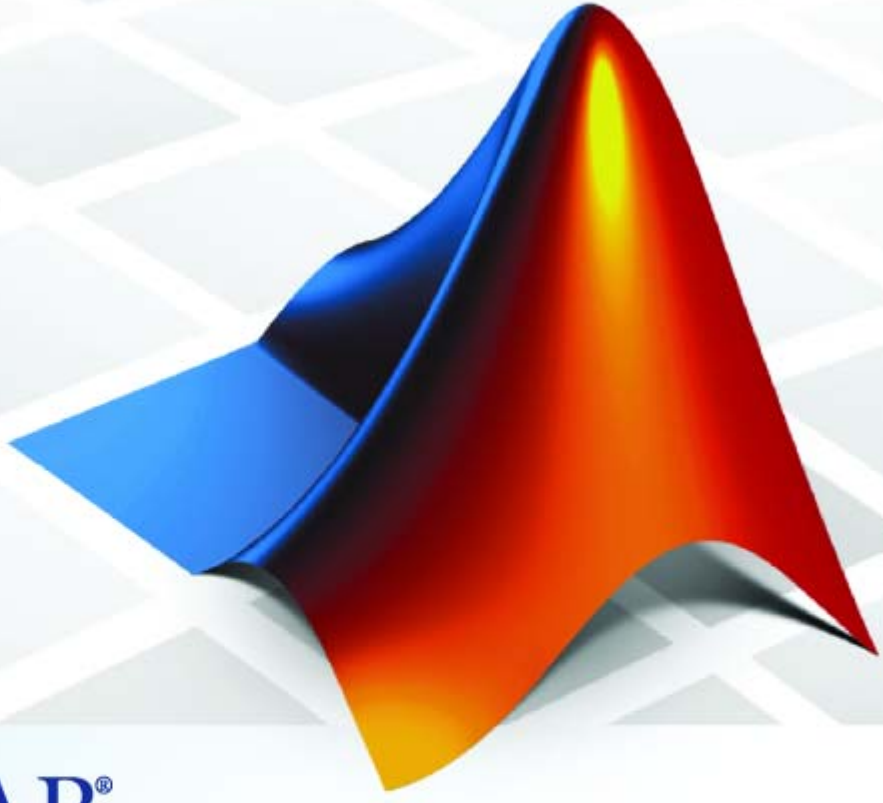
V

W

X

Y

MATLAB® 7
Graphics



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Graphics

Trademarks

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, SimBiology, SimHydraulics, SimEvents, and xPC TargetBox are registered trademarks and The MathWorks, the L-shaped membrane logo, Embedded MATLAB, and PolySpace are 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

March 2006	Online only	New for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for MATLAB 7.5 (Release 2007b) This publication was previously part of the Using MATLAB Graphics User Guide.

Contents

1 Plots and Plotting Tools

2 Data Exploration Tools

3 Annotating Graphs

4 Basic Plotting Commands

5 Creating Specialized Plots

6 Displaying Bit-Mapped Images

7 Printing and Exporting

8 Handle Graphics Objects

9 Figure Properties

10 Axes Properties

1

Plots and Plotting Tools

If you are viewing this document in the Help browser, you can watch the Interactive Plot Creation with the Plot Tools video demo for an overview of the major functionality. It covers much of the material presented in the following sections:

Figures, Plots, and Graphs

In this section...

Graphing In MATLAB

MATLAB® offers you a variety of data plotting functions plus a set of GUI tools to create, and modify graphic displays. The GUI tools afford most of the control over graphic properties and options that typed commands such as `annotate`, `get`, and `set` provide.

A *figure* is a MATLAB window that contains graphic displays (usually data plots) and UI components. You create figures explicitly with the `figure` function, and implicitly whenever you plot graphics and no figure is active. By default, figure windows are resizable and include pull-down menus and toolbars.

A *plot* is any graphic display you can create within a figure window. Plots can display tabular data, geometric objects, surface and image objects, and annotations such as titles, legends, and colorbars. Figures can contain any number of plots. Each plot is created within a 2-D or a 3-D data space called an *axes*. You can explicitly create axes with the `axes` or `subplot` functions.

A *graph* is a plot of *data* within a 2-D or 3-D axes. Most plots made with MATLAB are therefore graphs. When you graph a one-dimensional variable (e.g., `rand(100,1)`), MATLAB assigns the indices of the data vector (in this case `1:100`) as *x*-values, and plots the data vector as *y*-values. Some types of graphs can display more than one variable at a time, others cannot.

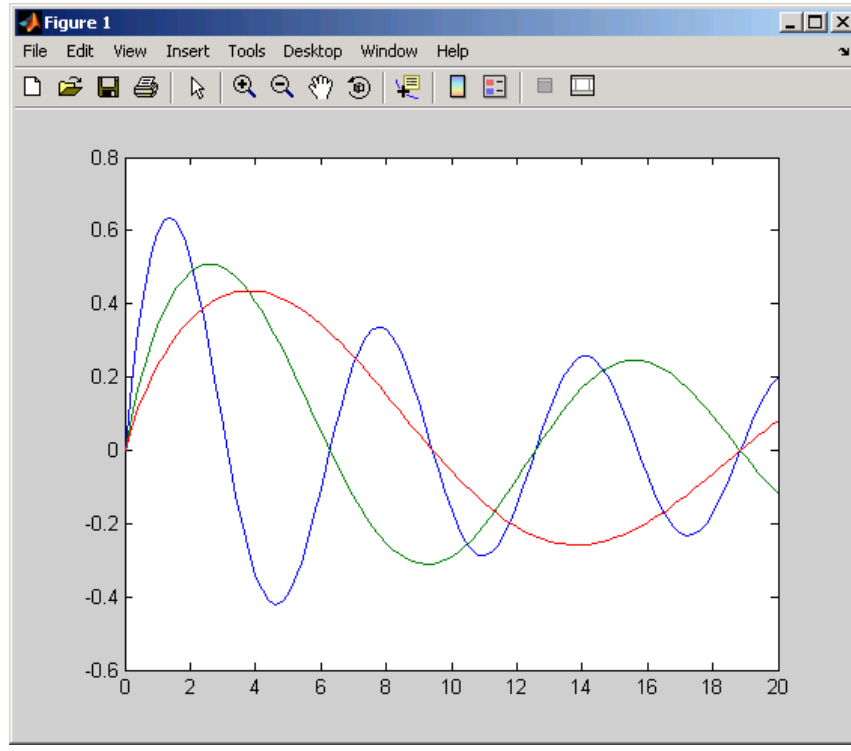
The contents and varieties of figures, plots and graphs that MATLAB can make are explained in the following sections.

Anatomy of a Graph

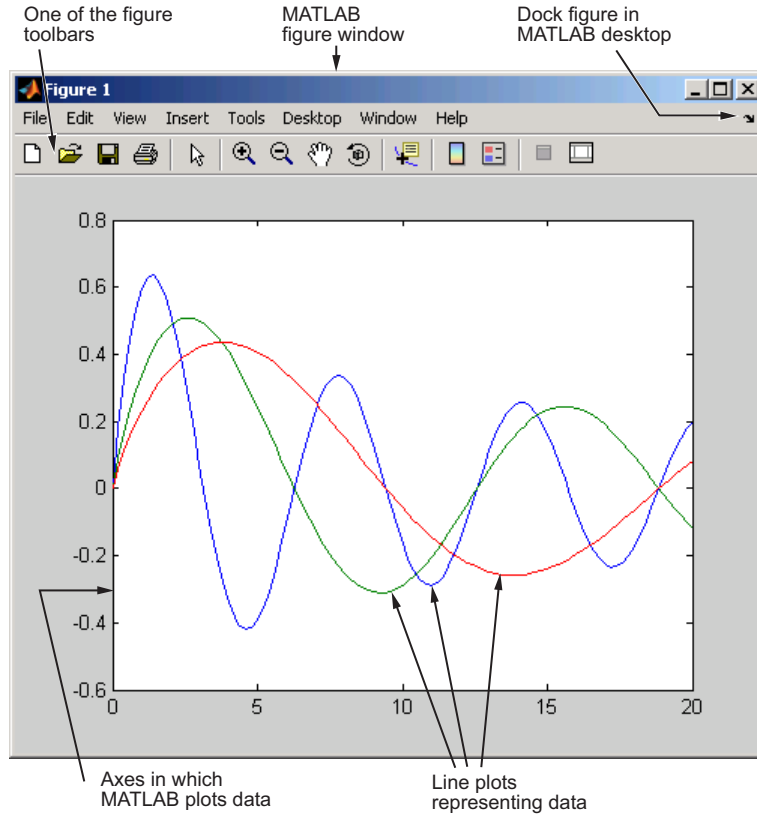
MATLAB plotting functions and tools direct their output to a figure window. Each figure is a separate window that you can dock in the desktop, and collect together with other plots in a *Figure Group*. To illustrate the basic components of a graph, execute the following M-code to create a plot of a family of sine curves:

```
x = [0:.2:20];
y = sin(x)./sqrt(x+1);
y(2,:) = sin(x/2)./sqrt(x+1);
y(3,:) = sin(x/3)./sqrt(x+1);
plot(x,y)
```

The resulting figure contains a 2-D set of axes and looks like this:



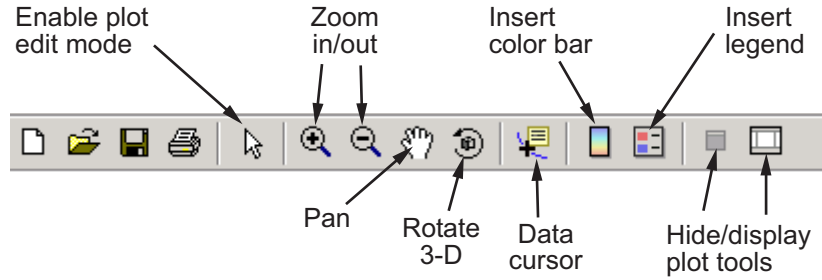
Some of the components and tools of figure windows are called out below:



MATLAB uses a default line style and color to distinguish the data sets plotted in the graph. You can change the appearance of these graphic components or add annotations to the graph to present your data in a particular way.

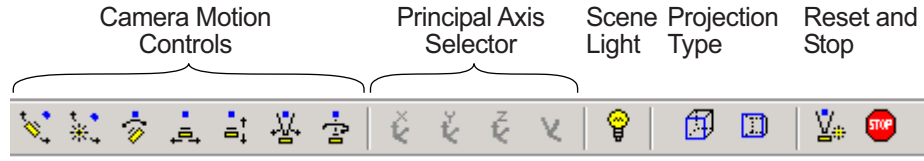
Figure Toolbars

Figure toolbars provide shortcuts to access commonly used features. These include operations such as saving and printing, plus tools for interactive zooming, panning, rotating, querying, and editing plots. The following picture shows the features available from this toolbar.

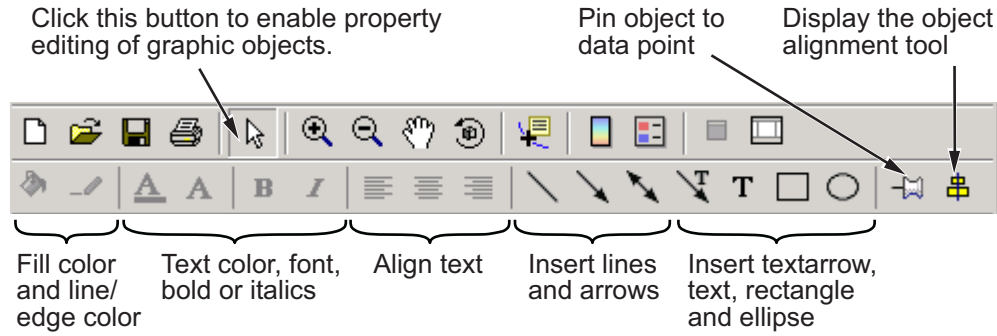


Note that you can enable two other toolbars from the **View** menu:

- Camera Toolbar — Use for manipulating 3-D views. See “View Control with the Camera Toolbar” in the MATLAB 3-D Visualization documentation for more information.



- Plot Edit Toolbar — Use for annotation and setting object properties. See “Annotation Tools on the Plot Edit Toolbar” on page 3-3 for more information.

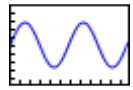
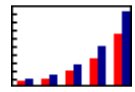
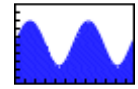
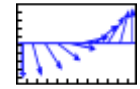
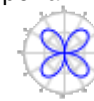
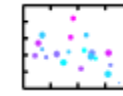


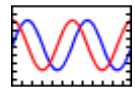
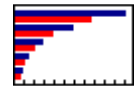
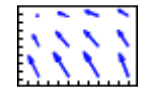
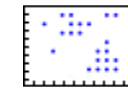
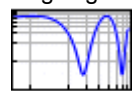
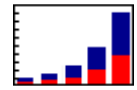
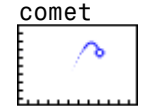
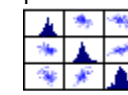
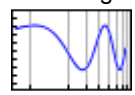
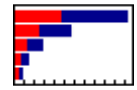
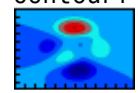
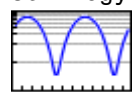
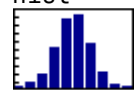
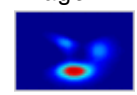
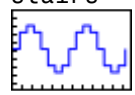
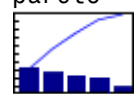
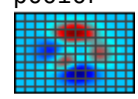
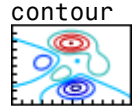
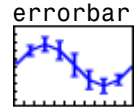
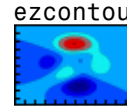
Types of Plots Available in MATLAB

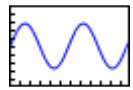
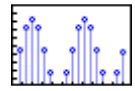
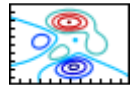
MATLAB can construct a wide variety of 2-D and 3-D plots without any programming required on your part. The following two tables classify and illustrate most of the kinds of plots you can create. They include line, bar, area, direction and vector field, radial, and scatter graphs. They also include 2-D and 3-D functions that generate and plot geometric shapes and objects. Most 2-D plots have 3-D analogs, and there are a variety of volumetric displays for 3-D solids and vector fields. Plot types that begin with "ez" (such as `exsurf`) are convenience functions that can plot arguments given as functions.

Two-Dimensional Plotting Functions

The table below shows 2-D plot functions available in MATLAB. If you are reading this online, you can click any icon to see the documentation for that function. Techniques for using many of the functions are also discussed in later sections of this document.

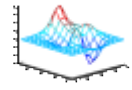
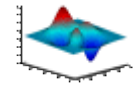
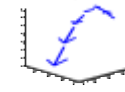
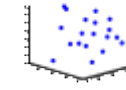
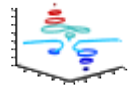
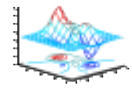
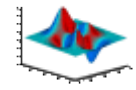
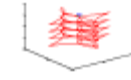
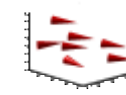
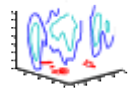
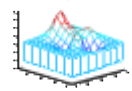
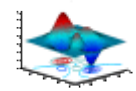
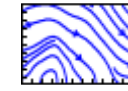
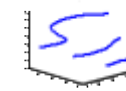
Line Graphs	Bar Graphs	Area Graphs	Direction Graphs	Radial Graphs	Scatter Graphs
`plot` 	`bar` (grouped) 	`area` 	`feather` 	`polar` 	`scatter` 

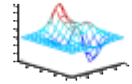
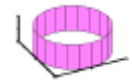
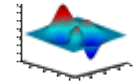
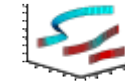
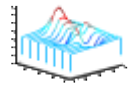
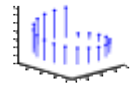
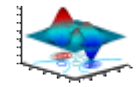
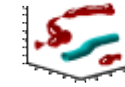
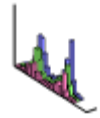
Line Graphs	Bar Graphs	Area Graphs	Direction Graphs	Radial Graphs	Scatter Graphs
plotyy 	barh (grouped) 	pie 	quiver 	rose 	spy 
loglog 	bar (stacked) 	fill 	comet 	compass 	plotmatrix 
semilogx 	barh (stacked) 	contourf 		ezpolar 	
semilogy 	hist 	image 			
stairs 	pareto 	pcolor 			
contour 	errorbar 	ezcontourf 			

Line Graphs	Bar Graphs	Area Graphs	Direction Graphs	Radial Graphs	Scatter Graphs
ezplot 	stem 				
ezcontour 					

Three-Dimensional Plotting Functions

The table below shows 3-D plot functions available in MATLAB. It includes functions that generate 3-D data (`cylinder`, `ellipsoid`, `sphere`), but most plot either arrays of data or functions. If you are reading this online, you can click any picture in the table to see the documentation for that function. For information about and examples of using 3-D plotting functions, see "Creating 3-D Graphs" in the 3-D Visualization documentation.

Line Graphs	Mesh Graphs and Bar Graphs	Area Graphs and Constructive Objects	Surface Graphs	Direction Graphs	Volumetric Graphs
plot3 	mesh 	pie3 	surf 	quiver3 	scatter3 
contour3 	meshc 	fill3 	surfl 	comet3 	coneplot 
contourslice 	meshz 	patch 	surfc 	streamslice 	streamline 

Line Graphs	Mesh Graphs and Bar Graphs	Area Graphs and Constructive Objects	Surface Graphs	Direction Graphs	Volumetric Graphs
ezplot3 	ezmesh 	cylinder 	ezsurf 		streamribbon 
waterfall 	stem3 	ellipsoid 	ezsurfc 		streamtube 
	bar3 	sphere 			
	bar3h 				

Choosing a Plot Type with the Plot Catalog

Most of the plotting functions shown in the previous tables are accessible through the **Figure Palette**, one of the Plot Tools you can access via the figure window **View** menu. When the Figure Palette is active and you select one, two or more variables listed within it, you can generate a plot of any appropriate type by right-clicking and selecting a plot type from the context menu that appears. The lowest item on that menu is **More Plots**. When you select **More Plots**, the *Plot Catalog* opens for you to browse through all plot types and generate one of them, either to display the variables you selected in the Figure Palette or a MATLAB expression you can specify in the Plot Catalog window. For more information, see “The Figure Palette” on page 1-18.

Plotting Tools — Interactive Plotting

In this section...

What Are Plotting Tools?

The modular, interactive plotting environment of MATLAB called *plotting tools* enables you to

- Create various type of graphs
- Select variables to plot directly from a workspace browser
- Easily create and manipulate subplots in the figure
- Add annotations such as arrows, lines, and text
- Set properties on graphics objects

You can open and configure plotting tools in many ways. To create a figure with the plotting tools attached, use the `plottools` command. You can also start the plotting tools from the figure toolbar by clicking the **Show Plot Tools** icon  .

Remove the plotting tools from a figure using the Hide Plot Tools icon  .

You can display the three basic plotting tools from the **View** menu by selecting **Figure Palette**, **Plot Browser**, or **Property Editor** .

The next section describes the individual components making up the plotting tools.

Plotting Tools Interface Overview

The Plotting Tools interface includes three panels that are associated with a figure.

- Figure Palette — Use to create and arrange subplot axes, view and plot workspace variables, and add annotations. Display the Figure Palette using the `figurepalette` command.
- Plot Browser — Use to select and control the visibility of the axes or graphic objects plotted in the figure. You can also add data to any selected axes by clicking the **Add Data** button. Display the Plot Browser using the `plotbrowser` command.
- Property Editor — Use to set common properties of the selected object. You can also open the Property Editor using the `propertyeditor` command. In the Property Editor you can click the **More Properties** button to display the Property Inspector, a GUI that displays most object properties and allows you to change any property's value (unless it is read-only). See "Accessing Object Properties with the Property Inspector" on page 1-29 for details.

Activating Plotting Tools

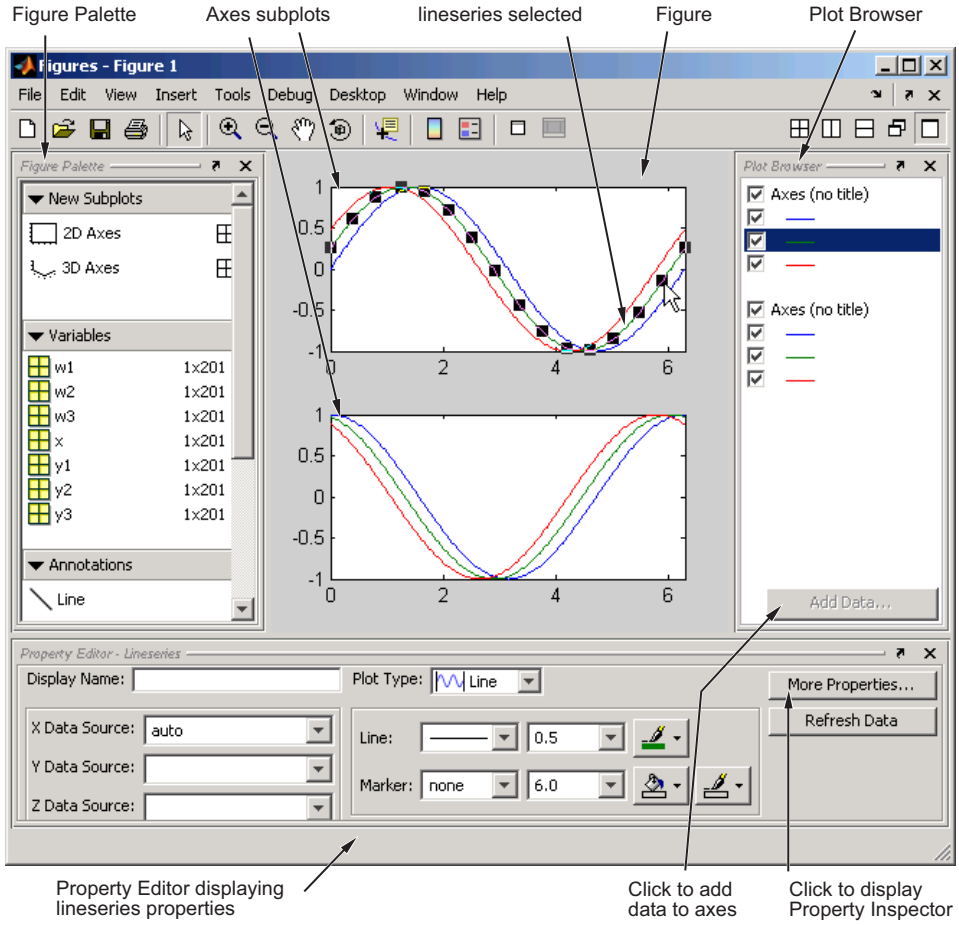
The example illustrated below shows the plotting tools attached to a figure containing two subplots of lineseries data. The code to produce the graphs is

```
x = 0:pi/100:2*pi;
y1 = sin(x);
y2 = sin(x+.25);
y3 = sin(x+.5);
subplot(2,1,1);
plot(x,y1,x,y2,x,y3);
axis tight;
w1 = cos(x);
w2 = cos(x+.25);
w3 = cos(x+.5);
subplot(2,1,2);
plot(x,w1,x,w2,x,w3);
axis tight;
```

You summon the plotting tools, either by selecting **Figure Palette**, **Plot Browser**, and **Property Editor** from the figure's **View** menu, or by typing

```
plottools
```

in the Command Window. Typing `plottools` or `plottools on` restores the configuration of tools the last time you were using them; use the View menu to show the ones you need and hide the ones you do not, and the mouse to dock and undock them. The default configuration of plotting tools is shown below. Note that MATLAB remembers the current arrangement of plot tools — whether they are visible or not — each time you exit; if you want to revert to the default configuration you need to restore the arrangement shown below manually.



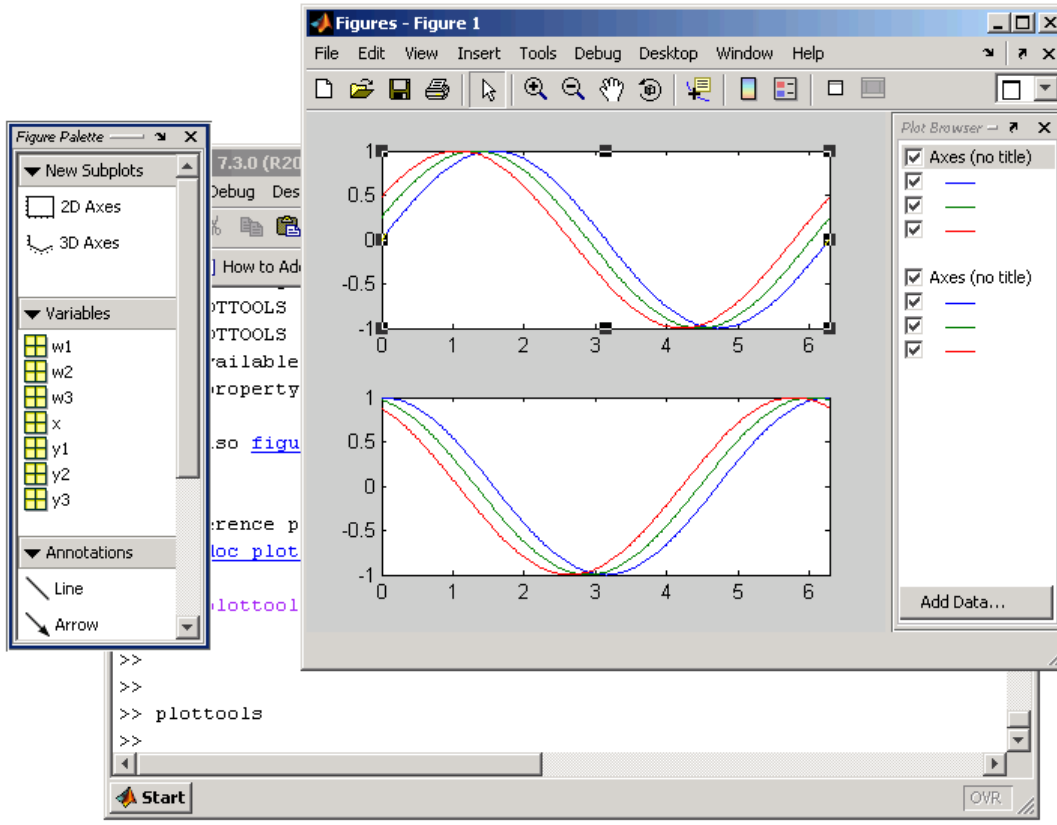
Managing Plotting Tools

Each of the plotting tools shown above can be docked or undocked from its figure, or dismissed by clicking the **x** at the right end of its titlebar. If you dismiss a tool and want it back again, you can raise it from the View menu or by typing one of several commands. For instance, if you had undocked, and then dismissed the figure palette, you could type either

```
plottools( 'on', 'figurepalette') or
```

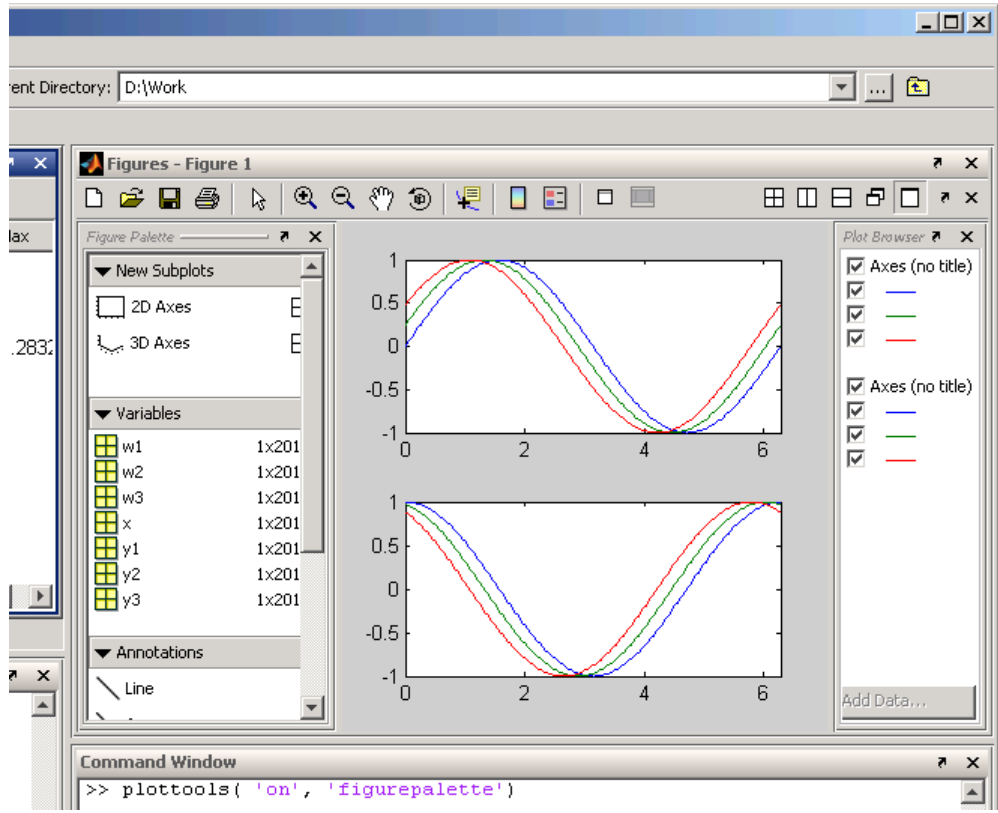
```
figurepalette
```

Your desktop configuration might then look like this:



Figures Groups. When you activate any plot tool for a figure (or dock the figure in the desktop), the figure becomes part of a Figures group. Figures groups are desktop containers that you can dock s in your desktop. Individual figures are not dockable except within the Figures group container. If you create subsequent figures, they will also dock in the Figures group, where they can be panelled or overlapped for viewing. A row of tabs appears along the bottom, one for each figure in the group.

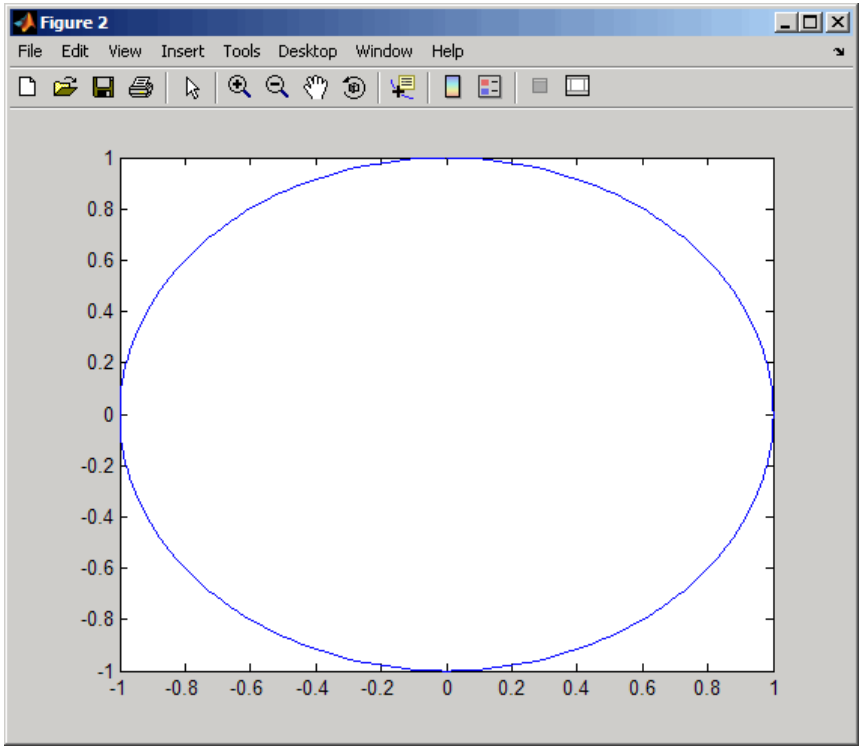
When you dock a plot tool in a figure group and then dock the figure group in the desktop, the tool is included in that section of the desktop as well, as the following illustration shows:



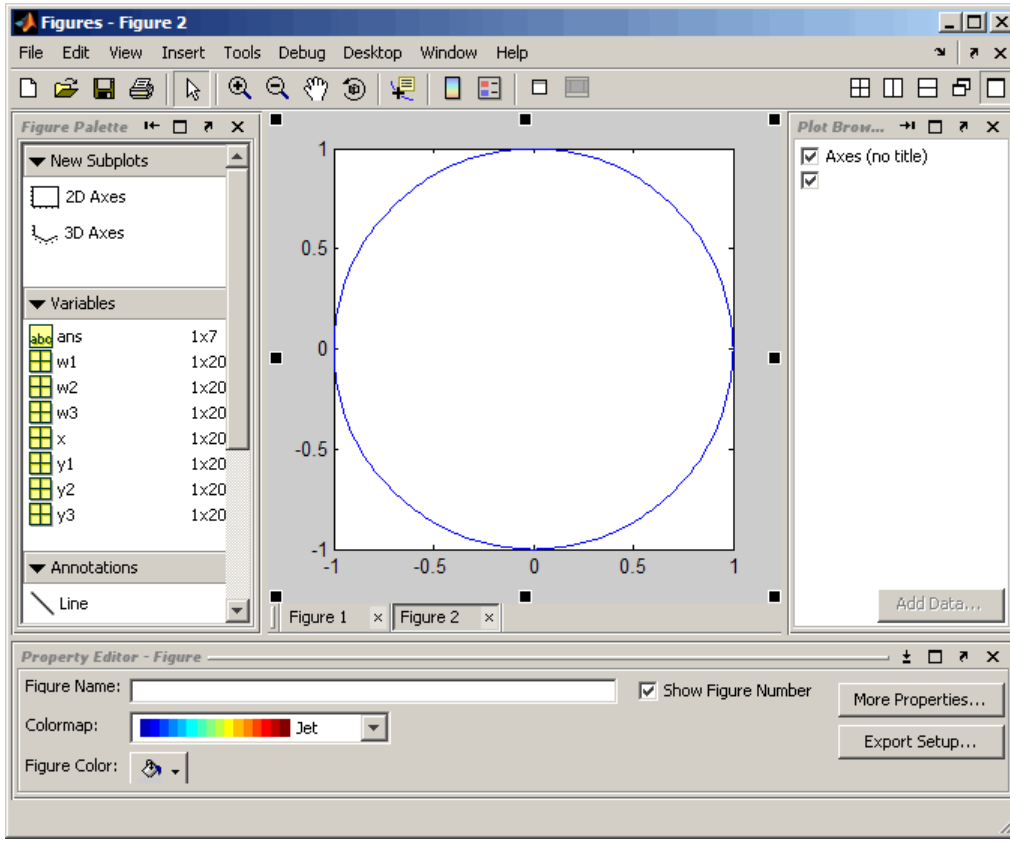
Working with Multiple Figures. When you create a new figure and plot into it, It is created without plotting tools enabled, even if another figure already has them open:

```
figure
plot(y1,w1)
```

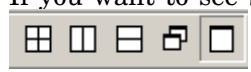
This generates a freestanding plot, like this:

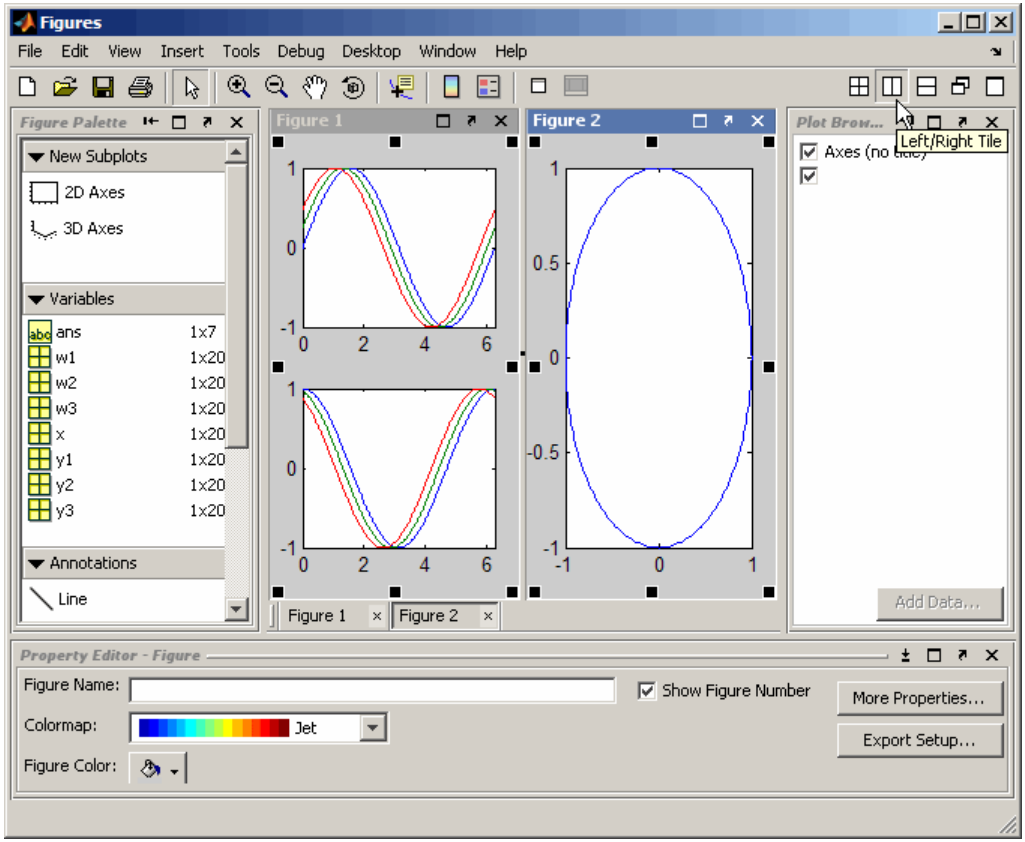


If you then open the plotting tools for the figure by clicking the Open Plotting Tools icon  , the figure docks in the figure window:



The new figure might seem to disappear if the Figures window is hidden, but it will overlay the existing plot within that window (it does not replace it). You can switch between the two figures by clicking the tabs at the bottom of the figure area. Be aware that clicking the x on the right side of a figure's tab deletes that figure entirely, without asking for confirmation.

If you want to see both figures at once, use the Tiling Palette  at the upper right corner of the Figures window to arrange the figures. For example, clicking the Left/Right tiling tool lays out the two figures side by side:



As the above illustrations shows, plot objects can be selected in both figures, but only one figure has focus at a time.

The Figure Palette

The Figure Palette contains three panels. Select the panel you want to view by clicking the respective button, which twists down the panel and exposes its contents.

The Figure Palette enables you to perform the following tasks with these panels:

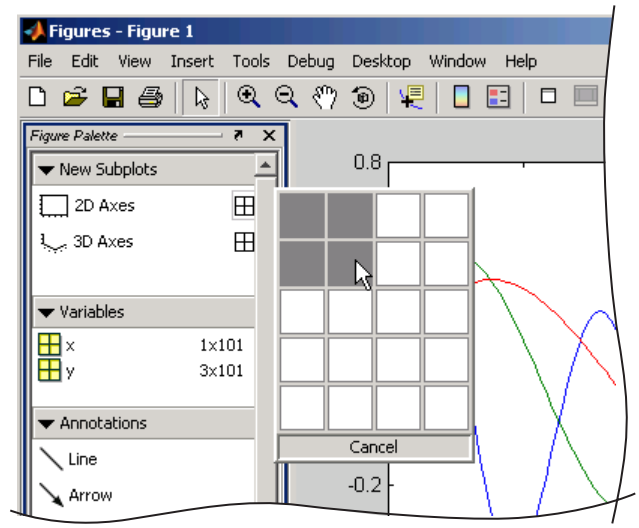
- **New Subplots** — Add 2-D or 3-D axes to the figure.

- **Variables** — Browse and plot workspace variables.
- **Annotations** — Add annotations to graphs.

Adding Subplot Axes

The **New Subplots** panel enables you to create a grid of either 2-D or 3-D axes. To display the selector, click the grid icon next to the axes type. MATLAB displays the selector grid.

As you move the cursor, squares darken to indicate the layout of axes that will be created if you release the mouse button. Click **Cancel** at the bottom of the grid to leave the figure unchanged.



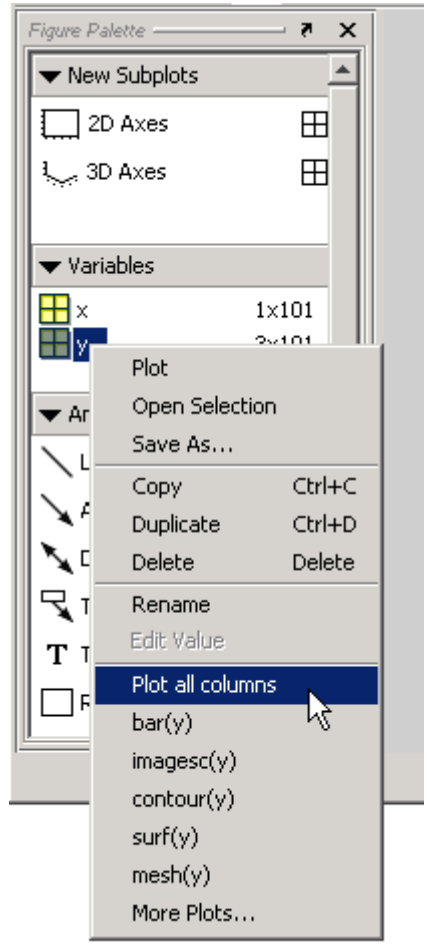
The picture above shows the **New Subplots** panel selected to display four equally sized axes in the figure. Existing axes resize as required to accommodate the new layout.

Plotting Workspace Variables

The **Variables** panel displays current workspace variables. Double-clicking a variable in this panel opens that variable in the Array Editor. If you select a

variable and right-click to display the context menu, you can select a graphics function to plot the variable.

For example, the following picture illustrates how to plot the columns of variable `Z`. This is equivalent to passing a matrix to the `plot` function.



The context menu contains a list of possible plot types based on the variable you select and also enables you to perform certain operation on the variable, such as opening it in the Array Editor, saving, copying, and so on.

Note that the context menu items may change when you select different variables because a particular variable might be incompatible some of the plot types.

Drag and Drop Plotting

You can also drag the variable directly into an axes, in which case MATLAB selects the first appropriate plot type for that variable. If there are multiple axes, you must first select the one you want to plot in and then drag the variable to that axes.

In the previous example, the variable `Z` would be plotted using the `plot` function if you were to drag it into an axes.

If the desired plotting function is not available from the context menu, you can select **More Plots** to display the Plot Catalog tool.

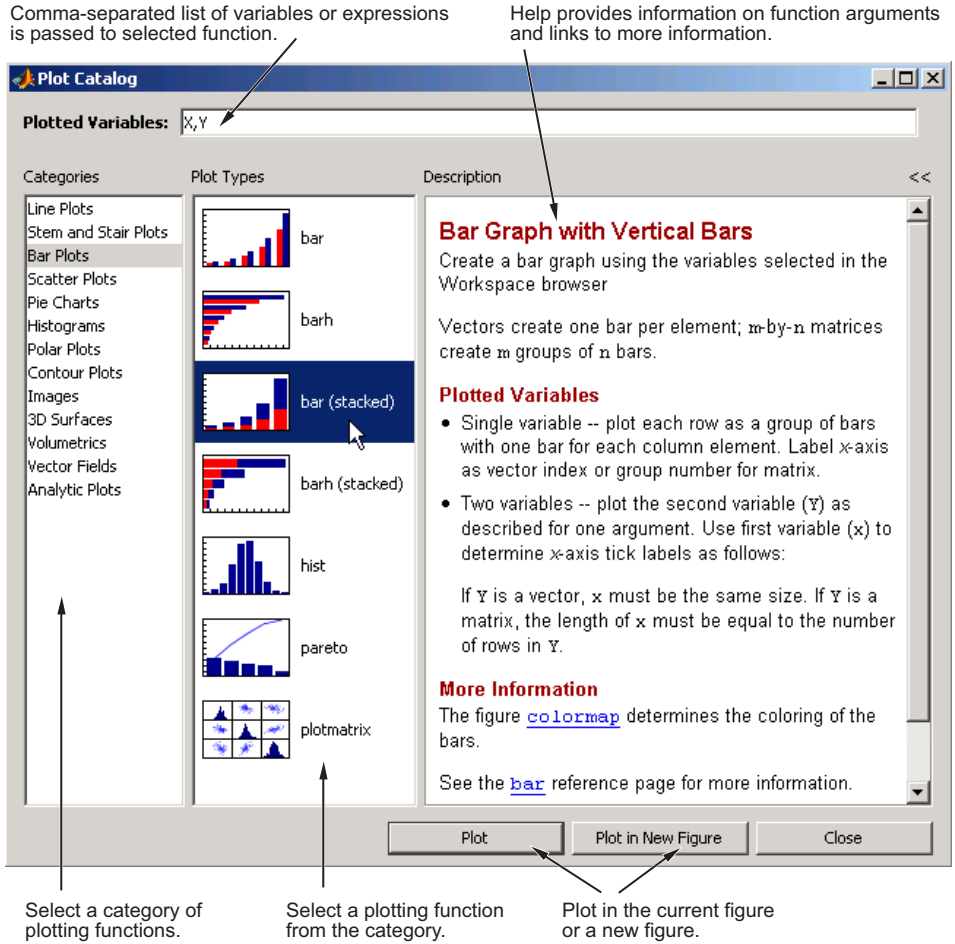
The Plot Catalog Tool

The Plot Catalog tool provides access to most of the MATLAB plotting functions. You can type any workspace variables or MATLAB expressions in the **Plotted Variables** field, which are then passed to the selected plotting function as arguments. Separate variables with a comma.

You can also enter function handles to pass to one of the “ez...” family of plotting functions from the Analytic Plots category.

Note The Plot Catalog does not prevent you from passing inappropriate or insufficient data to a plotting function; if the plot appears to be incorrect or if a plot fails to draw after you press **Plot** or **Plot in New Figure**, you should check the Command Window for warning and error messages.

The following picture shows the Plot Catalog tool and describes its components.



Adding Annotations to Graphs

The **Annotations** panel enables you to insert annotation objects into a plot. To add an object, first select the object you want to add, and then click and drag the mouse to position and size the object.

See “How to Annotate Graphs” on page 3-2 for more information about the various types of annotation objects.

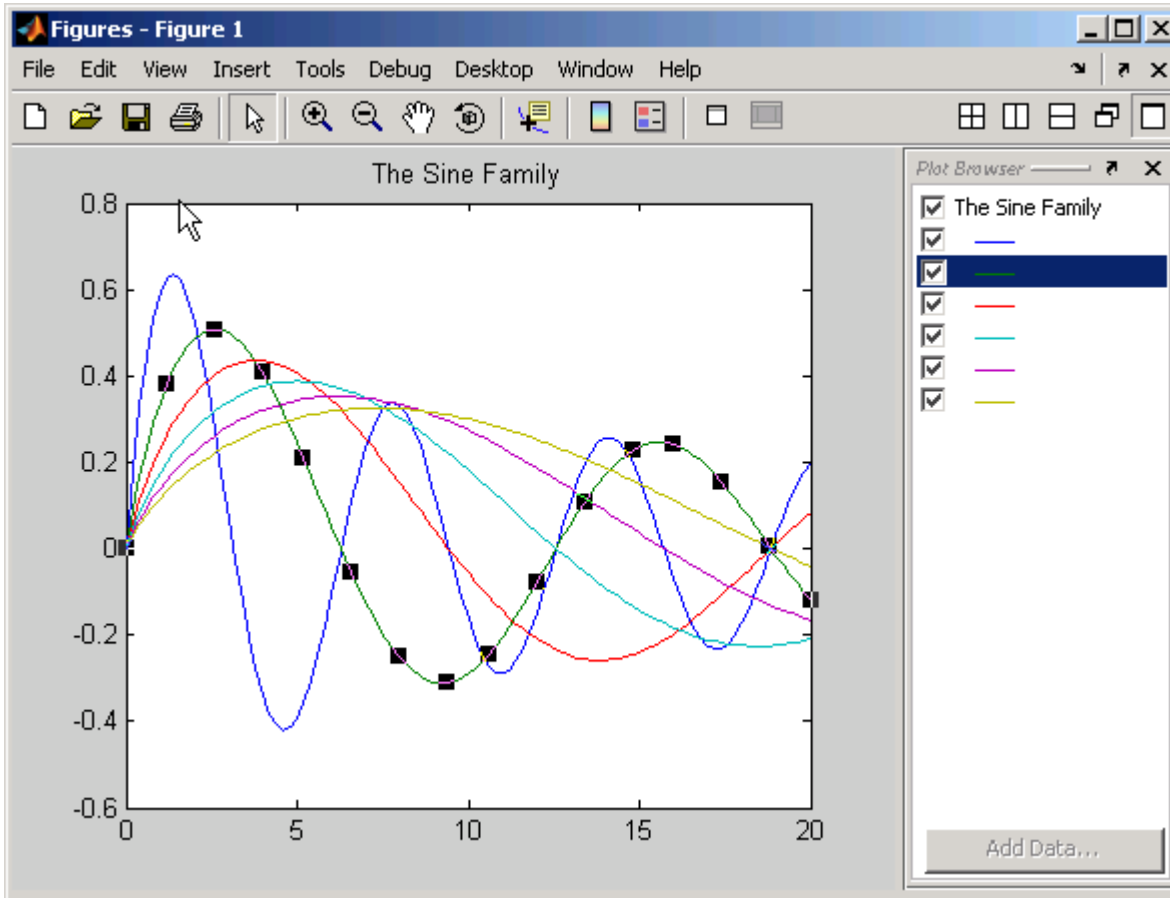
The Plot Browser

The Plot Browser provides a legend of all the graphs in the figure. It lists each axes and the objects (lines, surfaces, etc.) used to create the graph.

For example, suppose you plot an 11-by-11 matrix `z`. The `plot` function creates one line for each column in `z`.

```
plot(z,'DisplayName','z')
```

When you set the `DisplayName` property, the Plot Browser indicates which line corresponds to which column.

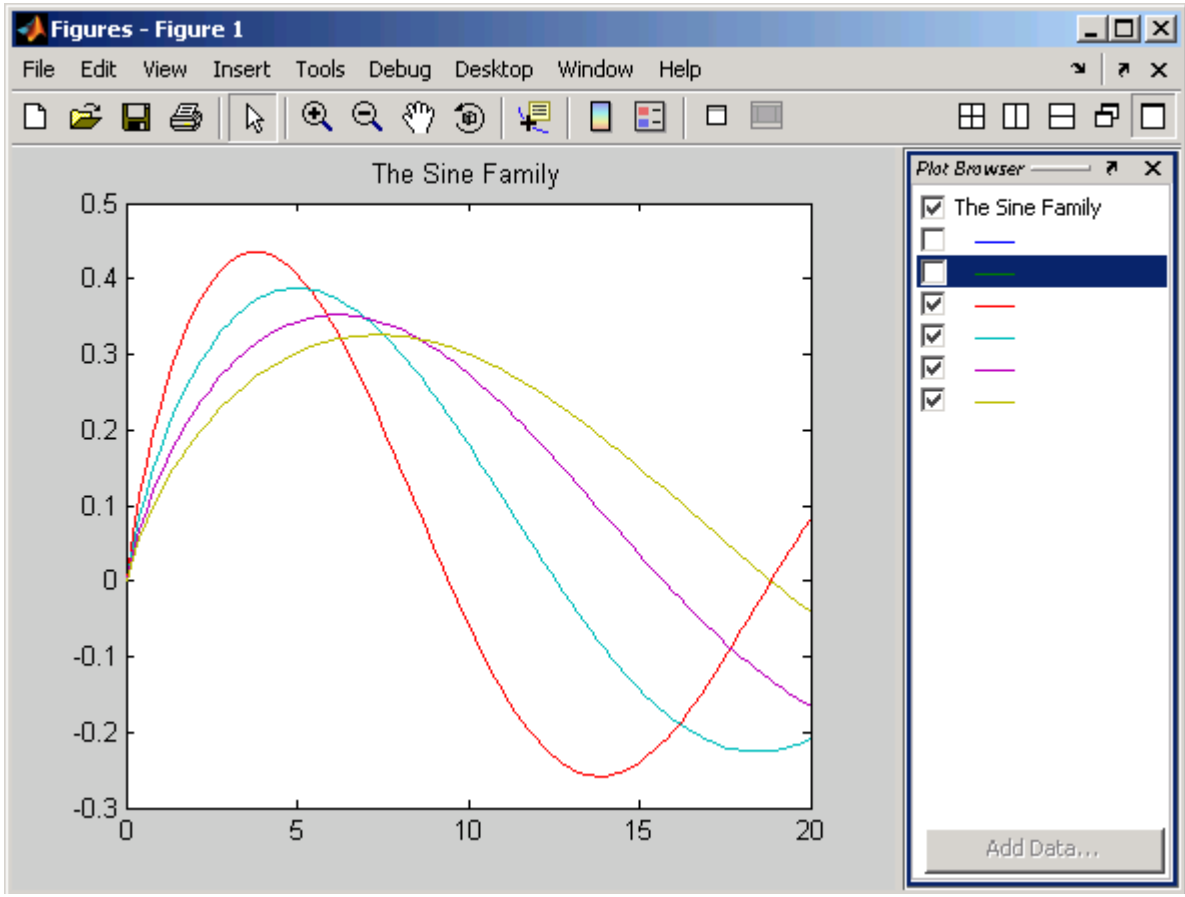


If you want to set the properties of an individual line, double-click on the line in the Plot Browser. Its properties are displayed in the Property Editor, which opens on the bottom of the figure.

You can select a line in the graph, and the corresponding entry in the Plot Browser is highlighted, enabling you to see which column in the variable produced the line.

Controlling Object Visibility

The check box next to each item in the Plot Browser controls the object's visibility. For example, suppose you want to plot only certain columns of data in z, perhaps the positive values. You can deselect the columns you do not want to display. The graph updates as you deselect each box and rescales the axes as required.



Deleting Objects

You can delete any selected item in the Plot Browser by selecting **Delete** from the right-click context menu.

Adding Data to Axes

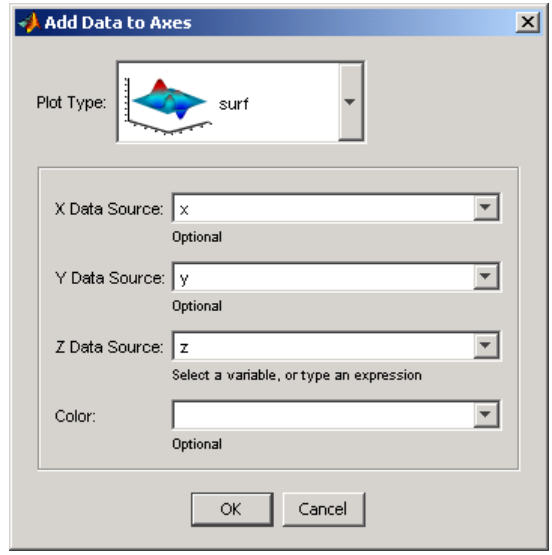
The Plot Browser provides the mechanism by which you add data to axes. The procedure is as follows:

1. Select a 2-D or 3-D axes from the **New Subplots** subpanel.
2. After creating the axes, select it in the Plot Browser panel to enable the **Add Data** button at the bottom of the panel.
3. Click the **Add Data** button to display the Add Data to Axes dialog.

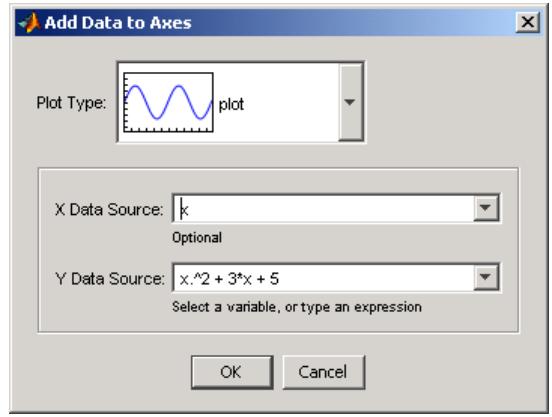
The Add Data to Axes dialog enables you to select a plot type and specify the workspace variables to pass to the plotting function. You can also specify a MATLAB expression, which is evaluated to produce the data to plot.

Selecting Workspace Variables to Create a Graph. Suppose you want to create a surface graph from three workspace variables defining the `XData`, `YData`, and `ZData` (see the `surf` function for more information on this type of graph).

In the workspace you have defined three variables, `x`, `y`, and `z`. To create the graph, configure the Add Data to Axes dialog as shown in the following picture.



Using a MATLAB Expression to Create a Graph. The following picture shows the Add Data to Axes dialog specifying a workspace variable `x` for the plot's x data and a MATLAB expression (`x.^2 + 3*x + 5`) for the y data.



You can use the default **X Data** value of `index` if you do not want to specify x data. In this case, MATLAB plots the y data versus the index of the y data value, which is equivalent to calling the `plot` command with only one argument.

The Property Editor

The Property Editor enables you to access a subset of the selected object's properties. When no object is selected, the Property Editor displays the figure's properties.

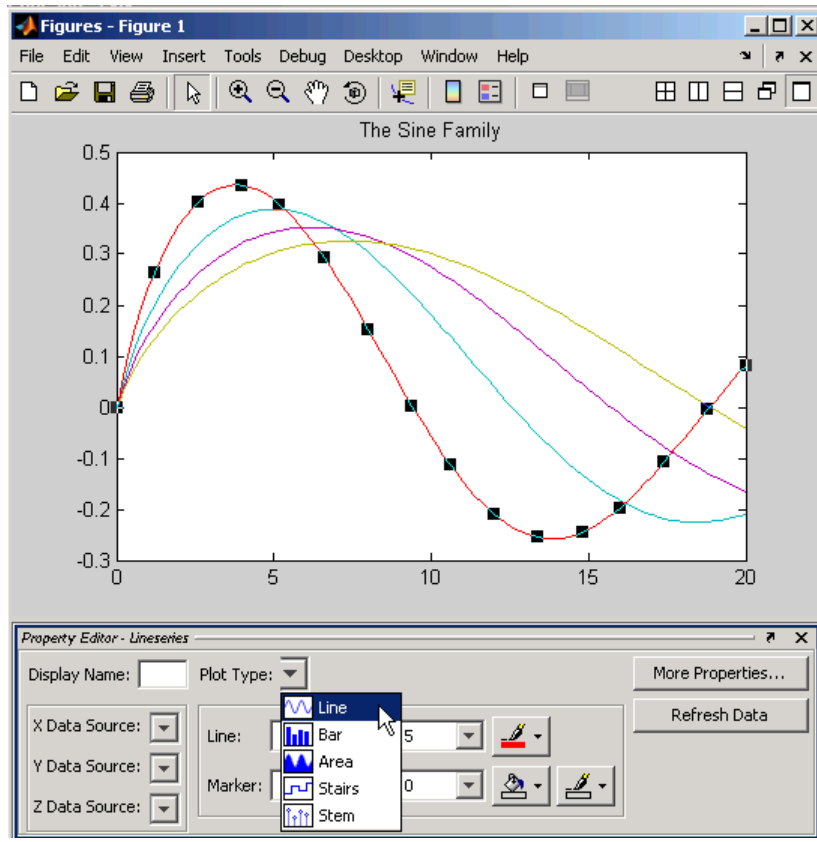
Ways to Display the Property Editor

There are a variety of ways to display the Property Editor:

- Double-click an object when plot edit mode is enabled.
- Select an object and right-click to display its context menu, then select **Properties.**
- Select **Property Editor** from the **View** menu.
- Use the `propertyeditor` command.

Changing Plot Types

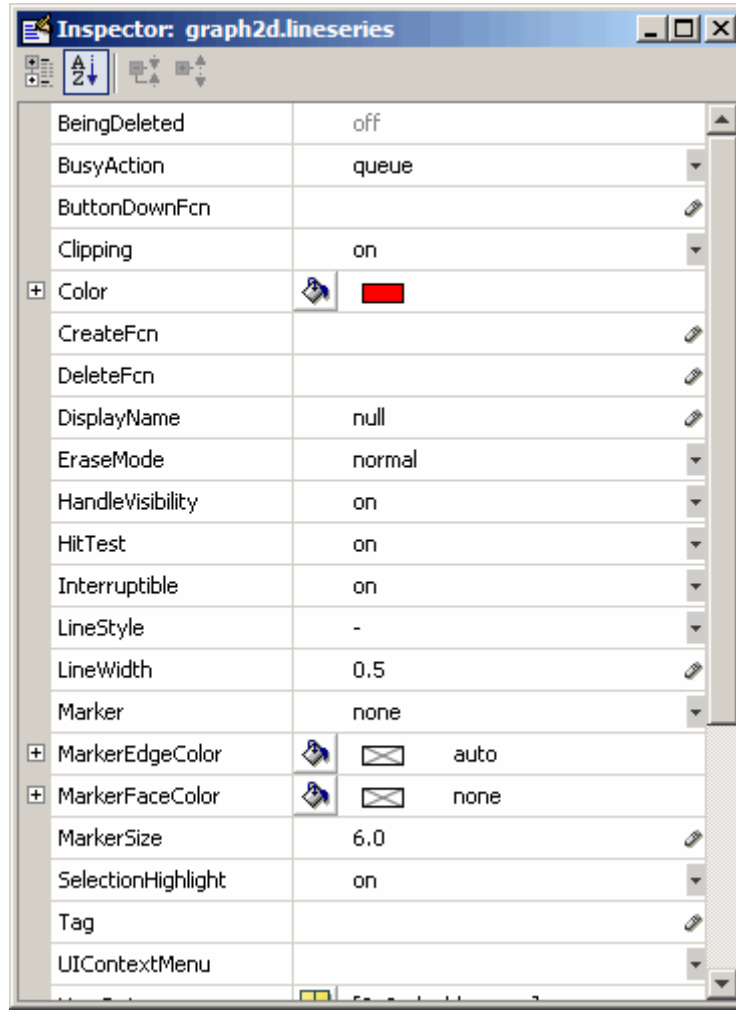
You can use the property editor to change the type of plot used to display data. For example, you can change the following line graph to a stem, stairs, area, or bar graph by changing the **Plot Type** field.



Accessing Object Properties with the Property Inspector

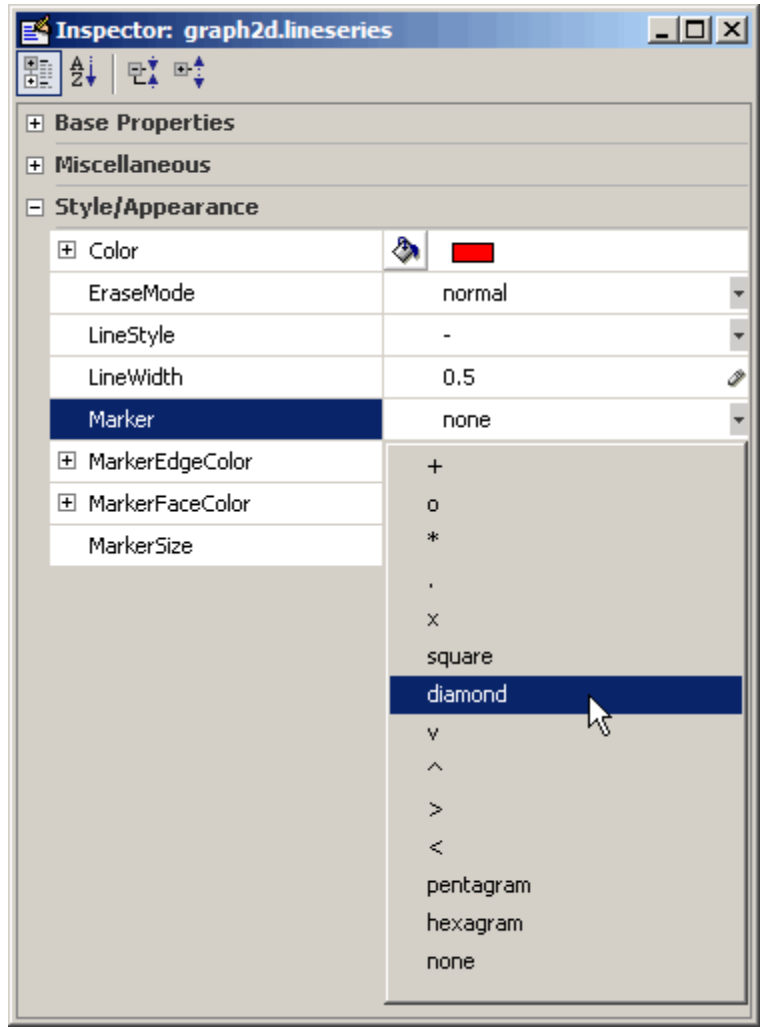
The Property Editor enables you to change the most commonly used object properties. To access more object properties, use the Property Inspector. Open the Property Inspector by clicking the **More Properties** button on the Property Editor or by typing `inspect` in the Command Window. The following

picture shows the Property Inspector displaying the properties of the same lineseries object as that in the previous picture.



The default view of properties is an alphabetic list; you can change to a tree view by clicking the icon in the upper left corner containing plus marks. Click the “AZ” icon to its right to return to an alphabetized list view. Properties that contain fields, such as RGB color components, have a plus mark to

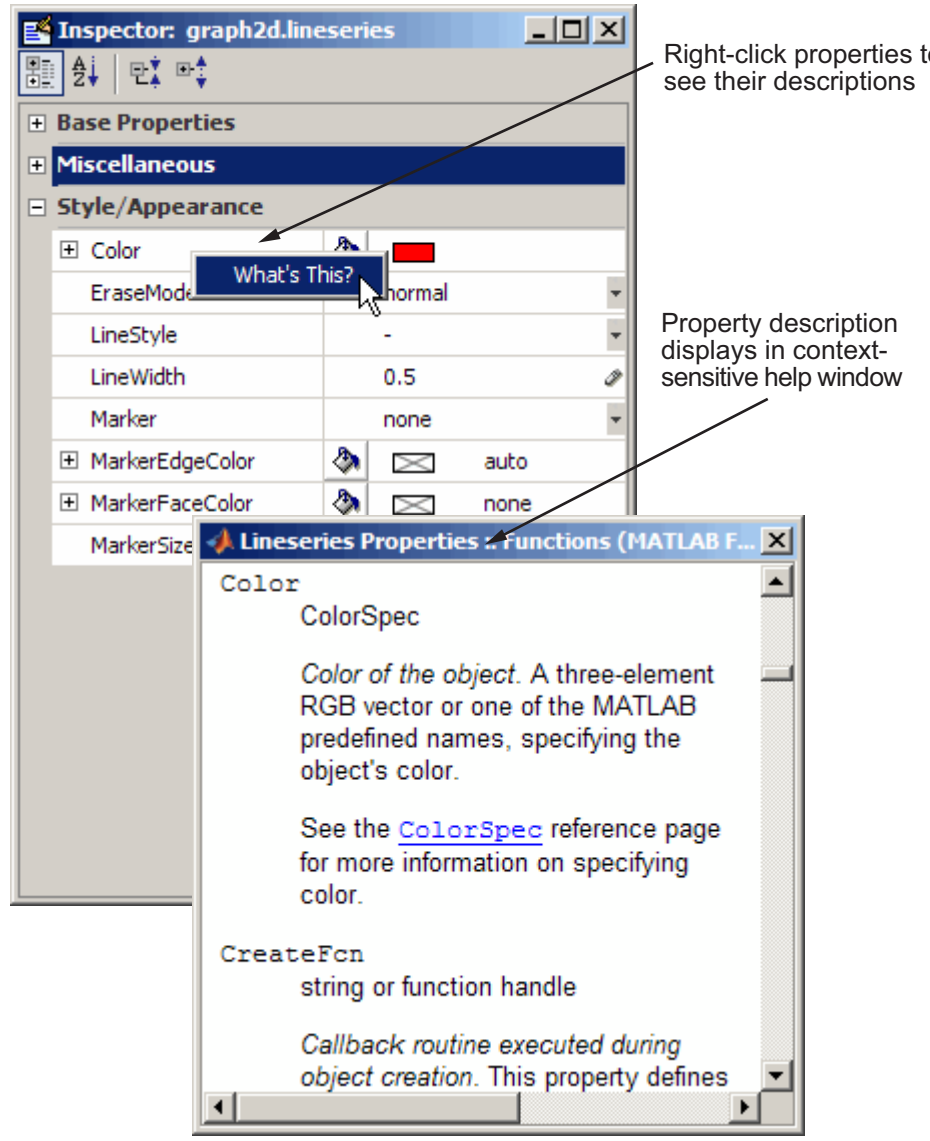
their left you can click to expose the individual values. You can change properties that have enumerated values, such as `Marker`, via drop-down lists (downward-pointing triangles on the right edge of the inspector window). The following figure shows the `Marker` property being set to `diamond` using the Property Inspector (note the tree view in which other groups of properties have been collapsed):



There are a few properties of objects that the Property Inspector does not show, for example `Parent` and `Children`. For complete descriptions of the properties of graphics objects, use the Handle Graphics Property Browser.

Getting Help for Object Properties

If you are not sure what a property does or what values it can take on, you can get a description of it from the Property Inspector. To do so, right-click on the name or values of a property and select **What's This** from the popup menu; a Help window opens displaying the property reference page for the current object, scrolled to the property you clicked. The following picture shows how this works:


Inspector: graph2d.lineseries
Base Properties
Miscellaneous
Style/Appearance
Color
What's This?
EraseMode
normal
LineStyle
-
LineWidth
0.5
Marker
none
MarkerEdgeColor
auto
MarkerFaceColor
none
MarkerSize
Lineseries Properties : Functions (MATLAB F...
Color
ColorSpec
Color of the object. A three-element RGB vector or one of the MATLAB predefined names, specifying the object's color.
See the ColorSpec reference page for more information on specifying color.
CreateFcn
string or function handle
Callback routine executed during object creation. This property defines
Right-click properties to see their descriptions
Property description displays in context-sensitive help window

Accessing Objects You Cannot Click

If you want to access the properties of light or uicontextmenus objects, you need to get the handle using MATLAB commands, because you cannot click on these objects.

For example, to get the handles of all light objects in the current axes, use `findobj`.

```
h = findobj(gca,'Type','light');
```

Then use the `inspect` command to display the Property Inspector.

```
inspect(h)    % Inspect all light objects
inspect(h(1)) % Inspect the first light object in list
```

Example — Working with Plotting Tools

In this section...

Identifying Workspace Data to Plot

This example illustrates how to use the plotting tools to graph a workspace variable versus an expression typed into the **Add Data to Axes** dialog.

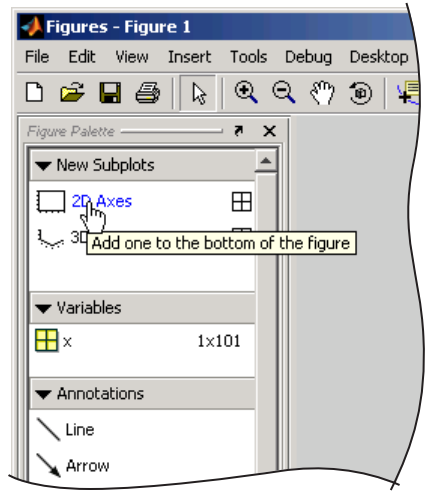
Create a variable in the workspace,

```
x = -2*pi:pi/25:2*pi;
```

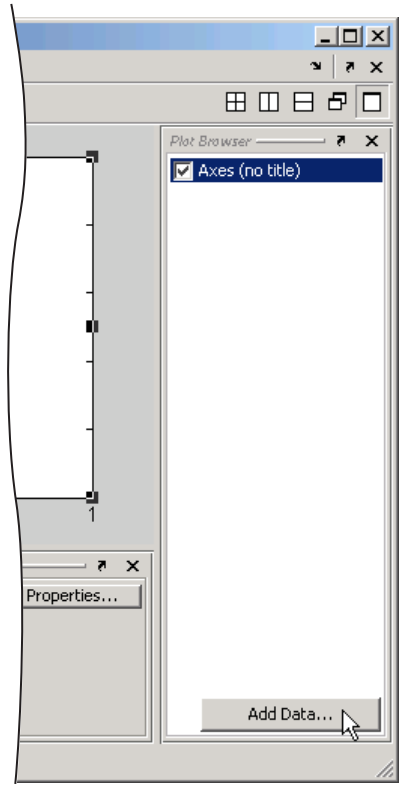
Use the `plottools` command to create a figure group with the plotting tools attached.

```
plottools
```

Click **2D Axes** in the **New Subplot** panel of the Figure Palette.

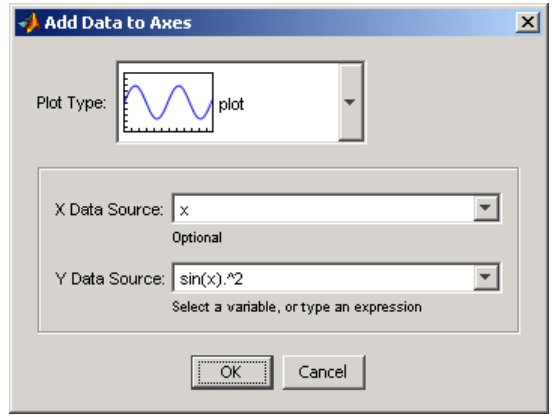


Once the axes appears, the **Add Data** button on the Plot Browser is activated.



Click this button to display the Add Data to Axes dialog. When the Add Data to Axes dialog is displayed, enter the following values:

- Select plot as the **Plot Type**.
- Set **X Data Source** to `x`.
- Set **Y Data Source** to `sin(x).^2`.
- Click **OK** to plot this data.

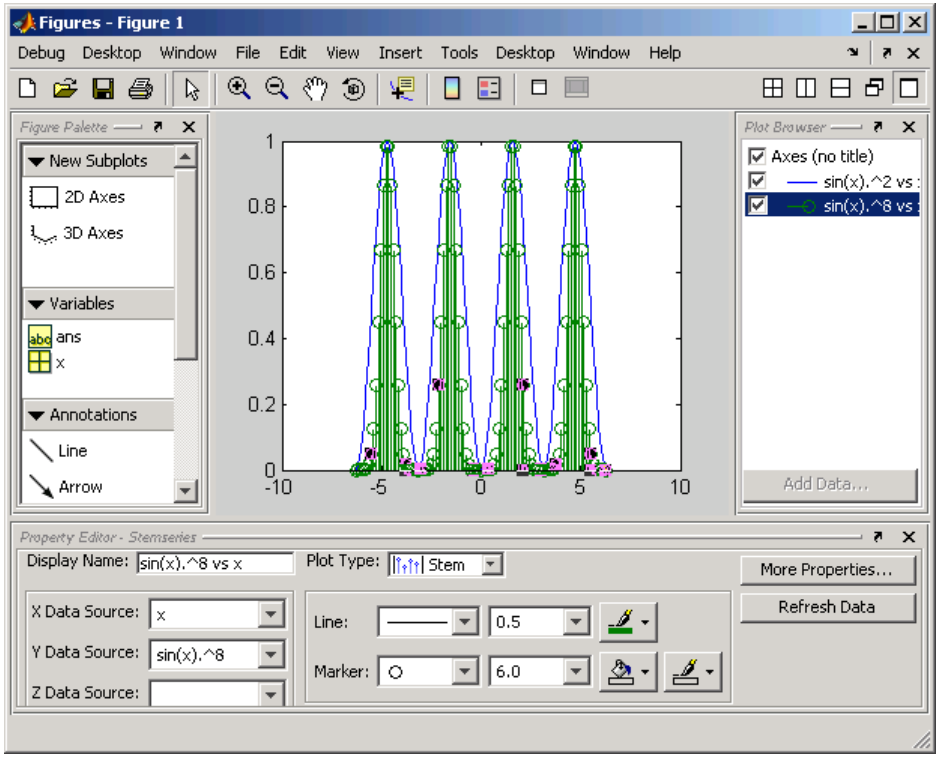


MATLAB draws a plot of `sin(x).^2` versus `x`.

Now add another plot to the same axes. Click **Add Data** again and specify the data to plot:

- Set **X Data Source** to `x`.
- Set **Y Data Source** set to `sin(x).^8`.
- Click **OK** to plot this data.

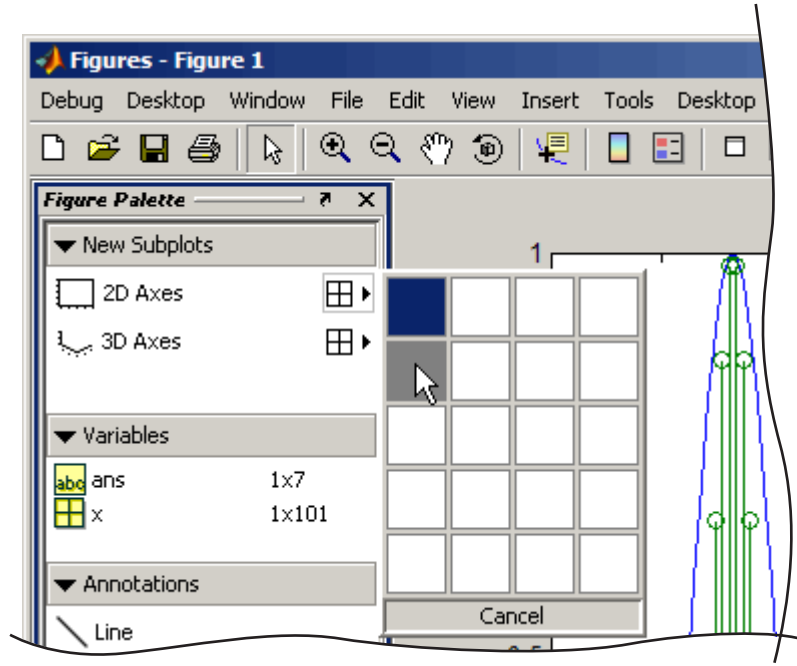
Select the last plot (the green line) and set the `Plot Type` in the Property Editor to `Stem`. The plot should now look like the following picture.



Adding a Subplot

Add a second axes below the current axes using the **New Subplots** panel. Click the right-facing arrowhead next to **2D Axes** and move the mouse to darken two squares, one on top of the other.

This creates a subplot axes below the existing axes. MATLAB resizes the existing axes so both fit in the figure.



Once MATLAB inserts the new axes, select its entry in the Plot Browser and then click **Add Data**.

When the Add Data to Axes dialog is displayed, enter the following values:

- Set **X Data Source** to `x`.
- Set **Y Data Source** to `sin(x).^3`.
- Click **OK** to plot this data.

Now add another plot overlaid on the first by clicking **Add Data** again and specify the data to plot:

- Set **X Data Source** to `x`.

- Set **Y Data Source** to `sin(x).^9`.
- Click **OK** to plot this data.

Select the plot labeled `sin(x).^9` under the second axes in the Plot Browser. Set the `Plot Type` in the Property Editor to `Area`.

Setting Axis Limits

Adjust the *x*-axis in both axes using the Property Editor.

- Select the first axes in the Plot Browser.
- Change **X Limits** to -7 and 7.

Repeat these steps for the second axes.

Adding Titles and Labels

Select the first axes in the Plot Browser and set the following properties in the Property Editor:

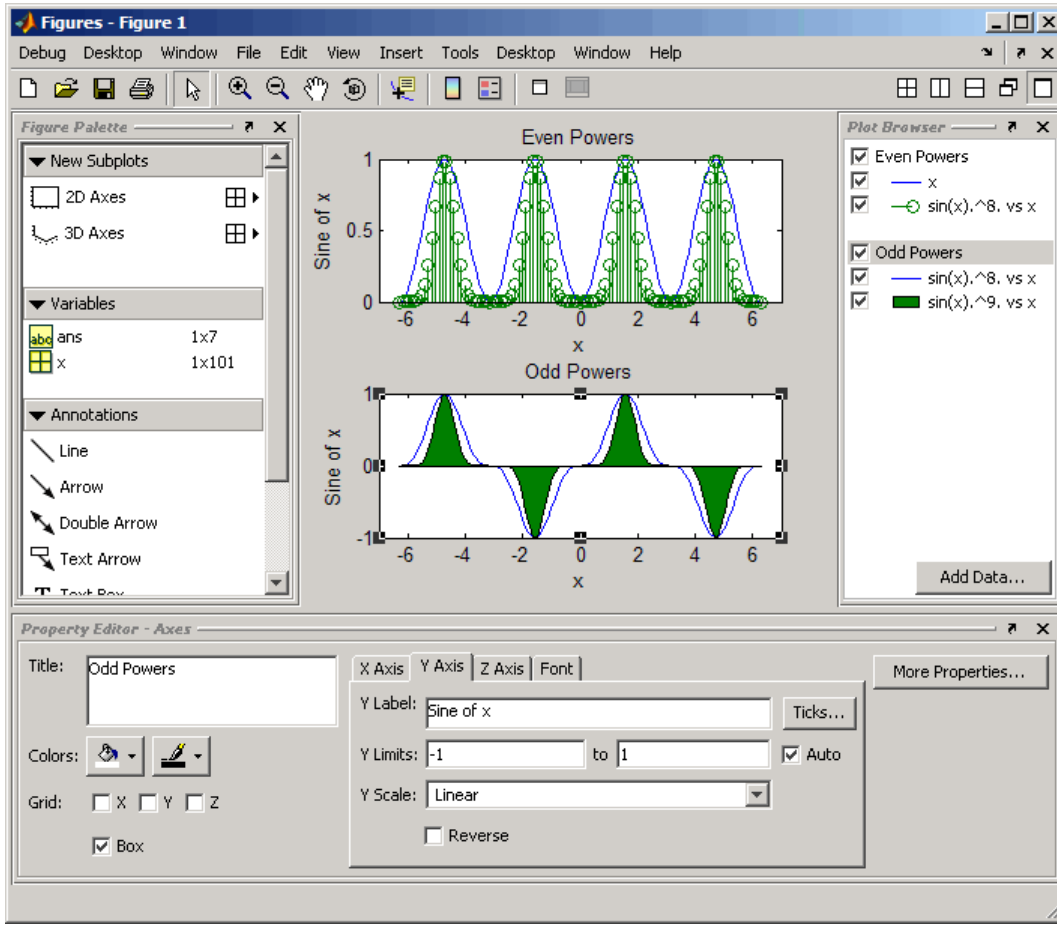
- Set **Title** to `Even Powers`.
- Set **X Label** to `X`.
- Click the **Y Axis** tab and set **Y Label** to `Sine of X`.

Select the second axes in the Plot Browser and set the following properties in the properties panel:

- Set **Title** to `Odd Powers`.
- Set **Axis label** to `Sine of X`.
- On the **Y Axis** tab, set **Axis label** to `Sine of X`.

Note that the Plot Browser reflects the new axes names.

The following picture shows the result of these steps.

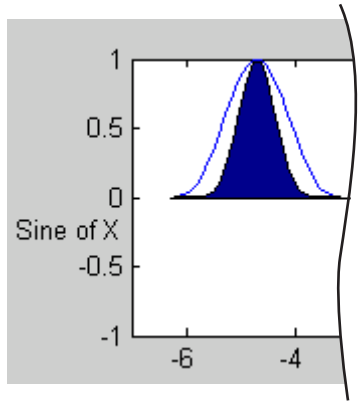


Select the text of the *y*-axis label on the first axes (now labeled `Even Powers` in the Plot Browser) and click the **More Properties** button on the Property Editor. Set the `Rotation` property to `0` and reposition the text by hand.

To make more space for the *y*-axis label, which is now in a horizontal position, select the axes and move it to the right with the mouse.

Repeat this process for the second axes (labeled `Odd Powers` in the Plot Browser).

The repositioned text label now looks like the following picture.



Note You can always undo your last change to the graph by selecting **Undo** from the **Edit** menu.

Example — Plotting from the Figure Palette

In this section...

Using the Plot Catalog

This example shows how to use the Figure Palette to select variables to plot. Suppose you have three variables in your workspace (`x`, `y`, `z`) created by the following statements:

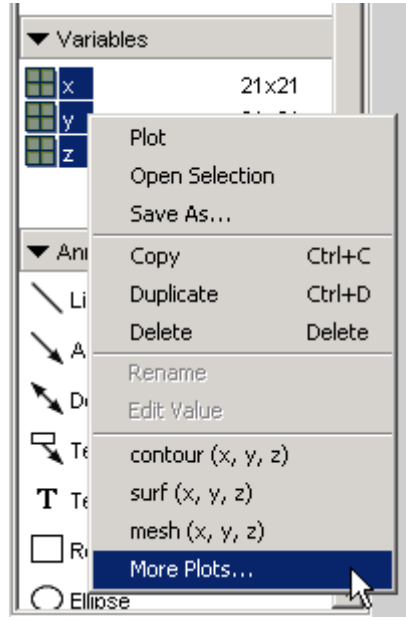
```
[x,y] = meshgrid([-2:.2:2]);
z = x.*exp(-x.^2-y.^2);
```

You decide to visualize this data as a surface/contour plot (as produced by the `surfc` function).

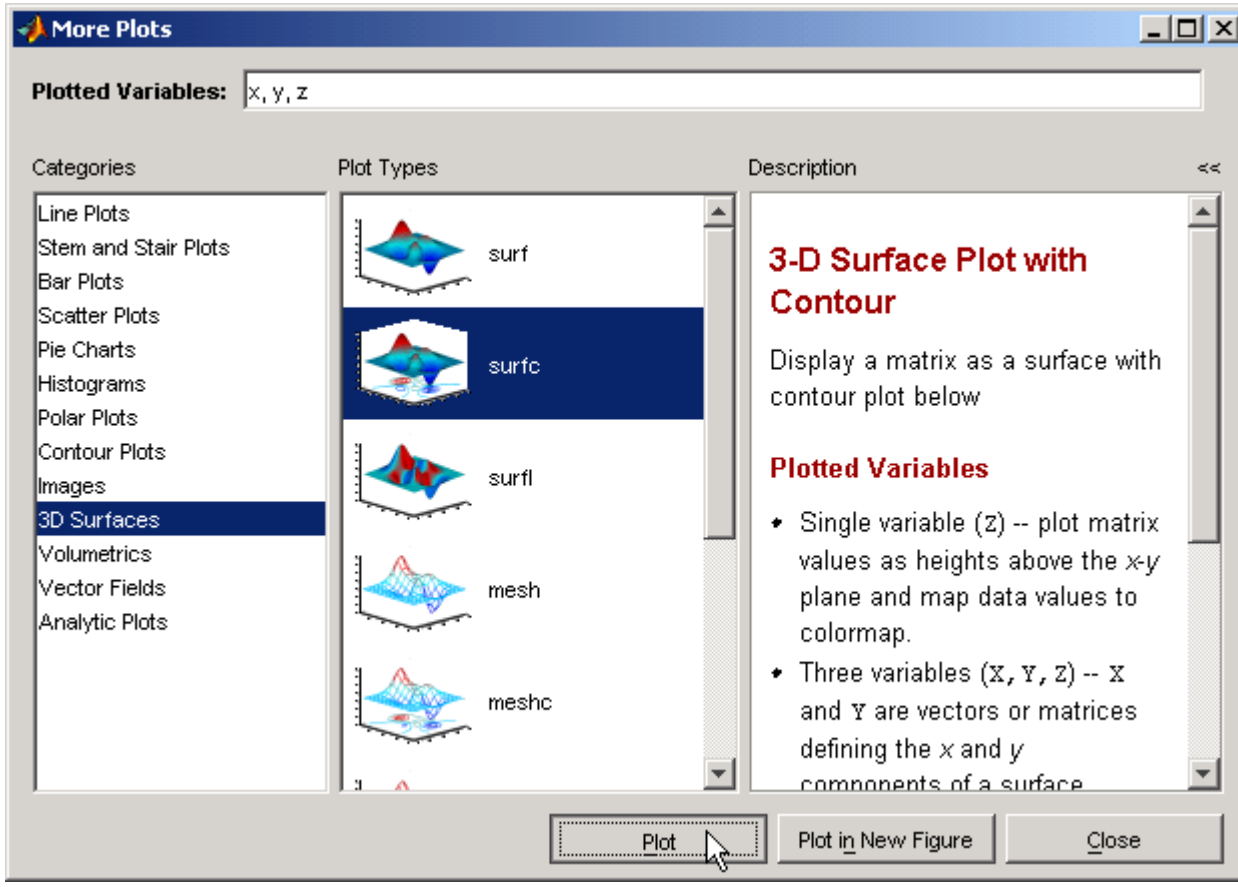
The first step is to display a figure with the Figure Palette tool attached. You can do this with the `figurepalette` command.

```
figurepalette
```

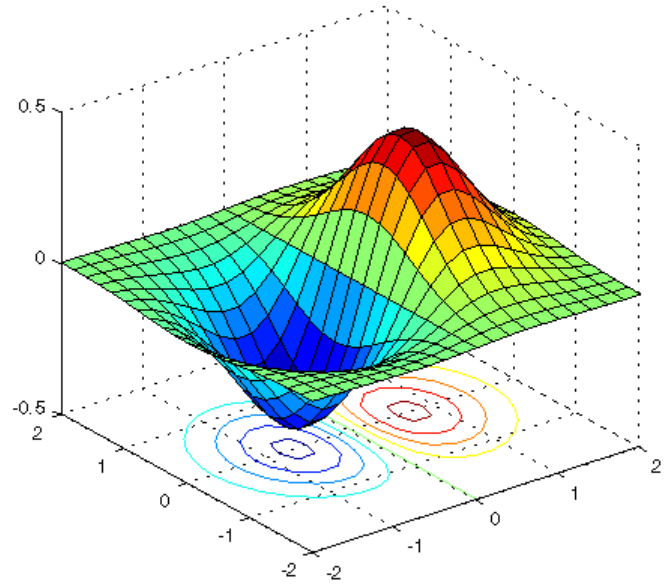
Expand the **Variables** panel and **shift-click** (for multiple selection) on the three variables you want to pass the plotting function. Since `surfc` is not in the list, select **More Plots**.


Variables
x
21x21
y
z
Plot
Open Selection
Save As...
Copy Ctrl+C
Duplicate Ctrl+D
Delete Delete
Rename
Edit Value
contour (x, y, z)
surf (x, y, z)
mesh (x, y, z)
More Plots...
Ellipse

From the Plot Catalog tool, select the **3D Surfaces** in the **Categories** column and **surfc** as the **Plot Type**, as shown in the following picture. To create the plot, click the **Plot** button.



MATLAB creates the following graph.



Plotting Expressions

You can enter MATLAB expressions in the Plot Catalog tool, as well as variables. For example, suppose you have created the following variables in the workspace.

```
t = 0:.01:20;
alpha =.055;
```

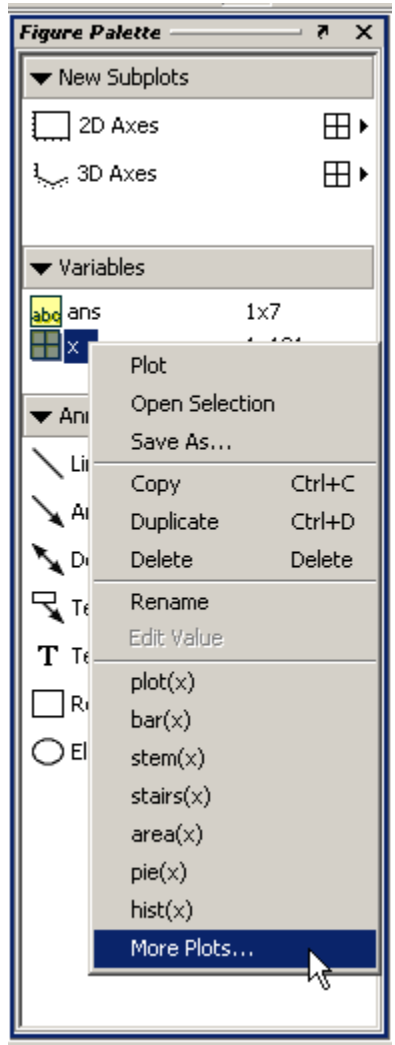
and you want to plot `t` versus this expression:

```
exp(-alpha*t).*sin(.5*t)
```

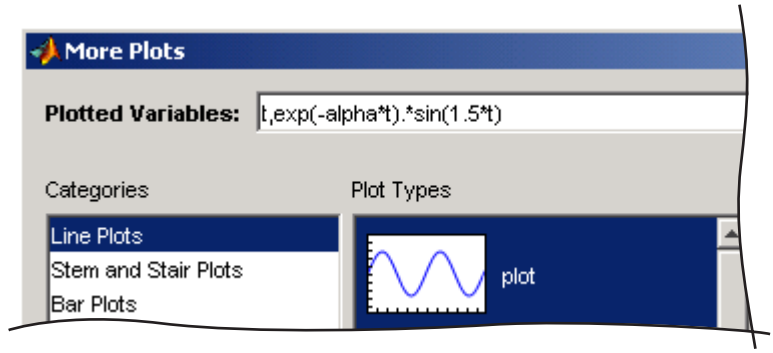
The first step is to display a figure with the Figure Palette tool attached. You can do this with the `figurepalette` command.

```
figurepalette
```

First select the variable `t` and right-click to display the context menu. Select **More Plots**.



When the Plot Catalog tool is displayed, add the expression to the **Plotted Variables** text field. Note that you can reference the variable `alpha` because you created it in the base workspace. See MATLAB Workspace for information about variables in the MATLAB workspace.



Click the **Plot** button to create the graph. Note that the previous step is the equivalent of issuing the following command:

```
plot(t,exp(-alpha*t).*sin(.5*t))
```

Example — Specifying a Data Source

In this section...

Creating the Graph

First define two variables by issuing these statements in the command window.

```
t = 0:.01:20;
alpha =.055;
```

Next plot `t` versus the expression `exp(-alpha*t).*sin(.5*t)` using the `plot` function or the plot tools.

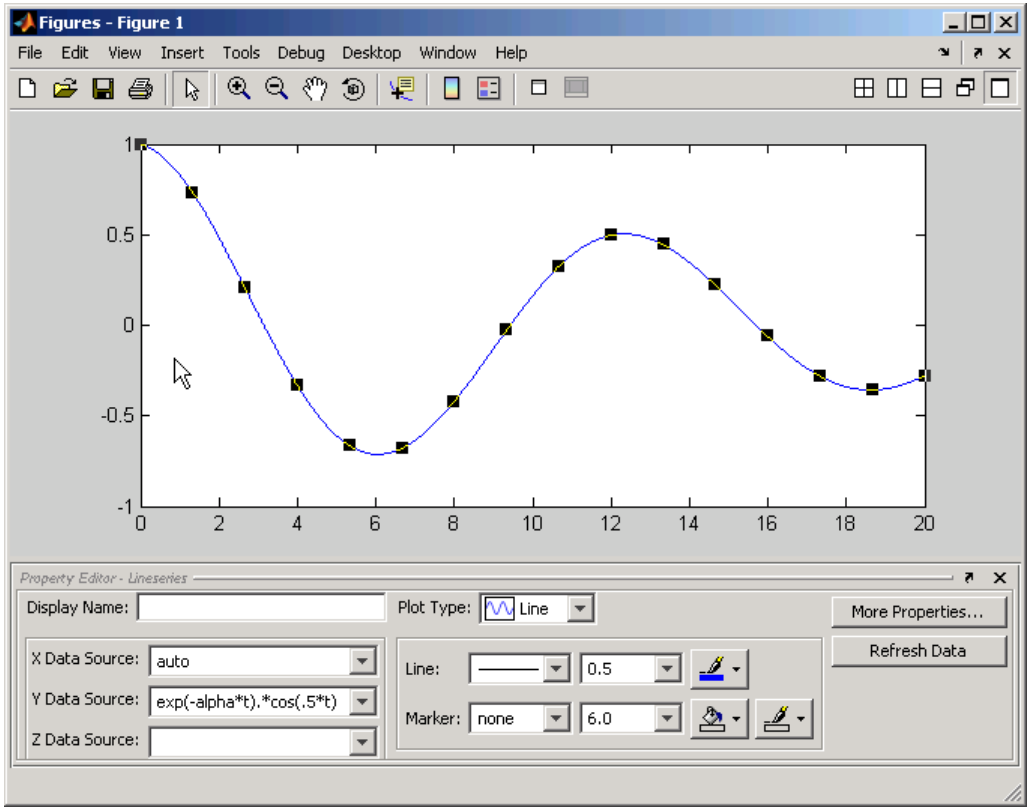
```
plot(t,exp(-alpha*t).*sin(.5*t))
```

Varying the Data Source

Plot objects have properties that enable you to specify the source of the data that defines the object. For example, you can specify a workspace variable name or a MATLAB expression as the value of the `XDataSource`, `YDataSource`, or `ZDataSource` property for a line in a plot (i.e., a lineseries object). You can then use the Property Editor to change the variable name or alter the expression, and the plot is updated to reflect the change.

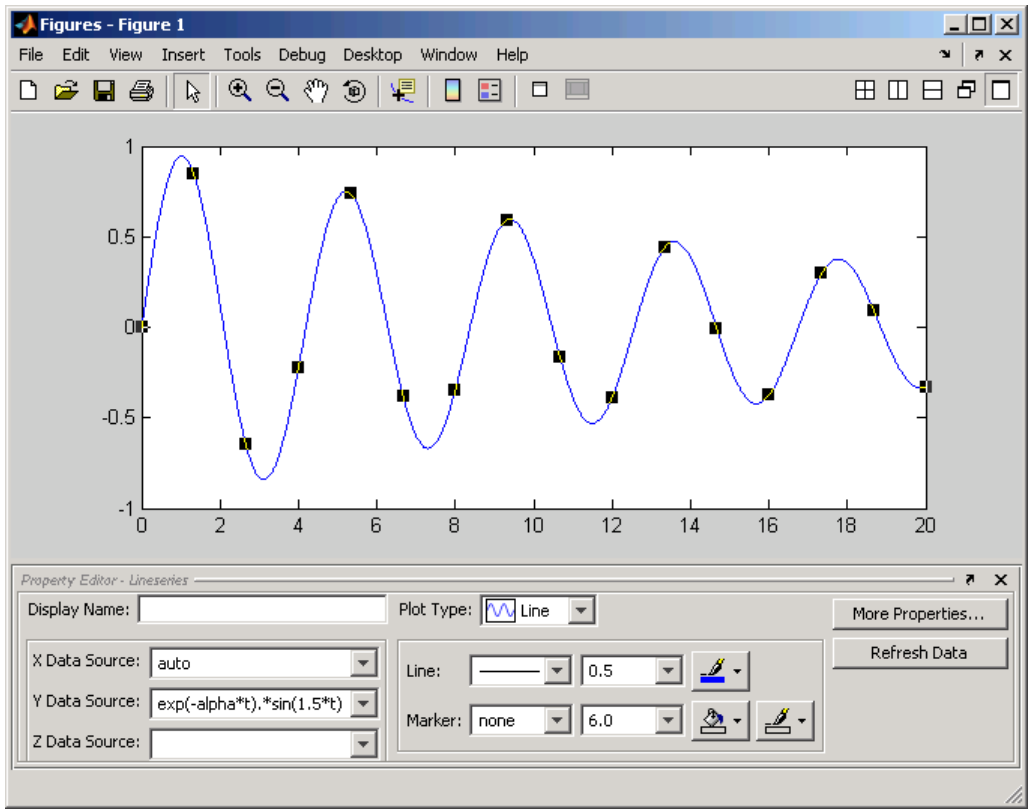
After creating the graph, you can use the Property Editor to couple the plotted line to the MATLAB expression.

1. Double-click on the plotted line to display its property panel.
2. Enter the MATLAB expression `exp(-alpha*t).*cos(.5*t)` in the **Y Data Source** text field.



You can now modify the expression in the **Y Data Source** text field and observe how the graph changes. After changing the text, click the **Refresh Data** button to update the data.

In the following picture, `alpha` is no longer negated, so the function grows instead of decays. Also the period has been shortened by changing `sin(.5*t)` to `sin(1.5*t)`.



Data Sources for Multiobject Graphs

Suppose you create a line graph from matrix data. For example,

```
z = peaks;
h = plot(z,'YDataSource','z');
```

Because MATLAB creates one lineseries object for each column of `z`, the following is true.

The data source for `h(1)` is `z(:,1)`.

The data source for `h(2)` is `z(:,2)`.

...

The data source for `h(n)` is `z(:,n)`.

Example — Generating M-Code to Reproduce a Graph

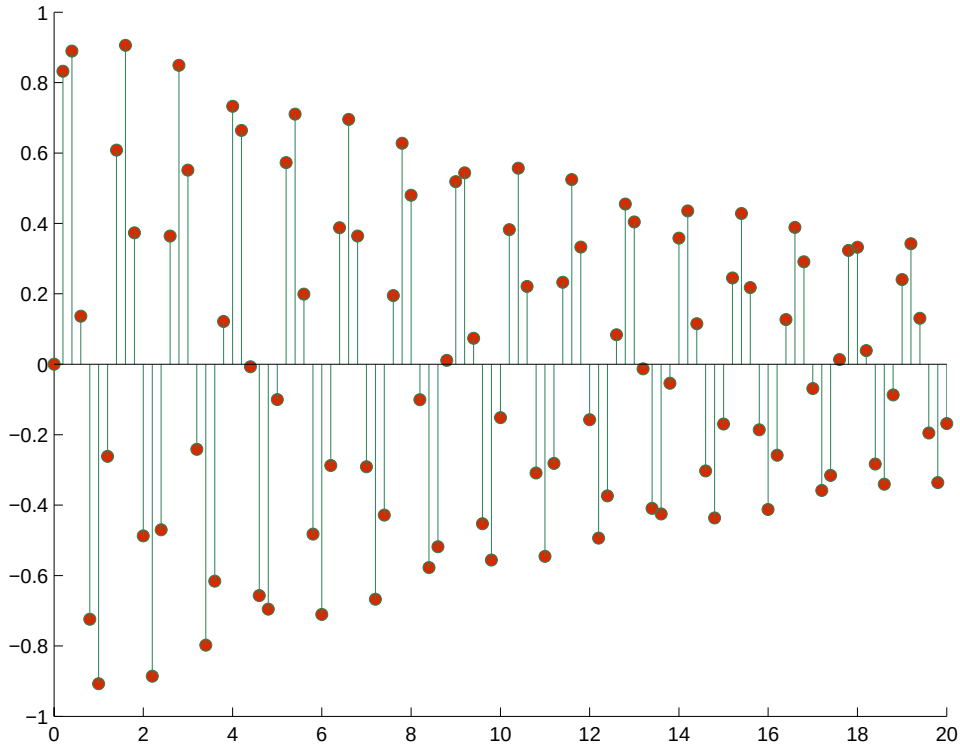
In this section...

Create a Stem Plot and Generate Code for It

Suppose you have created the following graph.

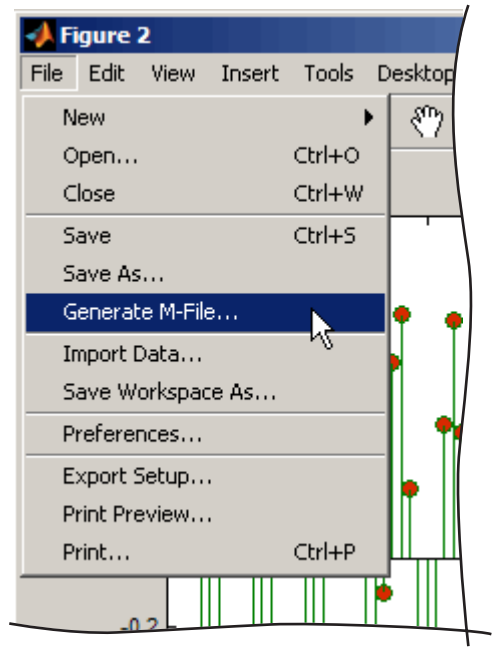
```
t = 0:.2:20;
alpha =.055;
stem(t,exp(-alpha*t).*sin(5*t))
```

Use the Property Editor to modify the graph. Select the stemseries and change the marker fill color to dark red, and marker edge color and line color to dark green. Remove the axes box, and change the font size for the axes labels to 8 to look like the following picture:



You can generate code to reproduce this graph by selecting **Generate M-File** from the **Figure** menu. MATLAB creates a function that recreates the graph and opens the generated M-File in the editor.

This feature is particularly useful for capturing property settings and other modifications you make using the plot tools GUI.



The M-file appears in an editor window and consists of the following code:

```
function createfigure(X1, Y1)
%CREATEFIGURE(X1,Y1)
%  X1:  stem x
%  Y1:  stem y

%  Auto-generated by MATLAB on 24-May-2006 14:23:45

% Create figure
figure1 = figure('Color',[1 1 1]);

% Create axes
axes('Parent',figure1,'FontSize',8);
hold('all');

% Create stem
stem(X1,Y1,'MarkerFaceColor',[0.8471 0.1608 0],...
    'MarkerEdgeColor',[0.1686 0.5059 0.3373],...
```

```
    'Color',[0 0.498 0]);
```

You must save the M-file before exiting MATLAB if you want to use it in future sessions.

Data Arguments

Generated functions do not store the data necessary to recreate the graph. You must supply the data arguments `t` as `X1` and `exp(-alpha*t).*sin(5*t)` as `Y1` to the function to recreate your graph. Of course, you can call the generated function with other argument pairs too.

Limitations

Attempting to generate code for graphics objects containing a large number of graphics objects (e.g., greater than 20 plotted lines) might be impractical.

Editing Plots

In this section...

Why Edit Plots?

MATLAB formats a graph to provide readability, setting the scale of axes, including tick marks on the axes, and using color and line style to distinguish the plots in the graph. However, if you are creating presentation graphics, you might want to change this default formatting or add descriptive labels, titles, legends, and other annotations to help explain your data.

MATLAB supports two ways to edit the plots you create:

- Using the mouse to select and edit objects interactively
- Using MATLAB functions at the command line or in an M-file

Interactive Plot Editing

If you enable plot editing mode in the MATLAB figure window, you can perform point-and-click editing of your graph. In this mode, you can modify the appearance of a graphics object by double-clicking on the object and changing the values of its properties. You access the properties through a graphical user interface called the Property Editor.

For more information about interactive editing, see "Working in Plot Edit Mode" on page 1-59.

For information about editing object properties in plot editing mode, see "The Property Editor" on page 1-28.

Using Functions to Edit Graphs

If you prefer to work from the MATLAB command line or if you are creating an M-file, you can use MATLAB commands to edit the graphs you create.

Taking advantage of the MATLAB Handle Graphics® system, you can use the `set` and `get` commands to change the properties of the objects in a graph.

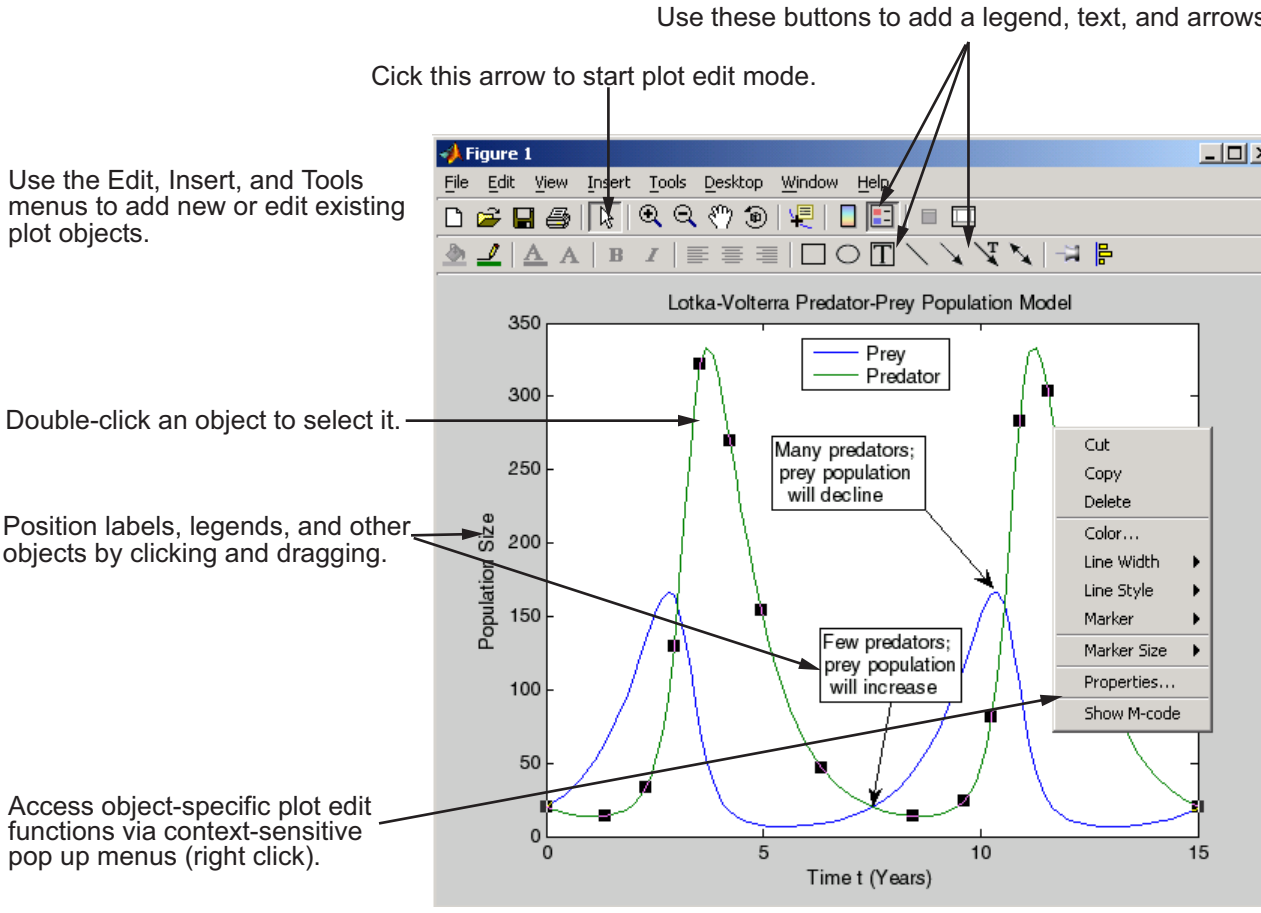
Note Plot editing mode provides an alternative way to access the properties of MATLAB graphic objects. However, you can only access a subset of object properties through this mechanism. You might need to use a combination of interactive editing and command-line editing to achieve the effect you desire.

Working in Plot Edit Mode

In this section...

Figure Windows in Plot Edit Mode

The MATLAB figure window supports a point-and-click editing mode that you can use to customize the appearance of your graph. This section describes how to enter plot edit mode and perform basic editing tasks, including selecting, cutting, copying, pasting, moving, and resizing objects and modifying other plot properties. The following figure illustrates some capabilities of plot edit mode.



Starting Plot Edit Mode

Before you can select objects in a figure by clicking on them, you must activate plot editing mode. There are several ways to activate plot edit mode:

- Choose the **Edit Plot** option on the figure window **Tools** menu.
- Click the selection button in the figure window toolbar.

- Choose an option from the **Edit** or **Insert** menu. For example, if you choose the **Axes Properties** option on the **Edit** menu, MATLAB activates plot edit mode and the axes appear selected.
- Run the `plotedit` command in the MATLAB Command Window.
- Start the plotting tools with the `plottools` command.

When a figure window is in plot edit mode, the **Edit Plot** option on the **Tools** menu is checked and the selection button in the toolbar is depressed.

Exiting Plot Edit Mode

To exit plot edit mode, click the selection button or click the **Edit Plot** option on the **Tools** menu. When plot edit mode is turned off, the selection button is no longer depressed.

Selecting Objects in a Graph

To select an object in a graph,

1. Start plot edit mode.
2. Move the cursor over the object and click it.

   Selection handles appear on the selected object.

Note When you manually select an object, its `Selected` property is set to `on`. Handles appear regardless of the setting of its `SelectionHighlight` property (which when `off` prevents handles from appearing outside of plot edit mode). Plot edit mode does not consider objects selected with `set` (as in `set(h,'Selected','on')`) to be selected, even if they have selection handles. Programmatically selected objects therefore do not respond to actions such as typing **Delete**. They can be dragged, however, because doing so selects them manually.

Selecting Multiple Objects

To select multiple objects at the same time,

1. Start plot edit mode.
2. Move the cursor over an object and **Shift**+click to select it. Repeat for each object you want to select.

You can perform actions on all the selected objects. For example, to remove a textbox annotation and an arrow annotation from a graph, select the objects and then select **Delete** or **Cut** from the **Edit** menu (**Cut** keeps a copy on the clipboard, **Delete** does not).

Deselecting Objects

To deselect an object, move the cursor off the object onto the figure window background and click the left mouse button (this deselects all selected objects and selects the one you clicked). You can also **Shift**+click on a selected object to deselect it (doing this will not deselect any other object).

Cutting, Copying, and Pasting Plot Objects

To cut an object from a graph, or copy and paste an object in a graph, perform these steps:

1. Start plot edit mode.
2. Select the object.
3. Select the **Cut**, **Copy**, or **Paste** option from the **Edit** menu or use standard shortcut keys for your platform.

Alternatively, with plot edit mode enabled, you can right-click on an object and then select an editing command from the context menu associated with the object.

Copying and Pasting Multiple Objects

When you cut or copy axes and plot objects such as lineseries or barseries from a figure and paste them there or elsewhere, the results depend on what you select and the type of container into which you paste the objects.

Copy and Paste Axes. The following semantics apply to copying and pasting axes into the same or different figure:

Select, Copy, and Paste Axes	Result of Pasting Axes
Select axes Ax1 from figure Fig1, copy and paste it into Fig2, which has no axes.	New axes Ax2 is created in Fig2. Ax2 inherits all properties of Ax1, including all children. Ax1 will be selected in Fig1; Ax2 will be selected in Fig2.
Select axes Ax1 from figure Fig1, copy and paste (from Edit Menu) into Fig2, that contains axis Ax2 which is not selected.	New axes Ax3 is created in Fig2. All children of Ax1 are copied to Ax3. All the selected objects in Fig2 are deselected, and the pasted axes Ax3 is selected. The selections in Fig1 are unchanged.
Select axes Ax1 from figure Fig1, copy and paste into Ax2 in the same or different figure.	New axis Ax3 is created having the same properties (including position) and children as Ax1; any selected objects in Ax2 are deselected, and axes Ax3 is selected. When pasting to a new figure, selections in Fig1 will be unchanged.
Select axes Ax1 from figure Fig1, copy and paste into same or different axes in Fig1.	New axes Ax2 is added to Fig1, offset from Ax1, and is the only selected object.

Note When an axis is pasted into an existing axes, the pasted axes becomes a peer of the existing axes and is offset slightly to visually indicate that the paste operation was successful.

Copy and Paste Plot Objects. The following semantics apply to copying and pasting plot objects (lines are used as examples) from one axes into the same or different figure:

Select, Copy, and Paste Objects	Result of Pasting Objects
Select and copy one or more lines from axes Ax1 and paste into selected axes Ax2 in the same or different figure.	The lines are added to Ax2; the pasted lines are the only selected objects in the destination figure.
Select and copy lines from axes Ax1 in Fig1 and paste into figure Fig2, which contains no axes or has no axes selected.	New axes Ax2 is created in Fig2 containing the lines, which are selected in it; Ax2 has default axes properties.
Select and copy lines from axes Ax1 and paste into selected axes Ax2 and Ax3.	Lines are pasted into both Ax2 and Ax3; all the pasted lines are selected.
Select and copy lines from axes Ax1 and paste into selected axes Ax1.	Nothing is pasted, as the extra content would be redundant.

Copy and Paste Plot Objects from Multiple Axes. The following semantics apply to copying and pasting plot objects (lines are used as examples) from one or more axes into the same or different figure:

Select, Copy, and Paste Objects	Result of Pasting Objects
Select and copy Line1 from axes Ax1 and Line2 from axes Ax2 and paste into axes Ax3.	Two lines are pasted into axes Ax3 and are the only selected objects there.
Select and copy lines from axes Ax1 and axes Ax2 and paste into figure Fig2, which contains no axes or has no axes selected.	New axes Ax3 is created in Fig2, into which all the lines are pasted; Ax3 has default axes properties.
Select and copy multiple lines from Axes Ax1 and axes Ax2 and paste into Ax1, Ax2, or some other axes.	Only those lines that did not originate in an axes are pasted into it, and the pasted lines are the only selected objects.

Copy and Paste Multiple Axes and Plot Objects. The following semantics apply to copying and pasting several axes and selected plot objects (lines are used as examples) from one or more axes into the same or different figure:

Select, Copy, and Paste Objects	Result of Pasting Objects
Select Line1 from axes Ax1, select axes Ax2, and paste into figure Fig2, which contains no axes or has no axes selected.	Ax2 and its contents is pasted as new axes Ax3; another new axis Ax4 is created into which the line is pasted
Select Line1 from axes Ax1, select axes Ax2, and paste into axes Ax3.	Line1 is pasted into axes Ax3 and axes Ax2 is pasted as new axis Ax4.
Select axes Ax1 and Line1 from Ax1, and paste into Ax1.	New axes Ax2 is created having all the properties of Ax1 but containing Line1 as its only child.
Select axes Ax1 and Line1 from Ax1 in Fig1 and paste into figure Fig2, which contains no axes or has no axes selected.	Line1 is pasted in new axes Ax2, and Ax1 and its children (including Line1) is pasted as new axis Ax3.
Select axes Ax1 and Line1 from Ax1, and paste into axes Ax2.	New axes Ax3 is created having all the properties of Ax1 but containing Line1 as its only child.

Copying and Pasting Annotation Objects

In plot edit mode you can copy and paste annotations such as textboxes, textarrows, rectangles, and ellipses, in various combinations. If any such objects happen to be pinned to their axes (see “Pinning — Attaching to a Point in the Graph” on page 3-23), their copies are pasted unpinned. As annotation objects are children of figures, they never create new axes when you paste them.

Moving and Resizing Objects

To move or resize an object in a graph, perform these steps:

1. Start plot edit mode.
2. Select the object. Selection handles appear on the object When the cursor is over the object, it turns into crossed arrows; outside the selection it reverts to a pointer.

To move the object, drag it to the new location. You can also nudge it one pixel up, down, left, or right with the appropriate arrow key on your keyboard. If you have selected **Snap to Layout Grid** from the **Tools** menu, each keypress makes objects move to the next grid position.

To resize the object, drag a selection handle.

You can shift-click to select multiple objects and move them as a group. Arrow keys work well for this. However, when you resize one of several selected objects, only that object changes size.

Note You can move text objects, but you cannot resize them (annotation text boxes can be resized, however). You can resize axes objects, but you can only move them by dragging their edges (or via their selection handles, one at a time).

Setting Object Properties

In MATLAB, every object in a graph supports a set of properties that control the graph's appearance and behavior. For example, line objects have properties that control thickness, color, and line style.

Double-clicking on an object displays the Property Editor. To edit the properties of the axes or figure, double-click on a region that does not contain other objects.

See "The Property Editor" on page 1-28 for more information.

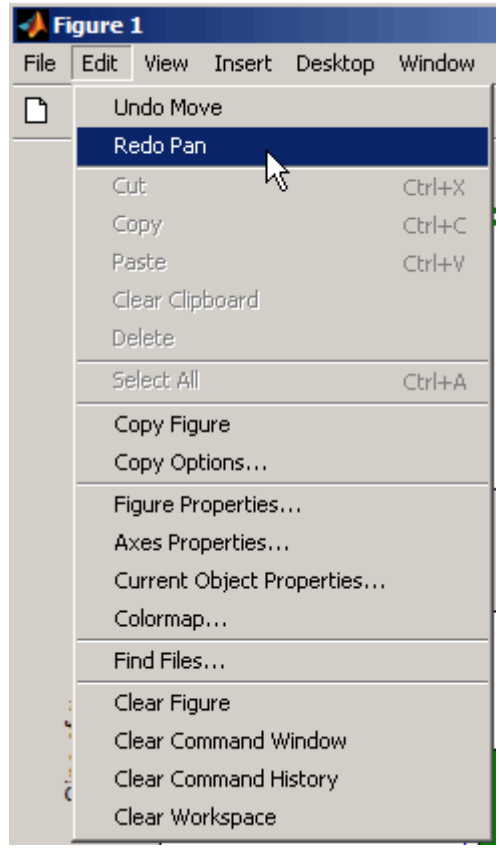
Undo/Redo — Eliminating Mistakes

The figure **Edit** menu contains two items that enable you to undo recent operation.

Undo — Remove the effect of the last operation.

Redo — Perform again the last operation that you removed by selecting **Undo**.

For example, if you create a plot, zoom in, pan the view, and then undo the pan operation, the menu looks as follows:



You could now undo the previous zoom operation or redo the pan operation you just undid.

Saving Your Work

In this section...

Saving a Graph in MAT-File Format

Note To save a figure in a format that is compatible with MATLAB versions before MATLAB 7, refer to "Plot Objects and Backward Compatibility" on page 8-22 for more information.

MATLAB supports a binary format in which you can save figures so that they can be opened in subsequent MATLAB sessions. MATLAB assigns these files the `.fig` filename extension.

To save a graph in a figure file,

1. Select **Save** from the figure window **File** menu or click the **Save** button on the toolbar. If this is the first time you are saving the file, the **Save As** dialog box appears.
2. Make sure that the `Save as type` is **MATLAB Figure (*.fig)**.
3. Specify the name you want assigned to the figure file.
4. Click **OK**.

The graph is saved as a figure file (`.fig`), which is a binary file format used to store figures.

You can also use the `saveas` command.

Use the `hgsave` command to create backward compatible FIG-files.

If you want to save the figure in a format that can be used by another application, see “Saving to a Different Format — Exporting Figures” on page 1-69.

Opening a Figure File

To open a figure file, perform these steps:

1. Select **Open** from the **File** menu or click the **Open** button on the toolbar.
2. Select the figure file you want to open and click **OK**.

   The figure file appears in a new figure window.

You can also use the `open` command.

Saving to a Different Format — Exporting Figures

To save a figure in a format that can be used by another application, such as the standard graphics file formats TIFF or EPS, perform these steps:

1. Select **Export Setup** from the **File** menu. This dialog provides options you can specify for the output file, such as the figure size, fonts, line size and style, and output format.
2. Select **Export** from the Export Setup dialog. A standard Save As dialog appears.
3. Select the format from the list of formats in the `Save as type` drop-down menu. This selects the format of the exported file and adds the standard filename extension given to files of that type.
4. Enter the name you want to give the file, less the extension.
5. Click **Save**.

Copying a Figure to the Clipboard

On Windows systems, you can also copy a figure to the clipboard and then paste it into another application:

1. Select **Copy Options** from the **Edit** menu. The **Copying Options** page of the **Preferences** dialog box appears.
2. Complete the fields on the **Copying Options** page and click **OK**.
3. Select **Copy Figure** from the **Edit** menu.

   The figure is copied to the Windows clipboard. You can then paste the figure from the Windows clipboard into a file in another application.

Printing Figures

Before printing a figure,

1. Select **Print Preview** from the **File** menu to set printing options, including plot size and position, and paper size and orientation.

   The **Print Preview** dialog box opens.
2. Make changes in the dialog box. Changes you can make are arranged by tabs on the left-hand pane. If you want the printed output to match the annotated plot you see on the screen exactly,
   a. On the **Layout** tab, click **Auto (Actual Size, Centered)**.
   b. On the **Advanced** tab, click **Keep screen limits and ticks**.

   For information about other options for print preview, click the **Help** button in the dialog box.

To print a figure, select **Print** from the figure window **File** menu and complete the **Print** dialog box that appears.

You can also use the `print` command.

Generating an M-File to Recreate a Graph

You can generate an M-file from a graph, which you can then use to regenerate the graph. This approach is a useful way to generate M-code for work you have performed with the plotting tools. To use this option,

1. Select **Generate M-file** from the **File** menu.

MATLAB displays the generated code in the MATLAB Editor.

2 Save the M-file using **Save As** from the Editor **File** menu.

Running the Saved M-File

Most of the generated M-files require you to pass in data as arguments. The M-file assumes you are using the same data originally used to create the graph.

Comments at the beginning of the M-file state the type of data expected. For example, the following statements illustrate a case where three input vectors are required.

```
function createplot(X1, Y1, Y2)
%CREATEPLOT(X1,Y1,Y2)
%  X1:  vector of x data
%  Y1:  vector of y data
%  Y2:  vector of y data
```

See “Example — Generating M-Code to Reproduce a Graph” on page 1-53 for another example.

2

Data Exploration Tools

Ways to Explore Graphical Data

In this section...

Introduction

After determining what type of graph best represents your data, you can further enhance the visual display of information using the tools discussed in this section. These tools enable you to explore data interactively, eliminating the need to set the plethora of graphics properties required to achieve the same results using MATLAB commands.

Once you have achieved the desired results, you can then generate the MATLAB code necessary to reproduce the graph you created interactively. See "Example — Generating M-Code to Reproduce a Graph" on page 1-53 for more information.

Types of Tools

See the following sections for information on specific tools.

- "Data Cursor — Displaying Data Values Interactively" on page 2-4
- "Enlarging the View" on page 2-19
- "Panning — Shifting Your View of the Graph" on page 2-23
- "Rotate 3D — Interactive Rotation of 3-D Views" on page 2-25
- Camera Toolbar — Interacting with 3-D Views

You can perform data analysis directly on graphs with curve fitting and time series tools; see

- "Linear Regression Analysis"
- "Interactive Fitting"
- "Time Series Tools"

These and other topics are covered in the “Preparing Data for Analysis” section of the Data Analysis documentation. You can also use `cftool` if you have Curve Fitting Toolbox.

Data Cursor — Displaying Data Values Interactively

In this section...

What Is a Data Cursor?

Data cursors enable you to read data directly from a graph by displaying the values of points you select on plotted lines, surfaces, images, and so on. You can place multiple datatips in a plot and move them interactively. If you save the figure, the datatips in it are saved, along with any other annotations present.

When data cursor mode is enabled, you can

- Click on any graphics object defined by data values and display the x, y, and z (if 3-D) values of the nearest data point.
- Interpolate the values of points between data points.
- Display multiple data tips on graphs.
- Display the data values in a cursor window that you can locate anywhere in the figure window or as a data tip (small text box) located next to the data point.
- Export data values as workspace variables.
- Print or export the graph with data tip or cursor window displayed for annotation purposes.
- Edit the data tip display function to customize what information is displayed and how it is presented
- Select a different data tip display function

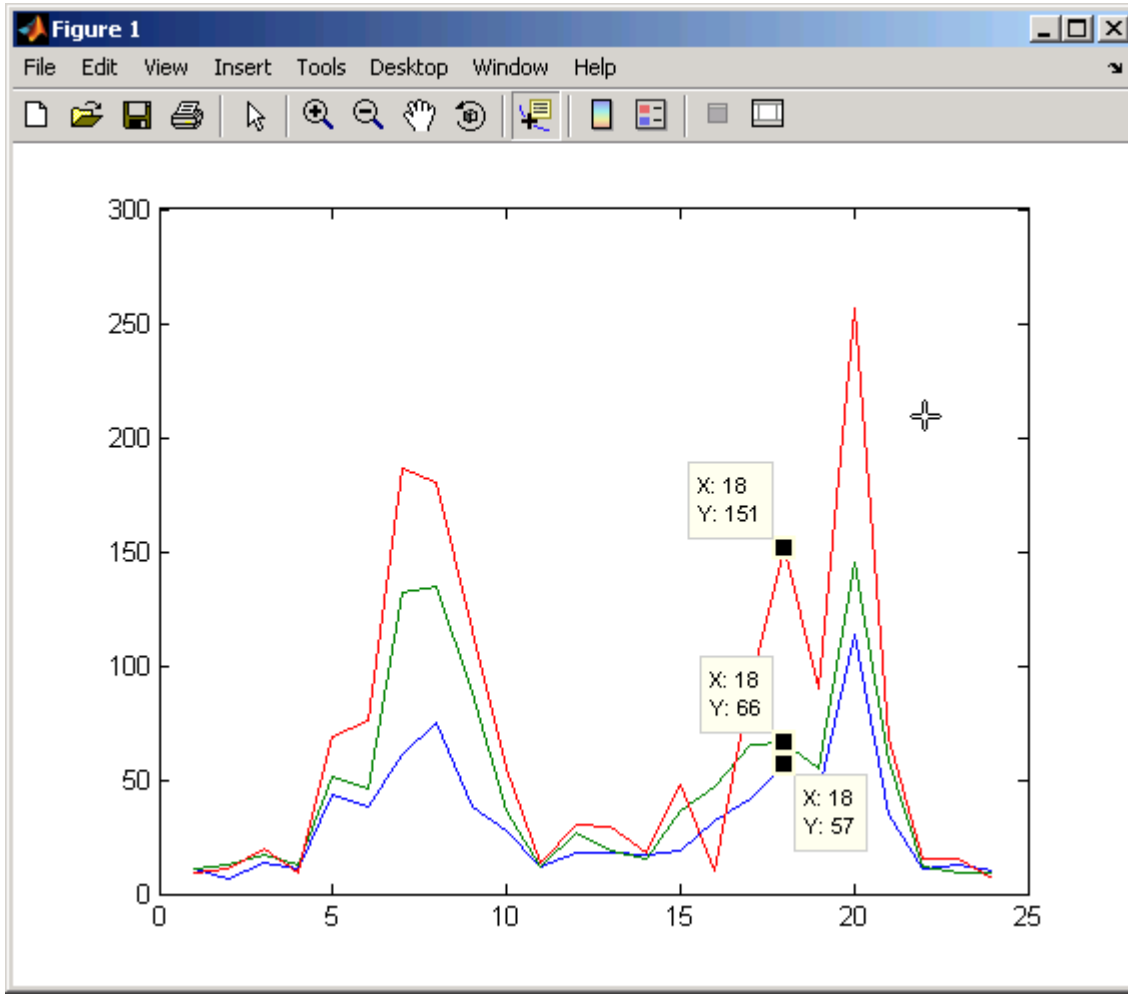
Enabling Data Cursor Mode

Select the data cursor icon in the figure toolbar  or select the **Data Cursor** item in the **Tools** menu.

Once you have enabled data cursor mode, clicking the mouse on a line or other graph object displays data values of the point clicked. Clicking elsewhere does not create or update data tips. To place additional data tips, as the picture below shows, see “Creating Multiple Data Tips” on page 2-11, below. In the picture, the black squares are located at points selected by the Data Cursor tool, and the data tips next to them display the x and y-values of those points.

The illustrations below use traffic count data stored in `count.dat`:

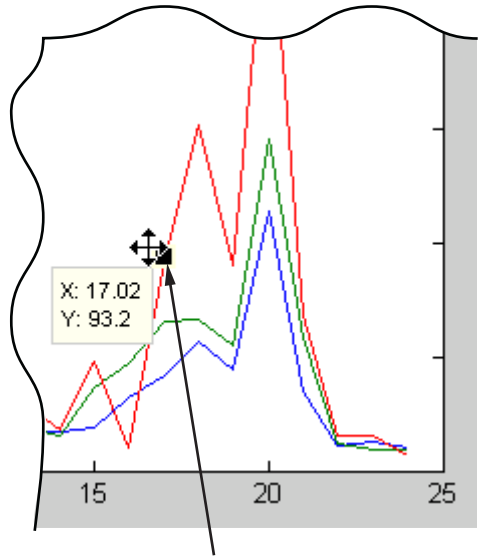
```
load count.dat
plot(count)
```



Moving the Marker

You can move the marker using the arrow keys and the mouse. The up and right arrows move the marker to data points having greater index values in the data arrays. The down and left arrow keys move the marker to data points having lesser index values. When you set **Selection Style** to **Mouse Position** using the tool's context menu, you can drag markers and position them anywhere along a line. However, you cannot drag markers between different

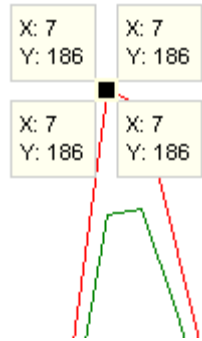
line or other series on a plot. The cursor changes to crossed arrows when it comes close enough to a marker for you to drag the datatip, as shown below:



Positioning the Datatip Text Box

You can position the data tip text box in any one of four positions with respect to the data point: upper right (the default), upper left, lower left, and lower right.

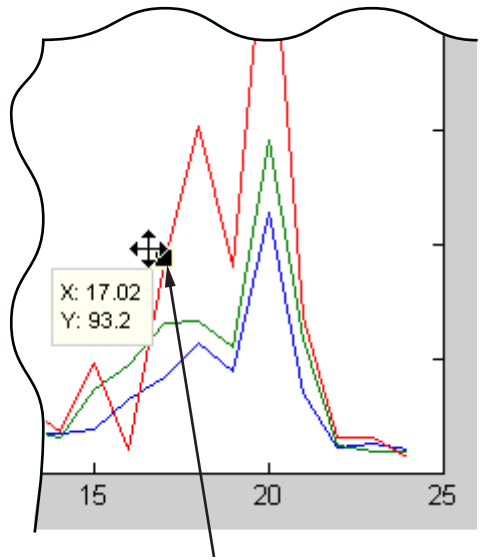
To position the datatip, press, but do not release the mouse button while over the datatip text box and drag it to one of the four positions, as shown below:



You can reposition a datatip, but not its text box, using the arrow keys as well.

Dragging the Datatip to Different Locations

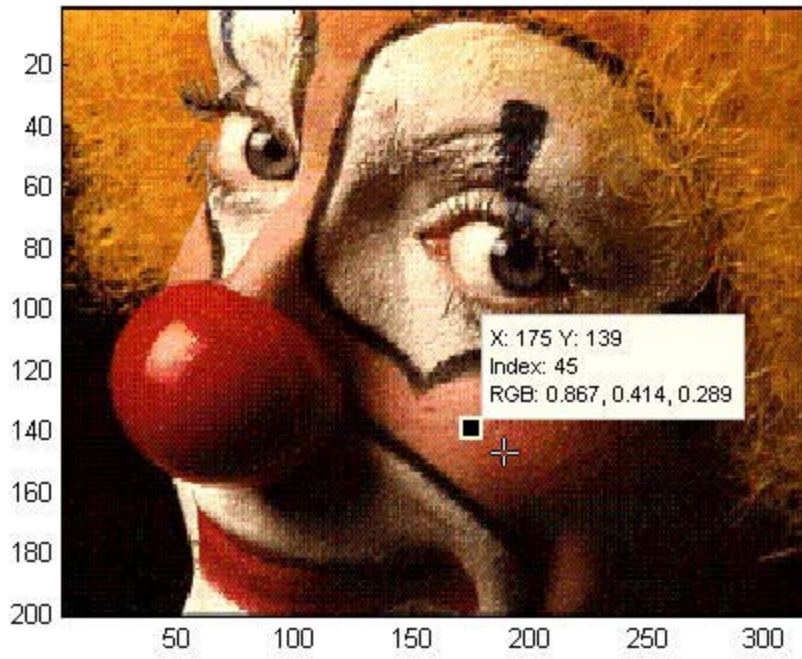
You can drag the datatip to different locations on the graph object by clicking down on the datatip and dragging the mouse. You can also use the arrow keys to move the datatip.



Note Surface plots and 3-D bar graphs can contain `NaN` values. If you drag a datatip to a location coded as `NaN`, the datatip will disappear (because its coordinates become `(NaN,NaN,NaN)`). You can continue to drag it invisibly, however, and it will reappear when it is over a non-`NaN` location. However, if you create a new datatip while the previous current one is invisible, the previous one cannot be retrieved.

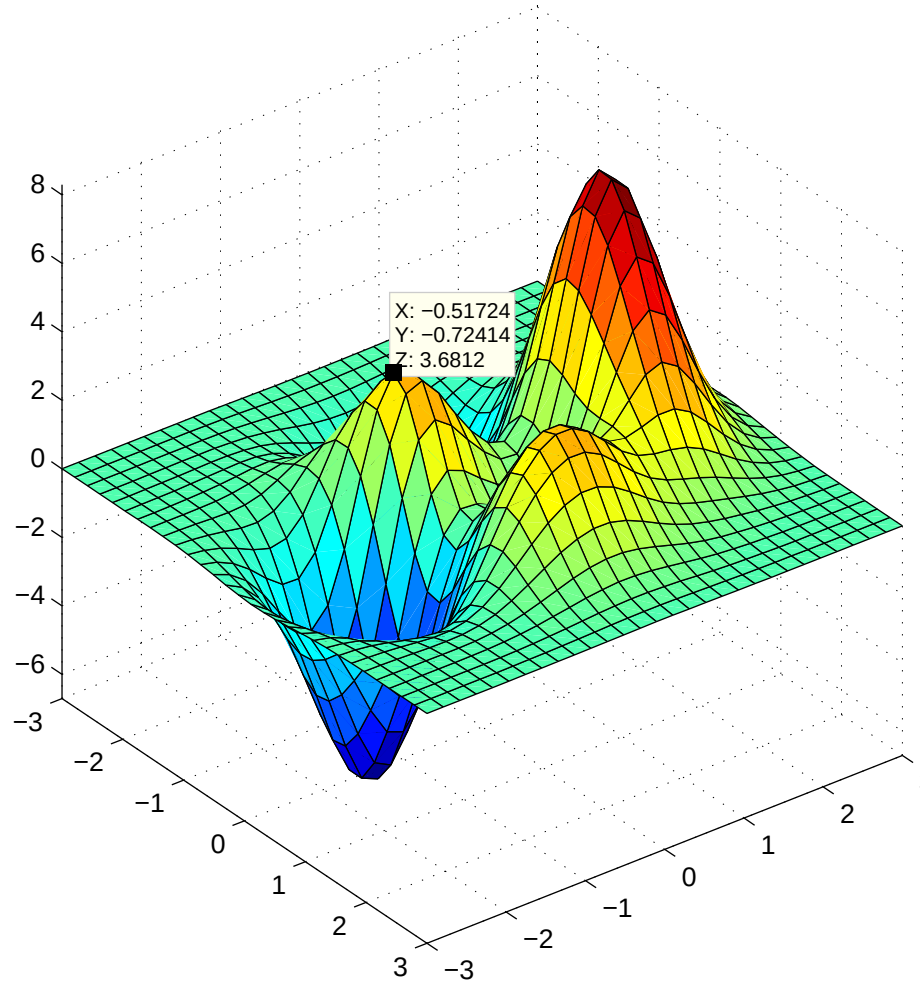
Datatips on Image Objects

Datatips on images display the *x*- and *y*-coordinates as well as the RGB values and a color index (for indexed images), as show below:



Datatips on 3-D Objects

You can use datatips to read data points on 3-D graphs as well. In 3-D views, data tips display the *x*-, *y*- and *z*-coordinates.

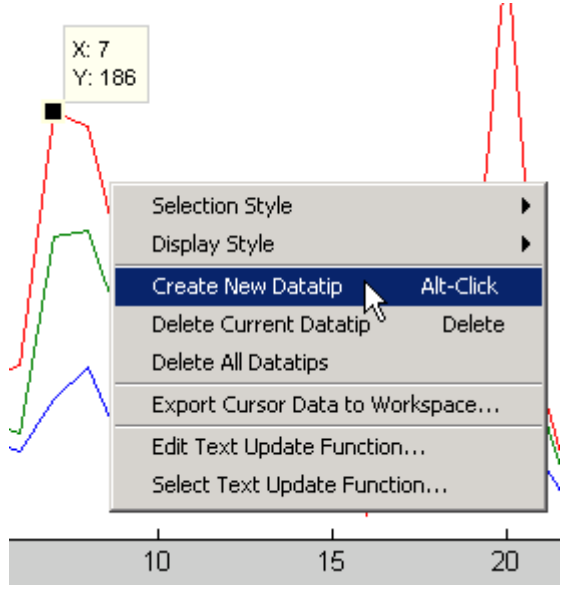

X: −0.51724
Y: −0.72414
Z: 3.6812
8
6
4
2
0
−2
−4
−6
−3
−2
−1
0
1
2
3
−3
−2
−1
0
1
2
3

Creating Multiple Data Tips

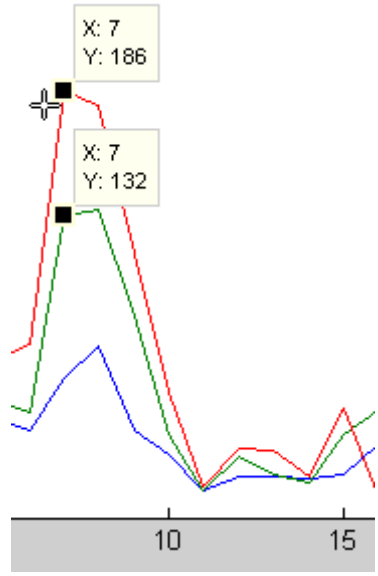
Normally, there is only one datatip displayed at one time. However, you can display multiple datatips simultaneously on a graph. This is a simple way to annotate a number of points on a graph.

Use the following procedure to create multiple datatips.

1. Enable data cursor mode from the figure toolbar. The cursor changes to a cross.
2. Click on the graph to insert a datatip.
3. Right-click to display the context menu. Select **Create New Datatip**.

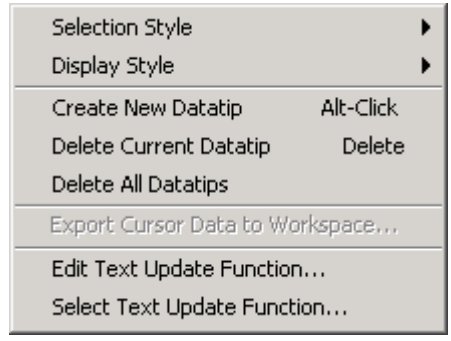


4. Click on the graph to place the second datatip.



Customizing Data Cursor Text

You can customize the text displayed by the data cursor using the `datacursormode` function. Use the last two items in the Data Cursor context menu to for this purpose:



- **Edit Text Update Function** — Opens an editor window to let you modify the function currently being used to place text in datatips

- **Select Text Update Function** — Opens an input file dialog for you to navigate to and select an M-file to use to format text in datatips you subsequently create

When you select **Edit Text Update Function** for the first time, an editor window opens with the default text update callback, which consists of the following code:

```
function output_txt = myfunction(obj,event_obj)
% Display the position of the data cursor
% obj          Currently not used (empty)
% event_obj    Handle to event object
% output_txt   Data cursor text string (string or cell array of strings).

pos = get(event_obj,'Position');
output_txt = {['X: ',num2str(pos(1),4)],...
    ['Y: ',num2str(pos(2),4)]};

% If there is a Z-coordinate in the position, display it as well
if length(pos) > 2
    output_txt{end+1} = ['Z: ',num2str(pos(3),4)];
end
```

You can modify this code to display properties of the graphics object other than position. If you want to do so, you should first save this code to an M-file before changing it, and select that file if you want to revert to default datatip displays during the same session.

If for example you save it as `def_datatip_cb.m`, and then modify the code and save it to another M-file, you can then choose between the default behavior and customized behavior by choosing **Select Text Update Function** from the context menu and selecting one of the callbacks you saved.

See the Examples section of the `datacursormode` reference page for more information on using data cursor objects and update functions.

Deleting Datatips

You can remove the most recently added datatip or all datatips. When in data cursor mode, right-click to display the context menu.

- Select **Delete Current Datatip** or press the **Delete** key to remove the last datatip that you added.
- Select **Delete All Datatips** to remove all datatips.

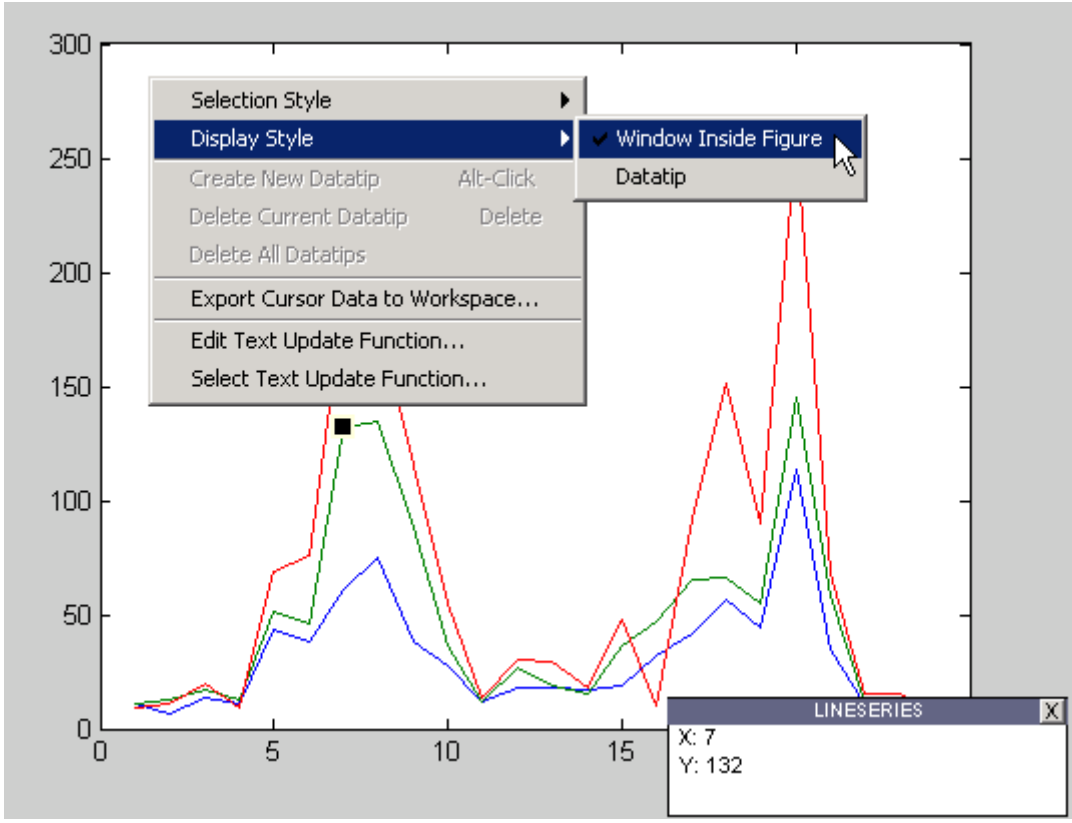
Display Style — Datatip or Cursor Window

By default, the data cursor displays values as a datatip (small text box located next to the data point). You can also display a single data value in a cursor window that is anchored within the figure window. You can place multiple datatips on a graph, which makes this display style useful for annotations.

The cursor window style is particularly useful when you want to drag the data cursor to explore image and surface data; numeric information in the window updates without obscuring the any of the figure's symbology.

To use the cursor window, change the display style as follows:

1. While in data cursor mode, right-click to display the context menu.
2. Mouse over the **Display Style** item.
3. Select **Window Inside Figure**.



Note If you change the data cursor **Display Style** from **Datatip** to **Window Inside Figure** with the context menu, only the most recent data tip is displayed; all other existing data tips are removed because the window can display only one datatip at a time.

Selection Style — Select Data Points or Interpolate Points on Graph

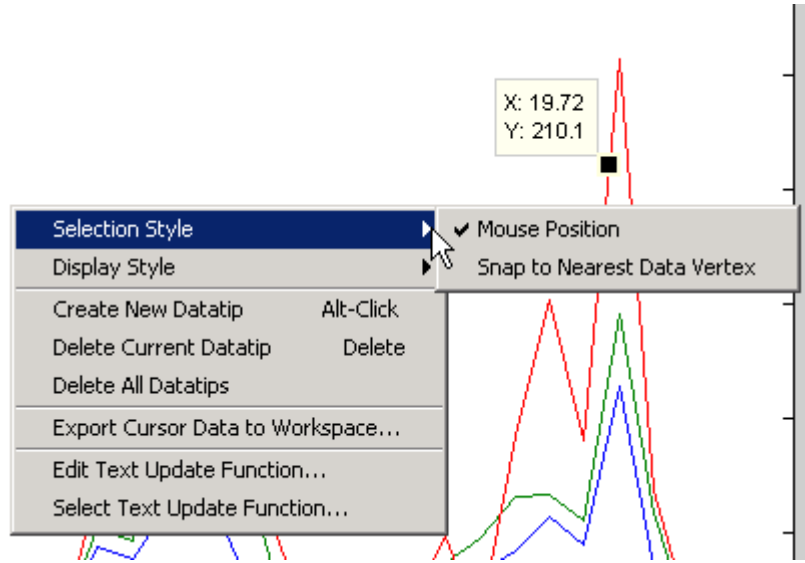
By default, the data cursor displays the values of the data point nearest to the point you click with the mouse, and the data marker snaps to this point. The data cursor can also determine the values of points that lie in between

the data defining the graph, by linearly interpolating between the two data points closest to the location you click the mouse.

Enabling Interpolation Mode

If you want to be able to select any point along a graph and display its value, use the following procedure:

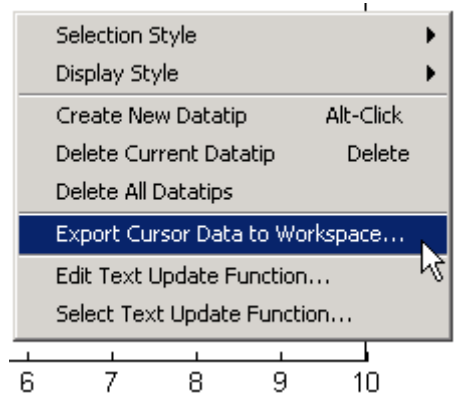
1. While in data cursor mode, right-click to display the context menu.
2. Mouse over the **Selection Style** item.
3. Select **Mouse Position**.



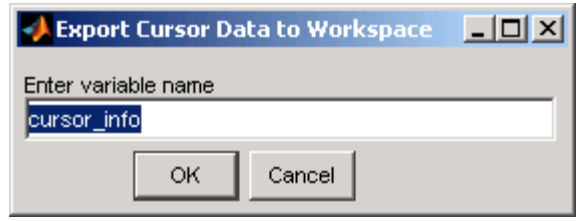
Note that interpolation mode is not honored when you are using the arrow keys to move a datatip to a new location.

Exporting Data Value to Workspace Variable

You can export the values displayed with the data cursor to MATLAB workspace variables. To do this, display the right-click context menu while in data cursor mode and select **Export Cursor Data to Workspace**.



MATLAB then displays a dialog that enables you to name the workspace variable.



When you click **OK**, MATLAB creates a structure with the specified name in your base workspace, containing the following fields:

- `Target` — Handle of the graphics object containing the data point
- `Position` — *x*- and *y*- (and *z*-) coordinates of the data cursor location in axes data units

Line and lineseries objects have an additional field:

- `DataIndex` — A scalar index into the data arrays that correspond to the nearest data point. The value is the same for each array.

For example, if you saved the workspace variable as `cursor_info`, then you would access the position data by referencing the `Position` field.

```
cursor_info.Position
ans =
    0.4189  0.1746  0
```

Enlarging the View

In this section...

Zooming in 2-D and 3-D

Zooming changes the magnification of a graph without changing the size of the figure or axes. Zooming is useful to see greater detail in a small area. As explained below, zooming behaves differently depending on whether it is applied to a 2-D or 3-D view.

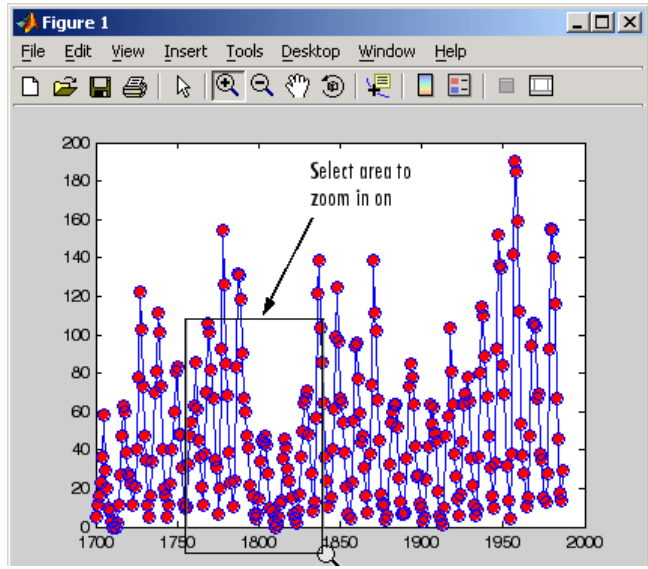
Enable zooming by clicking one of the zoom icons  . Select **+** to zoom in and **–** to zoom out.

Note, when in zoom in mode, you can use **Shift**+click to zoom out (i.e., press and hold down the **Shift** key while clicking the mouse). You can also right-click and zoom out or restore the plot to its original view using the context menu.

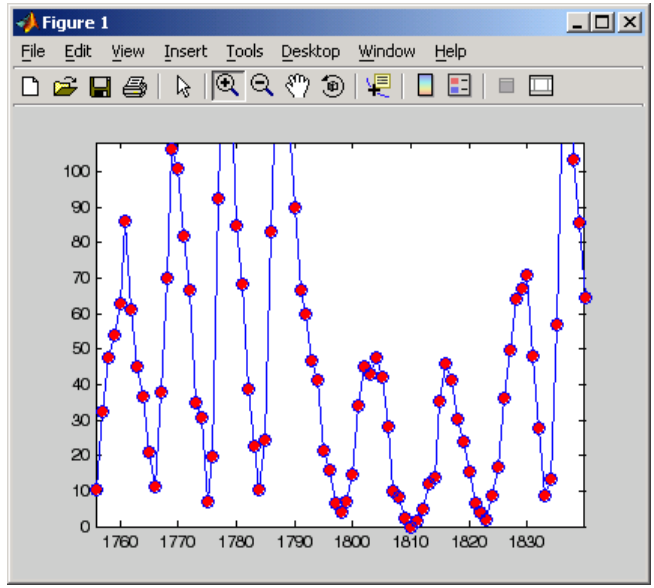
Zooming in 2-D Views

In 2-D views, click the area of the axes where you want to zoom in, or drag the cursor to draw a box around the area you want to zoom in on. MATLAB redraws the axes, changing the limits to display the specified area.

For example, selecting the region of the following plot,



results in a rescaling of the axes to display only that region.

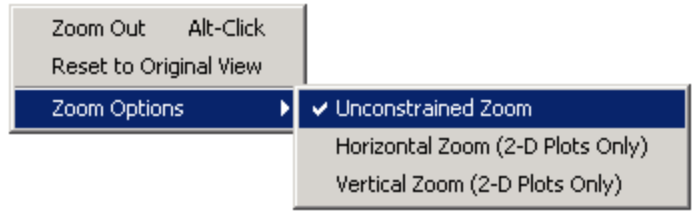


Undoing Zoom Actions

If you want to reset the graph to its original view, right-click to display the context menu and select **Reset to Original View**. You can also use the **Undo** item on the **Edit** menu to undo each operation you performed on your graph.

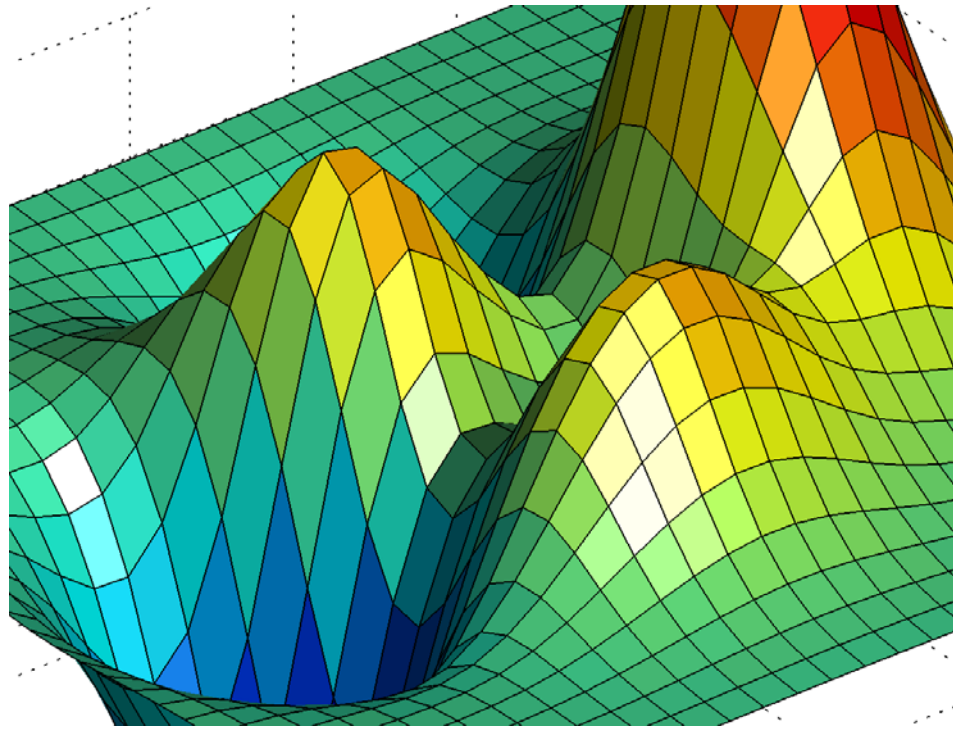
Zoom Constrained to Horizontal or Vertical

In 2-D views, you can constrain zoom to operate in either the horizontal or vertical direction. To do this, right-click to display the context menu while in zoom mode and select the desired zoom option.



Zooming in 3-D Views

In 3-D views, moving the cursor up or to the right zooms in, while moving the cursor down or to the left zooms out. Both toolbar icons enable the same behavior. 3-D zooming does not change the axes limits, as in 2-D zooming. Instead it changes the view (specifically, the axes `CameraViewAngle` property) as if you were looking through a camera with a zoom lens.

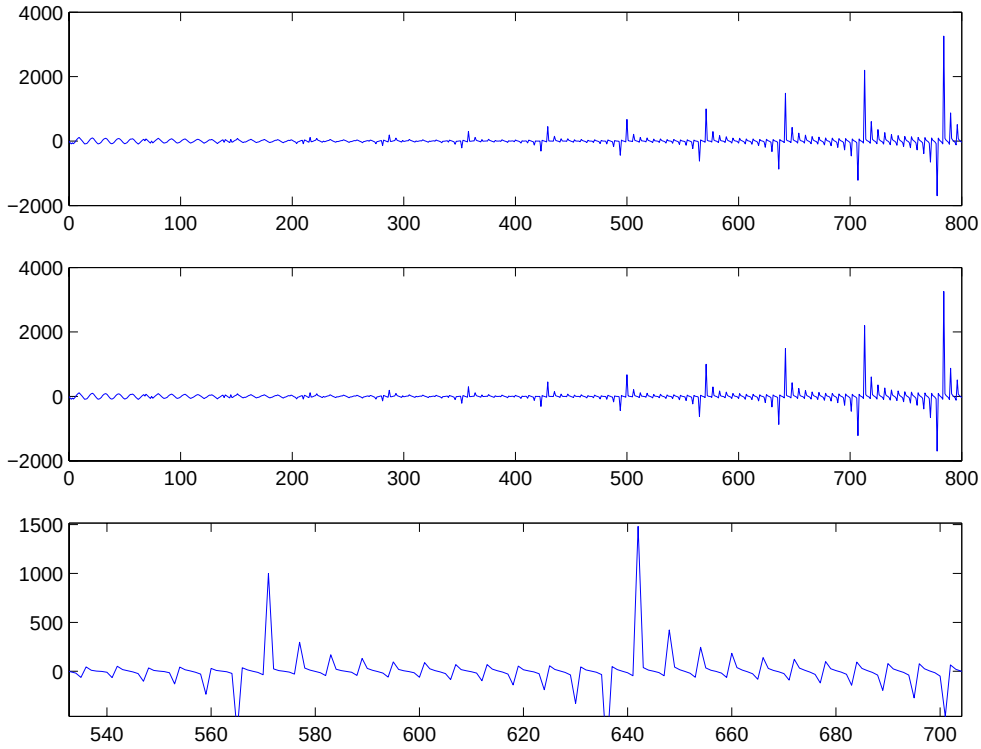


Panning — Shifting Your View of the Graph

You can move your view of a graph up and down as well as left and right with the pan tool. Panning is useful when you have zoomed in on a graph and want to translate the plot to view different portions.

Click this icon on the figure toolbar to enable panning  . In pan mode you can freely move up, down, left or right, or you can constrain movement to be vertical or horizontal only by right-clicking and selecting one of the **Pan Options** from the pan tool's context menu.

You can pan across both 2-D and 3-D views. 2-D panning has the effect of changing the axis limits that you are viewing, but it does not change the actual limits of the plot. For example, suppose you have a time-series waveform that you want to zoom in on to view detail, but you also want to be able to scan the entire plot.



3-D panning moves the axes with the object, because the 3-D view is not aligned to the *x*-, *y*-, or *z*-axis. The axes limits do not change as in 2-D panning.

Rotate 3D — Interactive Rotation of 3-D Views

In this section...

Enabling 3-D Rotation

MATLAB enables you to rotate graphs to any orientation with the mouse. Rotation involves the reorientation of the axes and all the graphics objects it contains. Therefore none of the data defining the graphics objects is affected by rotation; instead the orientation of the *x*-, *y*-, and *z*-axes changes with respect to the viewer.

There are three ways to enable Rotate 3D mode:

- Select **Rotate 3D** from the **Tools** menu.
- Click the Rotate 3D icon in the figure toolbar  .
- Execute the `rotate3d` command.

Once the mode is enabled, you press and hold the mouse button while moving the cursor to rotate the graph.

Selecting Predefined Views

When Rotate 3D mode is enabled, you can control various rotation options from the right-click context menu.

You can rotate to predefined views on the right-click context menu:

- **Reset to Original View** — Reset to the default MATLAB view (azimuth -37.5°, elevation 30°).
- **Go to X-Y View** — View graph along the *z*-axis (azimuth 0°, elevation 90°).

- **Go to X-Z View** — View graph along the *y*-axis (azimuth 0°, elevation 0°).
- **Go to Y-Z View** — View graph along the *x*-axis (azimuth 90°, elevation 0°).

Rotation Style for Complex Graphs

You can select from two rotation styles on the right-click context menu's **Rotation Options** submenu:

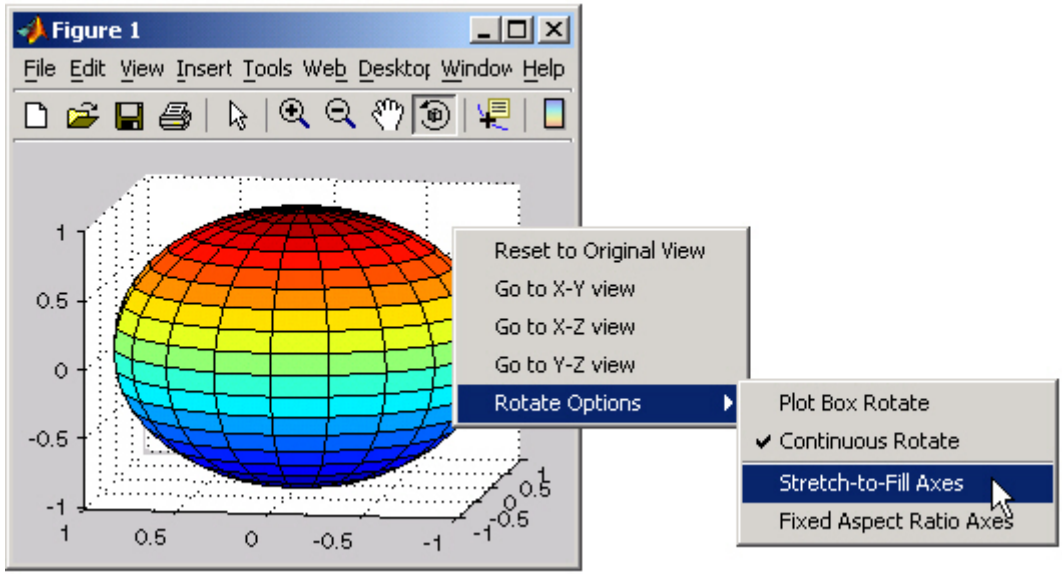
- **Plot Box Rotate** — Display only the axes bounding box for faster rotation of complex objects. Use this option if the default **Continuous Rotate** style is unacceptably slow.
- **Continuous Rotate** — Display all graphics during rotation.

Axes Behavior During Rotation

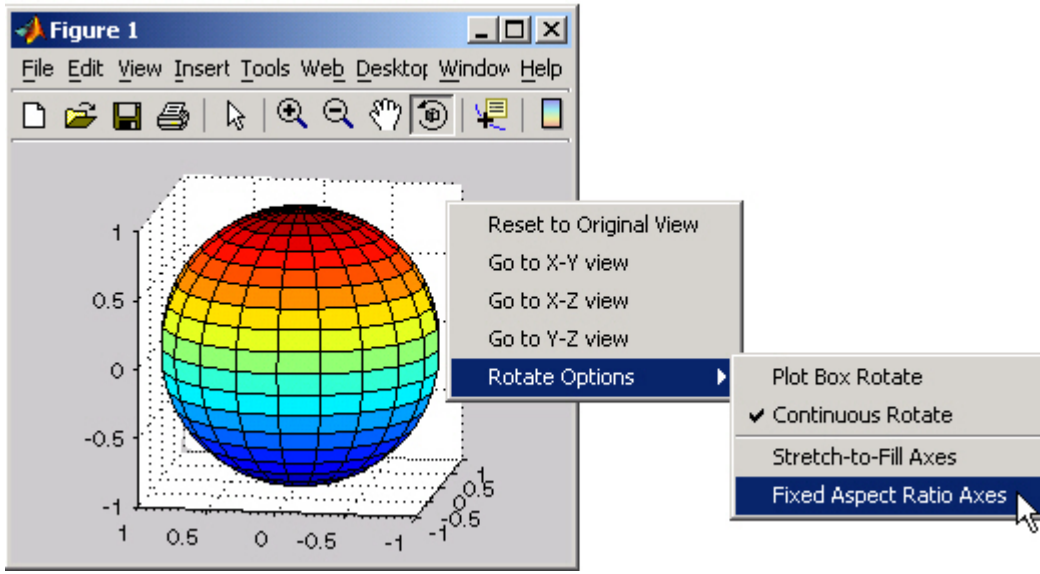
You can select two types of behavior with respect to the aspect ratio of axes during rotation:

- **Stretch-to-Fill Axes** – Default axes behavior is optimized for 2-D plots. Graphs fit the rectangular shape of the figure.
- **Fixed Aspect Ratio Axes** – Maintains a fixed shape of objects in the axes as they are rotated. Use this setting when rotating 3-D plots.

The following pictures illustrate a sphere as it is rotated with **Stretch-to-Fill Axes** selected. Note that the sphere is not round.



The next picture shows how the **Fixed Aspect Ratio Axes** option results in a sphere that maintains its proper shape as it is rotated.



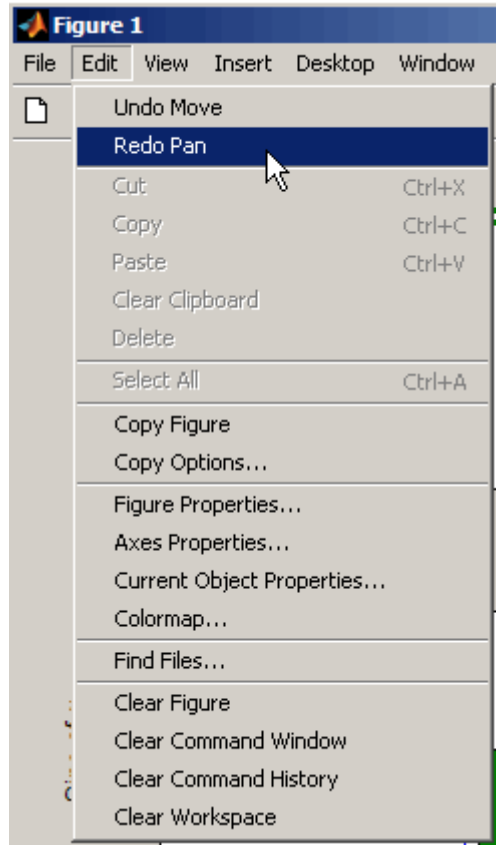
Undo/Redo — Eliminating Mistakes

The figure **Edit** menu contains two items that enable you to undo any zoom, pan, or rotate operation.

Undo — Remove the effect of the last operation.

Redo — Perform again the last operation that you removed by selecting **Undo**.

For example, if you create a plot, zoom in, pan the view, and then undo the pan operation, the menu looks as follows:



You could now undo the previous zoom operation or redo the pan operation you just undid.

3

Annotating Graphs

How to Annotate Graphs

In this section...

Graph Annotation Features

Annotating graphs with text and other explanatory material can improve the graph's ability to convey information. MATLAB provides a variety of features for annotating graphs, including those that

- Add text, lines and arrows, rectangles, ellipses, and other annotation objects anywhere on the figure
- Anchor annotations to locations in data space
- Add a legend and colorbar
- Add axis labels and titles
- Edit the properties of graphics objects

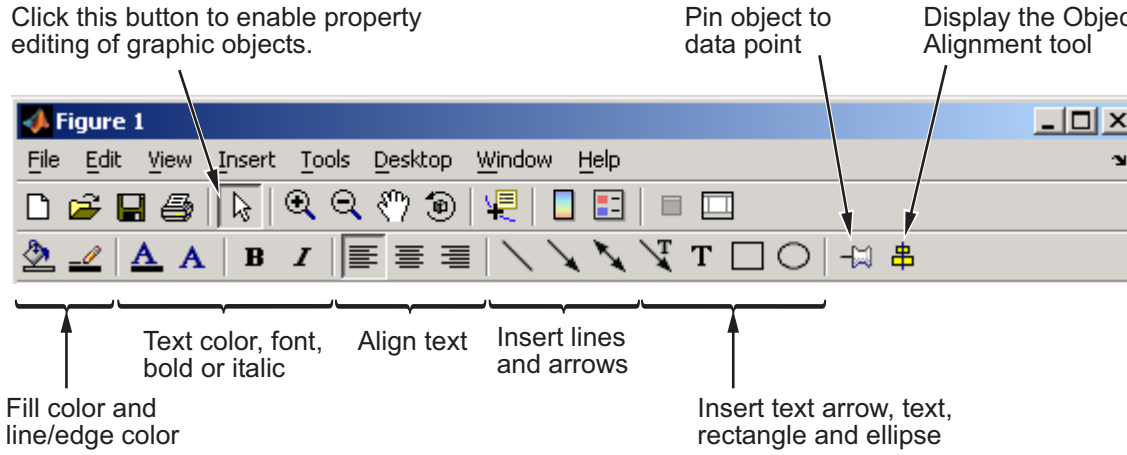
The following sections provide more information.

- "Enclosing Regions of a Graph in a Rectangle or an Ellipse" on page 3-6
- "Textbox Annotations" on page 3-8
- "Annotation Lines and Arrows" on page 3-13
- "Adding a Colorbar to a Graph" on page 3-16
- "Adding a Legend to a Graph" on page 3-20

- "Pinning — Attaching to a Point in the Graph" on page 3-23

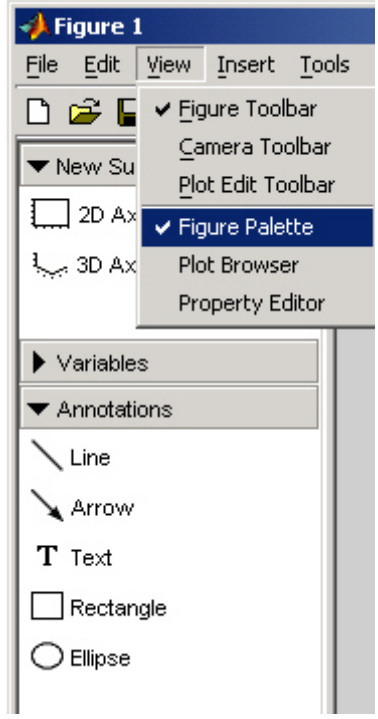
Annotation Tools on the Plot Edit Toolbar

Select **Plot Edit Toolbar** from the **View** menu to display the toolbar.



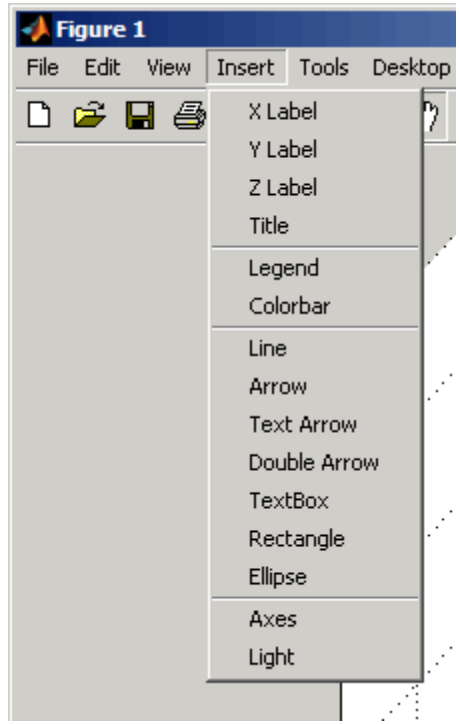
Annotation Tools on the Figure Palette

Basic annotation tools are available from the figure palette. Select **Figure Palette** from the **View** menu to display the figure palette.



Adding Annotations from the Insert Menu

Annotation features are available from the **Insert** menu.



Command Interface

You can add annotations using MATLAB commands. The following table lists the functions used to create annotations.

MATLAB Functions for Creating Annotations

Function	Purpose
`annotation`	Create annotations including lines, arrows, text arrows, double arrows, text boxes, rectangles, and ellipses
`xlabel`, `ylabel`, `zlabel`	Add a text label to the respective axis
`title`	Add a title to a graph

MATLAB Functions for Creating Annotations (Continued)

Function	Purpose
`colorbar`	Add a colorbar to a graph
`legend`	Add a legend to a graph

Removing Annotations

You can delete any annotation manually, and (if it has an explicit handle) programmatically. See “Deleting Annotations” in the MATLAB function reference documentation for details.

Enclosing Regions of a Graph in a Rectangle or an Ellipse

You can add a rectangle or an ellipse to draw attention to a specific region of a graph. While either object is selected, you can move and resize it as well as display a right-click context menu that enables you to modify behavior and appearance.

Insert the rectangle or ellipse by clicking the corresponding button in the plot edit toolbar

or by selecting **Rectangle** or **Ellipse** from the **Insert** menu. MATLAB changes the cursor to a cross indicating you can click down, drag, and release the left mouse button to define the size and shape of the object.

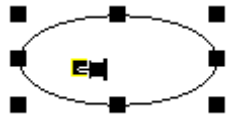
Pinning Rectangles and Ellipses

You can attach the rectangle to a particular point in the figure by pinning it to that point. There are three ways to pin the rectangle:

- Right-click the rectangle to display its context menu. Select **Pin to axes** to set a pin in the default location.

- Select the pin button in the figure toolbar (see “Pinning — Attaching to a Point in the Graph” on page 3-23).
- Select **Pin to axes** from the **Tools** menu. The cursor changes to a pin; click anywhere within the object to set a pin at that location.

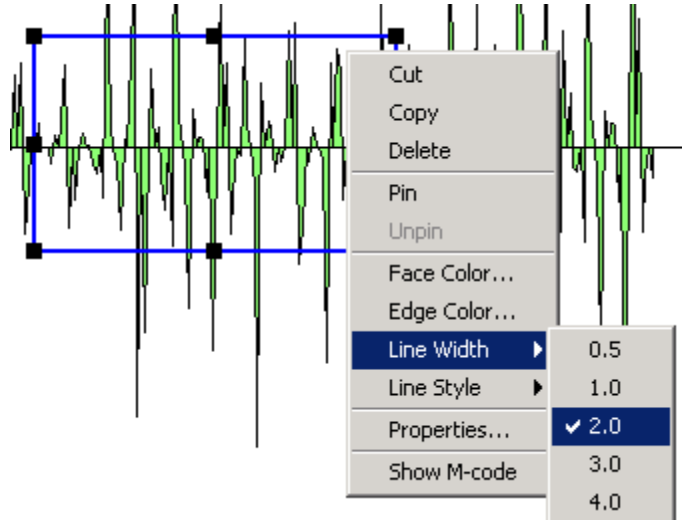
By default (using the first of the options described above), pinning attaches the lower left corner of the rectangle or ellipse to its current location in the axes data units. You can move the point of attachment by clicking the corner and dragging the anchor to another point. The cursor changes to a pin while you are dragging.



Note that you cannot drag or resize a rectangle or an ellipse when it is pinned.

Modifying the Rectangle or Ellipse from the Context Menu

Right-click the rectangle or ellipse to display its context menu.



The menu contains the following options:

- **Cut**, **Copy**, **Delete** — Cut to clipboard, copy to clipboard, or delete the selected object.
- **Pin to axes** — Pin the lower left corner to the current location (you can move the point of attachment by clicking and dragging the point while in plot edit mode).
- **Unpin** — Detach the rectangle from the attachment point.
- **Face Color** — Fill color for the rectangle or ellipse
- **Edge Color** — Color of the line used to draw the rectangle or ellipse
- **Line Width** — Width of the line used to draw the rectangle or ellipse
- **Line Style** — Type of line used to draw the rectangle or ellipse
- **Properties** — Display the Property Editor with textbox properties.
- **Show M-code** — Create M-code that recreates the graph.

Setting Rectangle and Ellipse Properties

You can use the Property Editor to set rectangle and ellipse properties by selecting **Properties** from the context menu. The Property Editor displays the same properties that are described above in the context menu section.

You can click the **More Properties** button on the Property Editor to display the Property Inspector. The Property Inspector displays all properties for the selected annotation object. However, you should not change some of these properties because doing so can affect the proper functioning of the annotation object. See the following sections for descriptions of the properties you can change on the respective objects.

- Annotation Rectangle Properties
- Annotation Ellipse Properties

Textbox Annotations

A textbox is a rectangle that can contain multiline text. You can attach the textbox to any point in the figure.

Insert a textbox by clicking the textbox button in the figure toolbar  , then click where you want to place the text string. The default behavior for textboxes is for them to resize to accommodate the amount of text you enter into them. You can also resize the textbox after typing or click and drag the box to a certain size when you create it (when you do this, the textbox stays that size no matter how much text you place within it).

You can also select **TextBox** from the **Insert** menu.

Selecting Textbox Objects

The selection behavior of the textbox object differs from other annotation objects.

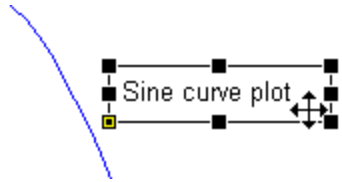
- To move a textbox, click the text once to select it.
- To edit the a textbox, double-click within the box.
- To display the Property Editor with textbox properties, right-click to display the context menu and select **Properties**.

Pinning the Textbox

You can attach the textbox to a particular point in the figure by pinning it to that point. There are three ways to pin the textbox:

- Right-click within the textbox to display its context menu and select **Pin to Axes**.
- Select the pin button in the figure toolbar and click a handle of the textbox (See “Pinning — Attaching to a Point in the Graph” on page 3-23).
- Select **Pin Object** from the **Edit** menu.

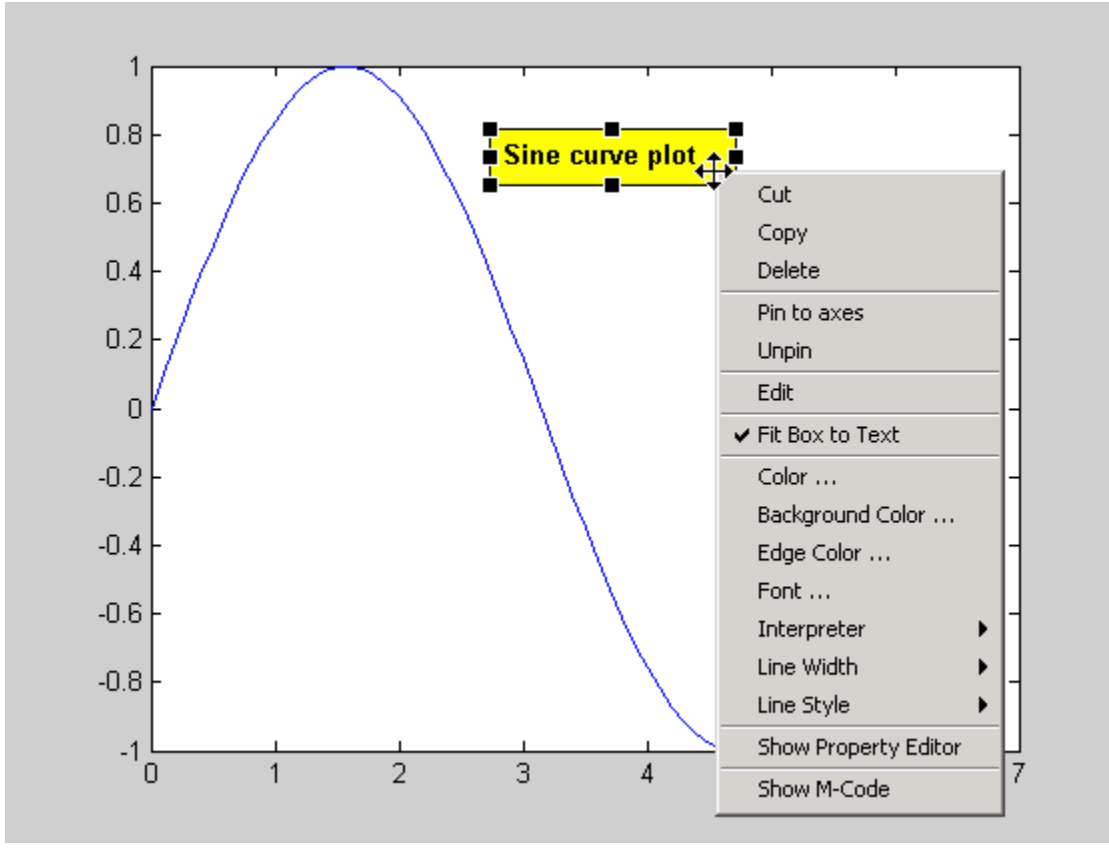
By default, pinning attaches the lower left corner of the textbox to its location in the axes data space. You can move the point of attachment by clicking on the corner and dragging the anchor to another point.



Note that you cannot drag the textbox when it is pinned.

Modifying the Textbox from the Context Menu

Right-clicking in a textbox displays its context menu, which enables you to perform a number of operations on the textbox. In the following picture, the textbox **Background Color** has been set to yellow and its **Font** has been set to bold using the context menu. The textbox has its default resizing behavior, as indicated by the checked item **Fit Box to Text**:



When you create a textbox without dragging it to have a specific size, **Fit Box to Text** is enabled, and the box will grow or shrink as you type or edit its text. If you drag when creating a textbox, or change its size by dragging any of its handles in plot edit mode, **Fit Box to Text** is disabled, but you can re-enable it using the context menu.

The menu contains the following options:

- **Cut**, **Copy**, **Delete** — Cut to clipboard, copy to clipboard, or delete the textbox.

- **Pin to axes** — Pin the textbox to the current location (you can move the point of attachment by clicking and dragging the textbox while in plot edit mode).
- **Unpin** — Detach the textbox from the attachment point.
- **Edit** — Enable edit mode to change the text. You can also double-click the textbox with the left mouse button to enable edit mode.
- **Fit Box to Text** — Resize textbox to accommodate text extents (or not)
- **Text Color** — Color of the text characters
- **Background Color** — Fill color of the rectangle enclosing the text
- **Edge Color** — Color of the textbox edge line (you must set **Line Style** to a value other than **none** to display edges)
- **Font** — Type of font used for the text
- **Interpreter** — Interpret characters as T_EX (`latex` or `tex`) or as literal characters (`none`).
- **Line Width** — Width of the textbox edge line
- **Line Style** — Style of line used for the textbox edge
- **Properties** — Display the Property Editor with textbox properties.
- **Show M-code** — Create M-code that recreates the graph.

Setting Textbox Properties

You can use the Property Editor to set textbox properties by selecting **Show Property Editor** from the textbox context menu. It displays the same properties that are described above in the context menu section.

You can click the **More Properties** button on the Property Editor to display the Property Inspector. The Property Inspector displays all textbox properties. However, you should not change some of these properties because doing so can affect the proper functioning of the textbox.

See Textbox Properties in the reference documentation for a description of the properties you can change.

Annotation Lines and Arrows

You can add lines and three types of arrows to a graph and attach them to any point in the figure. The three types of arrows include

- Single-headed arrow
- Arrow with attached text box
- Double-headed arrow

Insert a line or arrow by clicking the appropriate button in the figure toolbar  , then click down, drag the line or arrow to the desired point, and release the mouse. The arrowhead appears at the terminal end.

With the line or arrow selected, right-click to display the context menu, which provides access to a number of options.

Inserting a Text Arrow

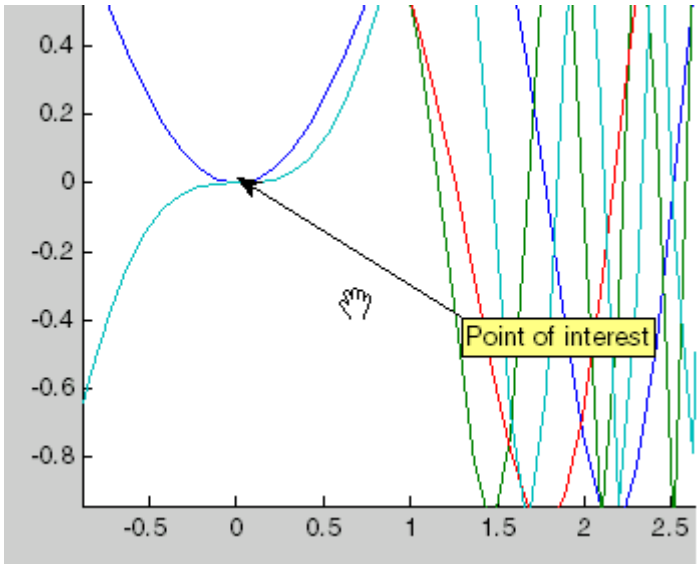
A text arrow combines a textbox with an arrow. It is useful for labeling points on a graph. Add a text arrow to a graph by selecting the arrow button that has a T above the arrow. Insert the text arrow and type text in the box.

Pinning the Arrowhead End

You can attach the arrowhead end to the point of interest on the graph while letting the text box automatically reposition itself as you zoom or pan the graph.

There are three ways to pin annotations:

- Right-click the object to display its context menu and select **Pin**.
- Select the **pin** button in the plot edit toolbar (See “Pinning — Attaching to a Point in the Graph” on page 3-23).
- Select **Pin Object** from the **Edit** menu.



You can pin the arrowhead and then pan the graph to get the desired view.

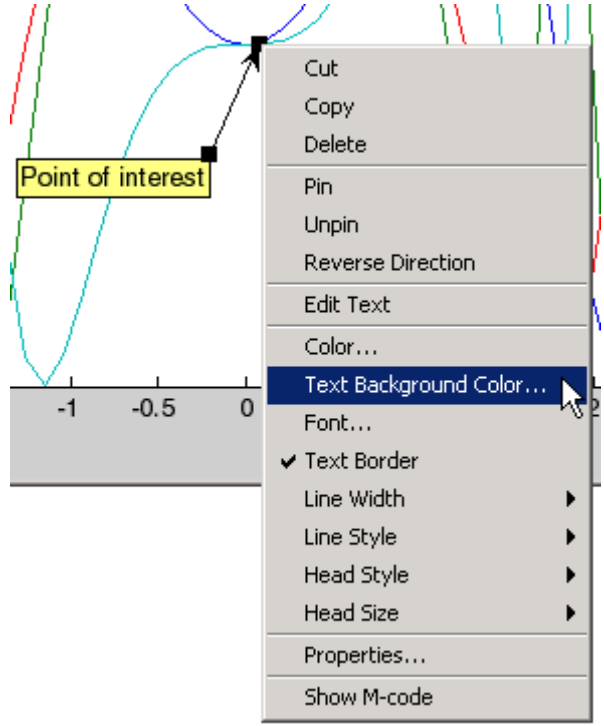
Modifying the Text Arrow from the Context Menu

Right-clicking on a text arrow displays its context menu, which enables you to perform a number of operations on the text arrow. The context menus for lines, arrows, and double arrows contain similar items:

- **Cut**, **Copy**, **Delete** — Cut to clipboard, copy to clipboard, or delete the textbox.
- **Pin to axes** — Pin the textbox to the current location (you can move the point of attachment by clicking and dragging the point while in plot edit mode).
- **Unpin** — Detach the textbox from the attachment point.
- **Reverse Direction** — Swap the arrow head and the textbox or move the arrowhead to the other end of a plain arrow.
- **Edit Text** — Enable edit mode to change the text characters.
- **Color** — Color of the text characters, textbox edge, and arrow
- **Text Background Color** — Color of the rectangle enclosing the text
- **Font** — Type of font used for the text

- **Line Width** — Width of the textbox edge line
- **Line Style** — Style of line used for the textbox edge
- **Head Style** — Type of arrowhead to use
- **Head Size** — Size of the arrowhead in points
- **Properties** — Display the Property Editor with textbox properties.
- **Show M-code** — Create M-code that recreates the graph.

For example, the following illustration shows the text border enabled and the text background color set to yellow.



Setting Line and Arrow Properties

You can use the Property Editor to set line and arrow properties by selecting **Properties** from the context menu. The Property Editor displays the same properties that are described above in the context menu section.

You can click the **More Properties** button on the Property Editor to display the Property Inspector. The Property Inspector displays all properties for the selected annotation object. However, you should not change some of these properties, because doing so can affect the proper functioning of the annotation. See the following sections in the reference documentation for descriptions of the properties you can change on the respective objects.

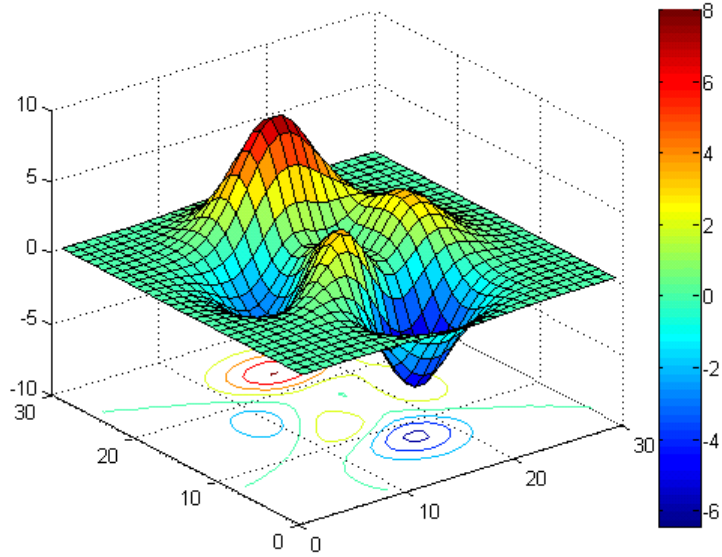
- Annotation Line Properties
- Annotation Arrow Properties
- Annotation Textarrow Properties
- Annotation Doublearrow Properties

Adding a Colorbar to a Graph

Colorbars display the current colormap and indicate the mapping from data values to colors. The following picture shows a surface plot with 2-D contour lines below. Note how the colorbar at the right indicates how the *z*-axis data values correspond to colors in both the surface and contour graphs.

Add a colorbar by clicking the colorbar button in the toolbar  or by selecting **Colorbar** from the **Insert** menu. When plot editing is enabled, you can select and then move and resize the colorbar. The following commands will also create this plot:

```
surfc(peaks(30))
colorbar
```



You can also use the `colorbar` function to add a colorbar to a 2-D graph.

Positioning Options for Colorbars

There are a variety of ways to reposition a colorbar in the figure.

- Enable plot edit mode, then select and drag the colorbar to the desired location.
- Right-click over the colorbar to display its context menu. Mouse over **Locations** and select one of the predefined locations for the colorbar.
- Right-click over the colorbar to display its context menu and select **Properties**. This displays the Property Editor, which provides a graphical positioning device for the colorbar.

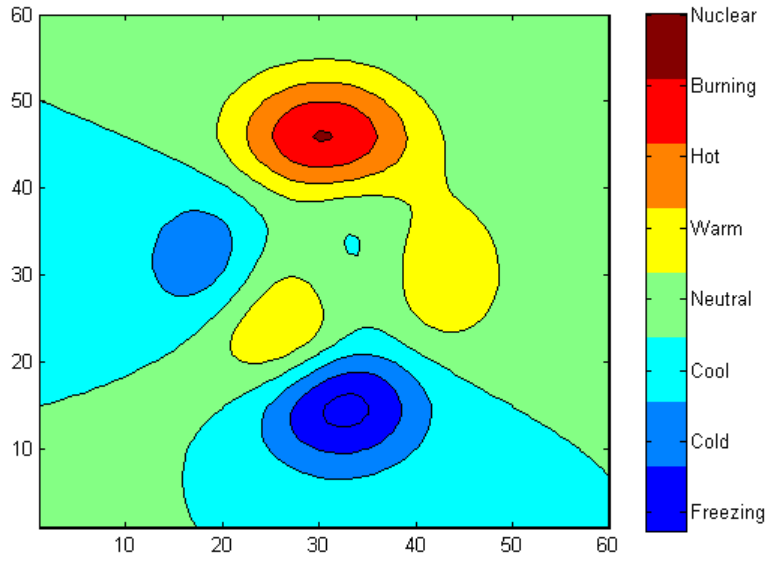
Labeling Colorbar Ticks

The default colorbar labels ticks with numeric values, which are placed at intervals specified by the colorbar's `YTick` parameter (for vertical colorbars) or its `XTick` parameter (for horizontal colorbars), within upper and lower limits specified by `CData`. You can override these limits by using the `caxis` function.

You can specify strings in place of the numeric labels on colorbars. This is useful for display of data on nominal or ordinal scales and for when you want to interpret the meaning of entries in the colormap for the viewer. To substitute strings for numbers along a colorbar, you define a label for each tick location. You do this by specifying a cell array of strings for `YTicklabel` (vertical colorbars) or `XTickLabel` (horizontal colorbars), as the examples below show.

Example 1: Default Vertical Colorbar with YTickLabels

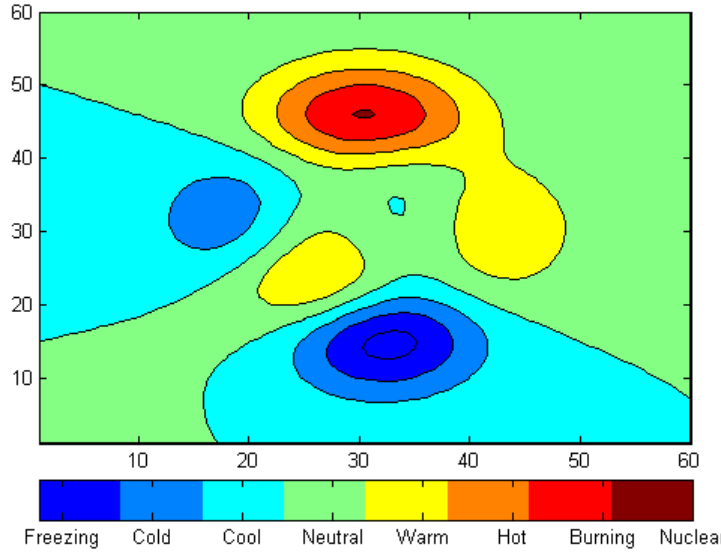
```
contourf(peaks(60));
colormap(jet(8));
hcb = colorbar('YTickLabel',...
{'Freezing','Cold','Cool','Neutral',...
'Warm','Hot','Burning','Nuclear'});
set(hcb,'YTickMode','manual')
```



Example 2: Horizontal Colorbar with XTickLabels

```
figure
contourf(peaks(60));
```

```
colormap(jet(8));
hcb = colorbar('Location','SouthOutside','XTickLabel',...
{'Freezing','Cold','Cool','Neutral',...
'Warm','Hot','Burning','Nuclear'});
set(hcb,'XTickMode','manual')
```



In these examples, the number of colors and the number of labels were set to be the same (8). This is typical for nominal (categorical) data, but not necessary if you do not object to having a range of colors associated with each label.

Note that if ticks change, for instance if `YTick` (`XTick`) values change or the plot is rescaled while `YTickMode` (`XTickMode`) or `YTickLabelMode` (`XTickLabelMode`) is `auto`, too few or too many colorbar labels may be displayed, and can sit next to colors they do not represent. When there are fewer labels than ticks, the labels will cycle, with the lowest one following the highest one, etc., to give each tick a label. This is probably not what you want, so you need to reset `YTick` (`XTick`) values in such cases. Finding the correct values can take some experimentation. The `set` functions in the above examples prevent MATLAB from changing the number of ticks when you resize figures.

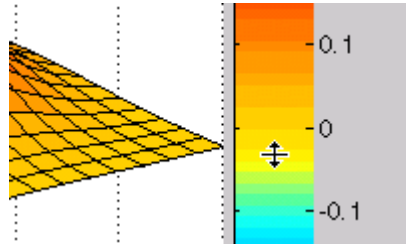
Selecting a Different Colormap

If you change the figure colormap, the colorbar updates automatically. Use one of the following methods to change the colormap.

- Right-click over the colorbar to display its context menu. Mouse over **Standard Colormaps** and select from the displayed list.
- Right-click over the colorbar to display its context menu and select **Properties**. Click the figure background to load the figure properties into the Property Editor. Select the colormap from the pull-down list.
- Use the `colormap` function.

Modifying the Colormap

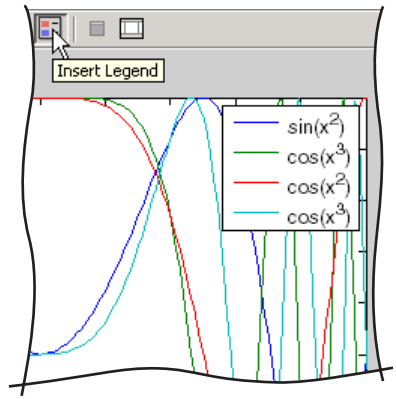
You can use a colorbar to modify the current colormap. To do this, select **Interactive Colormap Shift** from the right-click context menu. In this mode, you can left-click down on any color in the colorbar and, by dragging the mouse, shift the color-to-data mapping.



To perform more sophisticated operations on the colormap, launch the colormap editor by selecting **Launch Colormap Editor** from the colorbar's context menu. See the `colormapeditor` reference page for more information.

Adding a Legend to a Graph

Legends provide a key to the various data plotted on a graph. The following picture shows the legend for a graph of four functions of a variable plotted with lines of different colors. A graph can have only one legend, which applies to and will symbolize all data series contained by an axes, according to their form (e.g., lines, bars, pies, etc.). Note how you can assign an appropriate string to each line in the legend.



Add a legend by clicking the legend button in the toolbar  or by selecting **Legend** from the **Insert** menu. When plot editing is enabled, you can select and then move and resize the legend.

You can also use the `legend` function to add a legend to a graph, which gives additional controls over appearance. You must use this command in order to display a legend with more than 20 entries, as the legend toolbar button is limited to displaying legends for 20 elements (columns) of a lineseries only.

Specifying the Text

By default, the legend labels each plotted object (line, surface, etc.) with the strings `data1`, `data2`, etc. You can change this text by double-clicking on the text to enable edit mode. In edit mode, you can retype the text string.

You can use T_EX characters in the text strings to produce symbols. You can disable interpretation of characters as T_EX sequences by selecting **none** from the **Interpreter** submenu of the legend's right-click context menu.

See the Table of TEX symbols in the Text Properties reference documentation for more information.

Positioning the Legend

There are a number of ways to position the legend.

- Enable plot edit mode, select the legend, and drag it to the desired location.
- Right-click the legend to display its context menu, mouse over **Location,** and select one of the predefined locations from the submenu.
- Right-click the legend to display its context menu and select **Properties** to display the Property Editor, which provides a graphical device for positioning the legend.

You can also select a vertical or horizontal orientation for the legend. Use the **Orientation** item in the context menu to make this selection.

Changing the Appearance of the Legend

You can specify the following legend characteristics from the context menu:

- **Color** — Set the background color of the legend. In addition, you can specify the `Color` property as `'none'` to make the legend background be transparent.
- **EdgeColor** — Set the color of the line enclosing the legend box.
- **TextColor** — Set the color of the legend text.

  You can use a colorspec or an RGB color triplet to set the above three properties.
- **LineWidth** — Set the width of the edge line.
- **Font** — Set the font, font style, and font size of the text used in the legend.
- **Interpreter** — Set the text `Interpreter` property to use T_EX or plain text.
- **Orientation** — Orient the legend entries side by side (horizontal) or on top of each other.
- **Properties** — Display the Property Editor with legend properties.
- **Show M-code** — Generates M-code for recreating the legend.

Controlling the Appearance of Grouped Objects on a Legend

When you create a legend for groups of graphic objects such as lineseries, barseries, or stemseries, the default legend will show an individual legend entry for each of the graphics objects. Sometimes you might want only certain objects to appear in a legend, to show one legend entry for the entire series,

or to show the individual children of a series (however, not all series have children; you can use the Handle Graphics Property Browser to determine this). You can control how groups appear in the legend by setting values for their `Annotation` property via M-code. For information on how to customize legends in this manner, see “Controlling Legends” on page 8-99.

You can view the values of an object's `Annotation` property in the Property Inspector, but you can not set them there; you need to use M-code.

Pinning — Attaching to a Point in the Graph

Pinning is the attachment of an object to a particular point in the figure. Pinning enables you to pan or resize the figure while keeping annotations associated with the same point. For example, the following picture shows regions in two different graphs associated by pinning both ends of a double arrow.

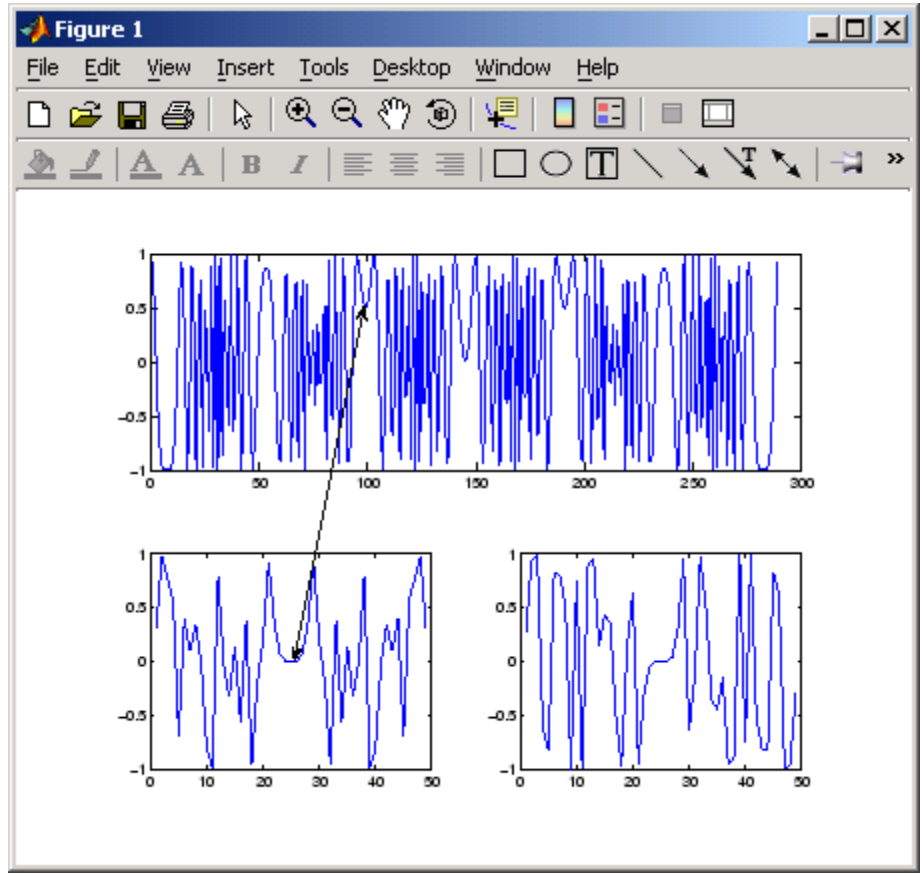
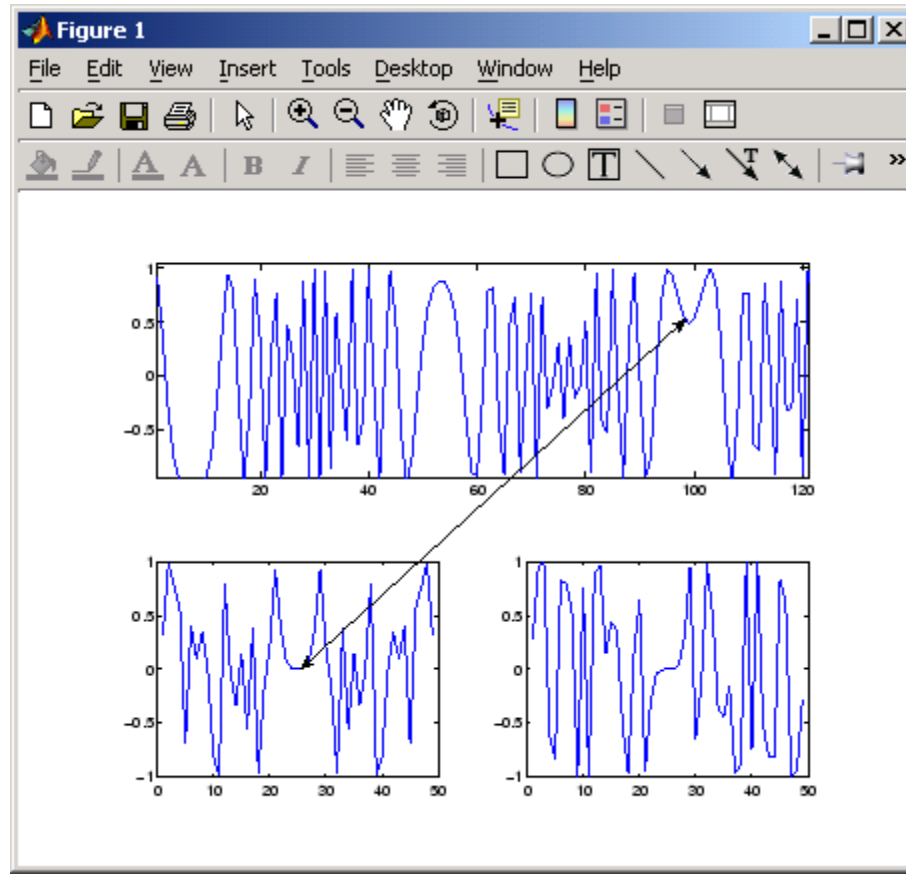


Figure 1
File Edit View Insert Tools Desktop Window Help

If you perform a horizontal zoom on the top axes (select **Horizontal Zoom** from the **Options** submenu of the **Tools** menu) and then pan the graph to show the first 120 seconds of the data, the double arrow continues to point to the same locations on the graph.



Pinning Objects

To pin an object, first enable pinning mode by clicking the **pin object** button  in the plot edit toolbar or selecting **Pin Object** from the **Edit** menu. Then click the point you want to pin.

To unpin an object, right-click to display the context menu and select **Unpin**.

You can pin annotation lines, arrows, rectangles, ellipses, and text boxes.

When this mode is enabled, axes, rectangle, arrows, and lines automatically align their upper left corners to the grid. As you move or resize one of these objects, the size or position snaps to the next grid location.

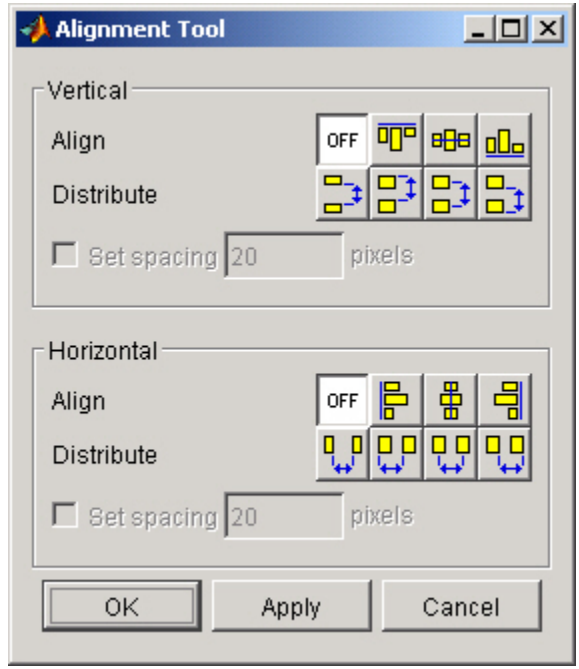
Alignment Tool — Aligning and Distributing Objects

In this section...

Alignment Tool Functionality

The Alignment Tool enables you to position objects with respect to each other and to adjust the spacing between selected objects. The specified align/distribute operations apply to all components that are selected when you click the **Apply** or **OK** buttons.

Display the Alignment Tool by clicking the Align/Distribute button  or by selecting **Align Distribute Tool** from the **Tools** menu.



The Alignment Tool provides two types of positioning operations:

- **Align** — Align all selected objects to a single reference line.
- **Distribute** — Space all selected objects uniformly with respect to each other.

You can align and distribute objects in the vertical and horizontal directions. The following sections provide more information.

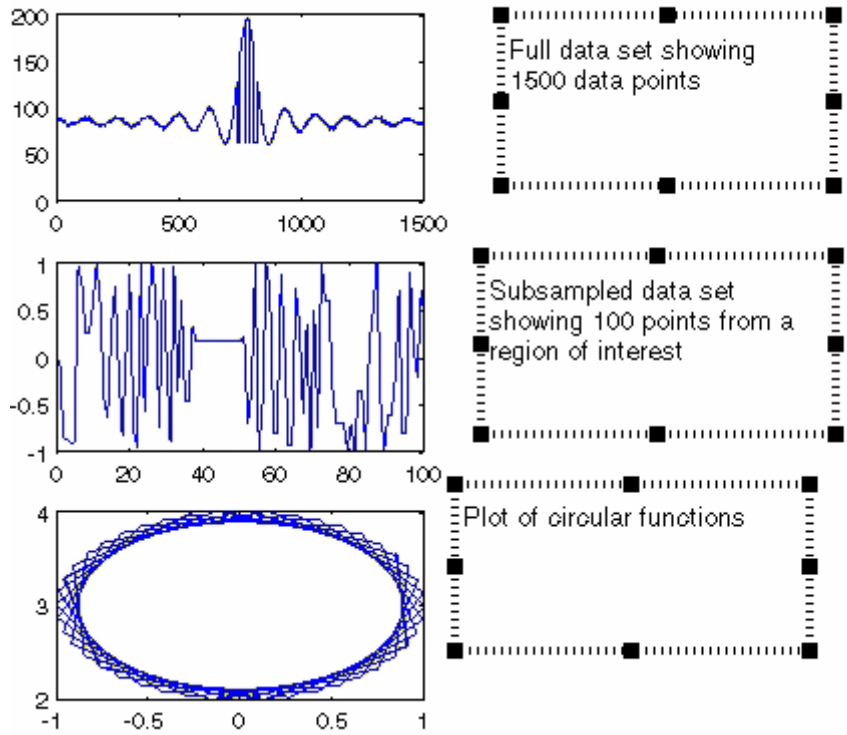
- "Example — Vertical Distribute, Horizontal Align" on page 3-28
- "Align/Distribute Menu Options" on page 3-31
- "Snap to Grid — Aligning Objects on a Grid" on page 3-33

Example — Vertical Distribute, Horizontal Align

This example illustrates how to align three textboxes with three corresponding axes. In this example, the text boxes were just plunked down close to the

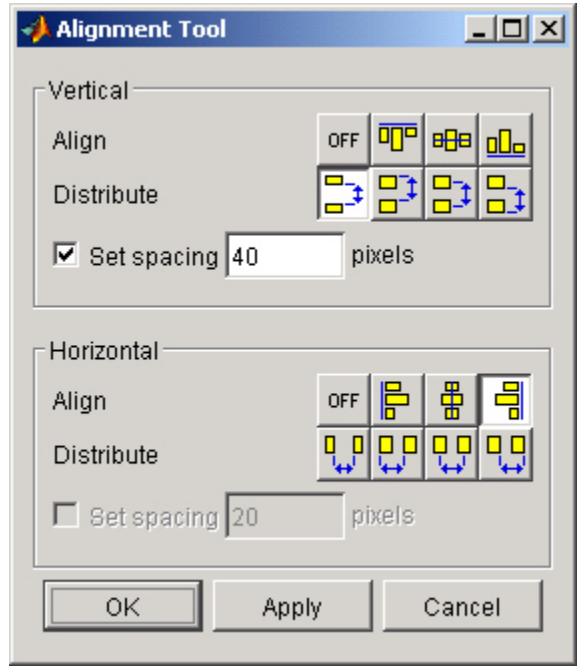
desired position and then right aligned and distributed to have 40 pixels between them.

The following picture shows the initial layout.

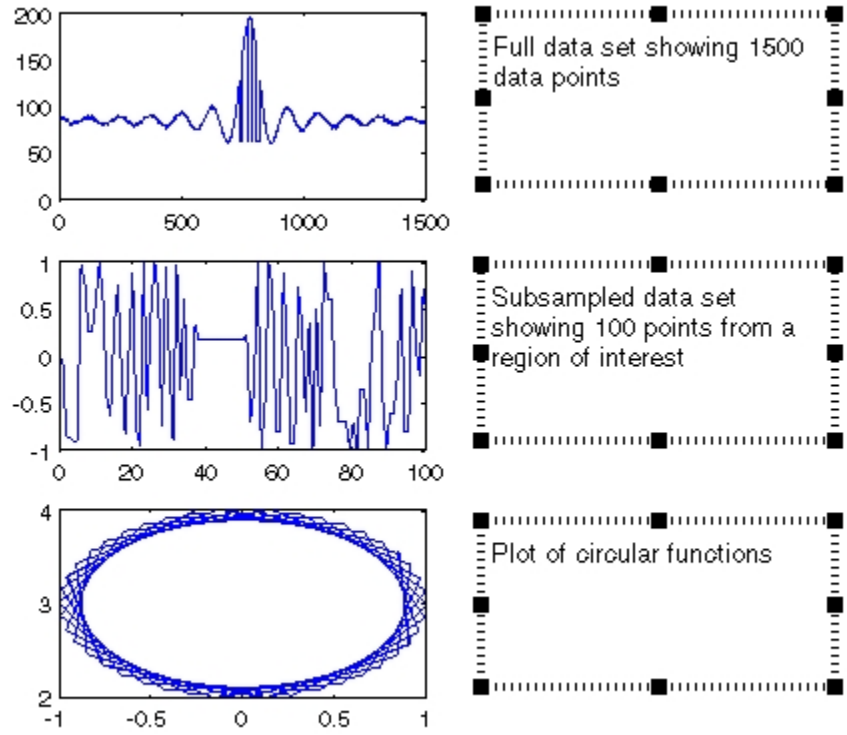


Use **Shift**+click to select all three textboxes and then configure the Alignment Tool as shown in the following picture.

- Set vertical distribution to 40 pixels.
- Set horizontal alignment to right-aligned.
- Click **Apply.**

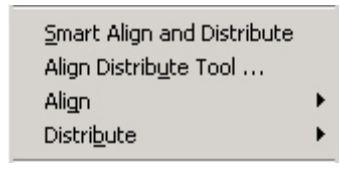

Alignment Tool
Vertical
Align
OFF
Distribute
Set spacing
40
pixels
Horizontal
Align
OFF
Distribute
Set spacing
20
pixels
OK
Apply
Cancel

The following picture shows the result.



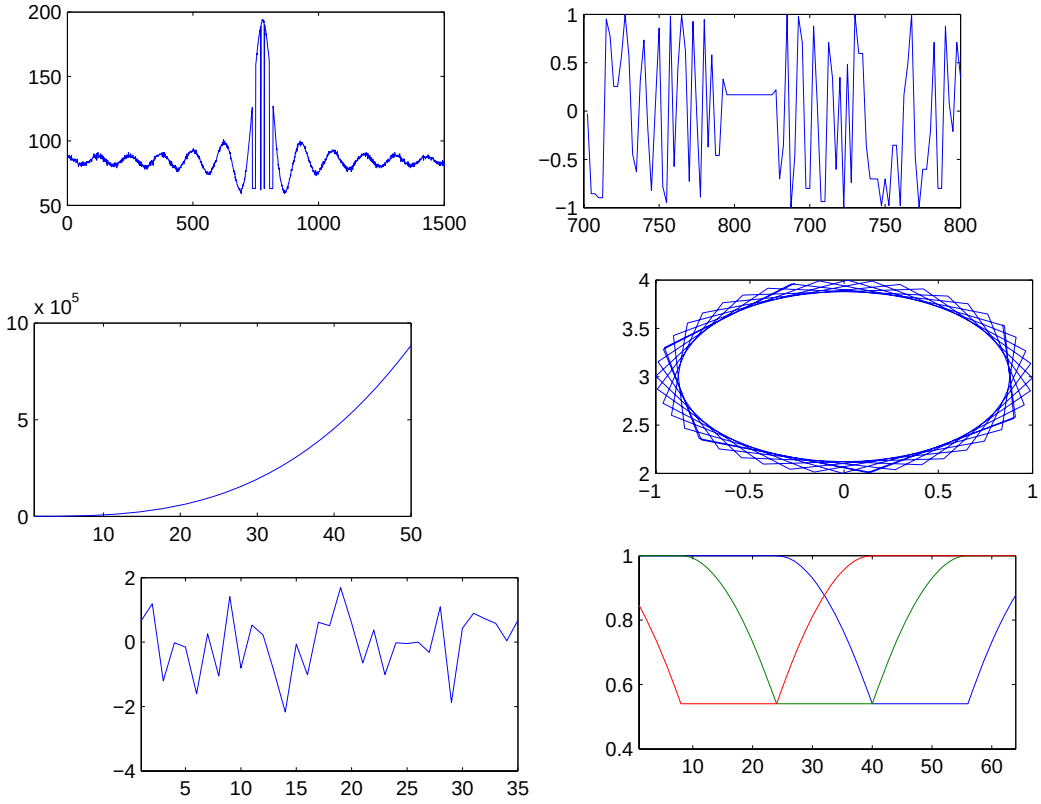
Align/Distribute Menu Options

The **Tools** menu contains the alignment and distribution options that are available via the Alignment Tool.



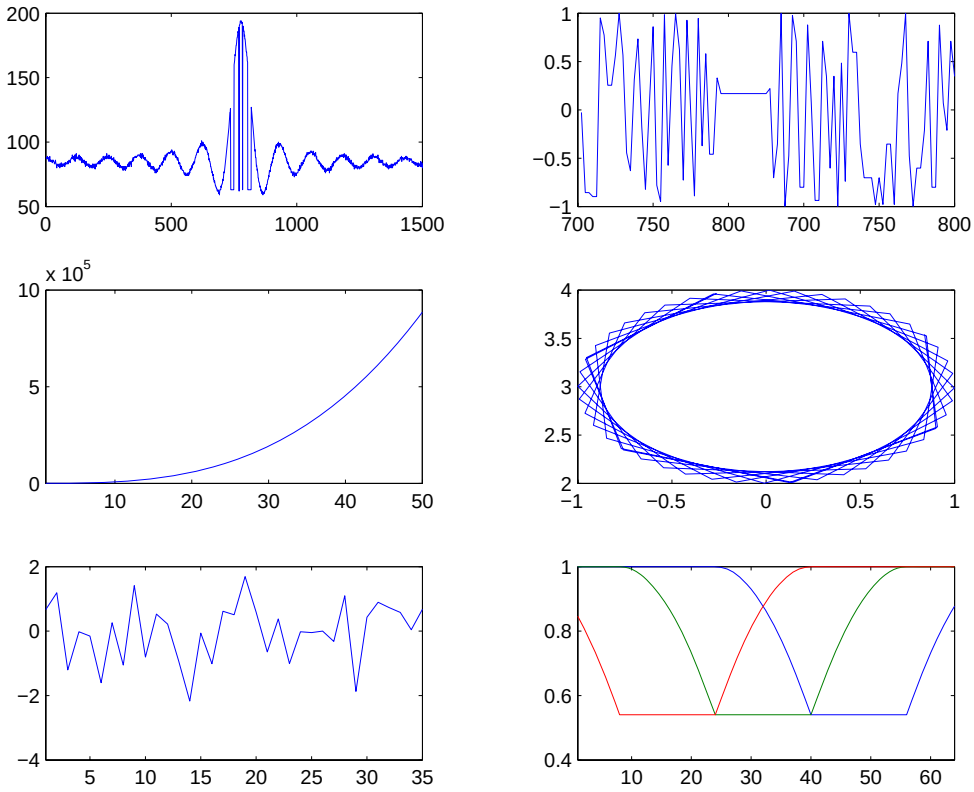
The **Smart Align and Distribute** option aligns objects into rows and columns with equal spacing between each object. It is useful when you have a number of objects to align that can be positioned in an m-by-n grid.

For example, the following figure contains six axes that have been placed approximately into two columns in the figure.



To align all axes in a grid, select each axes (**Shift**+click each one), then select **Smart Align and Distribute** from the **Tools** menu.

The resulting alignment and distribution of the axes are shown below.



Snap to Grid — Aligning Objects on a Grid

Figures have a layout grid that can aid the hand layout of objects displayed in the figure. You can also enable a snap-to-grid feature that forces objects to align with the grid increments when moved.

To display the grid on the figure background, select **View Layout Grid** from the **Tools** menu.

To force objects to align with the grid, select **Snap To Layout Grid** from the **Tools** menu.

To move objects in the figure, enable Plot Edit mode by selecting **Edit Plot** from the **Tools** menu. Click to select an object and then drag it to the desired location.

The following picture illustrates a figure with four subplots. You can select any of the four axes and move them. Note that all axis labels and the title move with the axes. Annotation objects move independently of the plot axes.

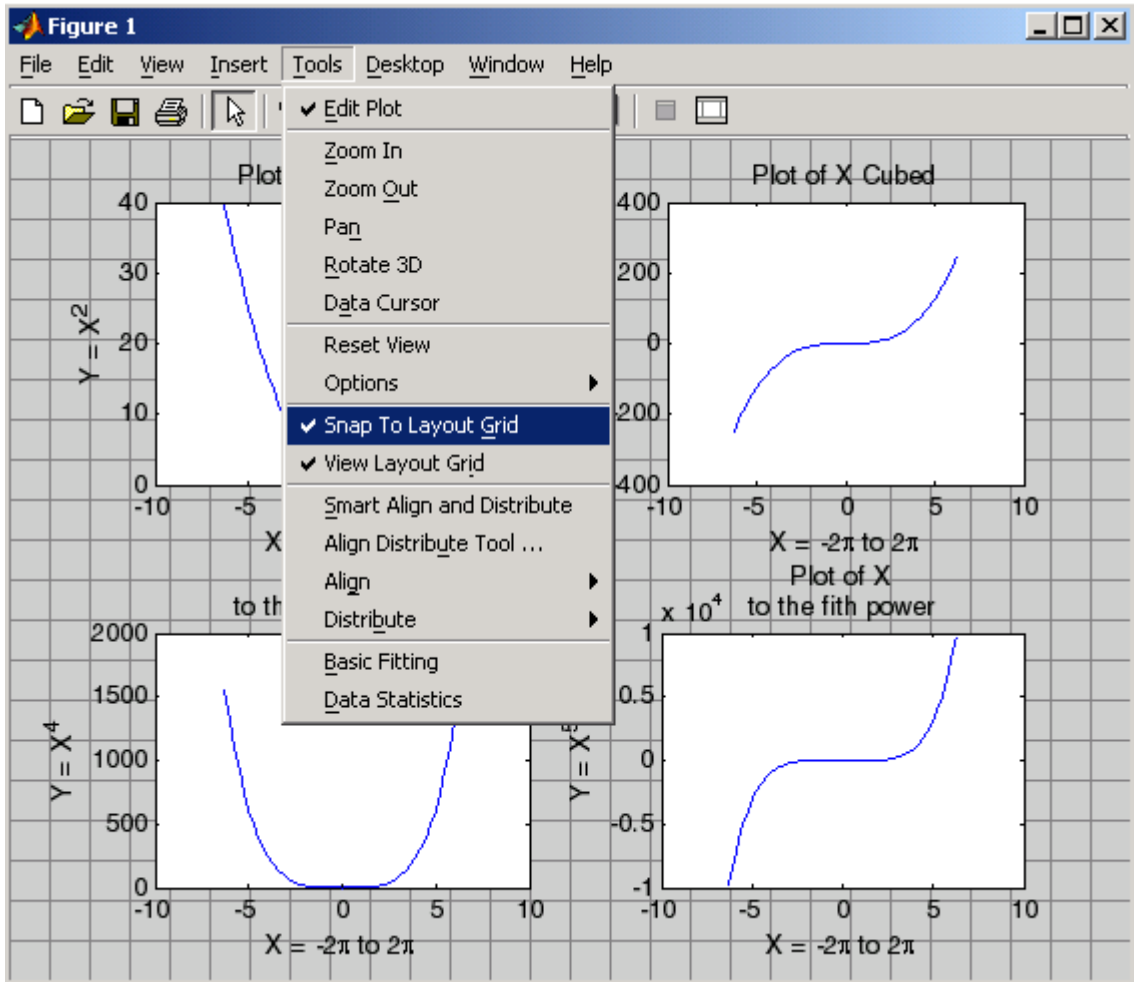


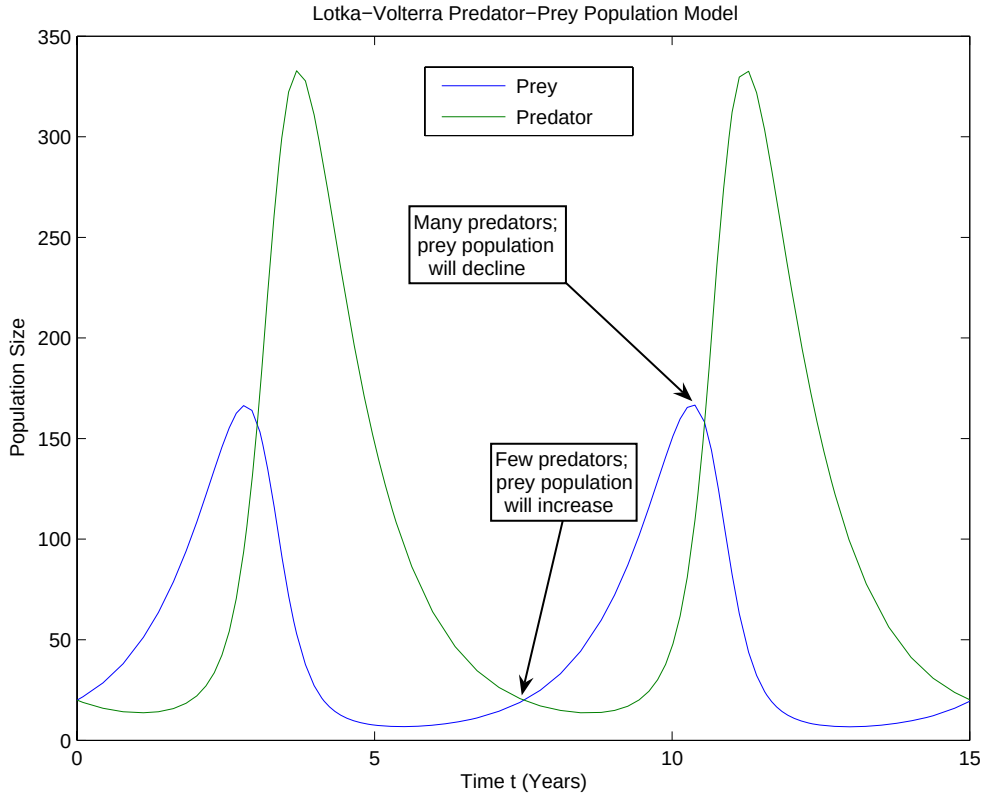
Figure 1
File Edit View Insert Tools Desktop Window Help
Edit Plot
Zoom In
Zoom Out
Pan
Rotate 3D
Data Cursor
Reset View
Options
Snap To Layout Grid
View Layout Grid
Smart Align and Distribute
Align Distribute Tool ...
Align
Distribute
Basic Fitting
Data Statistics
Plot of X Cubed
X = -2π to 2π
Plot of X
to the fith power
x 10^4
Y = X^2
Y = X^4
X = -2π to 2π
X = -2π to 2π

Adding Titles to Graphs

In this section...

What Is a Title?

In MATLAB, a title is a text string at the top of an axes. It appears in the figure border, not within the axes it describes. Titles typically define the subject of the graph. The following figure shows a title, centered at its top.



Note While you can use text annotations to create a title for your graph, it is not recommended. Titles are anchored to the top of the axes they describe; text annotations are not. If you move or resize your axes, the title remains at the top. Additionally, if you cut a title and then paste it back into a figure, the title is no longer anchored to the axes.

You can add a title to a graph in several ways, described in the following sections.

Using the Title Option on the Insert Menu

To add a title to a graph using the **Insert** menu,

1. Click the **Insert** menu in the figure menu bar and choose **Title**. MATLAB opens a text entry box at the top of the axes.

   Note When you select the **Title** option, MATLAB enables plot editing mode automatically.

2. Enter the text of the label.

3. When you are finished entering text, click anywhere in the figure background to close the text entry box around the title. If you click on another object in the figure, such as an axes or line, you close the title text entry box and also automatically select the object you clicked.

To change the font used in the title to bold, you must edit the title. You can edit the title as you would any other text object in a graph.

Using the Property Editor to Add a Title

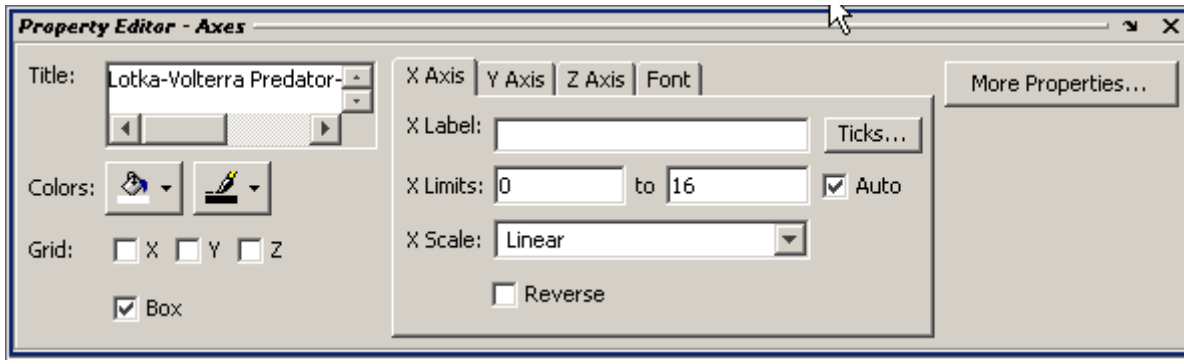
To add a title to a graph using the Property Editor,

1. Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.

2. Double-click an empty region of the axes in the graph. This starts the Property Editor. You can also start the Property Editor by right-clicking on the axes and selecting **Show Property Editor** from the context menu or by selecting **Property Editor** from the **View** menu.

   The Property Editor displays a property panel specific to axes objects. Titles are a property of axes objects.

3. Type the text of your title in the **Title** text entry box.



You can change the font, font style, position, and many other aspects of the title format.

- To move the title, select the text and drag it to the desired position.
- To edit the text, double-click the title and type new characters.
- To change the font and other text properties, select the title and right-click to display the context menu.

Using the title Function

To add a title to a graph at the MATLAB command prompt or from an M-file, use the `title` function.

For example, the following code adds a title to the current axes and sets the value of the `FontWeight` property to bold.

```
title('Lotka-Volterra Predator-Prey Population Model',...
  'FontWeight','bold')
```

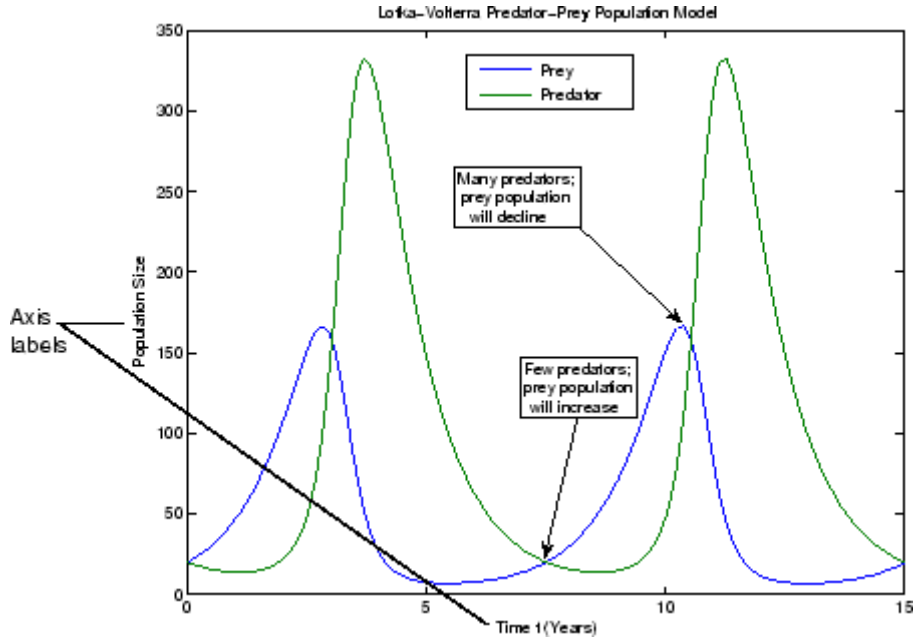
To edit a title from the MATLAB command prompt or from an M-file, use the `set` function.

Adding Axis Labels to Graphs

In this section...

What Are Axis Labels?

In MATLAB, an axis label is a text string aligned with the *x*-, y-, or *z*-axis in a graph. Axis labels can help explain the meaning of the units that each axis represents.



Note While you can use free-form text annotations to create axes labels, it is not recommended. Axis labels are anchored to the axes they describe; text annotations are not. If you move or resize your axes, the labels automatically move with the axes. Additionally, if you cut a label and then paste it back into a figure, the label is no longer anchored to the axes.

You can add axis labels to a graph in several ways, described in the following sections.

Using the Label Options on the Insert Menu

1. Click the **Insert** menu and choose the label option that corresponds to the axis you want to label: **X Label**, **Y Label**, or **Z Label**. MATLAB opens a text entry box along the axis or around an existing axis label.

   Note MATLAB opens up a horizontal text editing box for the *y*- and *z*-axis labels and automatically rotates the label into alignment with the axis when you finish entering text.

2. Enter the text of the label, or edit the text of an existing label.

3. Click anywhere else in the figure background to close the text entry box around the label. If you click on another object in the figure, such as an axes or line, you close the label text entry box but also automatically select the object you clicked.

Note After you use the **Insert** menu to add an axis label, plot edit mode is enabled in the figure, if it was not already enabled.

Using the Property Editor to Add Axis Labels

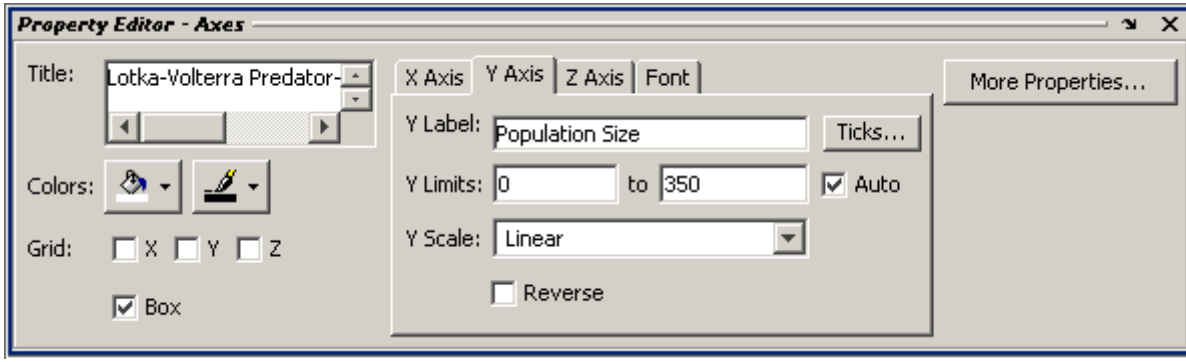
To add labels to a graph using the Property Editor,

1. Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.

2 Start the Property Editor by double-clicking on the axes in the graph. You can also start the Property Editor by right-clicking on the axes and selecting **Properties** from the context menu or by selecting **Property Editor** from the **View** menu.

   The Property Editor displays the set of property panels specific to axes objects.

3 Select the **X Axis**, **Y Axis**, or **Z Axis** tab, depending on which axis label you want to add. Enter the label text in the text entry box.



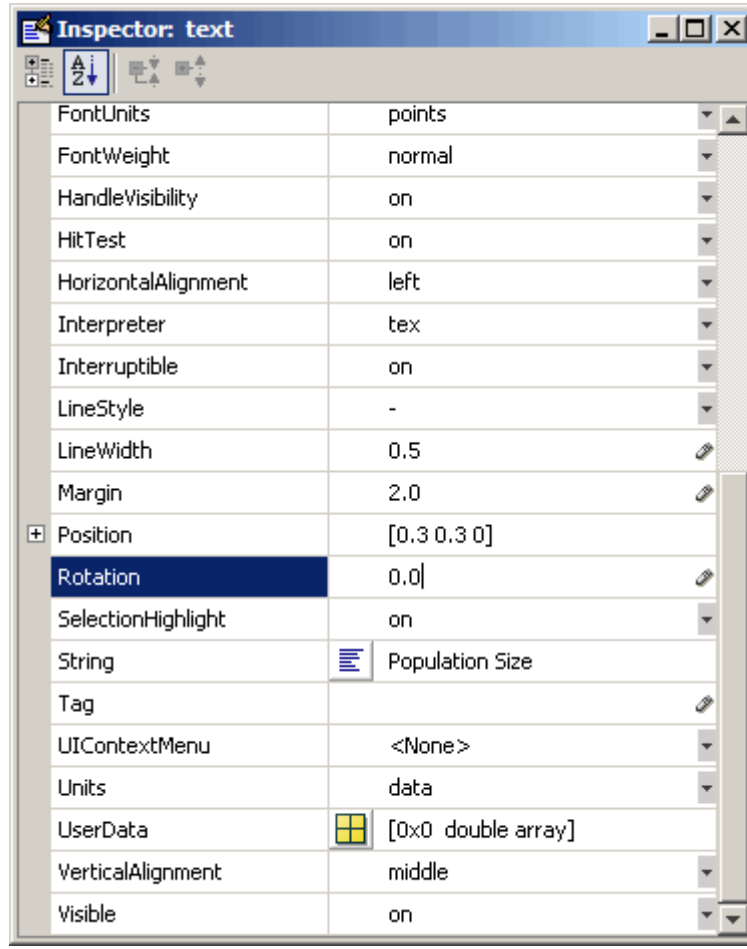
Rotating Axis Labels

You can rotate axis labels using the Property Editor:

1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.

2 Display the Property Editor by selecting (left-clicking) the axis label you want to rotate. Right-click over the selected text, then choose **Properties** from the context menu.

3 Click the **More Properties** button to display the Property Inspector.



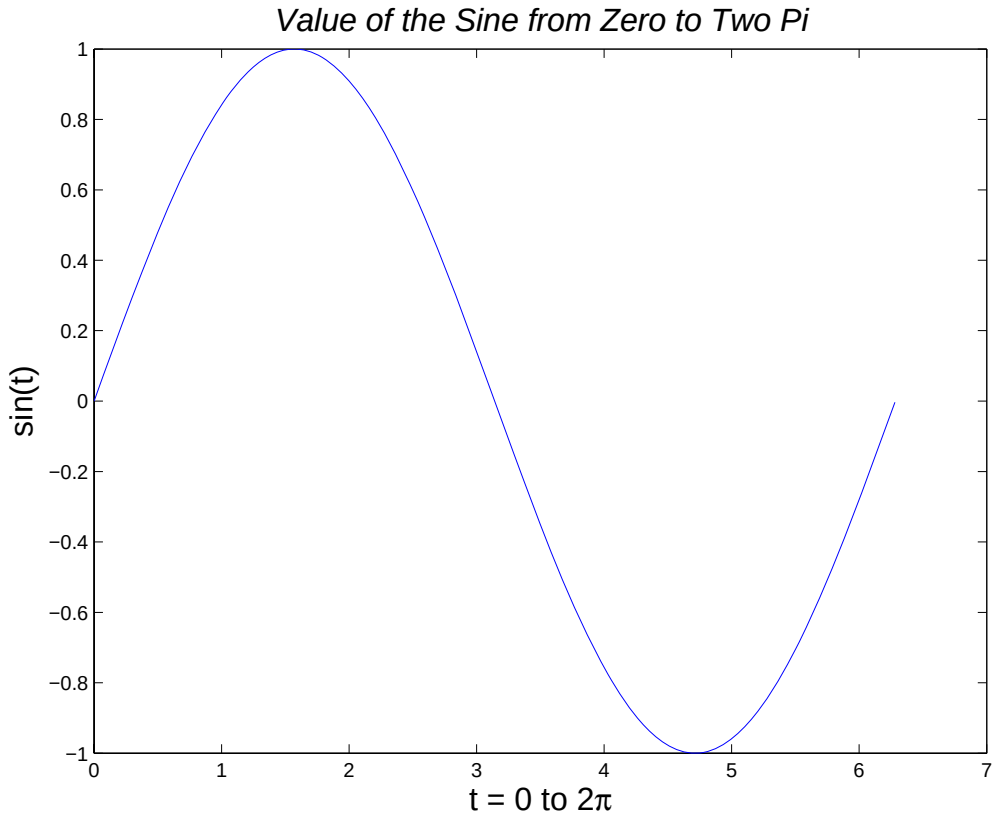
4 Select the **Rotation** property text field. A value of 0 degrees orients the label in the horizontal position.

5 With the left mouse button down on the selected label, drag the text to the desired location and release.

Using Axis-Label Commands

You can add *x*-, *y*-, and *z*-axis labels using the `xlabel`, `ylabel`, and `zlabel` functions. For example, these statements label the axes and add a title.

```
xlabel('t = 0 to 2\pi','FontSize',16)
ylabel('sin(t)','FontSize',16)
title('\it{Value of the Sine from Zero to Two Pi}','FontSize',16)
```



The labeling commands automatically position the text string appropriately. MATLAB interprets the characters immediately following the backslash (`\`) as TeX commands. These commands draw symbols such as Greek letters and arrows.

See the text `String` property for a list of TeX character sequences. See also the `texlabel` function for converting MATLAB expressions to TeX symbols.

Rotating Axis Labels Using Commands

Axis labels are text objects that you can rotate by specifying a value for the object's `Rotation` property. The handles of the *x*-, *y*-, and *z*-axis labels are stored in the axes `XLabel`, `YLabel`, and `ZLabel` properties respectively.

Therefore, to rotate the *y*-axis label so that the text is horizontal:

1. Get the handle of the text object using the axes `YLabel` property.
2. Set the `Rotation` property to 0.0 degrees.

For example, this statement rotates the text of the *y*-axis label on the current axes:

```
set(get(gca,'YLabel'),'Rotation',0.0)
```

Repositioning Axis Labels

You can reposition an axis label by dragging the text.

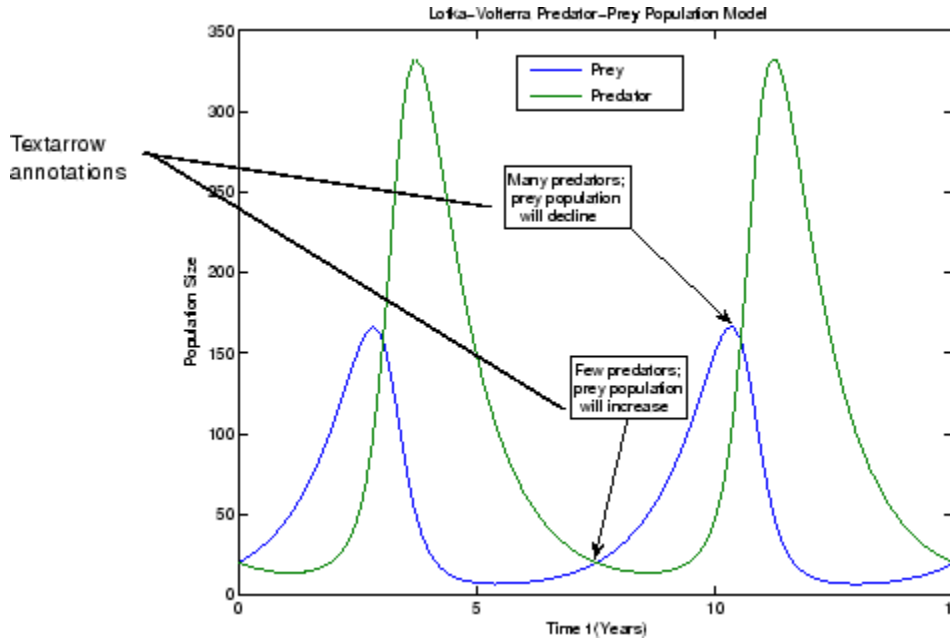
1. Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.
2. Select the text of the label you want to reposition (handles appear around the text object).
3. With the left mouse button down on the selected label, drag the text to the desired location and release.

Adding Text Annotations to Graphs

In this section...

What Are Text Annotations?

Text annotations are boxes containing text strings that you compose. The box can have a border and a background, or be invisible. The text can be in any installed text font, and can include TeX or LaTeX markup. You can add free-form text annotations anywhere in a MATLAB figure to help explain your data or bring attention to specific points in your data sets.



If you enable plot editing mode, you can create text annotations by clicking in an area of the graph or the figure background and entering text. You can also add text annotations from the command line, using the `text` or `gtext` function.

Using plot editing mode or `gtext` makes it easy to place a text annotation anywhere in a graph. Use the `text` function when you want to position a text annotation at a specific point in a data set.

Note Text annotations created using the `text` or `gtext` function are anchored to the axes. Text annotations created in plot edit mode are not. If you move or resize your axes, you will have to reposition your text annotations. For more information, see “Positioning Annotations in Data Space” on page 3-67.

Creating Text Annotations with the text or gtext Function

To create a text annotation using the `text` function, you must specify the text and its location in the graph, using *x*- and *y*-coordinates. You specify the coordinates in the units of the graph.

Use the `gtext` function when you want to position a text annotation at a specific point in the data space with the mouse.

For example, the following code creates text annotations at specific points in the Lotka-Volterra Predator-Prey Population Model graph.

```
str1(1) = {'Many Predators;'};
str1(2) = {'Prey Population'};
str1(3) = {'Will Decline'};
text(7,220,str1)

str2(1) = {'Few Predators;'};
str2(2) = {'Prey Population'};
str2(3) = {'Will Increase'};
text(5.5,125,str2)
```

This example also illustrates how to create multiline text annotations with cell arrays.

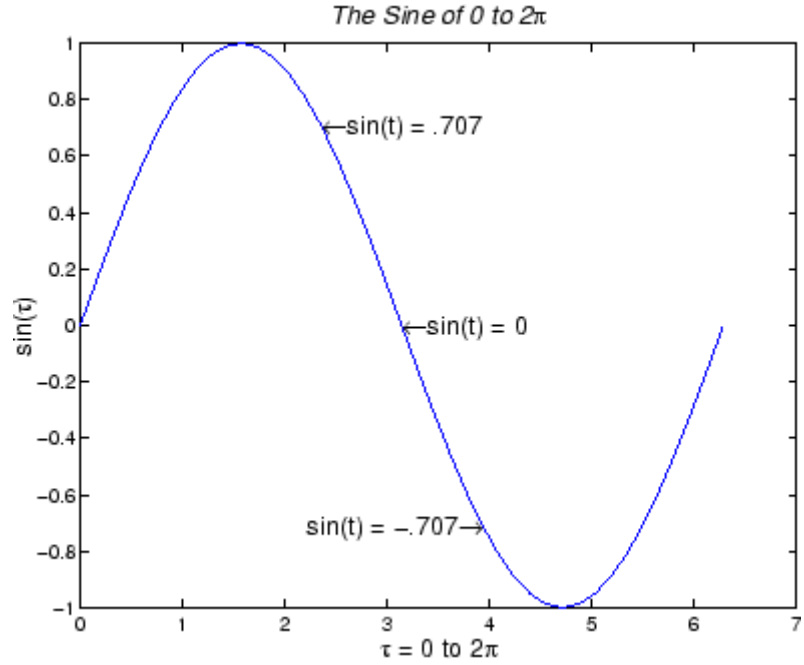
Calculating the Positions of Text Annotations

You can also calculate the positions of text annotations in a graph. The following code adds annotations at three data points on a graph.

```
text(3*pi/4,sin(3*pi/4),...
     '\leftarrowsin(t) = .707',...
     'FontSize',16)

text(pi,sin(pi),'\leftarrowsin(t) = 0',...
     'FontSize',16)

text(5*pi/4,sin(5*pi/4),'sin(t) = -.707\rightarrow',...
     'HorizontalAlignment','right',...
     'FontSize',16)
```

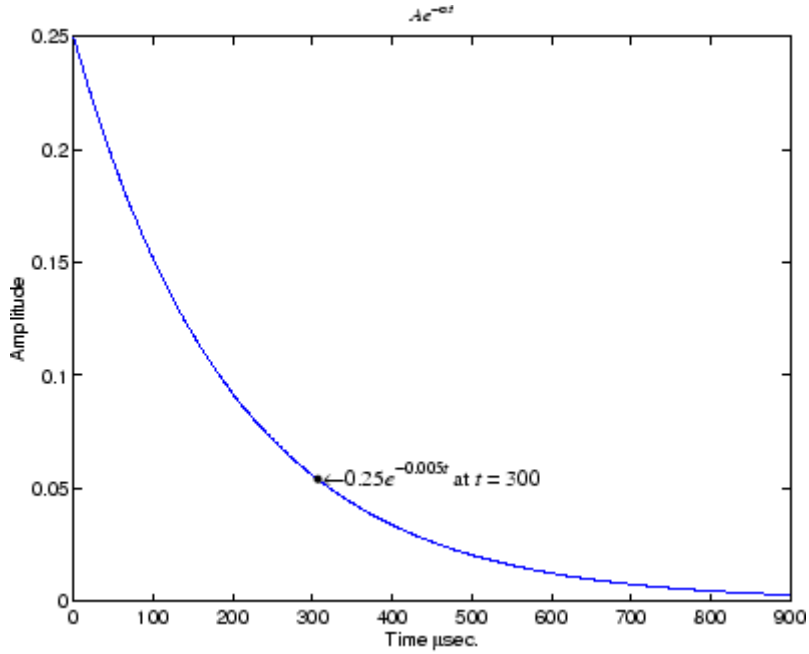
The `HorizontalAlignment` of the text string `'sin(t) = -.707 \rightarrow'` is set to `right` to place it on the left side of the point `[5*pi/4,sin(5*pi/4)]` on the graph. For more information about aligning text annotations, see "Text Alignment" on page 3-51.

Defining Symbols. For information on using symbols in text strings, see "Mathematical Symbols, Greek Letters, and TEX Characters" on page 3-54.



You can use text objects to annotate axes at arbitrary locations. MATLAB locates text in the data units of the axes. For example, suppose you plot the function $y = Ae^{-\alpha t}$ with A = 0.25, α = 0.005, and t = 0 to 900.

```
t = 0:900;
plot(t,0.25*exp(-0.005*t))
```



To annotate the point where the value of t = 300, calculate the text coordinates using the function you are plotting.

```
text(300,.25*exp(-0.005*300),...
'\bullet\leftarrow\fontname{times}0.25{\ite}^{-0.005{\itt}} at {\itt} =
300',...
'FontSize',14)
```

This statement defines the text `Position` property as

x = 300, $y = 0.25e^{-0.005 \times 300}$

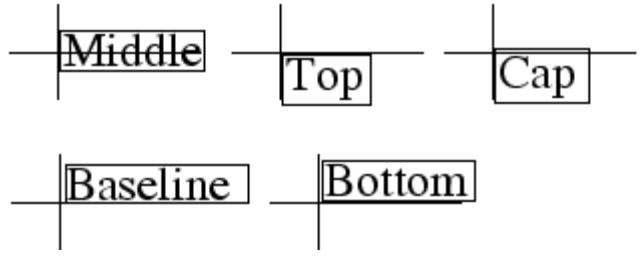
The default text alignment places this point to the left of the string and centered vertically with the rectangle defined by the text `Extent` property. The following section provides more information about changing the default text alignment.

Text Alignment

The `HorizontalAlignment` and the `VerticalAlignment` properties control the placement of the text characters with respect to the specified *x*-, *y*-, and z-coordinates. The following diagram illustrates the options for each property and the corresponding placement of the text.

Text `HorizontalAlignment` property viewed with the `VerticalAlignment` property set to `middle` (the default).

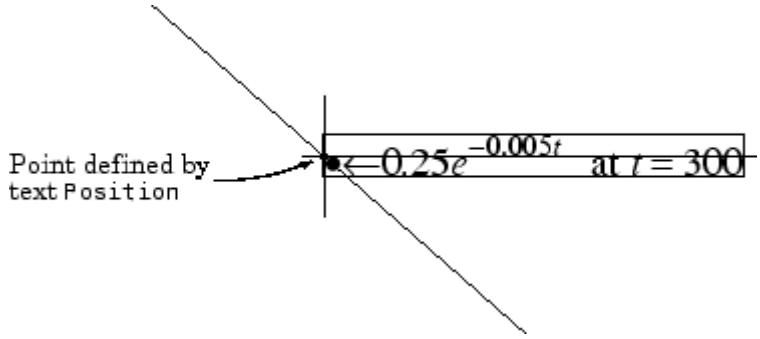
Text `VerticalAlignment` property viewed with the `HorizontalAlignment` property set to `left` (the default).



The default alignment is

- `HorizontalAlignment = left`
- `VerticalAlignment = middle`

MATLAB does not place the text `String` exactly on the specified `Position`. For example, the previous section showed a plot with a point annotated with text. Zooming in on the plot enables you to see the actual positioning of the text.



The small dot is the point specified by the text `Position` property. The larger dot is the bullet defined as the first character in the text `String` property.

Example — Aligning Text

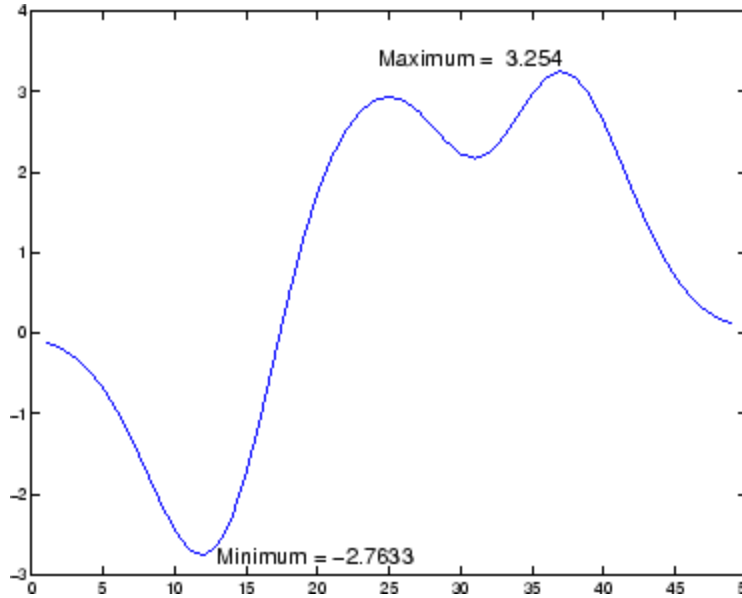
Suppose you want to label the minimum and maximum values in a plot with text that is anchored to these points and that displays the actual values. This example uses the plotted data to determine the location of the text and the values to display on the graph. One column from the `peaks` matrix generates the data to plot.

```
Z = peaks;
h = plot(Z(:,33));
```

The first step is to find the indices of the minimum and maximum values to determine the coordinates needed to position the text at these points (`get`, `find`). Then create the string by concatenating the values with a description of what the values are.

```
x = get(h,'XData');   % Get the plotted data
y = get(h,'YData');
imin = find(min(y) == y);  % Find the index of the min and max
imax = find(max(y) == y);
text(x(imin),y(imin),[' Minimum = ',num2str(y(imin))],...
 'VerticalAlignment','middle',...
 'HorizontalAlignment','left',...
 'FontSize',14)
text(x(imax),y(imax),['Maximum =  ',num2str(y(imax))],...
 'VerticalAlignment','bottom',...
```

```
'HorizontalAlignment','right',...
'FontSize',14)
```



The `text` function positions the string relative to the point specified by the coordinates, in accordance with the settings of the alignment properties. For the minimum value, the string appears to the right of the text position point; for the maximum value the string appears above and to the left of the text position point. The text always remains in the plane of the computer screen, regardless of the view.

Editing Text Objects

You can edit any of the text labels or annotations in a graph:

1. Start plot edit mode.
2. Double-click the string, or right-click the string and select **Edit** from the context menu.

   An editing bar (|) appears next to the text.

3 Make any changes to the text.

4 Click anywhere outside the text edit box to end text editing.

Note To create special characters in text, such as Greek letters or mathematical symbols, use T_EX sequences. See the text `string` property for a table of characters you can use. If you create special characters by using the **Font** dialog box (available via text objects' context menus, and also found in the Property Editor) and selecting the Symbol font family, you cannot edit that text object using MATLAB commands.

Mathematical Symbols, Greek Letters, and TEX Characters

You can include mathematical symbols and Greek letters in text using T_EX-style character sequences. This section describes how to construct a T_EX character sequence.

Two Levels of TEX Support

MATLAB provides two levels of T_EX support, controlled by the text `Interpreter` property:

- `tex` — Support for a subset of T_EX markup
- `latex` — Support for T_EX and LaT_EX markup

If you do not want the characters interpreted as T_EX markup, then set the `interpreter` property to `none`.

Available Symbols and Greek Letters

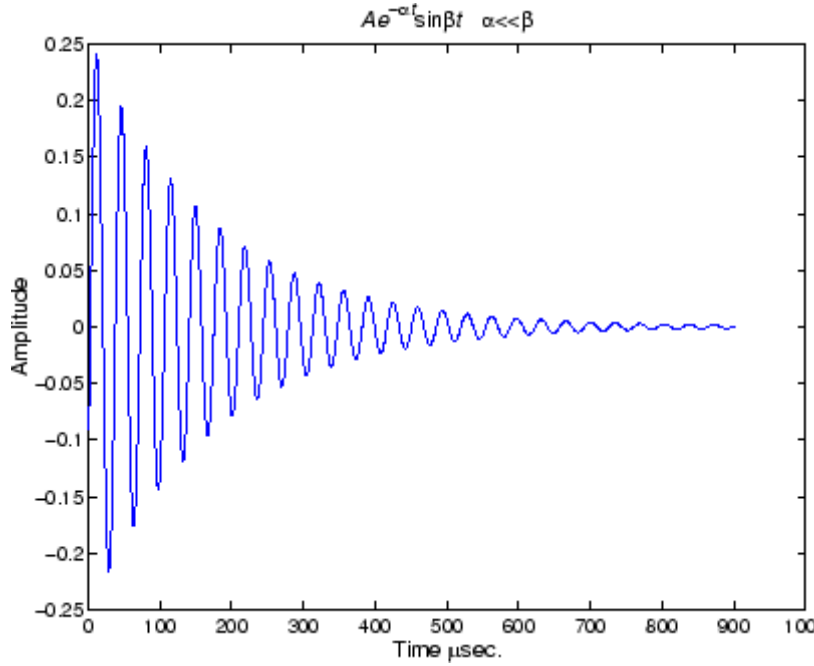
For a list of symbols and the character sequences used to define them, see the table of available T_EX characters in the Text Properties reference page.

In general, you can define text that includes symbols and Greek letters using the `text` function, assigning the character sequence to the `String` property of text objects. You can also include these character sequences in the string arguments of the `title`, `xlabel`, `ylabel`, and `zlabel` functions.

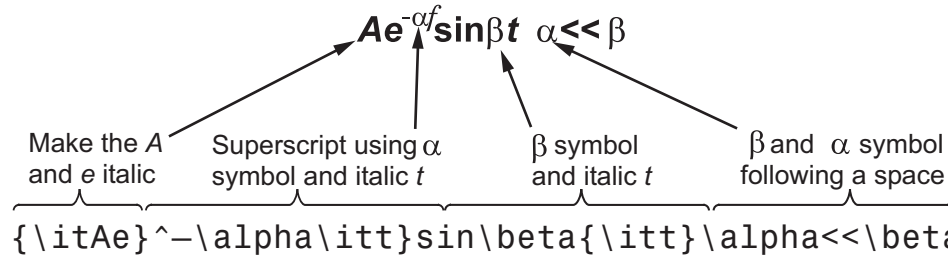
Example — Using a Mathematical Expression to Title a Graph

This example uses T_EX character sequences to create graph labels. The following statements add a title and x- and y-axis labels to an existing graph.

```
title('{\itAe}^{-\alpha\itt}sin\beta{\itt} \alpha<<\beta')
xlabel('Time \musec.')
ylabel('Amplitude')
```



The backslash character (\) precedes all T_EX character sequences. Looking at the string defining the title illustrates how to use these characters.



Controlling the Interpretation of TEX Characters

The text `Interpreter` property controls the interpretation of T_EX characters. If you set this property to `none`, MATLAB interprets the special characters literally.

Using Character and Numeric Variables in Text

Any string variable is a valid specification for the text `String` property. This section illustrates how to use matrix, cell array, and numeric variables as arguments to the `text` function.

Character Variables

For example, each row of the matrix `PersonalData` contains specific information about a person (note that all but the longest row are padded with a space so that each has the same number of columns).

```
PersonalData = ['Jack Straw ';'489 Main St.';'Wichita KN '];
```

To display the data, index into the desired row.

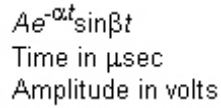
```
text(x1,y1,['Name: ',PersonalData(1,:)])
text(x2,y2,['Address: ',PersonalData(2,:)])
text(x3,y3,['City and State: ',PersonalData(3,:)])
```

Cell Arrays

Using a cell array enables you to create multiline text with a single text object. Each cell does not need to be the same number of characters. For example, the following statements,

```
key(1)={'{\itAe}^{-\alpha\itt}sin\beta{\itt}'};
key(2)={'Time in \musec'};
key(3)={'Amplitude in volts'};
text(x,y,key)
```

produce this output.



Numeric Variables

You can specify numeric variables in text strings using the `num2str` (number to string) function. For example, if you type on the command line

```
x = 21;
['Today is the ',num2str(x),'st day.']
```

MATLAB concatenates the three separate strings into one.

```
Today is the 21st day.
```

Since the result is a valid string, you can specify it as a value for the text `String` property.

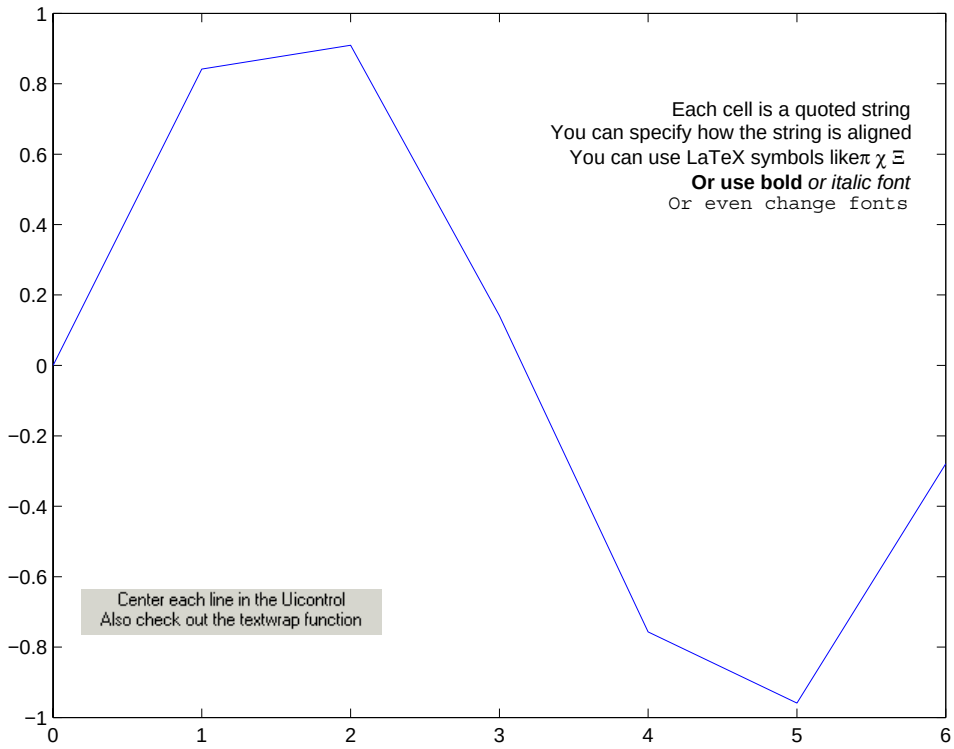
```
text(xcoord,ycoord,['Today is the ',num2str(x),'st day.'])
```

Example — Multiline Text

MATLAB supports multiline text strings using cell arrays. Simply define a string variable as a cell array with one line per cell. This example defines two cell arrays, one used for a `uicontrol` and the other as `text`.

```
str1(1) = {'Center each line in the Uicontrol'};
str1(2) = {'Also check out the textwrap function'};
str2(1) = {'Each cell is a quoted string'};
str2(2) = {'You can specify how the string is aligned'};
str2(3) = {'You can use LaTeX symbols like \pi \chi \Xi'};
str2(4) = {'\bfOr use bold \rm\itor italic font\rm'};
str2(5) = {'\fontname{courier}Or even change fonts'};
```

```
plot(0:6,sin(0:6))
uicontrol('Style','text','Position',[80 80 200 30],...
          'String',str1);
text(5.75,sin(2.5),str2,'HorizontalAlignment','right')
```



Example — Using LaTeX to Format Math Equations

The LaT$_e$X markup language evolved from T$_E$X, and has a superset of its capabilities. LaT$_e$X gives you more elaborate control over specifying and styling mathematical symbols.

The following example illustrates some LaTeX typesetting capabilities when used with the `text` function. Because the default interpreter is for TEX, you need to specify the parameter-value pair `'interpreter','latex'` when typesetting equations such as are contained in the following script:

```
%% LaTeX Examples--Some well known equations rendered in LaTeX
%
figure('color','white','units','inches','position',[2 2 4 6.5]);
axis off

%% A matrix; LaTeX code is
% \hbox {magic(3) is } \left( {\matrix{ 8 & 1 & 6 \cr
% 3 & 5 & 7 \cr 4 & 9 & 2 } } \right)
h(1) = text('units','inch', 'position',[.2 5], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    ['$$\hbox {magic(3) is } \left( {\matrix{ 8 & 1 & 6 \cr'...
    '3 & 5 & 7 \cr 4 & 9 & 2 } } \right)$$']);

%% A 2-D rotation transform; LaTeX code is
%  \left[ {\matrix{\cos(\phi) & -\sin(\phi) \cr
%  \sin(\phi) & \cos(\phi) \cr}}
%  \right] \left[ \matrix{x \cr y} \right]
%
%  $$ \left[ {\matrix{\cos(\phi)
%  & -\sin(\phi) \cr \sin(\phi) & \cos(\phi)  % \cr}}
%  \right] \left[ \matrix{x \cr y} \right] $$
%
h(2) = text('units','inch', 'position',[.2 4], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    ['$$\left[ {\matrix{\cos(\phi) & -\sin(\phi) \cr'...
    '\sin(\phi) & \cos(\phi) \cr}} \right]'...
    '\left[ \matrix{x \cr y} \right]$$']);

%% The Laplace transform; LaTeX code is
%  L\{f(t)\}  \equiv  F(s) = \int_0^\infty\!\!{e^{-st}f(t)dt}
%  $$ L\{f(t)\} \equiv  F(s) = \int_0^\infty\!\!{e^{-st}f(t)dt} $$
%  The Initial Value Theorem for the Laplace transform:
%  \lim_{s \rightarrow \infty} sF(s) = \lim_{t \rightarrow 0} f(t)
%  $$ \lim_{s \rightarrow \infty} sF(s) = \lim_{t \rightarrow 0}
%  f(t) $$
```

```
%
h(3) = text('units','inch', 'position',[.2 3], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    ['$$L\{f(t)\}  \equiv  F(s) = \int_0^\infty\!\!{e^{-st}'...
    'f(t)dt}$$']);

%% The definition of e; LaTeX code is
%  e = \sum_{k=0}^\infty {1 \over {k!} }
%  $$ e = \sum_{k=0}^\infty {1 \over {k!} } $$
%
h(4) = text('units','inch', 'position',[.2 2], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    '$$e = \sum_{k=0}^\infty {1 \over {k!} } $$');

%% Differential equation
% The equation for motion of a falling body with air resistance
% LaTeX code is
%  m \ddot y = -m g + C_D \cdot {1 \over 2} \rho {\dot y}^2 \cdot A
%  $$ m \ddot y = -m g + C_D \cdot {1 \over 2} \rho {\dot y}^2
%  \cdot A  $$
%
h(5) = text('units','inch', 'position',[.2 1], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    ['$$m \ddot y = -m g + C_D \cdot {1 \over 2}'...
    '\rho {\dot y}^2 \cdot A$$']);

%% Integral Equation; LaTeX code is
%  \int_{0}^{\infty} x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{4}
%  $$ \int_{0}^{\infty} x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{4} $$
%
h(6) = text('units','inch', 'position',[.2 0], ...
    'fontsize',14, 'interpreter','latex', 'string',...
    '$$\int_{0}^{\infty} x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{4}$$');
```

$$\text{magic}(3) \text{ is } \begin{pmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{pmatrix}$$

$$\begin{bmatrix} \cos(\phi) & -\sin(\phi) \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$L\{f(t)\} \equiv F(s) = \int_0^\infty e^{-st} f(t) dt$$

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

$$m\ddot{y} = -mg + C_D \cdot \frac{1}{2} \rho \dot{y}^2 \cdot A$$

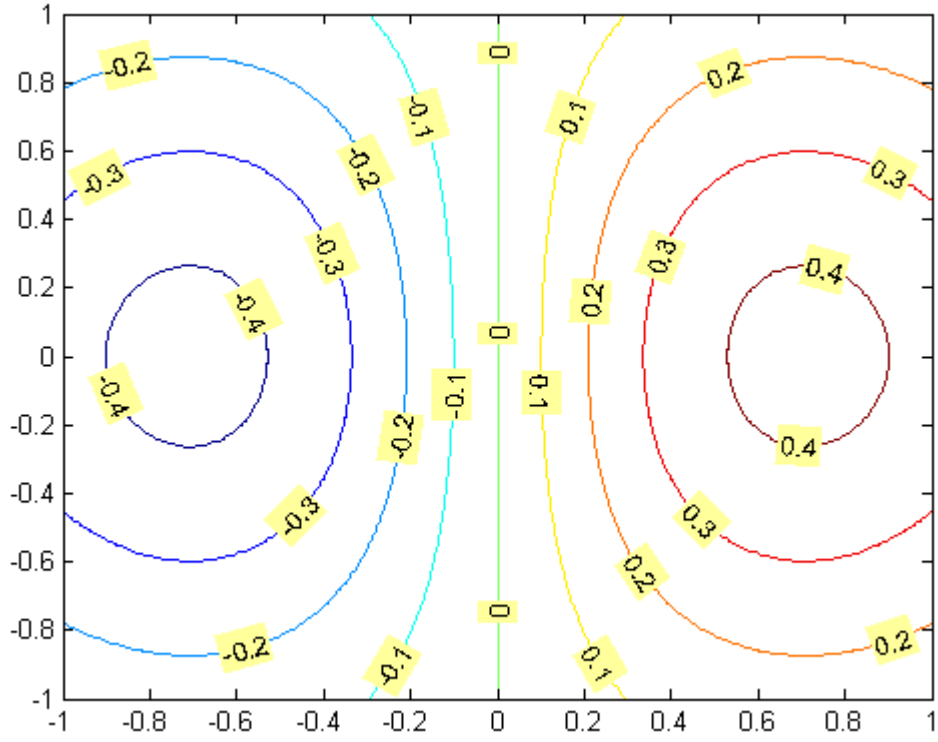
$$\int_0^\infty x^2 e^{-x^2} dx = \frac{\sqrt{\pi}}{4}$$

You can find out more about the LaTeX system at The LaTeX Project Web site, http://www.latex-project.org/.

Drawing Text in a Box

When you use the `text` function to display a character string, the string's position is defined by a rectangle called the `Extent` of the text. You can display this rectangle either as a box or a filled area. For example, you can highlight contour labels to make the text easier to read.

```
[x,y] = meshgrid(-1:.01:1);
z = x.*exp(-x.^2-y.^2);
[c,h]=contour(x,y,z);
h = clabel(c,h);
set(h,'BackgroundColor',[1 1 .6])
```



For additional features, see the following text properties:

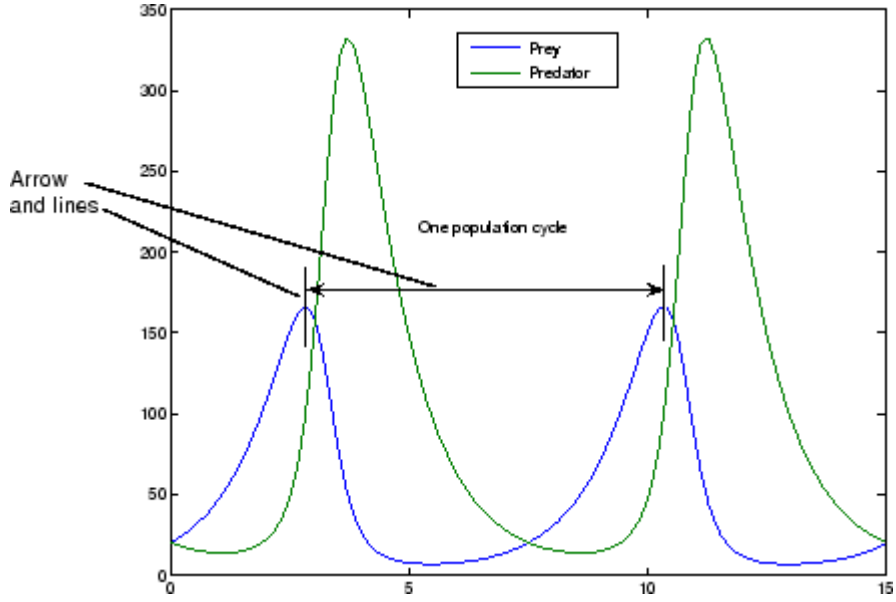
- `BackgroundColor` — Color of the rectangle's interior (none by default)
- `EdgeColor` — Color of the rectangle's edge (none by default)
- `LineStyle` — Style of the rectangle's edge line (first set `EdgeColor`)
- `LineWidth` — Width of the rectangle's edge line (first set `EdgeColor`)
- `Margin` — Increase the size of the rectangle by adding a margin to the text extent.

Adding Arrows and Lines to Graphs

In this section...

Creating Arrows and Lines in Plot Editing Mode

With plot editing mode enabled, you can add arrows and lines anywhere in a figure window.



You can also use arrow characters (TeX characters) to create arrows using the `text` function. However, arrows created this way can only point to the left or right, horizontally. See “Calculating the Positions of Text Annotations” on page 3-48 for an example.

To add an arrow or line annotation to a graph,

1 Click the **Insert** menu and choose the **Arrow** or **Line** option, or click the **Arrow** or **Line** button in the Plot Edit toolbar.

   MATLAB changes the cursor to a cross-hair.

2 Position the cursor in the figure where you want to start the line or arrow and press either mouse button. Hold the button down and move the mouse to define the length and direction of the line or arrow.

3 Release the mouse button.

Note After you add an arrow or line, plot edit mode is enabled in the figure, if it was not already enabled.

Editing Arrows and Line Annotations

You can edit the appearance of arrow and line annotations using the context menu.

With plot editing mode enabled, right-click the arrow or line annotation to display its context menu.

You can select an annotation and then choose **Show M-code** to obtain a code snippet that you can insert in a function or script to reproduce the annotation.

For more options, select **Properties** to display the Property Editor.

Positioning Annotations in Data Space

Example — Pinning Textarrows and Ellipses

Some annotation object types (arrow, doublearrow, textarrow, and ellipse) are attached to figures rather than to axes. This makes it difficult to programmatically place them precisely on an axes, especially if the axes changes its position. You can, however, still locate figure-based annotations in data space by applying a transformation to their coordinates. The following example shows how to do this using a function called `dsxy2figxy` that transforms figure coordinates to axes coordinates for the four types of annotations listed above.

Before following the steps given below, copy the code for `dsxy2figxy` and save in your current directory or elsewhere on the MATLAB path.

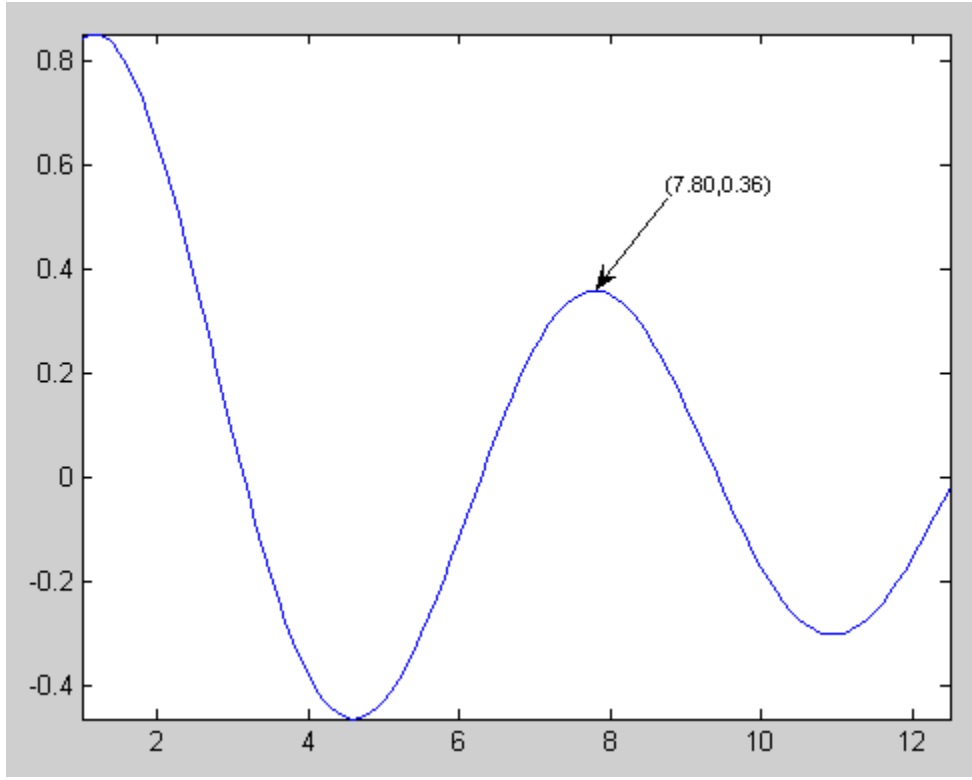
1 Create sine function data and make a line plot of it:

```
x1 = 1:.1:4*pi;
y1 = sin(x1)./sqrt(x1);
figure
plot(x1,y1)
axis tight
```

2 Interactively place a textarrow on the graph

Using `ginput`, interactively locate a textarrow annotation. Two clicks are required by `ginput`:

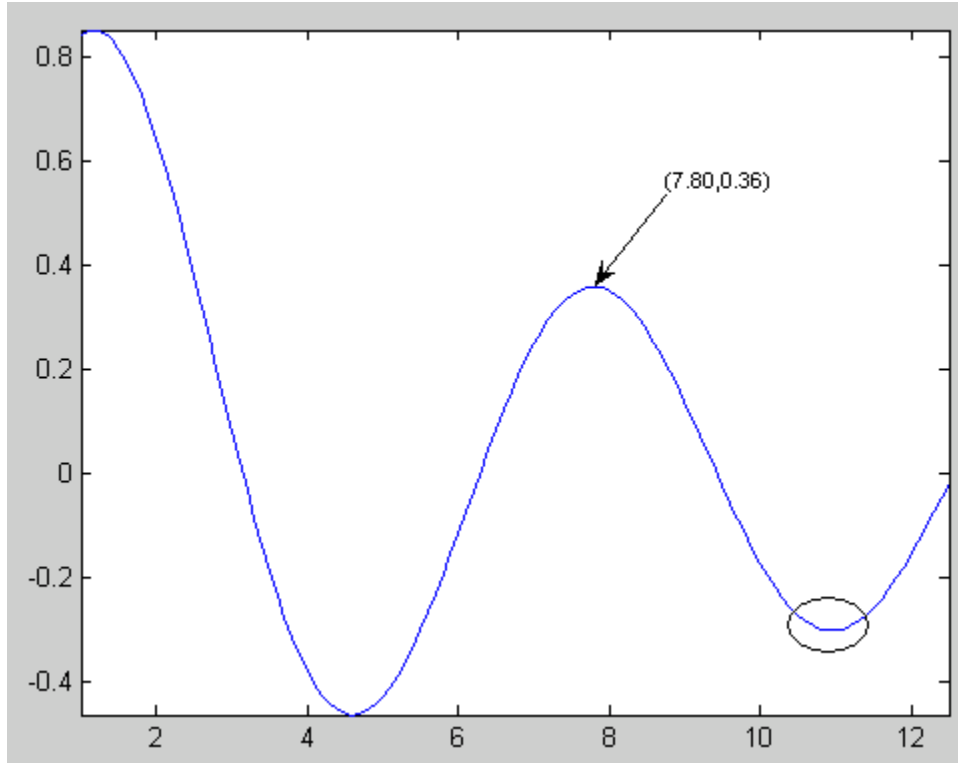
```
disp('Click graph to place arrow; first tail, then head:')
[axx axy] = ginput(2);      % Returns list of x, list of y in data space
% Get coords in figure space; gca will be correct even for subplots
% Transform from data space to fig space
[arrowx,arrowy] = dsxy2figxy(gca, axx, axy);
har = annotation('textarrow',arrowx,arrowy);
content = sprintf('(%4.2f,%4.2f)',axx(2), axy(2));
% Plot anno text centered at the tail of the arrow
set(har,'String',content,'Fontsize',8)
```



3 Place an ellipse on the axes

To place ellipses, a coordinate box (position rectangle) is needed instead of two *x*-*y* tuples. The function `dsxy2figxy` computes and returns a position rectangle if it is called with one:

```
disp('Click in the axes to define the bounding box of an ellipse:')
[axx axy] = ginput(2);      % Returns list of x, list of y in data space
abox(1) = min(axx); abox(2) = min(axy);     % Get least x and y coords
abox(3) = abs(axx(1)-axx(2));               % Get box width
abox(4) = abs(axy(1)-axy(2));               % Get box height
% Get coords in figure space; gca will be correct even for subplots
[bbox] = dsxy2figxy(gca, abox);    % Xform from axes to fig space
annotation('ellipse',bbox);
```



Here is the code for `dsxy2figxy` ; copy it to an M-file that can be called when you execute the example above:

```
function varargout = dsxy2figxy(varargin)
% dsxy2figxy -- Transform point or position from axis to figure coords
% Transforms [axx axy] or [xypos] from axes hAx (data) coords into coords
% wrt GCF for placing annotation objects that use figure coords into data
% space. The annotation objects this can be used for are
%    arrow, doublearrow, textarrow
%    ellipses (coordinates must be transformed to [x, y, width, height])
% Note that line, text, and rectangle anno objects already are placed
% on a plot using axes coordinates and must be located within an axes.
% Usage: Compute a position and apply to an annotation, e.g.,
%   [axx axy] = ginput(2);
%   [figx figy] = getaxannopos(gca, axx, axy);
```

```
%   har = annotation('textarrow',figx,figy);
%   set(har,'String',['(' num2str(axx(2)) ',' num2str(axy(2)) ')'])

%% Obtain arguments (only limited argument checking is performed).
% Determine if axes handle is specified
if length(varargin{1})== 1 && ishandle(varargin{1}) && ...
  strcmp(get(varargin{1},'type'),'axes')
 hAx = varargin{1};
 varargin = varargin(2:end);
else
 hAx = gca;
end;
% Parse either a position vector or two 2-D point tuples
if length(varargin)==1 % Must be a 4-element POS vector
 pos = varargin{1};
else
 [x,y] = deal(varargin{:});  % Two tuples (start & end points)
end
%% Get limits
axun = get(hAx,'Units');
set(hAx,'Units','normalized');  % Need normaized units to do the xform
axpos = get(hAx,'Position');
axlim = axis(hAx);              % Get the axis limits [xlim ylim (zlim)]
axwidth = diff(axlim(1:2));
axheight = diff(axlim(3:4));
%% Transform data from figure space to data space
if exist('x','var')     % Transform a and return pair of points
 varargout{1} = (x-axlim(1))*axpos(3)/axwidth + axpos(1);
 varargout{2} = (y-axlim(3))*axpos(4)/axheight + axpos(2);
else                    % Transform and return a position rectangle
 pos(1) = (pos(1)-axlim(1))/axwidth*axpos(3) + axpos(1);
 pos(2) = (pos(2)-axlim(3))/axheight*axpos(4) + axpos(2);
 pos(3) = pos(3)*axpos(3)/axwidth;
 pos(4) = pos(4)*axpos(4)/axheight;
 varargout{1} = pos;
end
%% Restore axes units
set(hAx,'Units',axun)
```

4

Basic Plotting Commands

Setting Up Figures

In this section...

Creating Figure Windows

MATLAB directs graphics output to a window that is separate from the Command Window. In MATLAB this window is referred to as a *figure*. The characteristics of this window are controlled by your computer's windowing system and MATLAB figure properties (see a description of each property). See Chapter 9, "Figure Properties" for some examples illustrating how to use figure properties.

Graphics functions automatically create new figure windows if none currently exist. If a figure already exists, MATLAB uses that window. If multiple figures exist, one is designated as the *current figure* and is used by MATLAB (this is generally the last figure used or the last figure you clicked the mouse in).

The `figure` function creates figure windows. For example,

```
figure
```

creates a new window and makes it the current figure. You can make an existing figure current by clicking it with the mouse or by passing its handle (the number indicated in the window title bar), as an argument to `figure`.

```
figure(h)
```

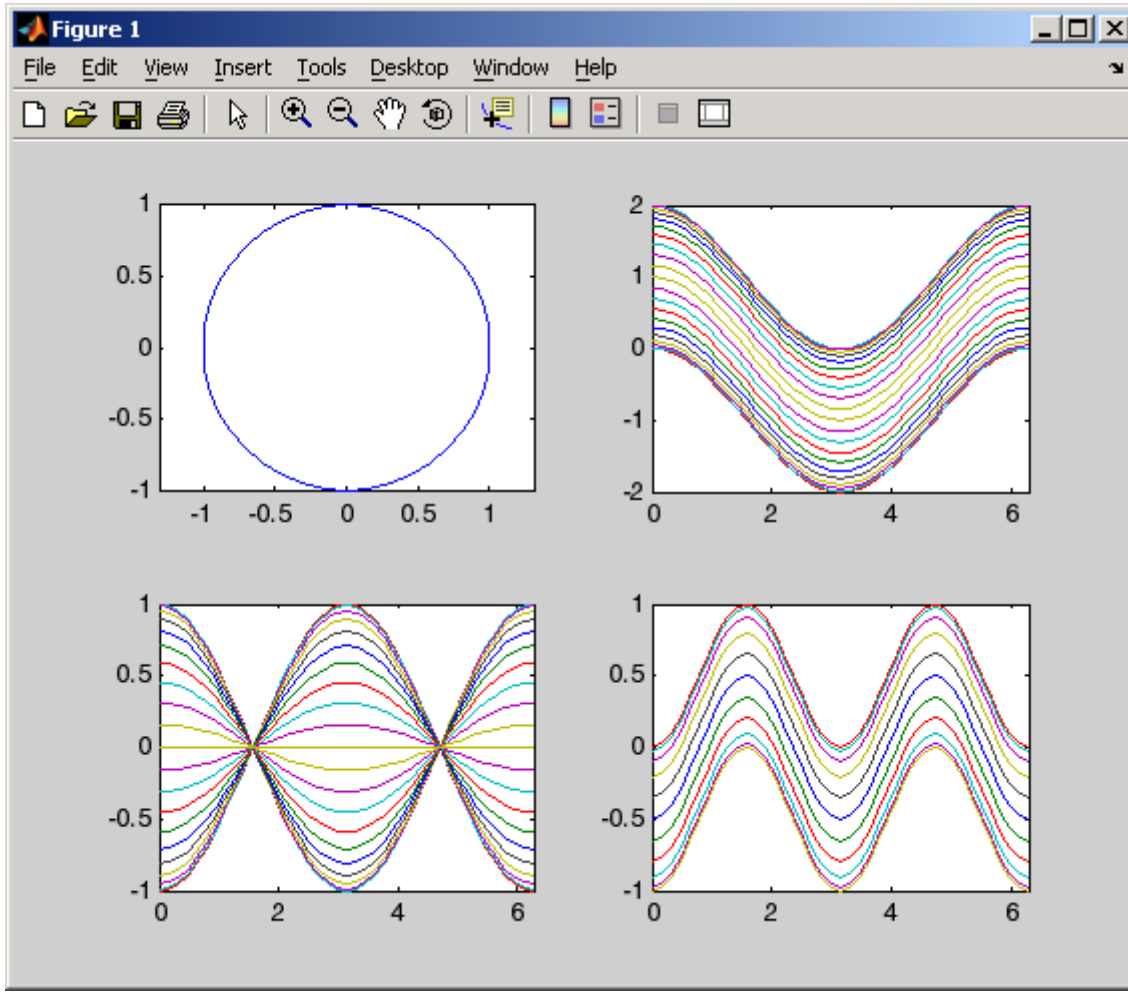
Displaying Multiple Plots per Figure

You can display multiple plots in the same figure window and print them on the same piece of paper with the `subplot` function.

`subplot(m,n,i)` breaks the figure window into an *m*-by-*n* matrix of small subplots and selects the `i`the subplot for the current plot. The plots are numbered along the top row of the figure window, then the second row, and so forth.

For example, the following statements plot data in four different subregions of the figure window.

```
t = 0:pi/20:2*pi;
[x,y] = meshgrid(t);
subplot(2,2,1)
plot(sin(t),cos(t))
axis equal
subplot(2,2,2)
z = sin(x)+cos(y);
plot(t,z)
axis([0 2*pi -2 2])
subplot(2,2,3)
z = sin(x).*cos(y);
plot(t,z)
axis([0 2*pi -1 1])
subplot(2,2,4)
z = (sin(x).^2)-(cos(y).^2);
plot(t,z)
axis([0 2*pi -1 1])
```



Each subregion contains its own axes with characteristics you can control independently of the other subregions. This example uses the `axis` function to set limits and change the shape of the subplots.

See the `axes`, `axis`, and `subplot` functions for more information.

Specifying the Target Axes

The current axes is the last one defined by `subplot`. If you want to access a previously defined subplot, for example to add a title, you must first make that axes current.

You can make an axes current in three ways:

- Click on the subplot with the mouse.
- Call `subplot` the `m`, `n`, `i` specifiers.
- Call `subplot` with the handle (identifier) of the axes.

For example,

```
subplot(2,2,2)
title('Top Right Plot')
```

adds a title to the plot in the upper right side of the figure.

You can obtain the handles of all the subplot axes with the statement

```
h = get(gcf,'Children');
```

MATLAB returns the handles of all the axes, with the most recently created one first. That is, `h(1)` is subplot 224, `h(2)` is subplot 223, `h(3)` is subplot 222, and `h(4)` is subplot 221. For example, to replace subplot 222 with a new plot, first make it the current axes with

```
subplot(h(3))
```

Default Color Scheme

The default figure color scheme produces good contrast and visibility for the various graphics functions. This scheme defines colors for the window background, the axis background, the axis lines and labels, the colors of the lines used for plotting and surface edges, and other properties that affect appearance.

The `colordef` function enables you to select from predefined color schemes and to modify colors individually. `colordef` predefines three color schemes:

- `colordef white` — Sets the axis background color to white, the window background color to gray, the colormap to `jet`, surface edge colors to black, and defines appropriate values for the plotting color order and other properties.
- `colordef black` — Sets the axis background color to black, the window background color to dark gray, the colormap to `jet`, surface edge colors to black, and defines appropriate values for the plotting color order and other properties.
- `colordef none` — Set the colors to match that of MATLAB 4. This is basically a black background with white axis lines and no grid. MATLAB programs that are based on the MATLAB 4 color scheme may need to call `colordef` with the `none` option to produce the expected results.

You can examine the `colordef.m` M-file to determine what properties it sets (enter `type colordef` at the MATLAB prompt).

Using High-Level Plotting Functions

In this section...

Functions for Plotting Line Graphs

MATLAB provides a variety of functions for displaying vector data as line plots, as well as functions for annotating and printing these graphs. The following table summarizes the functions that produce basic line plots. These functions differ in the way they scale the plot's axes. Each accepts input in the form of vectors or matrices and automatically scales the axes to accommodate the data.

Function	Description
`plot`	Graph 2-D data with linear scales for both axes
`plot3`	Graph 3-D data with linear scales for both axes
`loglog`	Graph with logarithmic scales for both axes
`semilogx`	Graph with a logarithmic scale for the *x*-axis and a linear scale for the *y*-axis

Function	Description
`semilogy`	Graph with a logarithmic scale for the *y*-axis and a linear scale for the *x*-axis
`plotyy`	Graph with *y*-tick labels on the left and right side

For a synopsis of all the high level plot functions with links to their reference pages, see "Types of Plots Available in MATLAB" on page 1-6.

Programmatic Plotting

The process of constructing a basic graph to meet your presentation graphics requirements is outlined in the following table. The table shows seven typical steps and some example code for each.

If you are performing analysis only, you may want to view various graphs just to explore your data. In this case, steps 1 and 3 may be all you need. If you are creating presentation graphics, you may want to fine-tune your graph by positioning it on the page, setting line styles and colors, adding annotations, and making other such improvements.

Step	Typical Code

1 Prepare your data	`x = 0:0.2:12;`
	`y1 = bessel(1,x);`
	`y2 = bessel(2,x);`
	`y3 = bessel(3,x);`

2 Select a window and position a plot region within the window	`figure(1)` `subplot(2,2,1)`
3 Call elementary plotting function	`h = plot(x,y1,x,y2,x,y3);`

4 Select line and marker characteristics	`set(h,'LineWidth',2,{'LineStyle'},`
	`{'--';':';'-.'})`
	`set(h,{'Color'},{'r';'g';'b'})`

Step	Typical Code
5 Set axis limits, tick marks, and grid lines	`axis([0 12 -0.5 1])` `grid on`

6 Annotate the graph with axis labels, legend, and text	`xlabel('Time')`
	`ylabel('Amplitude')`
	`legend(h,'First','Second','Third')`
	`title('Bessel Functions')`
	`[y,ix] = min(y1);`
	`text(x(ix),y,'First Min \rightarrow',...`
	`'HorizontalAlignment','right')`

7 Export graph	`print -depsc -tiff -r200 myplot`

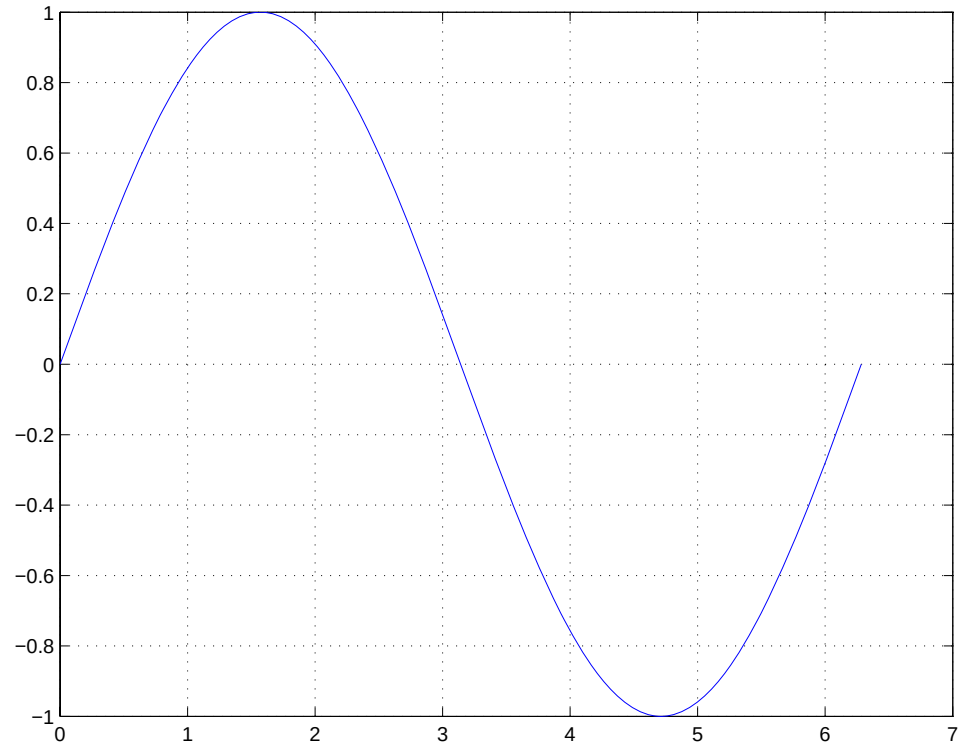
Creating Line Plots

The `plot` function has different forms depending on the input arguments. For example, if `y` is a vector, `plot(y)` produces a linear graph of the elements of `y` versus the index of the elements of `y`. If you specify two vectors as arguments, `plot(x,y)` produces a graph of `y` versus `x`.

For example, the following statements create a vector of values in the range [0, 2π] in increments of π/100 and then use this vector to evaluate the sine function over that range. MATLAB plots the vector on the *x*-axis and the value of the sine function on the *y*-axis.

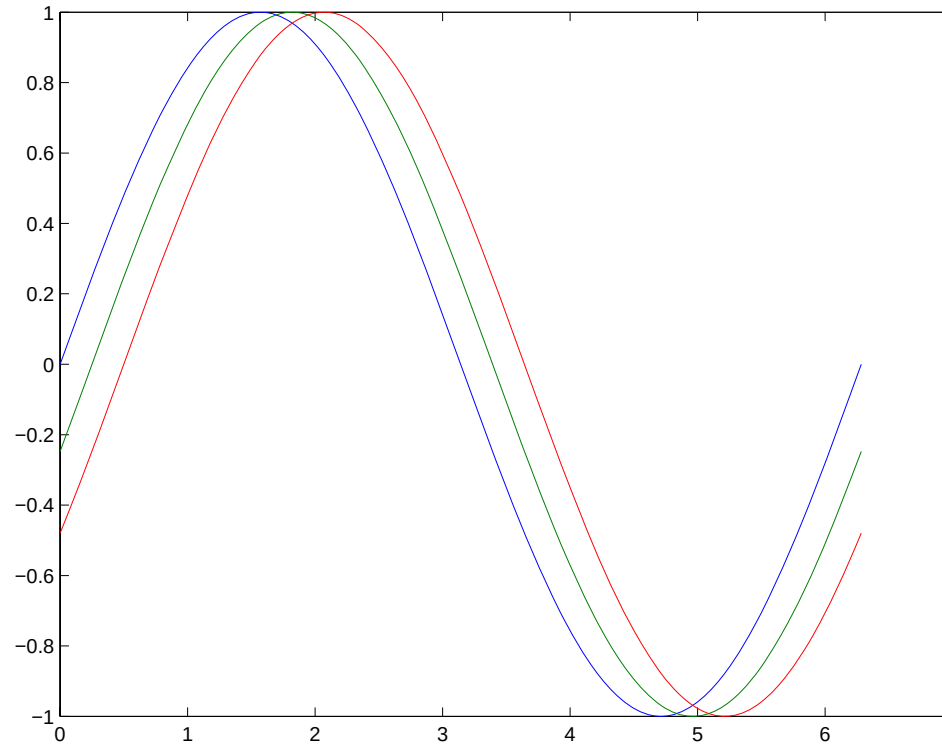
```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y)
grid on % Turn on grid lines for this plot
```

MATLAB automatically selects appropriate axis ranges and tick mark locations.



You can plot multiple graphs in one call to `plot` using *x-y* pairs. MATLAB automatically cycles through a predefined list of colors (determined by the axes `ColorOrder` property) to allow discrimination between sets of data. Plotting three curves as a function of `t` produces

```
y = sin(t);
y2 = sin(t-0.25);
y3 = sin(t-0.5);
plot(t,y,t,y2,t,y3)
```

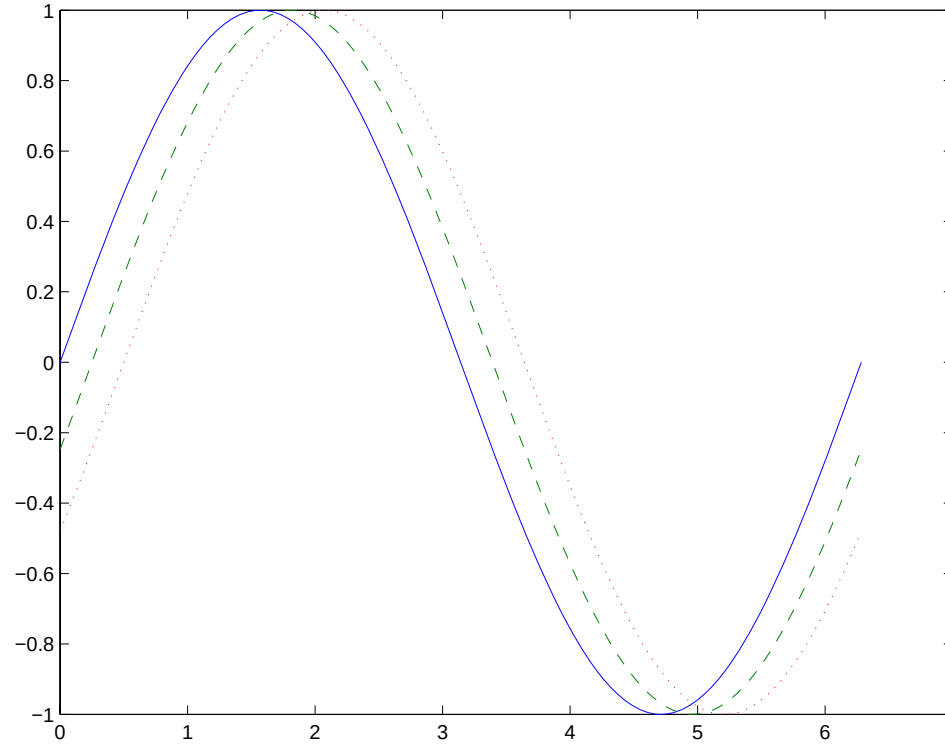


Specifying Line Style

You can assign different line styles to each data set by passing line style identifier strings to `plot`. For example,

```
t = 0:pi/100:2*pi;
y = sin(t);
y2 = sin(t-0.25);
y3 = sin(t-0.5);
plot(t,y,'-',t,y2,'--',t,y3,':')
```

The graph shows three lines of different colors and lines styles representing the value of the sine function with a small phase shift between each line, as defined by `y`, `y2`, and `y3`. The lines are blue solid, green dashed, and red dotted.



Colors, Line Styles, and Markers

The basic plotting functions accepts character-string arguments that specify various line styles, marker symbols, and colors for each vector plotted. In the general form,

```
plot(x,y,'linestyle_marker_color')
```

`linestyle_marker_color` is a character string (delineated by single quotation marks) constructed from

- A line style (e.g., dashed, dotted, etc.)
- A marker type (e.g., `x`, `*`, `o`, etc.)
- A predefined color specifier (`c`, `m`, `y`, `k`, `r`, `g`, `b`, `w`)

For example,

```
plot(x,y,':squarey')
```

plots a yellow dotted line and places square markers at each data point. If you specify a marker type, but not a line style, MATLAB draws only the marker.

The specification can consist of one or none of each specifier in any order. For example, the string

```
'go--'
```

defines a dashed line with circular markers, both colored green.

You can also specify the size of the marker and, for markers that are closed shapes, you can specify separately the colors of the edges and the face.

See the `LineSpec` discussion for more information.

Specifying the Color and Size of Lines

You can control a number of line style characteristics by specifying values for line properties:

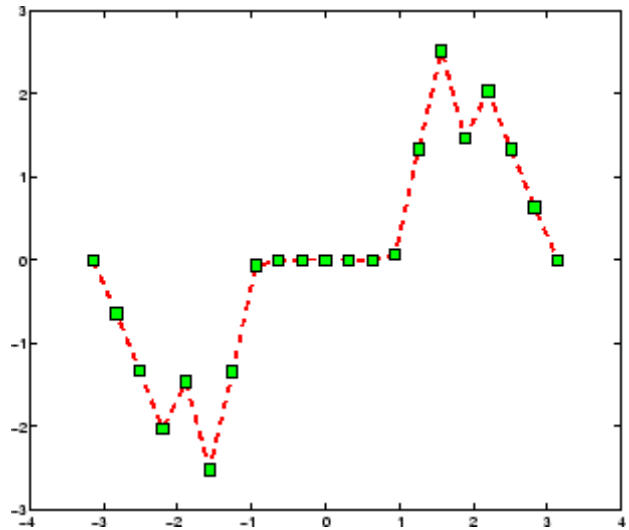
- `LineWidth` — Width of the line in units of points
- `MarkerEdgeColor` — Color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles)
- `MarkerFaceColor` — Color of the face of filled markers
- `MarkerSize` — Size of the marker in units of points

For example, these statements,

```
x = -pi:pi/10:pi;
y = tan(sin(x)) - sin(tan(x));
plot(x,y,'--rs','LineWidth',2,...
                'MarkerEdgeColor','k',...
                'MarkerFaceColor','g',...
                'MarkerSize',10)
```

produce a graph with

- A red dashed line with square markers
- A line width of two points
- The edge of the marker colored black
- The face of the marker colored green
- The size of the marker set to 10 points



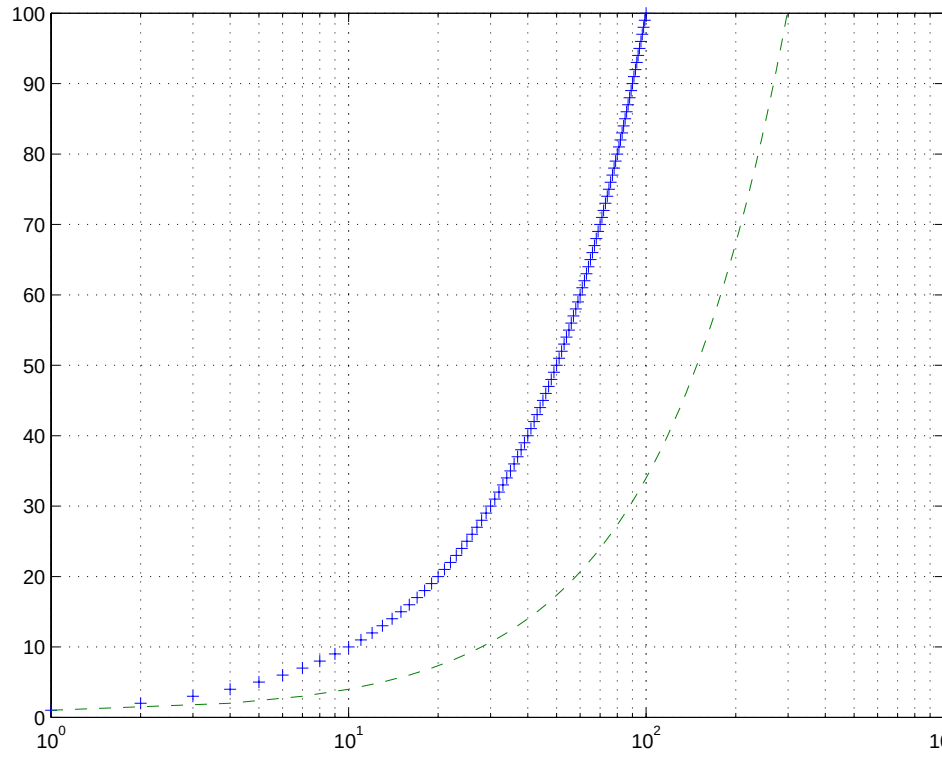
Adding Plots to an Existing Graph

You can add plots to an existing graph using the `hold` command. When you set `hold` to `on`, MATLAB does not remove the existing graph; it adds the new data to the current graph, rescaling if the new data falls outside the range of the previous axis limits.

For example, these statements first create a semilogarithmic plot, then add a linear plot.

```
semilogx(1:100,'+')
hold all % hold plot and cycle line colors
plot(1:3:300,1:100,'--')
hold off
grid on % Turn on grid lines for this plot
```

While MATLAB resets the *x*-axis limits to accommodate the new data, it does not change the scaling from logarithmic to linear.



Plotting Only the Data Points

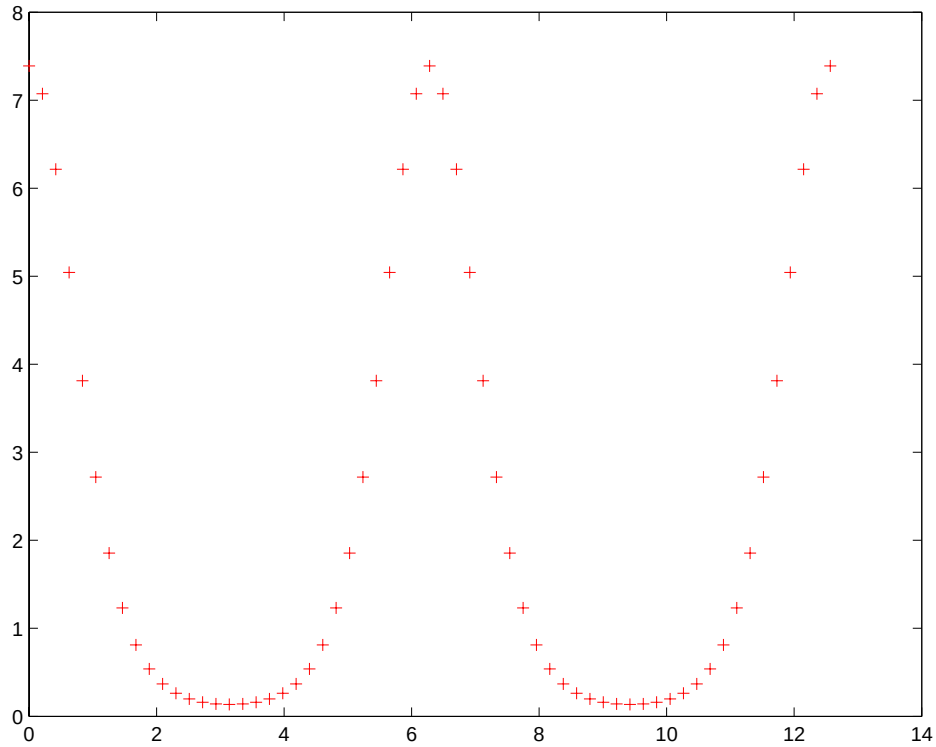
To plot a marker at each data point without connecting the markers with lines, use a specification that does not contain a line style. For example, given two vectors,

```
x = 0:pi/15:4*pi;
y = exp(2*cos(x));
```

calling `plot` with only a color and marker specifier

```
plot(x,y,'r+')
```

plots a red plus sign at each data point.

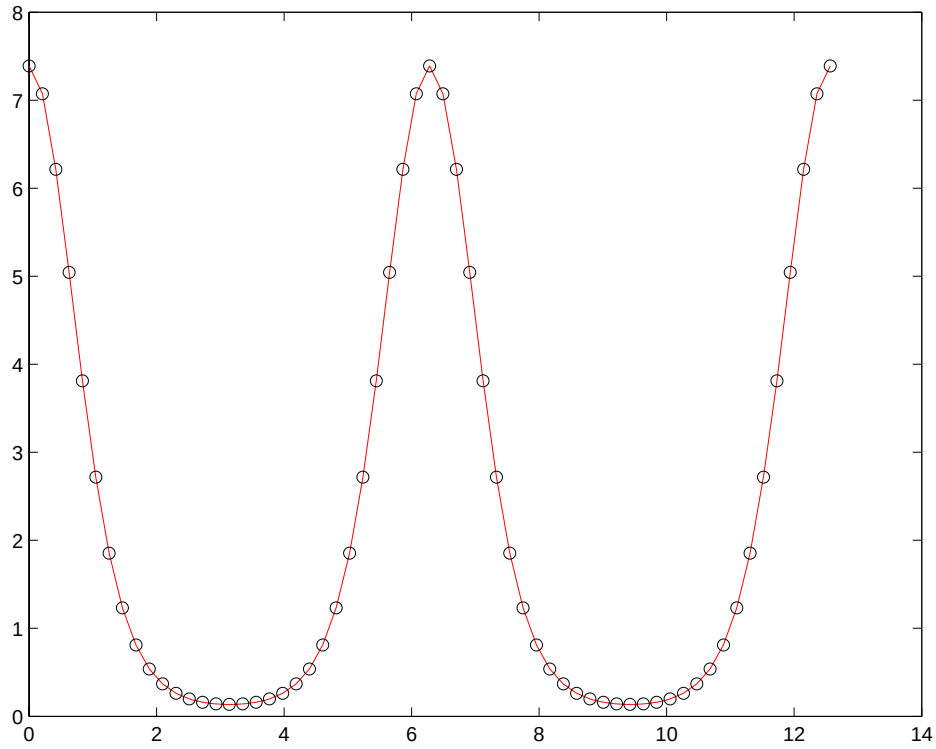


See `LineSpec` for a list of available line styles, markers, and colors.

Plotting Markers and Lines

To plot both markers and the lines that connect them, specify a line style and a marker type. For example, the following code plots the data as a red, solid line and then adds circular markers with black edges at each data point.

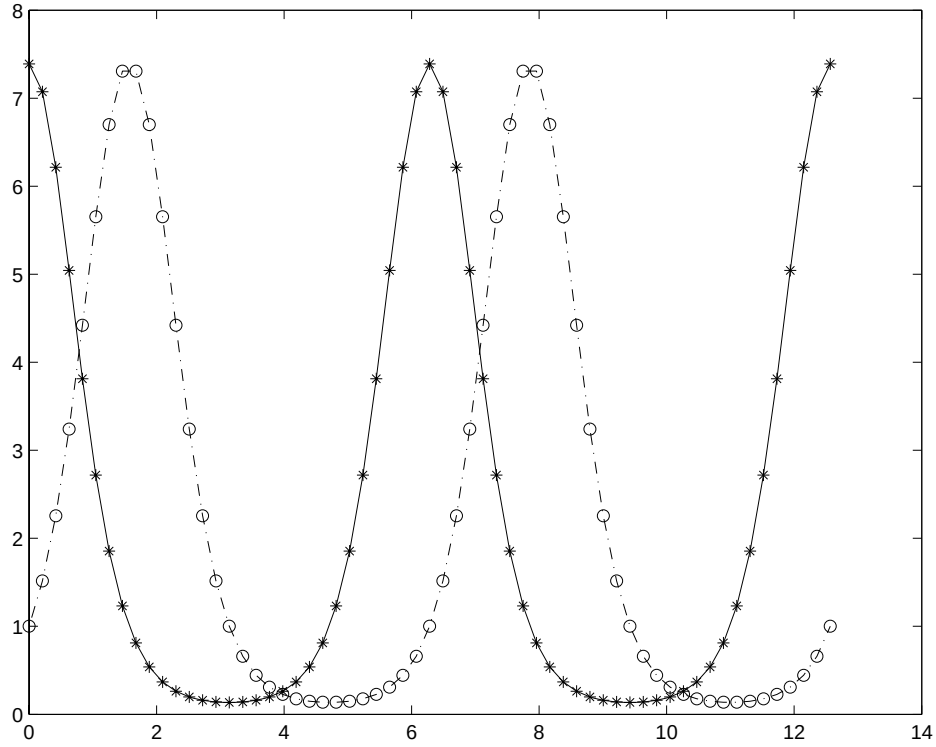
```
x = 0:pi/15:4*pi;
y = exp(2*cos(x));
plot(x,y,'-r',x,y,'ok')
```



Line Styles for Black and White Output

Line styles and markers enable you to discriminate different plots on the same graph when color is not available. For example, the following statements create a graph using a solid (`'-*k'`) line with asterisk markers colored black and a dash-dot (`'-.ok'`) line with circular markers colored black.

```
x = 0:pi/15:4*pi;
y1 = exp(2*cos(x));
y2 = exp(2*sin(x));
plot(x,y1,'-*k',x,y2,'-.ok')
```



Setting Default Line Styles

You can configure MATLAB to use line styles instead of colors for multiline plots by setting a default value for the axes `LineStyleOrder` property. For example, the command

```
set(0,'DefaultAxesLineStyleOrder',{'-o',':s','--+'})
```

defines three line styles and makes them the default for all plots.

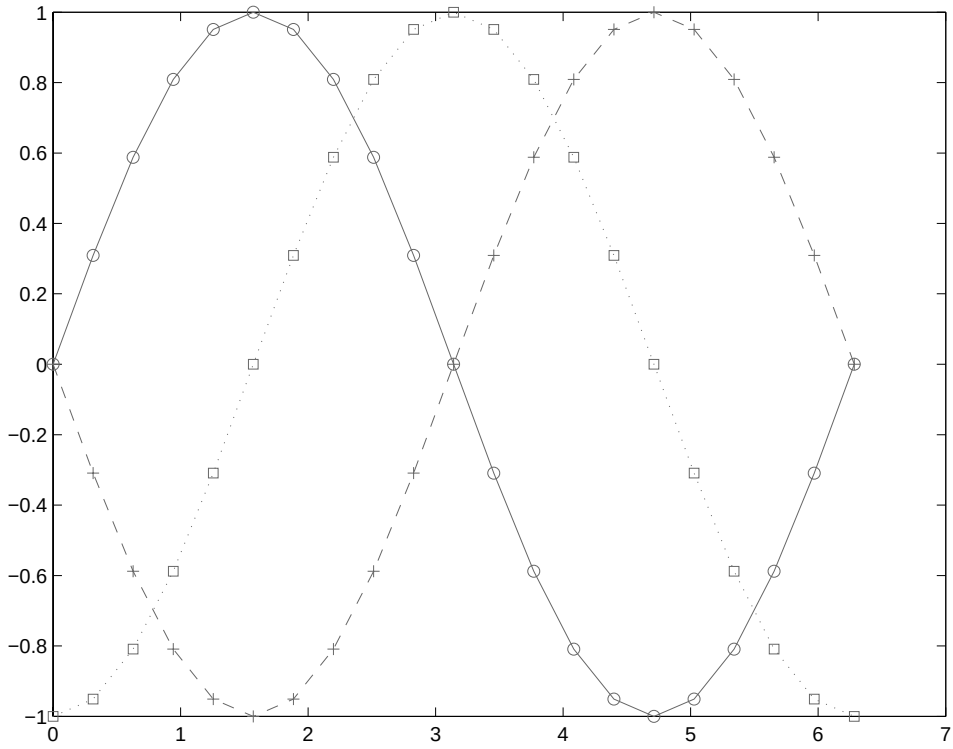
To set the default line color to dark gray, use the statement

```
set(0,'DefaultAxesColorOrder',[0.4,0.4,0.4])
```

See `ColorSpec` for information on how to specify color as a three-element vector of RGB values.

Now the plot function uses the line styles and colors you have defined as defaults. For example, these statements create a multiline plot.

```
x = 0:pi/10:2*pi;
y1 = sin(x);
y2 = sin(x-pi/2);
y3 = sin(x-pi);
plot(x,y1,x,y2,x,y3)
```



The default values persist until you quit MATLAB. To remove default values during your MATLAB session, use the reserved word `remove`.

```
set(0,'DefaultAxesLineStyleOrder','remove')
set(0,'DefaultAxesColorOrder','remove')
```

See “Setting Default Property Values” on page 8-51 for more information.

Line Plots of Matrix Data

When you call the `plot` function with a single matrix argument

```
plot(Y)
```

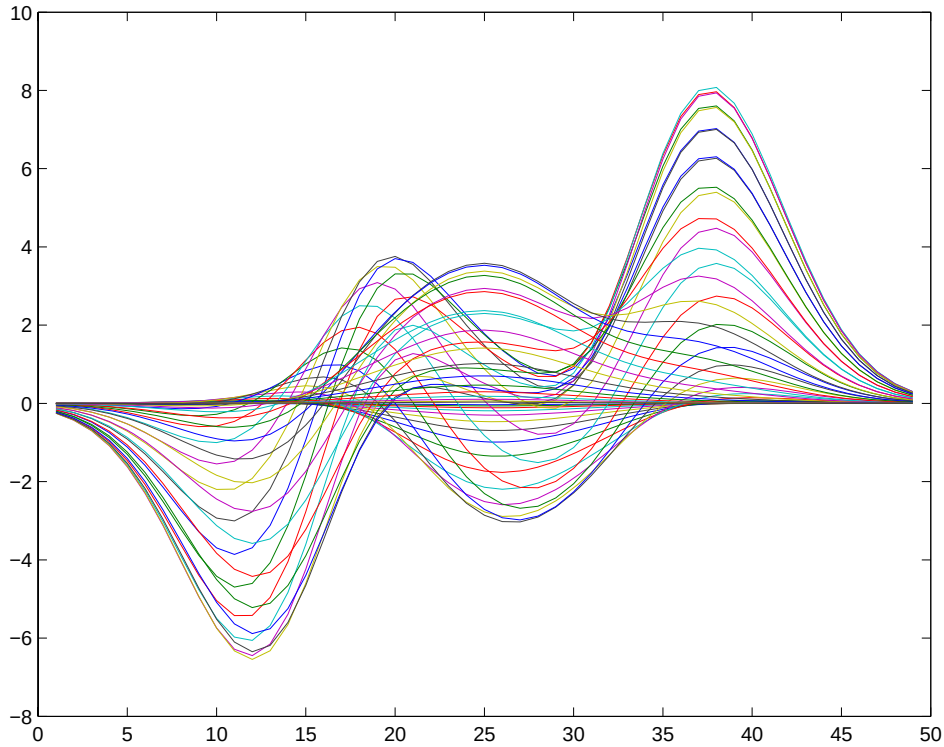
MATLAB draws one line for each column of the matrix. The *x*-axis is labeled with the row index vector `1:m`, where `m` is the number of rows in `Y`. For example,

```
Z = peaks;
```

returns a 49-by-49 matrix obtained by evaluating a function of two variables. Plotting this matrix

```
plot(Z)
```

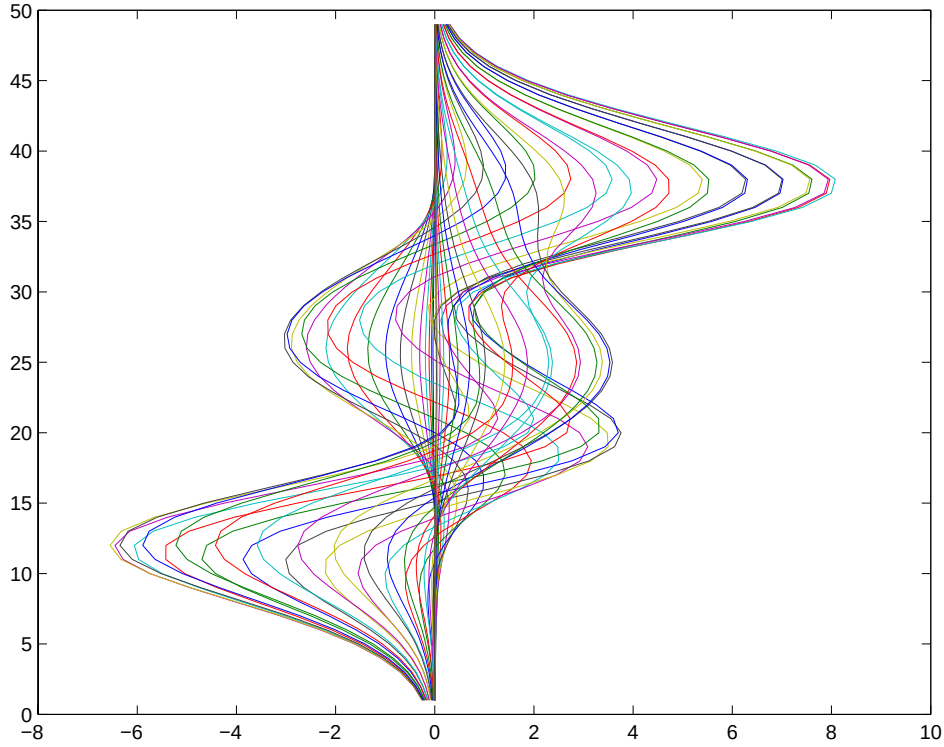
produces a graph with 49 lines.



In general, if `plot` is used with two arguments and if either `X` or `Y` has more than one row or column, then

- If `Y` is a matrix, and `x` is a vector, `plot(x,Y)` successively plots the rows or columns of `Y` versus vector `x`, using different colors or line types for each. The row or column orientation varies depending on whether the number of elements in `x` matches the number of rows in `Y` or the number of columns. If `Y` is square, its columns are used.
- If `X` is a matrix and `y` is a vector, `plot(X,y)` plots each row or column of `X` versus vector `y`. For example, plotting the `peaks` matrix versus the vector `1:length(peaks)` rotates the previous plot.

```
y = 1:length(peaks);
plot(peaks,y)
```



- If `X` and `Y` are both matrices of the same size, `plot(X,Y)` plots the columns of `X` versus the columns of `Y`.

You can also use the `plot` function with multiple pairs of matrix arguments.

```
plot(X1,Y1,X2,Y2,...)
```

This statement graphs each `X-Y` pair, generating multiple lines. The different pairs can be of different dimensions.

Plotting Imaginary and Complex Data

When the arguments to `plot` are complex (i.e., the imaginary part is nonzero), MATLAB ignores the imaginary part *except* when `plot` is given a single complex data argument. For this special case, the command is a shortcut for a plot of the real part versus the imaginary part. Therefore,

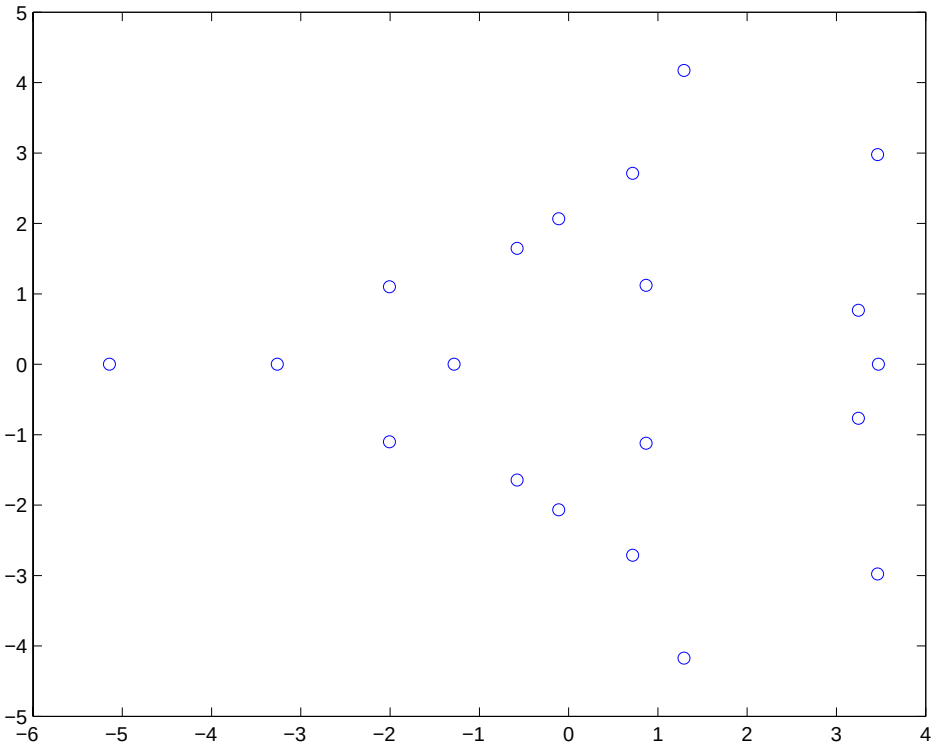
```
plot(Z)
```

where `Z` is a complex vector or matrix, is equivalent to

```
plot(real(Z),imag(Z))
```

For example, this statement plots the distribution of the eigenvalues of a random matrix using circular markers to indicate the data points.

```
plot(eig(randn(20,20)),'o','MarkerSize',6)
```



To plot more than one complex matrix, there is no shortcut; the real and imaginary parts must be taken explicitly.

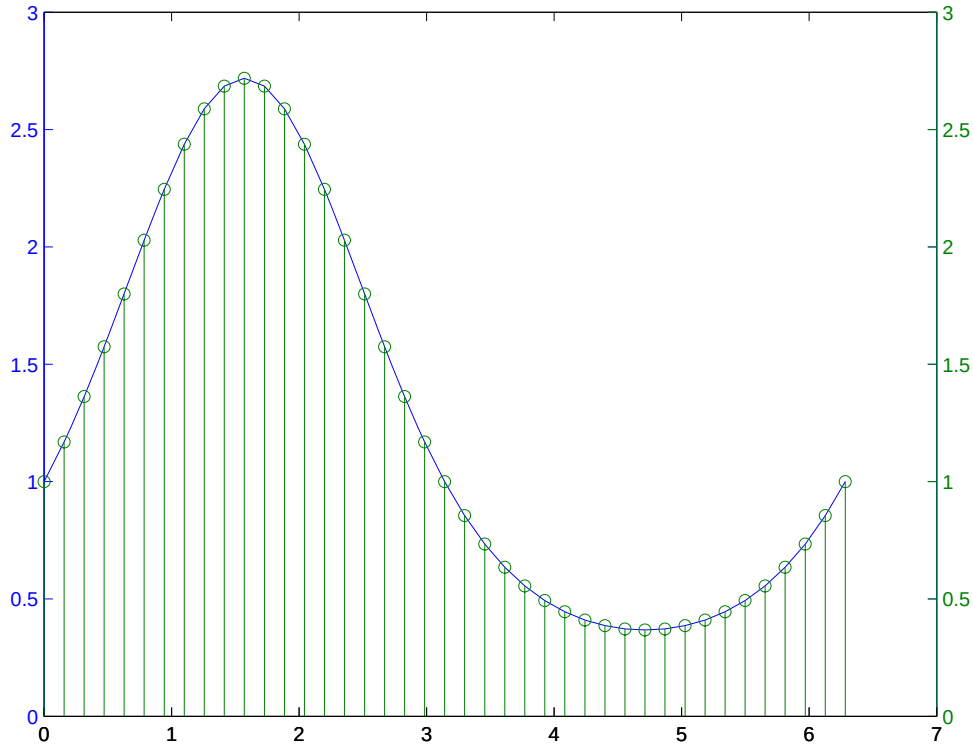
Plotting with Two Y-Axes

In this section...

Introduction

The `plotyy` function enables you to create plots of two data sets and use both left and right side *y*-axes. You can also apply different plotting functions to each data set. For example, you can combine a line plot with a stem plot of the same data.

```
t = 0:pi/20:2*pi;
y = exp(sin(t));
plotyy(t,y,t,y,'plot','stem')
```



Combining Linear and Logarithmic Axes

You can use `plotyy` to apply linear and logarithmic scaling to compare two data sets having different ranges of values.

```
t = 0:900; A = 1000; a = 0.005; b = 0.005;
z1 = A*exp(-a*t);
z2 = sin(b*t);
[haxes,hline1,hline2] = plotyy(t,z1,t,z2,'semilogy','plot');
```

This example saves the handles of the lines and axes created to adjust and label the graph. First, label the axes whose *y* value ranges from 10 to 1000.

This is the first handle in `haxes` because it was specified first in the call to `plotyy`. Use the `axes` function to make `haxes(1)` the current axes, which is then the target for the `ylabel` function.

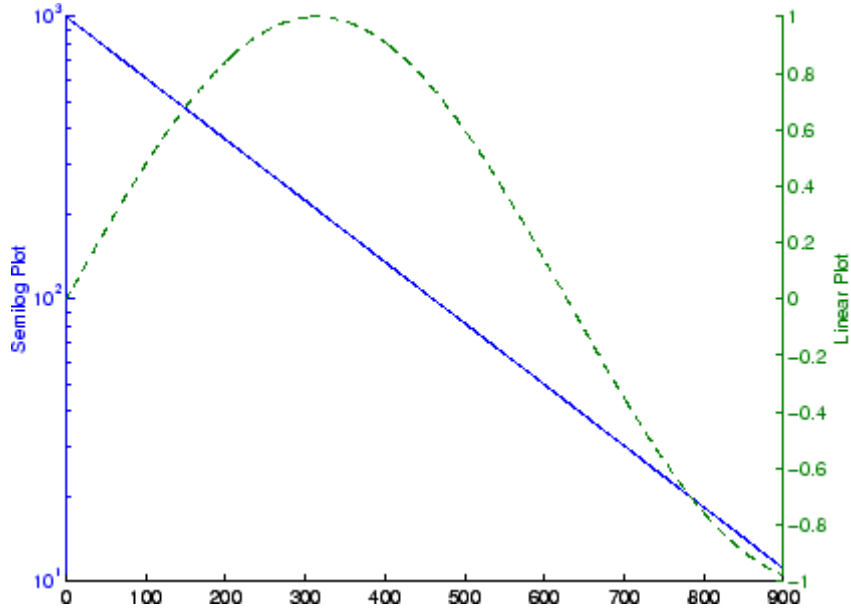
```
axes(haxes(1))
ylabel('Semilog Plot')
```

Now make the second axes current and call `ylabel` again.

```
axes(haxes(2))
ylabel('Linear Plot')
```

You can modify the characteristics of the plotted lines in a similar way. For example, to change the line style of the second line plotted to a dashed line, use the statement

```
set(hline2,'LineStyle','--')
```



See “Using Multiple X- and Y-Axes” on page 10-25 for an example that employs double *x*- and *y*-axes.

See `LineSpec` for additional line properties.

Setting Axis Parameters

In this section...

Axis Scaling and Ticks

When you create a graph, MATLAB automatically selects the axis limits and tick-mark spacing based on the data plotted. However, you can also specify your own values for axis limits and tick marks with the following functions:

- `axis` — Sets values that affect the current axes object (the most recently created or the last clicked on).
- `axes` — (Not axis) creates a new axes object with the specified characteristics.
- `get` and `set` — Enable you to query and set a wide variety of properties of existing axes.
- `gca` — Returns the handle (identifier) of the current axes. If there are multiple axes in the figure window, the current axes is the last graph created or the last graph you clicked on with the mouse. The following two sections provide more information and examples:

See “Defining the View” in the 3-D Visualization documentation for more extensive information on manipulating 3-D views.

Axis Limits and Ticks

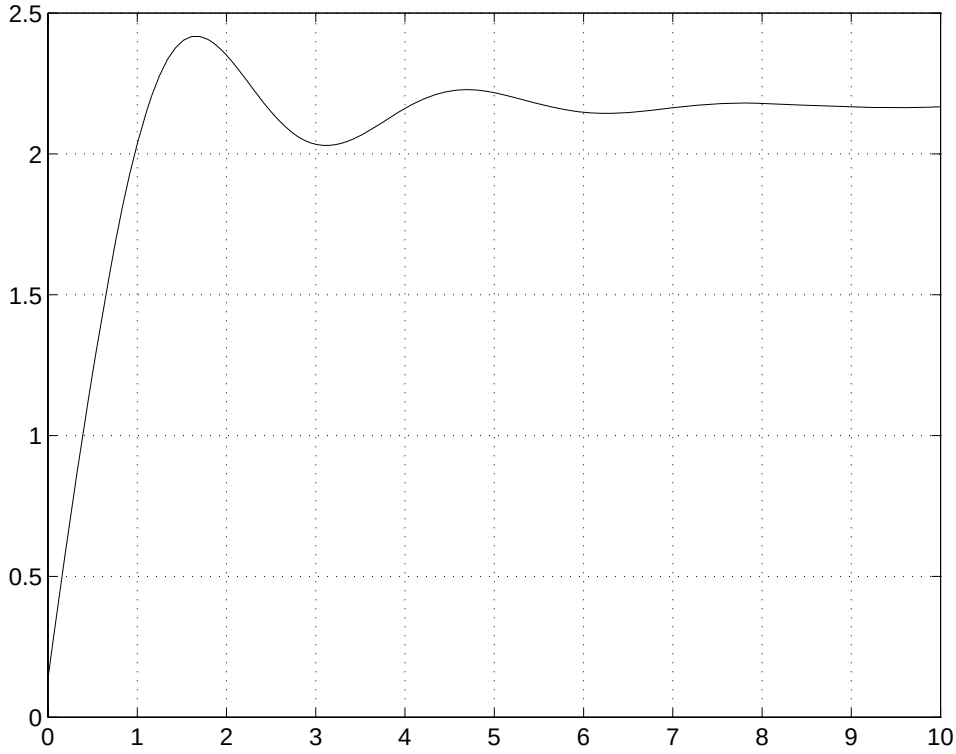
MATLAB selects axis limits based on the range of the plotted data. You can specify the limits manually using the `axis` function. Call `axis` with the new limits defined as a four-element vector.

```
axis([xmin,xmax,ymin,ymax])
```

Note that the minimum values must be less than the maximum values.

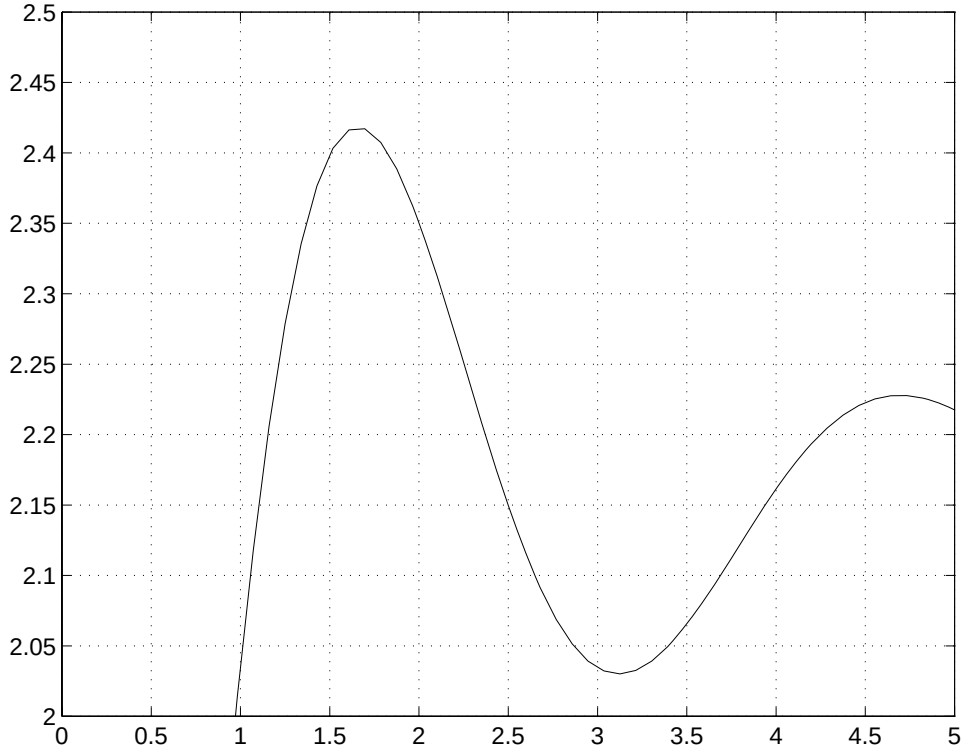
Semiautomatic Limits

If you want MATLAB to autoscale only one of a min/max set of axis limits, but you want to specify the other, use the MATLAB variable `Inf` or `-Inf` for the autoscaled limit. For example, this graph uses default scaling.



Compare the default limits to the following graph, which sets the maximum limit of the *x*-axis, but autoscales the minimum limit.

```
axis([-Inf 5 2 2.5])
```



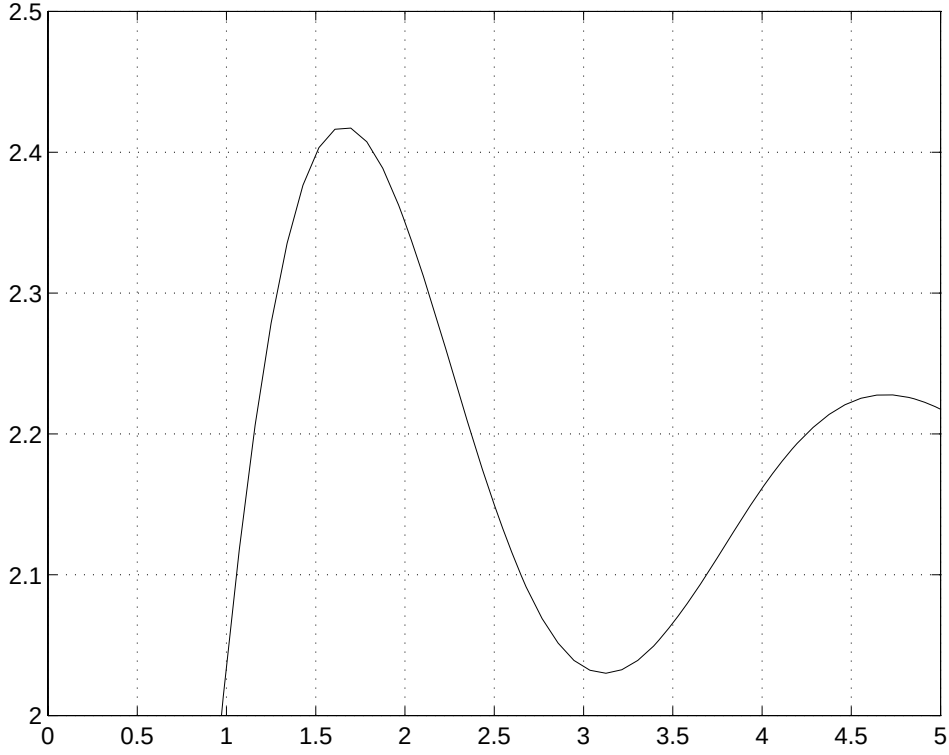
Axis Tick Marks

MATLAB selects the tick-mark locations based on the range of data so as to produce equally spaced ticks (for linear graphs). You can specify different tick marks by setting the axes `XTick` and `YTick` properties. Define tick marks as a vector of increasing values. The values do not need to be equally spaced.

For example, setting the *y*-axis tick marks for the graph from the preceding example,

```
set(gca,'ytick',[2 2.1 2.2 2.3 2.4 2.5])
```

produces a graph with only the specified ticks on the *y*-axis.



Note that if you specify tick-mark values that are outside the axis limits, MATLAB does not display them (that is, specifying tick marks cannot cause axis limits to change).

Example — Specifying Ticks and Tick Labels

You can adjust the axis tick-mark locations and the labels appearing at each tick mark. For example, this plot of the sine function relabels the *x*-axis with more meaningful values.

```
x = -pi:.1:pi;
y = sin(x);
plot(x,y)
set(gca,'XTick',-pi:pi/2:pi)
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

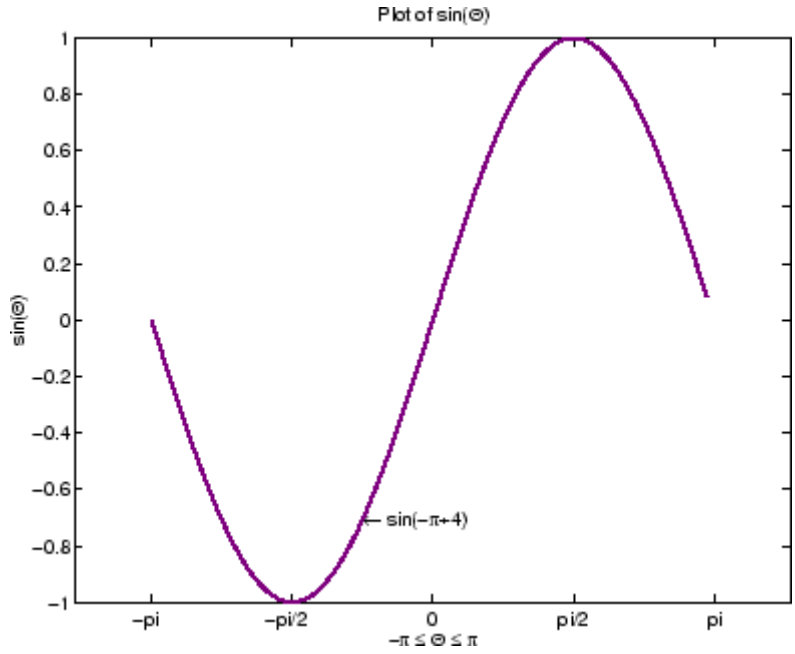
These functions (`xlabel`, `ylabel`, `title`, `text`) add axis labels and draw an arrow that points to the location on the graph where y = sin(-pi/4).

```
xlabel('-\pi \leq \Theta \leq \pi')
ylabel('sin(\Theta)')
title('Plot of sin(\Theta)')
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
     'HorizontalAlignment','left')
```

Setting Line Properties on an Existing Plot

Change the line color to purple by first finding the handle of the line object created by `plot` and then setting its `Color` property. Use `findobj` and the fact that MATLAB creates a blue line (RGB value [0 0 1]) by default. In the same statement, set the `LineWidth` property to 2 points.

```
set(findobj(gca,'Type','line','Color',[0 0 1]),...
    'Color',[0.5,0,0.5],'LineWidth',2)
```



The Greek symbols are created using TeX character sequences.

Setting Aspect Ratio

By default, MATLAB displays graphs in a rectangular axes that has the same aspect ratio as the figure window. This makes optimum use of space available for plotting. MATLAB provides control over the aspect ratio with the `axis` function.

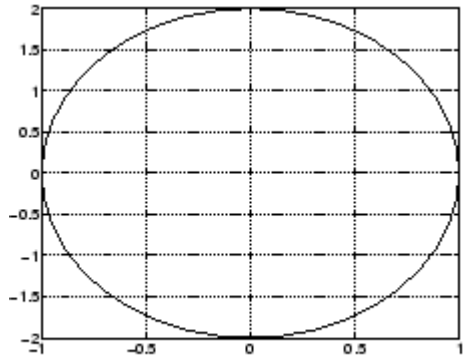
For example,

```
t = 0:pi/20:2*pi;
plot(sin(t),2*cos(t))
grid on
```

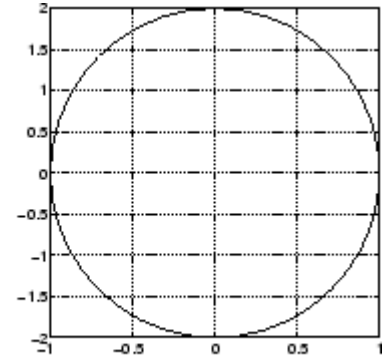
produces a graph with the default aspect ratio. The command

```
axis square
```

makes the *x*- and *y*-axes equal in length.



axis normal

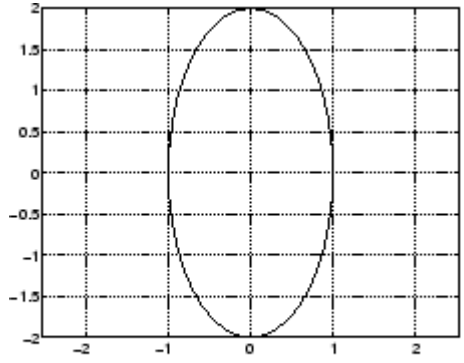


axis square

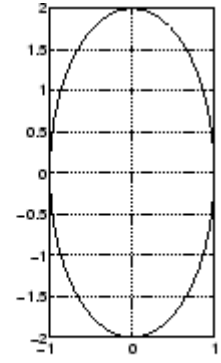
The square axes has one data unit in *x* to equal two data units in *y*. If you want the *x*- and *y*-data units to be equal, use the command

```
axis equal
```

This produces an axes that is rectangular in shape, but has equal scaling along each axis.



axis equal



axis equal tight

If you want the axes shape to conform to the plotted data, use the `tight` option in conjunction with `equal`.

```
axis equal tight
```

Note In order to format aspect ratio using `axis`, axes must exist and contain a plot. That is, you cannot pre-format an axes that has no actual *x*-, *y*-, or *z*-limits. To overcome this, you can preformat the axes with `axis` and issue the `hold on` command before plotting data.

5

Creating Specialized Plots

Bar and Area Graphs

In this section...

Types of Bar Graphs

Bar and area graphs display vector or matrix data. These types of graphs are useful for viewing results over a period of time, comparing results from different data sets, and showing how individual elements contribute to an aggregate amount. Bar graphs are suitable for displaying discrete data, whereas area graphs—like line graphs—are more suitable for displaying continuous data. The functions that plot bar and area graphs are listed below:

Function	Description
`bar`	Displays columns of *m*-by-*n* matrix as *m* groups of *n* vertical bars
`barh`	Displays columns of *m*-by-*n* matrix as *m* groups of *n* horizontal bars
`bar3`	Displays columns of *m*-by-*n* matrix as *m* groups of *n* vertical 3–D bars
`bar3h`	Displays columns of *m*-by-*n* matrix as *m* groups of *n* horizontal 3–D bars
`area`	Displays vector data as stacked area plots

Four of these five functions display bar graphs (there is only one type of area graph; see “Area Graphs” on page 5-19). Bar graphs differ according to whether they plot in 2-D or 3-D and create vertical or horizontal bars, as follows:

Orientation	Two–Dimensional	Three–Dimensional
Vertical	`bar`	`bar3`
Horizontal	`barh`	`bar3h`

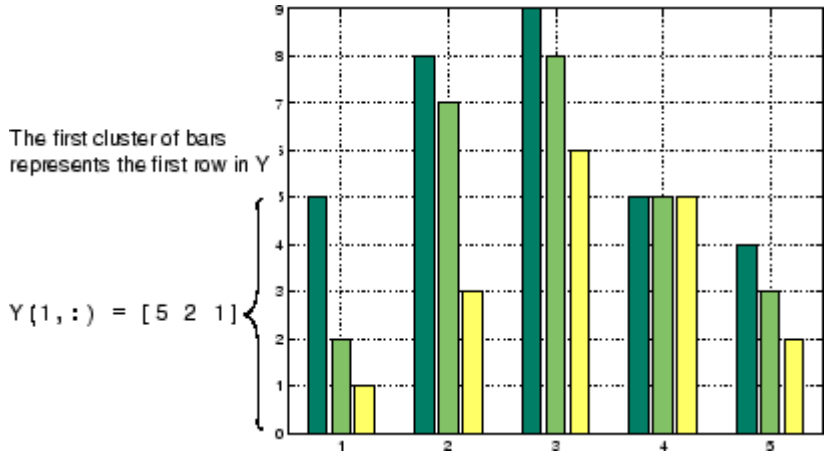
Grouped Bar Graph

By default, a bar graph represents each element in a matrix as one bar. Bars in a 2–D bar graph, created by the `bar` function, are distributed along the *x*-axis with each element in a column drawn at a different location. All elements in a row are clustered around the same location on the *x*-axis.

For example, define `Y` as a simple matrix and issue the `bar` function in its simplest form.

```
Y = [5 2 1
     8 7 3
     9 8 6
     5 5 5
     4 3 2];
bar(Y)
```

The bars are clustered together by rows and evenly distributed along the *x*-axis.

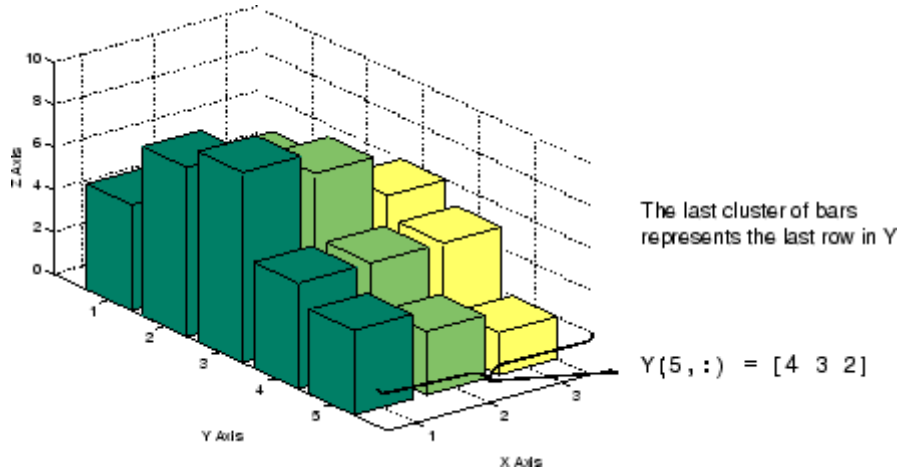


Detached 3-D Bars

The `bar3` function, in its simplest form, draws each element as a separate 3-D block, with the elements of each column distributed along the *y*-axis. Bars that represent elements in the first column of the matrix are centered at `1` along the *x*-axis. Bars that represent elements in the last column of the matrix are centered at `size(Y,2)` along the *x*-axis. For example,

```
bar3(Y)
```

displays five groups of three bars along the *y*-axis. Notice that larger bars obscure `Y(1,2)` and `Y(1,3)`.



By default, `bar3` draws detached bars. The statement `bar3(Y,'detach')` has the same effect.

Labeling the Graph. To add axes labels and *x* tick marks to this bar graph, use the statements

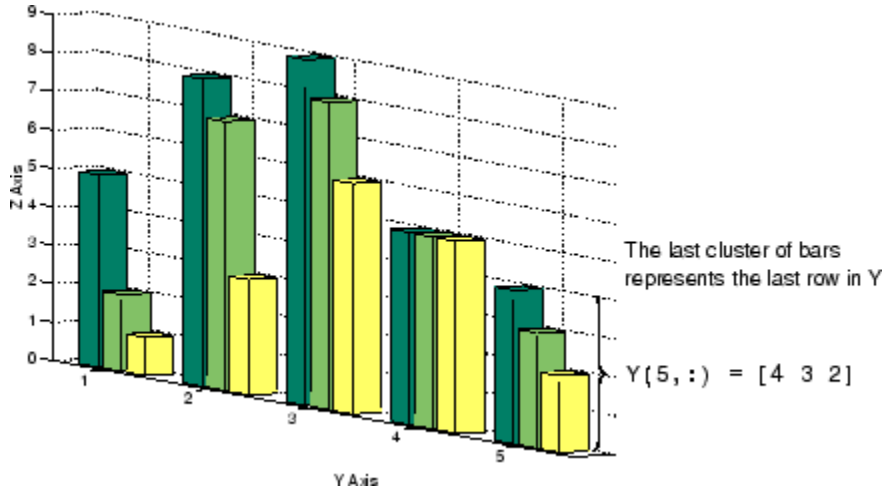
```
xlabel('X Axis')
ylabel('Y Axis')
zlabel('Z Axis')
set(gca,'XTick',[1 2 3])
```

Grouped 3-D Bars

Cluster the bars from each row beside each other by specifying the argument `'group'`. For example,

```
bar3(Y,'group')
```

groups the bars according to row and distributes the clusters evenly along the *y*-axis.

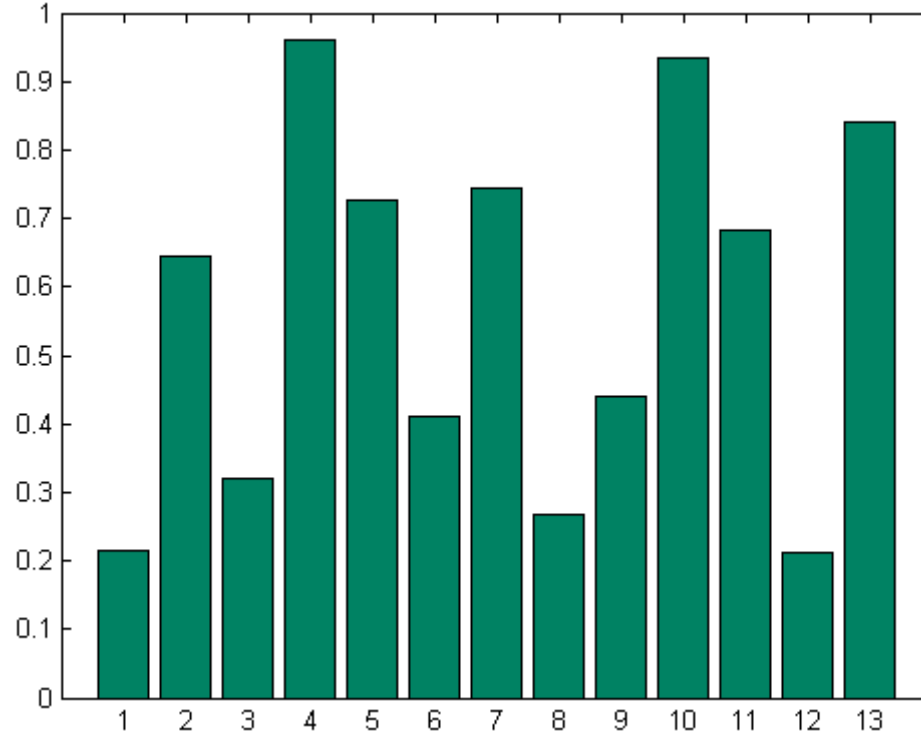


Coloring 2-D Bars According to Height

The `bar` and `barh` functions make all bars in a series the same color. With a little effort, however, you can assign a desired color to each bar. The typical approach is to associate bar colors with bar heights (Y-values). The following steps describe one way to do this, first using faceted shading and then using smooth (interpolated) shading:

1. Make up some numbers, plot a default bar plot, and assign a bichromatic colormap:

```
n = 13;
Z = rand(n,1);
h = bar(Z);
colormap(summer(n));
```



Notice that only the first color is used to color the faces.

2 Assign a new color to each bar. `bar` (and `barh`) creates a barseries object, which encapsulates a set of patch objects for the bars. The patches have face-vertex syntax. You must first get a handle for the children, and then obtain the vertices for the bars and the vertex color data:

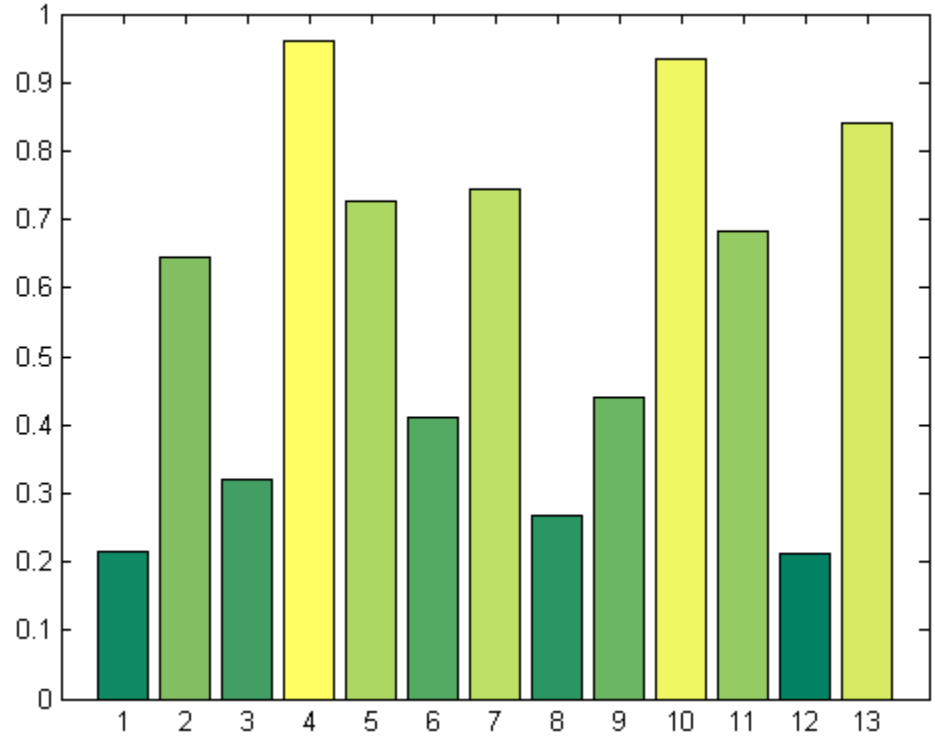
```
ch = get(h,'Children');
fvd = get(ch,'Faces');
fvcd = get(ch,'FaceVertexCData');
```

3 Sort the data to obtain an index for traversing the Faces array from the lowest to highest bar:

```
[zs, izs] = sortrows(Z,1);
```

4 Traverse the Faces and assign colors to the face-vertex color data as you go:

```
for i = 1:n
    row = izs(i);
    fvcd(fvd(row,:)) = i;
end
set(ch,'FaceVertexCData',fvcd)
```

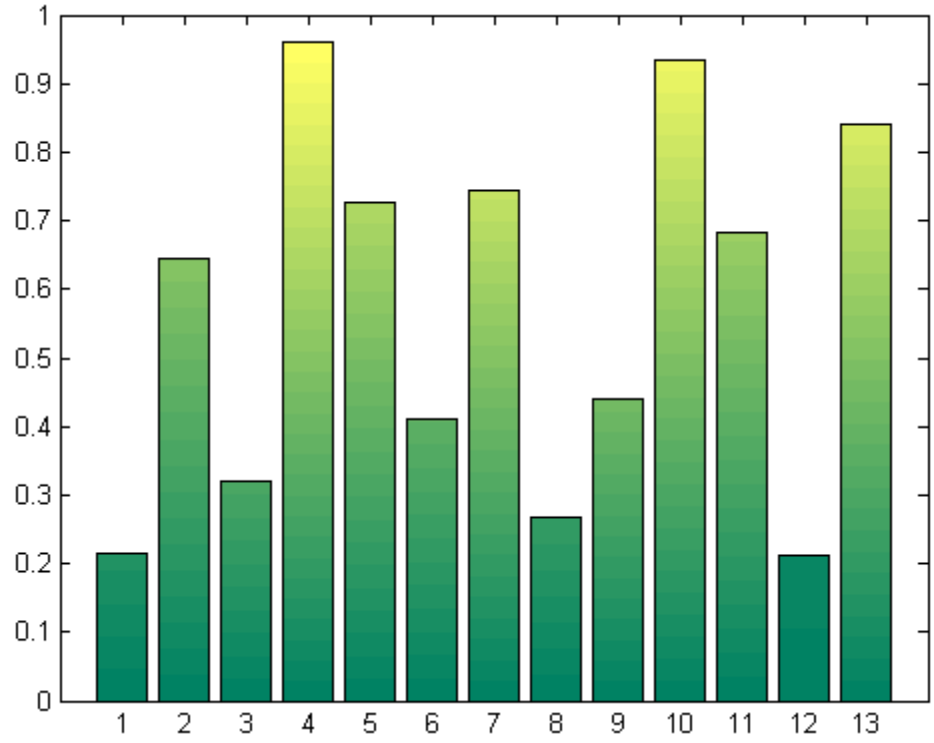


Note that the code assigns colors to bars based on their `YData` ranks, rather than by their `YData` values. This helps to distinguish bars by color, but also can assign to bars that are nearly the same height a wider range of colors than if the colors were directly mapped to `YData` values.

5 To make the graph more readable, you can set different colors for vertices on the baseline and on the top, then apply interpolated shading to change hue going up the bars. The following code colors the two vertices at the

base of each bar using the first color in the colormap, and assigns a color to the two vertices at the top proportionally to bar height. A longer color ramp than was used above is needed to obtain smooth gradiations of shading:

```
k = 128;                % Number of colors in color table
colormap(summer(k));    % Expand the previous colormap
shading interp          % Needed to graduate colors
for i = 1:n
    color = floor(k*i/n);       % Interpolate a color index
    row = izs(i);               % Look up actual row # in data
    fvcd(fvd(row,1)) = 1;       % Color base vertices 1st index
    fvcd(fvd(row,4)) = 1;
    fvcd(fvd(row,2)) = color;   % Assign top vertices color
    fvcd(fvd(row,3)) = color;
end
set(ch,'FaceVertexCData', fvcd);  % Apply the vertex coloring
set(ch,'EdgeColor','k')           % Give bars black borders
```



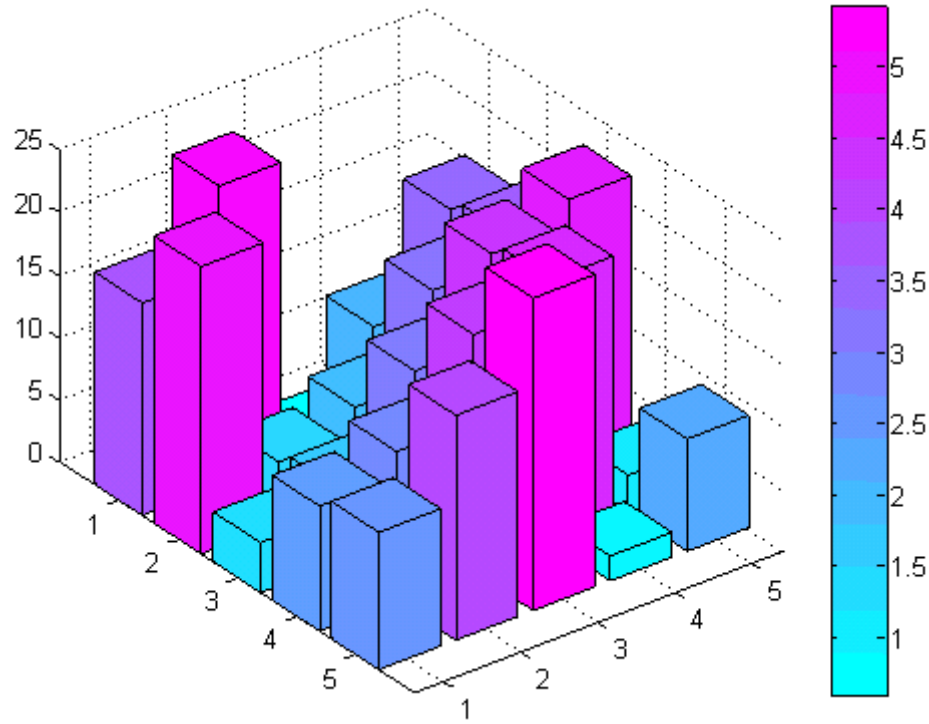
Coloring 3-D Bars According to Height

By default, all bars in a series (column) have the same color. You can modify a 3-D bar plot to color each bar according to how tall it is, but the technique is slightly different than the one used for coloring 2-D bars, described above. Applying a monochromatic or bichromatic colormap to such plots helps viewers see height distinctions more readily. Adding a colorbar can also help.

The graph will read better if you override the default behavior of `bar3` to shade the sides of the bars with contrasting hues. You can color bars by height and make the sides match the color of the top of each bar by executing the following code:

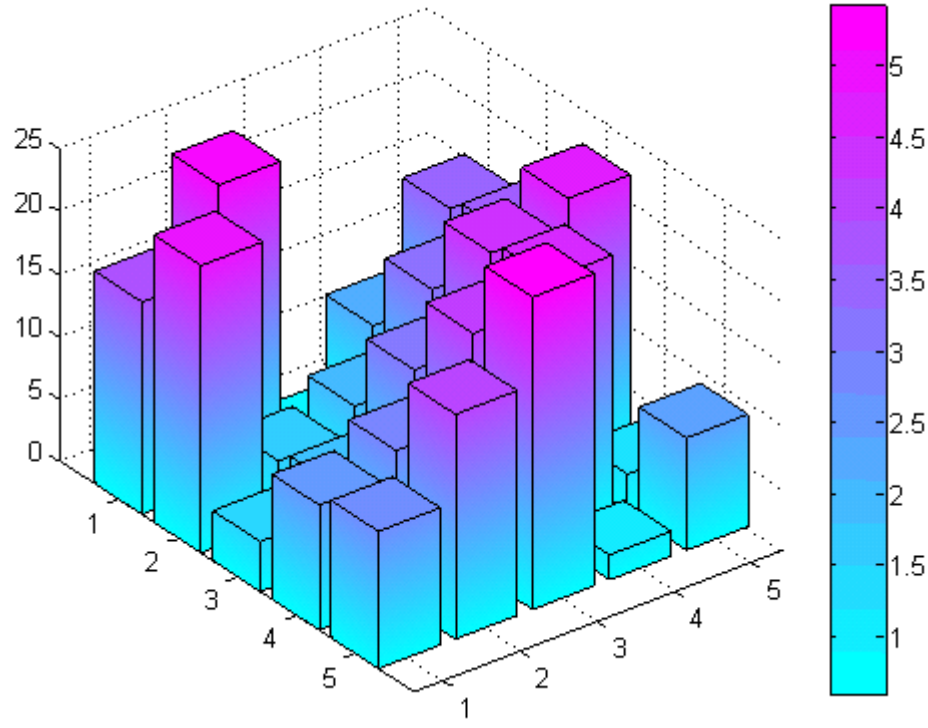
```
Z = magic(5);
h = bar3(Z);
```

```
for i = 1:length(h)
    zdata = ones(6*length(h),4);
    k = 1;
    for j = 0:6:(6*length(h)-6)
        zdata(j+1:j+6,:) = Z(k,i);
        k = k+1;
    end
    set(h(i),'Cdata',zdata)
end
colormap cool
colorbar
```



You can then make the plot even more readable by interpolating colors along the bars and giving their `EdgeColor` a contrasting color. The following code accomplishes this:

```
shading interp
for i = 1:length(h)
    zdata = get(h(i),'Zdata');
    set(h(i),'Cdata',zdata)
    set(h,'EdgeColor','k')
end
```



Stacked Bar Graphs to Show Contributing Amounts

Bar graphs can show how elements in the same row of a matrix contribute to the sum of all elements in the row. These types of bar graphs are referred to as stacked bar graphs.

Stacked bar graphs display one bar per row of a matrix. The bars are divided into *n* segments, where *n* is the number of columns in the matrix. For vertical

bar graphs, the height of each bar equals the sum of the elements in the row. Each segment is equal to the value of its respective element.

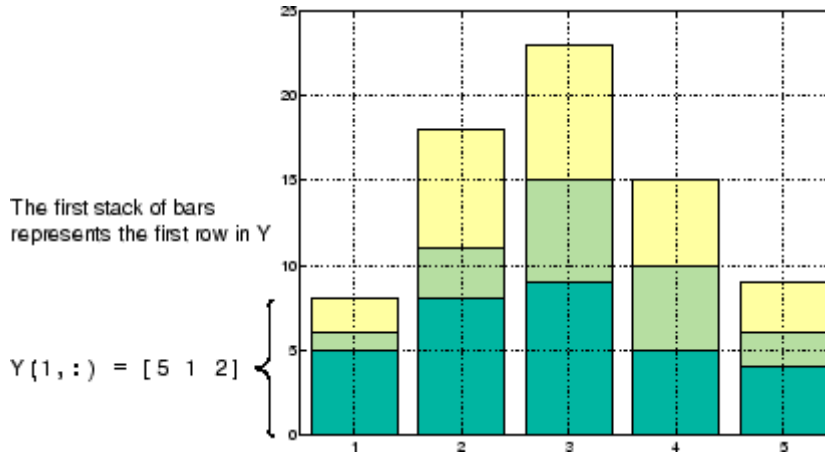
Redefining Y

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

Create stacked bar graphs using the optional `'stack'` argument. For example,

```
bar(Y,'stack')
grid on
set(gca,'Layer','top') % display gridlines on top of graph
```

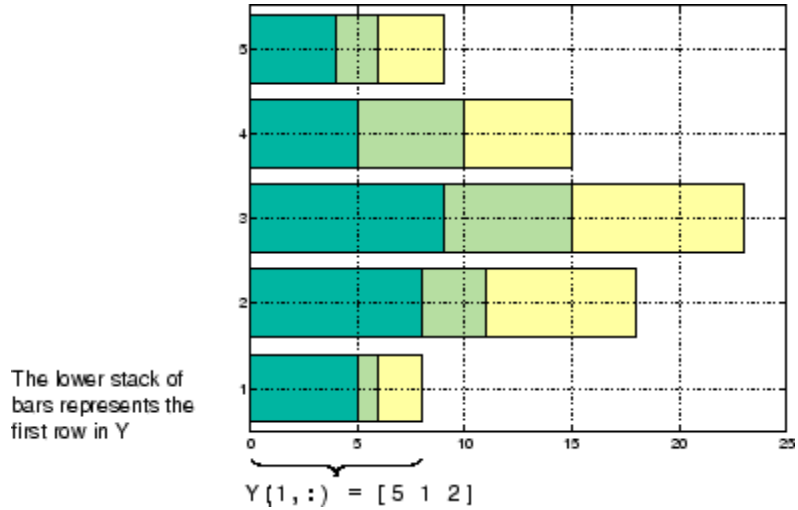
creates a 2-D stacked bar graph, where all elements in a row correspond to the same *x* location.



Horizontal Bar Graphs

For horizontal bar graphs, the length of each bar equals the sum of the elements in the row. The length of each segment is equal to the value of its respective element.

```
barh(Y,'stack')
grid on
set(gca,'Layer','top') % Display gridlines on top of graph
```



Specifying X-Axis Data

Bar graphs automatically generate *x*-axis values and label the *x*-axis tick lines. You can specify a vector of *x* values (or *y* values in the case of horizontal bar graphs) to label the axes.

For example, given temperature data,

```
temp = [29 23 27 25 20 23 23 27];
```

obtained from samples taken every five days during a thirty-five day period,

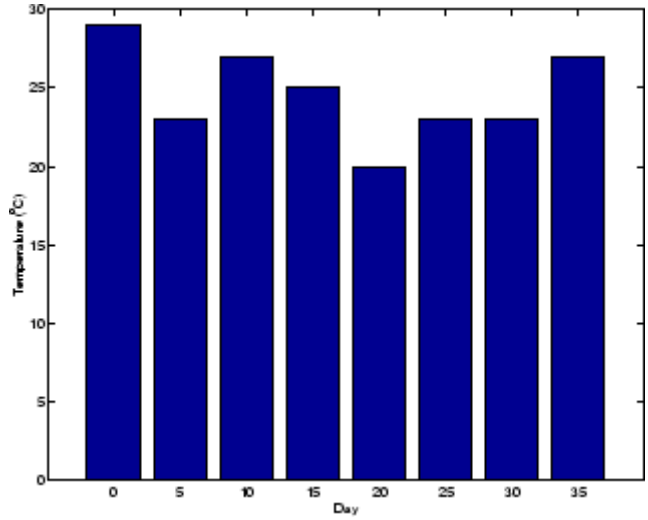
```
days = 0:5:35;
```

you can display a bar graph showing temperature measured along the *y*-axis and days along the *x*-axis using

```
bar(days,temp)
```

These statements add labels to the *x*- and *y*-axis.

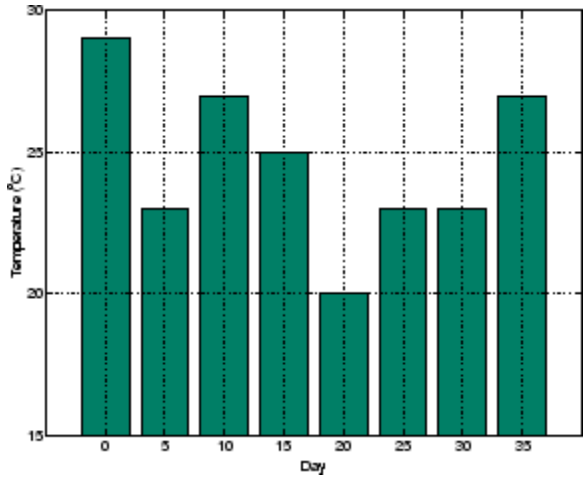
```
xlabel('Day')
ylabel('Temperature (^{o}C)')
```



Setting Y-Axis Limits

By default, the *y*-axis range is from 0 to 30. To focus on the temperature range from 15 to 30, change the *y*-axis limits.

```
set(gca,'YLim',[15 30],'Layer','top')
```



Overlaying Bar Graphs

In addition to grouping and stacking barseries, you can overlay several bars that share the same baseline and y-range by making each series of bars a different width and plotting the widest ones first. The following example shows how to accomplish this within an axes:

1 Define x and y data; it probably helps to make spacing of x-values constant:

```
x=[1 3 5 7 9];
y1=[10 25 90 35 16];
K=0.5;
```

2 Plot Series 1 in red, and set bar width to one-half an x-unit:

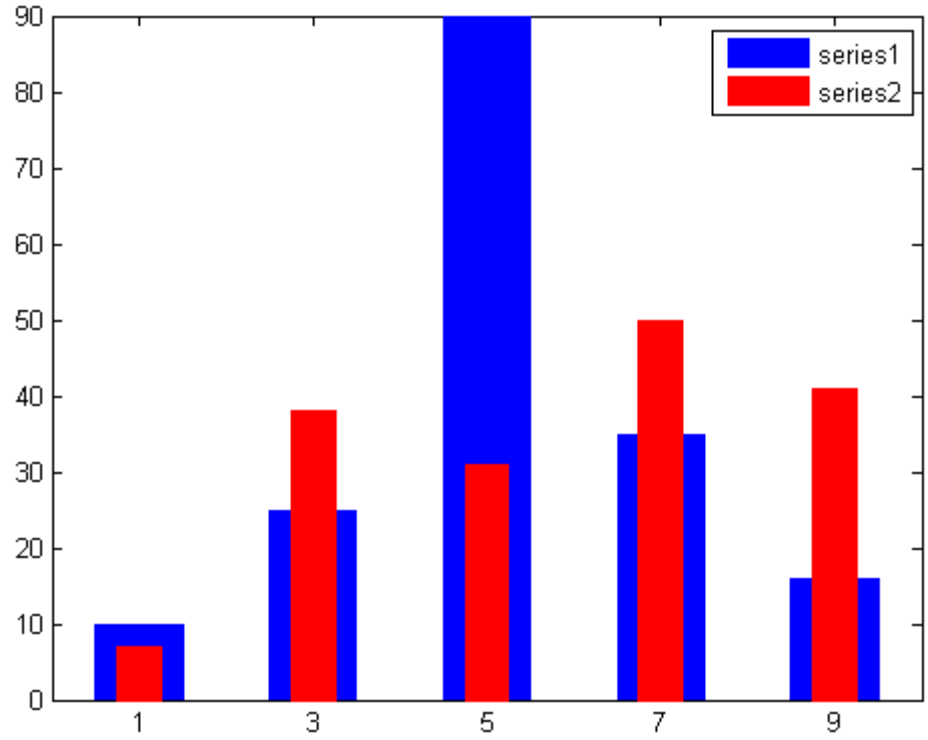
```
bar1=bar(x, y1, 'FaceColor', 'b', 'EdgeColor', 'b');
set(bar1,'BarWidth',K);
```

3 Define Series 2, and plot it in blue over the first one:

```
hold on;
y2=[7 38 31 50 41];
bar2=bar(x, y2, 'FaceColor', 'r', 'EdgeColor', 'r');
```

4 Set the width of the second series to half that of the first one:

```
set(bar2,'BarWidth',K/2);
hold off;
legend('series1','series2')
```



Overlaying Other Plots on Bar Graphs

You can overlay data on a bar graph by creating another axes in the same position. This enables you to have an independent *y*-axis for the overlaid dataset (in contrast to the `hold on` statement, which uses the same axes).

For example, consider a bioremediation experiment that breaks down hazardous waste components into nontoxic materials. The trichloroethylene (TCE) concentration and temperature data from this experiment are

```
TCE = [515 420 370 250 135 120 60 20];
temp = [29 23 27 25 20 23 23 27];
```

This data was obtained from samples taken every five days during a thirty-five day period.

```
days = 0:5:35;
```

Display a bar graph and label the *x*- and *y*-axis using the statements

```
bar(days,temp)
xlabel('Day')
ylabel('Temperature (^{o}C)')
```

Overlaying a Line Plot on the Bar Graph

To overlay the concentration data on the bar graph, position a second axes at the same location as the first axes, but first save the handle of the first axes.

```
h1 = gca;
```

Create the second axes at the same location before plotting the second dataset.

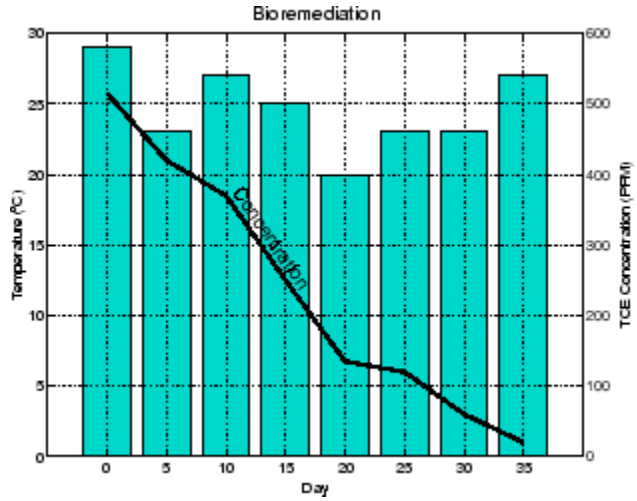
```
h2 = axes('Position',get(h1,'Position'));
plot(days,TCE,'LineWidth',3)
```

To ensure that the second axes does not interfere with the first, locate the *y*-axis on the right side of the axes, make the background transparent, and set the second axes' *x*-tick marks to the empty matrix.

```
set(h2,'YAxisLocation','right','Color','none','XTickLabel',[])
```

Align the *x*-axis of both axes and display the grid lines on top of the bars.

```
set(h2,'XLim',get(h1,'XLim'),'Layer','top')
```



Annotating the Graph. These statements annotate the graph.

```
text(11,380,'Concentration','Rotation',-55,'FontSize',16)
ylabel('TCE Concentration (PPM)')
title('Bioremediation','FontSize',16)
```

To print the graph, set the current figure's `PaperPositionMode` to `auto`, which ensures the printed output matches the display.

```
set(gcf,'PaperPositionMode','auto')
```

Area Graphs

The `area` function displays curves generated from a vector or from separate columns in a matrix. `area` plots the values in each column of a matrix as a separate curve and fills the area between the curve and the *x*-axis.

Area Graphs Showing Contributing Amounts

Area graphs are useful for showing how elements in a vector or matrix contribute to the sum of all elements at a particular *x* location. By default, `area` accumulates all values from each row in a matrix and creates a curve from those values.

Using this matrix,

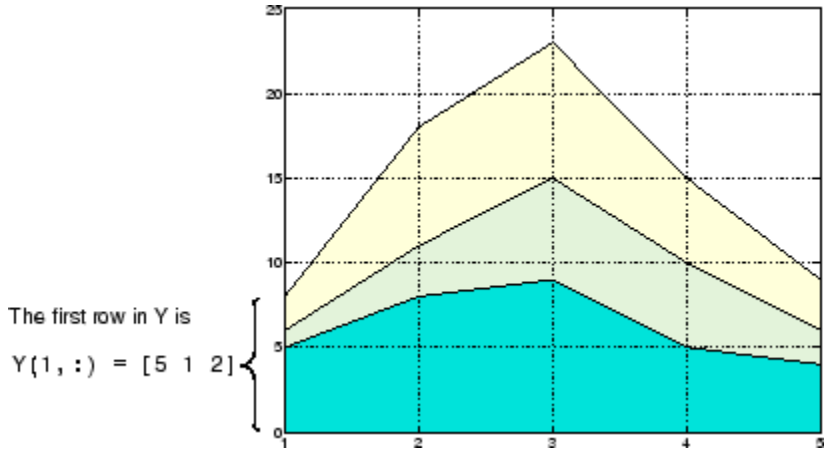
```
Y = [5 1 2
     8 3 7
     9 6 8
     5 5 5
     4 2 3];
```

the statement

```
area(Y)
```

displays a graph containing three area graphs, one per column.

The height of the area graph is the sum of the elements in each row. Each successive curve uses the preceding curve as its base.



Displaying the Grid on Top. To display the grid lines in the foreground of the area graph and display only five grid lines along the *x*-axis, use the statements

```
set(gca,'Layer','top')
set(gca,'XTick',1:5)
```

Comparing Data Sets with Area Graphs

Area graphs are useful for comparing different datasets. For example, given a vector containing sales figures,

```
sales = [51.6 82.4 90.8 59.1 47.0];
```

for the five-year period

```
x = 90:94;
```

and a vector containing profits figures for the same five-year period,

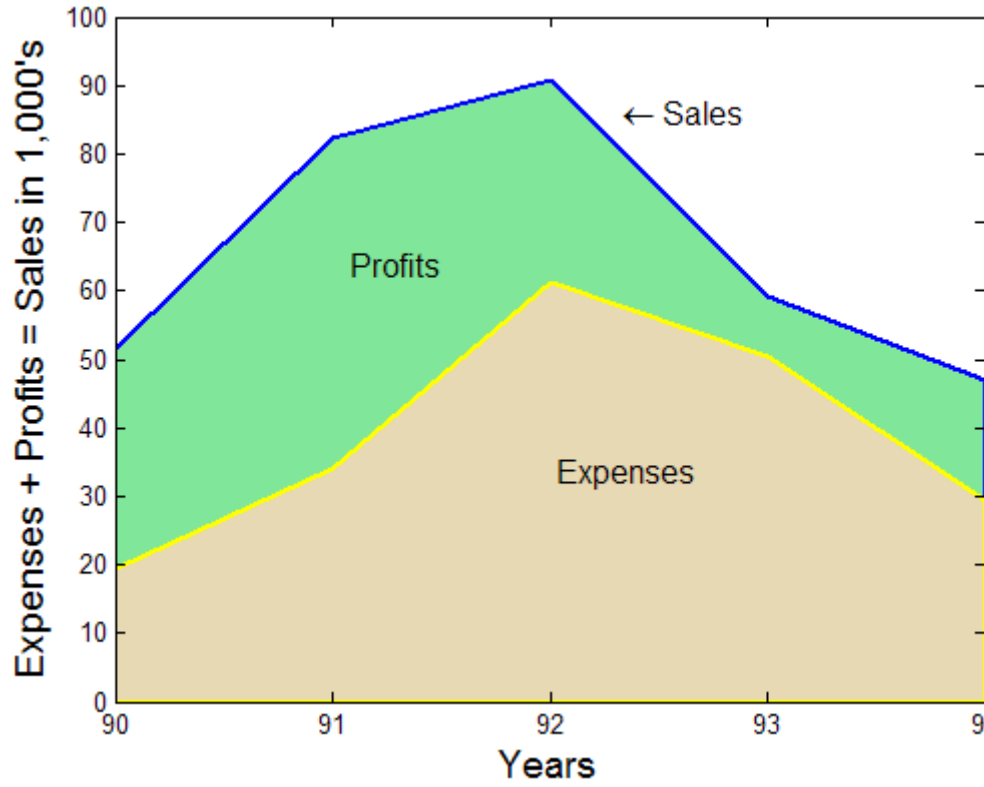
```
profits = [19.3 34.2 61.4 50.5 29.4];
```

display both as two separate area graphs within the same axes. Set the color of the area interior (`FaceColor`), its edges (`EdgeColor`), and the width of the edge lines (`LineWidth`). See `patch` for a complete list of properties.

```
area(x,sales,'FaceColor',[.5 .9 .6],...
             'EdgeColor','b',...
             'LineWidth',2)
hold on
area(x,profits,'FaceColor',[.9 .85 .7],...
               'EdgeColor','y',...
               'LineWidth',2)
hold off
```

To annotate the graph, use the statements

```
set(gca,'XTick',[90:94])
set(gca,'Layer','top')
gtext('\leftarrow Sales')
gtext('Profits')
gtext('Expenses')
xlabel('Years','FontSize',14)
ylabel('Expenses + Profits = Sales in 1,000''s','FontSize',14)
```


Expenses + Profits = Sales in 1,000's
100
90
80
70
60
50
40
30
20
10
0
90
91
92
93
94
Years
← Sales
Profits
Expenses

Pie Charts

In this section...

Creating a Pie Chart

Pie charts are a useful way to communicate the percentage that each element in a vector or matrix contributes to the sum of all elements. `pie` and `pie3` create 2-D and 3-D pie charts. A 3-D pie chart does not show any more or different information than a 2-D pie chart does; it simply adds depth to the presentation by plotting the chart on top of a cylindrical base.

This example shows how to use the `pie` function to visualize the contribution that three products make to total sales. Given a matrix `X` where each column of `X` contains yearly sales figures for a specific product over a five-year period,

```
X = [19.3 22.1 51.6;
     34.2 70.3 82.4;
     61.4 82.9 90.8;
     50.5 54.9 59.1;
     29.4 36.3 47.0];
```

sum each row in `X` to calculate total sales for each product over the five-year period.

```
x = sum(X);
```

You can offset the slice of the pie that makes the greatest contribution using the `explode` input argument. This argument is a vector of zero and nonzero values. Nonzero values offset the respective slice from the chart.

First, create a vector containing zeros.

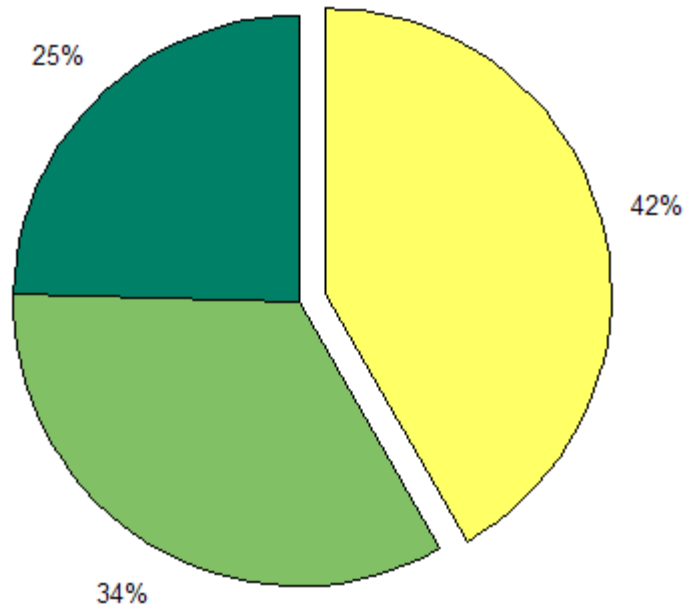
```
explode = zeros(size(x));
```

Then find the slice that contributes the most and set the corresponding `explode` element to `1`.

```
[c,offset] = max(x);
explode(offset) = 1;
```

The `explode` vector contains the elements `[0 0 1]`. To create the exploded pie chart, use the statement

```
h = pie(x,explode); colormap summer
```



Labeling the Pie Chart

The pie chart's labels are text graphics objects. To modify the text strings and their positions, first get the objects' strings and extents. Braces around a property name ensure that `get` outputs a cell array, which is important when working with multiple objects.

```
textObjs = findobj(h,'Type','text');
```

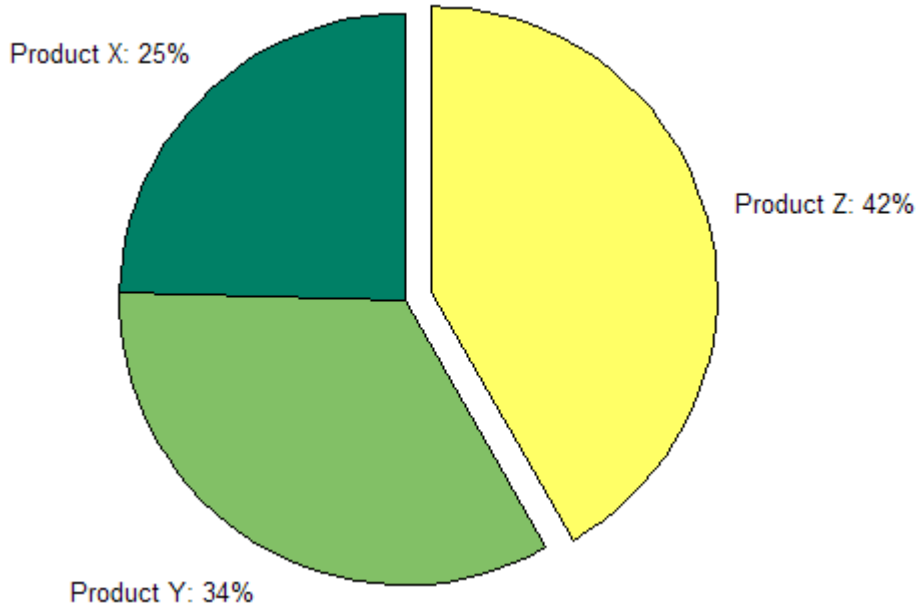
```
oldStr = get(textObjs,{'String'});
val = get(textObjs,{'Extent'});
oldExt = cat(1,val{:});
```

Create the new strings, then set the text objects' `String` properties to the new strings.

```
Names = {'Product X: ';'Product Y: ';'Product Z: '};
newStr = strcat(Names,oldStr);
set(textObjs,{'String'},newStr)
```

Find the difference between the widths of the new and old text strings and change the values of the `Position` properties.

```
val1 = get(textObjs, {'Extent'});
newExt = cat(1, val1{:});
offset = sign(oldExt(:,1)).*(newExt(:,3)-oldExt(:,3))/2;
pos = get(textObjs, {'Position'});
textPos =  cat(1, pos{:});
textPos(:,1) = textPos(:,1)+offset;
set(textObjs,{'Position'},num2cell(textPos,[3,2]))
```

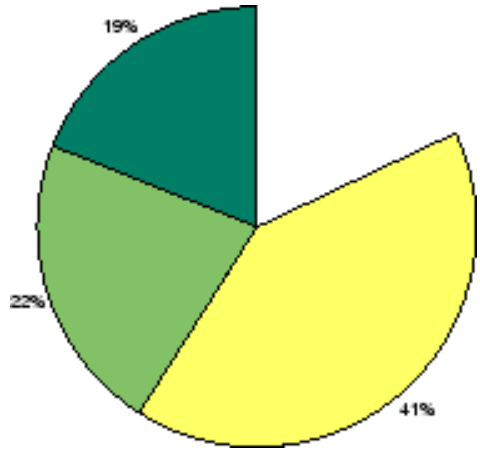


Removing a Piece from a Pie Chart

When the sum of the elements in the first input argument is equal to or greater than `1`, `pie` and `pie3` normalize the values. So, given a vector of elements x, each slice has an area of $x_i / sum(x_i)$, where x_i is an element of x. The normalized value specifies the fractional part of each pie slice.

When the sum of the elements in the first input argument is less than `1`, `pie` and `pie3` do not normalize the elements of vector `x`. They draw a partial pie. For example,

```
x = [.19 .22 .41];
pie(x)
```


19%
22%
41%

Histograms

In this section...

Functions for Creating Histograms

Histograms show the distribution of data values across a data range. They do this by dividing the data range into a certain number of intervals (called "binning" the data), tabulating the number of values that fall into each interval (or "bin"), and plotting the values in the bins using bars or wedges of varying height. The functions that create histograms are `hist` and `rose`.

Function	Description
`hist`	Displays data in a Cartesian coordinate system
`rose`	Displays data in a polar coordinate system

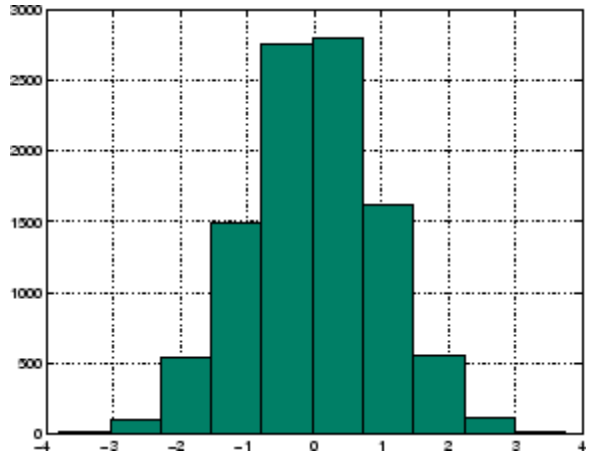
You can specify the number of bins to use as a scalar second argument. If omitted, the default is 10 (`hist`) or 20 (`rose`). Data values passed to `hist` can be in any units and can be n-by-m, but `rose` expects values to be in radians in a 1–by-n or n-by-1 vector. The height (or length when using `rose`) of the bins represents the number of values that fall in each bin. You can also vary the size of bins by specifying a vector for apportioning bin widths as the second argument.

Histograms in Cartesian Coordinates

The `hist` function shows the distribution of the elements in `Y` as a histogram with equally spaced bins between the minimum and maximum values in `Y`. If `Y` is a vector and is the only argument, `hist` creates up to 10 bins. For example,

```
yn = randn(10000,1);
hist(yn)
```

generates 10,000 random numbers and creates a histogram with 10 bins distributed along the *x*-axis between the minimum and maximum values of `yn`.

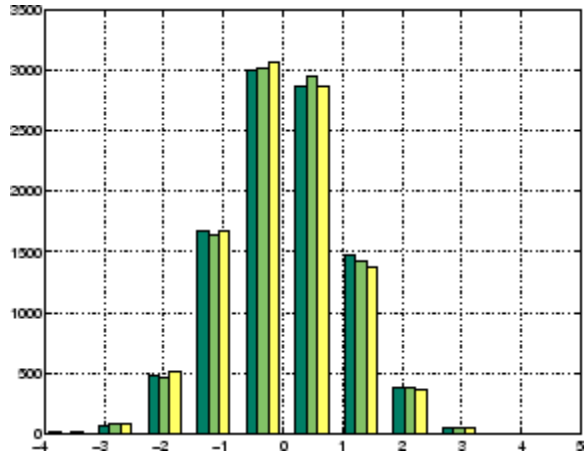


Matrix Input Argument

When `Y` is a matrix, `hist` creates a set of bins for each column, displaying each set in a separate color. The statements

```
Y = randn(10000,3);
hist(Y)
```

create a histogram showing 10 bins for each column in `Y`.



Histograms in Polar Coordinates

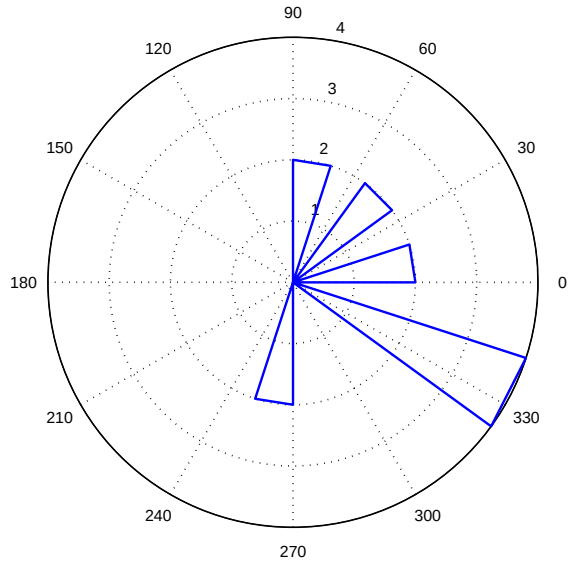
A `rose` plot is a histogram created in a polar coordinate system. For example, consider samples of the wind direction taken over a 12-hour period.

```
wdir = [45 90 90 45 360 335 360 270 335 270 335 335];
```

To display this data using the `rose` function, convert the data to radians, then use the data as an argument to the `rose` function. Increase the `LineWidth` property of the line to improve the visibility of the plot (`findobj`).

```
wdir = wdir * pi/180;
rose(wdir)
hline = findobj(gca,'Type','line');
set(hline,'LineWidth',1.5)
```

The plot shows that the wind direction was primarily 335° during the 12-hour period.



Specifying Number of Bins

`hist` and `rose` interpret their second argument in one of two ways — as the locations on the axis or the number of bins. When the second argument is a vector `x`, it specifies the locations on the axis and distributes the elements in `length(x)` bins. When the second argument is a scalar `x`, `hist` and `rose` distribute the elements in `x` bins.

For example, compare the distribution of data created by two MATLAB functions that generate random numbers. The `randn` function generates normally distributed random numbers, whereas the `rand` function generates uniformly distributed random numbers.

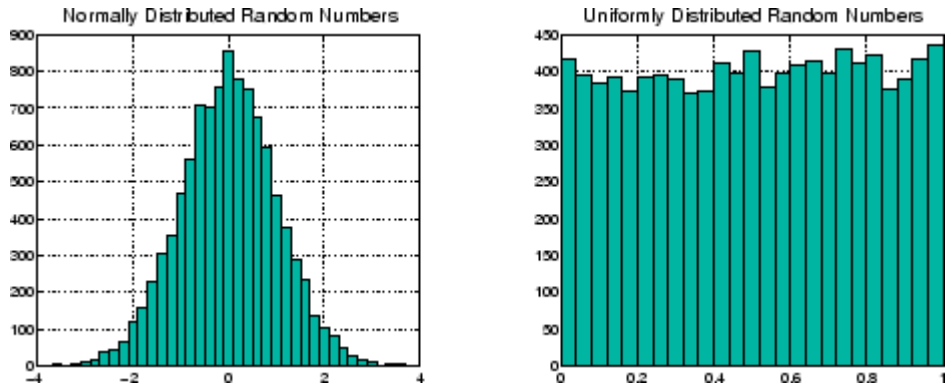
```
yn = randn(10000,1);
yu = rand(10000,1);
```

The first histogram displays the data distribution resulting from the `randn` function. The locations on the *x*-axis and number of bins depend on the vector `x`.

```
x = min(yn):.2:max(yn);
subplot(1,2,1)
hist(yn,x)
title('Normally Distributed Random Numbers','FontSize',16)
```

The second histogram displays the data distribution resulting from the `rand` function and explicitly creates 25 bins along the *x*-axis.

```
subplot(1,2,2)
hist(yu,25)
title('Uniformly Distributed Random Numbers','FontSize',16)
```



Note You can change the aspect ratio of the histogram plots using the mouse to resize the figure window. However, before creating hardcopy output, set the figure's `PaperPositionMode` to `auto` to produce printed output that matches the display. `set(gcf,'PaperPositionMode','auto')`

Discrete Data Graphs

In this section...

Functions for Creating Graphs of Discrete Data

In addition to bar graphs and pie charts, MATLAB has a number of specialized functions that are appropriate for displaying discrete data. Discrete data generally represents counts of things, such as traffic accidents by month or components produced or rejected during the course of a production run. This section describes how to use stem plots and stairstep plots to display this type of data. The functions for generating discrete data graphs provided in MATLAB are

Function	Description
`stem`	Displays a discrete sequence of *y*-data as stems from *x*-axis
`stem3`	Displays a discrete sequence of *z*-data as stems from *xy*-plane
`stairs`	Displays a discrete sequence of *y*-data as steps from *x*-axis

Two-Dimensional Stem Plots

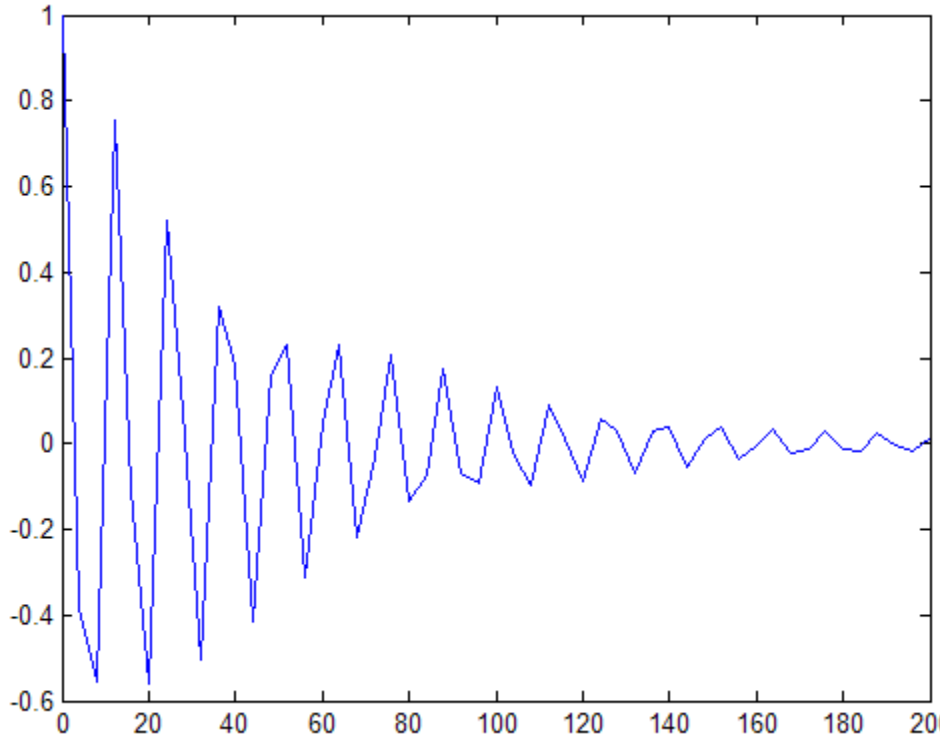
A stem plot displays data as lines (stems) terminated with a marker symbol at each data value. In a 2-D graph, stems extend from the *x*-axis.

The `stem` function displays two-dimensional discrete sequence data. For example, evaluating the function $y = e^{-\alpha t}\cos\beta t$ with the values

```
alpha = .02; beta = .5; t = 0:4:200;
y = exp(-alpha*t).*cos(beta*t);
```

yields a vector of discrete values for `y` at given values of `t`. A line plot shows the data points connected with a straight line.

```
plot(t,y)
```

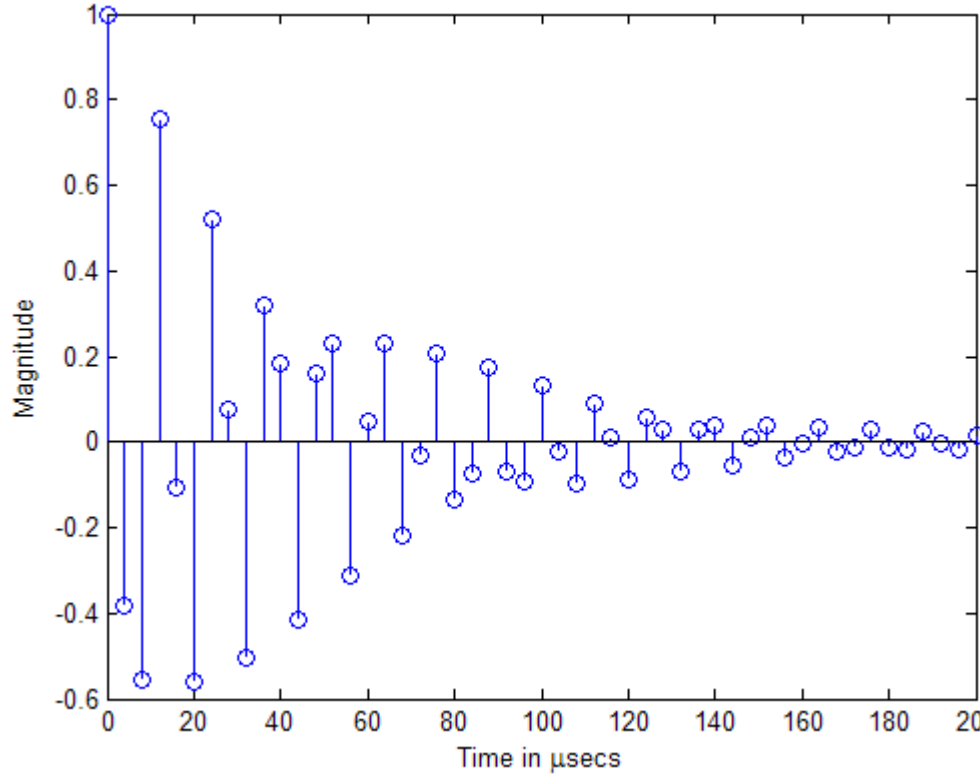


A stem plot of the same function plots only discrete points on the curve.

```
stem(t,y)
```

Add axes labels to the *x*- and *y*-axis.

```
xlabel('Time in \musecs')
ylabel('Magnitude')
```

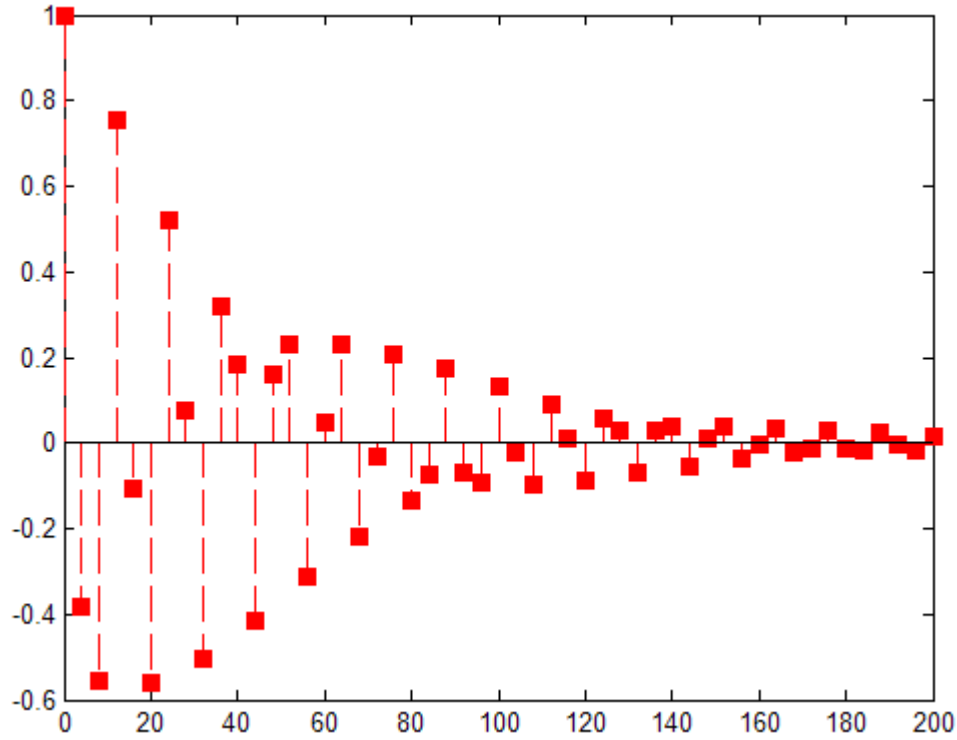


If you specify only one argument, the number of samples is equal to the length of that argument. In this example, the number of samples is a function of t, which contains 51 elements and determines the length of y.

Customizing the Graph

You can specify the line style, the type of marker, and the color used in the stem plot. For example, adding the string ':sr' specifies a dotted line (:), a square marker (s), and a red color (r). The `'fill'` argument colors the face of the marker.

```
stem(t,y,'--sr','fill')
```



Setting the aspect ratio of the *x*- and *y*-axis to 2:1 improves the utility of the graph. You can do this by setting the aspect ratio of the plot box using `pbaspect`.

```
pbaspect([2,1,1])
```

This is equivalent to setting the `PlotBoxApectRatio` property directly.

```
set(gca,'PlotBoxAspectRatio',[2,1,1])
```

See `LineSpec` for a list of line styles and marker types.

Combining Stem Plots with Line Plots

Sometimes it is useful to display more than one plot simultaneously with a stem plot to show how you arrived at a result. For example, create a linearly spaced vector with 60 elements and define two functions, `a` and `b`.

```
x = linspace(0,2*pi,60);
a = sin(x);
b = cos(x);
```

Create a stem plot showing the linear combination of the two functions.

```
stem_handles = stem(x,a+b);
```

Overlaying `a` and `b` as line plots helps visualize the functions. Before plotting the two curves, set `hold` to `on` so MATLAB does not clear the stem plot.

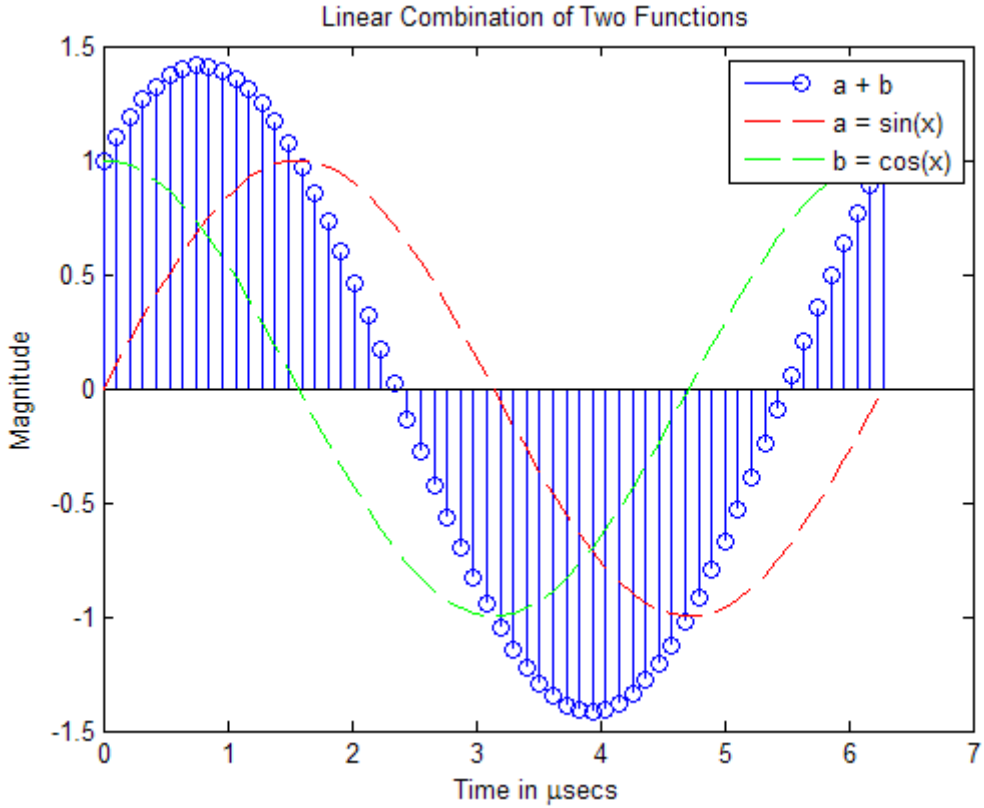
```
hold on
plot_handles = plot(x,a,'--r',x,b,'--g');
hold off
```

Use `legend` to annotate the graph. The stem and plot handles passed to `legend` identify the lines to label. Stem plots are composed of two lines; one draws the markers and the other draws the vertical stems. To create the legend, use the first handle returned by `stem`, which identifies the marker line.

```
legend_handles = [stem_handles(1);plot_handles];
legend(legend_handles,'a + b','a = sin(x)','b = cos(x)')
```

Labeling the axes and creating a title finishes the graph.

```
xlabel('Time in \musecs')
ylabel('Magnitude')
title('Linear Combination of Two Functions')
```



Three-Dimensional Stem Plots

`stem3` displays 3-D stem plots extending from the *xy*-plane. With only one vector argument, MATLAB plots the stems in one row at `x = 1` or `y = 1`, depending on whether the argument is a column or row vector. `stem3` is intended to display data that you cannot visualize in a 2-D view.

Example – 3-D Stem Plot of an FFT

Fast Fourier transforms are calculated at points around the unit circle on the complex plane. So, it is interesting to visualize the plot around the unit circle. Calculating the unit circle

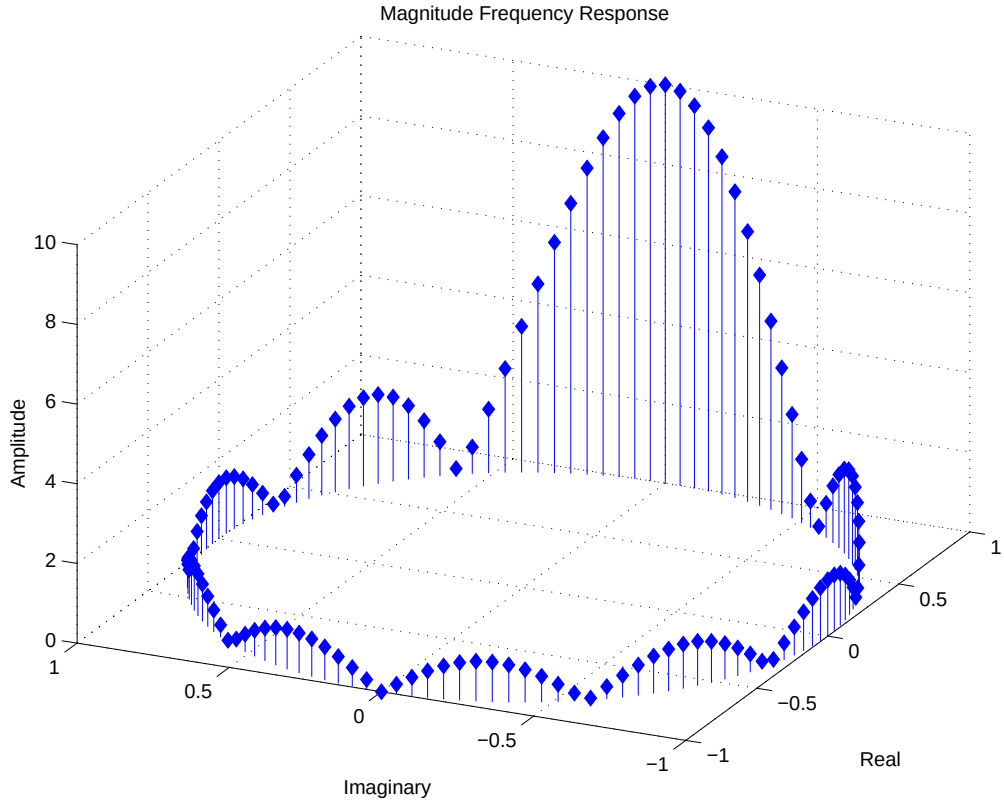
```
th = (0:127)/128*2*pi;
x = cos(th);
y = sin(th);
```

and the magnitude frequency response of a step function

```
f = abs(fft(ones(10,1),128));
```

displays the data using a 3-D stem plot, terminating the stems with filled diamond markers.

```
stem3(x,y,f','d','fill')
view([-65 30])
```



Label the Graph

Label the graph with the statements

```
xlabel('Real')
ylabel('Imaginary')
zlabel('Amplitude')
title('Magnitude Frequency Response')
```

To change the orientation of the view, turn on mouse-based 3-D rotation.

```
rotate3d on
```

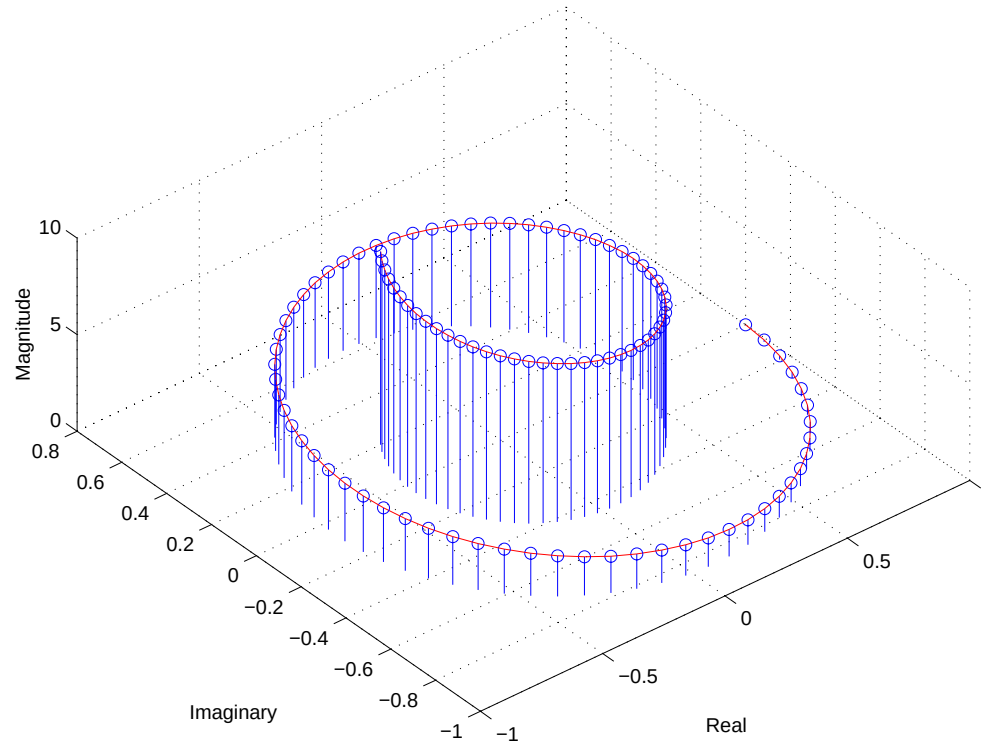
Example — Combining Stem and Line Plots

Three-dimensional stem plots work well for visualizing discrete functions that do not output a large number of data points. For example, you can use `stem3` to visualize the Laplace transform basis function, $y = e^{-st}$, for a particular constant value of `s`.

```
t = 0:.1:10; % Time limits
s = 0.1+i; % Spiral rate
y = exp(-s*t); % Compute decaying exponential
```

Using `t` as magnitudes that increase with time, create a spiral with increasing height and draw a curve through the tops of the stems to improve definition.

```
stem3(real(y),imag(y),t)
hold on
plot3(real(y),imag(y),t,'r')
hold off
view(-39.5,62)
```



Label the Graph

Add axes labels, with the statements

```
xlabel('Real')
ylabel('Imaginary')
zlabel('Magnitude')
```

Stairstep Plots

Stairstep plots display data as the leading edges of a constant interval (i.e., zero-order hold state). This type of plot holds the data at a constant *y*-value for all values between *x(i)* and *x(i+1)*, where *i* is the index into the *x* data.

This type of plot is useful for drawing time-history plots of digitally sampled data systems.

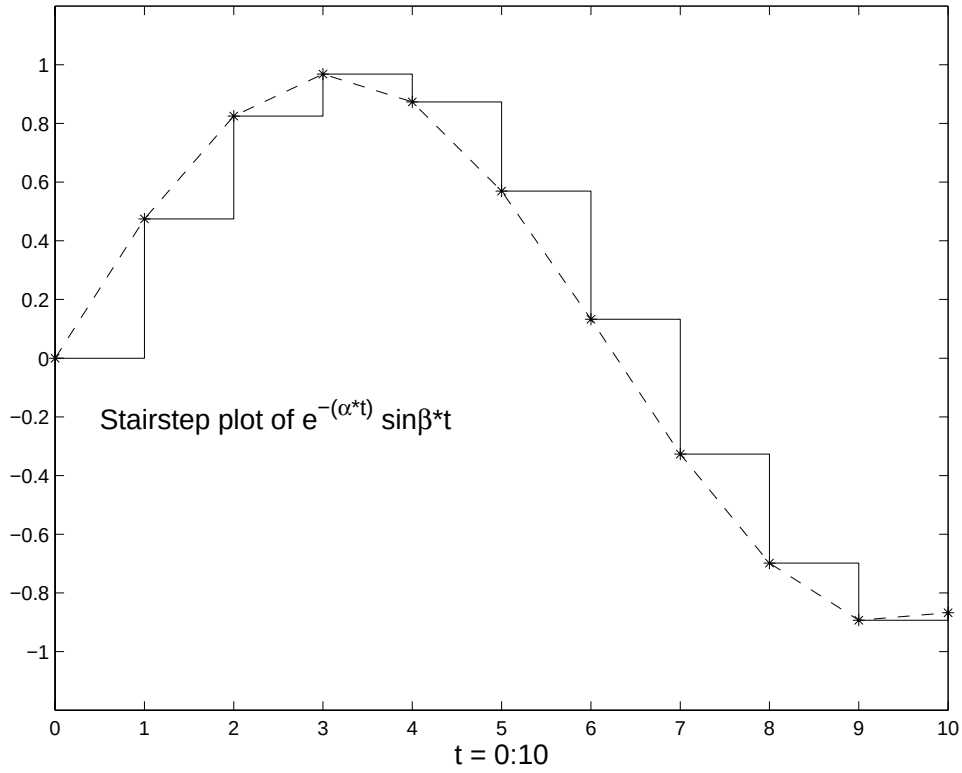
Example — Stairstep Plot of a Function

Define a function `f` that varies over time,

```
alpha = 0.01;
beta = 0.5;
t = 0:10;
f = exp(-alpha*t).*sin(beta*t);
```

Use `stairs` to display the function as a stairstep plot and a linearly interpolated function.

```
stairs(t,f)
hold on
plot(t,f,'--*')
hold off
```



Annotate the graph and set the axes limits.

```
label = 'Stairstep plot of e^{-(\alpha*t)} sin\beta*t';
text(0.5,-0.2,label,'FontSize',14)
xlabel('t = 0:10','FontSize',14)
axis([0 10 -1.2 1.2])
```

Direction and Velocity Vector Graphs

In this section...

Functions for Graphing Vector Quantities

MATLAB provides four functions for displaying data consisting of direction vectors and velocity vectors; three create 2-D plots and one creates 3-D plots:

Function	Description
`compass`	Displays vectors emanating from the origin of a polar plot
`feather`	Displays vectors extending from equally spaced points along a horizontal line
`quiver`	Displays 2-D vectors specified by (u,v) components
`quiver3`	Displays 3-D vectors specified by (u,v,w) components

For feather and compass plots, you can define the vectors using one or two arguments. The arguments specify the u and v components of the vectors relative to the origin. If you specify two arguments, the first specifies the u components of the vectors and the second the v components of the vectors. If you specify one argument, the functions treat the elements as complex numbers. The real parts are the u components and the imaginary parts are the v components.

For quiver plots, in addition to the u-v components, you also specify x,y locations (or x,y,z locations in the case of `quiver3`) to establish an origin for each vector.

Compass Plots

The `compass` function shows vectors emanating from the origin of a graph. The function takes Cartesian coordinates and plots them on a circular grid.

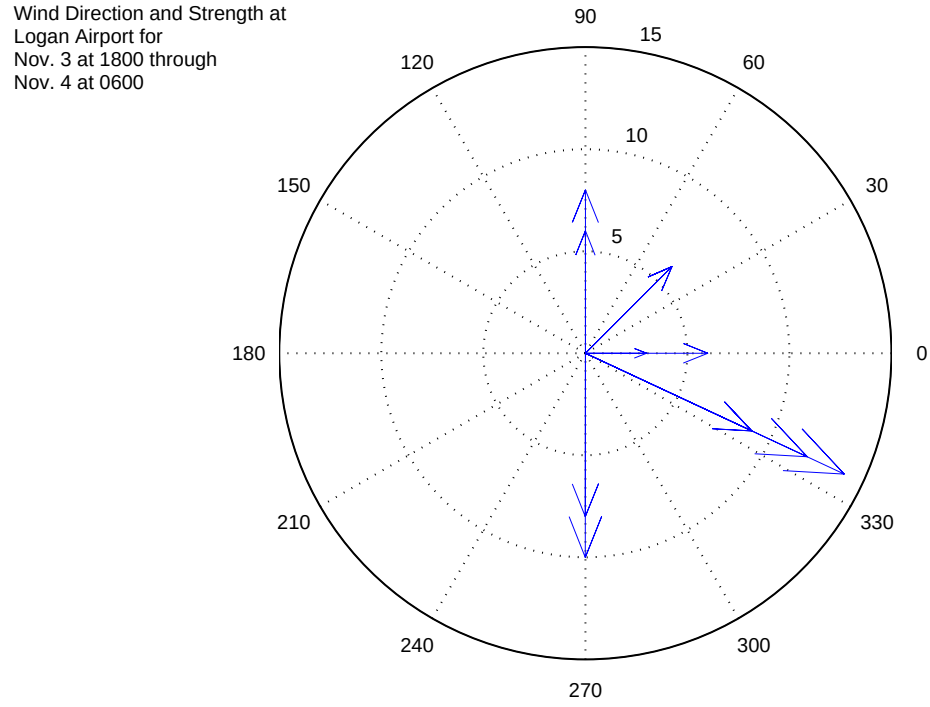
Example — Compass Plot of Wind Direction and Speed

This example shows a compass plot indicating the wind direction and strength during a 12-hour period. Two vectors define the wind direction and strength.

```
wdir = [45 90 90 45 360 335 360 270 335 270 335 335];
knots = [6 6 8 6 3 9 6 8 9 10 14 12];
```

Convert the wind direction, given as angles, into radians before converting the wind direction into Cartesian coordinates.

```
rdir = wdir * pi/180;
[x,y] = pol2cart(rdir,knots);
compass(x,y)
```



Create text to annotate the graph.

```
desc = {'Wind Direction and Strength at',
        'Logan Airport for ',
        'Nov. 3 at 1800 through',
        'Nov. 4 at 0600'};
text(-28,15,desc)
```

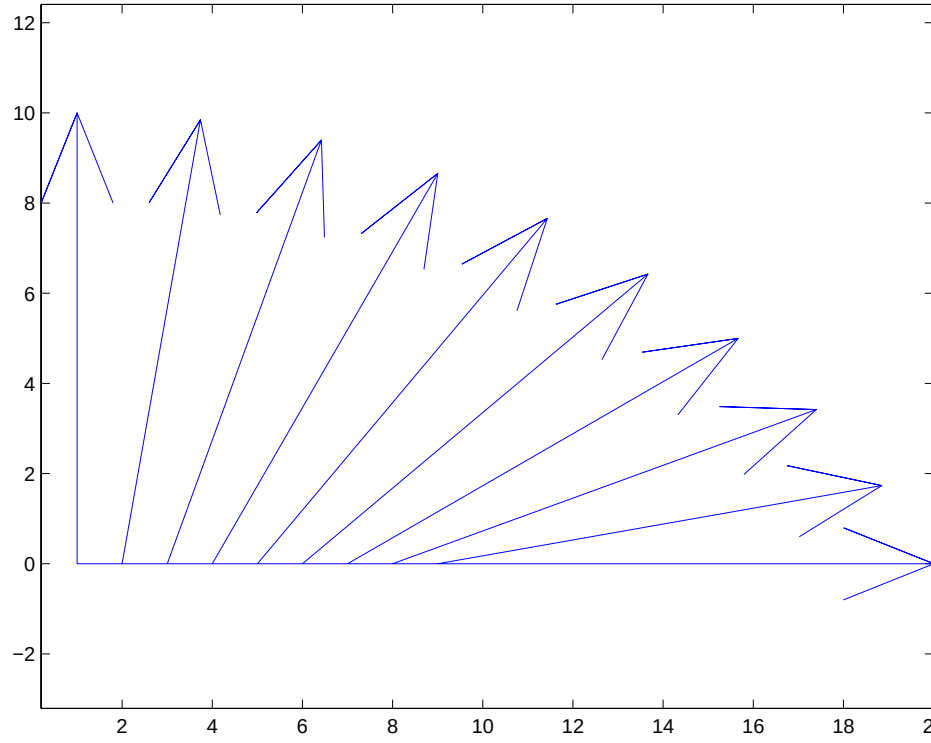
Feather Plots

The `feather` function shows vectors emanating from a straight line parallel to the *x*-axis. For example, create a vector of angles from 90° to 0° and a vector the same size, with each element equal to 1.

```
theta = 90:-10:0;
r = ones(size(theta));
```

Before creating a feather plot, transform the data into Cartesian coordinates and increase the magnitude of `r` to make the arrows more distinctive.

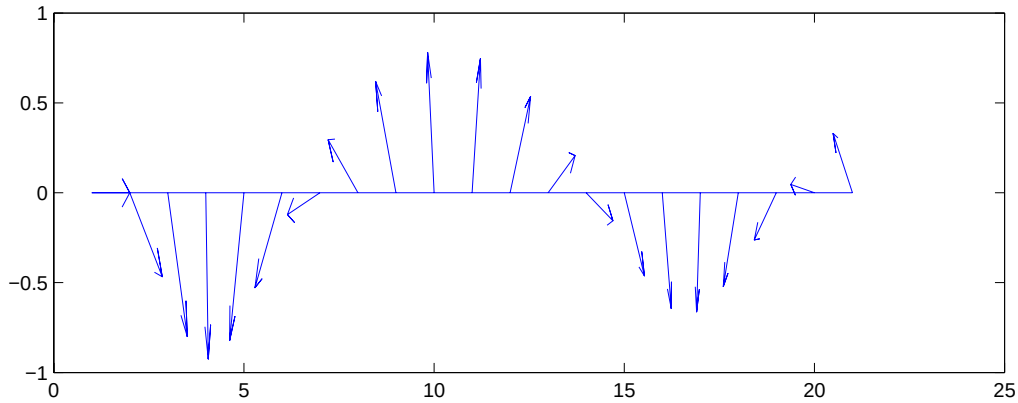
```
[u,v] = pol2cart(theta*pi/180,r*10);
feather(u,v)
axis equal
```



Plotting Complex Numbers

If the input argument `Z` is a matrix of complex numbers, `feather` interprets the real parts of `Z` as the *x* components of the vectors and the imaginary parts as the *y* components of the vectors.

```
t = 0:0.5:10;    % Time limits
s = 0.05+i;      % Spiral rate
Z = exp(-s*t);   % Compute decaying exponential
feather(Z)
```



Printing the Graph

This particular graph looks better if you change the figure's aspect ratio by stretching the figure lengthwise using the mouse. However, to maintain this shape in the printed output, set the figure's `PaperPositionMode` to `auto`.

```
set(gcf,'PaperPositionMode','auto')
```

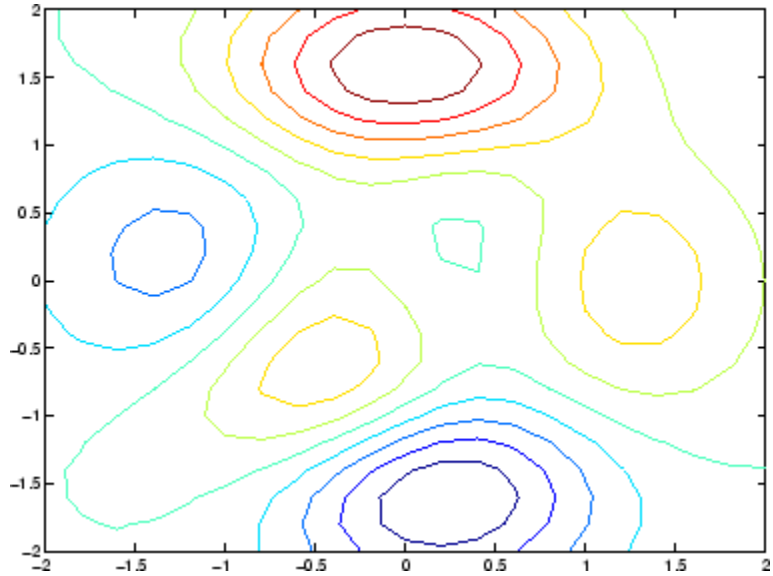
In this mode, MATLAB prints the figure as it appears on screen.

Two-Dimensional Quiver Plots

The `quiver` function shows vectors at given points in two-dimensional space. The vectors are defined by *x* and *y* components.

A quiver plot is useful when displayed with another plot. For example, create 10 contours of the `peaks` function (see “Contour Plots” on page 5-54 for more information).

```
n = -2.0:.2:2.0;
[X,Y,Z] = peaks(n);
contour(X,Y,Z,10)
```

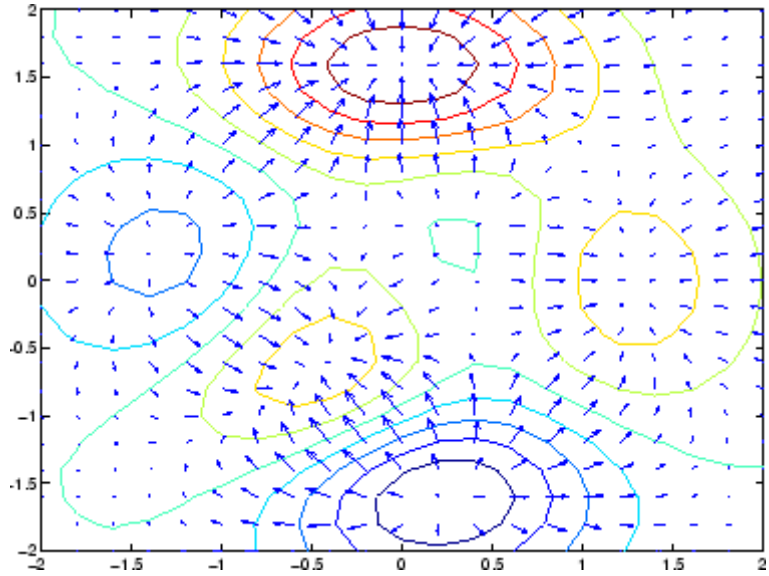


Now use `gradient` to create the vector components to use as inputs to `quiver`.

```
[U,V] = gradient(Z,.2);
```

Set `hold` to `on` and add the contour plot.

```
hold on
quiver(X,Y,U,V)
hold off
```



Three-Dimensional Quiver Plots

Three-dimensional quiver plots (`quiver3`) display vectors consisting of (u,v,w) components at (x,y,z) locations. For example, you can show the path of a projectile as a function of time, $z(t) = v_z t + \frac{a t^2}{2}$

First, assign values to the constants `vz` and `a`.

```
vz = 10;            % Velocity
a = -32;            % Acceleration
```

Then, calculate the height `z`, as time varies from `0` to `1` in increments of `0.1`.

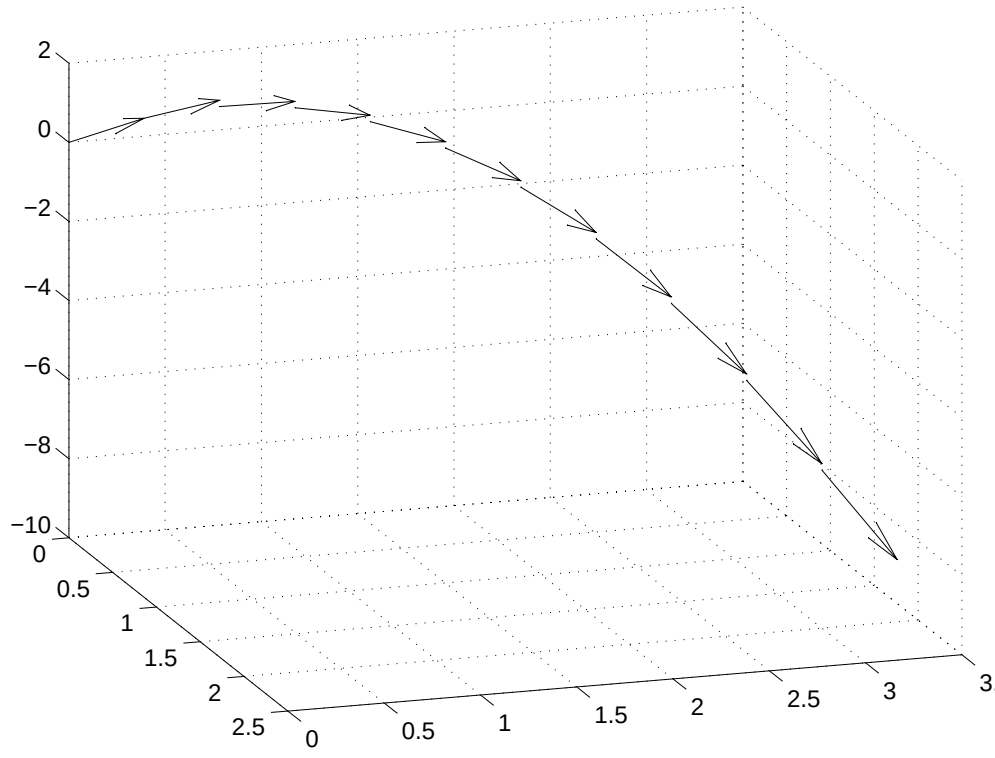
```
t = 0:.1:1;
z = vz*t + 1/2*a*t.^2;
```

Calculate the position in the x and y directions.

```
vx = 2;
x = vx*t;
vy = 3;
y = vy*t;
```

Compute the components of the velocity vectors and display the vectors using the 3-D quiver plot.

```
u = gradient(x);
v = gradient(y);
w = gradient(z);
scale = 0;
quiver3(x,y,z,u,v,w,scale)
view([70 18])
```


2
0
−2
−4
−6
−8
−10
0
0.5
1
1.5
2
2.5
0
0.5
1
1.5
2
2.5
3
3.5

Contour Plots

In this section...

Functions for Creating Contour Displays

The contouring display functions compute, plot, and label isolines (contour lines) for one or more matrices. These displays vary according to whether they plot plain contour lines, filled contour lines, raised contours, or contours in concert with mesh or surface plots. Two of the functions listed below support contouring. The low-level `contourc` function computes isolines but does not plot them. The `clabel` function places elevation labels on previously generated contours.

Function	Description
`contour`	Displays 2-D isolines generated from values given by a matrix Z
`contour3`	Displays 3-D isolines generated from values given by a matrix Z
`contourf`	Displays a 2-D contour plot and fills the area between the isolines with a solid color

Function	Description
`contourc`	Low-level function to calculate the contour matrix used by the other contour functions
`meshc`	Creates a mesh plot with a corresponding 2-D contour plot
`surfc`	Creates a surface plot with a corresponding 2-D contour plot
`clabel`	Generates labels using the contour matrix returned from calling the contouring function and displays the labels in the current figure

Creating Simple Contour Plots

`contour` and `contour3` display 2- and 3-D contours, respectively. They can be called with separate *x*, *y*, and *z* matrices, but need only one input argument — a *z* matrix interpreted as heights with respect to a plane. In this case, the contour functions determine the number of contours to display based on the minimum and maximum data values.

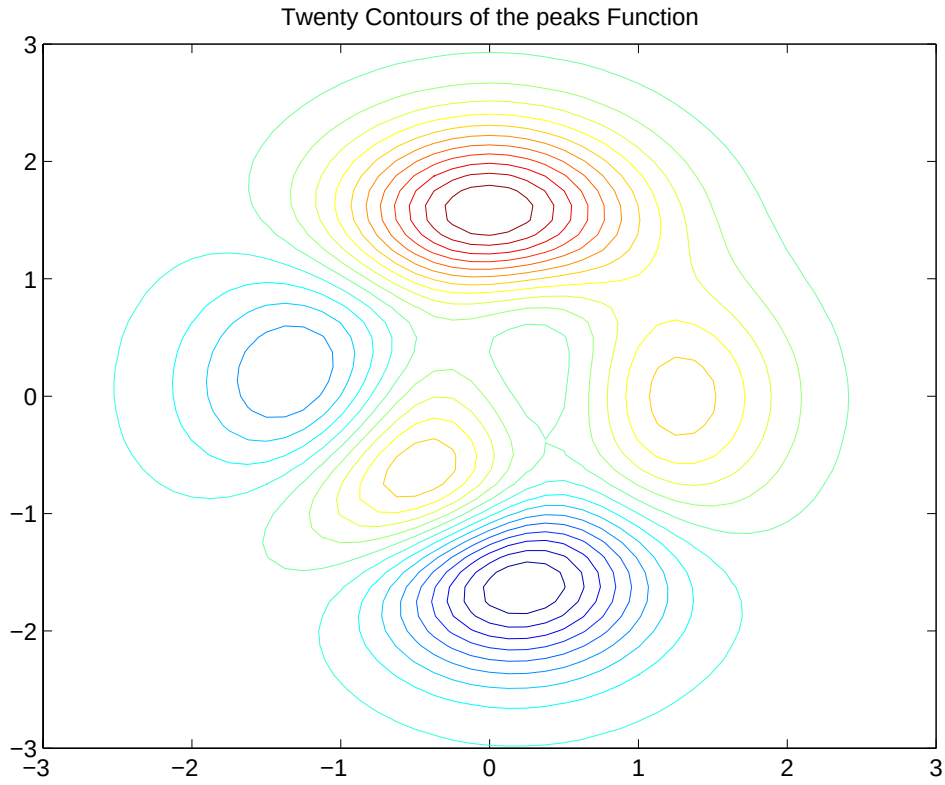
To explicitly set the number of contour levels displayed by the functions, you specify a second optional argument.

Contour Plot of the Peaks Function

The statements

```
[X,Y,Z] = peaks;
contour(X,Y,Z,20)
```

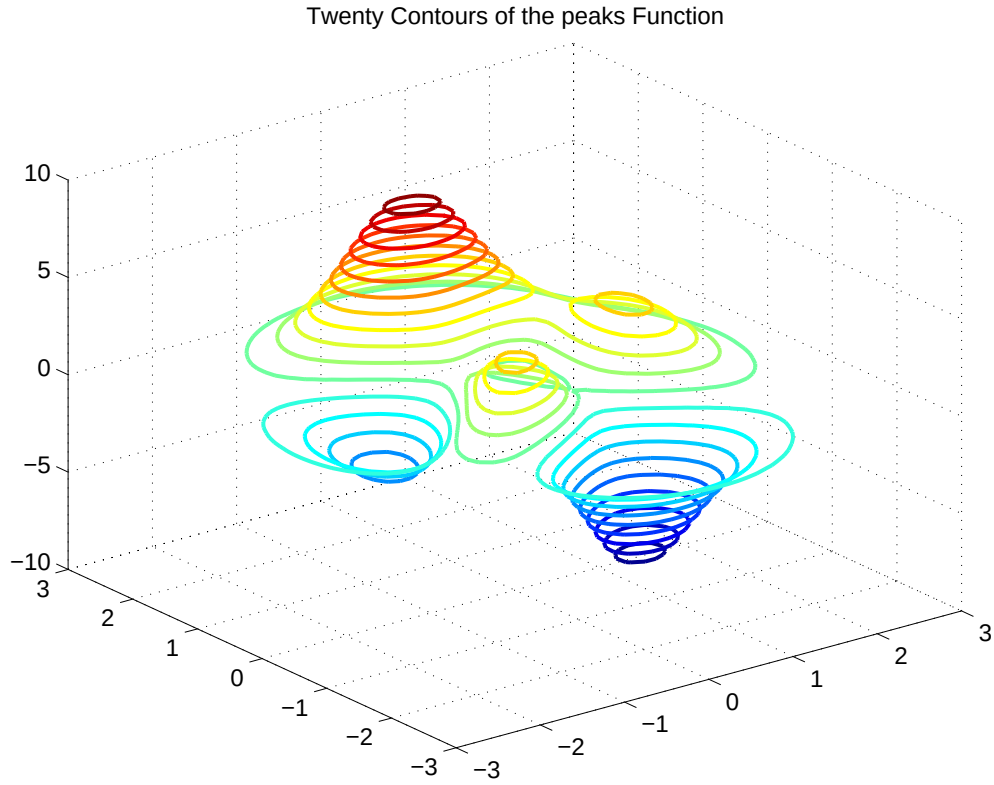
display 20 contours of the `peaks` function in a 2-D view.



The statements

```
[X,Y,Z] = peaks;
contour3(X,Y,Z,20)
h = findobj('Type','patch');
set(h,'LineWidth',2)
title('Twenty Contours of the peaks Function')
```

display 20 contours of the `peaks` function in a 3-D view and increase the line width to 2 points.



Labeling Contours

Each contour level has a value associated with it. `clabel` uses these values to display labels for 2-D contour lines. The contour matrix contains the values `clabel` uses for the labels. This matrix is returned by `contour`, `contour3`, and `contourf` and is described in “The Contouring Algorithm” on page 5-66.

`clabel` optionally returns the handles of the text objects used as labels. You can then use these handles to set the properties of the label string.

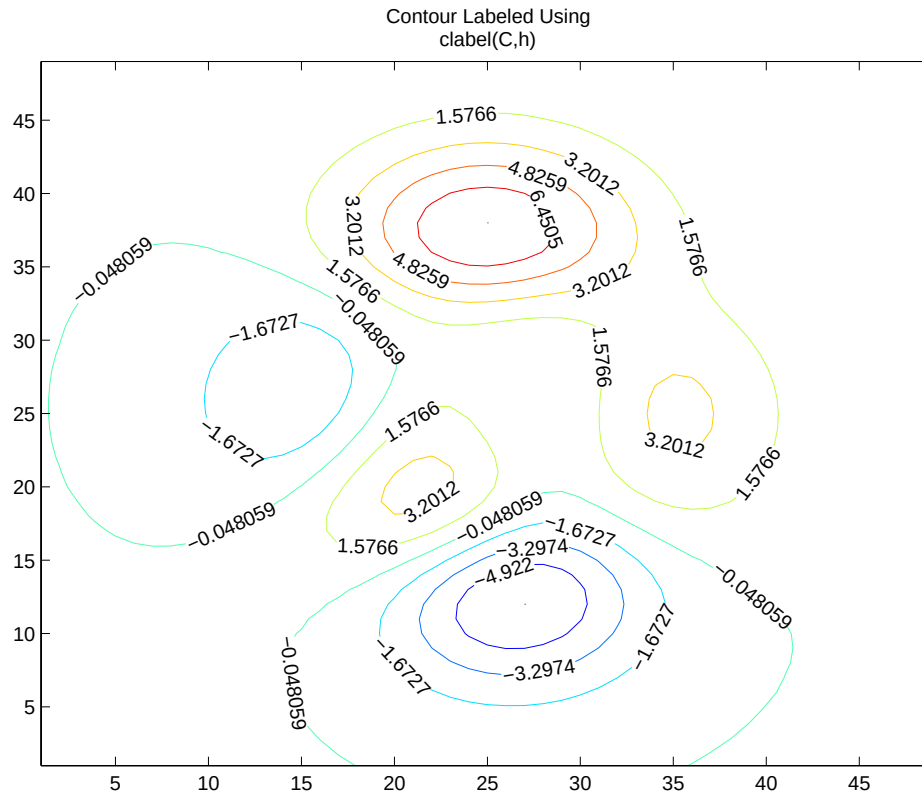
For example, display 10 contour levels of the `peaks` function,

```
Z = peaks;
[C,h] = contour(Z,10);
```

then label the contours and display a title.

```
clabel(C,h)
title({'Contour Labeled Using','clabel(C,h)'})
```

Note that `clabel` labels only those contour lines that are large enough to have an inline label inserted.



The `'manual'` option enables you to add labels by selecting the contour you want to label with the mouse.

You can also use this option to label only those contours you select interactively.

For example,

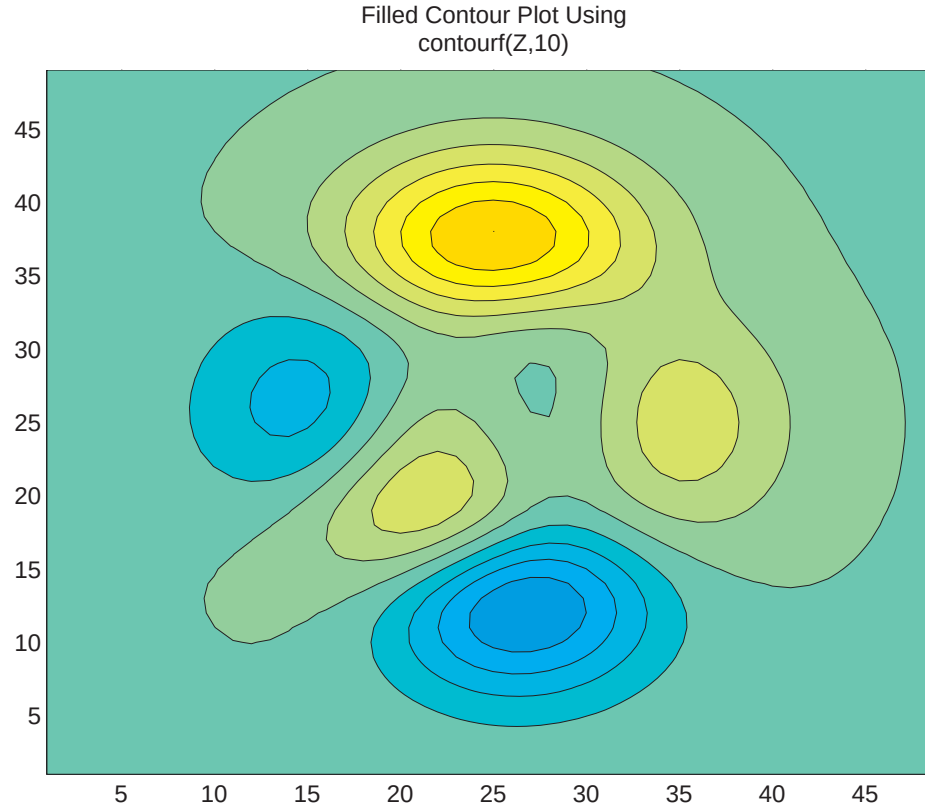
```
clabel(C,h,'manual')
```

displays a crosshair cursor when your cursor is inside the figure. Pressing any mouse button labels the contour line closest to the center of the crosshair.

Filled Contours

The `contourf` displays a two-dimensional contour plot and fills the areas between contour lines. Use `caxis` to control the mapping of contour to color. For example, this filled contour plot of the peaks data uses `caxis` to map the fill colors into the center of the colormap.

```
Z = peaks;
[C,h] = contourf(Z,10);
caxis([-20 20])
title({'Filled Contour Plot Using','contourf(Z,10)'})
```



Drawing a Single Contour Line at a Desired Level

The contouring functions permit you to specify the number of contour levels or the particular contour levels to draw. In the case of `contour`, the two forms of the function are `contour(Z,n)` and `contour(Z,v)`. `Z` is the data matrix, `n` is the number of contour lines, and `v` is a vector of specific contour levels.

MATLAB does not differentiate between a scalar and a one-element vector. So, if `v` is a one-element vector specifying a single contour at that level, `contour` interprets it as the number of contour lines, not the contour level. Consequently, `contour(Z,v)` behaves in the same manner as `contour(Z,n)`.

To display a single contour line, define `v` as a two-element vector with both elements equal to the desired contour level. For example, create a 3-D contour of the `peaks` function.

```
xrange = -3:.125:3;
yrange = xrange;
[X,Y] = meshgrid(xrange,yrange);
Z = peaks(X,Y);
contour3(X,Y,Z)
```

To display only one contour level at `Z = 1`, define `v` as `[1 1]`.

```
v = [1 1]
contour3(X,Y,Z,v)
```

Example — Visualizing Contour Construction

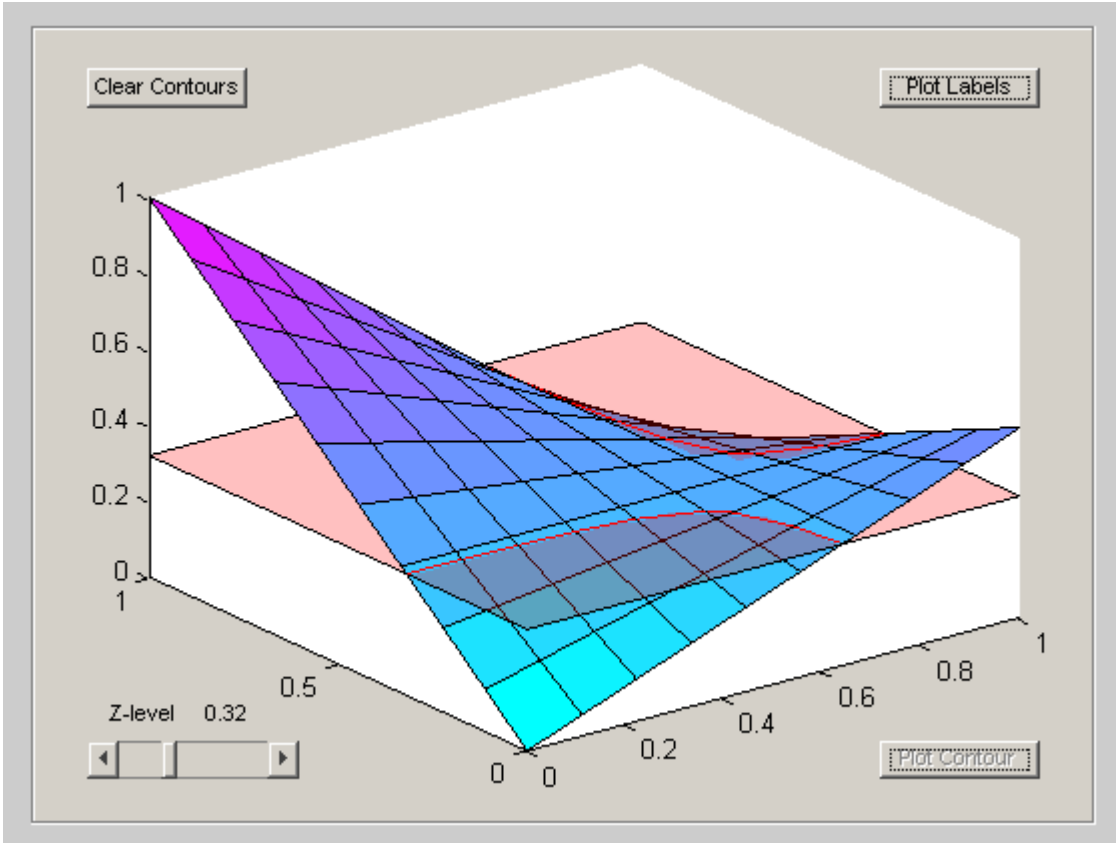
You can think of a contour as the intersection of a 3–D surface with a horizontal plane. The intersection defines 0 or more *level lines* that trace contours. The level lines either form loops or terminate at the outer edges of the surface. Contour loops can intersect at saddle points, and therefore require special handling in their vicinity.

Run the following interactive code to visualize how contour lines are constructed. Use the slider to move the plane up or down through the range of *z*-values, and click the **Plot Contour** button to draw a contour line that delineates where the plane slices through the surface. Click the **Plot Labels** button to add a label to the contour you just plotted. Click **Clear Contours** to remove all the contours and labels.

```
% Create x, y, and z arrays for a parametric surface
[x y]=meshgrid(linspace(0,1,10),linspace(0,1,10));
z = .5*x + y - 1.5*x.*y;
% Display with the surface function in 3-D
fh = figure; colormap cool;
hpl = uipanel(fh,'Units','normalized','position',[.025 .025 .95 .95]);
s=surface('xdata',x,'ydata',y,'zdata',z,'cdata',z);
view(3);hold on;
% Display a second surface, a horixontal plane at z = 0
p=surface('xdata',[0 1;0 1],'ydata',[0 0; 1 1],...
          'zdata',[0 0; 0 0],'cdata',[0 0;0 0]);
```

```
set(p,'facealpha',.25,'facecolor','red'); % Make cut plane transparent
% Create a slider control for contour elevations
hs = uicontrol(hpl,'style','slider','min',0,'max',100,...
            'units','normalized','position',[.05 .05 .2 .05],...
            'sliderstep', [.01 .05]);
set(hs,'callback',...   % Tell the slider what it should do
    ['lvl=get(hs,''value'')/100;,' ...
     'set(p,''zdata'',[lvl lvl; lvl lvl]),' ...
     'set(hto,''string'',num2str(lvl)),' ...
     'set(hbc,''enable'',''on'')']);
 lvl = 0;       % Initialize the z-level of the cutting plane
% Create a label for the slider and a text box to show its value
hst = uicontrol(hpl,'Style','text', 'String','Z-level',...
                'units','normalized','Position',[.05 .10 .1 .05]);
hto = uicontrol(hpl,'Style','text', 'String','0',...
                'units','normalized','Position',[.13 .10 .1 .05]);
% Create a pushbutton control for drawing contours with CONTOUR3
hbc = uicontrol(hpl,'style','pushbutton','enable','off',...
    'string','Plot Contour',...
    'units','normalized','position',[.80 .05 .15 .05]);
set(hbc,'callback',['[C hc] = contour3(x,y,z, [lvl lvl],''r'');' ...
    'set(hbl,''enable'',''on''), set(hbe,''enable'',''on''),' ...
    'set(hbc,''enable'',''off'')']);
% Create a pushbutton control for labelling with CLABEL,
% which uses the "contour matrix" returned from CONTOUR3
hbl = uicontrol(hpl,'style','pushbutton','enable','off',...
    'string','Plot Labels',...
    'units','normalized','position',[.80 .90 .15 ,.05]);
set(hbl,'callback',['clabel(C, hc,''color'',''r'',' ...
   '''fontweight'',''bold'');' 'set(hbl,''enable'',''off''), '...
   'set(hbe,''enable'',''on'')']);
% Create a pushbutton to clear away the contours and labels
hbe = uicontrol(hpl,'style','pushbutton','enable','off',...
    'string','Clear Contours',...
    'units','normalized','position',[.05 .90 .15 .05]);
set(hbe,'callback',['delete(findall(gca,''color'',''r''));' ...
    'set(hbe,''enable'',''off'')']);
```

Here is what the figure and its controls look like with a contour plotted at the cut line:



See “The Contouring Algorithm” on page 5-66, below, for an explanation of how MATLAB computes contour lines.

Index Contours

You can index contours to visually emphasize certain contour levels. This technique, commonly used on topographic maps to highlight contours at set altitudes such as 25, 50, 75 ... meters above sea level, provides visual cues analogous to major ticks on a graph’s axis. It is much easier to read a contour display that shows index contours because the heavier lines lessen the chance that one’s eye will jump between adjacent contours in scanning across the plot.

Example — Specifying Index Contours

The following code example highlights contours at elevations of -6, -5, -4, ... 7 for the output of the `peaks` function.

1 Generate a data matrix to contour:

```
z = peaks(100);
```

2 Compute 40 contour levels. Select contour levels so as to be round numbers; `zlevs` is the vector of contour levels to be plotted:

```
zmin = floor(min(z(:))); zmax = ceil(max(z(:)));
zinc = (zmax - zmin) / 40;
zlevs = zmin:zinc:zmax;
```

3 Specify the vertical distance between index contours; here it is unity, but it can be any modulus of values in zlevs.

```
zindex = 1;
```

4 Plot 2-D level lines with the CONTOUR function:

```
[c2,hc2] = contour(z,zlevs);
```

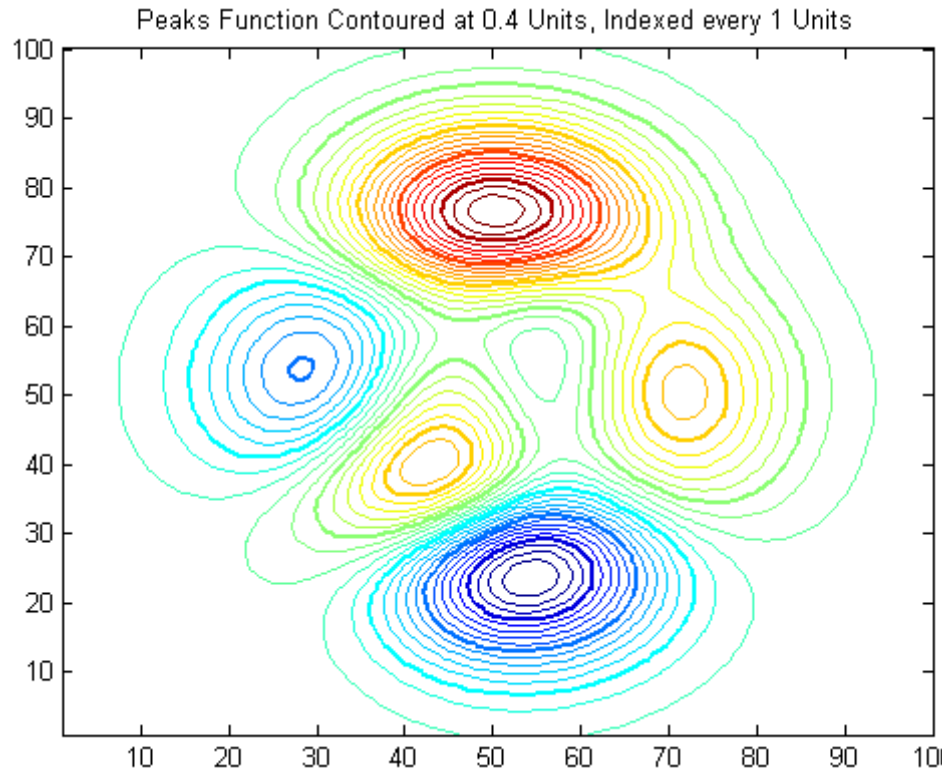
5 Create Index Contours by thickening level lines every `zindex` units

```
nc = get(hc2,'Children');
for i = 1:length(nc)
   ud = get(nc(i),'UserData');
   if (mod(ud,zindex) == 0)
       set(nc(i),'LineWidth',2);
   end
end
```

A contour line will thicken with each call to `set`.

6 Annotate to identify the contouring parameters used

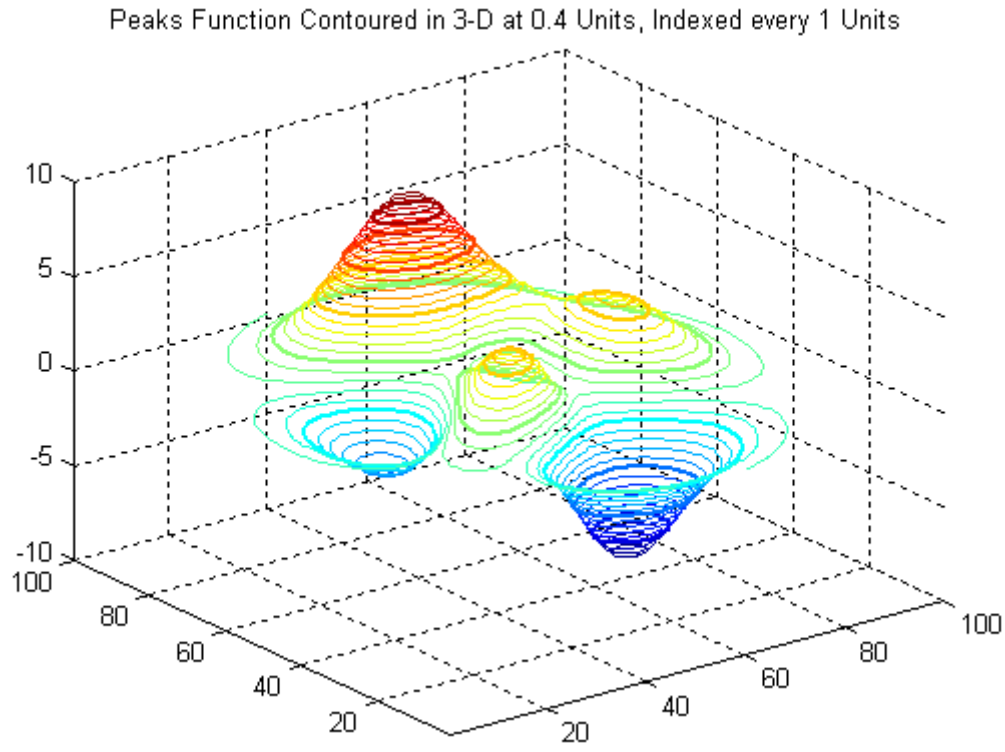
```
s = sprintf('%s %g %s %g %s', 'Peaks Function Contoured at', ...
    zinc, 'Units, Indexed every', zindex, 'Units');
title(s)
```



The loop of code in step 5 above works for `contour` but not `contour3`, because `contour3` does not create `contourgroup` objects containing `Children`. To accomplish the same result with `contour3` you must dereference the handle to the contours returned by `contour3` (`hc3`, below) differently, as follows:

```
figure;
[c3,hc3] = contour3(z,zlevs);
for i = 1:length(hc3)
   ud = get(hc3(i),'UserData');
   if (mod(ud,zindex) == 0)
       set(hc3(i),'LineWidth',2);
   end
end
s = sprintf('%s %g %s %g %s',...
```

```
    'Peaks Function Contoured in 3-D at', ...
    zinc, 'Units, Indexed every', zindex, 'Units');
title(s)
```



The Contouring Algorithm

The `contourc` function calculates the contour matrix for the other contour functions. It is a low-level function that is not called from the command line.

The contouring algorithm first determines which contour levels to draw. If you specified the input vector `v`, the elements of `v` are the contour level values, and `length(v)` determines the number of contour levels generated. If you do not specify `v`, the algorithm chooses no more than 20 contour levels that are divisible by 2 or 5.

The height matrix `Z` has either has associated `X` and `Y` matrices that locate each value in `Z` at the intersection of a row and a column, or these are inferred when they are unspecified. The width of rows and columns can vary, but typically is constant (i.e. `Z` is a regular grid).

Set the current level, `c`, equal to the lowest contour level to be plotted within the range `[min(Z) max(Z)]`. The contouring algorithm checks each edge of every square in the grid to see if `c` is between the two *z*-values for the edge points. If so, then a contour at that level crosses the edge, and a linear interpolation is performed:

`t=(c-Z0)/(Z1-Z0)`, where `Z0` is the *z*-value at one edge point, and `Z1` is the *z*-value at the other edge point.

Start indexing a new contour line (`i = 1`) for level `c` by interpolating *x* and *y*:

```
cx(i) = X0+t*(X1-X0)
cy(i) = Y0+t*(Y1-Y0)
```

Walk around the edges of the square just entered; the next edge with *z*-values that bracket `c` is where the contour exits. Increment `i`, compute `t` for the edge, and then `cx(i)` and `cy(i)`, as above.

Mark the square as having been visited. Keep checking the edges of each square entered to determine the exit edge until the line`(cx,cy)` closes on its initial point or exits the grid. If the square being entered is already marked, the contour line closes there. Copy `cx`, `cy`, `c`, and `i` to the contour line data structure (the matrix returned by contouring functions, described below).

Reinitialize `cx`, `cy`, and `i`. Move to an unmarked square and test its edges for intersections; when you find one at level `c`, repeat the above operations. Any number of contour lines can exist for a given level.

Clear all the markers, increment the contour level, and repeat until `c` exceeds `max(Z)`.

Extra logic is needed for squares where a contour passes through all four edges (saddle points) to determine which pairs of edges to connect.

`contour`, `contour3`, and `contourf` return a two-row matrix specifying all the contour lines. The format of the matrix is

```
C = [    value1    xdata(1)    xdata(2)...
         numv      ydata(1)    ydata(2)...]
```

The first row of the column that begins each definition of a contour line contains the value of the contour, as specified by `v` and used by `clabel`. Beneath that value is the number of (x,y) vertices in the contour line. Remaining columns contain the data for the (x,y) pairs. For example, the contour matrix calculated by `C = contour(peaks(3))` is

Three vertices at v = −0.2

Three vertices at v = 0

Five vertices at v = 0.8

```
Columns 1 through 7
-0.2000    1.8165    2.0000    2.1835         0    1.0003    2.0000
 3.0000    1.0000    1.0367    1.0000    3.0000    1.0000    1.1998

Columns 8 through 14
 3.0000         0    1.0000    1.0359    1.0000    0.2000    1.6669
 1.0002    3.0000    2.9991    2.0000    1.0018    5.0000    3.0000

Columns 15 through 21
 1.2324    2.0000    2.8240    2.3331    0.4000    2.0000    2.6130
 2.0000    1.3629    2.0000    3.0000    5.0000    2.8530    2.0000

Columns 22 through 28
 2.0000    1.4290    2.0000    0.6000    2.0000    2.4020    2.0000
 1.5261    2.0000    2.8530    5.0000    2.5594    2.0000    1.6892

Columns 29 through 35
 1.6255    2.0000    0.8000    2.0000    2.1910    2.0000    1.8221
 2.0000    2.5594    5.0000    2.2657    2.0000    1.8524    2.0000

Column 36
 2.0000
 2.2657
```

The circled values begin each definition of a contour line.

Changing the Offset of a Contour

The `surfc` and `meshc` functions display contours beneath a surface or a mesh plot. These functions draw the contour plot at the axes' minimum z-axis limit.

To specify your own offset, you must change the `ZData` values of the contour lines. First, save the handles of the graphics objects created by `meshc` or `surfc`.

```
h = meshc(peaks(20));
```

The first handle belongs to the mesh or surface. The remaining handles belong to the contours you want to change. To raise the contour plane, increment the *z* coordinate of each contour line by some amount by resetting its `Zdata` value:

```
for i = 2:length(h);
    newz = get(h(i),'Zdata') + 5;
    set(h(i),'Zdata',newz)
end
```

Displaying Contours in Polar Coordinates

You can contour data defined in the polar coordinate system. As an example, set up a grid in polar coordinates and convert the coordinates to Cartesian coordinates.

```
[th,r] = meshgrid((0:5:360)*pi/180,0:.05:1);
[X,Y] = pol2cart(th,r);
```

Then generate the complex matrix `Z` on the interior of the unit circle.

```
Z = X+i*Y;
```

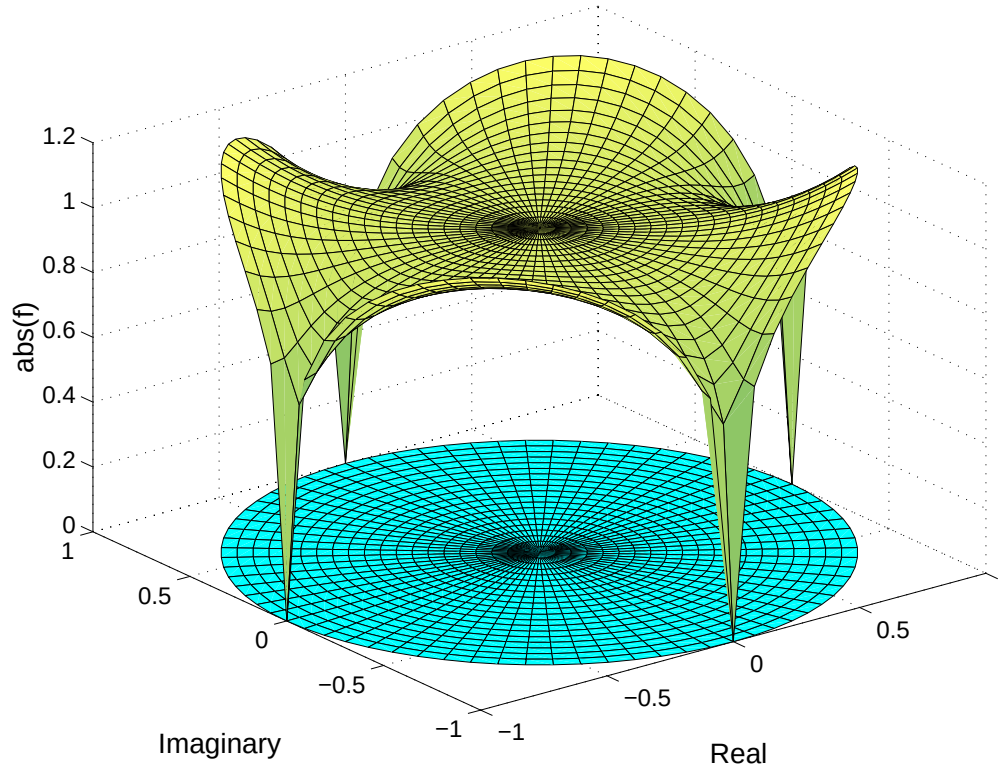
`X`, `Y`, and `Z` are points inside the circle.

Create and display a surface of the function $\sqrt[4]{Z^4 - 1}$.

```
f = (Z.^4-1).^(1/4);
surf(X,Y,abs(f))
```

Display the unit circle beneath the surface using the statements

```
hold on
surf(X,Y,zeros(size(X)))
hold off
```



Labeling the Graph

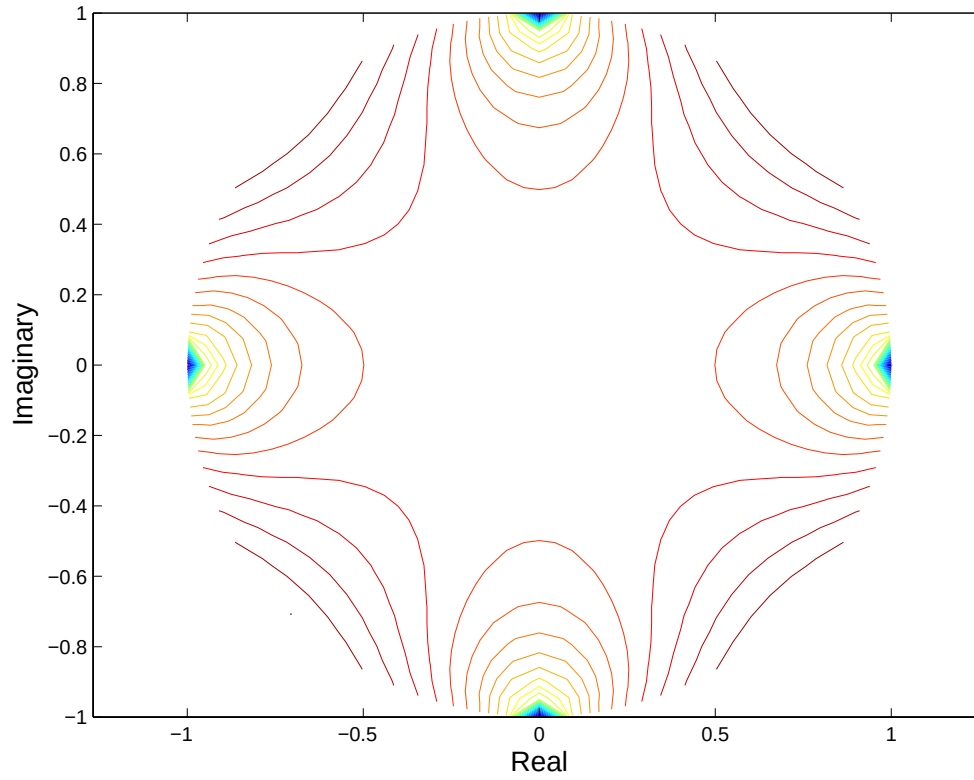
These statements add labels.

```
xlabel('Real','FontSize',14);
ylabel('Imaginary','FontSize',14);
zlabel('abs(f)','FontSize',14);
```

Contours in Cartesian Coordinates

These statements display a contour of the surface in Cartesian coordinates and label the *x*- and *y*-axis.

```
contour(X,Y,abs(f),30)
axis equal
xlabel('Real','FontSize',14);
ylabel('Imaginary','FontSize',14);
```



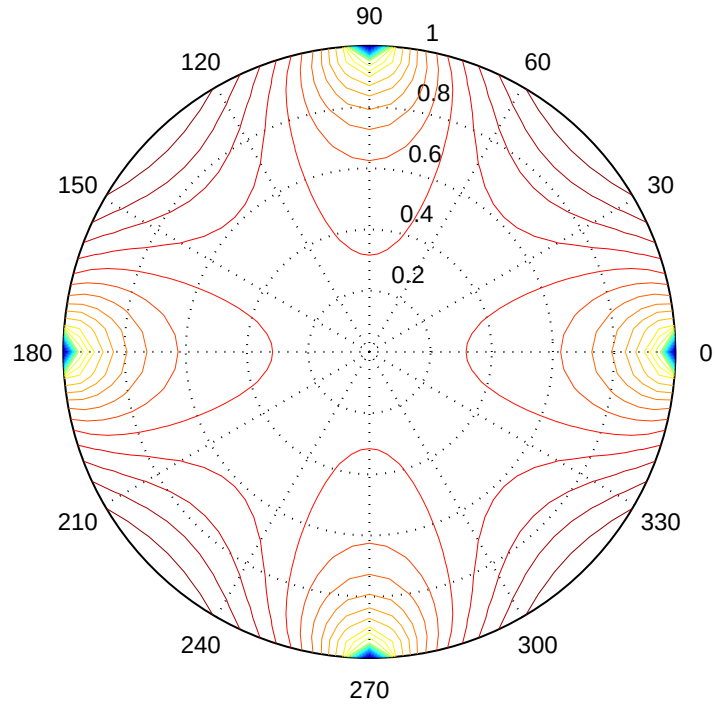
Contours on a Polar Axis

You can also display the contour within a polar axes. Create a polar axes using the `polar` function, and then delete the line specified with `polar`.

```
h = polar([0 2*pi], [0 1]);
delete(h)
```

With `hold on`, display the contour on the polar grid.

```
hold on
contour(X,Y,abs(f),30)
```



Preparing Data for Contouring

The various *contour* plotting functions, as well as the *mesh* and *surface* families of functions, accept 2-D matrices as inputs. For most applications, these input grids will represent continuous functions of two variables or relatively continuous fields of data. In many applications, source data might consist of *z*-values sampled over a two-dimensional domain in an irregular fashion, such as discrete spot elevations from GPS measurements (in the form of *x*, *y*, and *z* data vectors). To prepare such data for contour or mesh display, you need to *interpolate* it in some fashion.

MATLAB provides methods for interpolating data into vectors, grids and triangulated (Delaunay) tessellations. Input observations can be one-, two-, three- or higher-dimensional. By choosing and using these functions carefully you can control parameters and constraints for interpolation to model your assumptions about the underlying nature of the raw data. Typically, you would use the `interp2`, `meshgrid`, and `griddata` functions to interpolate *z*-values for scattered *x*-*y* data points into a 2-D grid. See "Interpolation" in the MATLAB Mathematics documentation for discussion and examples of data interpolation using these and other functions.

If the surface you are contouring is "noisy," contours depicting it will exhibit jaggedness. When you analyze and explore such data, you can filter it to attentuate high-frequency variations. One way to do this is with a convolution (with `conv2` or `filter2`) filter, as the following example demonstrates:

Example — Smoothing a Matrix for Plotting Contours

The `conv2` and `filter` functions can remove high-frequency components from a matrix representing a continuous surface or field to make the underlying data easier to visualize. This requires few assumptions

1. Create a function of two variables and plot contour lines at a specified, fixed interval.

```
Z = peaks(100);
figure;
set (gcf,'position',[400,100,600,600], 'color','w')
subplot(2,2,1);
cl = [-7:1:10];         % Define contour levels for all plots
contour(Z, cl)
```

```
axis([0 100 0 100]); colormap autumn;
set(gca,'Xtick',[0 100],'Ytick',[0 100]);
title('Peaks Surface (underlying data)')
```

2. Add uniform random noise with mean of 0 to the surface and plot resulting contours. Irregularities in the contours tend to obscure the trend of the data:

```
ZN = Z + rand(100) - .5;
subplot (2,2,2)
contour(ZN, cl)
axis([0 100 0 100]);
set(gca,'Xtick',[0 100],'Ytick',[0 100]);
title('Peaks Surface (noise added)')
```

3. Specify a 3x3 convolution kernal, `F`, for smoothing the matrix and use the `conv2` function to attenuate high spatial frequencies in the surface data:

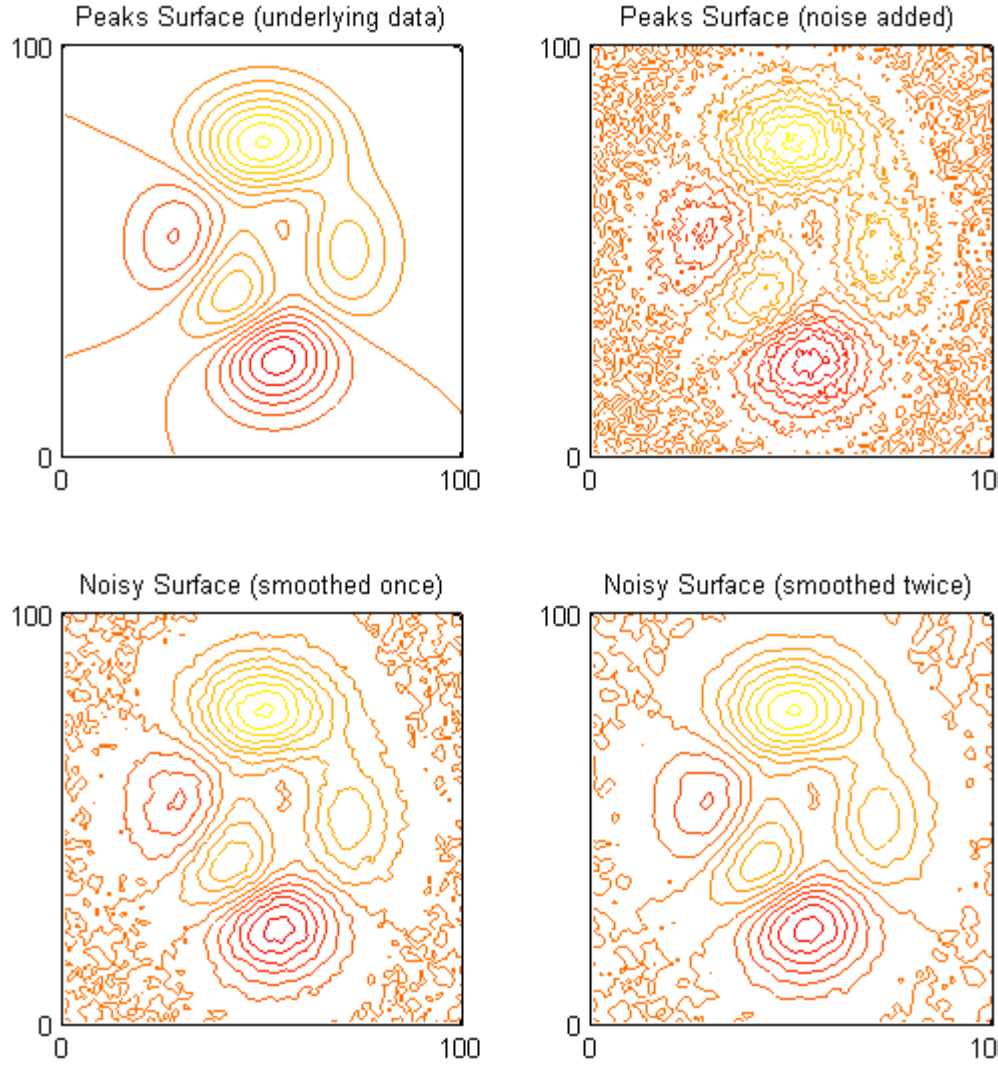
```
F = [.05 .1 .05; .1 .4 .1; .05 .1 .05];
ZC = conv2(ZN,F,'same');
```

4. Visually compare the smoothed surface to the original and the noisy ones:

```
subplot (2,2,3)
contour(ZC, cl)
axis([0 100 0 100]);
set(gca,'Xtick',[0 100],'Ytick',[0 100]);
title('Noisy Surface (smoothed once)')
```

5. Smooth the surface one more time using the same operator and compare (a larger or more uniform kernal could have achieved this in one pass):

```
ZC2 = conv2(ZC,F,'same');
subplot (2,2,4)
contour(ZC2, cl)
axis([0 100 0 100]);
set(gca,'Xtick',[0 100],'Ytick',[0 100]);
title('Noisy Surface (smoothed twice)')
```


Peaks Surface (underlying data)
Peaks Surface (noise added)
Noisy Surface (smoothed once)
Noisy Surface (smoothed twice)
100
0
0
100

Interactive Plotting

Example — Selecting Plotting Points from the Screen

You can interact with graphs or generate *x-y* coordinates interactively. The `ginput` function enables you to use the mouse or the arrow keys to select points to plot. `ginput` returns the coordinates of the pointer's position, either the current position or the position when a mouse button or key is pressed. See the `ginput` function for more information. You can use it to pick points on a graph to return their *x* and *y* values for processing, to outline an area of interest, or to draw arbitrary shapes.

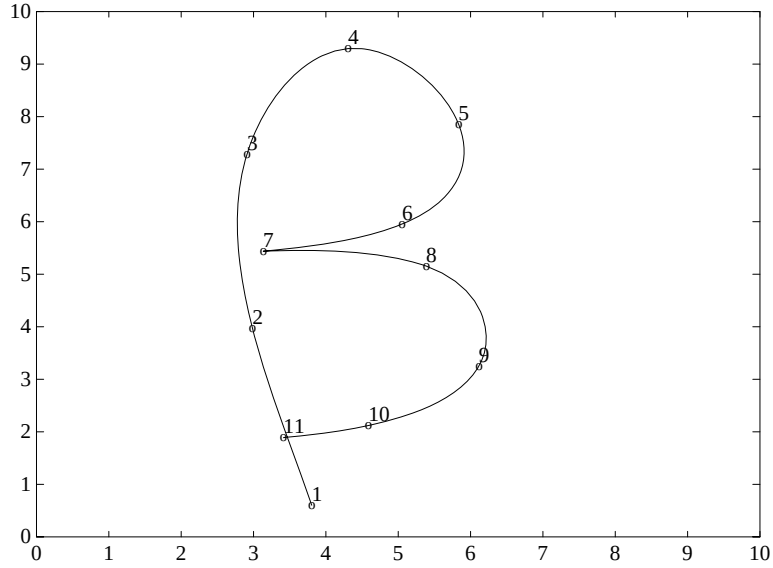
This example illustrates the use of `ginput` with the `spline` function to create a curve by interpolating in two dimensions.

First, select a sequence of points, `[x,y]`, in the plane with `ginput`. Then pass two one-dimensional splines through the points, evaluating them with a spacing one-tenth of the original spacing.

```
axis([0 10 0 10])
hold on
% Initially, the list of points is empty.
xy = [];
n = 0;
% Loop, picking up the points.
disp('Left mouse button picks points.')
disp('Right mouse button picks last point.')
but = 1;
while but == 1
    [xi,yi,but] = ginput(1);
    plot(xi,yi,'ro')
    n = n+1;
    xy(:,n) = [xi;yi];
end
% Interpolate with a spline curve and finer spacing.
t = 1:n;
ts = 1: 0.1: n;
xys = spline(t,xy,ts);

% Plot the interpolated curve.
```

```
plot(xys(1,:),xys(2,:),'b-');
hold off
```

This plot shows some typical output.



Animation

In this section...

Ways to Animate Plots

You can create animated sequences with MATLAB in two different ways:

- Save a number of different pictures and then play them back as a movie.
- Continually erase and then redraw the objects on the screen, making incremental changes with each redraw.

Movies are better suited to situations where each frame is fairly complex and cannot be redrawn rapidly. You create each movie frame in advance so the original drawing time is not important during playback, which is just a matter of blitting the frame to the screen. A movie is not rendered in real time; it is simply a playback of previously rendered frames.

The second technique, drawing, erasing, and then redrawing, makes use of different drawing modes supported by MATLAB. These modes allow faster redrawing at the expense of some rendering accuracy, so you must consider which mode to select.

This section provides an example of each technique. To see more sophisticated demonstrations of these features, type `demo` at the MATLAB prompt and explore the animation demonstrations.

Movies

You can save any sequence of graphs and then play the sequence back in a short *movie*. There are two steps to this process:

- Use `getframe` to generate each movie frame. Be sure that your computer is not in screen saver mode when `getframe` is called, and in the event that you are using several virtual desktops, that the one on which MATLAB is running is visible on your monitor.
- Use `movie` to run the movie a specified number of times at the specified rate.

Typically, you use `getframe` in a `for` loop to assemble the array of movie frames. `getframe` returns a structure having the following fields:

- `cdata` — Image data in a `uint8` matrix. The matrix has dimensions of height-by-width on indexed-color systems and height-by-width-by-3 on truecolor systems.
- `colormap` — The colormap in an n-by-3 matrix, where n is the number of colors. On truecolor systems, the `colormap` field is empty.

See `image` for more information on images.

Example — Visualizing an FFT as a Movie

This example illustrates the use of movies to visualize the quantity `fft(eye(n))`, which is a complex *n*-by-*n* matrix whose elements are various powers of the nth root of unity, `exp(i*2*pi/n)`.

Creating the Movie

Create the movie in a `for` loop calling `getframe` to capture the graph. Since the `plot` command resets the axes properties, call `axis equal` within the loop before `getframe`.

```
for k = 1:16
 plot(fft(eye(k+16)))
 axis equal
 M(k) = getframe;
end
```

Running the Movie

After generating the movie, you can play it back any number of times. To play it back 30 times, type

```
movie(M,30)
```

You can readily generate and smoothly play back movies with a few dozen frames on most computers. Longer movies require large amounts of primary memory or a very effective virtual memory system.

Movies that Include the Entire Figure

If you want to capture the contents of the entire figure window (for example, to include GUI components in the movie), specify the figure's handle as an argument to the `getframe` command. For example, suppose you want to add a slider to indicate the value of `k` in the previous example.

```
h = uicontrol('style','slider','position',...
 [10 50 20 300],'Min',1,'Max',16,'Value',1)
for k = 1:16
 plot(fft(eye(k+16)))
 axis equal
 set(h,'Value',k)
 M(k) = getframe(gcf);
end
```

In this example, the movie frame contains the entire figure. To play so that it looks like the original figure, make the playback axes fill the figure window.

```
clf
axes('Position',[0 0 1 1])
movie(M,30)
```

Erase Modes

You can select the method MATLAB uses to redraw graphics objects. One event that causes MATLAB to redraw an object is changing the properties of that object. You can take advantage of this behavior to create animated sequences. A typical scenario is to draw a graphics object, then change its position by respecifying the *x*-, *y*,- and *z*-coordinate data by a small amount with each pass through a loop.

You can create different effects by selecting different erase modes. This section illustrates how to use the three modes that are useful for dynamic redrawing:

- `none` — MATLAB does not erase the objects when it is moved.
- `background` — MATLAB erases the object by redrawing it in the background color. This mode erases the object and anything below it (such as grid lines).
- `xor` — This mode erases only the object and is usually used for animation.

All three modes are faster (albeit less accurate) than the normal mode used by MATLAB.

Example — Animating with Erase Modes

It is often interesting and informative to see 3-D trajectories develop in time. This example involves chaotic motion described by a nonlinear differential equation known as the Lorenz strange attractor. It can be written in the form $\frac{dy}{dt} = Ay$

with a vector-valued function *y(t)* and a matrix *A* that depends upon *y*.

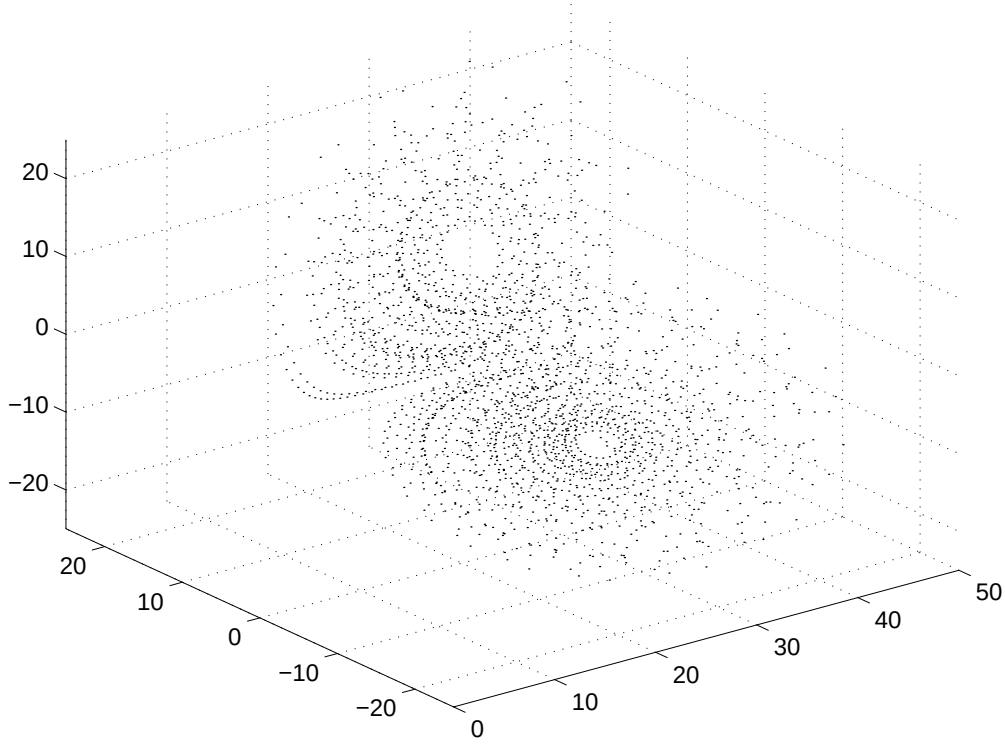
$$A(y) = \begin{bmatrix} -\frac{8}{3} & 0 & y(2) \\ 0 & -10 & 10 \\ -y(2) & 28 & -1 \end{bmatrix}$$

The solution orbits about two different attractive points without settling into a steady orbit about either. This example approximates the solution with the simplest possible numerical method — Euler's method with fixed step size. The result is not very accurate, but it has the same qualitative behavior as other methods.

```
A = [ -8/3 0 0; 0 -10 10; 0 28 -1 ];
y = [35 -10 -7]';
h = 0.01;
p = plot3(y(1),y(2),y(3),'.', ...
    'EraseMode','none','MarkerSize',5); % Set EraseMode to none
axis([0 50 -25 25 -25 25])
hold on
for i=1:4000
    A(1,3) = y(2);
```

```
    A(3,1) = -y(2);
    ydot = A*y;
    y = y + h*ydot;
 % Change coordinates
    set(p,'XData',y(1),'YData',y(2),'ZData',y(3))
    drawnow
end
```

The `plot3` statement sets `EraseMode` to `none`, indicating that the points already plotted should not be erased when the plot is redrawn. In addition, the handle of the plot object is saved. Within the `for` loop, a `set` statement references the plot object and changes its internally stored coordinates for the new location. While this manual cannot show the dynamically evolving output, this picture shows a snapshot.



Note that, as far as MATLAB is concerned, the graph created by this example contains only one dot. What you see on the screen are remnants of previous plots that MATLAB has been instructed not to erase. The only way to print this graph from MATLAB is with a screen capture.

Background Erase Mode. To see the effect of `EraseMode background`, add these statements to the previous program.

```
p = plot3(y(1),y(2),y(3),'square', ...
    'EraseMode','background','MarkerSize',10,...
    'MarkerEdgeColor',[1 .7 .7],'MarkerFaceColor',[1 .7 .7]);
```

```
for i=1:4000
    A(1,3) = y(2);
    A(3,1) = -y(2);
    ydot = A*y;
    y = y + h*ydot;
    set(p,'XData',y(1),'YData',y(2),'ZData',y(3))
    drawnow
end
hold off
```

Since `hold` is still `on`, this code erases the previously created graph by setting the `EraseMode` property to `background` and changing the marker to a "pink eraser" (a square marker colored pink).

Xor Erase Mode. If you change the `EraseMode` of the first `plot3` statement from `none` to `xor`, you will see a moving dot (`Marker '.'`) only. `Xor` mode is used to create animations where you do not want to leave remnants of previous graphics on the screen. However, you should not rely on `Xor` mode to work properly in all situations. Some platforms and graphics subsystems do not support it and where it does work, performance can be much slower compared to other erase modes or animation methods.

Additional Examples

The MATLAB demo `lorenz` provides a more accurate numerical approximation and a more elaborate display of the Lorenz strange attractor example. Other MATLAB demos illustrate animation techniques.

6

Displaying Bit-Mapped Images

Images in MATLAB

In this section...

What Is Image Data?

The basic data structure in MATLAB is the *array*, an ordered set of real or complex elements. This object is naturally suited to the representation of *images*, real-valued, ordered sets of color or intensity data. (MATLAB does not support complex-valued images.)

MATLAB stores most images as two-dimensional arrays (i.e., matrices), in which each element of the matrix corresponds to a single pixel in the displayed image. For example, an image composed of 200 rows and 300 columns of different colored dots would be stored in MATLAB as a 200-by-300 matrix. Some images, such as RGB, require a three-dimensional array, where the first plane in the third dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities.

This convention makes working with graphics file format images in MATLAB similar to working with any other type of matrix data. For example, you can select a single pixel from an image matrix using normal matrix subscripting.

```
I(2,15)
```

This command returns the value of the pixel at row 2, column 15 of the image `I`.

The following sections describe the different data and image types you can use, and give details on how to read, write, work with, and display graphics images; how to alter the display properties and aspect ratio of an image during display; how to print an image; and how to convert the data type or graphics format of an image.

Data Types

MATLAB supports three different numeric classes for image display: double-precision floating-point (`double`), 16-bit unsigned integer (`uint16`), and 8-bit unsigned integer (`uint8`). The image display commands interpret data values differently depending on the numeric class the data is stored in. Details on the inner workings of the storage for 8- and 16-bit images are included in "Working with 8-Bit and 16-Bit Images" on page 6-10.

By default, MATLAB stores most data in arrays of class `double`. The data in these arrays is stored as double-precision (64-bit) floating-point numbers. All MATLAB functions and capabilities work with these arrays.

For images stored in one of the graphics file formats supported by MATLAB, however, this data representation is not always ideal. The number of pixels in such an image can be very large; for example, a 1000-by-1000 image has a million pixels. Since each pixel is represented by at least one array element, this image would require about 8 megabytes of memory if it were stored as class `double`.

To reduce memory requirements, MATLAB supports storing image data in arrays of class `uint8` and `uint16`. The data in these arrays is stored as 8-bit or 16-bit unsigned integers. These arrays require one-eighth or one-fourth as much memory as data in `double` arrays.

Bit Depth

MATLAB supports reading the most commonly used bit depths (bits per pixel) of any of the supported graphics file formats. When the data is in memory, it can be stored as `uint8`, `uint16`, or `double`. For details on which bit depths are appropriate for each supported format, see `imread` and `imwrite`.

Supported Image Formats

MATLAB provides commands for reading, writing, and displaying several types of graphics file formats for images. As with MATLAB-generated images, once a graphics file format image is displayed, it becomes a Handle Graphics image object. MATLAB supports the following graphics file formats, along with others:

- BMP (Microsoft Windows Bitmap)

- GIF (Graphics Interchange Files)
- HDF (Hierarchical Data Format)
- JPEG (Joint Photographic Experts Group)
- PCX (Paintbrush)
- PNG (Portable Network Graphics)
- TIFF (Tagged Image File Format)
- XWD (X Window Dump)

For more information concerning the bit depths and image types supported for these formats, see `imread` and `imwrite`.

Functions for Reading, Writing and Displaying Images

Since images are essentially two-dimensional matrices, many MATLAB functions are capable of operating on and displaying images. The following are the most useful ones, and are discussed in the sections that follow:

Function	Purpose	Function Group
`axis`	Plot axis scaling and appearance	Display
`image`	Display image (create image object)	Display
`imagesc`	Scale data and display as image	Display
`imread`	Read image from graphics file	File I/O
`imwrite`	Write image to graphics file	File I/O
`imfinfo`	Get image information from graphics file	Utility
`ind2rgb`	Convert indexed image to RGB image	Utility

Image Types

In this section...

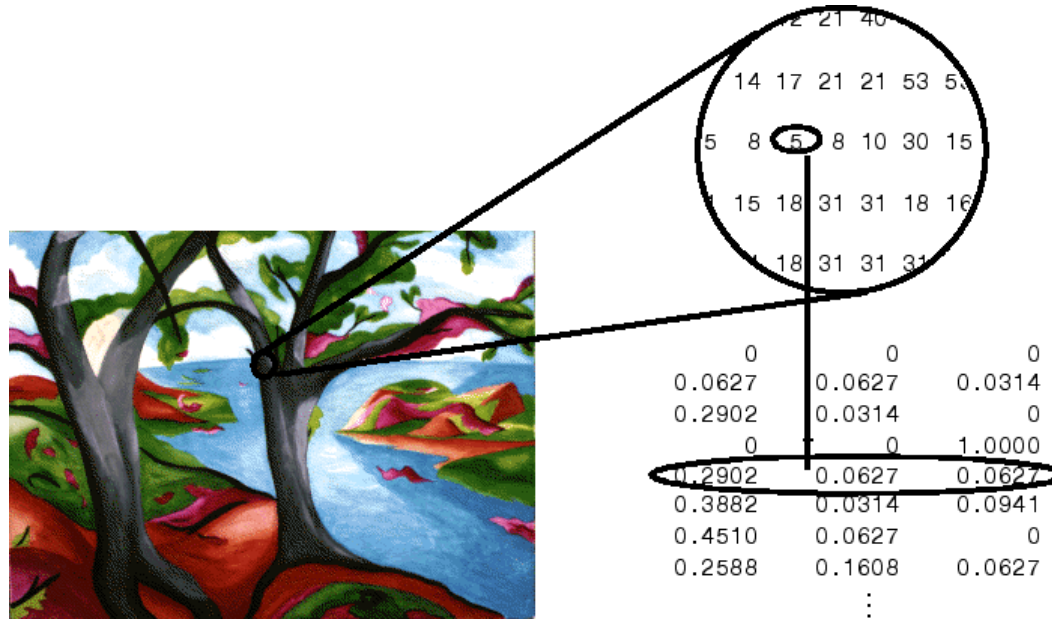
Indexed Images

An indexed image consists of a data matrix, `X`, and a colormap matrix, `map`. `map` is an *m*-by-3 array of class `double` containing floating-point values in the range [0, 1]. Each row of `map` specifies the red, green, and blue components of a single color. An indexed image uses "direct mapping" of pixel values to colormap values. The color of each image pixel is determined by using the corresponding value of `X` as an index into `map`. Values of `X` therefore must be integers. The value 1 points to the first row in `map`, the value 2 points to the second row, and so on. You can display an indexed image with the statements

```
image(X); colormap(map)
```

A colormap is often stored with an indexed image and is automatically loaded with the image when you use the `imread` function. However, you are not limited to using the default colormap — you can use any colormap that you choose. The description for the property `CDataMapping` describes how to alter the type of mapping used.

The next figure illustrates the structure of an indexed image. The pixels in the image are represented by integers, which are pointers (indices) to color values stored in the colormap.

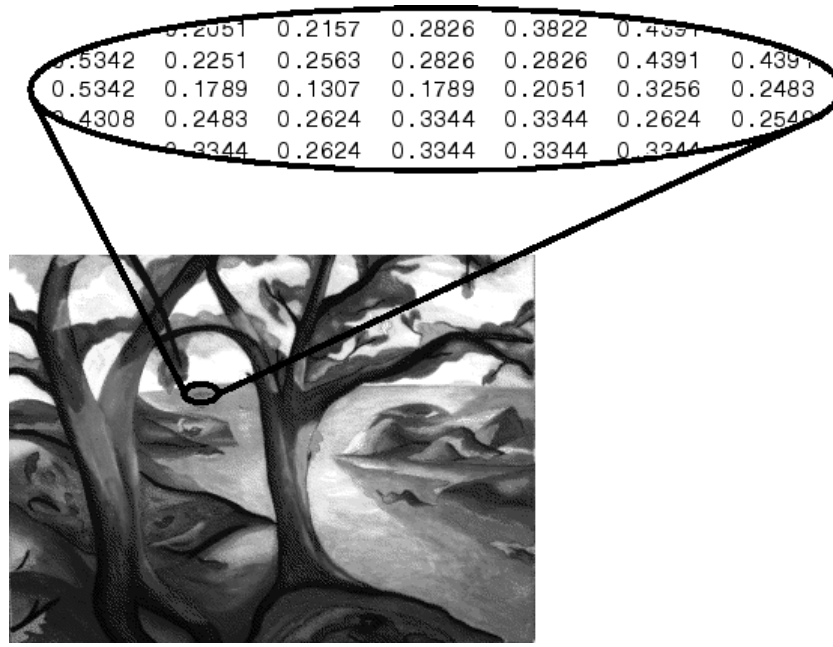


The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8` or `uint16`, there is an offset — the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on. The offset is also used in graphics file formats, to maximize the number of colors that can be supported. In the image above, the image matrix is of class `double`. Because there is no offset, the value 5 points to the fifth row of the colormap.

Intensity Images

An intensity image is a data matrix, `I`, whose values represent intensities within some range. MATLAB stores an intensity image as a single matrix, with each element of the matrix corresponding to one image pixel. The matrix can be of class `double`, `uint8`, or `uint16`. While intensity images are rarely saved with a colormap, MATLAB uses a colormap to display them. In essence, MATLAB handles intensity images as indexed images.

This figure depicts an intensity image of class `double`.



To display an intensity image, use the `imagesc` ("image scale") function, which enables you to set the range of intensity values. `imagesc` scales the image data to use the full colormap. Use the two-input form of `imagesc` to display an intensity image. For example,

```
imagesc(I,[0 1]); colormap(gray);
```

The second input argument to `imagesc` specifies the desired intensity range. The function `imagesc` displays `I` by mapping the first value in the range (usually 0) to the first colormap entry, and the second value (usually 1) to the last colormap entry. Values in between are linearly distributed throughout the remaining colormap colors.

Although it is conventional to display intensity images using a grayscale colormap, it is possible to use other colormaps. For example, the following statements display the intensity image `I` in shades of blue and green.

```
imagesc(I,[0 1]); colormap(winter);
```

To display a matrix `A` with an arbitrary range of values as an intensity image, use the single-argument form of `imagesc`. With one input argument, `imagesc` maps the minimum value of the data matrix to the first colormap entry, and maps the maximum value to the last colormap entry. For example, these two lines are equivalent.

```
imagesc(A); colormap(gray)
imagesc(A,[min(A(:)) max(A(:))]); colormap(gray)
```

RGB (Truecolor) Images

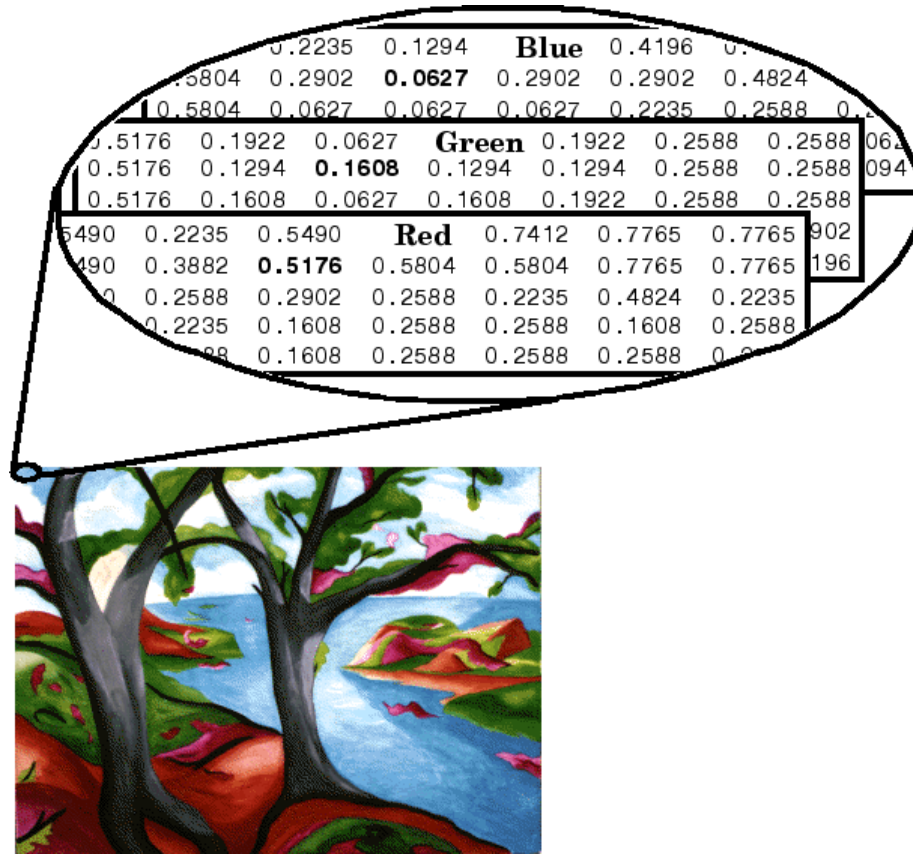
An RGB image, sometimes referred to as a "truecolor" image, is stored in MATLAB as an *m*-by-*n*-by-3 data array that defines red, green, and blue color components for each individual pixel. RGB images do not use a palette. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel's location. Graphics file formats store RGB images as 24-bit images, where the red, green, and blue components are 8 bits each. This yields a potential of 16 million colors. The precision with which a real-life image can be replicated has led to the nickname "truecolor image."

An RGB MATLAB array can be of class `double`, `uint8`, or `uint16`. In an RGB array of class `double`, each color component is a value between 0 and 1. A pixel whose color components are (0,0,0) is displayed as black, and a pixel whose color components are (1,1,1) is displayed as white. The three color components for each pixel are stored along the third dimension of the data array. For example, the red, green, and blue color components of the pixel (10,5) are stored in `RGB(10,5,1)`, `RGB(10,5,2)`, and `RGB(10,5,3)`, respectively.

To display the truecolor image `RGB`, use the `image` function. For example,

```
image(RGB)
```

The next figure shows an RGB image of class `double`.



To determine the color of the pixel at (2,3), you would look at the RGB triplet stored in (2,3,1:3). Suppose (2,3,1) contains the value `0.5176`, (2,3,2) contains `0.1608`, and (2,3,3) contains `0.0627`. The color for the pixel at (2,3) is

```
0.5176 0.1608 0.0627
```

Working with 8-Bit and 16-Bit Images

In this section...

8-Bit and 16-Bit Indexed Images

MATLAB usually works with double-precision (64-bit) floating-point numbers. However, to reduce memory requirements for working with images, MATLAB provides support for storing images as 8-bit or 16-bit unsigned integers by using the numeric classes `uint8` or `uint16`, respectively. An image whose data matrix has class `uint8` is called an 8-bit image; an image whose data matrix has class `uint16` is called a 16-bit image.

The `image` function can display 8- or 16-bit images directly without converting them to double precision. However, `image` interprets matrix values slightly differently when the image matrix is `uint8` or `uint16`. The specific interpretation depends on the image type.

If the class of `X` is `uint8` or `uint16`, its values are offset by 1 before being used as colormap indices. The value 0 points to the first row of the colormap, the value 1 points to the second row, and so on. The `image` command automatically supplies the proper offset, so the display method is the same whether `X` is `double`, `uint8`, or `uint16`.

```
image(X); colormap(map);
```

The colormap index offset for `uint8` and `uint16` data is intended to support standard graphics file formats, which typically store image data in indexed form with a 256-entry colormap. The offset allows you to manipulate and

display images of this form in MATLAB using the more memory-efficient `uint8` and `uint16` arrays.

Because of the offset, you must add 1 to convert a `uint8` or `uint16` indexed image to `double`. For example,

```
X64 = double(X8) + 1;
 or
X64 = double(X16) + 1;
```

Conversely, subtract 1 to convert a `double` indexed image to `uint8` or `uint16`.

```
X8 = uint8(X64 - 1);
 or
X16 = uint16(X64 - 1);
```

8-Bit and 16-Bit Intensity Images

Whereas the range of `double` image arrays is usually [0, 1], the range of 8-bit intensity images is usually [0, 255] and the range of 16-bit intensity images is usually [0, 65535]. Use the following command to display an 8-bit intensity image with a grayscale colormap.

```
imagesc(I,[0 255]); colormap(gray);
```

To convert an intensity image from `double` to `uint16`, first multiply by 65535.

```
I16 = uint16(round(I64*65535));
```

Conversely, divide by 65535 after converting a `uint16` intensity image to `double`.

```
I64 = double(I16)/65535;
```

8-Bit and 16-Bit RGB Images

The color components of an 8-bit RGB image are integers in the range [0, 255] rather than floating-point values in the range [0, 1]. A pixel whose color components are (255,255,255) is displayed as white. The `image` command displays an RGB image correctly whether its class is `double`, `uint8`, or `uint16`.

```
image(RGB);
```

To convert an RGB image from `double` to `uint8`, first multiply by 255.

```
RGB8 = uint8(round(RGB64*255));
```

Conversely, divide by 255 after converting a `uint8` RGB image to `double`.

```
RGB64 = double(RGB8)/255
```

To convert an RGB image from `double` to `uint16`, first multiply by 65535.

```
RGB16 = uint16(round(RGB64*65535));
```

Conversely, divide by 65535 after converting a `uint16` RGB image to `double`.

```
RGB64 = double(RGB16)/65535;
```

Mathematical Operations Support for uint8 and uint16

To use the following MATLAB functions with `uint8` and `uint16` data, first convert the data to type `double`: `conv2`, `convn`, `fft2`, `fftn`. For example, if `X` is a `uint8` image, cast the data to type `double`:

```
fft(double(X))
```

In these cases, the output is always `double`.

The `sum` function returns results in the same type as its input, but provides an option to use double precision for calculations.

Integer Mathematics in MATLAB

See "Arithmetic Operations on Integer Data Types" for more information on how mathematical functions work with data types that are not doubles.

Most of the functions in Image Processing Toolbox accept `uint8` and `uint16` input. If you plan to do sophisticated image processing on `uint8` or `uint16` data, you should consider adding Image Processing Toolbox to your MATLAB computing environment.

Other 8-Bit and 16-Bit Array Support

MATLAB supports several other operations on `uint8` and `uint16` arrays, including

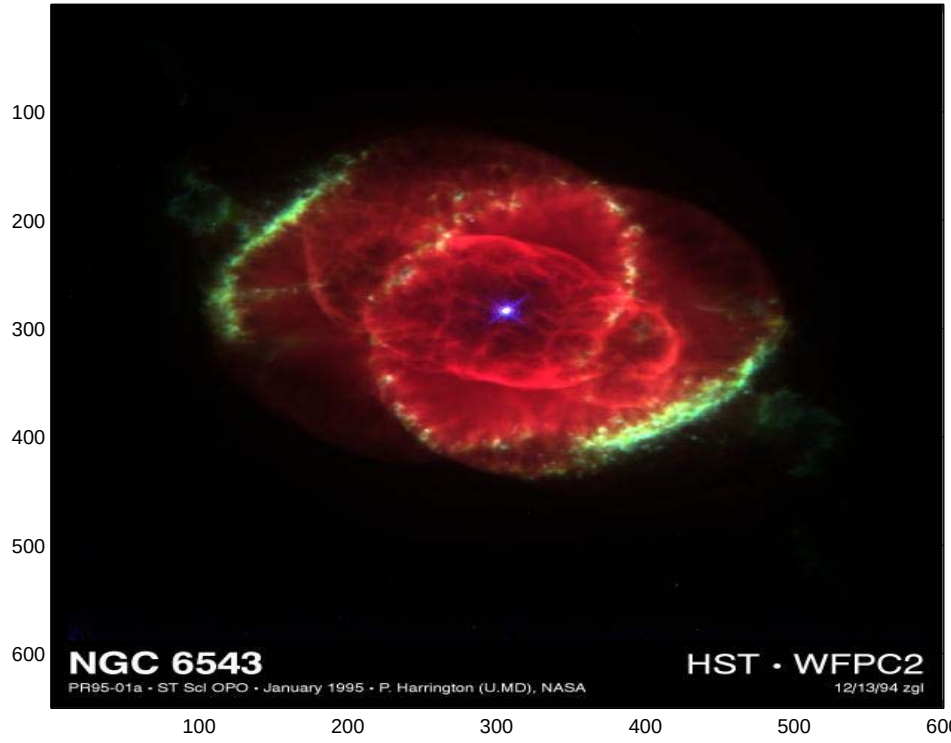
- Reshaping, reordering, and concatenating arrays using the functions `reshape`, `cat`, `permute`, and the `[]` and `'` operators
- Saving and loading `uint8` and `uint16` arrays in MAT-files using `save` and `load`. (Remember that if you are loading or saving a graphics file format image, you must use the commands `imread` and `imwrite` instead.)
- Locating the indices of nonzero elements in `uint8` and `uint16` arrays using `find`. However, the returned array is always of class `double`.
- Relational operators

Converting an 8-Bit RGB Image to Grayscale

MATLAB can perform arithmetic operations on integer data, which enables you to convert image types without first converting the numeric class of the image data.

This example reads an 8-bit RGB image into MATLAB and converts it to a grayscale image.

```
rgb_img = imread('ngc6543a.jpg'); % Load the image
image(rgb_img) % Display the RGB image
```



Now calculate the monochrome luminance by combining the RGB values according to the NTSC standard, which applies coefficients related to the eye's sensitivity to RGB colors.

```
I = .2989*rgb_img(:,:,1)...
 +.5870*rgb_img(:,:,2)...
 +.1140*rgb_img(:,:,3);
```

`I` is an intensity image with integer values ranging from a minimum of zero,

```
min(I(:))
ans =
```

```
    0
```

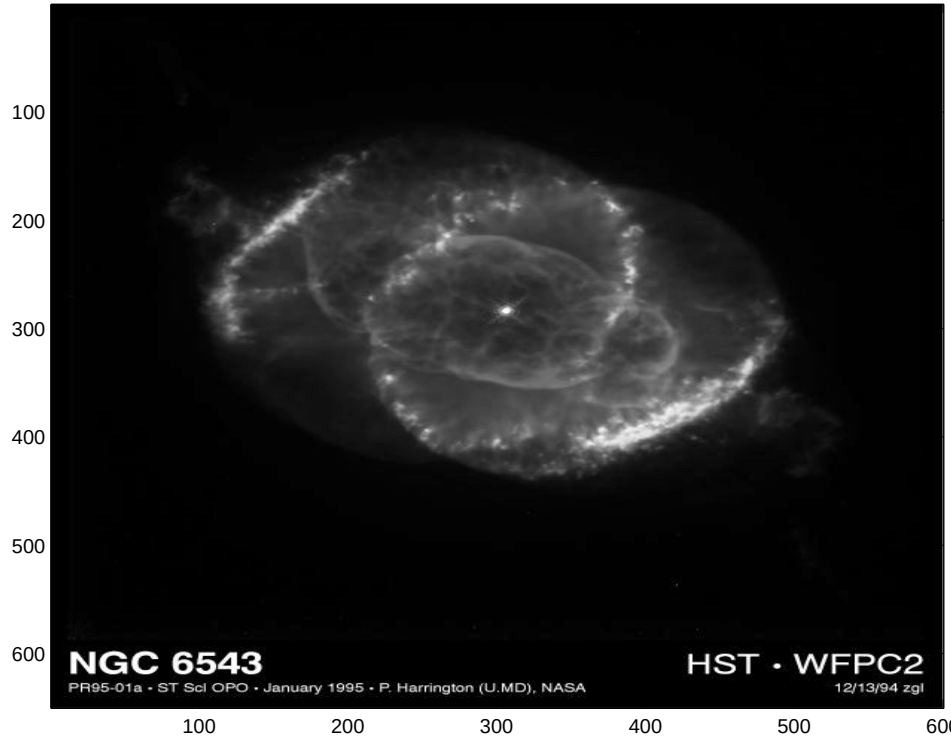
to a maximum of 255,

```
max(I(:))
ans =
 255
```

To display the image, use a grayscale colormap with 256 values. This avoids the need to scale the data-to-color mapping, which would be required if you used a colormap of a different size. You can use the `imagesc` function in cases where the colormap does not contain one entry for each data value.

Now display the image in a new figure using the gray colormap.

```
figure; colormap(gray(256)); image(I)
```



Related Information

Other colormaps with a range of colors that vary continuously from dark to light can produce usable images. For example, try `colormap(summer(256))` for a classic oscilloscope look. See `colormap` for more choices.

The `brighten` function enables you to increase or decrease the color intensities in a colormap to compensate for computer display differences or to enhance the visibility of faint or bright regions of the image (at the expense of the opposite end of the range).

Summary of Image Types and Numeric Classes

This table summarizes the way MATLAB interprets data matrix elements as pixel colors, depending on the image type and data class.

Image Type	double Data	uint8 or uint16 Data
Indexed	Image is an m-by-n array of integers in the range $[1, p]$. Colormap is a p-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n array of integers in the range $[0, p - 1]$. Colormap is a p-by-3 array of floating-point values in the range [0, 1].
Intensity	Image is an m-by-n array of floating-point values that are linearly scaled by MATLAB to produce colormap indices. Typical range of values is [0, 1]. Colormap is a p-by-3 array of floating-point values in the range [0, 1] and is typically grayscale.	Image is an m-by-n array of integers that are linearly scaled by MATLAB to produce colormap indices. The typical range of values is [0, 255] or [0, 65535]. Colormap is a p-by-3 array of floating-point values in the range [0, 1] and is typically grayscale.
RGB (Truecolor)	Image is an m-by-n-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n-by-3 array of integers in the range [0, 255] or [0, 65535].

Reading, Writing, and Querying Graphics Image Files

In this section...

Working with Image Formats

In its native form, a graphics file format image is not stored as a MATLAB matrix, or even necessarily as a matrix. Most graphics files begin with a header containing format-specific information tags, and continue with bitmap data that can be read as a continuous stream. For this reason, you cannot use the standard MATLAB I/O commands `load` and `save` to read and write a graphics file format image.

MATLAB provides special functions for reading and writing image data from graphics file formats. To read a graphics file format image use `imread`; to write a graphics file format image, use `imwrite`; to obtain information about the nature of a graphics file format image, use `imfinfo`.

This table gives a clearer picture of which MATLAB commands should be used with which image types.

Procedure	Functions to Use
Load or save a matrix as a MAT-file	`load` `save`
Load or save graphics file format image, e.g. BMP, TIFF	`imread` `imwrite`

Procedure	Functions to Use
Display any image loaded into MATLAB	`image` `imagesc`
Utilities	`imfinfo` `ind2rgb`

Reading a Graphics Image

The function `imread` reads an image from any supported graphics image file in any of the supported bit depths. Most of the images that you will read are 8-bit. When these are read into memory, MATLAB stores them as class `uint8`. The main exception to this rule is that MATLAB supports 16-bit data for PNG and TIFF images. If you read a 16-bit PNG or TIFF image, it is stored as class `uint16`.

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself can be of class `uint8` or `uint16`.

The following statement reads the image `ngc6543a.jpg` into the workspace variable `RGB` and then displays the image using the `image` function.

```
RGB = imread('ngc6543a.jpg');
image(RGB)
```

You can write (save) image data using the `imwrite` function. The statements

```
load clown % An image that is included with MATLAB
imwrite(X,map,'clown.bmp')
```

create a BMP file containing the clown image.

Writing a Graphics Image

When you save an image using `imwrite`, the default behavior is to automatically reduce the bit depth to `uint8`. Many of the images used in

MATLAB are 8-bit, and most graphics file format images do not require double-precision data. One exception to the MATLAB rule for saving the image data as `uint8` is that PNG and TIFF images can be saved as `uint16`. Because these two formats support 16-bit data, you can override the MATLAB default behavior by specifying `uint16` as the data type for `imwrite`. The following example shows writing a 16-bit PNG file using `imwrite`.

```
imwrite(I,'clown.png','BitDepth',16);
```

Subsetting a Graphics Image (Cropping)

Sometimes you want to work with only a portion of an image file or would like to break it up into subsections. You can specify the pixel coordinates of the rectangular subsection you want to work with and save it to a file from the command line. If you do not know the coordinates of the corner points of the subsection, you can choose them interactively, as the following example shows:

```
% Read demo RGB image from graphics file.
im = imread('street2.jpg');

% Display image with true aspect ratio
image(im); axis image

% Use ginput to select corner points of a rectangular
% region by pointing and clicking the mouse twice
p = ginput(2);

% Get the x and y corner coordinates as integers
sp(1) = min(floor(p(1)), floor(p(2))); %xmin
sp(2) = min(floor(p(3)), floor(p(4))); %ymin
sp(3) = max(ceil(p(1)), ceil(p(2)));   %xmax
sp(4) = max(ceil(p(3)), ceil(p(4)));   %ymax

% Index into the original image to create the new image
MM = im(sp(2):sp(4), sp(1): sp(3),:);

% Display the subsetted image with appropriate axis ratio
figure; image(MM); axis image

% Write image to graphics file.
imwrite(MM,'street2_cropped.tif')
```

If you knew what the image corner coordinates should be, you could manually define `sp` in the above example rather than using `ginput`.

You can also make MATLAB display a "rubber band box" as you interact with the image to subset it. See the code example for `rbbox` for details. For further information, see the documentation for the MATLAB functions `ginput` and `image`.

Obtaining Information About Graphics Files

The `imfinfo` function enables you to obtain information about graphics files in any of the standard formats listed above. The information you obtain depends on the type of file, but it always includes at least the following:

- Name of the file, including the directory path if the file is not in the current directory
- File format
- Version number of the file format
- File modification date
- File size in bytes
- Image width in pixels
- Image height in pixels
- Number of bits per pixel
- Image type: RGB (truecolor), intensity (grayscale), or indexed

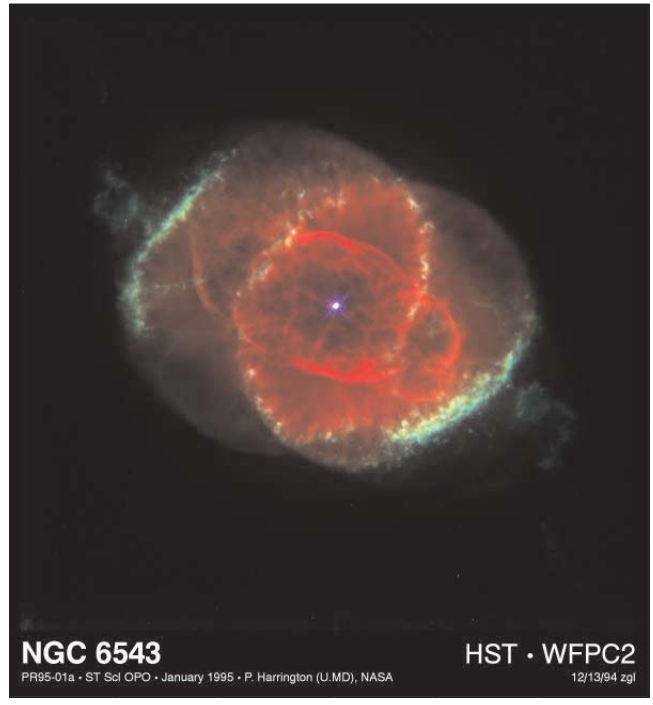
Displaying Graphics Images

In this section...

Summary of Image Types and Display Methods

To display a graphics file image, use either `image` or `imagesc`. For example, assuming `RGB` is an image,

```
figure('Position',[100 100 size(RGB,2) size(RGB,1)]);
image(RGB); set(gca,'Position',[0 0 1 1])
```



(This image was created with the support of the Space Telescope Science Institute, operated by the Association of Universities for Research in

Astronomy, Inc., from NASA contract NAs5-26555, and is reproduced with permission from AURA/STScI. Digital renditions of images produced by AURA/STScI are obtainable royalty-free. Credits: J.P. Harrington and K.J. Orkowski (University of Maryland), and NASA.)

This table summarizes display methods for the three types of images.

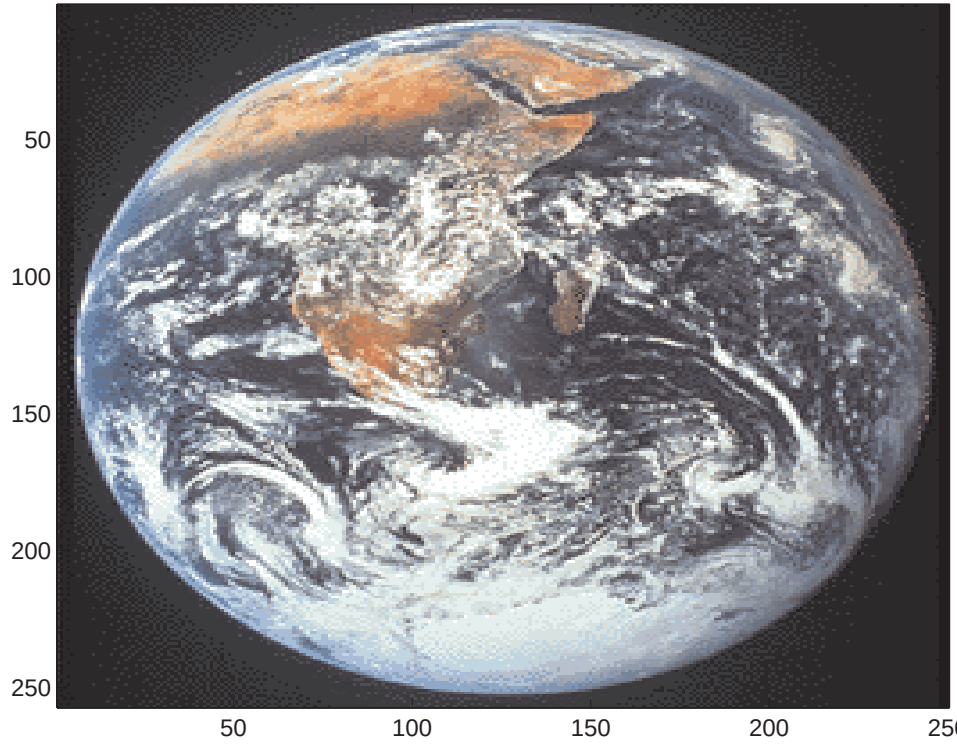
Image Type	Display Commands	Uses Colormap Colors
Indexed	`image(X);` `colormap(map)`	`Yes`
Intensity	`imagesc(I,[0 1]);` `colormap(gray)`	`Yes`
RGB (truecolor)	`image(RGB)`	`No`

Controlling Aspect Ratio and Display Size

The `image` function displays the image in a default-sized figure and axes. MATLAB stretches or shrinks the image to fit the display area. Sometimes you want the aspect ratio of the display to match the aspect ratio of the image data matrix. The easiest way to do this is with the command `axis image`.

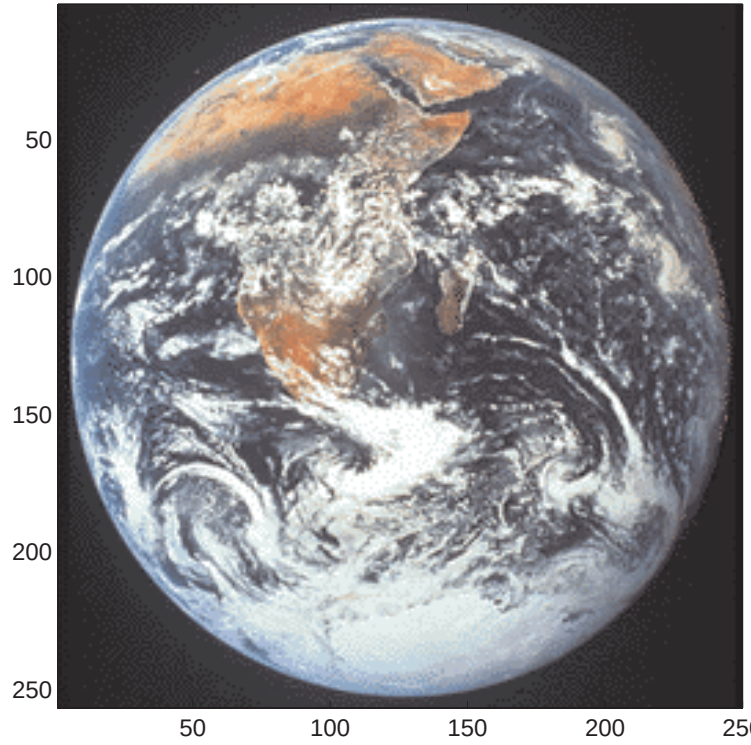
For example, these commands display the `earth` image in the `demos` directory using the default figure and axes positions.

```
load earth
image(X); colormap(map)
```



The elongated globe results from stretching the image display to fit the axes position. Use the `axis image` command to force the aspect ratio to be one-to-one.

```
axis image
```



The command `axis image` works by setting the `DataAspectRatio` property of the axes object to [1 1 1]. See `axis` and `axes` for more information on how to control the appearance of axes objects.

Sometimes you might want to display an image so that each element in the data matrix corresponds to a single screen pixel. To display an image with this one-to-one matrix-element-to-screen-pixel mapping, you need to resize the figure and axes. For example, these commands display the earth image so that one data element corresponds to one screen pixel.

```
[m,n] = size(X);
figure('Units','pixels','Position',[100 100 n m])
image(X); colormap(map)
set(gca,'Position',[0 0 1 1])
```

The figure's `Position` property is a four-element vector that specifies the figure's location on the screen as well as its size. The second statement above positions the figure so that its lower left corner is at position (100,100) on the screen and so that its width and height match the image width and height. Setting the axes position to [0 0 1 1] in normalized units creates an axes that fills the figure. The resulting picture is shown.

The Image Object and Its Properties

In this section...

Image CData

Note The commands `image` and `imagesc` create image objects. Image objects are children of axes objects, as are line, patch, surface, and text objects. Like all Handle Graphics objects, the image object has a number of properties you can set to fine-tune its appearance on the screen. The most important properties of the image object with respect to appearance are `CData`, `CDataMapping`, `XData`, `YData` and `Erasemode`. These properties are discussed in this and the following sections. For detailed information about these and all the properties of the image object, see `image`.

The `CData` property of an image object contains the data array. In the commands below, `h` is the handle of the image object created by `image`, and the matrices `X` and `Y` are the same.

```
h = image(X); colormap(map)
Y = get(h,'CData');
```

The dimensionality of the `CData` array controls whether MATLAB displays the image using colormap colors or as an RGB image. If the `CData` array is two-dimensional, then the image is either an indexed image or an intensity image, and in either case the image is displayed using colormap colors. If, on the other hand, the `CData` array is *m*-by-*n*-by-3, then MATLAB displays it as a truecolor image, ignoring the colormap colors.

Image CDataMapping

The CDataMapping property controls whether an image is indexed or intensity. An indexed image is displayed by setting the CDataMapping property to 'direct', in which case the values of the CData array are used directly as indices into the figure's colormap. When the image command is used with a single input argument, it sets the value of CDataMapping to 'direct'.

```
h = image(X); colormap(map)
get(h,'CDataMapping')
ans =

direct
```

Intensity images are displayed by setting the CDataMapping property to 'scaled'. In this case the CData values are linearly scaled to form colormap indices. The scale factors are controlled by the axes CLim property. The imagesc function creates an image object whose CDataMapping property is set to 'scaled', and it also adjusts the CLim property of the parent axes. For example,

```
h = imagesc(I,[0 1]); colormap(map)
get(h,'CDataMapping')
ans =

scaled

get(gca,'CLim')
ans =

[0 1]
```

XData and YData

The XData and YData properties control the coordinate system of the image. For an *m*-by-*n* image, the default XData is [1 n] and the default YData is [1 m]. These settings imply the following:

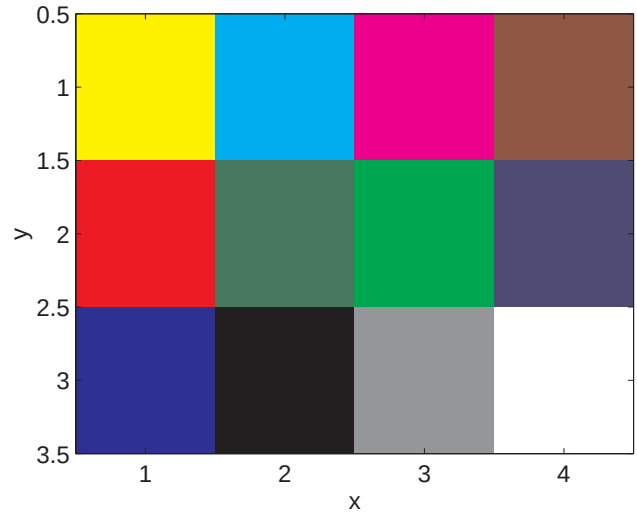
- The left column of the image has an *x*-coordinate of 1.
- The right column of the image has an *x*-coordinate of *n*.

- The top row of the image has a *y*-coordinate of 1.
- The bottom row of the image has a *y*-coordinate of *m*.

For example, the statements

```
X = [1 2 3 4; 5 6 7 8; 9 10 11 12];
h = image(X); colormap(colorcube(12))
xlabel x; ylabel y
```

produce the following picture.



The `XData` and `YData` properties of the resulting image object have the default values shown below.

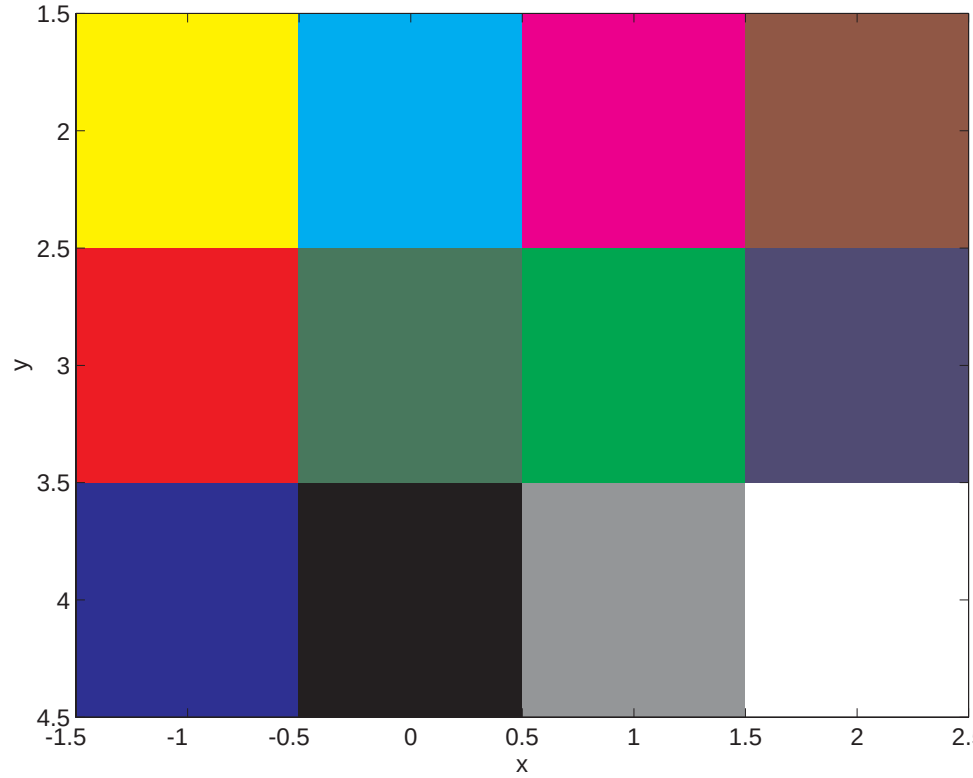
```
get(h,'XData')
ans =

     1     4

get(h,'YData')
ans =
```

```
     1     3
```

However, you can override the default settings to specify your own coordinate system. For example, the statements

```
X = [1 2 3 4; 5 6 7 8; 9 10 11 12];
image(X,'XData',[-1 2],'YData',[2 4]); colormap(colorcube(12))
xlabel x; ylabel y
```

produce the following picture.



EraseMode

The `EraseMode` property controls how MATLAB updates the image on the screen if the image object's `CData` property changes. The default setting of `EraseMode` is `'normal'`. With this setting, if you change the `CData` of the image object using the `set` command, MATLAB erases the image on the screen before redrawing the image using the new `CData` array. The erase step is a problem if you want to display a series of images quickly and smoothly.

You can achieve fast and visually smooth updates of displayed images as you change the image `CData` by setting the image object `EraseMode` property to `'none'`. With this setting, MATLAB does not take the time to erase the displayed image — it immediately draws the updated image when the `CData` changes.

Suppose, for example, that you have an *m*-by-*n*-by-3-by-*x* array `A`, containing *x* different truecolor images of the same size. You can display them dynamically with

```
h = image(A(:,:,:,1),'EraseMode','none');
for i = 2:x
    set(h,'CData',A(:,:,:,i))
end
```

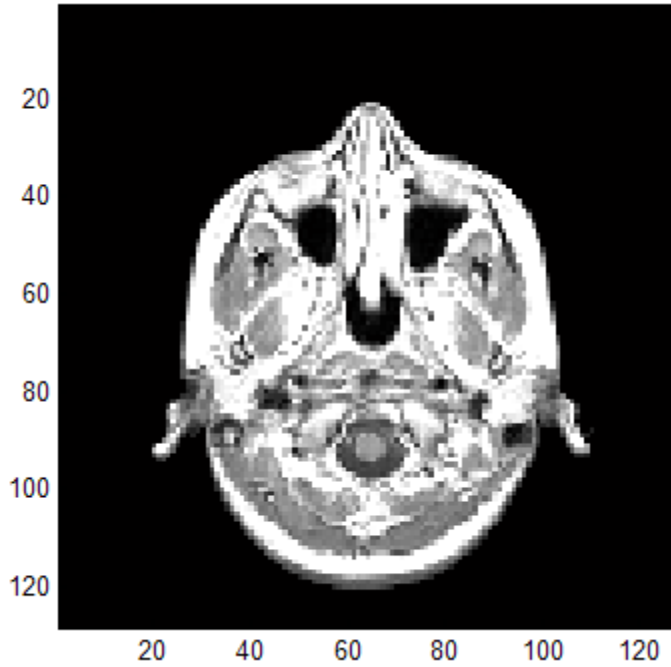
Rather than creating a new image object each time through the loop, this code simply changes the `CData` of the image object (which was created on the first line using the `image` command). Because the image `EraseMode` is set to `'none'`, changes to the `CData` do not cause the image on the screen to be erased each time through the loop.

Try the following code example that displays the `mri.mat` medical image sequence that ships with MATLAB; it sets `erasemode` to `none`, so that there is no flashing between images when`CData` is updated:

```
load mri
x = size(D);
frames = x(4);
instant = 0.05;  % Seconds to pause between frames
h = image(D(:,:,:,1),'EraseMode','none');
colormap(map); axis image;
% Color axes yellow to illustrate differences in erase modes
```

```
set(gca,'color','y')
for i = 2:frames  % Loop to the last image, replacing CData
    set(h,'CData',D(:,:,:,i))
    pause(instant)
end
for i =frames:-1:1  % Loop back to the first image
    set(h,'CData',D(:,:,:,i))
     pause(instant)
end
```

The first and last frames are the same and look like this:



Now set `erasemode` to `normal` or `background` and again execute the code in the `for` loops; you will see the yellow background flash in between image updates.

Note Setting `erasemode` to `xor` can have unpredictable and undesirable results, depending on your platform and its display system, and may not be particularly efficient.

When you are using this animation technique with larger data sets, in order to achieve maximum speed you can turn off double buffering for the figure, as follows:

```
set(gcf,'doubleBuffer','off')
```

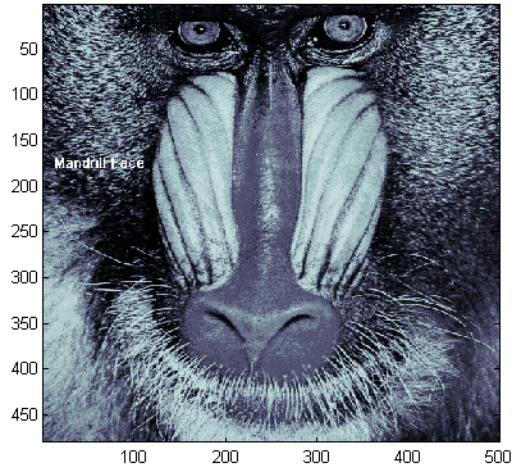
Adding Text to Images

You can use basic array indexing to rasterize text strings into an existing image, as follows:

Draw the text strings using `text`, and then capture a bitmapped version of them using `getframe`. Then find the black pixels and convert their subscripts to indexes using `sub2ind`. Use these subscripts to "paint" the text into the image into which you want to add the text string, then save that image. Here is an example using the image in the demo MAT-file `mandrill.mat`:

```
% Create the text in an axis:
t = text(.05,.1,'Mandrill Face', ...
         'FontSize',12, 'FontWeight','demi');

% Capture the text from the screen:
F = getframe(gca,[10 10 200 200]);

% Close the figure:
close

% Select any plane of the resulting RGB image:
c = F.cdata(:,:,1);

% Note: If you have Image Processing Toolbox installed,
% you can convert the RGB data from the frame to black or white:
% c = rgb2ind(F.cdata,2);

% Determine where the text was (black is 0):
```

```
[i,j] = find(c == 0);

% Read in or load the image that is to contain the text:
load mandrill

% Use the size of that image, plus the row/column locations
% of the text, to determine locations in the new image:
ind = sub2ind(size(X),i,j);

% Index into new image, replacing pixels with white:
X(ind) = uint8(255);

% Display and color the new image:
imagesc(X)
axis image
colormap(bone)
```



Additional Techniques for Fast Image Updating

To increase the rate at which MATLAB can update the `CData` property of an image object you can optimize `CData` and set some related figure and axes properties:

- Use the smallest datatype possible. Using a `uint8` datatype for your image will be faster than using a `double` datatype.

  Part of the process of setting the image's `CData` property includes copying the matrix for the image's use. The overall size of the matrix is dependent on the size of its individual elements. Using smaller individual elements (i.e., a smaller datatype) decreases matrix size,and reduces the amount of time needed to copy the matrix.

- Use the smallest acceptable matrix.

  If the speed at which the image is displayed is your highest priority, you may need to compromise on the size and quality of the image. Again, decreasing the size will reduce the time needed to copy the matrix.

- Make the axes exactly the same size (in pixels) as the `CData` matrix.

  Maintaining a one-to-one correspondence between the data and the onscreen pixels will eliminate the need for interpolation. For example:

  ```
  set(gca,'Units','pixels')
  pos = get(gca,'Position')
  width = pos(3);
  height = pos(4);
  ```

  When the size of your `CData` exactly equals `[width height]`, each element of the array corresponds directly to a pixel. Otherwise, MATLAB must interpolate the values in the "CData" array, so it will fit the axes at their current size.

- Set the limit mode properties (`XLimMode` and `YLimMode`) of your axes to `manual`.

  If they are set to `auto`, then every time an object (such as an image, line, patch, etc.) changes some aspect of its data, the axes must recalculate its related properties. For example, if you specify

  ```
  image(firstimage);
  set(gca, 'xlimmode','manual',...
  'ylimmode','manual',...
  'zlimmode','manual',...
  'climmode','manual',...
  'alimmode','manual');
  ```

the axes will not recalculate any of the limit values before redrawing the image.

- Set the figure's `DoubleBuffer` property to `off`.

  ```
  set(gcf,'doublebuffer','off');
  ```

  Set the `DoubleBuffer` property `on` for to obtain flicker-free animation. However, to maximize rendering speed, set `DoubleBuffer` to `off`.

- Alternately, you might consider using a `movie` object if the main point of your task is to simply display a series of images onscreen.

  The MATLAB `movie` object utilizes underlying system graphics resources directly, instead of executing MATLAB object code. This is faster than repeatedly setting an image's `CData` property, as outlined above.

Printing Images

When you set the axes `Position` to `[0 0 1 1]` so that it fills the entire figure, the aspect ratio is not preserved when you print because MATLAB adjusts the figure size when printing according to the figure's `PaperPosition` property. To preserve the image aspect ratio when printing, set the figure's `PaperPositionMode` to `'auto'` from the command line.

```
set(gcf,'PaperPositionMode','auto')
print
```

When `PaperPositionMode` is set to `'auto'`, the width and height of the printed figure are determined by the figure's dimensions on the screen, and the figure position is adjusted to center the figure on the page. If you want the default value of `PaperPositionMode` to be `'auto'`, enter this line in your `startup.m` file.

```
set(0,'DefaultFigurePaperPositionMode','auto')
```

Printed images may not always be the same size as they are on your monitor. The size depends on accurately specifying the numbers of pixels per inch that you monitor is displaying.

To specify the pixels-per-inch on your display, you need to do the following (in Windows):

1. Go into your Display Properties by right-clicking on an empty space on your desktop and choose Properties.
2. Click the Settings pane
3. Click the Advanced pushbutton and choose the General pane
4. Switch DPI setting to Custom setting and hold a real ruler up to the picture of the ruler on the screen and drag until they match.

Until you do this, neither MATLAB nor any other software can determine how big images on the screen are, and printed images cannot match the size.

Note that on the Macintosh platform, pixels per inch is hard-coded to 72.

Converting the Data or Graphic Type of Images

Converting between data types changes the way MATLAB interprets the image data. If you want the resulting array to be interpreted properly as image data, you need to rescale or offset the data when you convert it. (See the earlier sections "Image Types" on page 6-5 and "8-Bit and 16-Bit Indexed Images" on page 6-10 for more information about offsets.)

For certain operations, it is helpful to convert an image to a different image type. For example, if you want to filter a color image that is stored as an indexed image, you should first convert it to RGB format. To do this efficiently, use the `ind2rgb` function, which originated in Image Processing Toolbox. When you apply the filter to the RGB image, MATLAB filters the intensity values in the image, as is appropriate. If you attempt to filter the indexed image, MATLAB simply applies the filter to the indices in the indexed image matrix, and the results may not be meaningful.

You can also perform certain conversions just using MATLAB syntax. For example, if you want to convert a grayscale image to RGB, you can concatenate three copies of the original matrix along the third dimension.

```
RGB = cat(3,I,I,I);
```

The resulting RGB image has identical matrices for the red, green, and blue planes, so the image is displayed as shades of gray.

Sometimes you will want to change the graphics format of an image, perhaps for compatibility with another software product. This process is very straightforward. For example, to convert an image from a BMP to a PNG, load the BMP using `imread`, set the data type to `uint8`, `uint16`, or `double`, and then save the image using `imwrite`, with `'PNG'` specified as your target format. See `imread` and `imwrite` for the specifics of which bit depths are supported for the different graphics formats, and for how to specify the format type when writing an image to file.

7

Printing and Exporting

Choosing a Printer Driver (p. 7-85)	Factors to consider when using a nondefault print driver, and information specific to drivers supported by MATLAB
Troubleshooting (p. 7-94)	Solutions to frequently asked questions and common problems encountered while printing or exporting graphics

Overview of Printing and Exporting

In this section...

Print and Export Operations

There are four basic operations that you can perform in printing or transferring figures you've created in MATLAB to specific file formats for other applications to use.

Operation	Description
Print	Send a figure from the screen directly to the printer.
Print to File	Write a figure to a PostScript file to be printed later.
Export to File	Export a figure in graphics format to a file, so that you can import it into an application.
Export to Clipboard	Copy a figure to the Windows clipboard, so that you can paste it into an application.

Graphical User Interfaces

In addition to typing MATLAB commands, you can use interactive tools for either Microsoft Windows or UNIX to print and export graphics. The table below lists the GUIs available for doing this and explains how to open them from figure windows.

Dialog Box	How to Open	Description
Print (Windows and UNIX)	**File > Print** or `printdlg` function	Send figure to the printer, select the printer, print to file, and several other options
Print Preview	**File > Print Preview** or `printpreview` function	View and adjust the final output
Export	**File > Export**	Export the figure in graphics format to a file
Copy Options	**Edit > Copy Options**	Set format, figure size, and background color for Copy to Clipboard
Figure Copy Template	**File > Preferences**	Change text, line, axes, and UI control properties

You can open the Print and Print Preview dialog boxes from an M-file or from the command line with the `printdlg` and `printpreview` functions.

Command Line Interface

You can print a MATLAB figure from the command line or from an M-file. Use the `set` function to set the properties that control how the printed figure looks. Use the `print` function to specify the output format and start the print or export operation.

Note Printed output from MATLAB and Print Previews of it are not guaranteed to duplicate the look of figures on your display screen in every detail. Many factors, including the complexity of the figure, available fonts, and whether a native printer driver or a driver built in to MATLAB is used, affect the final output and can cause printed output to differ from what you see on your screen.

Modifying Properties with set

The `set` function changes the values of properties that control the look of a figure and objects within it. These properties are stored with the figure; some

are also properties of children such as axes or annotations. When you change one of the properties, the new value is saved with the figure and affects the look of the figure each time you print it until you change the setting again.

To change the print properties of the current figure, the `set` command has the form

```
set(gcf, 'Property1', value1, 'Property2', value2, ...)
```

where `gcf` is a function call that returns the handle of the current figure, and each property-value pair consists of a named property followed by the value to which the property is set.

For example,

```
set(gcf, 'PaperUnits', 'centimeters', 'PaperType', 'A4', ...)
```

sets the units of measure and the paper size. “Changing a Figure’s Settings” on page 7-43 describes commonly used print properties. The Figure Properties reference page contains a complete list of the properties.

Examining Properties with get

You can also use the `get` function to retrieve the value of a specific property.

```
a = get(gcf, 'Property')
```

Note You can also peruse and modify figure and other object properties with the Property Inspector GUI, which you can open with the `inspect` command. To open the current figure in the Property Inspector, type `inspect(gcf)`

Printing and Exporting with print

The `print` function performs any of the four actions shown in the table below. You control what action is taken, depending on the presence or absence of certain arguments.

Action	Print Command
Print a figure to a printer	`print`
Print a figure to a file for later printing	`print` *`filename`*
Copy a figure in graphics format to the clipboard	`print -dfileformat`
Export a figure to a graphics format file that you can later import into an application	`print -dfileformat filename`

You can also include optional arguments with the `print` command. For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, use

```
print -f2 -r600 -depsc spline2d
```

The functional form of this command is

```
print('-f2', '-r600', '-depsc', 'spline2d');
```

Printing on UNIX without a Display

If you run MATLAB with the `-nodisplay` startup option, or run without the `DISPLAY` environment variable set, you can use most `print` options that apply to the UNIX platform, but some restrictions apply. See XXX for details.

Specifying Parameters and Options

The table below lists parameters you can modify for the figure to be printed or exported. To change one of these parameters, use the Print Preview or the UNIX Print dialog box, or use the `set` or `print` function.

See “Changing a Figure’s Settings” on page 7-43 for more detailed instructions.

Parameter	Description
Figure size	Set size of the figure on printed page
Figure position	Set position of figure on printed page
Paper size	Select printer paper, specified by dimension or type
Paper orientation	Specify way figure is oriented on page
Position mode	Specify figure position yourself or have MATLAB determine position automatically
Graphics format	Select format for exported data (e.g., EPS, JPEG)
Resolution	Specify how finely your figure is to be sampled
Renderer	Select method (algorithm) for drawing graphics
Renderer mode	Specify the renderer yourself or have MATLAB automatically determine which renderer to use
Axes tick marks	Keep axes tick marks and limits as shown or have MATLAB adjust depending on figure size
Background color	Keep background color as shown on screen or force it to white
Line and text color	Keep line and text objects as shown on screen or print them in black and white
UI controls	Show or hide all user interface controls in figure
Bounding box	Leave space between outermost objects in plot and edges of its background area
CMYK	Automatically convert RGB values to CMYK values
Character set encoding	Select character set for PostScript printers

Default Settings and How to Change Them

If you have not changed the default print and export settings, MATLAB prints or exports the figure

- 8-by-6 inches with no window frame

- Centered, in portrait format, on 8.5-by-11 inch paper if available
- Using white background color for the figure and axes
- Scaling ticks and limits of the axes to accommodate the printed size

Setting Defaults for a Figure

In general, to change the property settings for a specific figure, follow the instructions given in the section “Changing a Figure’s Settings” on page 7-43.

Any settings you change with the Print Preview and Print dialog boxes or with the `set` function are saved with the figure and affect each printing of the figure until you change the settings again.

The settings you change with the **Figure Copy Template Preferences** and **Copy Options Preferences** panels alter the figure as it is displayed on the screen.

Setting Defaults for the Session

MATLAB enables you to set the session defaults for figure properties. Set the session default for a property using the syntax

```
set(0, 'DefaultFigurepropertyname', 'value')
```

where *`propertyname`* is one of the named figure properties. This example sets the paper orientation for all subsequent print operations in the current MATLAB session.

```
set(0, 'DefaultFigurePaperOrientation', 'landscape')
```

The Figure Properties reference page contains a complete list of the properties.

To see what default properties you can set that will be applied to all subsequent figures in the same MATLAB session, type

```
set(0,'default')
```

To see their current settings, type

```
get(0,'default')
```

Setting Defaults Across Sessions

MATLAB enables you to set the session-to-session defaults for figure properties, the print driver, and the `print` function.

Print Device and Print Command. Set the default print driver and the default print command in your `printopt.m` file. This file contains instructions for changing these settings and for displaying the current defaults. Open `printopt.m` in your editor by typing the command

```
edit printopt
```

Scroll down about 40 lines until you come to this comment line:

```
%---> Put your own changes to the defaults here (if needed)
```

Add your changes after that line. For example, to change the default driver, first find the line that sets `dev`, and then replace the text string with an appropriate value. So, to set the default driver to HP LaserJet III, modify the line to read

```
dev = '-dljet3';
```

For the full list of values for `dev`, see the Drivers section of the `print` reference page.

Note If you set `dev` to be a graphics format, such as `-djpeg`, MATLAB exports the figure to that type of file rather than printing it.

Figure Properties. Set the session-to-session default for a property by including commands like the following in your `startup.m` file:

```
set(0, 'DefaultFigure'propertyname', 'value')
```

where *`propertyname`* is one of the named figure properties. For example,

```
set(0, 'DefaultFigureInvertHardcopy', 'off')
```

keeps the figure background in the screen color.

This is the same command you use to change a session default, except by adding it to your `startup.m` file, it executes automatically every time MATLAB is invoked.

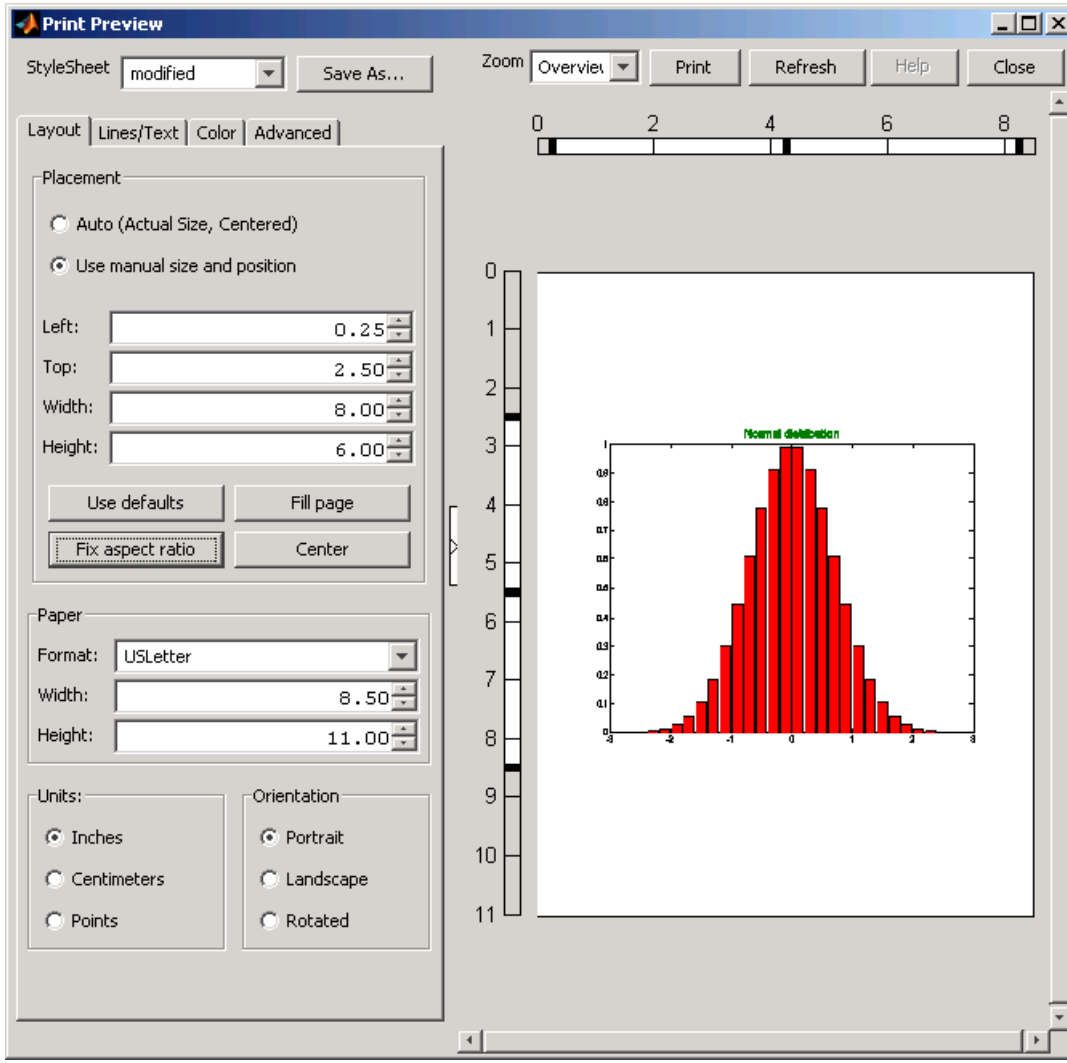
Note Options you specify in arguments to the `print` command override properties set using MATLAB commands or the Print Preview dialog box, which in turn override any MATLAB default settings specified in `printopt.m` or `startup.m`.

How to Print or Export

In this section...

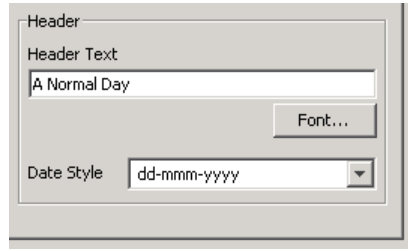
Using Print Preview

Before you print or export a figure, preview the image by selecting **Print Preview** from the figure window's **File** menu. If necessary, you can use the `set` function to adjust specific characteristics of the printed or exported figure. Adjustments that you make in the Print Preview dialog also set figure properties; these changes can affect the output you get should you print the figure later with the `print` command. See "Changing a Figure's Settings" on page 7-43 for details.

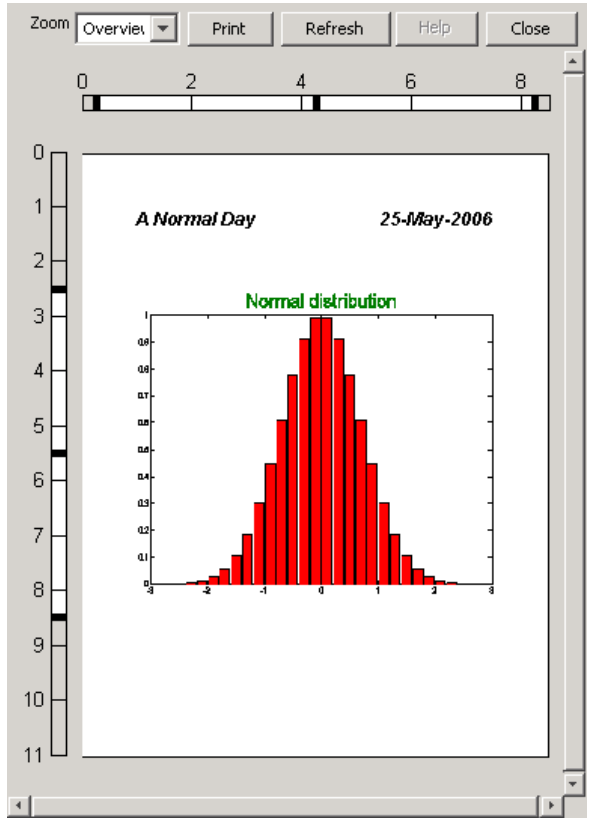


Adding a Header to the Printed Page

You can add a header to the page you are about to print by clicking the **Lines/Text** tab at the top of the Print Preview dialog box. At the bottom of that panel are the **Header** controls, as shown here:



The print header includes any text you want to appear at the top of the printed page. It can also include the current date. In the **Header Text** edit box, enter the text of the header. Under `Date Type`, select from a number of possible formats with which to display the current date and/or time. The default is to include no date. Click the **Font** button to change the font, font style, font size, or script type for the header text and date format. If you don't see the header as you specified it, click the **Refresh** button over the preview pane. A page containing a header plus date in bold italics is shown in the preview below:



Click **Print** to open the standard print dialog box to print the page. Click **Close** to close the dialog box and apply these settings to your figure.

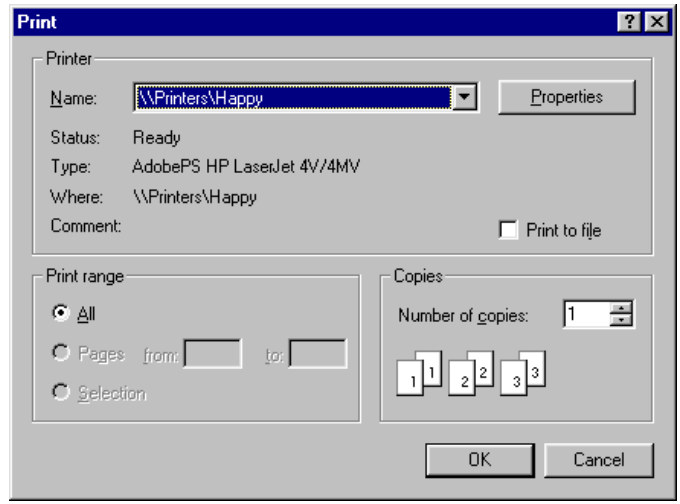
Printing a Figure

This section tells you how to print your figure to a printer:

- “Printing with the Print GUI on Windows” on page 7-15
- “Printing with the Print GUI on UNIX” on page 7-16
- “Printing Using MATLAB Commands” on page 7-19

Printing with the Print GUI on Windows

MATLAB for Windows uses the standard Windows Print dialog box, which most Windows software products share. To open the Windows Print dialog box, select **Print** from the figure window's **File** menu or click the **Print** button in the Print Preview dialog box.



- To print a figure, first select a printer from the list box, then click **OK**.
- To save it to a file, click the **Print to file** check box, click **OK**, and when the Print to File window appears, enter the filename you want to save the figure to. MATLAB creates the file in your current working directory.

Settings you can change in the Windows Print dialog box are as follows:

Properties. To make changes to settings specific to a printer, click the **Properties** button. This opens the Windows Document Properties window.

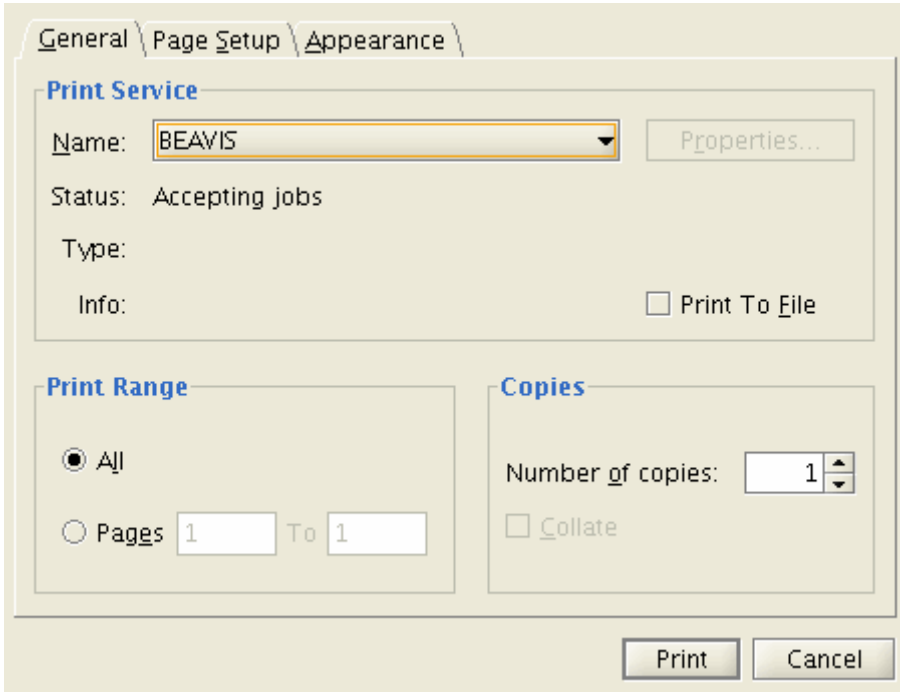
Print range. You can only select **All** in this panel. The selection does not affect your printed output.

Copies. Enter the number of copies you want to print.

You can also open the Print dialog programmatically via the `printdlg` function.

Printing with the Print GUI on UNIX

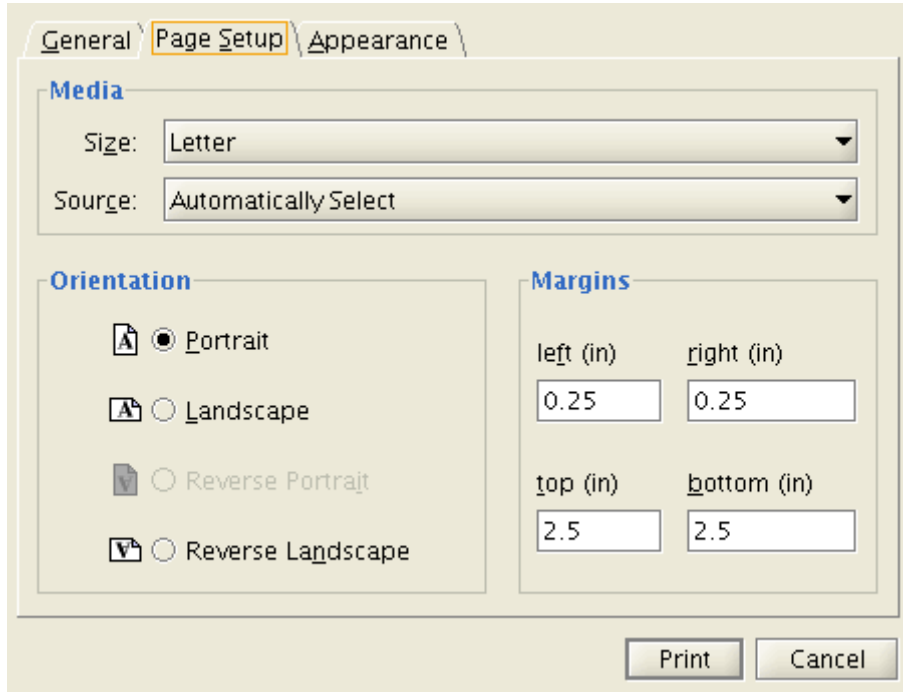
MATLAB for UNIX has a Print dialog box containing three tabs. To open the Print dialog box, select **Print** from the figure window's **File** menu. It opens showing the **General** tab's contents:



To print a figure, click the **Name** button under **Print Service** and select a printer from the list box.

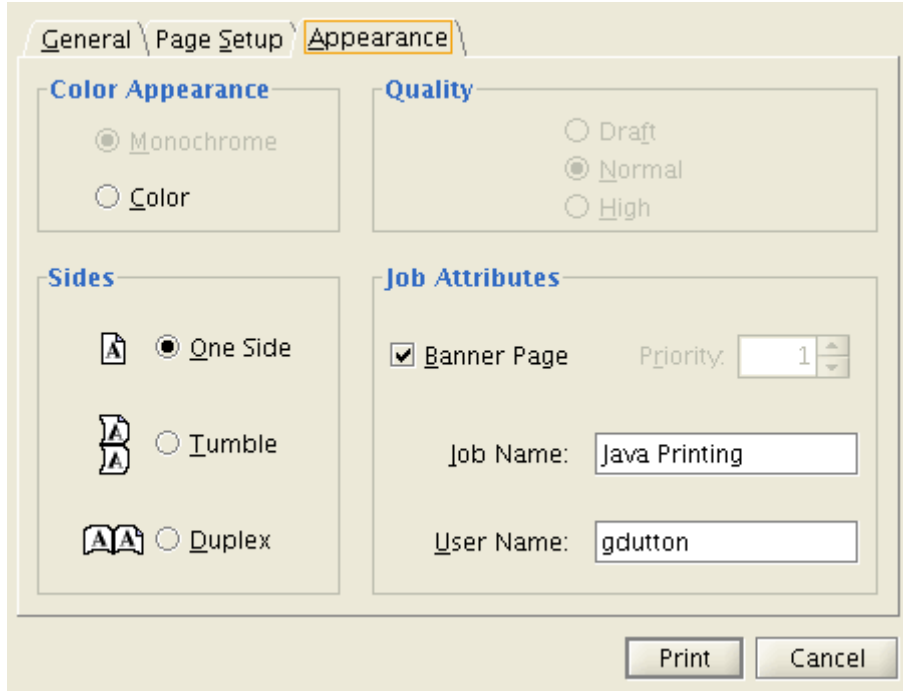
Note MATLAB assumes printers accessed from the Print dialog are PostScript-enabled. If you want to print to a non-PostScript device, you will need to use **File > Save As** and specify the **Save as type** or issue a `print` command specifying the appropriate driver with the `-d` flag.

The **Page Setup** tab on the Print dialog looks like this:



You can set paper characteristics and margins with the controls on this tab. You might want to use the Print Preview dialog instead, however, as it allows you to do the same things and gives you visual feedback at the same time. For details, see “Using Print Preview” on page 7-11.

The **Appearance** Print dialog tab lets you control several aspects of your print jobs:



The **Appearance** options include Duplex and Tumble printing, whether a banner page should precede the printed page, whether to print in color, and what quality of printing to use. You can also use **Print Preview** to control color.

Related settings in the Print Preview dialog box include

Printing in Color. Depending on the capabilities of the printer you are using, you can print in black and white, grayscale, or color by selecting the appropriate button in the **Color Scale** panel of the Print Preview **Color** tab. You can also choose a background color that is the same or different from the figure's color.

Figure Size and Position on Printed Page. If you want the printed plot to have the same size as it does on your screen, select **Auto (Actual Size, Centered)** on the **Layout** tab. If you want the printed output to have a specific size, select **Use manual size and position**.

See “Setting the Figure Size and Position” on page 7-47 for more information.

Axes Limits and Ticks. To force MATLAB to print the same number of ticks and the same limit values for the axes as used on the screen, select **Keep screen limits and ticks** on the **Advanced** tab of the Print Preview dialog box. To let MATLAB scale the limits and ticks of the axes based on the size of the printed figure, select **Recompute limits and ticks**.

See “Setting the Axes Ticks and Limits” on page 7-60 for more information.

Printing Using MATLAB Commands

Use the `print` function to print from the MATLAB command line or from a program. See “Printing and Exporting with print” on page 7-5 for more information.

To send the current or most recently active figure to a printer, simply type

```
print
```

The Printing Options table on the `print` reference page shows a full list of options that you can use with the `print` function. For example, the following command prints Figure No. 2 with 600 dpi resolution, using the Canon BubbleJet BJ200 printer driver:

```
print -f2 -r600 -dbj200
```

Printing to a File

Instead of sending your figure to the printer right now, you have the option of “printing” it to a file, and then sending the file to the printer later on. You can also append additional figures to the same file using the `print` command.

This section tells you how to save your figure to a file:

Printing to a File with the Print GUI on Windows

1. To open the Print dialog box, select **Print** from the figure window's **File** menu.
2. Select the check box labeled **Print to file**, and click the **OK** button.
3. The **Print to file** dialog box appears, allowing you to specify the output directory and filename.

Printing to a File with the Print GUI on UNIX

1. To open the Print dialog box, select **Print** from the figure window's **File** menu.
2. Select the radio button labeled **File**, and either fill in or browse for the directory and filename.

Printing to a File Using MATLAB Commands

To print the figure to a PostScript file, type

```
print filename
```

If you don't specify the filename extension, MATLAB uses an extension that is appropriate for the print driver being used.

You can also include an `-options` argument when printing to a file. For example, to append the current figure to an existing file, type

```
print -append filename
```

The only way to append to a file is by using the `print` function. There is no dialog box that enables you to do this.

Note If you print a figure to a file, the file can only be printed and cannot be imported into another application. If you want to create a figure file that you can import into an application, see the next section, "Exporting to a File"

Appending Additional Figures to a File. Once you have printed one figure to a PostScript file, you can append other figures to that same file using the `-append` option of the `print` function. You can only append using the `print` function.

This example prints Figure No. 2 to PostScript file `myfile.ps`, and then appends Figure No. 3 to the end of the same file:

```
print -f2 myfile
print -f3 -append myfile
```

Exporting to a File

Export a figure in a graphics format to a file if you want to import it into another application, such as a word processor. You can export to a file from the Windows or UNIX Export Setup dialog box or from the command line.

This section tells you how to export your figure to a file:

- "Using the Export Setup GUI" on page 7-21
- "Exporting Using MATLAB Commands" on page 7-28

It also covers

- "Exporting with getframe" on page 7-29
- "Saving Multiple Figures to an AVI File" on page 7-30
- "Importing MATLAB Graphics into Other Applications" on page 7-30

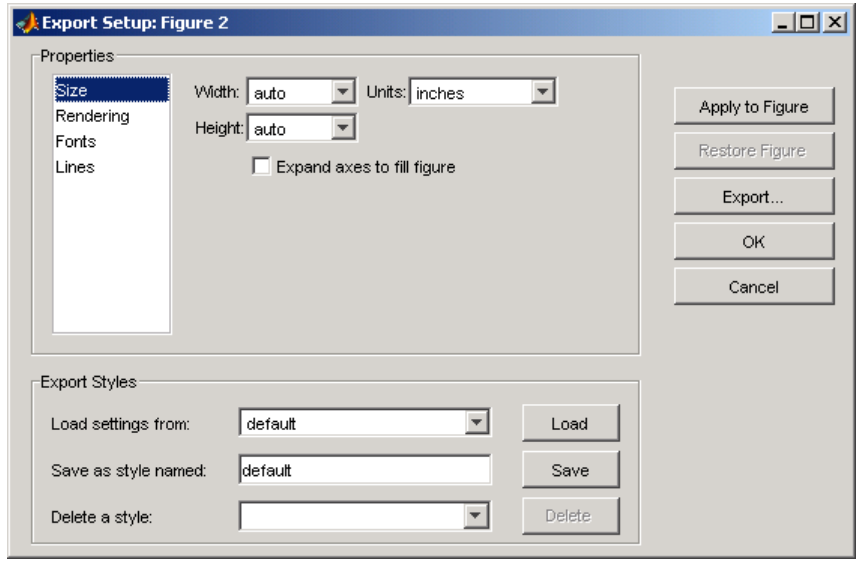
For further information, see "Choosing a Graphics Format" on page 7-73.

Using the Export Setup GUI

MATLAB displays the export Setup GUI when you select **Export Setup** from the **File** menu of a figure window. This GUI has four dialog boxes that enable you to adjust the size, rendering, font, and line appearance of your figure prior to exporting it. You select each of these dialog boxes by clicking **Size**, **Rendering**, **Fonts**, or **Lines** from the**Properties** list. For a description of each dialog box, see

Adjusting the Figure Size

Click **Size** in the Export Setup dialog box to display this dialog box.

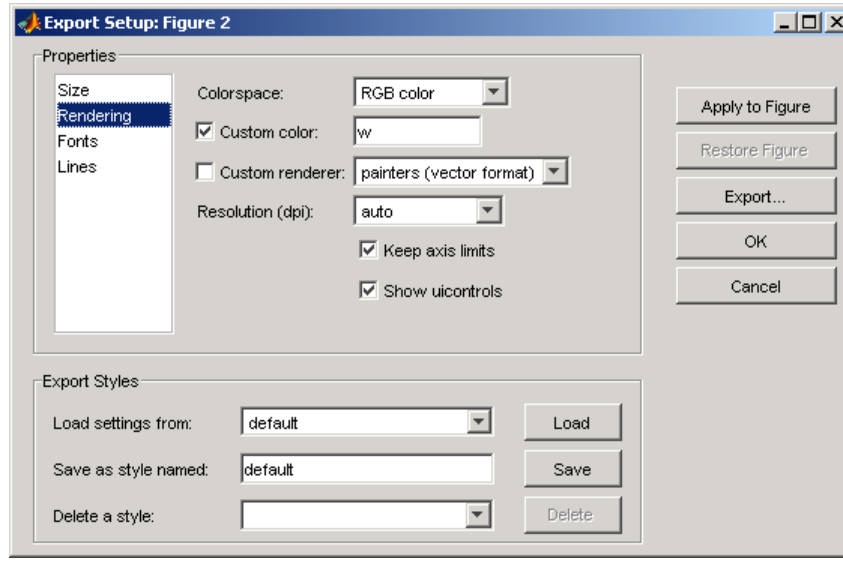


The Size dialog box modifies the size of the figure as it will appear when imported from the export file into your application. If you leave the **Width** and **Height** settings on `auto`, the figure remains the same size as it appears on your screen. You can change the size of the figure by entering new values in the **Width** and **Height** text boxes and then clicking **Apply to Figure**. To go back to the original settings, click **Restore Figure**.

To save any settings that you change, or to load settings that you used earlier, see "Saving and Loading Settings" on page 7-27.

Changing the Rendering

Click **Rendering** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Colorspace. Use the drop-down list to select a colorspace. Your choices are

- `Black and white`
- `Grayscale`
- `RGB color`
- `CMYK color`

Custom Color. Click the check box and enter a color to be used for the figure background. Valid entries are

- `white`, `yellow`, `magenta`, `red`, `cyan`, `green`, `blue`, or `black`
- Abbreviated name for the same colors — `w`, `y`, `m`, `r`, `c`, `g`, `b`, `k`
- Three-element RGB value — See the help for `colorspec` for valid values. Examples: `[1 0 1]` is magenta. `[0 .5 .4]` is a dark shade of green.

Custom Renderer. Click the check box and select a renderer from the drop-down list:

- `painters (vector format)`
- `OpenGL (bitmap format)`
- `Z-buffer (bitmap format)`

Resolution. You can select one of the following from the drop-down list:

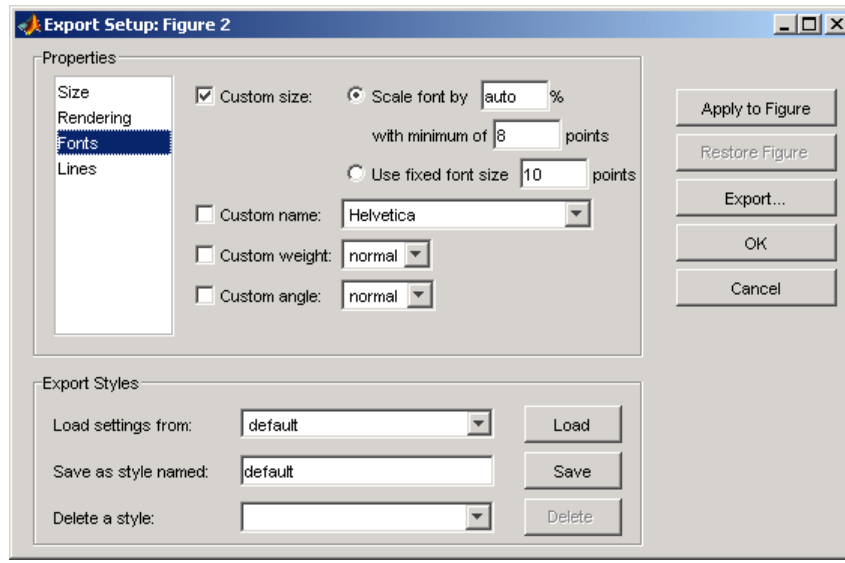
- `Screen` — The same resolution as used on your screen display
- A specific numeric setting — 150, 300, or 600 dpi
- `auto` — MATLAB selects a suitable setting

Keep axis limits. Click the check box to keep axis tick marks and limits as shown. If unchecked, have MATLAB adjust depending on figure size.

Show uicontrols. Click the check box to show all user interface controls in the figure. If unchecked, hide user interface controls.

Changing Font Characteristics

Click **Fonts** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Custom Size. Click the check box and use the radio buttons to select a relative or absolute font size for text in the figure.

- **Scale font by N %** — Increases or decreases the size of all fonts by a relative amount, N percent. Enter the word `auto` to have MATLAB select the appropriate font size.
- **With minimum of N points** — You can specify a minimum font size when scaling the font by a percentage.
- **Use fixed font size N points** — Sets the size of all fonts to an absolute value, N points.

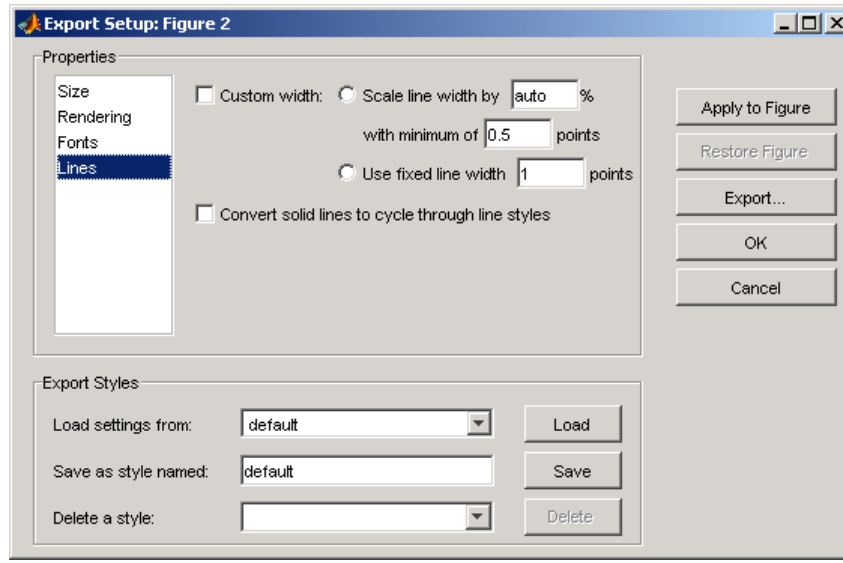
Custom Name. Click the check box and use the drop-down list to select a font name from those offered in the drop-down list.

Custom Weight. Click the check box and use the drop-down list to select the weight or thickness to be applied to text in the figure. Choose from `normal`, `light`, `demi`, or `bold`.

Custom Angle. Click the check box and use the drop-down list to select the angle to be applied to text in the figure. Choose from `normal`, `italic`, or `oblique`.

Changing Line Characteristics

Click **Lines** in the Export Setup dialog box to display this dialog box.



You can change the settings in this dialog box as follows:

Custom width. Click the check box and use the radio buttons to select a relative or absolute line size for the figure.

- **Scale line width by N %** — Increases or decreases the width of all lines by a relative amount, N percent. Enter the word `auto` to have MATLAB select the appropriate line width.

- **With minimum of N points** — Specify a minimum line width when scaling the font by a percentage.
- **Use fixed line width N points** — Sets the width of all lines to an absolute value, N points.

Convert solid lines to cycle through line styles. When colored graphics are imported into an application that does not support color, lines that could formerly be distinguished by unique color are likely to appear the same. For example, a red line that shows an input level and a blue line showing output both appear as black when imported into an application that does not support colored graphics.

Clicking this check box causes MATLAB to export lines using different line styles, such as solid, dotted, or dashed lines rather than differentiating between lines based on color.

Saving and Loading Settings

If you think you might use these export settings at another time, you can save them now and reload them later. At the bottom of each Export Setup dialog box, there is a panel labeled **Export Styles**. To save your current export styles, type a name into the **Save as style named** text box, and then click **Save**.

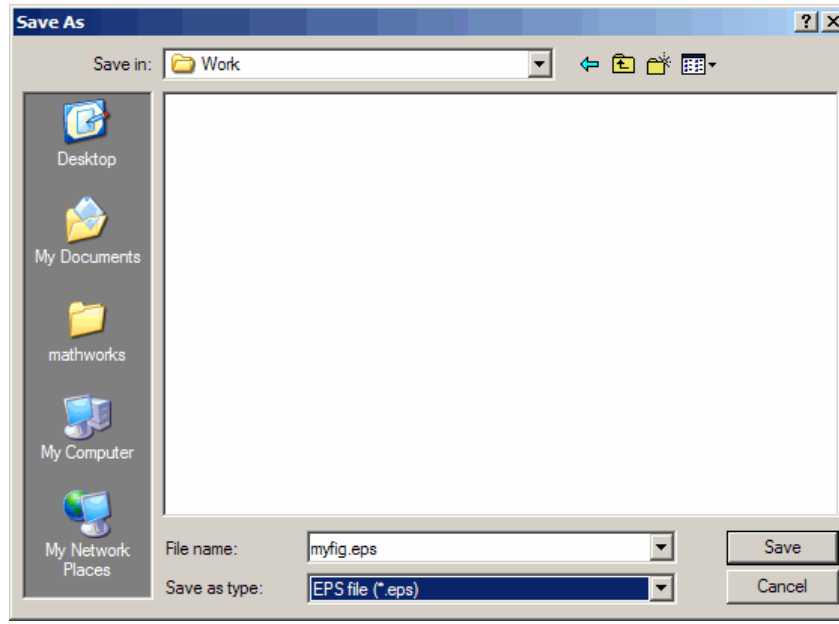
If you then click the **Load settings from** drop-down list, the name of the style you just saved appears among the choices of export styles you can load. To load a style, select one of the choices from this list and then click **Load**.

To delete any style you no longer have use for, select that style name from the **Delete a style** drop-down list and click **Delete**.

Exporting the Figure

When you finish setting the export style for your figure, you can export the figure to a file by clicking the **Export** button on the right side of any of the four Export Setup dialog boxes. As new window labeled **Save As** opens.

Select a directory to save the file in from the **Save in** list at the top. Select a file type for your file from the **Save as type** drop-down list at the bottom, and then enter a file name in the **File name** text box. Click the **Save** button to export the file.



For information on the graphics file formats supported by MATLAB, see "Choosing a Graphics Format" on page 7-73.

Exporting Using MATLAB Commands

Use the `print` function to print from the MATLAB command line or from a program. See "Printing and Exporting with print" on page 7-5 for basic information on printing from the command line.

To export the current or most recently active figure, type

```
print -dfileformat filename
```

where `fileformat` is a graphics format supported by MATLAB and `filename` is the name you want to give to the export file. MATLAB selects the filename extension, if you don't specify it.

You can also specify a number of options with the `print` function. These are shown in the Printing Options table on the `print` reference page.

For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution and using the EPS color graphics format, type

```
print -f2 -r600 -depsc spline2d
```

Graphics file formats are explained in more detail in the sections "Choosing a Graphics Format" on page 7-73 and "Description of Selected Graphics Formats" on page 7-80.

Exporting with getframe

You can use the `getframe` function with `imwrite` to export a graphic. `getframe` is often used in a loop to get a series of frames (figures) with the intention of creating a movie. Note that no matter what the intrinsic resolution of the graphics might be, `getframe` only captures them at screen resolution.

Some of the benefits of using this export method over using `print` are

- You can use `getframe` to capture a portion of the figure, rather than the whole figure.
- `imwrite` offers greater flexibility for setting format-specific options, such as the bit depth and compression.

The drawbacks of using this method are

- `imwrite` uses built-in MATLAB formats only
- `getframe` and `imwrite` are limited to screen resolution

.Consequently, you do not have access to the Ghostscript formats available to you when exporting with the `print` function or **Export** menu.

How to Use getframe and imwrite. Use `getframe` to capture a figure and `imwrite` to save it to a file. `getframe` returns a structure containing the fields `cdata` and `colormap`. The `colormap` field is empty on true color displays. The following example captures the current figure and exports it to a PNG file.

```
I = getframe(gcf);
imwrite(I.cdata, 'myplot.png');
```

You should use the proper syntax of `imwrite` for the type of image captured. In the example above, the image is captured from a true color display. Because the `colormap` field is empty, it is not passed to `imwrite`.

Example — Exporting a Figure Using getframe and imwrite. This example offers device independence—it works for either RGB-mode or indexed-mode monitors.

```
X=getframe(gcf);
if isempty(X.colormap)
   imwrite(X.cdata, 'myplot.bmp')
else
   imwrite(X.cdata, X.colormap, 'myplot.tif')
end
```

For information about available file formats and format-specific options, see the `imwrite` reference page. For information about creating a movie from a series of frames, see the reference pages for `getframe` and `movie`, or see “Movies” on page 5-78 in Chapter 5, “Creating Specialized Plots”.

Saving Multiple Figures to an AVI File

You can also save multiple figures to an AVI file using the MATLAB `avifile` and `addframe` functions. AVI files can be used for animated sequences and do not need MATLAB to run, but do require an AVI viewer. For more information, see “Exporting Audio/Video Data” in the MATLAB Programming documentation.

Importing MATLAB Graphics into Other Applications

You can include MATLAB graphics in a wide variety of applications for word processing, slide preparation, modification by a graphics program,

presentation on the Internet, and so on. In general, the process is the same for all applications:

1. Use MATLAB to create the figure you want to import into another application.
2. Export the MATLAB figure to one of the supported graphics file formats, selecting a format that is both appropriate for the type of figure and supported by the target application. See “Choosing a Graphics Format” on page 7-73 for help.
3. Use the import features of the target application to import the graphics file.

Edit Before You Export. Vector graphics may be fully editable in a few high-end applications, but most applications do not support editing beyond simple resizing. Bitmaps cannot be edited with quality results unless you use a software package devoted to image processing. In general, you should try to make all the necessary settings while your figure is still in MATLAB.

Importing into Microsoft Applications. To import your exported figure into a Microsoft application, select **Picture** from the **Insert** menu. Then select **From File** and navigate to your exported file. If you use the clipboard to perform your export operations, you can take advantage of the recommended MATLAB settings for Word and PowerPoint.

Example — Importing an EPS Graphic into LaTeX. This example shows how to import an EPS file named `peaks.eps` into LaTeX.

```
\documentclass{article}

\usepackage{graphicx}

\begin{document}

\begin{figure}[h]
\centerline{\includegraphics[height=10cm]{peaks.eps}}
\caption{Surface Plot of Peaks}
\end{figure}

\end{document}
```

EPS graphics can be edited after being imported to LaTeX. For example, you can specify the height in any LaTeX-compatible dimension. To set the height to 3.5 inches, use the command

```
height=3.5in
```

You can use the `angle` function to rotate the graph. For example, to rotate the graph 90 degrees, add

```
angle=90
```

to the same line of code that sets the height, i.e., `[height=10cm,angle=90]`.

Exporting to the Windows or Macintosh Clipboard

You can export a figure to the Windows or Macintosh clipboard. The formats used are discussed below.

Windows Clipboard Format

MATLAB uses one of two graphics formats to write clipboard data on Windows: EMF color vector or BMP 8-bit color bitmap.

By default, MATLAB chooses the graphics format for you, based on the rendering method used to display the figure. For figures rendered with OpenGL or Z-buffer, MATLAB uses the BMP format. For figures rendered with Painter's, the EMF format is used. For information about how MATLAB selects a rendering method, see "The Default Renderer for MATLAB" on page 7-55.

To override the selection by MATLAB, specify the format of your choice using either the Windows Copy Options Preferences dialog box, or the `-d` switch in the print command.

Macintosh Clipboard Format

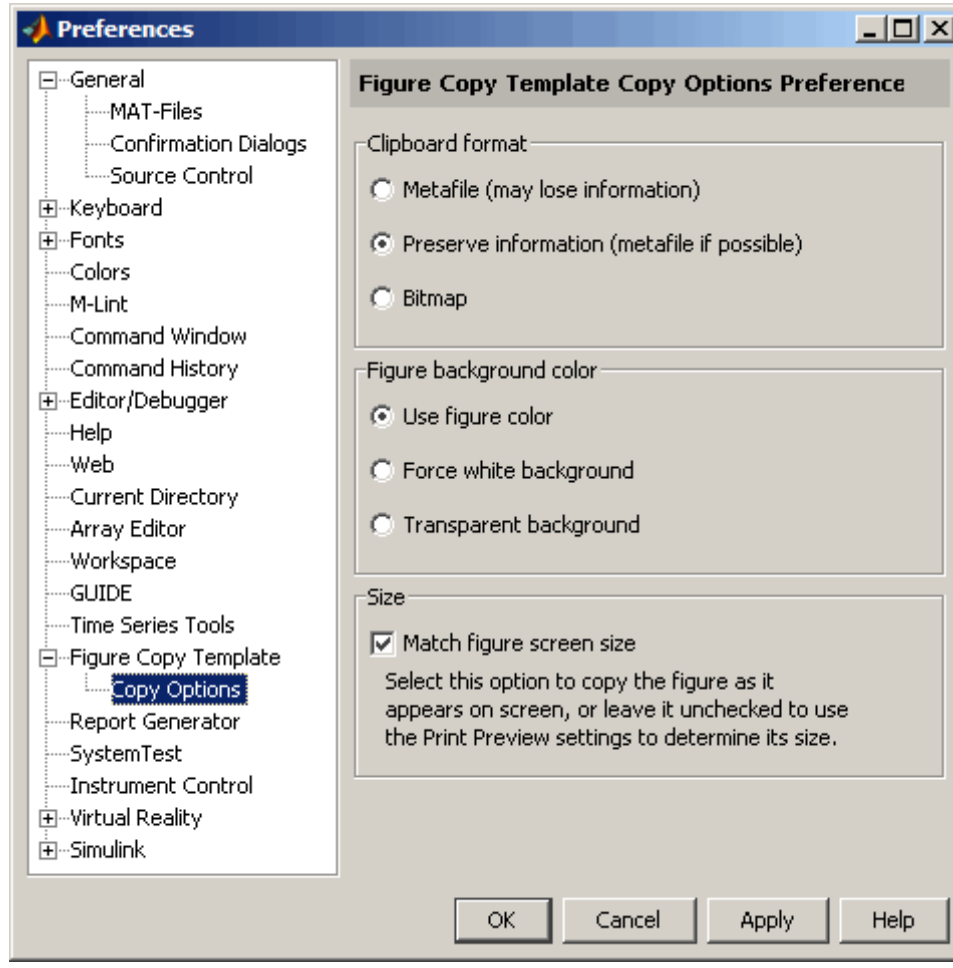
On Macintosh (using Java figures, the default), clipboard data are always written as RGB truecolor bitmaps. The entire figure window is captured.

Exporting to the Clipboard Using GUIs

Before you export the figure to the clipboard, you can use the Copy Options Preferences dialog box to select a nondefault graphics format, or to adjust certain figure settings. These settings become the new defaults for all figures exported to the clipboard.

Note When exporting to the clipboard in Windows metafile format (e.g., `print -dmeta`), the settings from the figure Copy Options Preferences template are ignored.

To open the Copy Options Preferences dialog box, select **Copy Options** from the figure window's **Edit** menu. Any changes you make with this dialog box affect only the clipboard copy of the figure; they do not affect the way the figure looks on the screen.



Settings you can change in the Copy Options Preferences dialog box are as follows:

Clipboard format. To copy the figure in EMF color vector format, select **Metafile**. To have MATLAB select the format for you, select **Preserve information**. To use BMP 8-bit color bitmap format, select **Bitmap**. MATLAB uses the metafile format whenever possible.

Note On Macintosh, the **Copy Options** dialog box does not have the **Clipboard format** options.

Figure background color. To keep the background color the same as it appears on the screen, select **Use figure color**. To make the background white, select **Force white background**. For a background that is transparent, for example, a slide background to frame the axes part of a figure, select **Transparent background**.

Size. Select **Match figure screen size** to copy the figure as it appears on the screen, or leave it unselected to use the **Width** and **height** options in the Export Setup dialog to determine its size.

1. Open the Copy Options Preferences dialog box if you need to make any changes to those preferences used in copying to the clipboard.
2. Click **OK** to see the new preferences. These will be used for all future figures exported to the clipboard.
3. Select **Copy Figure** from the figure window's **Edit** menu to copy the figure to the clipboard.

Exporting to the Windows or Macintosh Clipboard Using MATLAB Commands

Export to the clipboard using the `print` function with a graphics format, but no filename. You must use one of the following clipboard formats: `-dbitmap` or `-dmeta`. These switches create a Windows bitmap (BMP) or an enhanced metafile (EMF), respectively.

For example, to export the current figure to the clipboard in enhanced metafile format, type

```
print -dmeta
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Examples of Printing and Exporting

In this section...

Printing a Figure at Screen Size

By default, MATLAB prints your figure at 8-by-6 inches. This size includes the area delimited by the background. This example shows how to print or export your figure the same size it is displayed on your screen.

Using the Graphical User Interface

1. Resize your figure window to the size you want it to be when printed.
2. Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
3. In the **Placement** panel, select **Auto (Actual Size, Centered)**.
4. Click **Print** in the upper right corner to print the figure.
5. The Print dialog box opens for you to print the figure.

Using MATLAB Commands

Set the `PaperPositionMode` property to `auto` before printing the figure.

```
set(gcf, 'PaperPositionMode', 'auto');
print
```

If later you want to print the figure at its original size, set `PaperPositionMode` back to `'manual'`.

Printing with a Specific Paper Size

By default, MATLAB uses 8.5-by-11 inch paper. This example shows how to change the paper size to 8.5-by-14 inches by selecting a paper type (Legal).

Using the Graphical User Interface

1. Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
2. Select the `Legal` paper type from the list in the **Paper** panel. The `Width` and `Height` fields update to `8.5` and `14`, respectively.
3. Make sure that **Units** is set to `inches`.
4. Click **Print** in the upper right corner to print the figure.
5. The Print dialog box opens for you to print the figure.

Using MATLAB Commands

Set the `PaperUnits` property to `inches` and the `PaperType` property to `Legal`.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperType', 'Legal');
```

Alternatively, you can set the `PaperSize` property to the size of the paper, in the specified units.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperSize', [8.5 14]);
```

Printing a Centered Figure

This example sets the size of a figure to 5.5-by-3 inches and centers it on the paper.

Using the Graphical User Interface

1. Select **Print Preview** from the figure window's **File** menu, and select the **Layout** tab.
2. Make sure **Use manual size and position** is selected.
3. Enter `5.5` in the **Width** field and `3` in the **Height** field.
4. Make sure that **Units** field is set to `inches`.
5. Click **Center**.
6. Click **OK**.
7. Click **Print** to open the Print dialog box and print the figure.

Using MATLAB Commands

1. Start by setting `PaperUnits` to `inches`.

```
set(gcf, 'PaperUnits', 'inches')
```

2. Use `PaperSize` to return the size of the current paper.

```
papersize = get(gcf, 'PaperSize')

papersize =
              8.5000   11.0000
```

3. Initialize variables to the desired width and height of the figure.

```
width = 5.5;         % Initialize a variable for width.
height = 3;          % Initialize a variable for height.
```

4. Calculate a left margin that centers the figure horizontally on the paper. Use the first element of `papersize` (width of paper) for the calculation.

```
left = (papersize(1)- width)/2

left =
          1.5000
```

5 Calculate a bottom margin that centers the figure vertically on the paper. Use the second element of `papersize` (height of paper) for the calculation.

```
bottom = (papersize(2)- height)/2

bottom =
          4
```

6 Set the figure size and print.

```
myfiguresize = [left, bottom, width, height];
set(gcf, 'PaperPosition', myfiguresize);
print
```

Exporting in a Specific Graphics Format

Export a figure to a graphics-format file when you want to import it at a later time into another application such as a word processor.

Using the Graphical User Interface

1 Select **Save As** from the figure window's **File** menu.

2 Use the **Save in** field to navigate to the directory in which you want to save your file.

3 Select a graphics format from the **Save as type** list.

4 Enter a filename in the **File name** field. An appropriate file extension, based on the format you chose, is displayed.

5 Click **Save** to export the figure.

Using MATLAB Commands

From the command line, you must specify the graphics format as an option. See the `print` reference page for a complete list of graphics formats and their corresponding option strings.

This example exports a figure to an EPS color file, `myfigure.eps`, in your current directory.

```
print -depsc myfigure
```

This example exports Figure No. 2 at a resolution of 300 dpi to a 24-bit JPEG file, `myfigure.jpg`.

```
print -djpeg -f2 -r300 myfigure
```

This example exports a figure at screen size to a 24-bit TIFF file, `myfigure.tif`.

```
set(gcf, 'PaperPositionMode', 'auto')   % Use screen size
print -dtiff myfigure
```

Exporting in EPS Format with a TIFF Preview

Use the `print` function to export a figure in EPS format with a TIFF preview. When you import the figure, the application can display the TIFF preview in the source document. The preview is color if the exported figure is color, and black and white if the exported figure is black and white.

This example exports a figure to an EPS color format file, `myfigure.eps`, and includes a color TIFF preview.

```
print -depsc -tiff myfigure
```

This example exports a figure to an EPS black-and-white format file, `myfigure.eps`, and includes a black-and-white TIFF preview.

```
print -deps -tiff myfigure
```

Exporting a Figure to the Clipboard

Export a figure to the clipboard in graphics format when you want to paste it into another Windows or Macintosh application such as a word processor.

Using the Graphical User Interface

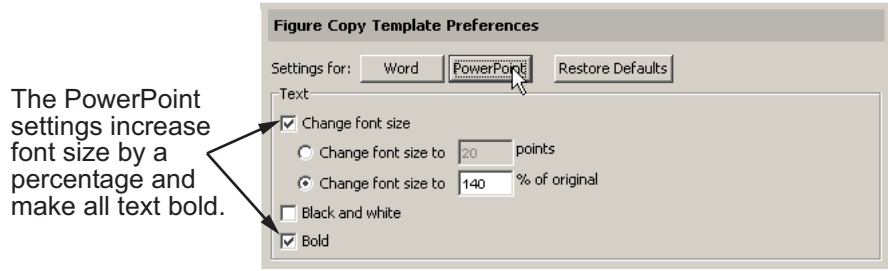
This example exports a figure to the clipboard in enhanced metafile (EMF) format. Figure settings are chosen that would make the exported figure suitable for use in a Microsoft Word™or PowerPoint™ slide. Note that changing the settings modifies the figure displayed on the screen.

1 Create a figure containing text. You can use the following code.

```
x = -pi:0.01:pi;
h = plot(x, sin(x));
title('Sine Plot');
```

2 Select **Preferences** from the **File** menu of either the figure or main desktop window. Then select **Figure Copy Template** from the Preferences dialog box.

3 In the **Figure Copy Template Preferences** panel, click the **PowerPoint** button. The MATLAB suggested settings for PowerPoint are added to the template.



The Text Panel after pushing the **PowerPoint** button

4 In the **Lines** panel, change the **Custom width** to 4 points.

5 In the **Uicontrols and axes** panel, select **Keep axes limits and tick spacing** to prevent MATLAB from possibly rescaling tick marks and limits when you export.

6 Click **Apply to Figure**. The changes appear in the figure window.

If you don't like the way your figure looks with the new settings, restore it to its original settings by clicking the **Restore Figure** button.

7 In the left pane of the Preferences dialog box, expand the **Figure Copy Template** topic. Select **Copy Options**.

8 In the **Copy Options** panel, select **Metafile** to tell MATLAB to export the figure in EMF format.

9 Check that **Transparent background** is selected. This choice makes the figure background transparent and allows the slide background to frame the axes part of the figure.

10 Clear the **Match figure screen size** check box so that you can use your own figure size settings.

11 Click **OK**.

12 Select **Export Setup** from the figure window's **File** menu.

13 Select the **Size** properties, and set **Width** to `6` and **Height** to `4.5`. Make sure that **Units** are set to `inches`.

14 Click **Close**.

15 Select **Copy Figure** from the **Edit** menu. Your figure is now exported to the clipboard and can be pasted into another Windows application, such as PowerPoint.

Using MATLAB Commands

Use the `print` function and one of two clipboard formats (`-dmeta`, `-dbitmap`) to export a figure to the clipboard. Do *not* specify a filename.

This example exports a figure to the clipboard in enhanced metafile (EMF) format.

```
print -dmeta
```

This example exports a figure to the clipboard in bitmap (BMP) 8-bit color format.

```
print -dbitmap
```

Changing a Figure's Settings

In this section...

Parameters that Affect Printing

The table below shows parameters that you can set before submitting your figure to the printer.

The Parameter column lists all parameters that you can change.

The Default column shows the default setting that MATLAB uses.

The Dialog Box column shows which dialog box to use to set that parameter. If you can make this setting on only one platform, this is noted in parentheses: (W) for Windows, and (U) for UNIX.

Some dialog boxes have tabs at the top to enable you to select a certain category. These categories are denoted in the table below using the format `<dialogbox>/<tabname>`. For example, **Print Preview/Layout...** in this column means to use the Print Preview dialog box, selecting the **Layout** tab.

The `print` Command or `set` Property column shows how to set the parameter using the MATLAB `print` or `set` function. When using `print`, the table shows the appropriate command option (for example, `print -loose`). When using `set`, it shows the property name to set along with the type of object (for example, (Line) for line objects).

Parameter	Default	Dialog Box	`print` Command or `set` Property
Select figure	Last active window	None	`print -f`*`handle`*
Select printer	System default	**Print**	`print -p`*`printer`*
Figure size	8-by-6 inches	**Print Preview/Layout**	`PaperSize` (Figure) `PaperUnits` (Figure)
Position on page	0.25 in. from left, 2.5 in. from bottom	**Print Preview/Layout**	`PaperPosition` (Figure) `PaperUnits` (Figure)
Position mode	Manual	**Print Preview/Layout**	`PaperPositionMode` (Figure)
Paper type	Letter	**Print Preview/Layout**	`PaperType` (Figure)
Paper orientation	Portrait	**Print Preview/Layout**	`PaperOrientation` (Figure)
Renderer	Selected by MATLAB	**Print Preview/Advanced**	`print -zbuffer | -painters | -opengl`
Renderer mode	Auto	**Print Preview/Advanced**	`RendererMode` (Figure)
Resolution	Depends on driver or graphics format	**Print Preview/Advanced**	`print -r`*`resolution`*

Parameter	Default	Dialog Box	`print` Command or `set` Property
Axes tick marks	Recompute	**Print Preview/Advanced**	`XTickMode, etc.` (Axes)
Background color	Force to white	**Print Preview/Color**	`Color` (Figure) `InvertHardCopy` (Figure)
Font size	As in the figure	**Print Preview/Lines/Text**	`FontSize` (Text)
Bold font	Regular font	**Print Preview/Lines/Text**	`FontWeight` (Text)
Line width	As in the figure	**Print Preview/Lines/Text**	`LineWidth` (Line)
Line style	Black or white	**Figure Copy Template**	`LineStyle` (Line)
Line and text color	Black and white	**Print Preview/Lines/Text**	`Color` (Line, Text)
CMYK color	RGB color	**Print Preview/Color** (U)	`print -cmyk`
UI controls	Printed	**Print Preview/Advanced**	`print -noui`
Bounding box	Tight	N/A	`print -loose`
Copy background	Transparent	**Copy Options** (W)	See "Background color"
Copy size	Same as screen size	**Copy Options** (W)	See "Figure Size"

Selecting the Figure

By default, MATLAB prints the current figure. If you have more than one figure open, the current figure is the last one that was active. To make a different figure active, click it to bring it to the foreground.

Using MATLAB Commands

Specify a figure handle using the command

```
print -fhandle
```

This example sends Figure No. 2 to the printer. A figure's number is usually its handle.

```
print -f2
```

Selecting the Printer

You can select the printer you want to use with the Print dialog box or with the `print` function.

Using the Graphical User Interface

1. Select **Print** from the figure window's **File** menu.
2. Select the printer from the list box near the top of the Print dialog box.
3. Click **OK**.

Using MATLAB Commands

You can select the printer using the `-P` switch of the `print` function.

This example prints Figure No. 3 to a printer called `Calliope`.

```
print -f3 -PCalliope
```

If the printer name has spaces in it, put single quotation marks around the `-P` option, as shown here.

```
print '-Pmy local printer'
```

Using a Network Print Server. On Windows systems, you can print to a network print server using the form shown here for a printer named `trinity` located on a computer named `PRINTERS`.

```
print -P\\PRINTERS\trinity
```

Note On Windows, when you use the `-P` option to identify a printer to use, if you specify any driver other than `-dwin` or `-dwinc`, MATLAB writes the output to a file with an appropriate extension but does not send it to the printer; you can then copy that file to a printer.

Setting the Figure Size and Position

The default output figure size is 8 inches wide by 6 inches high, which maintains the aspect ratio (width to height) of the MATLAB figure window. The figure's default position is centered both horizontally and vertically when printed to a paper size of 8.5-by-11 inches.

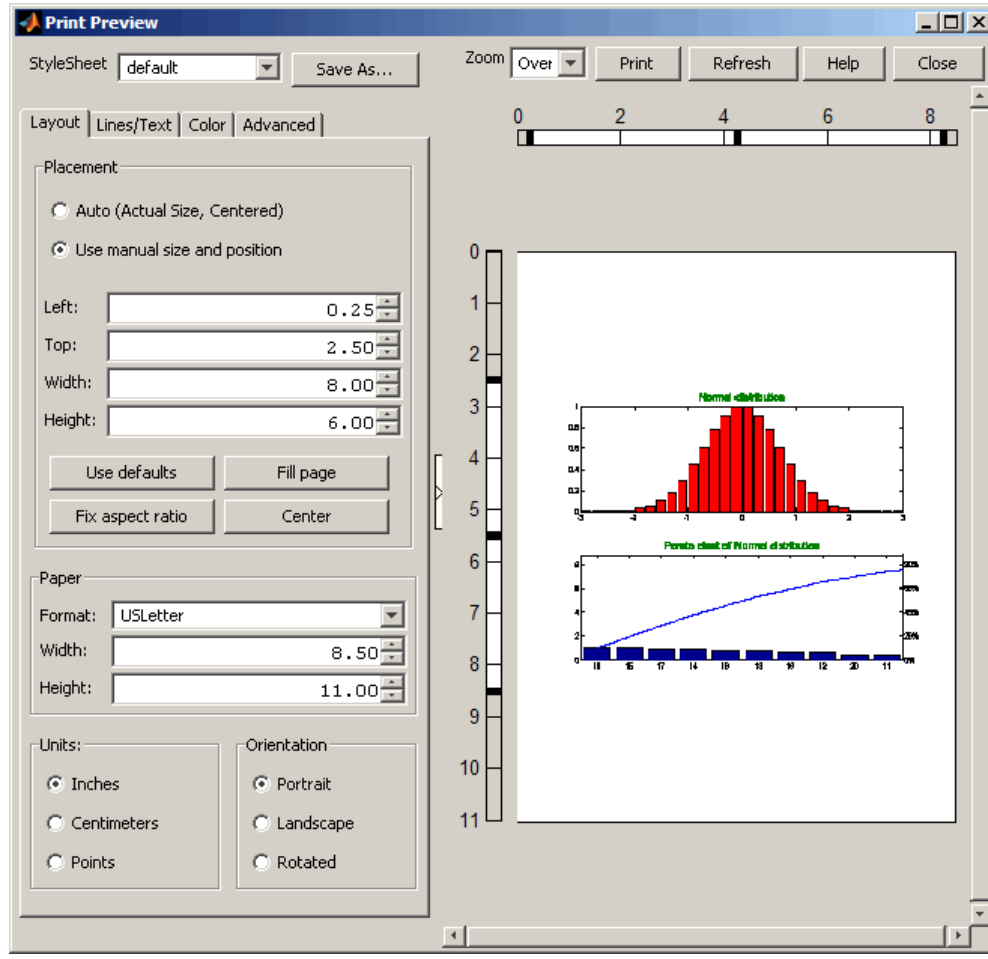
You can change the size and position of the figure:

- "Using the Graphical User Interface" on page 7-47
- "Using MATLAB Commands" on page 7-49

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Click the **Layout** tab to make changes to the size and position of your figure on the printed page.

Use the text edit boxes on the left to enter new dimensions for your figure. Or, use the handlebars on the rulers in the right-hand pane to drag the margins and location of your figure with the mouse. The outer handlebars move the figure toward or away the nearest margin, while the central handlebar repositions the figure on the page without changing its proportions. Guidelines appear while you are using the handlebars.



Settings you can change in the **Layout** tab are as follows:

Placement. Choose whether you want the figure to be the same size as it is displayed on your screen, or you want to manually change its size using the options in the **Layout** pane.

When you select the **Use manual size and position** mode, type the widths of any of the four margins and the preview image responds after each entry you

make. Select units of measure (inches/centimeters/points) with pushbuttons on the **Units** section on the bottom of the pane.

You can use the four buttons at the bottom of the Placement section to expand the figure to fill the page, make its aspect ratio (ratio of y-extent to x-extent) as printed match that of the figure, center the figure on the page, or restore the setup to what it was when you opened the Print Preview dialog. Selecting **Fill page** can alter the aspect ratio of your image. To get the maximum figure size without altering the aspect ratio, select **Fix aspect ratio**.

Auto (actual size, centered). Select this option to center the figure on the page; it will be the same size as it is in the figure window. The four buttons below the control are dimmed when you select this option.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

To print your figure with a specific size or position, make sure the `PaperPositionMode` property is set to `manual` (the default). Then set the `PaperPosition` property to the desired size and position.

The `PaperPosition` property references a four-element row vector that specifies the position and dimensions of the printed output. The form of the vector is

```
[left bottom width height]
```

where

- `left` specifies the distance from the left edge of the paper to the left edge of the figure.
- `bottom` specifies the distance from the bottom of the paper to the bottom of the figure.
- `width` and `height` specify the figure's width and height.

The MATLAB default values for `PaperPosition` are

```
[0.25 2.5 8.0 6.0]
```

This example sets the figure size to a width of 4 inches and height of 2 inches, with the origin of the figure positioned 2 inches from the left edge of the paper and 1 inch from the bottom edge.

```
set(gcf, 'PaperPositionMode', 'manual');
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperPosition', [2 1 4 2]);
```

Note `PaperPosition` specifies a bottom margin, rather than a top margin as Print Preview does. When you set the top margin using Print Preview, MATLAB uses this setting to calculate the bottom margin, and updates the `PaperPosition` property appropriately.

Setting the Paper Size or Type

Set the paper size by specifying the dimensions or by choosing from a list of predefined paper types. If you do not set a paper size or type, MATLAB uses the default paper size of 8.5-by-11 inches.

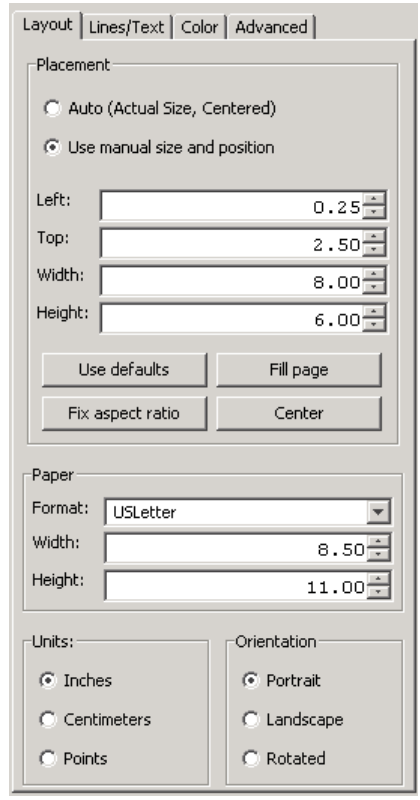
Paper-size and paper-type settings are interrelated—if you set a paper type, MATLAB updates the paper size. For example, if you set the paper type to `US Legal`, MATLAB updates the width of the paper to 8.5 inches and the height to 14 inches.

You can change the paper size and orientation:

- "Using the Graphical User Interface" on page 7-50
- "Using MATLAB Commands" on page 7-52

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Click the **Layout** tab to make changes to the paper type and orientation of the figure on the printed page.



Settings you can change in the **Layout** tab are as follows:

Paper Format, Units and Orientation. Select a paper type from the list under **Format**. If there is no paper type with suitable dimensions, enter your own dimensions in the **Width** and **Height** fields. Make sure **Units** is set appropriately to `inches`, `centimeters`, or `points`. If you change units after setting a paper width and height, the **Width** and **Height** fields update to use the units you just selected. The page region in the preview pane updates to show the new paper format or size when you change them.

Use the **Orientation** buttons to select how you want the figure to be oriented on the printed page. The illustration under "Setting the Paper Orientation" on page 7-52 shows the three types of orientation you can choose from.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

Set the `PaperType` property to one of the built-in MATLAB paper types, or set the `PaperSize` property to the dimensions of the paper.

When you select a paper type, the unit of measure is not automatically updated. We recommend that you set the `PaperUnits` property first.

For example, these commands set the units to `centimeters` and the paper type to `A4`.

```
set(gcf, 'PaperUnits', 'centimeters');
set(gcf, 'PaperType', 'A4');
```

This example sets the units to `inches` and sets the paper size of 5-by-7 inches.

```
set(gcf, 'PaperUnits', 'inches');
set(gcf, 'PaperSize', [5 7]);
```

If you set a paper size for which there is no matching paper type, the `PaperType` property is automatically set to `'custom'`.

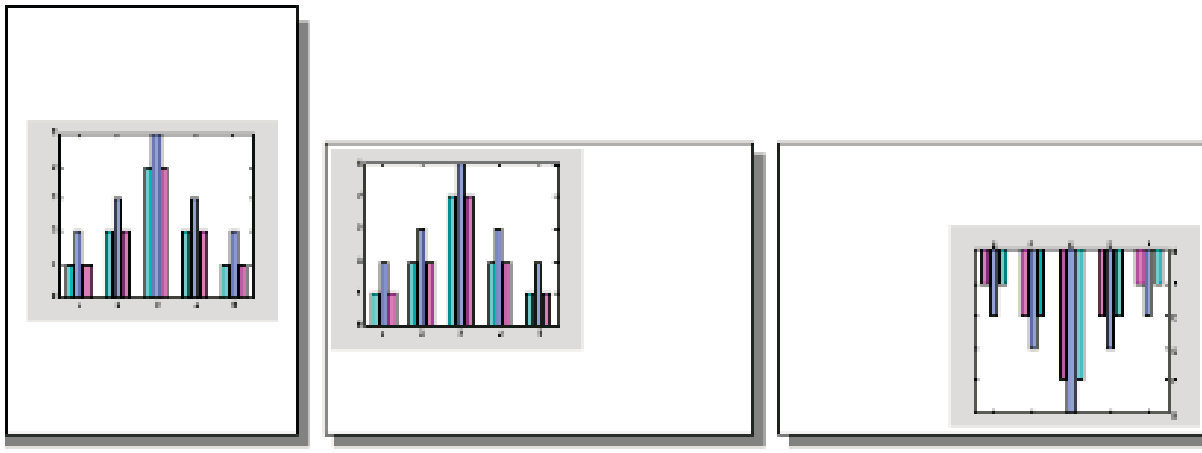
Setting the Paper Orientation

Paper orientation refers to how the paper is oriented with respect to the figure. The choices are **Portrait** (the default), **Landscape**, and **Rotated**.

You can change the orientation of the figure:

- "Using the Graphical User Interface" on page 7-53
- "Using MATLAB Commands" on page 7-53

The figure below shows the same figure printed using the three different orientations.



Note The **Rotated** orientation is not supported by all printers. When the printer does not support it, landscape is used.

Using the Graphical User Interface

1. Select **Print Preview** from the figure window's **File** menu and select the **Layout** tab. (See "Using the Graphical User Interface" on page 7-50).
2. Select the appropriate option button under **Orientation**.
3. Click **Close**.

Using MATLAB Commands

Use the `PaperOrientation` figure property or the `orient` function. Use the `orient` function if you always want your figure centered on the paper.

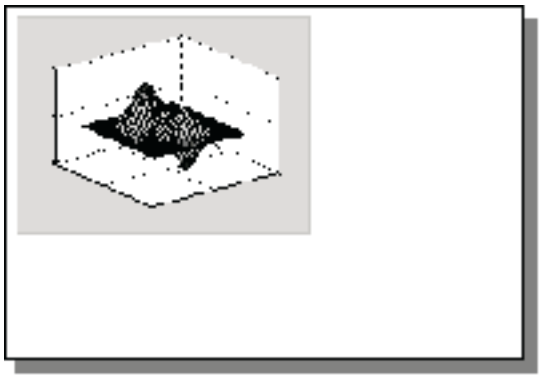
The following example sets the orientation to `landscape`:

```
set(gcf, 'PaperOrientation', 'landscape');
```

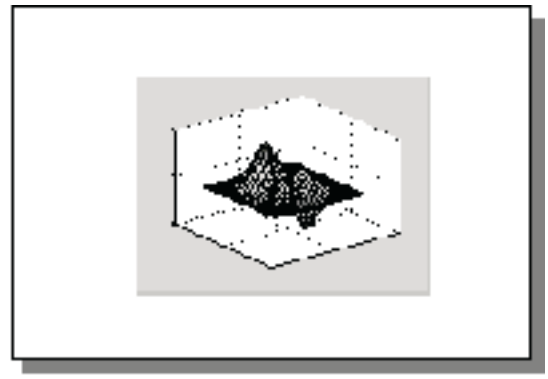
Centering the Figure. If you set the `PaperOrientation` property from `portrait` to either of the other two orientation schemes, you might find that what was previously a centered image is now positioned near the paper's edge. You can either adjust the position (use the `PaperPosition` property), or you can use the `orient` function, which always centers the figure on the paper.

The `orient` function takes the same argument names as `PaperOrientation`. For example,

```
orient rotated;
```



Orientation set to '`landscape`' using '`PaperOrientation`' property.



Orientation set to '`landscape`' using `orient` function.

Selecting a Renderer

A renderer is software and/or hardware that processes graphics data (such as vertex coordinates) to display, print, or export a figure. You can change the renderer that MATLAB uses when printing a figure:

- "Using the Graphical User Interface" on page 7-56
- "Using MATLAB Commands" on page 7-57

Renderers Supported by MATLAB

MATLAB supports three rendering methods with the following characteristics:

Painter's

- Draws figures using vector graphics
- Generally produces higher resolution results
- The fastest renderer when the figure contains only simple or small graphics objects
- The only renderer possible when printing with the HPGL print driver or exporting to an Adobe Illustrator file
- The best renderer for creating PostScript or EPS files
- Cannot render figures that use RGB color for patch or surface objects
- Does not show lighting or transparency

Z-buffer

- Draws figures using bitmap (raster) graphics
- Faster and more accurate than Painter's
- Can consume a lot of system memory if MATLAB is displaying a complex scene
- Shows lighting, but not transparency

OpenGL

- Draws figures using bitmap (raster) graphics
- Generally faster than Painter's or Z-buffer
- In some cases, enables MATLAB to access graphics hardware that is available on some systems
- Shows both lighting and transparency

For more detailed information about the rendering methods, see `Renderer` on the Figure Properties reference page.

The Default Renderer for MATLAB

By default, MATLAB tries to optimize the rendering method based on the attributes of the figure (its complexity and the settings of various Handle Graphics properties) and in some cases, the printer driver or file format used.

In general, MATLAB uses

- Painter's for line plots, area plots (bar graphs, histograms, etc.), and simple surface plots
- Z-buffer when the computer screen is not true color or when the `opengl` function was called with `selection_mode` set to `neverselect`
- OpenGL for complex surface plots using interpolated shading and any figure using lighting

The `RendererMode` property tells MATLAB whether to automatically select the renderer based on the contents of the figure (when set to `auto`), or to use the `Renderer` property that you have indicated (when set to `manual`).

Reasons for Manually Setting the Renderer

Two reasons to set the renderer yourself are

- To make your printed or exported figure look the same as it did on the screen. The rendering method used for printing and exporting the figure is not always the same method used to display the figure.
- To avoid unintentionally exporting your figure as a bitmap within a vector format. For example, MATLAB typically renders high-complexity plots using OpenGL or Z-buffer. If you export a high-complexity figure to the EPS or EMF vector formats without specifying a rendering method, MATLAB might use OpenGL or Z-buffer, each of which creates bitmap graphics.

  Storing a bitmap in a vector file can generate a very large file that takes a long time to print. If you use one of these formats and want to make sure that your figure is saved as a vector file, be sure to set the rendering method to Painter's.

Using the Graphical User Interface

1. Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu. Select the **Advanced** tab.
2. In the `Renderer` drop-down menu, select the desired rendering method from the list box.

3 Click **Close**.

Using MATLAB Commands

You can use the `Renderer` property or a switch with the `print` function to set the renderer for printing or exporting. These two lines each set the renderer for the current figure to Z-buffer.

```
set(gcf, 'Renderer', 'zbuffer');
```

or

```
print -zbuffer
```

The first example saves the new value of `Renderer` with the figure; the second example only affects the current print or export operation.

Note that when you set the `Renderer` property, the `RendererMode` property is automatically reset from `auto` (the factory default) to `manual`.

Setting the Resolution

Resolution refers to how accurately your figure is rendered when printed or exported. Higher resolutions produce higher quality output. The specific definition of resolution depends on whether your figure is output as a bitmap or as a vector graphic.

You can change the resolution that MATLAB uses to print a figure:

- "Using the Graphical User Interface" on page 7-59
- "Using the Graphical User Interface on UNIX" on page 7-71
- "Using MATLAB Commands" on page 7-59

Default Resolution and When You Can Change It

The default resolution depends on the renderer used and the graphics format or printer driver specified. The following two tables summarize the default resolutions and whether you can change them.

Resolutions Used with Graphics Formats

Graphics Format	Default Resolution	Can Be Changed?
Built-in MATLAB export formats, (except for EMF, EPS, and ILL)	150 dpi (always use OpenGL or Z-buffer)	Yes
EMF export format (Enhanced Metafile)	150 dpi	Yes
EPS (Encapsulated PostScript)	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
ILL export format (Adobe Illustrator)	72 dpi (always uses Painter's)	No
Ghostscript export formats	72 dpi (always uses OpenGL or Z-buffer)	No

Resolutions Used with Printer Drivers

Printer Driver	Default Resolution	Can Be Changed?
Windows and PostScript drivers	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
Ghostscript driver	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
HPGL driver	1116 dpi (always uses Painter's)	Yes

Choosing a Setting

You might need to determine your resolution requirements through experimentation, but you can also use the following guidelines.

For Printing. The default resolution of 150 dpi is normally adequate for typical laser-printer output. However, if you are preparing figures for high-quality printing, such as a textbook or color brochures, you might want to use 200 or 300 dpi. The resolution you can use can be limited by the printer's capabilities.

For Exporting. If you are exporting your figure, base your decision on the resolution supported by the final output device. For example, if you will import your figure into a word processing document and print it on a printer that supports a maximum resolution setting of 300 dpi, you could export your figure using 300 dpi to get a precise one-to-one correspondence between pixels in the file and dots on the paper.

Note The only way to set resolution when exporting is with the `print` function.

Impact of Resolution on Size and Memory Needed

Resolution affects file size and memory requirements. For both printing and exporting, the higher the resolution setting, the longer it takes for MATLAB or your printer to render your figure.

Using the Graphical User Interface

To set the resolution for built-in MATLAB printer drivers:

1. From the Print dialog box, click **Properties**. This opens a new dialog box. (This box can differ from one printer to another.)
2. You may be able to set the resolution from this dialog. If not, then click **Advanced** to get to a dialog box that enables you to do this.
3. Set the resolution, and then click **OK**. (The resolution setting might be labeled by another name, such as "Print Quality.")

Using MATLAB Commands

If you use a Windows printer driver, you can only set the resolution using the Windows Document Properties dialog box.

Otherwise, to set the resolution for printing or exporting, the syntax is

```
print -rnumber
```

where `number` is the number of dots per inch. To print or export a figure using screen resolution, set `number` to `0` (zero).

This example prints the current figure with a resolution of 100 dpi:

```
print -r100
```

This example exports the current figure to a TIFF file using screen resolution:

```
print -r0 -dtiff myfile.tif
```

Setting the Axes Ticks and Limits

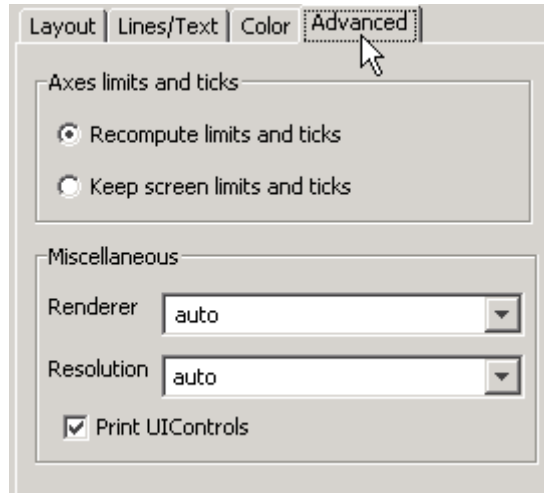
The MATLAB default output size, 8-by-6 inches, is normally larger than the screen size. If the size of your printed or exported figure is different from its size on the screen, MATLAB scales the number and placement of axes tick marks to suit the output size. This section shows you how to lock them so that they are the same as they were when displayed.

You can change the resolution that MATLAB uses to print a figure:

- "Using the Graphical User Interface" on page 7-61
- "Using MATLAB Commands" on page 7-62

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Advanced** tab to make changes to the axes, UI controls, or renderer settings.



Settings you can change in the **Advanced** tab are as follows, by panel:

Axes limits and ticks. If the size of your printed or exported figure is different from its size on the screen, MATLAB scales the number and placement of axes tick marks to suit the output size. Select **Keep screen limits and ticks** to lock them so that they are the same as they were when displayed. If you want MATLAB to adjust the ticks and limits when scaling for printing, select **Recompute limits and ticks**.

Miscellaneous. Use the **Renderer** drop-down menu to tell MATLAB which renderer to use in printing the figure. Set the renderer to `Painters`, `Z-buffer`, or `OpenGL`, or select `auto` to let MATLAB decide which one to use, depending on the characteristics of the figure. (See "Selecting a Renderer" on page 7-54).

Use the **Resolution** drop-down menu to specify the resolution, in dots per inch (DPI), at which to render and print the figure. You can select `150`, `300`, or `600` DPI, or type in a different value (positive integer).

Figure UI Controls. By default, user interface controls are included in your printed or exported figure. Clear the **Print UIControls** check box to exclude them. (See "Excluding User Interface Controls form Printed Output" on page 7-71).

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

To set the `XTickMode`, `YTickMode`, and `ZTickMode` properties to `manual`, type

```
set(gca, 'XTickMode', 'manual');
set(gca, 'YTickMode', 'manual');
set(gca, 'ZTickMode', 'manual');
```

Setting the Background Color

You can keep the background the same as is shown on the screen when printed, or change the background to white. There are two types of background color settings in a figure: the axes background and the figure background. The default displayed color of both backgrounds is gray, but you can set them to any of several colors.

Regardless of the background colors in your displayed figure, by default, MATLAB always changes them to white when you print or export. This section shows you how to retain the displayed background colors in your output.

Using the Graphical User Interface

To retain the background color on a per figure basis:

1. Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu. Select the **Color** tab.
2. Select **Same as figure**.
3. Click **Close**.

If you are exporting your figure using the clipboard, use the **Copy Options** panel of the Preferences dialog box.

Using MATLAB Commands

To retain your background colors, use

```
set(gcf, 'InvertHardCopy', 'off');
```

The following example sets the figure background color to `blue`, the axes background color to `yellow`, and then sets `InvertHardCopy` to `off` so that these colors appear in your printed or exported figure.

```
set(gcf, 'color', 'blue');
set(gca, 'color', 'yellow');
set(gcf, 'InvertHardCopy', 'off');
```

Setting Line and Text Characteristics

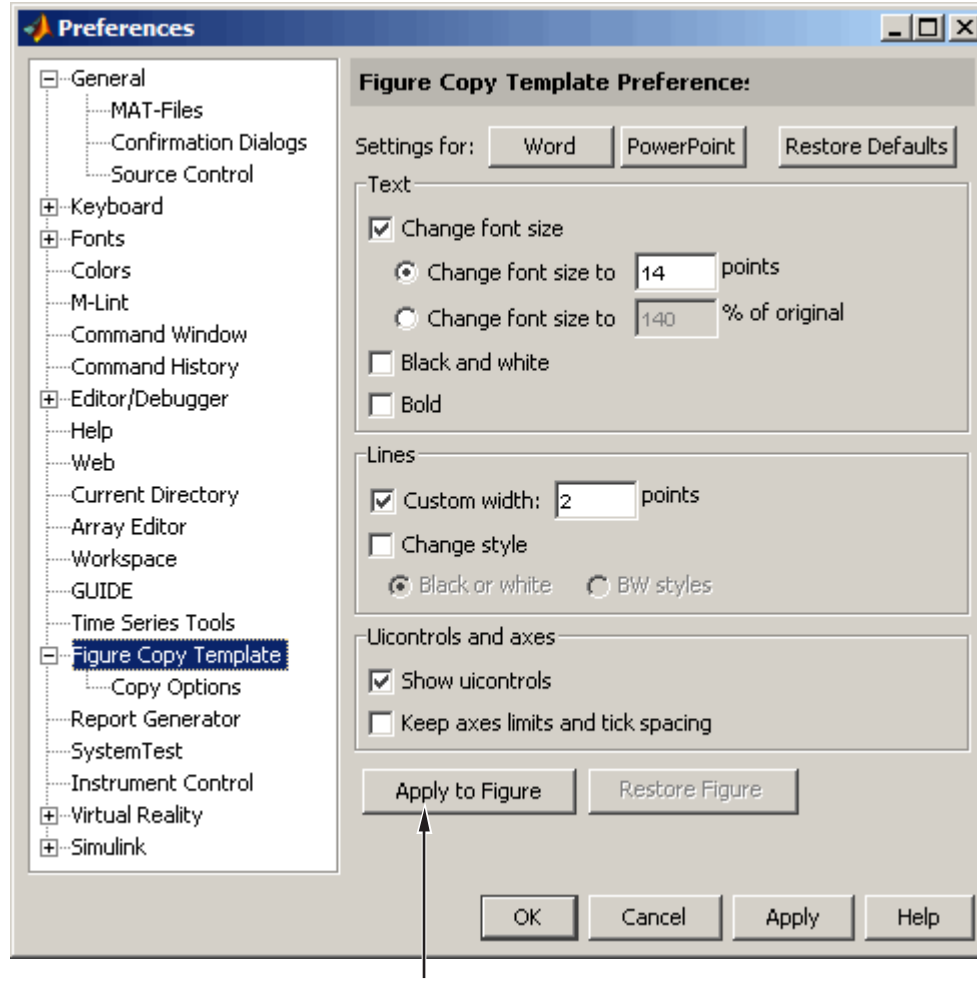
If you transfer your figures to Word or PowerPoint applications, you can set line and text characteristics to values recommended for those applications. The Figure Copy Template Preferences dialog box provides Word and PowerPoint options to make these settings, or you can set certain line and text characteristics individually.

You can change line and text characteristics:

- "Using the Graphical User Interface" on page 7-64
- "Using MATLAB Commands" on page 7-65

Using the Graphical User Interface

To open Figure Copy Template Preferences, select **Preferences** from the **File** menu, and then click **Figure Copy Template** in the left pane.



Note the difference between **Apply to Figure** and **Apply**.
Use **Apply to Figure** to modify the figure in the figure window.
Use **Apply** or **OK** to save your preferences.

Settings you can change in the Figure Copy Template Preferences dialog box are as follows:

Word or PowerPoint. Click **Word** or **PowerPoint** to apply settings recommended for MATLAB.

Text. Use options under **Text** to modify the appearance of all text in the figure. You can change the font size, change the text color to black and white, and change the font style to bold.

Lines. Use the **Lines** options to modify the appearance of all lines in the figure:

- **Custom width** — Change the line width.
- **Change style (Black or white)** — Change colored lines to black or white.
- **Change style (B&W styles)** — Change solid lines to different line styles (e.g., solid, dashed, etc.), and black or white color.

UIControls and axes. If your figure includes user interface controls, you can choose to show or hide them by clicking **Show uicontrols**. Also, to keep axes limits and tick marks as they appear on the screen, click **Keep axes limits and tick spacing**. To allow MATLAB to scale axes limits and tick marks based on the size of the printed figure, clear this box.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

You can use the `set` function on selected graphics objects in your figure to change individual line and text characteristics.

For example, to change line width to `1.8` and line style to a dashed line, use

```
lineobj = findobj('type', 'line');
set(lineobj, 'linewidth', 1.8);
set(lineobj, 'linestyle', '--');
```

To change the font size to 15 points and font weight to bold, use

```
textobj = findobj('type', 'text');
set(textobj, 'fontunits', 'points');
set(textobj, 'fontsize', 15);
set(textobj, 'fontweight', 'bold');
```

Setting the Line and Text Color

When colored lines and text are dithered to gray by a black-and-white printer, it does not produce good results for thin lines and the thin lines that make up text characters. You can, however, force all line and text objects in the figure to print in black and white, thus improving their appearance in the printed copy. When you select this setting, the lines and text are printed all black or all white, depending on the background color.

The default is to leave lines and text in the color that appears on the screen.

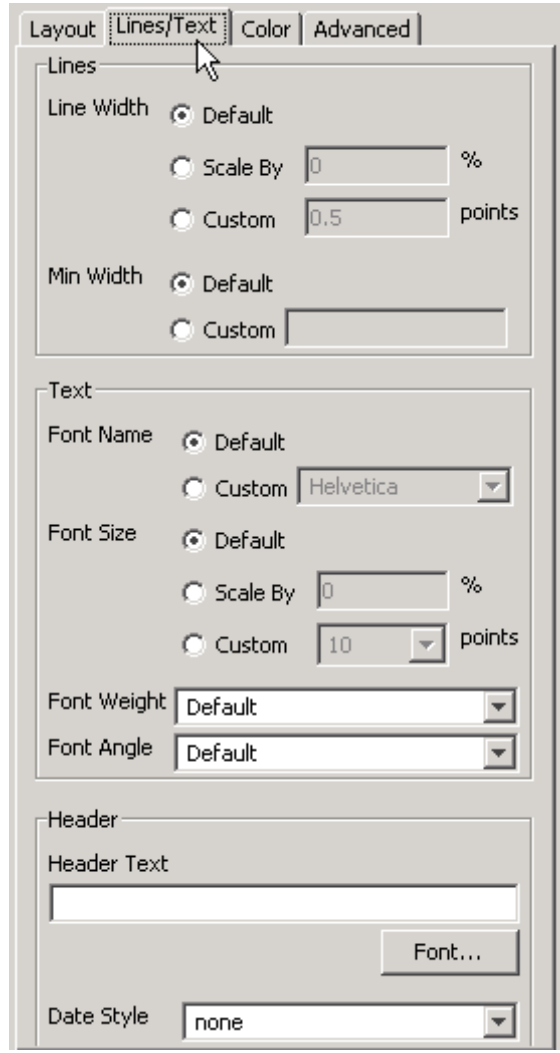
Note Your background color might not be the same as what you see on the screen. See the **Color** tab for an option that preserves the background color when printing.

You can change the resolution that MATLAB uses to print a figure:

- "Using the Graphical User Interface" on page 7-66
- "Using MATLAB Commands" on page 7-68

Using the Graphical User Interface

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Lines and Text** tab to make changes to the color of all lines and text on the printed page. The controls for the **Lines and Text** tab are shown below:



Settings you can change in **Lines and Text** are as follows:

Lines. The default option in this panel causes lines to print at the same width they are portrayed in the figure window. You can scale line width from 0 percent upwards for printing using the **Scale By** field. To print lines at a particular point size, select **Custom**. All lines on the plot will be the same weight when you use the **Custom** option; the **Scale By** option respects relative line weight.

When you scale lines downward, you can prevent them from becoming too faint by setting the **Min Width** option to **Custom** and specifying a minimum line width in points in that field.

Text. The default is to print text in the same font and at the same size as it is in the figure. To change the font (for all text) select **Custom** and choose a new font from the drop-down list that is then enabled. Scale the font size using the **Scale By** option. To print text at a particular point size, select **Custom**. All text on the plot will be printed at the point size you specify when you use the **Custom** option; the **Scale By** option respects relative font size. You can specify the **Font Weight** (normal, light, demi, or bold) and **Font Angle** (`normal`, `italic`, or `oblique`) for all text as well, using the drop-down menus at the bottom of the **Text** panel.

Header. Type any text that you want to appear at the top of the printed figure in the **Header Text** edit field. If you want today's date and/or time appended to the header text, select the appropriate format from the **Date Style** popup menu. To specify and style the header font (which is independent of the font used in the figure), click the **Font** button and choose a font name, size, and style from the **Font** selection window that appears.

Note Changes you make using Print Preview affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

There is no equivalent MATLAB command that sets line and text color depending on background color. Set the color of lines and text using the `set` function on either line or text objects in your figure.

This example sets all lines and text to black:

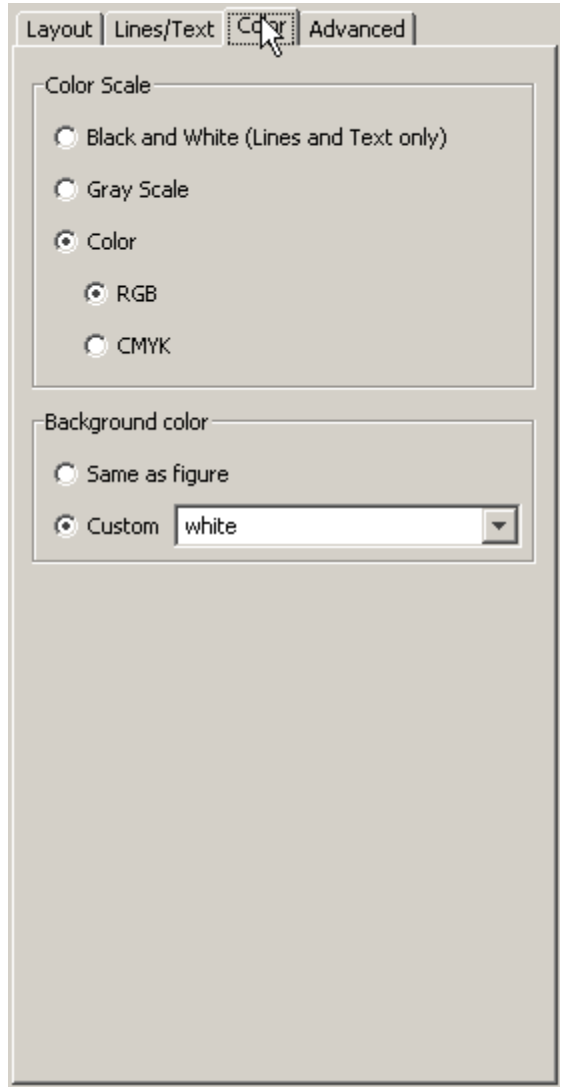
```
set(findobj('type', 'line'), 'color', 'black');
set(findobj('type', 'text'), 'color', 'black');
```

Specifying a Colorspace for Printing and Exporting

By default, MATLAB produces color output in the RGB color space (red, green, blue). If you plan to publish and print MATLAB figures using printing industry standard four-color separation, you might want to use the CMYK color space (cyan, magenta, yellow, black).

Using the Graphical User Interface on Windows

Select **Print Preview** from the figure window's **File** menu to open the Print Preview dialog box. Select the **Color** tab to make changes to the color of all lines and text on the printed page. The controls for the **Color** tab are shown below:



You can print the contents of your figure in color, grayscale, or black-and-white by selecting the appropriate button in the panel. When you select **Color**, you can choose between an **RGB** (red/green/blue) or a **CMYK** (cyan/magenta/yellow/black) color specification, if your printer is capable of it.

Independently of the **Color Scale** controls, you can specify a **Background color** for printing. Select **Same as figure** to use the color used in the figure itself (default is gray), or specify a **Custom** color from the combo box popup menu. The choices are `black`, `white`, and several RGB color triplet values; you type any valid MATLAB colorspec in this field as well, such as `g`, `magenta`, or `.3 .4 .5`.

The background color you specify is respected even if you choose **Black and White** or **Gray Scale** in the **Color Scale** panel.

Using the Graphical User Interface on UNIX

1. Select **Print** from the figure window's **File** menu.
2. Click the **Appearance** tab.
3. In the **Color Appearance** panel, select **Color**.
4. Click **Print**.

On any platform, you can also indicate whether to print in color, grayscale or black-and-white with the Print Preview dialog box.

Using MATLAB Commands

Use the `-cmyk` option with the `print` function. This example prints the current figure in CMYK using a PostScript Level II color printer driver.

```
print -dpsc2 -cmyk
```

Excluding User Interface Controls form Printed Output

User interface controls are objects that you create and add to a figure. For example, you can add a button to a figure that, when clicked, conveniently runs another M-file. By default, user interface controls are included in your printed or exported figure. This section shows how to exclude them.

Using the Graphical User Interface

1. Open the Print Preview dialog box by selecting **Print Preview** from the figure window's **File** menu, and then select the **Advanced** tab.
2. Under **Miscellaneous**, clear the **Print UIControls** check box.
3. Click **Close**.

Using MATLAB Commands

Use the `-noui` switch. This example specifies a color PostScript driver and excludes UI controls.

```
print -dpsc -noui
```

This example exports the current figure to a color EPS file and excludes UI controls.

```
print -depsc -noui myfile.eps
```

Producing Uncropped Figures

In most cases, MATLAB crops the background tightly around the objects in the figure. Depending on the printer driver or file format you use, you might be able to produce uncropped output. An uncropped figure has increased background area and is often desirable for figures that contain UI controls.

The setting you make in MATLAB changes the PostScript `BoundingBox` property saved with the figure.

Using MATLAB Commands

Use the `-loose` option with the `print` function. For Windows, the uncropped option is only available if you print to a file.

This example exports the current figure, uncropped, to an EPS file.

```
print -deps -loose myfile.eps
```

Choosing a Graphics Format

In this section...

What Are Graphic Formats?

A graphics file format is a specification for storing and organizing data in a file. MATLAB supports many different graphics file formats. Some are built into MATLAB and others are Ghostscript formats. File formats also differ in color support, graphics style (bitmap or vector), and bit depth.

This section provides information to help you decide which graphics format to use when exporting your figure to a file or to the Windows clipboard. It covers

Before deciding on a graphics format, check what formats are supported by your target application and platform. See the `print` reference page for a complete list of graphics formats supported in MATLAB. Once you decide on which format to use in exporting your figure, follow the instructions in "Exporting to a File" on page 7-21 or "Exporting to the Windows or Macintosh Clipboard" on page 7-32.

Frequently Used Graphics Formats

Here are some of the more frequently used graphics formats. For a complete list, see the Graphics Format table on the `print` reference page. For a more complete description of these formats, see "Description of Selected Graphics Formats" on page 7-80.

Format	Description	Command Line -device Parameter
EPS color, and black and white	Export line plots or simple graphs to a file. **Note** An EPS file does not display within some applications unless you add a TIFF preview image to it. See the example "Exporting in EPS Format with a TIFF Preview" on page 7-40.	`-deps` (black and white) `-depsc` (color) `-depsc -tiff` (TIFF preview)
JPEG 24-bit	Export plots with surface lighting or transparency to a file. This format can be displayed by most Web browsers.	`-djpeg` `-djpeg`*`number`*, where *`number`* is the compression.
TIFF 24-bit bitmap color	Export plots with surface lighting or transparency to a file. Widely available. A good format to choose if you are not sure what formats your application supports.	`-dtiff`
BMP 8-bit color bitmap	Export a figure to the clipboard (Windows only).	`-dbitmap`
EMF color vector format	Export a figure to the clipboard (Windows only).	`-dmeta`

Factors to Consider in Choosing a Format

There are at least five main factors to consider when choosing a graphics format to use in exporting a figure:

- Implementation — Is it a built-in MATLAB or Ghostscript format?
- Graphics Format — Is it bitmap or vector graphics format?
- Bit Depth — What bit depth does the format offer?
- Color Support — What color support does it have?
- Model/Publication — Is it a Simulink® model or specific publication type?

The Graphics Format table on the `print` reference page provides information on the first four of these factors for each format that MATLAB supports.

Built-In MATLAB or Ghostscript Formats

Some graphics formats are built-in MATLAB formats and others are provided by Ghostscript. In some cases (such as the Windows Bitmap format), the format is available both as a built-in format and a Ghostscript format. In general, when this is the case, we recommend that you choose the MATLAB format, especially if you plan to read the image back into MATLAB later.

The choice of MATLAB versus Ghostscript is important when any of these properties affects your output:

- "Font Support" on page 7-77
- "Resolution" on page 7-78
- "Importing into MATLAB" on page 7-78

Choosing Bitmap or Vector Graphic Output

MATLAB file formats are created using either bitmap or vector graphics. Bitmap formats store graphics as 2-D arrays of pixels. Vector formats use drawing commands to store graphics as geometric objects. Whether to use a bitmap or vector format depends mostly on the type of objects in your figure.

The choice of bitmap versus vector graphics is important when any of these properties or capabilities affects your output:

- "Degree of Complexity" on page 7-78
- "Lighting and Transparency" on page 7-78

- Line and text quality
- "File Size" on page 7-79
- "Resizing After Import" on page 7-79

To create vector output, the Painters renderer is required. Under some circumstances you might need to manually select it in the Print Preview or Export Setup GUI. The painters renderer does not support lighting or transparency.

To create bitmap output, either the OpenGL or the Z-buffer renderer is required. Under some circumstances you might need to manually select one of these in the Print Preview or Export Setup GUI. These renderers both support lighting, but only OpenGL supports transparency.

See "Impact of Rendering Method on the Output" on page 7-80 for more information.

Bit Depth

Bit depth is the number of bits a format uses to store each pixel. This determines the number of colors the exported figure can contain.

Bit depth applies mostly to bitmap graphics. An 8-bit image uses 8 bits per pixel (bpp), enabling it to define 2^8, or 256, unique colors. The other supported bit depths are 1-bit (2 colors), 4-bit (16 colors), and 24-bit (16 million colors).

In vector files that don't normally have a bit depth, the color of objects is specified by drawing commands stored in the file. However, vector files can contain bitmaps under the following conditions:

- Image objects saved in vector formats are always saved as bitmaps, regardless of the rendering method used.
- For vector files created using the OpenGL or Z-buffer renderer, everything in the figure is saved as a bitmap.

The Graphics Format table on the `print` reference page indicates the bit depth of each format. If file size is not critical, make sure you choose a format with a bit depth that supports the number of colors or shades of gray in your displayed figure.

Color Support

Each graphics format can produce color, grayscale, or monochrome output. Check the Graphics Format table to see the level of color support for each format type. To preserve the color in your exported file, you must select a color graphics format. Bit depth also affects color.

Exporting Simulink Models

Simulink models can only be exported to EPS or a Ghostscript format. Note that you can only use the `print` function to export a model, not the Export dialog box.

High Resolution or Web Publications

If you want to use a figure in a journal or other publication, use a format that enables you to set a high resolution, such as TIFF or EPS.

If you want to use a figure in a Web publication, use either the PNG or the JPEG format. If you need to save an image as a GIF file, you can use the `imwrite` function. You need to convert M-by-N-by-3 truecolor CData (such as the `getframe` function provides) to an M–by–N 8–bit array and a colormap in order to write a GIF. Alternatively, you can export your figure as a TIFF file and convert it to a GIF using another software application, or capture a figure as an image using a screen capture utility and save it in formats the utility supports.

Properties Affected by Choice of Format

The figure properties listed in this section are affected when you select a graphics format when exporting to a file or the Windows clipboard.

Font Support

Ghostscript formats support a limited number of fonts. If you use an unsupported font, MATLAB substitutes Courier. See “PostScript and Ghostscript Supported Fonts” on page 7-88 for more information.

Resolution

Generally, higher resolution means higher quality. Your choice of resolution should be based in part on the device to which you will ultimately print it. Experimentation with different resolution settings can be helpful.

You cannot change the resolution of a Ghostscript format. The resolution is low (72 dpi) and might not be appropriate for publications.

Importing into MATLAB

If you want to read an exported figure back into MATLAB, it is best to use one of the built-in MATLAB formats. You should not use PostScript or a proprietary format such as Adobe Illustrator (`.ai`), Windows metafile (`.emf`), or portable document format (`.pdf`) files.

Degree of Complexity

Bitmaps are preferable for high-complexity plots, where complexity is determined by the number of polygons, the number of polygons with interpolated shading, the number of markers, the presence of truecolor images, and other factors. An example of a high-complexity plot is a surface plot that uses interpolated shading.

Vector formats are preferable for most 2-D plots and for some low-complexity surface plots.

Lighting and Transparency

Surface lighting and transparency are only supported by bitmap graphics formats. If you use a vector format, the lighting and transparency disappear. Note that of the two renderers intended for bitmaps (OpenGL and Z-buffer) only OpenGL supports transparency.

Note If you export to an EPS (vector) file using the Painters renderer and include a TIFF preview, the preview image is a bitmap and shows lighting or transparency when displayed on your screen. Remember that the underlying format vector file, which is what normally gets printed, does not support these features.

Lines and Text

Generally, vector formats create better lines and text than bitmap formats. MATLAB renderers do not antialias lines or text.

File Size

In general, bitmap formats produce smaller files for complex plots than vector formats, and vector formats produce smaller files for simple plots than bitmap formats.

You can calculate the size of a figure exported to an uncompressed bitmap by multiplying the figure size by its resolution and the bit depth of the chosen format. For example, if a figure is 2 inches by 3 inches and has a resolution of 100 dpi (dots per inch), it will consist of (2x100)x(3x100), or 60,000 pixels. If exported to an 8-bit file, it uses 480,000 bits, or 60 KB. If exported to a 24-bit file, it uses three times the number of bytes, or 180 KB.

Vector format file size is affected by the complexity and number of objects in your figure. As the complexity and number of objects increase, the number of drawing commands increases.

Resizing After Import

You can resize a vector graphics figure after importing it into another software application without losing quality. (Not all applications that support vector formats enable you to resize them.)

This is not true of bitmap formats. Resizing a bitmap causes round-off errors that result in jagged edges and degradation of picture quality. This degradation is particularly obvious in lines and text and is highly discouraged.

Color

The Graphics Format table on the `print` reference page indicates the color support and bit depth of each format. If file size is not critical, make sure you choose a format with a bit depth that supports the number of colors or shades of gray in your displayed figure.

Impact of Rendering Method on the Output

If you specify a bitmap format when exporting, the exported file always contains a bitmap regardless of your current renderer setting. If you have the renderer set to Painters, which normally produces a vector format, that setting is ignored under these circumstances.

Vector format files, however, can store your figure as a vector or bitmap graphic depending on the renderer used to export it. If you do not specify a rendering method and MATLAB chooses the OpenGL or Z-buffer renderer, your exported vector file contains a bitmap. If you want your figure exported as a vector graphic, be sure to set the rendering method to Painter's.

Description of Selected Graphics Formats

This section contains details about some of the export file formats MATLAB supports. For information about formats not listed here, consult a graphics file format reference.

Formats covered in this section are

Adobe Illustrator 88 Files

Adobe Illustrator (ILL) is a vector format that is fully compatible with Adobe Illustrator software. An Illustrator file created in MATLAB can be further processed with Adobe Illustrator running on any platform. (Note that when you view it in Illustrator, it has no template.)

By default, Illustrator files are color and saved in portrait orientation. The Illustrator `group` command is used to give the illustrations a hierarchy similar to that of the Handle Graphics or Simulink graphic.

Some limitations of the Illustrator format are

- Interpolated patches and surfaces cannot be created. The color of each polygon is determined by the average of the `CData` values for all of the polygon's vertices.
- Images cannot be exported in this format.
- The resolution setting of 72 dpi cannot be changed.
- No fonts are downloaded to the Illustrator file. Any unavailable fonts are replaced with fonts that are available.

EMF Files

Enhanced Metafiles (EMF) are vector files similar in nature to Encapsulated PostScript (EPS), capable of producing near publication-quality graphics. EMF is an excellent format to use if you plan to import your image into a Microsoft application and want the flexibility to edit and resize your image once it has been imported. It is the only MATLAB supported vector format you can edit from within a Microsoft application. (Note that your editing ability is limited. For the best results, do all your editing in MATLAB.)

A drawback of using EMF files is that they are generally only supported by Windows-based applications.

EPS Files

The Encapsulated PostScript (EPS) vector format is the most reliable and consistent file format MATLAB supports. It is widely recognized in desktop publishing and word processing packages on both UNIX and Windows platforms. EPS is the only MATLAB supported export format that can produce CMYK output. (PostScript printer drivers also support this feature.)

This format is your best choice for producing publication-quality graphics. It might not be appropriate for figures containing interpolated shading because it creates a very large file that is difficult to print. For such figures, use the TIFF format with a high-resolution setting. For more information about format choices, see "Choosing Bitmap or Vector Graphic Output" on page 7-75.

When imported into Microsoft applications, an EPS file does not display unless you add a TIFF preview image to it.

The preview image is simple to add (see the next section, “Creating a Preview Image”). However, if you print your file to a non-PostScript printer, the TIFF preview is used as the printed image. The resolution of the preview image is 72 dpi, resulting in much lower quality than the EPS image. If there is no preview image, your printout to a non-PostScript printer contains an error message in place of the graphic. Many high-end graphics packages, like Adobe Illustrator, can print an EPS file to a non-PostScript printer.

You cannot edit figures when using EPS files in Microsoft applications; they can only be annotated.

Note The best vector format to use with Microsoft applications is EMF. See “EMF Files” on page 7-81.

EPS format has limited font support. When MATLAB exports a graphic to the EPS file format, it does not try to determine whether the fonts you have used in your axes text objects are supported by the EPS format. Unsupported fonts are replaced with Courier.

Creating a Preview Image. You cannot create TIFF preview images using the graphical user interface. Use the `print` command with the `-tiff` switch. For example, to create an EPS Level 2 image with TIFF preview in file `myfile.eps`, type

```
print -depsc2 -tiff myfile.eps
```

TIFF Files

The Tagged Image File Format (TIFF) is a very widely used bitmap format and can produce publication-quality graphics if you use a high-resolution setting (such as 200 or 300 dpi).

TIFF is a good format to choose if you are not sure what formats your target application supports, or if you want to import the graphic into more than one application without having to export it to several different formats. It can also be imported into most image-processing applications and converted to other formats, if necessary. For example, the `print` command does not produce GIF files, but there are many applications that can convert TIFF files to GIF.

You can also use `getframe` to create a snapshot of a figure and `imwrite` to save that image as a GIF file.

JPEG Files

The Joint Photographic Experts Group (JPEG) bitmap format is one of the dominant formats used in Web graphics. The 24-bit version MATLAB supports more colors than the popular GIF format.

JPEG files always use JPEG compression. This is a lossy compression scheme, meaning that some data is thrown away during compression. When you export to a JPEG image, you can set the amount of compression to use. The more compression you use, the more data is thrown away. The compression amount is referred to as JPEG quality, where the highest setting results in the highest quality image, but the lowest amount of compression.

Setting JPEG Quality. You cannot set the quality using the graphical user interface. Use the `print` command with the `-djpeg` format switch, including the desired quality value as a suffix. This example exports to a JPEG file using a quality setting of 100.

```
print -djpeg100 myfile.jpg
```

By default, MATLAB uses a quality setting of 75. Possible values are from 1 to 100. Note that the highest setting of 100 still results in some data loss, although the result is usually visually indistinguishable from the original.

How to Specify a Format for Exporting

To select a graphics format to use when exporting, choose a format from the Graphics Format table on the `print` reference page, and specify that format in either the Export dialog box or in the MATLAB `print` function.

Using the Graphical User Interface

When exporting your figure to a file:

1. Select **Export** from the figure window's **File** menu.
2. Select a format from the **Save as type** list box.

3 Enter the filename you want to use and browse for the directory to save the file in.

4 Click **Save**.

Using MATLAB Commands

To specify a nondefault graphics format for the figure you are exporting, include the `-d` switch with the `print` command. For example, to export the current figure to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, type

```
print -r600 -depsc spline2d
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Choosing a Printer Driver

In this section...

What Are Printer Drivers?

A MATLAB printer driver formats your figure into instructions that your printer understands. There are two main types of MATLAB printer drivers: built-in MATLAB, and Ghostscript. See the Printer Driver table on the `print` reference page for a complete list of supported drivers. Specifying the printer driver does not change the selected printer. This following sections provide information to help you decide which printer driver to use when printing your figure.

Built-in MATLAB Drivers

Built-in MATLAB drivers are written specifically for MATLAB and include Windows, PostScript, and HPGL.

MATLAB provides built-in Windows printer drivers so that your print requests can work with the Windows Print Manager. The Print Manager enables you to monitor printer queues and control various aspects of the printing process.

HPGL support is provided for the HP 7475A plotter and fully compatible plotters. HPGL files can also be imported into documents of other applications, such as Microsoft Word, although add-on filters for them may be needed.

Ghostscript Drivers

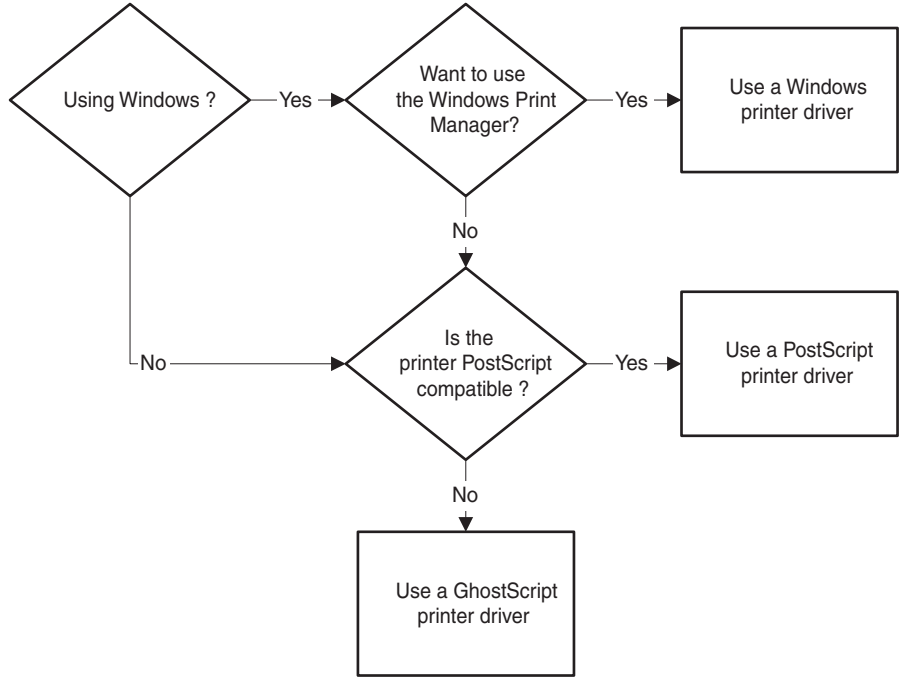
Ghostscript drivers use Ghostscript to convert your figure into printer-model-specific instructions. MATLAB generates a PostScript representation of the figure and Ghostscript generates the printer instructions from that. Examples of Ghostscript drivers are Epson and HP.

Factors to Consider in Choosing a Driver

The choice of printer driver depends upon several considerations:

- What platform you are using
- What kind of printer you have
- What color model you want to use
- What font support you need
- Any driver-specific settings you need

The following flowchart gives an overview of how to choose a driver based on the platform you are using and the type of printer you have.



Deciding What Type of Printer Driver to Use

Platform Considerations

On Windows, you can use any of the driver types shown in the flowchart. If you use the Windows driver, you can use the Windows Print Manager.

On UNIX, you can use either PostScript or Ghostscript drivers.

On either platform, if you have a PostScript-compatible printer, it is better to use a PostScript driver than a Ghostscript driver, because doing so avoids the unnecessary Ghostscript conversion step and is likely to create more accurate renditions.

Printer Type

Printer support is different among the Windows, PostScript, and Ghostscript drivers. Consult the manual for your printer to see what driver to use.

Windows drivers support most printer models, but sometimes the printer's native driver is incompatible with the MATLAB Windows driver. If you are getting printing errors, see "Trouble with Native Drivers on Windows" on page 7-90.

Some Ghostscript drivers are specific to certain printer models. For example, MATLAB provides different drivers to support the HP DeskJet 500, 500C, and 550C models, plus a generic driver for the series. When this is the case, try the model-specific driver first. If that doesn't work, try the generic driver.

Color Model

By default, MATLAB uses a black-and-white driver. The built-in MATLAB and Ghostscript drivers print both color and black and white. The Printer Drivers table on the `print` reference page indicates which drivers are color.

Colored surfaces and images print in grayscale when you use a black-and-white driver. Colored lines and text can be printed in color, grayscale, or black and white, depending on the color support of the driver and color capability of your printer. Results can vary depending on whether images, text, lines, patches, or surfaces are being printed.

Font Support

In MATLAB, the fonts supported for printing depend upon the MATLAB printer driver you specify and sometimes upon which platform you are using.

PostScript and Ghostscript Supported Fonts. The table below lists the fonts supported by the MATLAB PostScript and Ghostscript drivers. This same set of fonts is supported on both Windows and UNIX. If you use a font that is not on this list, it is replaced with Courier.

AvantGarde	Helvetica-Narrow	Times-Roman
Bookman	NewCenturySchlbk	ZapfChancery
Courier	Palatino	ZapfDingbats
Helvetica	Symbol	

If you set the font using the `set` function, use the names exactly as shown above. This example sets the font of the current text object to Helvetica-Narrow using MATLAB commands.

```
set(gca, 'FontName', 'Helvetica-Narrow');
```

If you use the Property Editor dialog box (available under **Axes Properties** or **Current Object Properties** on the **Edit** menu) to set the font, the list of available fonts shows those that are supported by your system. If you choose one that is not in the table above, your resulting file uses `Courier`.

Windows Drivers Supported Fonts. The MATLAB Windows drivers support any system-supported font. To see the list of fonts installed on your system, open the **Font name** list on the **Text** or **Style** tab of the Property Editor.

If you use the `set` function to set fonts, type in the name exactly as it appears in the Property Editor. For example, if you have the Script font installed on your system, set the title of your figure to `Script` using the following code.

```
h = get(gca, 'Title');
set(h, 'FontName', 'Script');
```

If you specify a font supplied with MATLAB that is not available on your platform as a system font, figures might not print or export properly.

HPGL Driver Supported Fonts. HPGL drivers support only one font. However, you can set its size and color.

Settings That Are Driver Specific

Some print settings are only supported by specific drivers. This table summarizes the settings and which driver supports them.

Setting	Driver(s)
Appending figures to a PostScript file	PostScript
BoundingBox (setting figure to print uncropped)	PostScript, Ghostscript
CMYK	PostScript
Resolution set with user interface	PostScript, Windows
Resolution set with `print` function	PostScript

Driver-Specific Information

This section provides additional information about the various types of printer drivers available to MATLAB users. It covers the following topics:

Setting the Windows Driver

When you specify a Windows driver (`-dwin` or `-dwinc`), MATLAB interprets this to mean that the print request will use the Windows Print Manager. It also means that MATLAB will assign the default Windows driver based on your current printer's color property setting. In other words, MATLAB does not differentiate between `-dwin` or `-dwinc` in `printopt.m` and you might not get the expected output color: if you choose `-dwin`, lines and text will print in black and white; with `-dwinc`, lines and text print in their screen colors (assuming your printer does print in color).

There are two ways to ensure that MATLAB uses `-dwin` or `-dwinc`: specify the driver when you print, or use the printer's Document Properties dialog box to set the default driver.

You can use the printer's Document Properties dialog box to set the default driver for all print requests. This dialog box sets the printer's color property, which in turn sets the default Windows driver.

To access this dialog box, click the **Properties** button on the Windows Print or Print Setup dialog box. See your Windows and printer's documentation if you need help with this dialog box. Document Properties dialog boxes vary from printer to printer.

Sometimes, even when you use the Windows Document Properties dialog box, you can receive incorrect color results because some Windows printers return inaccurate information about their color property setting.

Trouble with Native Drivers on Windows

Occasionally, printing problems are due to a bug in the native printer driver or an incompatibility between the native printer driver and the MATLAB driver.

If you are having trouble, try installing a different native printer driver. A newer version might be available from the manufacturer or reseller. You may also be able to use the native driver from a different printer, such as an earlier model from the same manufacturer.

If this doesn't help, try using a PostScript or Ghostscript driver.

Level 1 or Level 2 PostScript Drivers

Choosing between the Level 1 and Level 2 MATLAB PostScript drivers does not affect the quality of your output. Make the choice based on what your printer supports and on any file size or speed concerns.

Level 1 PostScript produces good results on a Level 2 printer, but Level 2 PostScript does not print properly on a Level 1 printer.

Level 2 PostScript files are generally smaller and render more quickly than Level 1 files. If your printer supports Level 2 PostScript, use one of the Level 2 drivers. If your printer does not support Level 2, or if you're not sure, use a Level 1 driver.

Early PostScript 1 Printers

If you have an early PostScript 1 printer, such as some of the PostScript printers manufactured before 1990, you may notice problems in the text of MATLAB printouts. Your printer might not support the `ISOLatin1Encoding` operator that MATLAB uses for PostScript files. If this is the case, use Adobe's PostScript default character-set encoding. You can specify this by using the `-adobecset` option with the `print` command.

Background Fills in HPGL Drivers

The HPGL driver cannot do background fills. Therefore, you should ensure that your figure is set to print with a white background (the default), and that any lines and text in your figure are drawn in a color dark enough to be seen on a white background. For more information about background color, see "Setting the Background Color" on page 7-62.

Color Selection in HPGL Drivers

The HP 7475A plotter supports six pens, none of which can be white. If MATLAB tries to draw in white while rendering in HPGL mode, the driver ignores all drawing commands until a different color is chosen.

Pen 1, which is assumed to be black, is used for drawing axes. The remaining pens are used for the first five colors specified in the `ColorOrder` property of the current axes object. If `ColorOrder` specifies fewer than five colors, the unspecified pens are not used.

For Simulink systems, which ordinarily use a maximum of eight colors, the six pens available on the plotter are assumed to be

- Pen 1: black
- Pen 2: red
- Pen 3: green
- Pen 4: blue
- Pen 5: cyan
- Pen 6: magenta

If you attempt to draw a MATLAB object containing a color that is not a known pen color, the driver chooses the nearest approximation to the unlisted color.

Limitations of HPGL Drivers

The HPGL driver has these limitations:

- Display colors and plotted colors sometimes differ.
- Areas (faces on mesh and surface plots, patches, blocks, and arrowheads) are not filled.
- There is no hidden line or surface removal.
- Text is printed in the plotter's default font.
- Line width is determined by pen width.
- Images and UI controls cannot be plotted.
- Interpolated edge lines between two vertices are drawn with the pen whose color best matches the average color of the two vertices.
- Figures cannot be rendered using Z-buffer or OpenGL; this driver always uses the Painter's algorithm.

How to Specify the Printer Driver to Use

If you need to use a driver other than the default driver for your system, choose a new driver from the Printer Driver table on the `print` reference page, and set it either as a new default or just for the current figure you are working on.

Setting the Default Driver for All Figures

If you do not indicate a specific printer driver, MATLAB uses the default driver specified by the variable `dev` in the `printopt.m` file. The factory default driver depends on the platform.

Platform	Factory Default Printer Driver	Driver Code
Windows	Black-and-white Windows	`-dwin`
UNIX & MAC	Black-and-white Level II PostScript	`-dps2`

To change the default driver for all figures, edit `printopt.m` and change the value for `dev` to match one of the driver codes listed in the Printer Drivers table on the `print` reference page (`printopt.m` contains instructions for modifying it). See "Setting Defaults Across Sessions" on page 7-9 for details.

Setting a Driver for the Current Figure Only

You can change the printer driver from the MATLAB command line. To specify a nondefault printer driver for the figure you are printing, include the `-d` switch with the `print` command. For example, to print the current figure using the MATLAB built-in Windows color printer driver `winc`, type

```
print -dwinc
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Troubleshooting

In this section...

Introduction

This section describes some common problems you might encounter when printing or exporting your figure. If you don't find your problem listed here, try searching the Knowledge Base maintained by the MathWorks Technical Support Department. Go to `http://www.mathworks.com/support` and enter a topic in the search field.

Common Problems

- **Printing Problems**
  - "Printer Drivers" on page 7-95
  - "Default Settings" on page 7-96
  - "Color vs. Black and White" on page 7-97
  - "Printer Selection" on page 7-97
  - "Rotated Text" on page 7-98
  - "ResizeFcn Warning" on page 7-98
- **Exporting Problems**
  - "Background Color" on page 7-98
  - "Default Settings" on page 7-99
  - "Microsoft Word" on page 7-99
  - "File Format" on page 7-100

Printing Problems

Printer Drivers

I am using a Windows printer driver and encountering problems such as segmentation violations, general protection faults, application errors, and unexpected output.

Try one of the following solutions:

- Check the table of `drivers` in the `print` reference page to see if there are other drivers you can try.
  - If your printer is PostScript compatible, try printing with one of the MATLAB built-in PostScript drivers.
  - If your printer is not PostScript compatible, see if one of the MATLAB built-in Ghostscript devices is appropriate for your printer model. These devices use Ghostscript to convert PostScript files into other formats, such as HP LaserJet and Canon BubbleJet.
- Contact the printer vendor to obtain a different native printer driver. The behavior you are experiencing might occur only with certain versions of the native printer driver. If this doesn't help and you are using Windows, try

reinstalling the drivers that were shipped with your Windows installation disk.

- Export the figure to a graphics-format file, and then import it into another application before printing it. For information about exporting figures with MATLAB, see “Exporting to a File” on page 7-21.

PostScript Output

When I use the print function with the `-deps` switch, I receive this error message.

```
Encapsulated PostScript files cannot be sent to the printer.
File saved to disk under name 'figure2.eps'
```

As the error message indicates, your figure was saved to a file. EPS is a graphics file format and cannot be sent to a printer using a printer driver. To send your figure directly to a printer, try using one of the PostScript driver switches. See the table of drivers in the `print` reference page. To print an EPS file, you must first import it into a word processor or other software program.

Default Settings

My printer uses a different default paper type than the MATLAB default type of letter. How can I change the default paper type so that I don’t have to set it for each new figure?

You can set the default value for any property by adding a line to `startup.m`. Adding the following line sets the default paper type to A4.

```
set(0, 'DefaultFigurePaperType', 'A4');
```

In your call to `set`, combine the word `Default` with the name of the object `Figure` and the property name `PaperType`.

I set the paper orientation to landscape, but each time I go to print a new figure, the orientation setting is portrait again. How can I change the default orientation so that I won’t have to set it for each new figure?

See the explanation for the previous question. Adding the following line to `startup.m` sets the default paper orientation to landscape.

```
set(0, 'DefaultFigurePaperOrient', 'landscape')
```

Color vs. Black and White

I want the lines in my figure to print in black, but they keep printing in color.

You must be using a color printer driver. You can specify a black-and-white driver using the `print` function or the Print Preview dialog box to force the lines for the current figure to print in black. See “Setting the Line and Text Color” on page 7-66 for instructions.

A white line in my figure keeps coming out black when I print it.

There are two things that can cause this to happen. Most likely, the line is positioned over a dark background. By default, MATLAB inverts your background to white when you print, and changes any white lines over the background to black. To avoid this, retain your background color when you print. See “Setting the Background Color” on page 7-62.

The other possibility is that you are using a Windows printer driver and the printer is sending inaccurate color information to MATLAB. See .

I am using a color printer, but my figure keeps printing in black and white.

By default, MATLAB uses a black-and-white printer driver. You need to specify a color printer driver. For instructions, see “Choosing a Printer Driver” on page 7-85. If you are already using a Windows color driver, the printer might be returning inaccurate information about its color property. See .

Printer Selection

I have more than one printer connected to my system. How do I specify which one to print my figure with?

You can use either the Print dialog box, or the MATLAB `print` function, specifying the printer with the `-P` switch. For instructions using either method, see "Selecting the Printer" on page 7-46.

Rotated Text

I have some rotated text in my figure. It looks fine on the screen, but when I print it, the resolution is poor.

You are probably using bitmapped fonts, which don't rotate well. Try using TrueType fonts instead.

ResizeFcn Warning

I get a warning about my `ResizeFcn` being used when I print my figure.

By default, MATLAB resizes your figure when converting it to printer coordinates. Therefore, MATLAB calls any `ResizeFcn` you have created for the figure and issues a warning. You can avoid this warning by setting the figure to print at screen size.

Exporting Problems

Background Color

I generated a figure with a black background and selected "Use figure color" from the Copy Options panel of the Preferences dialog box. But when I exported my figure, its background was changed to white.

You must have exported your figure to a file. The settings in **Copy Options** only apply to figures copied to the clipboard.

There are two ways to retain the displayed background color: use the Print Preview dialog box or set the `InvertHardCopy` property to `off`. See "Setting the Background Color" on page 7-62 for instructions on either method.

Default Settings

I want to export all of my figures using the same size. Is there some way to do this so that I don't have to set the size for each individual figure?

You can set the default value for any property by adding a line to `startup.m`. Adding the following line sets the default figure size to 4-by-3 inches.

```
set(0, 'DefaultFigurePaperPosition', [0 0 4 3]);
```

In your call to `set`, combine the word `Default` with the name of the object `Figure` and the property name `PaperPosition`.

I use the clipboard to export my figures as metafiles. Is there some way to force all of my copy operations to use the metafile format?

On Windows, use the **Copy Options** panel of the Preferences dialog box. Any settings made here, including whether MATLAB copies your figure as a metafile or bitmap, apply to all copy operations. See "Exporting to the Windows or Macintosh Clipboard" on page 7-32 for instructions.

Microsoft Word

I exported my figure to an EPS file, and then tried to import it into my Word document. My printout has an empty frame with an error message saying that my EPS picture was not saved with a preview and will only print to a PostScript printer. How do I include a TIFF preview?

Use the `print` command with the `-tiff` switch. For example,

```
print -deps -tiff filename
```

Note that if you print to a non-PostScript printer with Word, the preview image is used for printing. This is a low-resolution image that lacks the quality of an EPS graphic. For more information about preview images and other aspects of EPS files, see "EPS Files" on page 7-81.

When I try to resize my figure in Word, its quality suffers.

You must have used a bitmap format. Bitmap files generally do not resize well. If you are going to export using a bitmap format, try to set the figure's size while it's still in MATLAB. See "Setting the Figure Size and Position" on page 7-47 for instructions.

As an alternative, you can use one of the vector formats, EMF or EPS. Figures exported in these formats can be resized in Word without affecting quality.

I exported my figure as an EMF to the clipboard. When I paste it into Word, some of the labels are printed incorrectly.

This problem occurs with some versions of Word and Windows. Try editing the labels in Word.

File Format

I tried to import my exported figure into a word processing document, but I got an error saying the file format is unrecognized.

There are two likely causes: you used the `print` function and forgot to specify the export format, or your word processing program does not support the export format. Include a format switch when you use the `print` function; simply including the file extension is not sufficient. For instructions, see "Exporting to a File" on page 7-21.

If this does not solve your problem, check what formats the word processor supports.

I tried to append a figure to an EPS file, and received an error message

You cannot append figures to an EPS file. The `-append` option is only valid for PostScript files, which should not be confused with EPS files. PostScript is a printer driver; EPS is a graphics file format.

Of the supported export formats, only HDF supports storing multiple figures, but you must use the `imwrite` function to append them. For an example, see the reference page for `imwrite`.

Size of Exported File

I've always used the EPS format to export my figures, but recently it started to generate huge files. Some of my files are now several megabytes!

Your graphics have probably become complicated enough that MATLAB is using the OpenGL or Z-buffer renderer instead of the Painter's renderer. It does this to improve display time or to handle attributes that Painter's cannot, such as lighting. However, using OpenGL or Z-buffer causes a bitmap to be stored in your EPS file, which with large figures leads to a large file.

There are two ways to fix the problem. You can specify the Painter's renderer when you export to EPS, or you can use a bitmap format, such as TIFF. The best renderer and type of format to use depend upon the figure. See "Choosing Bitmap or Vector Graphic Output" on page 7-75 if you need help deciding. For information about the rendering methods and how to set them, see "Selecting a Renderer" on page 7-54.

Making Movies

I am processing a large number of frames in MATLAB. I would like these frames to be saved as individual files for later conversion into a movie. How can I do this?

Use `getframe` to capture the frames, `imwrite` to write them to a file, and `movie` to create a movie from the files. For more information about using `getframe` and `imwrite` to capture and write the frames, see "Exporting with getframe" on page 7-29. For more information about creating a movie from the captured frames, see the reference page for `movie`.

You can also save multiple figures to an AVI file. AVI files can be used for animated sequences that do not need MATLAB to run. However, they do require an AVI viewer. For more information, see "Exporting Audio/Video Data" in the MATLAB Programming documentation.

Extended Operations

There are some export operations that cannot be performed using the Export dialog box.

You need to use the `print` function to do any of the following operations:

- Export to a supported file format not listed in the Export dialog box. The formats not available from the Export dialog box include HDF, some variations of BMP and PCX, and the raw data versions of PBM, PGM, and PPM.
- Specify a resolution.
- Specify one of the following options:
  - TIFF preview
  - Loose bounding box for EPS files
  - Compression quality for JPEG files
  - CMYK output on Windows
- Perform batch exporting.

General Problems

Background Color

When I output my figure, its background is changed to white. How can I get it to have the displayed background color?

By default, when you print or export a figure, MATLAB inverts the background color to white. There are two ways to retain the displayed background color: use the Print Preview dialog box or set the `InvertHardCopy` property to `off`. See “Setting the Background Color” on page 7-62 for instructions on either method.

If you are exporting your figure to the clipboard, you can also use the **Copy Options** panel of the Preferences dialog box. Setting the background here sets it for all figures copied to the clipboard.

Default Settings

I need to produce diagrams for publications. There is a list of requirements that I must meet for size of the figure, fonts types, etc. How can I do this easily and consistently?

You can set the default value for any property by adding a line to `startup.m`. As an example, the following line sets the default axes label font size to 12.

```
set(0, 'DefaultAxesFontSize', 12);
```

In your call to `set`, combine the word `Default` with the name of the object `Axes` and the property name `FontSize`.

Dimensions of Output

The dimensions of my output are huge. How can I make it smaller?

Check your settings for figure size and resolution, both of which affect the output dimensions of your figure.

The default figure size is 8-by-6 inches. You can use the Print Preview dialog box or the `PaperPosition` property to set the figure size. See "Setting the Figure Size and Position" on page 7-47.

The default resolution depends on the export format or printer driver used. For example, built-in MATLAB bitmap formats, like TIFF, have a default resolution of 150 dpi. You can change the resolution by using the `print` function and the `-r` switch. For default resolution values and instructions on how to change them, see "Setting the Resolution" on page 7-57.

I selected "Auto (actual size, centered)" from the Print Preview menu, but my output looks a little bigger, and my font looks different.

You probably output your figure using a higher resolution than your screen uses. Set your resolution to be the same as the screen's.

As an alternative, if you are exporting your figure, see if your application enables you to select a resolution. If so, import the figure at the same resolution it was exported with. For more information about resolution and how to set it when exporting, see "Setting the Resolution" on page 7-57.

Axis and Tick Labels

When I resize my figure below a certain size, my x-axis label and the bottom half of the x-axis tick labels are missing from the output.

Your figure size might be too small to accommodate the labels. Labels are positioned a fixed distance from the *x*-axis. Since the *x*-axis itself is positioned a relative distance away from the window's edge, the label text might not fit. Try using a larger figure size or smaller fonts. For instructions on setting the size of your figure, see "Setting the Figure Size and Position" on page 7-47. For information about setting font size, see the Text Properties reference page.

In my output, the x-axis has fewer ticks than it did on the screen.

MATLAB has rescaled your ticks because the size of your output figure is different from its displayed size. There are two ways to prevent this: select **Keep screen limits and ticks** from the **Advanced** tab of the Print Preview dialog box, or set the `XTickMode`, `YTickMode`, and `ZTickMode` properties to `manual`. See "Setting the Axes Ticks and Limits" on page 7-60 for details.

UI Controls

My figure contains UI controls. How do I prevent them from appearing in my output?

Use the `print` function with the `-noui` switch. For details, see "Excluding User Interface Controls form Printed Output" on page 7-71.

Cropping

I can't output my figure using the uncropped setting (i.e., a loose BoundingBox).

Only PostScript printer drivers and the EPS export format support uncropped output. There is a workaround for Windows printer drivers, however. Using the `print` function, save your figure to a file that can be printed later. For an example see "Producing Uncropped Figures" on page 7-72.

Text Object Font

I have a problem with text objects when printing with a PostScript printer driver or exporting to EPS. The fonts are correct on the screen, but are changed in the output.

You have probably used a font that is not supported by EPS and PostScript. All unsupported fonts are converted to Courier. See “PostScript and Ghostscript Supported Fonts” on page 7-88 for the list of the supported fonts.

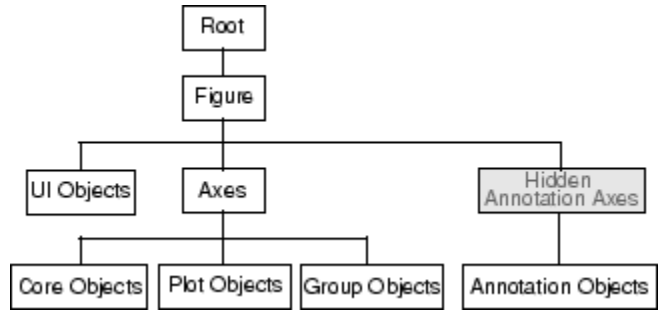
8

Handle Graphics Objects

Organization of Graphics Objects

Graphics objects are the basic drawing elements used by MATLAB to display data. Each instance of an object is associated with a unique identifier called a *handle*. Using this handle, you can manipulate the characteristics (called object *properties*) of an existing graphics object. You can also specify values for properties when you create a graphics object.

These objects are organized into a hierarchy, as shown by the following diagram.



The hierarchical nature of Handle Graphics is based on the interdependencies of the various graphics objects. For example, to draw a line object, MATLAB needs an axes object to orient and provide a frame of reference to the line. The axes, in turn, needs a figure window to display the axes and its child objects.

Types of Graphics Objects

In this section...

Introduction

There are two basic types of graphics objects:

- Core graphics objects — Used by high-level plotting functions and by composite objects to create plot objects
- Composite objects — Composed of core graphics objects that have been grouped together to provide a more convenient interface

Composite objects form the basis for four subcategories of graphics objects.

- Plot objects — Composed of basic graphics objects, but enable properties to be set on plot object level
- Annotation objects — Exist on a layer separate from other graphics objects
- Group objects — Create groups of objects that can behave as one in certain respects. You can parent any axes child object (except light) to a group object, including other group object.
- UI objects — User interface objects are used to construct graphical user interfaces.

Graphics objects are interdependent, so the graphics display typically contains a variety of objects that, in conjunction, produce a meaningful graph or picture.

Information on Specific Graphics Objects

See the following sections for more information on the various types of graphics objects:

- "Graphics Windows — the Figure" on page 8-6

For information on user interface objects and their application, see “Creating Graphical User Interfaces” in the MATLAB documentation.

Graphics Windows — the Figure

In this section...

Introduction

Figures are the windows in which MATLAB displays graphics. Figures contain menus, toolbars, user-interface objects, context menus, axes and, as axes children, all other types of graphics objects.

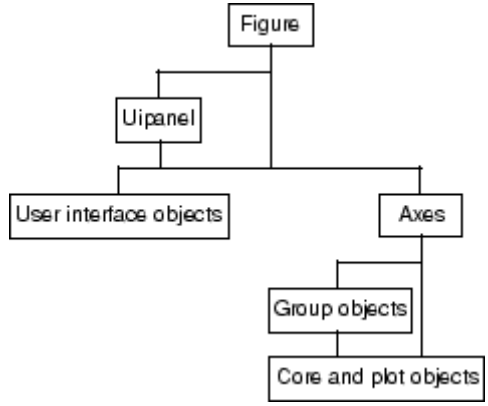
MATLAB places no limits on the number of figures you can create (your computer systems might impose limitations, however).

Figures play two distinct roles in MATLAB:

- Containing data graphs
- Containing graphical user interfaces

Figures can contain both graphs and GUIs components at the same time. For example, a GUI might be designed to plot data and therefore contain an axes as well as user interface objects. See "Example — Using Figure Panels" on page 8-90 for an example of such a GUI.

The following diagram illustrates the types of objects that figures can contain.



Note that both figures and axes have children that act as containers. A uipanel can contain user interface objects and be parented to a figure and group objects (hggroup and hgtransform) can contain axes children (except light objects) and be parented to an axes.

See “Objects That Can Contain Other Objects” on page 8-88 for more information.

Figures Used for Graphing Data

MATLAB functions that draw graphics (e.g., `plot` and `surf`) create figures automatically if none exist. If there are multiple figures open, one figure is always designated as the “current” figure, and is the target for graphics output.

The `gcf` command returns the handle of the current figure or creates a new figure if one does not exist. For example, you can enter the following command to see a list of figure properties.

```
get(gcf)
```

The root object `CurrentFigure` > property returns the handle of the current figure, if one exists, or returns empty if there are no figures open. For example,

```
get(0,'CurrentFigure')
ans =
[]
```

See “Controlling Graphics Output” on page 8-69 for more information on how MATLAB determines where to display graphics.

Figure Children for Graphs

Figures that display graphs need to contain axes to provide the frame of reference for objects such as lines and surfaces, which are used to represent data. These data representing objects can be contained in group objects or contained directly in the axes. See “Example — Transforming a Hierarchy of Objects” on page 8-36 for an example of how to use group objects.

Figures can contain multiple axes arranged in various locations within the figure and can be of various sizes. See “Automatic Axes Resize” on page 10-9 and “Multiple Axes per Figure” on page 10-15 for more information on axes.

Figures Used for GUIs

GUIs range from sophisticated applications to simple warning dialog boxes. You can modify many aspects of the figure to fit the intended use by setting figure properties. For example, the following figure properties are often useful when creating GUIs.

- Show or hide the figure menu, while displaying custom-designed menus (`MenuBar`)
- Change the figure title (`Name`)
- Control user access to the figure handle (`HandleVisibility`)
- Create a callback that executes whenever the user resizes the figure (`ResizeFcn`)
- Control display of the figure toolbar (`Toolbar`)
- Assign a context menu (`UIContextMenu`)
- Define callbacks that execute when users click drag or release the mouse over the figure (`WindowButtonDownFcn`, `WindowButtonMotionFcn`, `WindowButtonUpFcn`). Also see the `ButtonDownFcn` property.
- Specify whether the figure is modal (`WindowStyle`)

See the Figure Properties reference page for more information on figure characteristics you can specify.

See the “Creating Graphical User Interfaces” documentation for more information about using `figure` to create GUIs.

Root Object — the Figure Parent

The parent of a figure is the root object. You cannot instantiate the root object because its purpose is only to store information. It maintains information on the state of MATLAB, your computer system, and some MATLAB defaults.

There is only one root object and all other objects are its descendants. You do not create the root object; it exists when you start MATLAB. You can, however, set the values of root properties and thereby affect the graphics display.

For more information see Root Properties object properties.

More Information on Figures

See the `figure` reference page for information on creating figures.

See “Callback Properties for Graphics Objects” on page 8-107 for information on figure events for which you can define callbacks.

See Chapter 9, “Figure Properties” for information on other figure properties.

Core Graphics Objects

In this section...

Introduction

Core graphics objects include basic drawing primitives like line, text, and polygon shells (patch objects); specialized objects like surfaces, which are composed of a rectangular grid of vertices; images; and light objects, which are not visible but affect the way some objects are colored.

Axes contain objects that represent data, such as line, surfaces, contourgroups, etc.

The following table lists the core graphics objects and links to the reference pages of the functions used to create each object.

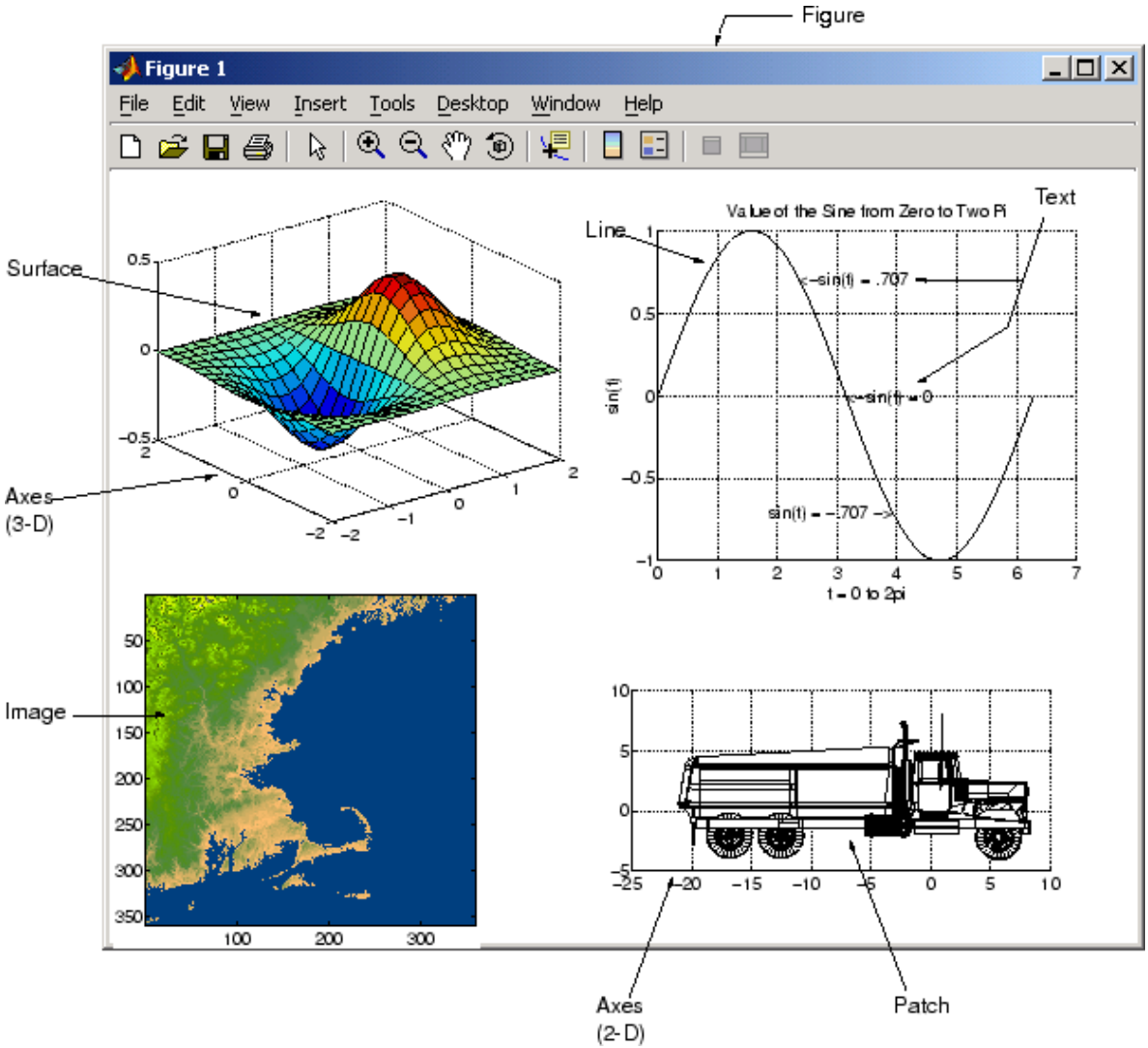
Core Graphics Objects

Function	Purpose
`axes`	Axes objects define the coordinate system for displaying graphs. Axes are always contained within a figure.
`image`	2-D representation of a matrix where numeric values are mapped to colors. Images can also be 3-D arrays of RGB values.
`light`	Directional light source located within the axes. Lights affect patches and surfaces, but cannot themselves be seen.

Core Graphics Objects (Continued)

Function	Purpose
`line`	A line is drawn by connecting the data points that define it.
`patch`	Filled polygons with separate edge properties. A single patch can contain multiple faces, each colored independently with solid or interpolated colors.
`rectangle`	2-D object that has settable edge and face color, and variable curvature (can draw ellipses)
`surface`	3-D grid of quadrilaterals created by plotting the value of each element in a matrix as a height above the *x-y* plane
`text`	Character strings positioned in the coordinate system defined by the axes

The following picture illustrates some typical core graphics objects.



Description of Core Graphics Objects

This section describes the core graphics objects.

Axes

Axes objects define a frame of reference in a figure window for the display objects that are generally defined by data. For example, MATLAB creates a line by connecting each data point with a line segment. The axes determines the location of each data point in the figure by defining axis scales (x, y, and z or radius and angle, etc.)

Axes are children of figures and are parents of core, plot, and group objects.

Note that, while annotation objects are also children of axes, they can be parented only to the hidden annotation axes (see the `annotation` function for more information).

All functions that draw graphics (e.g., `plot`, `surf`, `mesh`, and `bar`) create an axes object if one does not exist. If there are multiple axes within the figure, one axes is always designated as the "current" axes, and is the target for display of the above-mentioned graphics objects (uicontrols and uimenus are not children of axes).

Image

A MATLAB image consists of a data matrix and possibly a colormap. There are three basic image types that differ in the way that data matrix elements are interpreted as pixel colors — indexed, intensity, and truecolor. Since images are strictly 2-D, you can view them only at the default 2-D view.

Light

Light objects define light sources that affect all patch and surface objects within the axes. You cannot see lights, but you can set properties that control the style of light source, color, location, and other properties common to all graphics objects.

Line

Line objects are the basic graphics primitives used to create most 2-D and some 3-D plots. High-level functions `plot`, `plot3`, and `loglog` (and others)

create line objects. The coordinate system of the axes positions and orients the line.

Patch

Patch objects are filled polygons with edges. A single patch can contain multiple faces, each colored independently with solid or interpolated colors. `fill`, `fill3`, and `contour3` create patch objects. The coordinate system of the axes positions and orients the patch.

Rectangle

Rectangle objects are 2-D filled areas having a shape that can range from a rectangle to an ellipse. Rectangles are useful for creating flow-chart-type drawings.

Surface

Surface objects are 3-D representations of matrix data created by plotting the value of each matrix element as a height above the *x*-*y* plane. Surface plots are composed of quadrilaterals whose vertices are specified by the matrix data. MATLAB can draw surfaces with solid or interpolated colors or with only a mesh of lines connecting the points. The coordinate system of the axes positions and orients the surface.

The high-level function `pcolor` and the `surf` and `mesh` group of functions create surface objects.

Text

Text objects are character strings. The coordinate system of the parent axes positions the text. The high-level functions `title`, `xlabel`, `ylabel`, `zlabel`, and `gtext` create text objects.

Example — Creating Core Graphics Objects

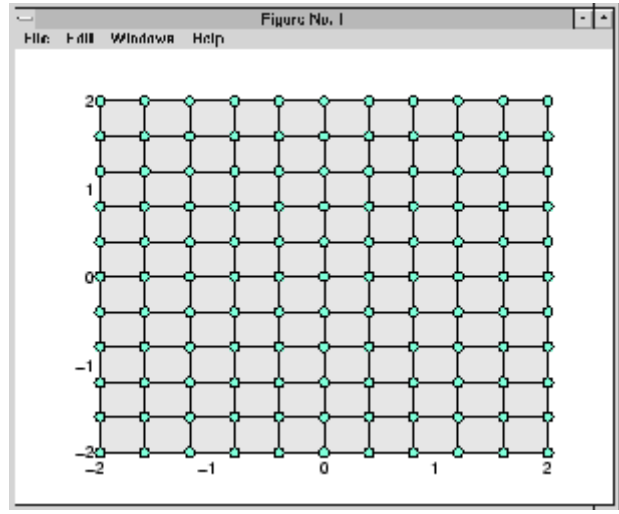
Object creation functions have a syntax of the form

```
handle = function('propertyname',propertyvalue,...)
```

You can specify a value for any object property (except those that are read only) by passing property name/value pairs as arguments. The function returns the handle of the object it creates, which you can use to query and modify properties after creating the object.

This example evaluates a mathematical function and creates three graphics objects using the property values specified as arguments to the `figure`, `axes`, and `surface` commands. MATLAB uses default values for all other properties.

```
[x,y] = meshgrid([-2:.4:2]);
Z = x.*exp(-x.^2-y.^2);
fh = figure('Position',[350 275 400 300],'Color','w');
ah = axes('Color',[.8 .8 .8],'XTick',[-2 -1 0 1 2],...
          'YTick',[-2 -1 0 1 2]);
sh = surface('XData',x,'YData',y,'ZData',Z,...
             'FaceColor',get(ah,'Color')+.1,...
             'EdgeColor','k','Marker','o',...
             'MarkerFaceColor',[.5 1 .85]);
```

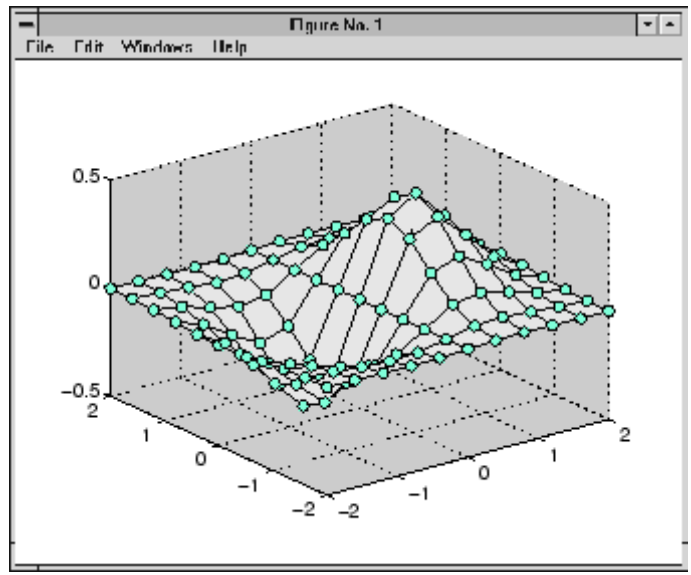


Note that the `surface` function does not use a 3-D view like the high-level `surf` functions. Object creation functions simply add new objects to the current axes without changing axes properties, except the `Children` property,

which now includes the new object and the axis limits (`XLim`, `YLim`, and `ZLim`), if necessary.

You can change the view using the camera commands or use the `view` command.

```
view(3)
```



Parenting

By default, all statements that create graphics objects do so in the current figure and the current axes (if the object is an axes child). However, you can specify the parent of an object when you create it. For example, the statement

```
axes('Parent',figure_handle,...)
```

creates an axes in the figure identified by `figure_handle`. You can also move an object from one parent to another by redefining its `Parent` property.

```
set(gca,'Parent',figure_handle)
```

High-Level Versus Low-Level

Many MATLAB graphics functions call the object creation functions to draw graphics objects. However, high-level routines also clear the axes or create a new figure, depending on the settings of the axes and figure `NextPlot` properties.

In contrast, core object creation functions simply create their respective graphics objects and place them in the current parent object. They do not respect the settings of the figure or axes `NextPlot` property.

For example, if you call the `line` function,

```
line('XData',x,'YData',y,'ZData',z,'Color','r')
```

MATLAB draws a red line in the current axes using the specified data values. If there is no axes, MATLAB creates one. If there is no figure window in which to create the axes, MATLAB creates it as well.

If you call the `line` function a second time, MATLAB draws the second line in the current axes without erasing the first line. This behavior is different from high-level functions like `plot` that delete graphics objects and reset all axes properties (except `Position` and `Units`). You can change the behavior of high-level functions by using the `hold` command or by changing the setting of the axes `NextPlot` property.

See “Controlling Graphics Output” on page 8-69 for more information on this behavior and on using the `NextPlot` property.

Simplified Calling Syntax

Object creation functions have convenience forms that allow you to use a simpler syntax. For example,

```
text(.5,.5,.5,'Hello')
```

is equivalent to

```
text('Position',[.5 .5 .5],'String','Hello')
```

Note that using the convenience form of an object creation function can cause subtle differences in behavior when compared to formal property name/property value syntax.

A Note About Property Names

By convention, MATLAB documentation capitalizes the first letter of each word that makes up a property name, such as `LineStyle` or `XTickLabelMode`. While this makes property names easier to read, MATLAB does not check for uppercase letters. In addition, you need use only enough letters to identify the name uniquely, so you can abbreviate most property names.

In M-files, however, using the full property name can prevent problems with futures releases of MATLAB if a shortened name is no longer unique because of the addition of new properties.

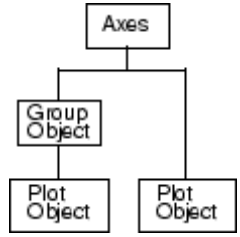
Plot Objects

In this section...

Introduction

A number of high-level plotting functions create plot objects. The properties of plot objects provide easy access to the important properties of the core graphics objects that the plot objects contain.

Plot object parents can be axes or group objects (`hggroup` or `hgtransform`). See “Objects That Can Contain Other Objects” on page 8-88 for examples.



This table lists the plot objects and the graphing functions that use them. Click object names to see a description of their properties.

Plot Objects

Object	Purpose
`areaseries`	Used to create `area` graphs
`barseries`	Used to create `bar` graphs
`contourgroup`	Used to create `contour` graphs
`errorbarseries`	Used to create `errorbar` graphs

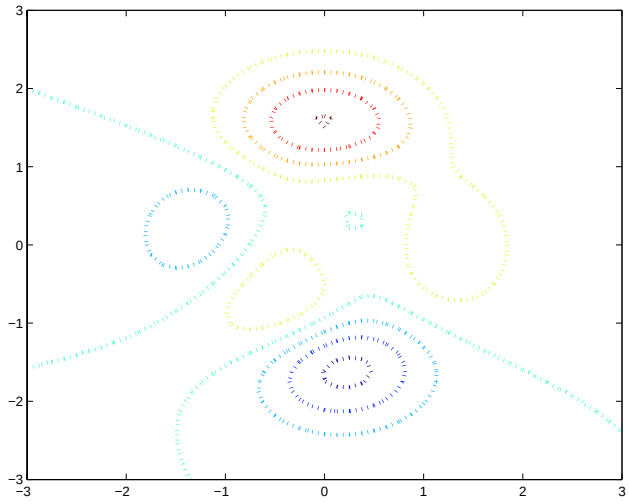
Plot Objects (Continued)

Object	Purpose
`lineseries`	Used by line plotting functions (`plot`, `plot3`, etc.)
`quivergroup`	Used to create `quiver` and `quiver3` graphs
`scattergroup`	Used to create `scatter` and `scatter3` graphs
`stairseries`	Used to create stairstep graphs (`stairs`)
`stemseries`	Used to create `stem` and `stem3` graphs
`surfaceplot`	Used by the `surf` and `mesh` group of functions

Creating a Plot Object

For example, the following statements create a contour graph of the `peaks` function and then set the line style and width of the contour lines.

```
[x,y,z] = peaks;
[c,h] = contour(x,y,z);
set(h,'LineWidth',3,'LineStyle',':')
```



The contour plot object enables you to set the line width and style of the contour graph by setting two properties. Looking at the core objects contained in the contour plot object reveals a number of patch objects whose edges are used to implement the contour line, which you would otherwise need to set individually.

```
child_handles = get(h,'Children');
get(child_handles,'Type')
ans =
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
    'patch'
```

Identifying Plot Objects Programmatically

Plot objects all return `hggroup` as the value of the `Type` property. If you want to be able to identify plot objects programmatically but do not have access to the object's handle, set a value for the object's `Tag` property.

For example, the following statements create a bar graph with five `barseries` objects and assign a different value for the `Tag` property on each object.

```
h = bar(rand(5));
set(h,{'Tag'},{'bar1','bar2','bar3','bar4','bar5'}')
```

Note that the cell array of property values must be transposed (`'`) to have the proper shape. See the `set` function for more information on setting properties.

No User Default Values

Note that you cannot define default values for plot objects.

Plot Objects and Backward Compatibility

Note The `v6` option discussed in this section is now obsolete and will be removed in a future version of MATLAB.

Plotting functions that create plot objects can introduce incompatibilities with code written before MATLAB Version 7.x. However, all plotting functions that return handles to plot objects support an optional argument (`'v6'`) that forces the functions to use core objects, as was the case in MATLAB before Version 7.

- See "Plot Objects" on page 8-19 for a list of functions that create plot objects.
- See "Core Graphics Objects" on page 8-10 for a list of core graphics objects.

Saving Figures That Are Compatible with Previous Version of MATLAB

Create backward-compatible FIG-files by following these two steps.

- Ensure that any plotting functions used to create the contents of the figure are called with the `'v6'` argument, where applicable.
- Use the `'-v6'` option with the `hgsave` command.

For example,

```
h = figure;
t = 0:pi/20:2*pi;
plot('v6',t,sin(t).*2)
hgsave(h,'myFigFile','-v6')
```

You can set a general MATLAB preference to ensure that figures saved by selecting the **Save** item under the figure **File** menu are backward compatible. To access MATLAB preferences, select **Preferences** from the Desktop **File** menu. Expand the **General** node and select **MAT Files**. Click **Ensure backward compatibility (-v6)**. Note that this setting affects all FIG-files and MAT-files that you create.

Linking Graphs to Variables — Data Source Properties

In this section...

Introduction

Plot objects enable you to link a MATLAB expression with properties that contain data. For example, the `lineseries` object has data source properties associated with the `XData`, `YData`, and `ZData` properties. These properties are called `XDataSource`, `YDataSource`, and `ZDataSource`.

To use a data source property,

1. Assign the name of a variable to the data source property that you want linked to an expression.
2. Calculate a new value for the variable.
3. Call the `refreshdata` function to update the plot object data.

`refreshdata` enables you to specify whether to use a variable in the base workspace or the workspace of the function from which you call `refreshdata`.

Data Source Example

The following example illustrates how to use this technique.

```
function datasource_ex
t = 0:pi/20:2*pi;
y = exp(sin(t));
h = plot(t,y,'YDataSource','y');
for k = 1:.1:10
 y = exp(sin(t.*k));
 refreshdata(h,'caller') % Evaluate y in the function workspace
 drawnow; pause(.1)
end
```

Changing the Size of Data Variables

If you change one data source property to a variable that contains data of a different dimension, you might cause the function to generate a warning and not render the graph until you have changed all data source properties to appropriate values.

Annotation Objects

In this section...

Introduction

Users typically create annotation objects from the Plot Edit toolbar or the **Insert** menu (select **Plot Edit** in the **View** menu to display the Plot Edit toolbar). However, you can also create annotation objects using the `annotation` function.

Annotation objects are created in a hidden axes that extends the full width and height of the figure. This enables you to specify the locations of annotation objects anywhere in the figure using normalized coordinates (the lower left corner is the point 0,0, the upper right corner is the point 1,1).

Annotation Object Properties

Note You should not change any of the properties of the annotation axes or parent any graphics objects to this axes. Use the `annotation` function or the graphics tools to create annotation objects.

The following links access descriptions of the properties you can set on the respective annotation objects.

- Annotation arrow properties
- Annotation doublearrow properties
- Annotation ellipse properties
- Annotation line properties
- Annotation rectangle properties

- Annotation textarrow properties
- Annotation textbox properties

To modify the appearance of annotation objects created with the plotting tools, use “The Property Editor” on page 1-28.

Example — Enclosing Subplots with an Annotation Rectangle

The following example shows how to create a rectangle annotation object and use it to highlight two subplots in a figure. This example uses the axes properties `Position` and `TightInset` to determine the location and size of the annotation rectangle.

1 First create an array of subplots.

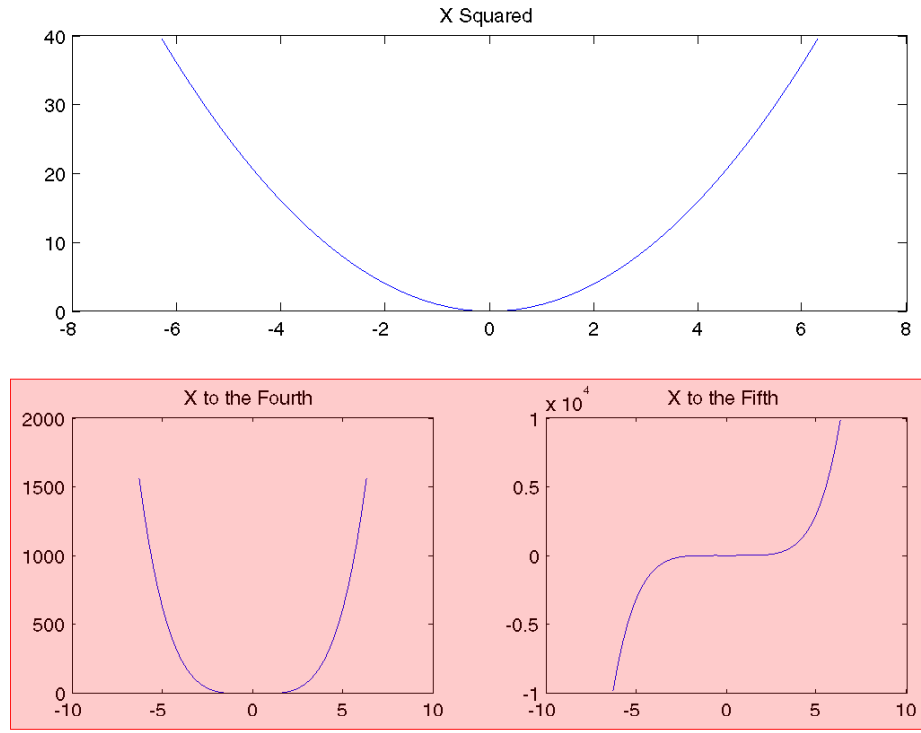
```
x = -2*pi:pi/12:2*pi;
y = x.^2;
subplot(2,2,1:2)
plot(x,y)
h1=subplot(223);
y = x.^4;
plot(x,y)
h2=subplot(224);
y = x.^5;
plot(x,y)
```

2 Determine the location and size of the annotation rectangle required to enclose axes, tick mark labels, and title using the axes `Position` and `TightInset` properties.

```
p1 = get(h1,'Position');
t1 = get(h1,'TightInset');
p2 = get(h2,'Position');
t2 = get(h2,'TightInset');
x1 = p1(1)-t1(1); y1 = p1(2)-t1(2);
x2 = p2(1)-t2(1); y2 = p2(2)-t2(2);
w = x2-x1+t1(1)+p2(3)+t2(3); h = p2(4)+t2(2)+t2(4);
```

3 Create the annotation rectangle to enclose the lower two subplots. Make the rectangle a translucent red with a solid border.

```
annotation('rectangle',[x1,y1,w,h],...
 'FaceAlpha',.2,'FaceColor','red','EdgeColor','red');
```



Group Objects

In this section...

Introduction

Group objects enable you to treat a number of axes child objects as one group. For example, you can make the entire group visible or invisible, select all objects when only one is clicked, or apply a transform matrix to reposition the objects by setting only one property on the group object.

There are two group objects:

- hggroup — Use when you want to create a group of objects and control the visibility or selectability of the group based on what happens to any individual object in the group. Create hggroup objects with the `hggroup` function.
- hgtransform — Use when you want to transform a group of objects. Transforms include rotation, translation, scaling, etc. See "Example — Transforming a Hierarchy of Objects" on page 8-36 for an example. Create hgtransform objects with the `hgtransform` function.

Note that the difference between the hggroup and hgtransform objects is ability of the hgtransform object to apply a transform matrix (via its `Matrix` property) to all objects for which it is the parent.

Note You cannot parent light objects to hggroup or hgtransform objects.

Creating a Group

You create a group by parenting axes children to an hggroup or hgtransform object. For example,

```
hb = bar(rand(5)); % creates 5 barseries objects
hg = hggroup;
set(hb,'Parent',hg) % parent the barseries to the hggroup
set(hg,'Visible','off') % makes all barseries invisible
```

Group objects can be the parent of any number of axes children, including other group objects.

Note Many plotting functions clear the axes (i.e., remove axes children) before drawing the graph. Clearing the axes also deletes any hggroup or hgtransform objects in the axes.

Transforming Objects

The hgtransform object's `Matrix` property enables you to apply a transform to all the hgtransform's children in unison. Typical transforms include rotation, translation, and scaling. You define a transform with a four-by-four transformation matrix, which is described in the following sections.

Creating a Transform Matrix

The `makehgtform` function simplifies the construction of matrices to perform rotation, translation, and scaling. See the "Example — Transforming a Hierarchy of Objects" on page 8-36 section for information on creating transform matrices using `makehgtform`.

Rotation

Rotation transforms rotate objects about the *x*-, *y*-, or *z*-axis, with positive angles rotating counterclockwise while sighting along the respective axis toward the origin. If the desired angle of rotation is [[THETA]], the following matrices define this rotation about the respective axis.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x & 0 \\ 0 & \sin\theta_x & \cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To create a transform matrix for rotation about an arbitrary axis, use the `makehgtform` function.

Translation

Translation transforms move objects with respect to their current locations. Specify the translation as distances t_x, t_y, and t_z in data space units. The following matrix shows the location of these elements in the transform matrix.

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Scaling transforms change the sizes of objects. Specify scale factors s_x, s_y, and s_z and construct the following matrix.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The Default Transform

The default transform is the identity matrix, which you can create with the `eye` function. Here is the identity matrix:

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

See “Undoing Transform Operations” on page 8-32 for related information.

Absolute vs. Relative Transforms

Transforms are specified in absolute terms, not relative to the current transform. For example, if you apply a transform that translates the hgtransform object 5 units in the *x* direction and then you apply another transform that translates it 4 units in the *y* direction, the resulting position of the object is 4 units in the *y* direction from its original position.

If you want transforms to accumulate, you must concatenate the individual transforms into a single matrix. See “Combining Transforms into One Matrix” on page 8-31 for more information.

Combining Transforms into One Matrix

It is usually more efficient to combine various transform operations into one matrix by concatenating (multiplying) the individual matrices and setting the `Matrix` property to the result. Note that matrix multiplication is not commutative, so the order in which you multiply the matrices affects the result. For example, suppose you want to perform an operation that scales, translates, and then rotates. You would multiply the matrices as follows:

```
C = R*T*S % operations are performed from right to left
```

where S is the scaling matrix, T is the translation matrix, R is the rotation matrix, and C is the composite of the three operations. You then set the hgtransform object’s `Matrix` property to C:

```
set(hgtransform_handle,'Matrix',C)
```

Note that the following sets of statements are not equivalent:

```
set(hgtransform_handle,'Matrix',C)% Transform as above
set(hgtransform_handle,'Matrix',eye(4) ) % Undo transform
```

versus

```
C = eye(4)*R*T*S % Multiply identity matrix as last step
set(hgtransform_handle,'Matrix',C)
```

Concatenating the identity matrix to other matrices has no effect on the composite matrix.

Undoing Transform Operations

Since transform operations are specified in absolute terms (not relative to the current transform), you can undo a series of transforms by setting the current transform to the identity matrix. For example, the following statement

```
set(hgtransform_handle,'Matrix',eye(4))
```

returns the object *hgtransform_handle* to its untransformed orientation.

Rotations Away From the Origin

Since rotations are performed about the origin, it is often necessary to translate the hgtransform object so that the desired axis of rotation is temporarily at the origin. After applying the rotation transform matrix, you then translate the hgtransform object back to its original position. The following example illustrates how to do this.

Suppose you want to rotate a surface about the *y*-axis at the center of the surface (the *y*-axis that passes through the point `x = 20` in this example).

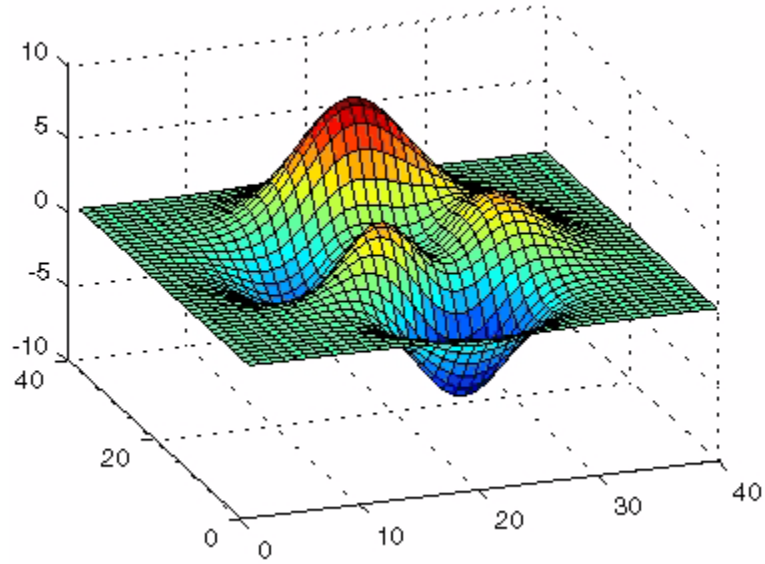
Note If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB editor.

---

1 Create a surface and an hgtransform object. Parent the surface to the hgtransform object.

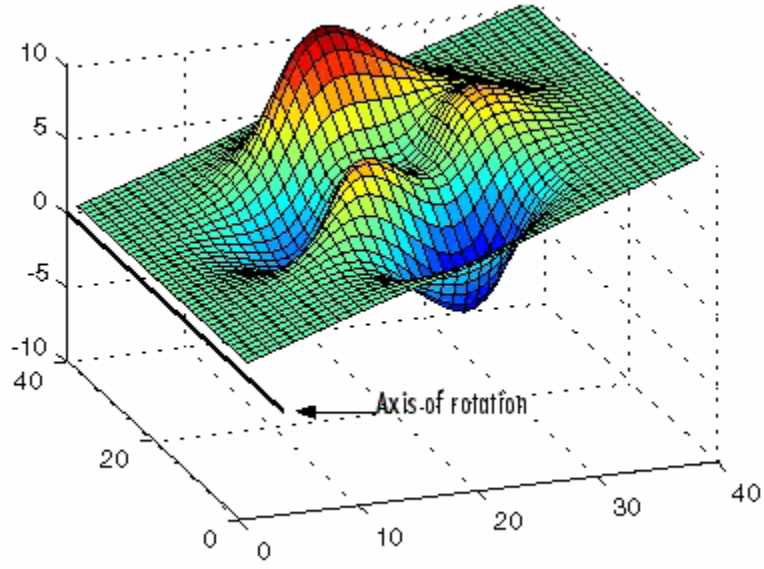
```
h = surf(peaks(40)); view(-20,30)
t = hgtransform;
set(h,'Parent',t)
```

The following picture shows the surface.



2 Create and set a *y*-axis rotation matrix to rotate the surface by -15 degrees.

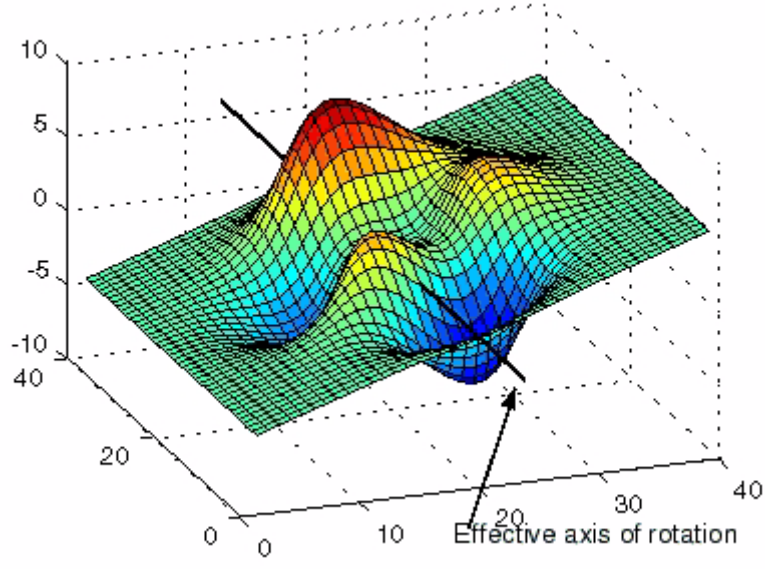
```
ry_angle = -15*pi/180; % Convert to radians
Ry = makehgtform('yrotate',ry_angle);
set(t,'Matrix',Ry)
```



Notice that the surface rotated -15 degrees about the *y*-axis that passes through the origin. However, to rotate about the *y*-axis that passes through the point `x = 20`, you must translate the surface in *x* by 20 units.

3 Create two translation matrices, one to translate the surface -20 units in *x* and another to translate 20 units back. Concatenate the two translation matrices with the rotation matrix in the correct order and set the transform.

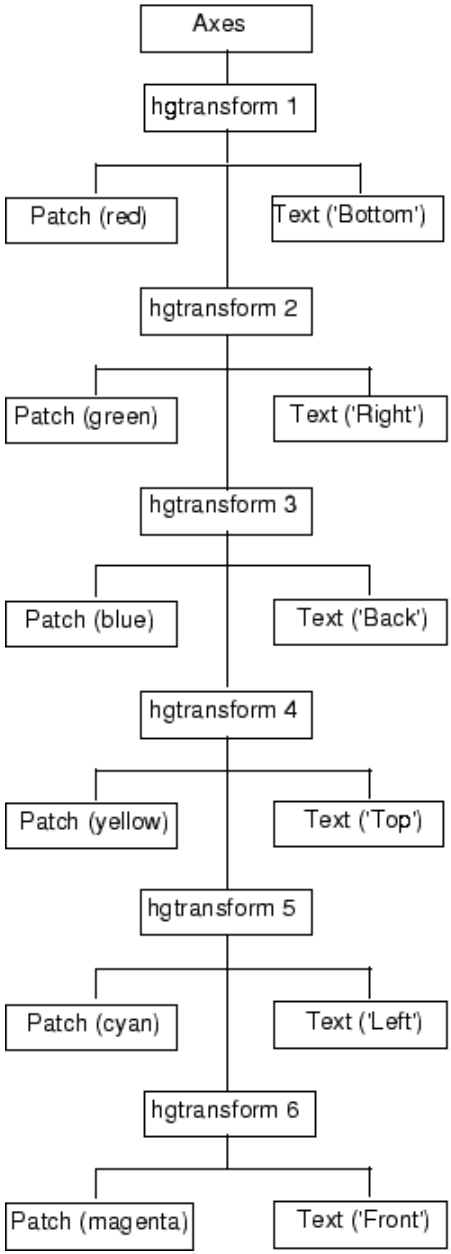
```
Tx1 = makehgtform('translate',[-20 0 0]);
Tx2 = makehgtform('translate',[20 0 0]);
set(t,'Matrix',Tx2*Ry*Tx1)
```


10
5
0
-5
-10
40
20
0
0
10
20
30
40
Effective axis of rotation

Example — Transforming a Hierarchy of Objects

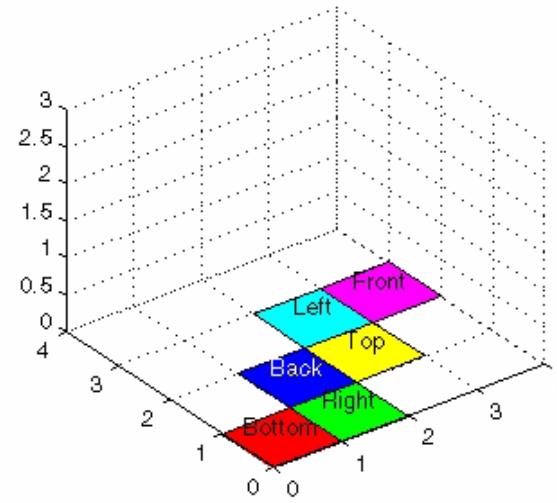
This example creates a hierarchy of hgtransform objects, which are then transformed in sequence to create a cube from six squares. The example illustrates how you can parent hgtransform objects to other hgtransform objects to create a hierarchy and how transforming members of a hierarchy affects subordinate members.

The following picture illustrates the hierarchy.



The diagram on the left represents the object hierarchy in the picture below.

Through a series of simple rotations and translations, the six squares are folded into a cube.



Note If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB editor.

1. Set the figure `Renderer` property to `zbuffer` so MATLAB uses double buffering to prevent flashing during the loop. Set up the figure and the view.

```
set(gcf,'Renderer','zbuffer');
% Set axis limits and view
set(gca,'XLim',[0 4], 'YLim',[0 4], 'ZLim', [0 3]);
view(3); axis equal; grid on
```

2. Define a hierarchy of hgtransform objects.

```
t(1) = hgtransform;
t(2) = hgtransform('parent',t(1));
t(3) = hgtransform('parent',t(2));
t(4) = hgtransform('parent',t(3));
t(5) = hgtransform('parent',t(4));
t(6) = hgtransform('parent',t(5));
```

3. Create the patch and text objects and parent each pair to the respective hgtransform object.

   Note that the data defining each patch object and the locations of all text objects are the same and are assigned by a single call to `set`. The objects are then translated to the desired positions on screen.

```
% Patch data
X = [0 0 1 1];
Y = [0 1 1 0];
Z = [0 0 0 0];
% Text data
Xtext = .5;
Ytext = .5;
Ztext = .15;
% Parent corresponding pairs of objects (patch and text)
% into the object hierarchy
p(1) = patch('FaceColor','red','Parent',t(1));
txt(1) = text('String','Bottom','Parent',t(1));
```

```
p(2) = patch('FaceColor','green','Parent',t(2));
txt(2) = text('String','Right','Parent',t(2));
p(3) = patch('FaceColor','blue','Parent',t(3));
txt(3) = text('String','Back','Color','white','Parent',t(3));
p(4) = patch('FaceColor','yellow','Parent',t(4));
txt(4) = text('String','Top','Parent',t(4));
p(5) = patch('FaceColor','cyan','Parent',t(5));
txt(5) = text('String','Left','Parent',t(5));
p(6) = patch('FaceColor','magenta','Parent',t(6));
txt(6) = text('String','Front','Parent',t(6));
% Set the patch x, y, and z data
set(p,'XData',X,'YData',Y,'ZData',Z)
% Set the position and alignment of the text
set(txt,'Position',[Xtext Ytext Ztext],...
    'HorizontalAlignment','center',...
    'VerticalAlignment','middle')
```

4 Translate the squares (patch objects) to the desired locations. Note that as hgtransform object 2 is translated, all its children (including hgtransform objects 3 through 6) are also translated. Therefore, each translation requires moving the square by only one unit in either the x or y direction. Hgtransform object 1 is left at its original position.

```
% Set up initial translation transform matrices
% Translate 1 unit in x
Tx = makehgtform('translate',[1 0 0]);
% Translate 1 unit in y
Ty = makehgtform('translate',[0 1 0]);
% Set the Matrix property of each hgtransform object (2-6)
set(t(2),'Matrix',Tx);
drawnow
set(t(3),'Matrix',Ty);
drawnow
set(t(4),'Matrix',Tx);
drawnow
set(t(5),'Matrix',Ty);
drawnow
set(t(6),'Matrix',Tx);
```

Object Properties

In this section...

Introduction

A graphics object's properties control many aspects of its appearance and behavior. Properties include general information such as the object's type, its parent and children, and whether it is visible, as well as information unique to the particular class of object.

For example, from any given figure object you can obtain the identity of the last key pressed in the window, the location of the pointer, or the handle of the most recently selected menu.

Note The simplest way to access the documentation of all object properties is using the Handle Graphics Property Browser.

Storing Object Information

MATLAB organizes graphics information into a hierarchy and stores information about objects in properties. For example, root properties contain the handle of the current figure and the current location of the pointer (cursor), figure properties maintain lists of their descendants and keep track of certain events that occur within the window, and axes properties contain information about how each child object uses the figure colormap and the color order used by the `plot` function.

Changing Values

You can query the current value of any property and specify most property values (although some are set by MATLAB and are read only). Property values apply uniquely to a particular instance of an object; setting a value for one object does not change this value for other objects of the same type.

Order Dependence of Setting Property Values

MATLAB sets the values of properties in the order in which properties are assigned values in a statement. For example, the following calls to the `figure` function create very different results. This statement,

```
figure('Position',[1 1 400 300],'Units','inches')
```

creates a figure in the lower left corner of the screen that is 400 pixels in width and 300 pixels in height. If you reverse the order of the `Position` and `Units` properties, MATLAB creates a figure that is too large to display (400 by 300 inches):

```
figure('Units','inches','Position',[1 1 400 300])
```

Properties Are Interpreted from Left to Right

In the first figure above, MATLAB creates a figure of the specified size using the default `Units` (pixels) and then sets the `Units` to inches. In the second case MATLAB sets the `Units` to inches and uses these units to interpret the specified figure `Position`. MATLAB interprets the property values from left to right:

```
set(gcf,'Units','pixels')
get(gcf,'Position')
ans =
  1.0e+004 *
    0.0097    2.7760    0.1924    0.1137
% Change the Units, set the Position,
% and change Units again in one statement
set(gcf,'Units','pixels','Position',[1 1 400 300],'Units','inches')
get(gcf,'Position')
ans =
         0         0    4.1667    3.1250
```

Default Values

You can set default values that affect all subsequently created objects. Whenever you do not define a value for a property, either as a default or when you create the object, MATLAB uses “factory-defined” values.

Note that plot objects do not allow you to set default values.

The reference entry for each object creation function provides a complete list of the properties associated with the graphics object.

Properties Common to All Objects

Some properties are common to all graphics objects, as illustrated in the following table.

Property	Description
`BeingDeleted`	Has a value of `on` when object’s `DeleteFcn` has been called
`BusyAction`	Controls the way MATLAB handles callback routine interruption defined for the particular object
`ButtonDownFcn`	Callback routine that executes when button press occurs
`Children`	Handles of all this object’s child objects
`Clipping`	Mode that enables or disables clipping (meaningful only for axes children)
`CreateFcn`	Callback routine that executes when this type of object is created
`DeleteFcn`	Callback routine that executes when you issue a command that destroys the object
`HandleVisibility`	Allows you to control the availability of the object’s handle from the command line and from within callback routines
`HitTest`	Determines if object can become the current object when selected by a mouse click
`Interruptible`	Determines whether a callback routine can be interrupted by a subsequently invoked callback routine
`Parent`	The object’s parent
`Selected`	Indicates whether object is selected

Property	Description
`SelectionHighlight`	Specifies whether object visually indicates the selection state
`Tag`	User-specified object label
`Type`	The type of object (figure, line, text, etc.)
`UserData`	Any data you want to associate with the object
`Visible`	Determines whether or not the object is visible

Properties Common to All Objects

Some properties are common to all graphics objects, as illustrated in the following table.

Property	Description
`BeingDeleted`	Has a value of `on` when object's `DeleteFcn` has been called
`BusyAction`	Controls the way MATLAB handles callback routine interruption defined for the particular object
`ButtonDownFcn`	Callback routine that executes when button press occurs
`Children`	Handles of all this object's child objects
`Clipping`	Mode that enables or disables clipping (meaningful only for axes children)
`CreateFcn`	Callback routine that executes when this type of object is created
`DeleteFcn`	Callback routine that executes when you issue a command that destroys the object
`HandleVisibility`	Allows you to control the availability of the object's handle from the command line and from within callback routines
`HitTest`	Determines if object can become the current object when selected by a mouse click
`Interruptible`	Determines whether a callback routine can be interrupted by a subsequently invoked callback routine
`Parent`	The object's parent
`Selected`	Indicates whether object is selected
`SelectionHighlight`	Specifies whether object visually indicates the selection state
`Tag`	User-specified object label
`Type`	The type of object (figure, line, text, etc.)
`UserData`	Any data you want to associate with the object
`Visible`	Determines whether or not the object is visible

Setting and Querying Property Values

In this section...

Using Set and Get

The `set` and `get` functions specify and retrieve the value of existing graphics object properties. They also enable you to list possible values for properties that have a fixed set of values. (You can also use the Property Editor to set many property values.

The basic syntax for setting the value of a property on an existing object is

```
set(object_handle,'PropertyName','NewPropertyValue')
```

To query the current value of a specific object's property, use a statement like

```
returned_value = get(object_handle,'PropertyName');
```

Property names are always quoted strings. Property values depend on the particular property.

See “Accessing Object Handles” on page 8-58 and the `findobj` command for information on finding the handles of existing objects.

Setting Property Values

You can change the properties of an existing object using the `set` function and the handle returned by the creating function. For example, this statement moves the *y*-axis to the right side of the plot on the current axes.

```
set(gca,'YAxisLocation','right')
```

If the handle argument is a vector, MATLAB sets the specified value on all identified objects.

You can specify property names and property values using structure arrays or cell arrays. This can be useful if you want to set the same properties on a number of objects. For example, you can define a structure to set axes properties appropriately to display a particular graph.

```
view1.CameraViewAngleMode = 'manual';
view1.DataAspectRatio = [1 1 1];
view1.ProjectionType = 'Perspective';
```

To set these values on the current axes, type

```
set(gca,view1)
```

Listing Possible Values

You can use `set` to display the possible values for many properties without actually assigning a new value. For example, this statement obtains the values you can specify for line object markers.

```
set(obj_handle,'Marker')
```

MATLAB returns a list of values for the `Marker` property for the type of object specified by `obj_handle`. Braces indicate the default value.

```
[ + | o | * | . | x | square | diamond | v | ^ | > | < | pentagram
| hexagram | {none} ]
```

To see a list of all settable properties along with possible values of properties that accept string values, use `set` with just an object handle.

```
set(object_handle)
```

For example, for a surface object, MATLAB returns

```
CData
CDataScaling: [ {on} | off]
EdgeColor: [ none | {flat} | interp ] ColorSpec.
EraseMode: [ {normal} | background | xor | none ]
FaceColor: [ none | {flat} | interp | texturemap ] ColorSpec.
LineStyle: [ {-} | -- | : | -. | none ]
  .
  .
```

```
  .
Visible: [ {on} | off ]
```

If you assign the output of the `set` function to a variable, MATLAB returns the output as a structure array. For example,

```
a = set(gca);
```

The field names in `a` are the object's property names and the field values are the possible values for the associated property. For example,

```
a.GridLineStyle
ans =

    '-'
    '--'
    ':'
    '-.'
    'none'
```

returns the possible values for the axes grid line styles. Note that while property names are not case sensitive, MATLAB structure field names are. For example,

```
a.gridlinestyle
??? Reference to non-existent field 'gridlinestyle'.
```

returns an error.

Querying Property Values

Use `get` to query the current value of a property or of all the object's properties. For example, check the value of the current axes `PlotBoxAspectRatio` property.

```
get(gca,'PlotBoxAspectRatio')
ans =
     1     1     1
```

MATLAB lists the values of all properties, where practical. However, for properties containing data, MATLAB lists the dimensions only (for example, `CurrentPoint` and `ColorOrder`).

```
AmbientLightColor = [1 1 1]
Box = off
CameraPosition = [0.5 0.5 2.23205]
CameraPositionMode = auto
CameraTarget = [0.5 0.5 0.5]
CameraTargetMode = auto
CameraUpVector = [0 1 0]
CameraUpVectorMode = auto
CameraViewAngle = [32.2042]
CameraViewAngleMode = auto
CLim: [0 1]
CLimMode: auto
Color: [0 0 0]
CurrentPoint: [ 2x3  double]
ColorOrder: [ 7x3  double]
  .
  .
  .
Visible = on
```

Querying Individual Properties

You can obtain the data from the property by getting that property individually.

```
get(gca,'ColorOrder')
ans =
         0         0    1.0000
         0    0.5000         0
    1.0000         0         0
         0    0.7500    0.7500
    0.7500         0    0.7500
    0.7500    0.7500         0
    0.2500    0.2500    0.2500
```

Returning a Structure

If you assign the output of `get` to a variable, MATLAB creates a structure array whose field names are the object property names and whose field values are the current values of the named property.

For example, if you plot some data, `x` and `y`,

```
h = plot(x,y);
```

and get the properties of the line object created by `plot`,

```
a = get(h);
```

you can access the values of the line properties using the field name. This call to the `text` command places the string 'x and y data' at the first data point and colors the text to match the line color.

```
text(x(1),y(1),'x and y data','Color',a.Color)
```

If `x` and `y` are matrices, `plot` draws one line per column. To label the plot of the second column of data, reference that line.

```
text(x(1,2),y(1,2),'Second set of data','Color',a(2).Color)
```

Querying Groups of Properties

You can define a cell array of property names and conveniently use it to obtain the values for those properties. For example, suppose you want to query the values of the axes "camera mode" properties. First define the cell array.

```
camera_props(1) = {'CameraPositionMode'};
camera_props(2) = {'CameraTargetMode'};
camera_props(3) = {'CameraUpVectorMode'};
camera_props(4) = {'CameraViewAngleMode'};
```

Use this cell array as an argument to obtain the current values of these properties.

```
get(gca,camera_props)
ans =
     'auto' 'auto' 'auto' 'auto'
```

Factory-Defined Property Values

MATLAB defines values for all properties, which are used if you do not specify values as arguments or as defaults. You can obtain a list of all factory-defined values with the statement

```
a = get(0,'Factory');
```

`get` returns a structure array whose field names are the object type and property name concatenated, and field values are the factory value for the indicated object and property. For example, this field,

```
UimenuSelectionHighlight: 'on'
```

indicates that the factory value for the `SelectionHighlight` property on uimenu objects is `on`.

You can get the factory value of an individual property with

```
get(0,'FactoryObjectTypePropertyName')
```

For example,

```
get(0,'FactoryTextFontName')
```

Setting Default Property Values

In this section...

Factory- and User-Defined Values

All object properties have values built into MATLAB (i.e., factory-defined values). You can also define your own default values at any point in the object hierarchy.

Note that you cannot define default values for plot objects.

How MATLAB Searches for Default Values

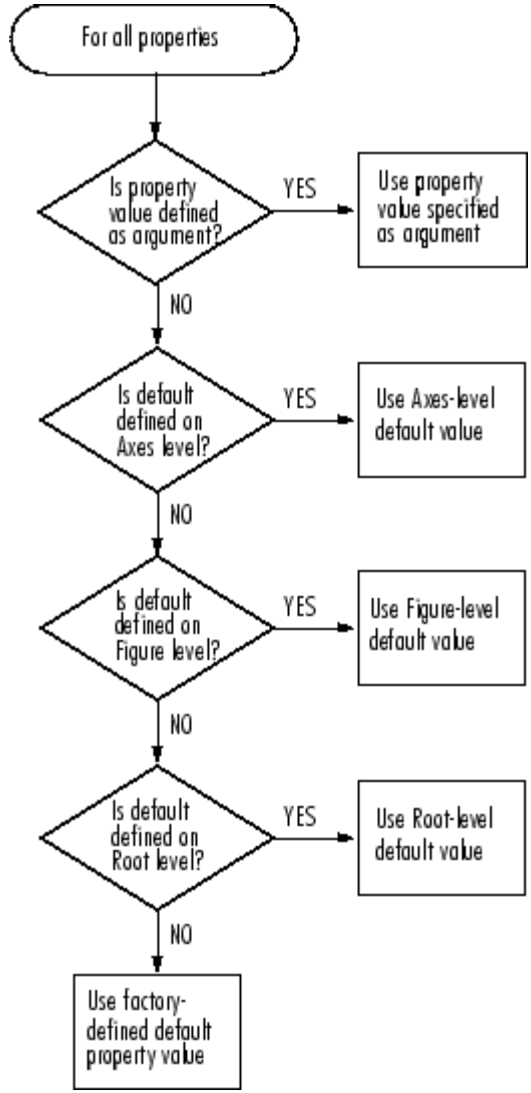
MATLAB searches for a default value beginning with the current object and continuing through the object's ancestors until it finds a user-defined default value or until it reaches the factory-defined value. Therefore, a search for property values is always satisfied.

The closer to the root of the hierarchy you define the default, the broader its scope. If you specify a default value for line objects on the root level, MATLAB uses that value for all lines (because the root is at the top of the hierarchy). If you specify a default value for line objects on the axes level, then MATLAB uses that value for line objects drawn only in that axes.

If you define default values on more than one level, the value defined on the closest ancestor takes precedence because MATLAB terminates the search as soon as it finds a value.

Note that setting default values affects only those objects created after you set the default. Existing graphics objects are not affected.

This diagram shows the steps MATLAB follows in determining the value of a graphics object property.



Defining Default Values

To specify default values, create a string beginning with the word `Default` followed by the object type and finally the object property. For example, to specify a default value of 1.5 points for the line property `LineWidth` at the level of the current figure, use the statement

```
set(gcf,'DefaultLineLineWidth',1.5)
```

The string `DefaultLineLineWidth` identifies the property as a line property. To specify the figure color, use `DefaultFigureColor`. Note that it is meaningful to specify a default figure color only on the root level.

```
set(0,'DefaultFigureColor','b')
```

Use `get` to determine what default values are currently set on any given object level; for example,

```
get(gcf,'default')
```

returns all default values set on the current figure.

Setting Properties to the Default

Specifying a property value of `'default'` sets the property to the first encountered default value defined for that property. For example, these statements result in a green surface `EdgeColor`:

```
set(0,'DefaultSurfaceEdgeColor','k')
h = surface(peaks);
set(gcf,'DefaultSurfaceEdgeColor','g')
set(h,'EdgeColor','default')
```

Because a default value for surface `EdgeColor` exists on the figure level, MATLAB encounters this value first and uses it instead of the default `EdgeColor` defined on the root.

Removing Default Values

Specifying a property value of `'remove'` gets rid of user-defined default values. The statement

```
set(0,'DefaultSurfaceEdgeColor','remove')
```

removes the definition of the default Surface `EdgeColor` from the root.

Setting Properties to Factory-Defined Values

Specifying a property value of `'factory'` sets the property to its factory-defined value. (Theproperty descriptions provides access to the factory settings for properties having predefined sets of values.)

For example, these statements set the `EdgeColor` of surface `h` to black (its factory setting) regardless of what default values you have defined.

```
set(gcf,'DefaultSurfaceEdgeColor','g')
h = surface(peaks);
set(h,'EdgeColor','factory')
```

Reserved Words

Setting a property value to `default`, `remove`, or `factory` produces the effects described in the previous sections. To set a property to one of these words (e.g., a text or uicontrol `String` property set to the word `Default`), you must precede the word with the backslash character. For example,

```
h = uicontrol('Style','edit','String','\Default');
```

Examples — Setting Default Line Styles

The `plot` function cycles through the colors defined by the axes `ColorOrder` property when displaying multiline plots. If you define more than one value for the axes `LineStyleOrder` property, MATLAB increments the line style after each cycle through the colors.

You can set default property values that cause the `plot` function to produce graphs using varying linestyles, but not varying colors. This is useful when you are working on a monochrome display or printing on a black and white printer.

First Example

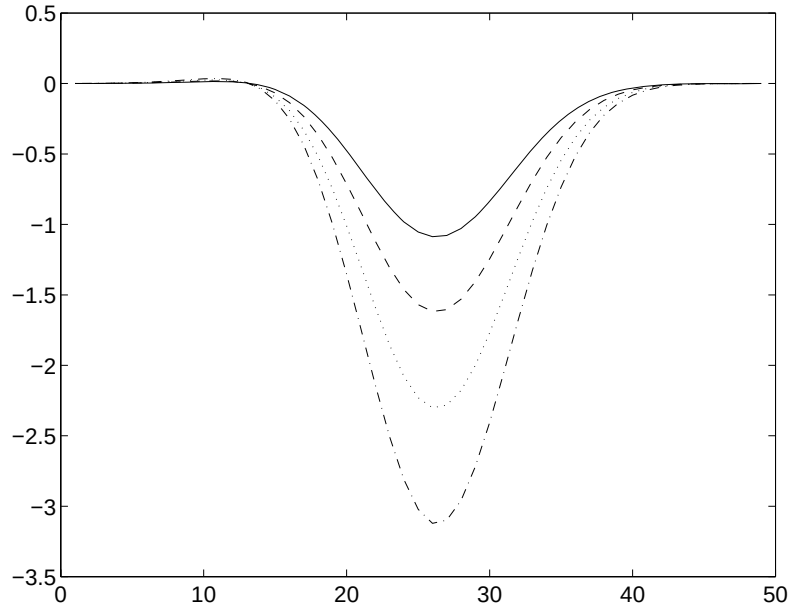
This example creates a figure with a white plot (axes) background color, then sets default values for axes objects on the root level.

```
whitebg('w') %create a figure with a white color scheme
set(0,'DefaultAxesColorOrder',[0 0 0],...
      'DefaultAxesLineStyleOrder','-|--|:|-.')
```

Whenever you call `plot`,

```
Z = peaks; plot(1:49,Z(4:7,:))
```

it uses one color for all data plotted because the axes `ColorOrder` contains only one color, but cycles through the linestyles defined for `LineStyleOrder`.



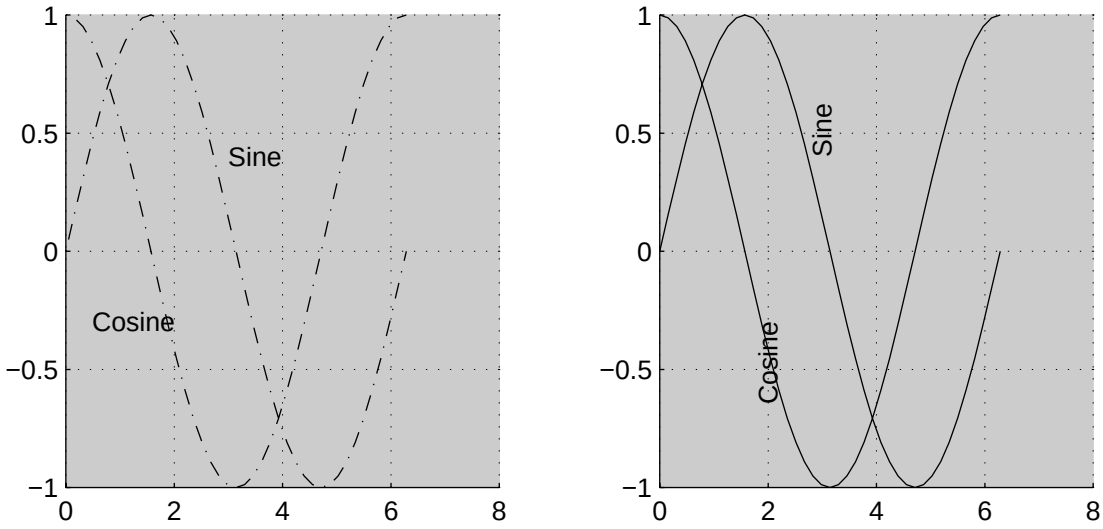
Second Example

This example sets default values on more than one level in the hierarchy. These statements create two axes in one figure window, setting default values on the figure level and the axes level.

```
t = 0:pi/20:2*pi;
s = sin(t);
c = cos(t);
% Set default value for axes Color property
```

```
figh = figure('Position',[30 100 800 350],...
              'DefaultAxesColor',[.8 .8 .8]);

axh1 = subplot(1,2,1); grid on
% Set default value for line LineStyle property in first axes
set(axh1,'DefaultLineLineStyle','-.')
line('XData',t,'YData',s)
line('XData',t,'YData',c)
text('Position',[3 .4],'String','Sine')
text('Position',[2 -.3],'String','Cosine',...
     'HorizontalAlignment','right')

axh2 = subplot(1,2,2); grid on
% Set default value for text Rotation property in second axes
set(axh2,'DefaultTextRotation',90)
line('XData',t,'YData',s)
line('XData',t,'YData',c)
text('Position',[3 .4],'String','Sine')
text('Position',[2 -.3],'String','Cosine',...
     'HorizontalAlignment','right')
```

Issuing the same `line` and `text` statements to each subplot region results in a different display, reflecting different default settings.



Because the default axes `Color` property is set on the figure level of the hierarchy, MATLAB creates both axes with the specified gray background color.

The axes on the left (subplot region 121) defines a dash-dot line style (`-.`) as the default, so each call to the `line` function uses dash-dot lines. The axes on the right does not define a default line style, so MATLAB uses solid lines (the factory setting for lines).

The axes on the right defines a default text `Rotation` of 90 degrees, which rotates all text by this amount. MATLAB obtains all other property values from their factory settings, which results in nonrotated text on the left.

To install default values whenever you run MATLAB, specify them in your `startup.m` file. Note that MATLAB might install default values for some appearance properties when started by calling the `colordef` command.

Accessing Object Handles

In this section...

Introduction

MATLAB assigns a handle to every graphics object it creates. All object creation functions optionally return the handle of the created object. If you want to access the object's properties (e.g., from an M-file) you should assign its handle to a variable at creation time to avoid searching for it later.

You can always obtain the handle of an existing object with the `findobj` function or by listing its parent's `Children` property.

See "Searching for Objects by Property Values — findobj" on page 8-60 for examples.

See "Protecting Figures and Axes" on page 8-77 for more information on how object handles are hidden from normal access.

Special Object Handles

The root object's handle is always zero. The handle of a figure is either

- An integer that, by default, is displayed in the window title bar
- A floating point number requiring full MATLAB internal precision

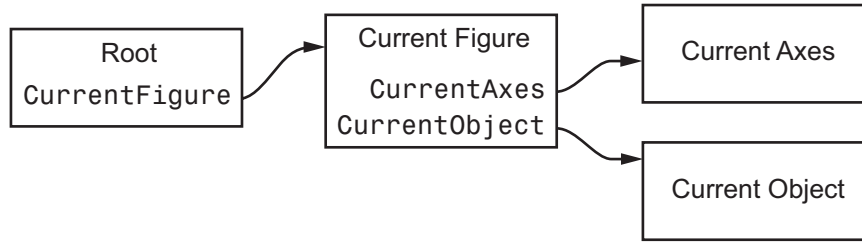
The figure `IntegerHandle` property controls the type of handle the figure receives.

All other graphics object handles are floating-point numbers. You must maintain the full precision of these numbers when you reference handles. Rather than attempting to read handles off the screen and retype them, you must store the value in a variable and pass that variable whenever MATLAB requires a handle.

The Current Figure, Axes, and Object

An important concept in Handle Graphics is that of being current. The current figure is the window designated to receive graphics output. Likewise, the current axes is the target for commands that create axes children. The current object is the last graphics object created or clicked on by the mouse.

MATLAB stores the three handles corresponding to these objects in the ancestor's property list.



These properties enable you to obtain the handles of these key objects.

```
get(0,'CurrentFigure');
get(gcf,'CurrentAxes');
get(gcf,'CurrentObject');
```

The following commands are shorthand notation for the `get` statements.

- `gcf` — Returns the value of the root `CurrentFigure` property
- `gca` — Returns the value of the current figure's `CurrentAxes` property
- `gco` — Returns the value of the current figure's `CurrentObject` property

You can use these commands as input arguments to functions that require object handles. For example, you can click on a line object and then use `gco` to specify the handle to the `set` command,

```
set(gco,'Marker','square')
```

or list the values of all current axes properties with

```
get(gca)
```

You can get the handles of all the graphic objects in the current axes (except those with hidden handles),

```
h = get(gca,'Children');
```

and then determine the types of the objects.

```
get(h,'type')
ans =
      'text'
      'patch'
      'surface'
      'line'
```

While `gcf` and `gca` provide a simple means of obtaining the current figure and axes handles, they are less useful in M-files. This is particularly true if your M-file is part of an application layered on MATLAB where you do not necessarily have knowledge of user actions that can change these values.

See “Controlling Graphics Output” on page 8-69 for information on how to prevent users from accessing the handles of graphics objects that you want to protect.

Searching for Objects by Property Values — findobj

The `findobj` function provides a means to traverse the object hierarchy quickly and obtain the handles of objects having specific property values. To serve as a means of identification, all graphics objects have a `Tag` property that you can set to any string. You can then search for the specific property/value pair.

For example, suppose you create a checkbox that is sometimes inactivated in the GUI. By assigning a unique value for the `Tag` property, you can always find that particular instance and set its properties.

```
uicontrol('Style','checkbox','Tag','save option')
```

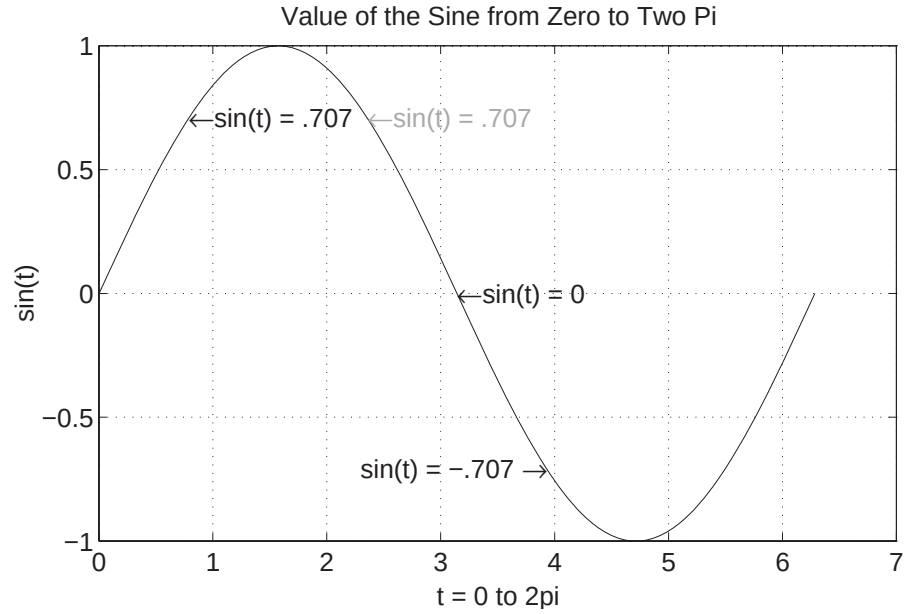
Use `findobj` to locate the object whose `Tag` property is set to `'save option'` and disable it.

```
set(findobj('Tag','save option'),'Enable','off')
```

If you do not specify a starting object, `findobj` searches from the root object, finding all occurrences of the property name/property value combination that you specify.

Example — Finding Objects

This plot of the sine function contains text objects labeling particular values of the function.



Suppose you want to move the text string labeling the value sin(t) = .707 from its current location at `[pi/4,sin(pi/4)]` to the point `[3*pi/4,sin(3*pi/4)]` where the function has the same value (shown grayed out in the picture). To do this, you need to determine the handle of the text object labeling that point and change its `Position` property.

To use `findobj`, pick a property value that uniquely identifies the object. This example uses the text `String` property.

```
text_handle = findobj('String','\leftarrowsin(t) = .707');
```

Next move the object to the new position, defining the text `Position` in axes units.

```
set(text_handle,'Position',[3*pi/4,sin(3*pi/4),0])
```

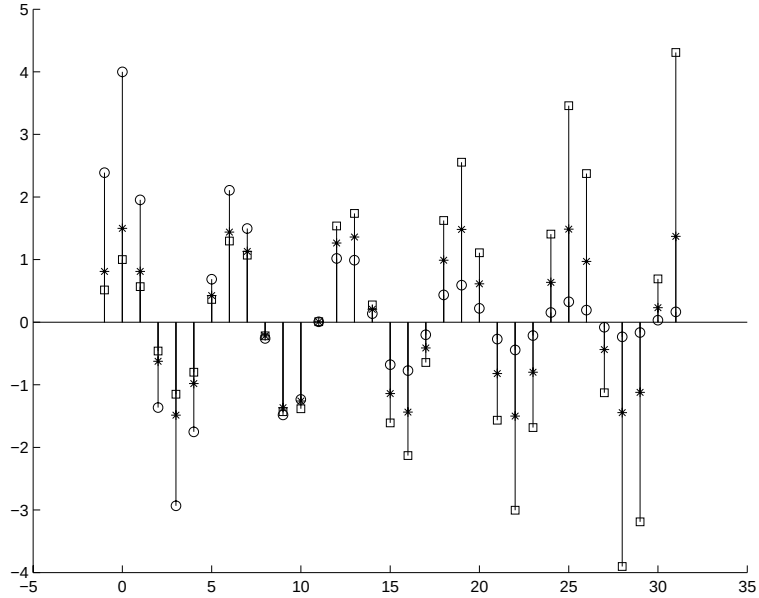
`findobj` also lets you restrict the search by specifying a starting point in the hierarchy, instead of beginning with the root object. This results in faster searches if there are many objects in the hierarchy. In the previous example, you know the text object of interest is in the current axes, so you can type

```
text_handle = findobj(gca,'String','\leftarrowsin(t) = .707');
```

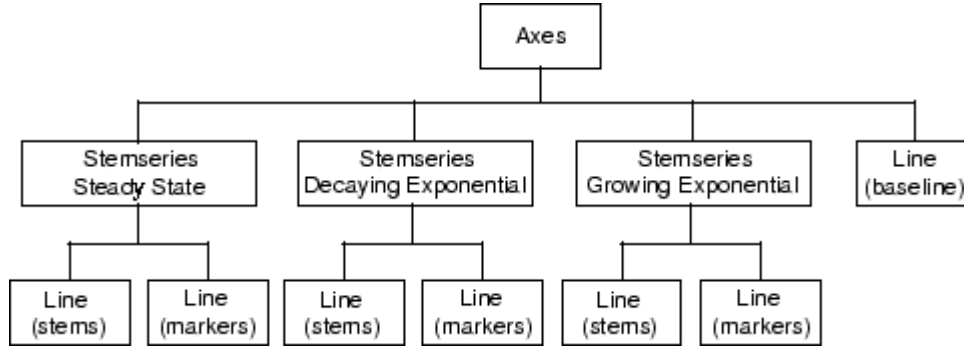
Example — Using Logical Operators and Regular Expression

Suppose you create the following graph and want to modify certain properties of the objects created.

```
x = 0:30;
y = [1.5*cos(x);4*exp(-.1*x).*cos(x);exp(.05*x).*cos(x)]';
h = stem(x,y);
set(h(1),'Color','black',...
            'Marker','o',...
            'Tag','Decaying Exponential')
set(h(2),'Color','black',...
            'Marker','square',...
            'Tag','Growing Exponential')
set(h(3),'Color','black',...
            'Marker','*',...
            'Tag','Steady State')
```



The following instance diagram shows the graphics objects created in the graph. Each of the three sets of data produces a stemseries object, which in turn uses two lines to create the stem graph; one line for the stems and one for the markers that terminate each stem. There is also a line used for the baseline.



Controlling the Depth of the Search. Now make the baseline into a dashed line. Because it is parented directly to the axes, you can use the following statement to access only this line:

```
set(findobj(gca,'-depth',1,'Type','line'),'LineStyle','--')
```

By setting `-depth` to `1`, `findobj` searches only the axes and its immediate children. As you can see from the above instance diagram, the baseline is the only line object parented directly to the axes.

Limiting the Search with Regular Expressions. Increase the value of the `MarkerSize` property by 2 points on all stemseries objects that do not have their property `Tag` set to `'Steady State'`.

```
h = findobj('-regexp','Tag','^(?!Steady State$).');
set(h,{'MarkerSize'},num2cell(cell2mat(get(h,'MarkerSize'))+2))
```

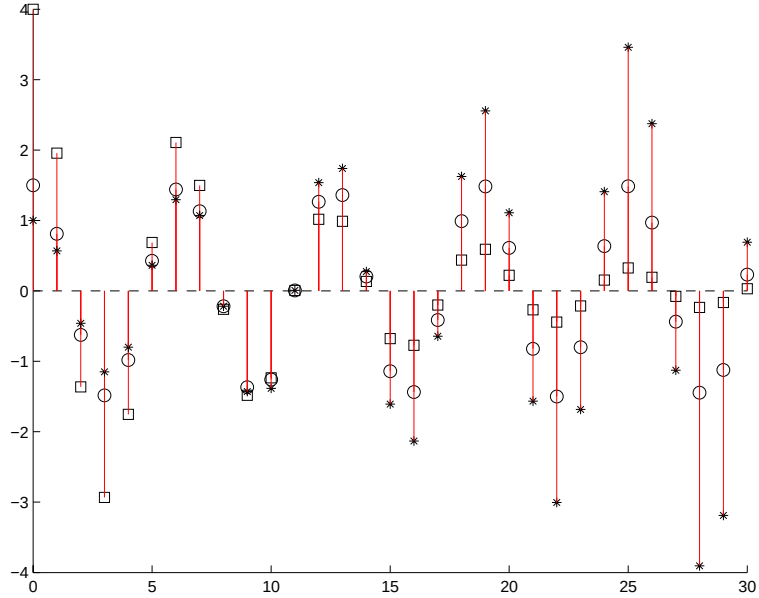
See the `regexp` function for more information on using regular expressions in MATLAB.

Using Logical Operators. Change the color of the stem lines, but not the stem markers. To do this, you must access the line objects contained by the three stemseries objects. You cannot just set the stemseries `Color` property because it sets both the line and marker colors.

Search for objects that are of `Type` `line`, have `Marker` set to `none`, and do not have `LineStyle` set to `'--'`, which is the baseline.

```
h = findobj('type','line','Marker','none',...
    '-and','-not','LineStyle','--');
set(h,'Color','red')
```

The following picture shows the graph after making the various changes described in this section.



Copying Objects

You can copy objects from one parent to another using the `copyobj` function. The new object differs from the original object only in the value of its `Parent` property and its handle; it is otherwise a clone of the original. You can copy a number of objects to a new parent, or one object to a number of new parents, as long as the result maintains the correct parent/child relationship.

When you copy an object having child objects, MATLAB copies all children as well.

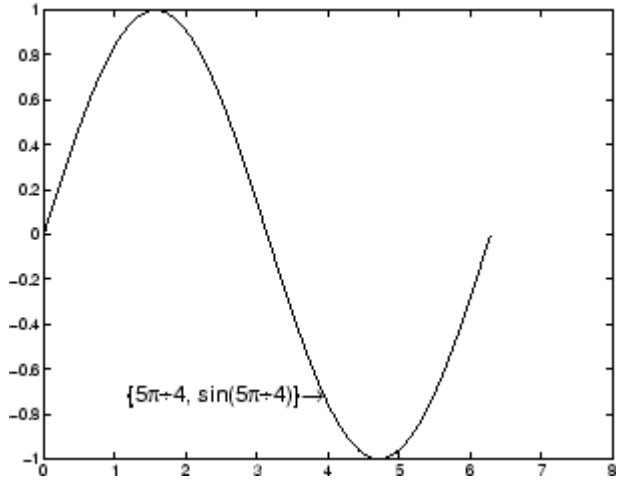
Example — Copying Objects

Suppose you are plotting a variety of data and want to label the point having the *x*- and *y*-coordinates determined by $5\pi \div 4$, $\sin(5\pi \div 4)$ in each plot. The `text` function allows you to specify the location of the label in the coordinates defined by the *x*- and *y*-axis limits, simplifying the process of locating the text.

```
text('String','\{5\pi\div4, sin(5\pi\div4)\}\rightarrow',...
     'Position',[5*pi/4,sin(5*pi/4),0],...
     'HorizontalAlignment','right')
```

In this statement, the `text` function

- Labels the data point with the string $\{5\pi \div 4, \sin(5\pi \div 4)\}$ using TeX commands to draw a right-facing arrow and mathematical symbols
- Specifies the `Position` in terms of the data being plotted
- Places the data point to the right of the text string by changing the `HorizontalAlignment` to `right` (the default is `left`)

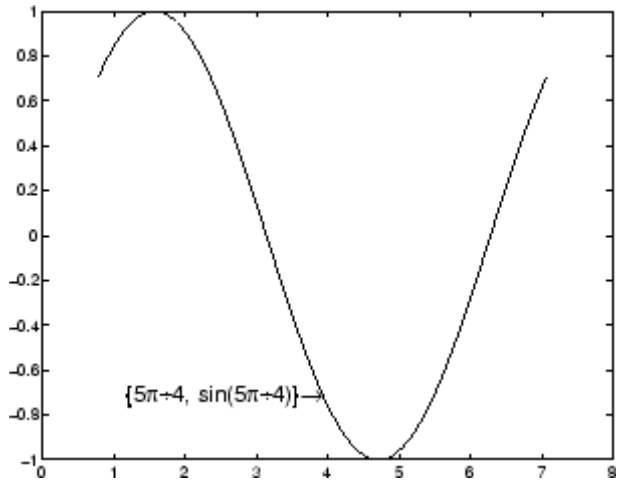


To label the same point with the same string in another plot, copy the text using `copyobj`. Because the last statement did not save the handle to the text object, you can find it using `findobj` and the `'String'` property.

```
text_handle = findobj('String',...
              '\{5\pi\div4,sin(5\pi\div4)\}\rightarrow');
```

After creating the next plot, add the label by copying it from the first plot.

```
copyobj(text_handle,gca).
```



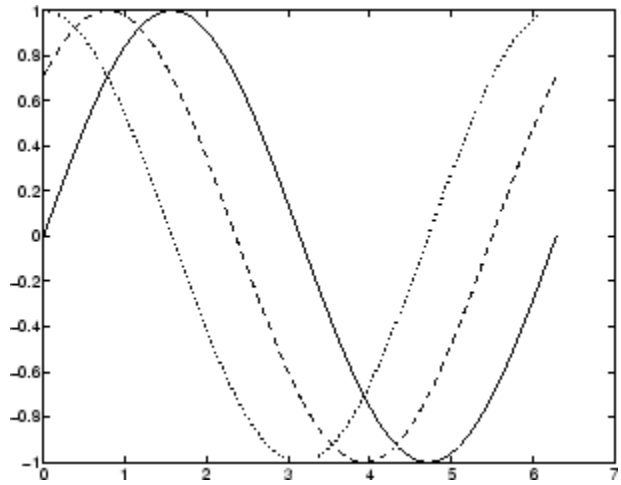
This particular example takes advantage of the fact that text objects define their location in the axes data space. Therefore the text `Position` property did not need to change from one plot to another.

Deleting Objects

You can remove a graphics object with the `delete` command, using the object's handle as an argument. For example, you can delete the current axes (and all of its descendants) with the statement

```
delete(gca)
```

You can use `findobj` to get the handle of a particular object you want to delete. For example, to find the handle of the dotted line in this multiline plot,



use `findobj` to locate the object whose `LineStyle` property is `':'`

```
line_handle = findobj('LineStyle',':');
```

then use this handle with the `delete` command.

```
delete(line_handle)
```

You can combine these two statements, substituting the `findobj` statement for the handle.

```
delete(findobj('LineStyle',':'))
```

Controlling Graphics Output

In this section...

Figure Targets

MATLAB allows many figure windows to be open simultaneously during a session. A MATLAB application might create figures to display graphical user interfaces as well as plotted data. It is necessary then to protect some figures from becoming the target for graphics display and to prepare (e.g., reset properties and clear existing objects from) others before receiving new graphics.

Specifying the Target for Graphics Output

By default, MATLAB functions that create graphics objects display them in the current figure and current axes (if an axes child). You can direct the output to another parent by explicitly specifying the `Parent` property with the creating function. For example,

```
plot(1:10,'Parent',axes_handle)
```

where *axes_handle* is the handle of the target axes. The `uicontrol` and `uimenu` functions have a convenient syntax that enables you to specify the parent as the first argument,

```
uicontrol(figure_handle,...)
uimenu(parent_menu_handle,...)
```

or you can set the Parent property. Many plotting functions accept an axes handle as the first argument as well.

Making a Figure and Axes Current

You can specify which figure and which axes within the figure are the target for graphics output. There are two ways to do this, each with different associated behavior.

Making Current and Update. If figure_handle is the handle to an existing figure, then the following statement,

```
figure(figure_handle)
```

- Makes figure_handle the current figure
- Restacks figure_handle to be the front-most figure displayed
- Makes figure_handle visible if it was not
- Refreshes figure_handle and process all pending window events

The same behavior applies to axes. Therefore, the following statement,

```
axes(axes_handle)
```

- Makes axes_handle the current axes
- Restacks axes_handle to be the front-most axes displayed
- Makes axes_handle visible if it was not
- Refreshes the figure containing the axes and process all pending window events for that figure

Make Current Without Changing State. You can make a figure or axes current without causing MATLAB to change the object's state by setting the figure's root object CurrentFigure property or the figure object's CurrentAxes property to the handle of the figure or axes you want to accept graphics output.

If figure_handle is the handle to an existing figure, then the following statement,

```
set(0,'CurrentFigure',figure_handle)
```

makes `figure_handle` the current figure without changes its state. Similarly, if axes_handle is the handle of an axes object, then the following statement

```
set(h,'CurrentAxes',axes_handle)
```

makes it the current axes assuming `h` is the handle of the figure that contains it.

Preparing Figures and Axes for Graphics

By default, commands that generate graphics output display the graphics objects in the current figure without clearing or resetting figure properties. However, if the graphics objects are axes children, MATLAB clears the axes and resets most axes properties to their default values before displaying the objects.

You can change this behavior by setting the figure and axes `NextPlot` properties.

Using NextPlot to Control Output Target

MATLAB high-level graphics functions check the values of the `NextPlot` properties to determine whether to add, clear, or clear and reset the figure and axes before drawing. Low-level object creation functions do not check the `NextPlot` properties. They simply add the new graphics objects to the current figure and axes.

Low-level functions are designed primarily for use in M-files where you can implement whatever drawing behavior you want. However, when you develop a MATLAB-based application, controlling MATLAB drawing behavior is essential to creating a program that behaves predictably.

This table summarizes the possible values for the `NextPlot` property.

NextPlot	Figure	Axes
`new`	Create a new figure and use it as the current figure.	Not an option for axes.

NextPlot	Figure	Axes
`add`	Add new graphics objects without clearing or resetting the current figure. (Default setting)	Add new graphics objects without clearing or resetting the current axes.
`replacechildren`	Remove all child objects, but do not reset figure properties. Equivalent to `clf`.	Remove all child objects, but do not reset axes properties. Equivalent to `cla`.
`replace`	Remove all child objects and reset figure properties to their defaults. Equivalent to `clf reset`.	Remove all child objects and reset axes properties to their defaults. Equivalent to `cla reset`. (Default setting)

Note that a `reset` returns all properties, except `Position` and `Units`, to their default values.

The `hold` command provides convenient access to the `NextPlot` properties. The statement

```
hold on
```

sets both figure and axes `NextPlot` properties to `add`.

The statement

```
hold off
```

sets the axes `NextPlot` property to `replace`.

Targeting Graphics Output with newplot

MATLAB provides the `newplot` function to simplify the process of writing graphics M-files that conform to the settings of the `NextPlot` properties.

`newplot` checks the values of the `NextPlot` properties and takes the appropriate action based on these values. You should place `newplot` at the beginning of any M-file that calls object creation functions.

When your M-file calls `newplot`, the following possible actions occur:

1. `newplot` checks the current figure's `NextPlot` property:
   - If there are no figures in existence, `newplot` creates one and makes it the current figure.
   - If the value of `NextPlot` is `add`, `newplot` makes the figure the current figure.
   - If the value of `NextPlot` is `new`, `newplot` creates a new figure and makes it the current figure
   - If the value of `NextPlot` is `replacechildren`, `newplot` deletes the figure's children (axes objects and their descendants) and makes this figure the current figure.
   - If the value of `NextPlot` is `replace`, `newplot` deletes the figure's children, resets the figure's properties to the defaults, and makes this figure the current figure.
2. `newplot` checks the current axes' `NextPlot` property:
   - If there are no axes in existence, `newplot` creates one and makes it the current axes.
   - If the value of `NextPlot` is `add`, `newplot` makes the axes the current axes.
   - If the value of `NextPlot` is `replacechildren`, `newplot` deletes the axes' children and makes this axes the current axes.
   - If the value of `NextPlot` is `replace`, `newplot` deletes the axes' children, resets the axes' properties to the defaults, and makes this axes the current axes.

MATLAB Default Behavior

Consider the default situation where the figure `NextPlot` property is `add` and the axes `NextPlot` property is `replace`. When you call `newplot`, it

1. Checks the value of the current figure's `NextPlot` property (which is `add`) and determines MATLAB can draw into the current figure with no further action. If there is no current figure, `newplot` creates one, but does not recheck its `NextPlot` property.

2. Checks the value of the current axes' `NextPlot` property (which is `replace`), deletes all graphics objects from the axes, resets all axes properties (except `Position` and `Units`) to their defaults, and returns the handle of the current axes.

Example — Using newplot

To illustrate the use of `newplot`, this example creates a function that is similar to the built-in `plot` function, except it automatically cycles through different line styles instead of using different colors for multiline plots.

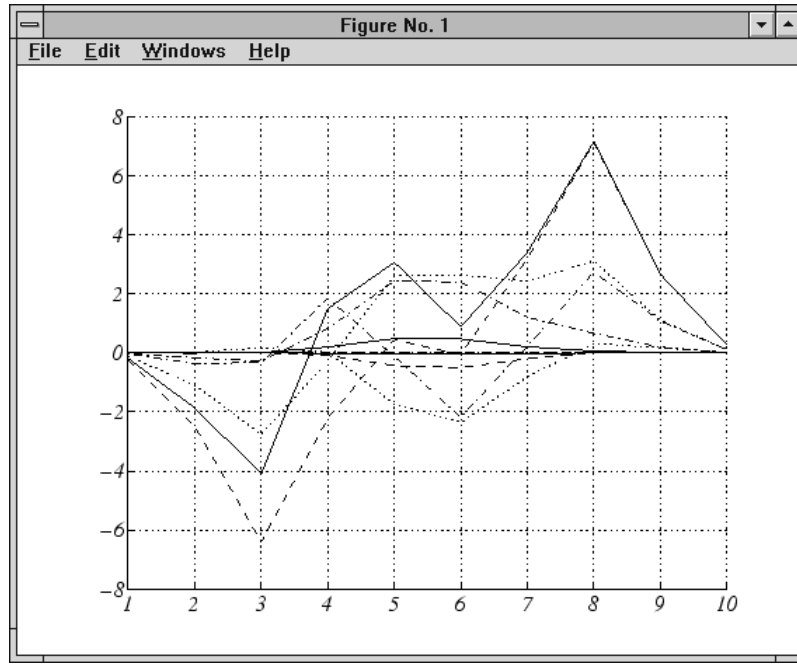
```
function my_plot(x,y)
cax = newplot;   % newplot returns handle of current axes
LSO = ['- ';'--';': ';'-.'];
set(cax,'FontName','Times','FontAngle','italic')
set(get(cax,'Parent'),'MenuBar','none') %
line_handles = line(x,y,'Color','b');
style = 1;
for i = 1:length(line_handles)
    if style > length(LSO), style = 1;end
    set(line_handles(i),'LineStyle',LSO(style,:))
    style = style + 1;
end
grid on
```

The function `my_plot` uses the high-level `line` function syntax to plot the data. This provides the same flexibility in input argument dimension that the built-in `plot` function supports. The `line` function does not check the value of the figure or axes `NextPlot` property. However, because `my_plot` calls `newplot`, it behaves the same way the high-level `plot` function does — with default values in place, `my_plot` clears and resets the axes each time you call it.

`my_plot` uses the handle returned by `newplot` to access the target figure and axes. This example sets axes font properties and disables the figure's menu bar. Note how the figure handle is obtained via the axes `Parent` property.

This picture shows typical output for the `my_plot` function.

```
my_plot(1:10,peaks(10))
```



Basic Plotting M-File Structure

This example illustrates the basic structure of graphics M-files:

- Call `newplot` early to conform to the `NextPlot` properties and to obtain the handle of the target axes.
- Reference the axes handle returned by `newplot` to set any axes properties or to obtain the figure's handle.
- Call object creation functions to draw graphics objects with the desired characteristics.

The MATLAB default settings for the NextPlot properties facilitate writing M-files that adhere to the standard behavior: reuse the figure window, but clear and reset the axes with each new graph. Other values for these properties allow you to implement different behaviors.

Replacing Only the Child Objects — replacechildren

The replacechildren value for NextPlot causes newplot to remove child objects from the figure or axes, but does not reset any property values (except the list of handles contained in the Children property).

This can be useful after setting properties you want to use for subsequent graphs without having to reset properties. For example, if you type on the command line

```
set(gca,'ColorOrder',[0 0 1],'LineStyleOrder','-|--|:|-.',...
        'NextPlot','replacechildren')
plot(x,y)
```

plot produces the same output as the M-file my_plot in the previous section, but only within the current axes. Calling plot still erases the existing graph (i.e., deletes the axes children), but it does not reset axes properties. The values specified for the ColorOrder and LineStyleOrder properties remain in effect.

Testing for Hold State

There are situations in which your M-file should change the visual appearance of the axes to accommodate new graphics objects. For example, if you want the M-file my_plot from the previous example to accept 3-D data, it makes sense to set the view to 3-D when the input data has *z*-coordinates.

However, to be consistent with the behavior of the MATLAB high-level routines, it is good practice to test whether hold is on before changing parent axes or figure properties. When hold is on, the axes and figure NextPlot properties are both set to add.

The M-file my_plot3 accepts 3-D data and also checks the hold state, using ishold, to determine whether it should change the view.

```
function my_plot3(x,y,z)
```

```
cax = newplot;
hold_state = ishold; % ishold tests the current hold state
LSO = ['- ';'--';': ';'-.'];
if nargin == 2
    hlines = line(x,y,'Color','k');
    if ~hold_state % Change view only if hold is off
        view(2)
    end
elseif nargin == 3
    hlines = line(x,y,z,'Color','k');
    if ~hold_state % Change view only if hold is off
        view(3)
    end
end
ls = 1;
for hindex = 1:length(hlines)
    if ls > length(LSO),ls = 1;end
    set(hlines(hindex),'LineStyle',LSO(ls,:))
    ls = ls + 1;
end
```

If `hold` is `on` when you call `my_plot3`, it does not change the view. If `hold` is `off`, `my_plot3` sets the view to 2-D or 3-D, depending on whether there are two or three input arguments.

Protecting Figures and Axes

There are situations in which it is important to prevent particular figures or axes from becoming the target for graphics output (i.e., preventing them from becoming the `gcf` or `gca`). An example is a figure containing the uicontrols that implement a user interface.

You can prevent MATLAB from drawing into a particular figure or axes by removing its handle from the list of handles that are visible to the `newplot` function, as well as any other functions that either return or implicitly reference handles (i.e., `gca`, `gcf`, `gco`, `cla`, `clf`, `close`, and `findobj`). Two properties control handle hiding: `HandleVisibility` and `ShowHiddenHandles`.

HandleVisibility Property

`HandleVisibility` is a property of all objects. It controls the scope of handle visibility within three different ranges. Property values can be

- `on` — The object's handle is available to any function executed on the MATLAB command line or from an M-file. This is the default setting.
- `callback` — The object's handle is hidden from all functions executing on the command line, even if it is on the top of the screen stacking order. However, during callback routine execution (MATLAB statements or functions that execute in response to user action), the handle is visible to all functions, such as `gca`, `gcf`, `gco`, `findobj`, and `newplot`. This setting enables callback routines to take advantage of the MATLAB handle access functions, while ensuring that users typing at the command line do not inadvertently disturb a protected object.
- `off` — The object's handle is hidden from all functions executing on the command line and in callback routines. This setting is useful when you want to protect objects from possibly damaging user commands.

For example, if a GUI accepts user input in the form of text strings, which are then evaluated (using the `eval` function) from within the callback routine, a string such as `'close all'` could destroy the GUI. To protect against this situation, you can temporarily set `HandleVisibility` to `off` on key objects.

```
user_input = get(editbox_handle,'String');
set(gui_handles,'HandleVisibility','off')
eval(user_input)
set(gui_handles,'HandleVisibility','commandline')
```

Values Returned by gca and gcf. When a protected figure is topmost on the screen, but has unprotected figures stacked beneath it, `gcf` returns the topmost unprotected figure in the stack. The same is true for `gca`. If no unprotected figures or axes exist, calling `gcf` or `gca` causes MATLAB to create one in order to return its handle.

Accessing Protected Objects

The root `ShowHiddenHandles` property enables and disables handle visibility control. By default, `ShowHiddenHandles` is `off`, which means MATLAB obeys the setting of the `HandleVisibility` property. When `ShowHiddenHandles` is

set to `on`, all handles are visible from the command line and within callback routines. This can be useful when you want access to all graphics objects that exist at a given time, including the handles of axes text labels, which are normally hidden.

The `close` function also allows access to nonvisible figures using the `hidden` option. For example,

```
close('hidden')
```

closes the topmost figure on the screen, even if it is protected. Combining `all` and `hidden` options,

```
close('all','hidden')
```

closes all figures.

Handle Validity Versus Handle Visibility

All handles remain valid regardless of whether they are visible or not. If you know an object's handle, you can set and get its properties. By default, figure handles are integers that are displayed at the top of the window.

You can provide further protection to figures by setting the `IntegerHandle` property to `off`. MATLAB then uses a floating-point number for figure handles.

The Figure Close Request Function

In this section...

Introduction

MATLAB executes a callback routine defined by the figure's `CloseRequestFcn` whenever you

- Issue a `close` command on a figure.
- Quit MATLAB while there are visible figures. (If a figure's `Visible` property is set to `off`, MATLAB does not execute its close request function when you quit MATLAB; the figure is just deleted).
- Close a figure from the windowing system using a close box or a close menu item.

The close request function enables you to prevent or delay the closing of a figure or the termination of a MATLAB session. This is useful to perform such actions as

- Displaying a dialog box requiring the user to confirm the action
- Saving data before closing
- Preventing unintentional command-line deletion of a graphical user interface built with MATLAB

The default callback routine for the `CloseRequestFcn` is an M-file called `closereq`. It contains the statements

```
if isempty(gcbf)
   if length(dbstack) == 1
      warning('MATLAB:closereq',...
      'Calling closereq from the command line is now obsolete, use close instead');
```

```
    end
    close force
else
    delete(gcbf);
end
```

This callback honors `HandleVisibility` and therefore does not delete the figure when you use the `close` command without specifying the figure handle. For example,

```
h = figure('HandleVisibility','off')
close     % figure does not close
close all % figure does not close
close(h)  % figure closes
```

Quitting MATLAB

When you quit MATLAB, the current figure's `CloseRequestFcn` is called, and if the figure is deleted, the next figure in the root's list of children (i.e., the root's `Children` property) becomes the current figure, and its `CloseRequestFcn` is in turn executed, and so on. You can use `gcbf` to specify the figure handle from within a user-written close request function.

If you change a figure's `CloseRequestFcn` so that it does not delete the figure, then issuing the close command on that figure does not cause it to be deleted. Furthermore, if you attempt to quit MATLAB, the quit is aborted because MATLAB does not delete the figure.

Errors in the Close Request Function

If the `CloseRequestFcn` generates an error when executed, MATLAB aborts the close operation. However, errors in the `CloseRequestFcn` do not abort attempts to quit MATLAB. If an error occurs in a figure's `CloseRequestFcn`, MATLAB closes the figure unconditionally following a `quit` or `exit` command.

Overriding the Close Request Function

The `delete` command always deletes the specified figure, regardless of the value of its `CloseRequestFcn`. For example, the statement

```
delete(get(0,'Children'))
```

deletes all figures whose handles are not hidden (i.e., the figures' `HandleVisibility` property is not set to `off`). If you want to delete all figures regardless of whether their handles are hidden, you can set the root `ShowHiddenHandles` property to `on`. The root `Children` property then contains the handles of all figures. For example, the statements

```
set(0,'ShowHiddenHandles','yes')
delete(get(0,'Children'))
```

unconditionally delete all figures.

Saving Handles in M-Files

In this section...

About Saving Handles

Graphics M-files frequently use handles to access property values and to direct graphics output to a particular target. MATLAB provides utility routines that return the handles to key objects (such as the current figure and axes). In M-files, however, these utilities might not be the best way to obtain handles because

- Querying MATLAB for the handle of an object or other information is less efficient than storing the handle in a variable and referencing that variable.
- The current figure, axes, or object might change during M-file execution because of user interaction.

Save Information First

It is good practice to save relevant information about the MATLAB state in the beginning of your M-file. For example, you can begin an M-file with

```
cax = newplot;
cfig = get(cax,'Parent');
hold_state = ishold;
```

rather than querying this information each time you need it. Remember that utility commands like `ishold` obtain the values they return whenever called. (The `ishold` command issues a number of `get` commands and string compares (`strcmp`) to determine the hold state.)

If you are temporarily going to alter the hold state within the M-file, you should save the current values of the `NextPlot` properties so you can reset them later.

```
ax_nextplot = lower(get(cax,'NextPlot'));
fig_nextplot = lower(get(cfig,'NextPlot'));
```

```
.
.
.
set(cax,'NextPlot',ax_nextplot)
set(cfig,'NextPlot',fig_nextplot)
```

Properties Changed by Built-In Functions

To achieve their intended effect, many built-in functions change axes properties, which can then affect the workings of your M-file. This table lists the MATLAB built-in graphics functions and the properties they change. Note that these properties change only if `hold` is `off`.

Function	Axes Property: Set To
`fill`	`Box: on` `CameraPosition`: 2-D view `CameraTarget`: 2-D view `CameraUpVector`: 2-D view `CameraViewAngle`: 2-D view

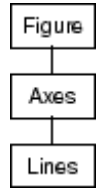
`fill3`	`CameraPosition`: 3-D view
	`CameraTarget`: 3-D view
	`CameraUpVector`: 3-D view
	`CameraViewAngle`: 3-D view
	`XScale`: `linear`
	`YScale`: `linear`
	`ZScale`: `linear`

Function	Axes Property: Set To
image (high-level)	Box: on Layer: top CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XDir: normal XLim: [0 size(CData,2)]+0.5 XLimMode: manual YDir: reverse YLim: [0 size(CData,1)]+0.5YLimMode: manual
loglog	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: log YScale: log
plot	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view

Function	Axes Property: Set To
plot3	CameraPosition: 3-D view CameraTarget: 3-D view CameraUpVector: 3-D view CameraViewAngle: 3-D view XScale: linear YScale: linear ZScale: linear
semilogx	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: log YScale: linear
semilogy	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: linear YScale: log

Objects That Can Contain Other Objects

Certain graphics objects can contain other objects. Consider a graph for example. In a graph, data is represented by an object like a line. Normally, the parent of the line is an axes (i.e., the handle of the line's `Parent` property is set to the handle of the axes that contains it). A figure is normally the parent of an axes. A typical object diagram of a graph would look like this:



When graphs become more complicated and represent data with multiple objects, it can be useful to group these objects together so you can perform operations on the group as a whole.

The following sections discuss how to use two container objects that group axes children within a graph and user interface components within a figure.

Using Panel Containers in Figures — Uipanels

In this section...

Introduction

Figures can contain axes and user interface objects directly, or you can parent these objects to uipanels, which you then parent to a figure. Uipanels are useful for the design of GUIs because they enable you to define subregions in a figure in which you can lay out components.

MATLAB interprets the `Position` property of all objects parented to a uipanel relative to the uipanel's position. If you move the uipanel, the children automatically move with it.

Uipanels can also contain other uipanels, as well as axes, uicontrols, and uibuttongroups. See the `uipanel` reference page for more information on uipanels.

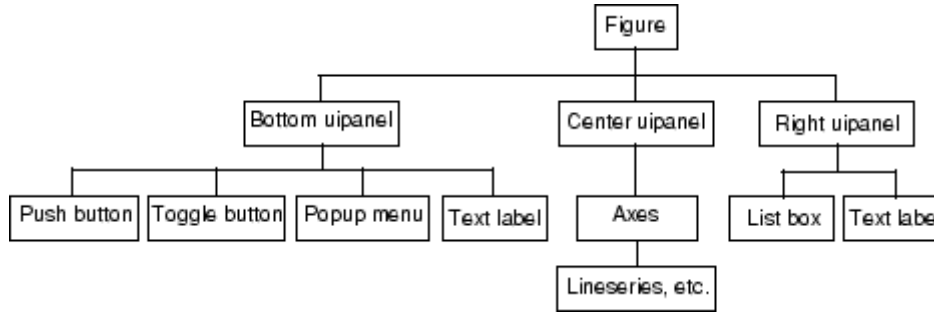
You can create multiple axes in a uipanel and direct plotting into any of them. However, some plotting functions do not allow you to specify the parent of the graphics objects they create, so they create a new axes (and possibly a figure). To include such a graph in a uipanel, you can reparent the axes to the panel once the plot is made.

Figure Resize Functions

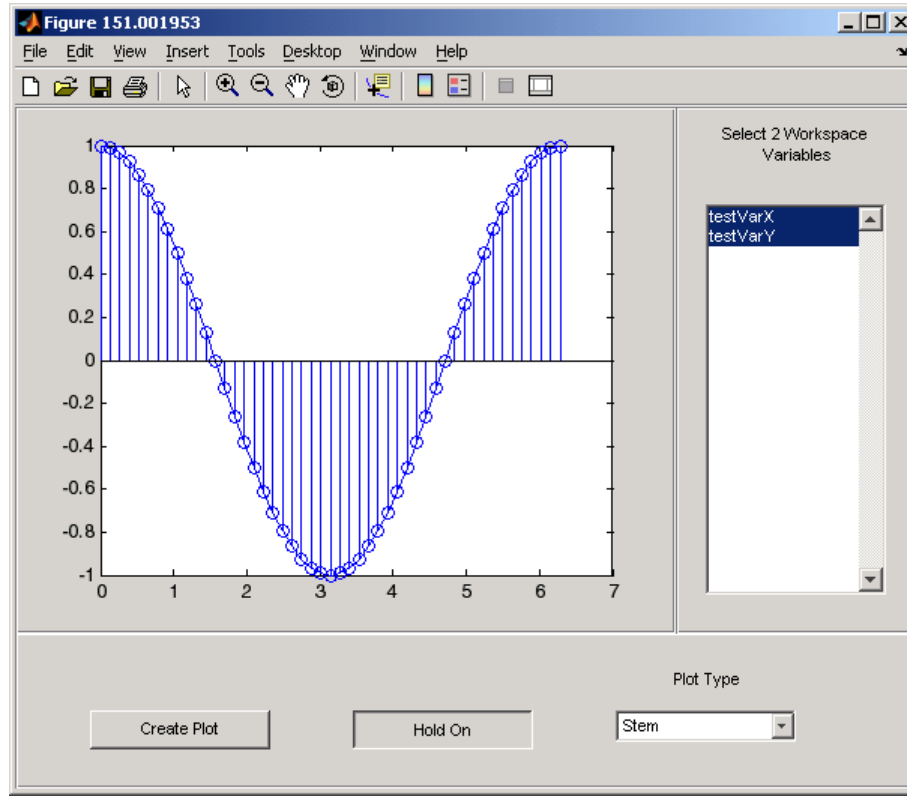
Containing various parts of a GUI in uipanels simplifies the process of programming figure resize behavior because you can write a separate resize function for each panel. The following example illustrates how to do this.

Example — Using Figure Panels

This example uses three uipanel objects as containers for the GUI's components. All three uipanels are then parented to the figure, as shown in the following containment hierarchy.



Here is a picture of the GUI with some data plotted in the axes.



Complete Example Code

Note If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB editor.

This GUI enables you to select workspace variables from a list box and select a plot type from a pop-up menu. You can add plots to the existing graph by clicking the **Hold** toggle button and initiate the plot by clicking the **Create Plot** button.

Use the link above to run the example and open the GUI code in the MATLAB editor.

Creating the Uipanels

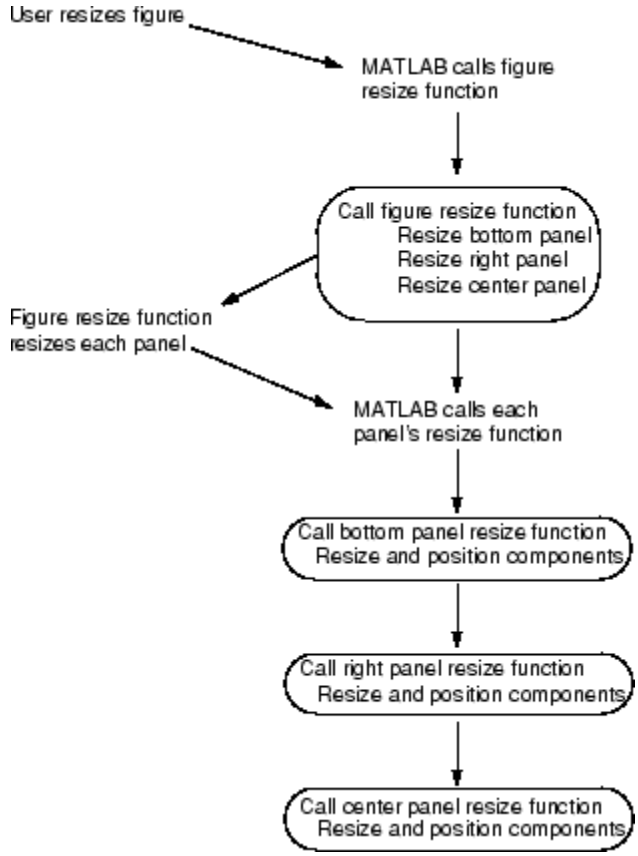
The following code shows the definition of the figure and the bottom panel. Setting `Units` to `characters` ensures that your GUI is properly sized on different computer systems. The `Position` property specifies the location and size of each component in units set by the `Units` property.

```
% Create the figure
f = figure('Units','characters',...
  'Position',[30 30 120 35],...
  'Color',panelColor,...
  'HandleVisibility','callback',...
  'IntegerHandle','off',...
  'Renderer','painters',...
  'ResizeFcn',@figResize);
% Create the bottom uipanel
botPanel = uipanel('BorderType','etchedin',...
  'BackgroundColor',panelColor,...
  'Units','characters',...
  'Position',[1/20 1/20 119.9 8],...
  'Parent',f,...
  'ResizeFcn',@botPanelResize);
```

Programming the Resize Functions

As you resize the figure, MATLAB calls the figure resize function (specified by the object's `ResizeFcn` property), which, in this example, computes a new size for each uipanel. Because the figure resize function resizes the uipanels, MATLAB automatically calls the resize function of each uipanel once the figure resize function completes execution. The uipanel resize functions then adjust the sizes and locations of the components they contain.

The following diagram illustrates the sequence of events that occurs when a user resizes the figure.



The following code shows the figure, bottom panel, and right panel resize functions. As each function is called, it sets the object's size and position to values that are proportional to the original layout.

See "Nested Functions" for more information.

```
% Figure resize function
function figResize(src,evt)
 fpos = get(f,'Position');
```

```
 set(botPanel,'Position',...
  [1/20 1/20 fpos(3)-.1 fpos(4)*8/35])
 set(rightPanel,'Position',...
   [fpos(3)*85/120 fpos(4)*8/35 fpos(3)*35/120 fpos(4)*27/35])
 set(centerPanel,'Position',...
  [1/20 fpos(4)*8/35 fpos(3)*85/120 fpos(4)*27/35]);
end
% Bottom panel resize function
function botPanelResize(src,evt)
 bpos = get(botPanel,'Position');
 set(plotButton,'Position',...
  [bpos(3)*10/120 bpos(4)*2/8 bpos(3)*24/120 2])
 set(holdToggle,'Position',...
  [bpos(3)*45/120 bpos(4)*2/8 bpos(3)*24/120 2])
 set(popUp,'Position',...
  [bpos(3)*80/120 bpos(4)*2/8 bpos(3)*24/120 2])
 set(popUpLabel,'Position',...
  [bpos(3)*80/120 bpos(4)*4/8 bpos(3)*24/120 2])
end
% Right panel resize function
function rightPanelResize(src,evt)
 rpos = get(rightPanel,'Position');
 set(listBox,'Position',...
  [rpos(3)*4/32 rpos(4)*2/27 rpos(3)*24/32 rpos(4)*20/27]);
 set(listBoxLabel,'Position',...
  [rpos(3)*4/32 rpos(4)*24/27 rpos(3)*24/32 rpos(4)*2/27]);
end
```

Note that the center panel does not need a resize function because the axes automatically resize to fit the container (either a figure or uipanel).

To see the complete code listing for this example, see “Complete Example Code” on page 8-91.

Grouping Objects Within Axes — hgtransform

In this section...

Introduction

MATLAB provides two objects that are designed to group any of the objects normally parented to axes. These two objects are

- Hggroup — Parent objects to an hggroup object when you want to reference the objects as a group. For example, to select or control visibility of all the group members.
- Hgtransform — This object also enables you to transform (rotate, translate, etc.) the objects as a group.

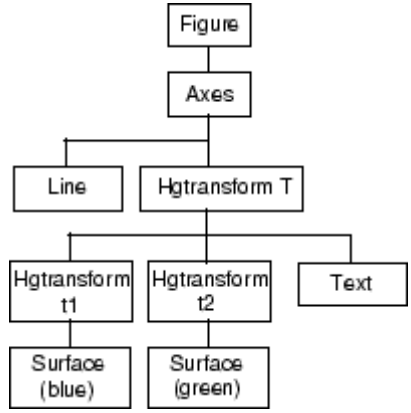
See "Group Objects" on page 8-28 for more information about hggroup and hgtransform objects.

Example — Translating Grouped Objects

This example shows how using a hierarchy of hgtransform objects makes it possible to translate the contained graphics objects both independently and as a group. The example creates a cross-like cursor with a text readout in the center, which displays data values.

The cursor is constructed from two surfaces, each of which is contained in an hgtransform object so they can be translated independently to overlap, and a text object. These two hgtransform objects are then contained by a third hgtransform object, which also contains the text. This third hgtransform (with handle `T` in the diagram and code) enables the cursor to be transformed as a group.

The following diagram shows the containment hierarchy for this example. The axes contains a line, which is used to plot the data that the cursor moves along. The axes also contains the hierarchy of hgtransform objects that construct the cursor.



Note If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB editor.

Set Up the Axes and Figure

The first step is to create an axes with fixed limits so MATLAB does not rescale the limits as the cursor moves along the line. Creating the axes automatically creates a figure to contain it.

Set figure properties to use the OpenGL renderer:

```
h_axes = axes('XLim',[-10 10],'YLim',[-5 5]);
set(get(h_axes,'Parent'),'Renderer','opengl')
```

Define the Transform Matrices and Hgtransform Objects

The cross part of the cursor is formed from two surface objects, which are translated to overlap. Each surface is contained in its own hgtransform object (handles `t1` and `t2`) because they are translated in different directions. Both hgtransform objects are themselves contained in another hgtransform object (handle `T`).

See `makehgtform`, `hgtransform`.

```
% Create transform matrices
tmtx1 = makehgtform('translate',[-.5 0 0]);
tmtx2 = makehgtform('translate',[0 -.5 0]);

% Create hgtransform objects
T = hgtransform; % Contains the cursor
t1 = hgtransform('Parent',T,'Matrix',tmtx1);
t2 = hgtransform('Parent',T,'Matrix',tmtx2);
```

Create the Surface and Text Objects

The cursor is composed of two surface objects and a text object (to display data values). The two surfaces are parented to their respective hgtransform objects. The text is parented directly to the top-level hgtransform. The text object does not need coordinates because it is translated along with the surfaces in the top-level hgtransform object (`T`).

See `cylinder`, `surface`, `text`.

```
% Define surfaces and text
[sx,sy,sz] = cylinder([0 2 0]); % Use cylinder to generate data
surface(sz,sy,sx,'FaceColor','green',...
                 'EdgeColor','none','FaceAlpha',.2,'Parent',t1);
surface(sx,sz./1.5,sy,'FaceColor','blue',...
                 'EdgeColor','none','FaceAlpha',.2,'Parent',t2);
h_text = text('FontSize',12,'FontWeight','bold',...
                 'HorizontalAlignment','center',...
                 'VerticalAlignment','Cap','Parent',T);
```

Generate Data and Plot a Line

This example uses a line plot of a mathematical function to create a path along which to move the cursor.

```
% Plot the data x, y, and z
x = -10:.05:10;
y = [cos(x) + exp(-.01*x).*cos(x) + exp(.07*x).*sin(3*x)];
z = 1:length(x);
line(x,y)
```

Translate the Cursor Along the Plotted Line

To move the cursor along the line, a new transform matrix is calculated using each set of `x`, `y`, and `z` data points and used to set the `Matrix` property of the top-level hgtransfrom `T`. At the same time, the text object `String` property is updated to display the value of the current `y` data point.

The surfaces and the text translate together because all are contained in the top-level hgtransform object.

```
% Loop through the line data to move the cursor
for ind = 1:length(x)
    set(T,'Matrix',...
   makehgtform('translate',[x(ind) y(ind) z(ind)]))
    set(h_text,'String',num2str(y(ind)))
    drawnow,pause(.01)
end
```

Controlling Legends

In this section...

Legend Control Options

Graphics objects that are used to represent data, such as lines, surfaces, patches, etc. can be represented in figure legends (see legend for information on creating legends). By setting object properties, you can:

- Include a particular graphics object in the legend (the default)
- Exclude a particular graphics object from the legend
- Group graphics object together by parenting them to an hggroup or hgtransform object and represent the group as a single item in the legend ("Group Objects" on page 8-28)
- Display only the children of an object and not the parent in the legend. This is useful when the graph contains plot objects ("Plot Objects" on page 8-19)
- You can specify the text label used in the legend for any object

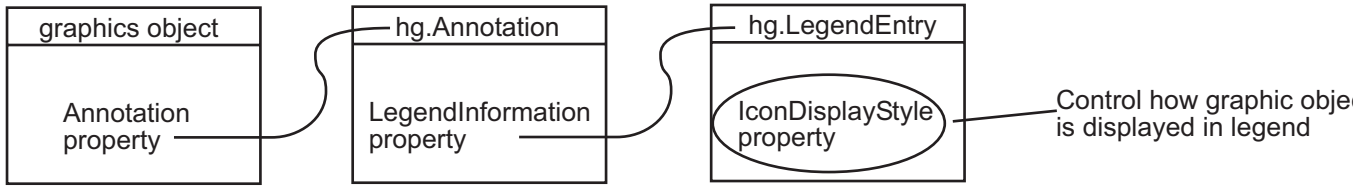
Properties for Controlling Legend Content

Graphics objects have two properties that control the options described above:

- `Annotation` — controls whether the graphics object appears in the legend and determines if the object or its children appear in the legend.

- `DisplayName` — specifies the text label used by the legend for the object. However, specifying a string with the legend commands resets the value of `DisplayName` property.

Accessing the Annotation Control Objects



Querying the `Annotation` property returns the handle of an `hg.Annotation` object. The `hg.Annotation` object has a property called `LegendInformation`, which contains an `hg.LegendEntry` object. The `hg.LegendEntry` object has a property called `IconDisplayStyle` that you can set to one of three values.

IconDisplayStyle Value	Behavior
`on`	Represent this object in a figure legend
`off`	Do not include this object in a figure legend
`children`	Display legend entries for this object's children and not the object itself (applies only to objects that have children, otherwise, the same a `on`)

For example, if `object_handle` is the handle of a graphics object, you can use the following statements to set the object's `IconDisplayStyle`. In this case, the graphics object, `object_handle`, is not included in the legend because its `IconDisplayStyle` property is set to `off`.

```
hAnnotation = get(object_handle,'Annotation');
hLegendEntry = get(hAnnotation','LegendInformation');
set(hLegendEntry,'IconDisplayStyle','off')
```

Updating a Legend

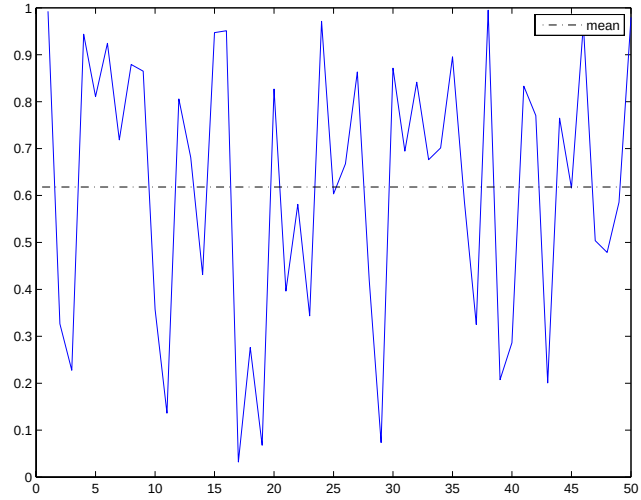
If a legend exist and you change its `IconDisplayStyle` setting, you must call `legend` to update the display. See the legend command for the options available.

Example — Excluding a Particular Object From a Legend

This example creates a graph (using random data in this case) and also draws a line that indicates the mean value of the data. The legend displays a key for the mean data line only because the blue data line object has its.`IconDisplayStyle` property of the associated `LegendEntry` object set to `off`.

```
function annotation_property_line
dat = rand(50,1);
hLine = plot(dat);
plotMean % Nested function draws a line at mean value
set(get(get(hLine,'Annotation'),'LegendInformation'),...
    'IconDisplayStyle','off'); % Exclude line from legend
legend('mean')
    function plotMean
    xlimits = get(gca,'XLim');
    meanValue = mean(dat);
    meanLine = line([xlimits(1) xlimits(2)],[meanValue meanValue],...
        'Color','k','LineStyle','-.');
    end
end
```

Here is the graph:



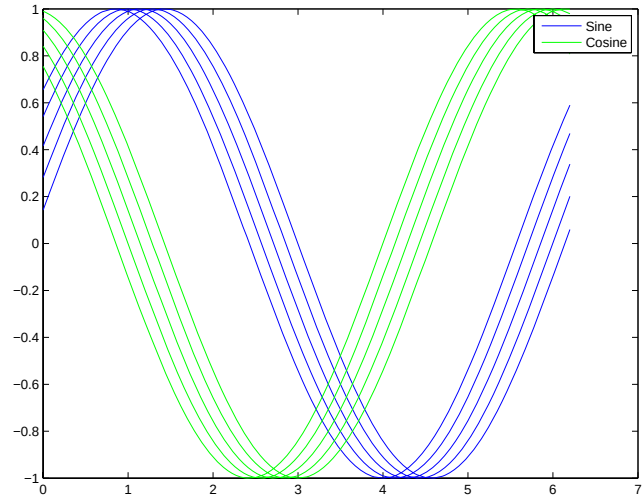
Example — One Legend Entry for a Group of Objects

You can group graphics objects in an hggroup or hgtransform object and represent the whole group as one item in a legend. This example creates two series of graphs (sines and cosines of the same data).

- The lines drawn to represent the sine are parented to one hggroup object
- The lines drawn to represent the cosine are parented to another hggroup object
- Both hggroup objects need their associated `IconDisplayStyle` property set to `on`
- The legend then displays entries for both hggroup objects, but not their children (the plotted lines)

```
t = 0:.1:2*pi;
for k=1:5
    offset = k/7;
    m(:,k) = t+offset';
end
```

```
hSLines = plot(t,sin(m),'Color','b');hold on
hCLines = plot(t,cos(m),'Color','g');
hSGroup = hggroup;
hCGroup = hggroup;
set(hSLines,'Parent',hSGroup)
set(hCLines,'Parent',hCGroup)
set(get(get(hSGroup,'Annotation'),'LegendInformation'),...
    'IconDisplayStyle','on'); % Include this hggroup in the legend
set(get(get(hCGroup,'Annotation'),'LegendInformation'),...
    'IconDisplayStyle','on'); % Include this hggroup in the legend
legend('Sine','Cosine')
```

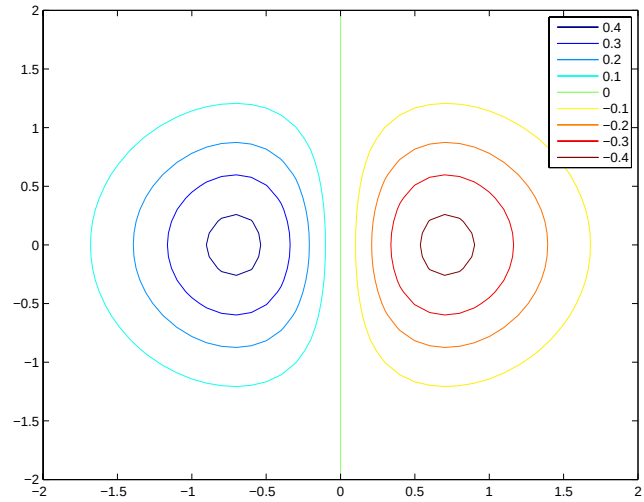


Example — Showing Children of Group Objects in Legend

You can include the children of a group in the legend by setting the group object's `IconDisplayStyle` to `children`. This is useful when graphs contain plot objects, which are groups of core graphics objects. For example, consider the following contour graph.

```
[X,Y] = meshgrid(-2:.1:2);
```

```
Z = X.*exp(-X.^2-Y.^2);
[mC hC] = contour(X,Y,Z);
set(get(get(hC,'Annotation'),'LegendInformation'),...
    'IconDisplayStyle','Children');
%{
Assigns each line object's DisplayName property a string
based on the value of the contour interval it represents
%}
k =1; ind = 1; hLines = get(hC,'Children');
while k < size(mC,2),
   set(hLines(ind),'DisplayName',num2str(mC(1,k)))
   k = k+mC(2,k)+1; ind = ind+1;
end
% Display the legend using DisplayName labels
legend('show')
```

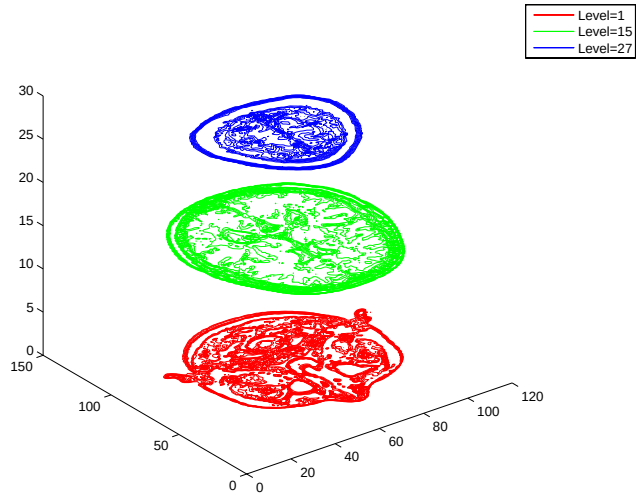


Example — Grouping Objects to Reduce the Legend Entries

Some functions that visualize large data sets create many objects to render graphs. For example, `contourslice` uses patch objects to generate contour

slices of volume data. This example groups the 1829 patch objects into hggroup objects according to which plane the objects represent and sets corresponding values for the `DisplayName` property, resulting in a legend with only three items.

```
load mri
D = squeeze(D);
phandles = contourslice(D,[],[],[1,15,27],8);view(3)
gh(1) = hggroup; gh(2) = hggroup; gh(3) = hggroup;
%set(gh,'Parent',gca)
for k=1:length(phandles)
   zd = get(phandles(k),'ZData');
   plane = num2str(zd(1));
   switch plane
      case '1'
         set(phandles(k),'Parent',gh(1),'EdgeColor','r')
      case '15'
         set(phandles(k),'Parent',gh(2),'EdgeColor','g')
      case '27'
         set(phandles(k),'Parent',gh(3),'EdgeColor','b')
      otherwise
         disp('Don''t know what to do with it')
   end
end
hA = get(gh,'Annotation');
hLL = get([hA{:}],'LegendInformation');
set([hLL{:}],{'IconDisplayStyle'},...
   {'on','on','on'}')
set(gh,{'DisplayName'},{'Level=1','Level=15','Level=27'}')
legend show
```


Level=1
Level=15
Level=27
30
25
20
15
10
5
0
150
100
50
0
0
20
40
60
80
100
120

Callback Properties for Graphics Objects

In this section...

What is a Callback?

A *callback* is a function that executes when a specific event occurs on a graphics object. You specify a callback by setting the appropriate property of the object. This section describes the events (specified via properties) for which you can define callbacks. See "Function Handle Callbacks" on page 8-109 for information on how to define callbacks.

Graphics Object Callbacks

All graphics objects have three properties for which you can define callback routines:

- `ButtonDownFcn` — Executes when users click the left mouse button while the cursor is over the object or within a 5-pixel border around the object
- `CreateFcn` — Executes during object creation after all properties are set
- `DeleteFcn` — Executes just before deleting the object

User Interface Object Callbacks

User interface objects (e.g., uicontrol and uimenu) have a `Callback` property through which you define the function to execute when users activate these devices (e.g., click a push button or select a menu).

Figure Callbacks

Figures have additional properties that execute callback routines with the appropriate user action. Only the `CloseRequestFcn` property has a callback defined by default:

- `CloseRequestFcn` — Executes when a request is made to close the figure (by a `close` command, by the window manager menu, or by quitting MATLAB)
- `KeyPressFcn` — Executes when users press a key while the cursor is within the figure window
- `ResizeFcn` — Executes when users resize the figure window
- `WindowButtonDownFcn` — Executes when users click a mouse button while the cursor is over the figure background, a disabled uicontrol, or the axes background
- `WindowButtonMotionFcn` — Executes when users move the mouse button within the figure window (but not over menus or title bar)
- `WindowButtonUpFcn` — Executes when users release the mouse button, after having pressed the mouse button within the figure

Function Handle Callbacks

In this section...

Introduction

Handle Graphics objects have a number of properties for which you can define callback functions. When a specific event occurs (e.g., a user clicks a push button or deletes a figure), the corresponding callback function executes. You can specify the value of a callback property as a

- String that is a MATLAB command or the name of an M-file
- Cell array of strings
- Function handle or a cell array containing a function handle and additional arguments

The following sections illustrate how to define function handle callbacks for Handle Graphics objects.

- "Introduction" on page 8-109 describes how to define a function handle callback.
- "Why Use Function Handle Callbacks" on page 8-111 provides information on the advantages of using function handle callbacks.
- "Example — Using Function Handles in GUIs" on page 8-113 shows how to create a simple GUI that uses function handle callbacks.

For general information on function handles, see the function handle reference page.

Function Handle Syntax

In Handle Graphics, functions that you want to use as function handle callbacks must define at least two input arguments in the function definition:

- The handle of the object generating the callback (the source of the event)
- The event data structure (can be empty for some callbacks)

MATLAB passes these two arguments implicitly whenever the callback executes. For example, consider the following statements, which are contained in a single M-file.

```
function myGui
% Create a figure and specify a callback
figure('WindowButtonDownFcn',@myCallback)
   .
   .
   .
% Callback subfunction defines two input arguments
function myCallback(src,eventdata)
   .
   .
   .
```

The first statement creates a figure and assigns a function handle to its `WindowButtondownFcn` property (created by using the @ symbol before the function name). This function handle points to the subfunction `myCallback`. The definition of `myCallback` must specify the two required input arguments in its function definition line.

Passing Additional Input Arguments

You can define the callback function to accept additional input arguments by adding them to the function definition. For example,

```
function myCallback(src,eventdata,arg1,arg2)
```

When using additional arguments for the callback function, you must set the value of the property to a cell array (i.e., enclose the function handle and arguments in curly braces). For example,

```
figure('WindowButtonDownFcn',{@myCallback,arg1,arg2})
```

Defining Callbacks as a Cell Array of Strings — Special Case

Defining a callback as a cell array of strings is a special case because MATLAB treats it differently from a simple string. Setting a callback property to a string causes MATLAB to evaluate that string in the base workspace when the callback is invoked. However, setting a callback to a cell array of strings requires the following:

- The cell array must contain the name of an M-file that is on the MATLAB path as the first string element.
- The M-file callback must define at least two arguments (the handle of the callback object and an empty matrix).
- Any additional strings in the cell array are passed to the M-file callback as arguments.

For example,

```
figure('WindowButtonDownFcn',{myCallback,arg1})
```

requires you to define a function M-file that uses three arguments,

```
function myCallback(src,eventdata,arg1)
```

Why Use Function Handle Callbacks

Using function handles to specify callbacks provides some advantages over the use of strings, which must be either MATLAB commands or the name of an M-file that will be on the MATLAB path at run-time.

Single File for All Code

Function handles enable you to use a single M-file for all callbacks. This is particularly useful when you are creating graphical user interfaces, because you can include both the layout commands and callbacks in one file.

For information on how to access subfunctions, see the “Calling a Function Using Its Handle” section of MATLAB Programming.

Keeping Variables in Scope

When MATLAB evaluates function handles, the same variables are in scope as when the function handle was created. (In contrast, callbacks specified as strings are evaluated in the base workspace.) This simplifies the process of managing global data, such as object handles in a GUI.

For example, suppose you create a GUI with a list box that displays workspace variables and a push button whose callback creates a plot using the variables selected in the list box. The push button callback needs the handle of the list box to query the names of the selected variables. Here's what to do.

1. Create the list box and save the handle:

```
h_listbox = uicontrol('Style','listbox',... etc.);
```

2. Pass the list box handle to the push button's callback, which is defined in the same M-file:

```
h_plot_button = uicontrol('Style','pushbutton',...
 'Callback',{@plot_button_callback,h_listbox},...,etc.);
```

The handle of the list box is now available in the plot button's callback without relying on global variables or using `findobj` to search for the handle. See "Example — Using Function Handles in GUIs" on page 8-113 for an example that uses this technique.

Callback Object Handle and Event Data

MATLAB passes additional information to the callback when it is executed. This information includes the handle of the callback object (the source of the callback event) and event data that is specific to the particular callback property.

For example, the event data returned for the figure `KeyPressFcn` property is a structure that contains information about which keys were pressed.

Information about the event data associated with any given callback property is included with the property's documentation. Use the Handle Graphics Property Browser to access property documentation.

Function Handles Stay in Scope

A function handle can point to a function that is not in scope at the time of execution. For example, the function can be a subfunction in another M-file.

For a general discussion of function handles, see the "Function Handles" and "Anonymous Functions" in the MATLAB documentation.

Example — Using Function Handles in GUIs

This example creates a simple GUI that plots workspace variables. It is defined in a single M-file that contains both the layout commands and the callbacks. This example uses function handles to specify callback functions. Callbacks are implemented as nested functions to reduce the need to pass variables as arguments.

Complete Example Code

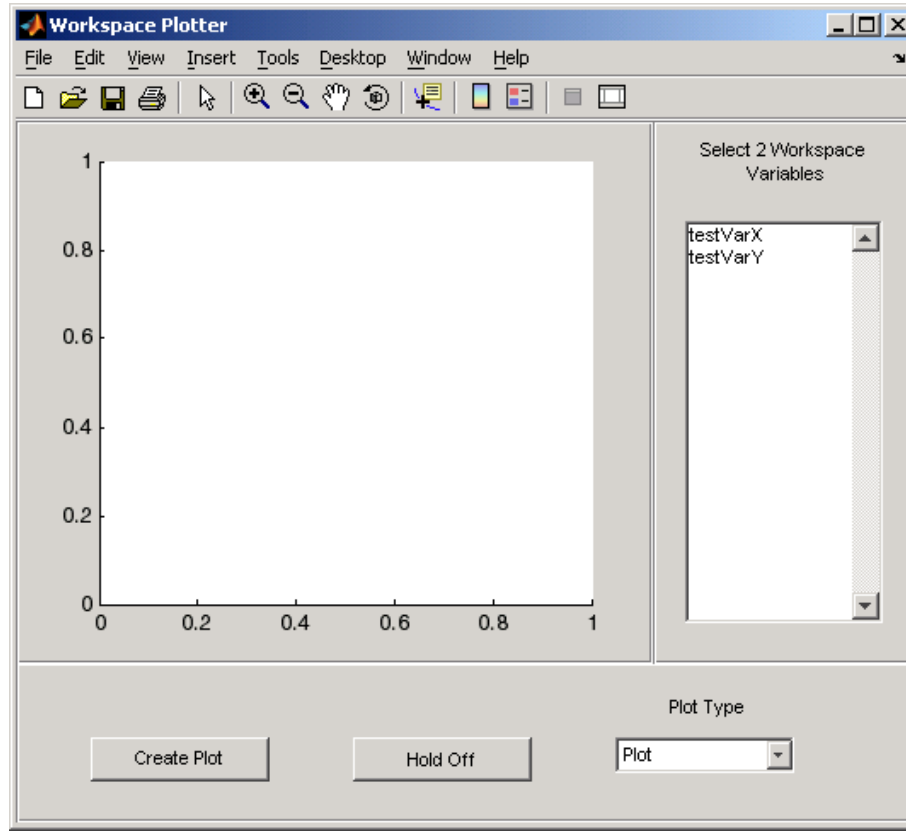
The documentation for this example does not list all the code used to lay out and program the GUI. To see a complete code listing, use the links in the note box below.

Note If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB editor.

See "Function Handle Callbacks" on page 8-109 for more information on the use of function handle callbacks.

The GUI Layout

The following picture shows the GUI after running the example code. The program creates two variables (`testvarX` and `testVarY`) in the base workspace for testing purposes.



The GUI layout is split among three uipanel containers. One contains the axes, the right side panel contains a list box to display workspace variables, and the bottom panel contains the plot and hold buttons and the plot type pop-up menu.

Initialize the GUI

The list box and the hold toggle button need to be initialized before the GUI is ready to use. This is accomplished by executing their callbacks. Note that because you are calling these functions directly, MATLAB does not implicitly pass the first two arguments, as it would if these functions were executed as callbacks in response to an event. You therefore must explicitly pass all arguments in these function calls.

```
% Initialize list box and make sure
% the hold toggle is set correctly
listBoxCallback(listBox,[])
holdToggleCallback(holdToggle,[])
```

The Callback Functions

The GUI components that have callbacks are the list box, toggle button, and plot push button. In addition, the figure's three uipanels define resize functions that MATLAB executes whenever users resize the figure.

See "Programming the Resize Functions" on page 8-92 for information on writing callback functions for the figure and uipanel `ResizeFcn` properties.

List Box Callback. The list box callback generates a list of the current variables in the base workspace using the `evalin` and `who` functions. It then assigns this list to the list box `String` property so that it displays these variable names.

Note how the function takes advantage of the fact that the first argument passed to the callback is the handle of the callback object (i.e., the source of the callback event, which is the list box). Therefore, whenever you click in the list box, MATLAB updates the list to display the current workspace variables.

```
%% Callback for list box
function listBoxCallback(src,evt)
 % Load workspace vars into list box
 vars = evalin('base','who');
 set(src,'String',vars)
end % listBoxCallback
```

Plot Button Callback. The plot button callback performs three tasks:

- Gets the names of the variables selected by the user in the list box
- Gets the type of plot selected by the user in the pop-up menu
- Constructs and evaluates the plotting command in the base workspace

```
%% Callback for plot button
function plotButtonCallback(src,evt)
 % Get workspace variables
```

```
    vars = get(listBox,'String');
    var_index = get(listBox,'Value');
    if length(var_index) ~= 2
     errordlg('You must select two variables',...
     'Incorrect Selection','modal')
     return
    end
    % Get data from base workspace
    x = evalin('base',vars{var_index(1)});
    y = evalin('base',vars{var_index(2)});
    % Get plotting command
    selected_cmd = get(popUp,'Value');
    % Make the GUI axes current and create plot
    axes(a)
    switch selected_cmd
    case 1 % user selected plot
     plot(x,y)
    case 2 % user selected bar
     bar(x,y)
    case 3 % user selected stem
     stem(x,y)
    end
  end % plotButtonCallback
```

Hold State Toggle Button Callback. The toggle button callback requires the handles of the GUI figure and axes. Because these callbacks are written as nested functions, the figure handle (`f`) and the axes handle (`a`) are in scope within the callback.

You want the GUI to toggle the hold state, but the GUI figure handle is hidden. It is necessary, therefore, to use the axes handle as the first argument to the `hold` function.

```
%% Callback for hold state toggle button
function holdToggleCallback(src,evt)
 button_state = get(src,'Value');
 if button_state == get(src,'Max')
  % toggle button is depressed
  hold(a,'on')
  set(src,'String','Hold On')
```

```
 elseif button_state == get(src,'Min')
  % toggle button is not depressed
  hold(a,'off')
  set(src,'String','Hold Off')
 end
end % holdToggleCallback
```

Optimizing Graphics Performance

In this section...

Introduction

This section discusses techniques that can help increase the speed with which MATLAB creates graphs. These techniques generally apply to cases where you are creating many graphs of similar data and can therefore improve rendering speed by preventing MATLAB from performing unnecessary operation.

Whether a given technique improves performance depends on the particular application. The `profile` function can help you determine where your code is consuming the most time.

General Performance Guidelines

The following list provides some general guidelines for optimizing performance:

- Set automatic-mode properties to manual whenever possible to prevent MATLAB from performing unnecessary operations.
- Modify existing objects instead of creating new ones.
- Use low-level core objects when creating objects repeatedly.
- Do not recreate legends or other annotations in a program loop; add these after you finish modifying the graph.

- Set the text `Interpreter` property to `none` if you are not using T_eX characters.
- Try various renderers and erase modes. MATLAB might not have auto-selected the fastest renderer for your application.

The remainder of this section provides more details on these and other techniques.

- "Disable Automatic Modes" on page 8-119 — information on optimizing the use of axes objects.
- "Changing Graph Data Rapidly" on page 8-121 — an example of techniques for interactive plotting.
- "Performance of Bit-Mapped Images" on page 8-125 — information on optimizing the use of image objects.
- "Performance of Patch Objects" on page 8-126 — information on optimizing the use of patch objects.
- "Performance of Surface Objects" on page 8-127 — information on optimizing the use of surface objects.

Disable Automatic Modes

Graphics objects have properties that control many aspects of their behavior and appearance. The axes object in particular has many mode properties that enables it to respond to changes in the data represented in a graph.

For example, when you plot data, the axes determines appropriate axis limits, tick-mark placement, and labeling. Any changes you make to the plotted data (adding another line graph, for example) causes the axes to recompute the axis limits and to determine what values to use for the tick marks.

Fixing Axis Limits

The process of recalculating axis limits and the locations of the tick marks along each axis contributes to the time it takes to create a graph. If you are plotting data into the same axes repeatedly, you can improve performance by manually setting some or all of the axis limits, which, in turn, disables axis scaling and tick picking.

For example, suppose you are plotting time series graphs in which you always view a 200 second time interval along the x-axis and your data ranges from -1 to 1. The following statement creates an axes with these limits and, in the process, sets the limit-picking mode to manual. Thereafter, MATLAB does not change the limits or recalculate tick mark locations (as long as you do not call a high-level plotting function like `plot`):

```
axes('XLim',[0 200],'YLim',[-1 1])
```

Set All Modes to Manual

To maximize the efficiency with which MATLAB can update your graphs, you should disable all automatic operation so that MATLAB does not need to spend time determining if it is even necessary to recalculate a property value. The following steps illustrate this technique:

1. Create a figure and select the renderer you want to use. Line graphs should use `painters` to take advantage of its line thinning algorithm.

   ```
   figure('Renderer','painters')
   ```

   Setting a property automatically sets its associated mode property to manual.

2. Create an axes explicitly and set all properties (such as the axis limits) for which you can predetermine the appropriate value.

3. Set all other mode property values to `manual` (see table below).

4. If you are creating line graphs using the `painters` renderer, set the axes `DrawMode` property to `fast`.

5. If you cannot determine the appropriate value for all mode properties, create your first graph and then use the `set` command to set mode properties to `manual`. See “Changing Graph Data Rapidly” on page 8-121 for an example.

The following table lists the axes mode properties and provides an explanation of what the mode controls.

Mode Property	What It Controls
`ALimMode`	Transparency limits mode
`CameraPositionMode`	Positioning of the viewpoint
`CameraTargetMode`	Positioning of the camera target in the axes
`CameraUpVectorMode`	The direction of "up" in 2-D and 3-D views
`CameraViewAngleMode`	The size of the projected scene and stretch-to-fit behavior
`CLimMode`	Mapping of data values to colors
`DataAspectRatioMode`	Relative scaling of data units along x-, y-, and z-axes and stretch-to-fit behavior
`DrawMode`	Controls the way MATLAB renders graphics objects (use with line graphs)
`PlotBoxAspectRatioMode`	Relative scaling of plot box along x-, y-, and z-axes and stretch-to-fit behavior
`TickDirMode`	Direction of axis tick marks (in for 2-D, out for 3-D)

`XLimMode`
`YLimMode`
`ZLimMode`
	Limits of the respective x, y, and z axes

`XTickMode`
`YTickMode`
`ZTickMode`
	Tick mark spacing along the respective x-, y-, and z-axes

`XTickLabelMode`
`YTickLabelMode`
`ZTickLabelMode`
	Tick mark labels along the respective x-, y-, and z-axes

Changing Graph Data Rapidly

MATLAB plotting functions perform a wide variety of operations in the process of creating a graph to make plotting easier. For example, the `plot`

function clears the current axes before drawing new lines, selects a line color or a marker type, searches for user-defined default values, and so on.

Low-Level Functions for Speed

The features that make plotting functions easy to use also consume computer resources. If you want to maximize graphing performance, you should use low-level functions and disable certain automatic features.

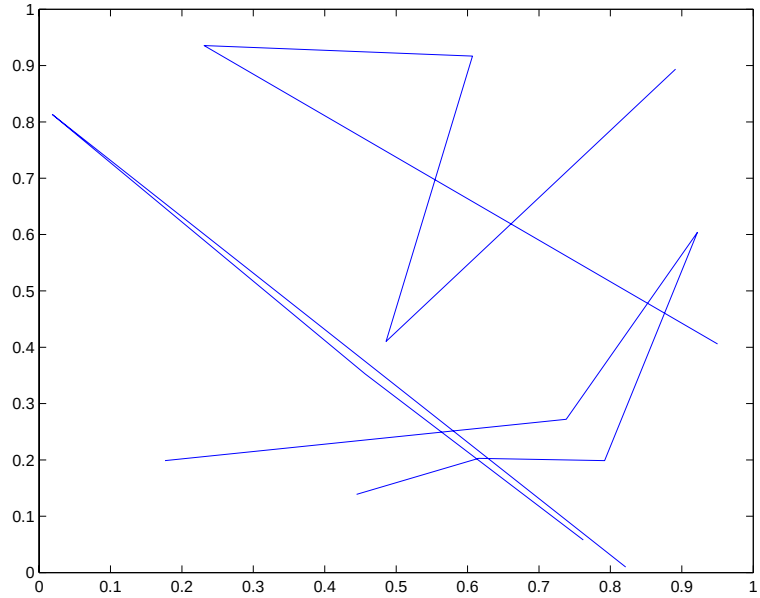
Low-level graphics functions (e.g., `line` vs. `plot`, surface vs. surf) perform fewer operation and therefore can be faster when you are creating many graphics objects. See "High-Level Versus Low-Level" on page 8-17 for more information on how these functions differ.

Avoid Creating Graphics Objects

Each graphics object requires a certain amount of the computer's resources to create and store information, such as the value of all the object's properties. It is therefore more efficient to use fewer graphics objects if possible.

For example, you can add NaNs to vertex data (which causes that vertex to not be rendered) to create line segments that look like separate lines. You must place the NaNs at identical locations in each vector of data:

```
x = [rand(5,1);nan;rand(4,1);nan;rand(6,1)];
y = [rand(5,1);nan;rand(4,1);nan;rand(6,1)];
line(x,y);
```



Update the Object's Data

If you want to view different data on what is basically the same graph, it is more efficient to update the data of the existing objects (lines, text, etc.) rather than recreating the entire graph.

For example, suppose you want to visualize the effect on your data of varying certain parameters. Follow these steps:

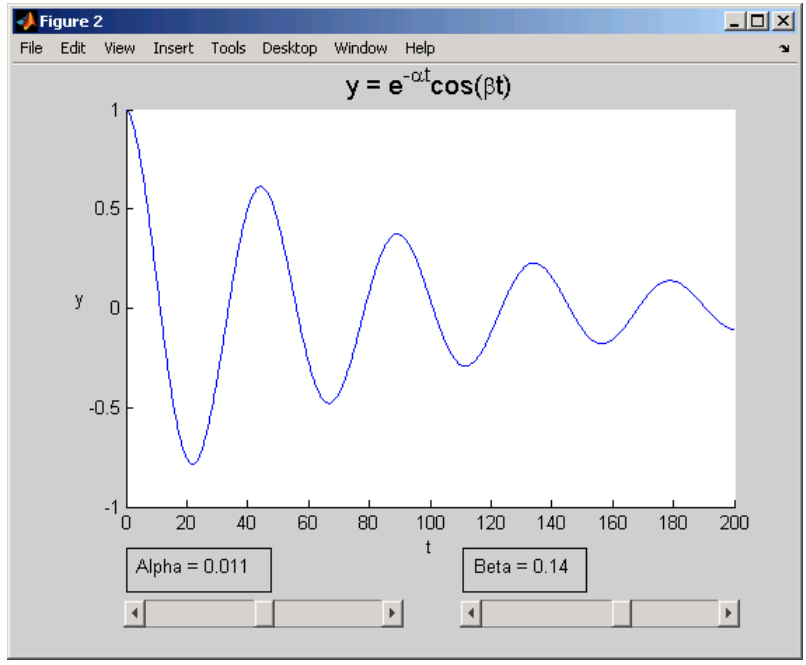
1. Set the limits of any axis that can be determined in advance (you can use `max` and `min` to determine the range of your data).

2. Recalculate the data using the new parameters.

3. Use the new data to update the data properties of the lines, text, etc. objects used in the graph.

4 Call `drawnow` to flush the event queue and update the figure (and all child objects in the figure).

The following example illustrates the use of these techniques in a GUI that uses sliders to vary two parameters in a mathematical expression, which is then plotted.



Note If you are using the MATLAB Help browser, you can run this example or open it in the MATLAB editor.

Specify Axes with Plotting Function for Better Performance

Most plotting functions accept an axes handle as an argument. This handle determines the target axes for the plot. Specifying the parent axes as an argument is often faster than using the `axes` function to make a particular

axes the current axes. For example, suppose you want six subplot in one figure and want to plot to each one in sequence:

```
x = 0:.05:2*pi;
for k=1:6
    h(k) = subplot(6,1,k);
end
for k=1:6
    y = sin(x*100*rand)*rand;
    plot(h(k),x,y) % Specify target axes in plotting function
    % Avoid using axes(h(k); plot(x,y) in a loop
end
```

Keeping Track of the Target Figure and Axes

You can also explicitly set the figure `CurrentAxes` property to avoid specifying the axes handle with a number of functions that can operate on the current axes by default. For example, in the code below, both `stem` and `axis` operate on the current axes if you do not specify an axes as an argument.

```
fHandle = figure;
for k=1:6
    h(k) = subplot(6,1,k);
end
x = 0:.1:2*pi;
for k=1:6
    y = sin(x*100*rand)*rand;
    % Explicitly set current axes as a figure property
    set(fHandle,'CurrentAxes',h(k)); stem(x,y); axis([0 6.28 -1 1])
end
```

Both techniques described in this section give better performance than calling the `axes` function to control the current axes.

Performance of Bit-Mapped Images

Images can be defined with lower precision (than `double`) values to reduce the total amount of data required. MATLAB performs many operations on nondouble data types you can use smaller image data without converting the data to type `double`. See “Working with 8-Bit and 16-Bit Images” on page 6-10 for more information.

Direct Color Mapping

Where possible, use indexed images because this type of image can apply direct mapping of pixel values to colormap values (`CDataMapping` set to `direct`). With direct mapping, MATLAB does not need to scale the data and then map it to values in the colormap.

See the `CDataMapping` image property for more information.

Use Truecolor for Smaller Images

The use of truecolor (red, green, and blue values) eliminates the need for color mapping. However, with very large images, the data can be quite large and thereby slow performance.

Direct Mapping of Transparency Values

If you are using an alphamap of transparency values, prescale the alpha data so you can use the most efficient alpha data mapping (`AlphaDataMapping` set to `none`)

See the `AlphaDataMapping` image property for more information.

Performance of Patch Objects

You can improve the speed with which MATLAB renders patch objects using the following techniques.

Define Patch Faces as Triangles

If you are using patch objects that have many vertices per patch face, you should modify your data so that each face has only three vertices, but still looks like your original object. This eliminates the tessellation step from the rendering process.

Use Data Thinning

It is sometimes possible (or even desirable) to reduce the number of vertices in a patch and still produce the desired results.

See the `reducepatch` and `reducevolume` functions for more information.

Direct Color Mapping

Where possible, use direct color mapping for coloring patches. (`CDataMapping` set to `direct`). With direct mapping, MATLAB does not need to scale the data and then map it to values in the colormap.

See the `CDataMapping` patch property for more information.

Use Truecolor for Smaller Patches

The use of truecolor (red, green, and blue values) eliminates the need for color mapping. However, with very large patches, the data can be quite large and thereby slow performance.

Direct Mapping of Transparency Values

If you are using an alphamap of transparency values, prescale the alpha data so you can use the most efficient alpha data mapping (`AlphaDataMapping` set to `none`)

See the `AlphaDataMapping` patch property for more information.

Performance of Surface Objects

You can improve the speed with which MATLAB renders surface objects using the following techniques.

Direct Color Mapping

Where possible, use direct color mapping for coloring surfaces. (`CDataMapping` set to `direct`). With direct mapping, MATLAB does not need to scale the data and then map it to values in the colormap.

See the `CDataMapping` surface property for more information.

Use Truecolor for Smaller Surfaces

The use of truecolor (red, green, and blue values) eliminates the need for color mapping. However, with very large surfaces, the data can be quite large and thereby slow performance.

Mapping of Transparency Values

If you are using an alphamap of transparency values, prescale the alpha data so you can use the most efficient alpha data mapping (`AlphaDataMapping` set to `none`)

See the `AlphaDataMapping` surface property for more information.

Use Texture-Mapped Face Color

If you are using surface objects in an animation or want to be able to pan and rotate them quickly, you can achieve better rendering performance with large surfaces by setting `EdgeColor` to `none` and `FaceColor` to `texture`.

This technique is particularly useful if you want a high resolution surface without creating an objects whose data is large and therefore, very slow to transform. For example,

```
h1 = surf(peaks(1000));
shading interp
cd1 = get(h1,'CData');
surf(peaks(24),'FaceColor','Texture','EdgeColor','none',...
 'CData',cd1)
```

9

Figure Properties

Figure Objects

Figure graphics objects are the windows in which MATLAB displays graphics. Figure properties enable you to control many aspects of these windows, such as their size and position on the screen, the coloring of graphics objects displayed within them, and the scaling of printed pictures.

This section discusses some of the features that are implemented through figure properties and provides examples of how to use these features.

See Figure Properties for a description of each property

Related Information About Figures

For more information about figures, see the following links:

- “Graphics Windows — the Figure” on page 8-6
- “Preparing Figures and Axes for Graphics” on page 8-71
- “Protecting Figures and Axes” on page 8-77
- “The Figure Close Request Function” on page 8-80
- “Using Panel Containers in Figures — Uipanels” on page 8-89
- “Programming the Resize Functions” on page 8-92
- “Figure Callbacks” on page 8-107
- “Introduction” on page 10-36
- “Displaying Multiple Plots per Figure” on page 4-2

Docking Figures in the Desktop

In this section...

Introduction

You can dock figures in the MATLAB desktop by clicking the **dock** button, , which appears on the right end of the menu bar. Once docked, figures are placed in a figure group container, which you can also dock and undock.

You can select from a variety of arrangements of the figures in the container. The following picture shows how to select various figure arrangements. Once docked, the figure container displays the toolbar and menubar of the figure with focus.

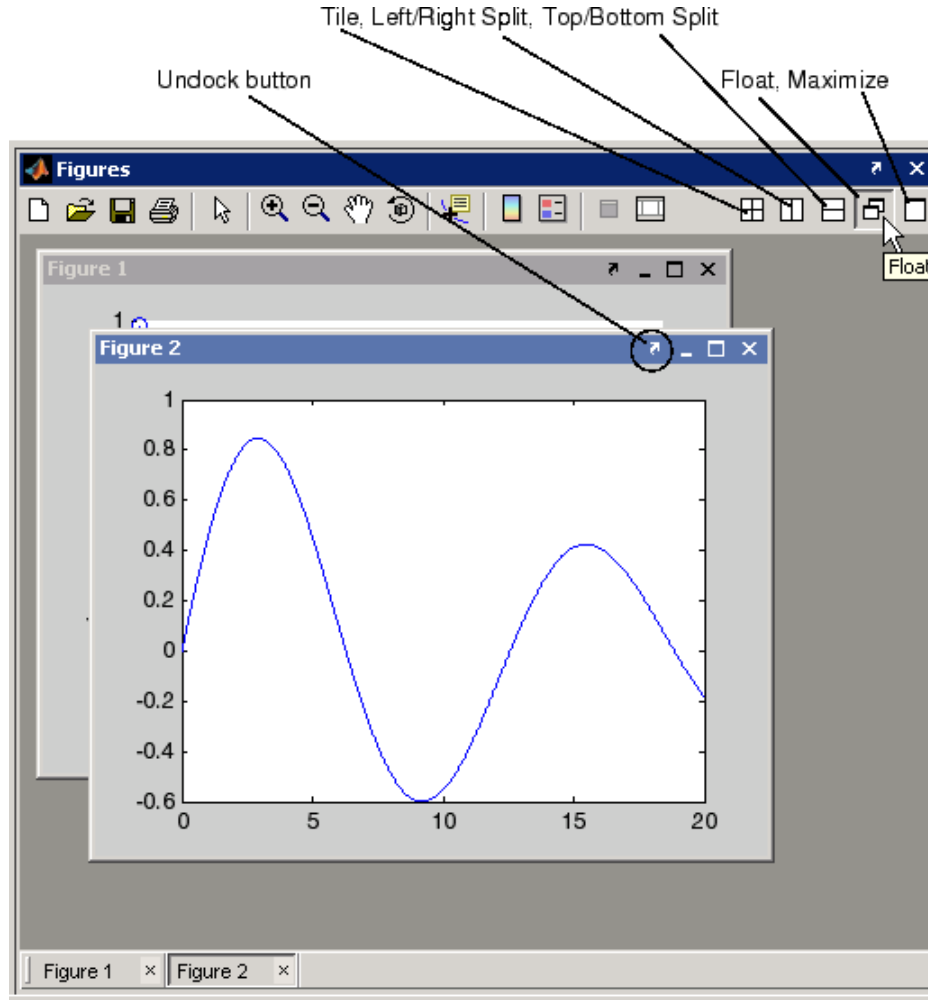


Figure Properties That Affect Docking

There are two figure properties that are related to figure docking — `DockControls` and `WindowStyle`.

DockControls

The `DockControls` property controls the display of the controls used to dock figures. Setting `DockControls` to `off` removes the **dock** button from the menubar and disables docking from the figure **Desktop** menu.

WindowStyle

When you set the `WindowStyle property` to `docked`, MATLAB docks the figure in the desktop within a figure group container.

If `WindowStyle` is set to `docked`,

- MATLAB automatically sets `DockControls` to `on`.
- You cannot set the `DockControls` property to `off`.
- You cannot set the figure `Position` property.

Docking Figures Automatically

If you want MATLAB to always dock figures, set the default value of the `WindowStyle` property to `docked`. The following statement,

```
set(0,'DefaultFigureWindowStyle','docked')
```

creates a default value for the `WindowStyle` property on the root level. Issuing this statement on the command line sets the `WindowStyle` of all figures for the duration of your MATLAB session (unless you change the value).

Place this statement in your `startup.m` file to make MATLAB always dock figures. See `startup` for more information on `startup.m`.

Creating a Nondockable Figure

In cases where you do not want users to be able to dock figures (e.g., figures used for GUIs), you should set figure properties as follows:

- `DockControls` to `off`
- `WindowStyle` to `normal` or `modal`
- `HandleVisibility` to `off` or `callback`

Positioning Figures

In this section...

Introduction

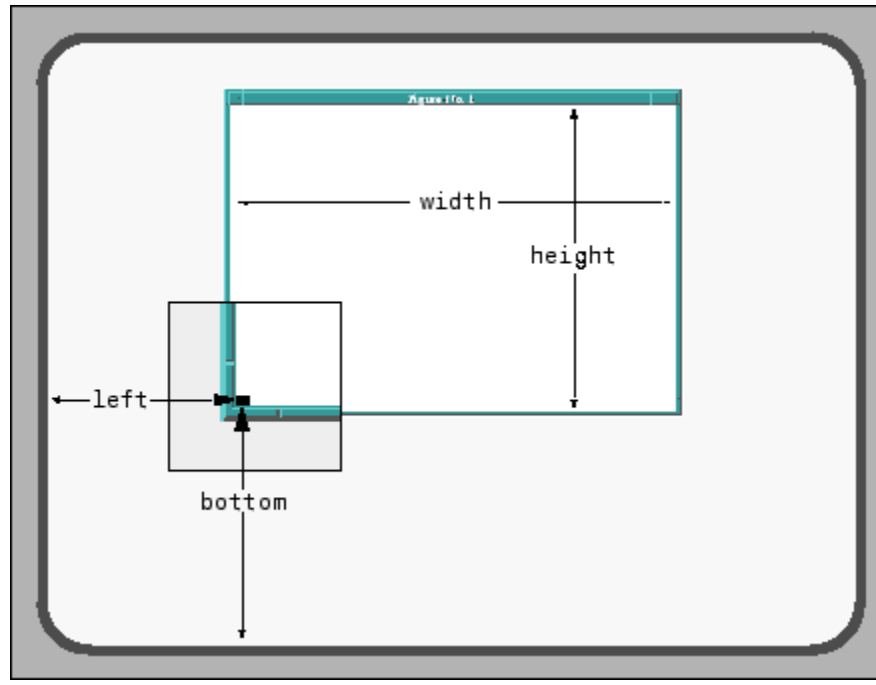
The figure `Position` property controls the size and location of the figure window on the root screen. At startup, MATLAB determines the size of your computer screen and defines a default value for `Position`. This default creates figures about one-quarter of the screen's size and places them centered left to right and in the top half of the screen.

The Position Vector

MATLAB defines the figure `Position` property as a vector.

```
[left bottom width height]
```

`left` and `bottom` define the position of the first addressable pixel in the lower left corner of the window, specified with respect to the lower left corner of the screen. `width` and `height` define the size of the interior of the window (i.e., exclusive of the window border).



MATLAB does not measure the window border when placing the figure; the `Position` property defines only the internal active area of the figure window.

Because figures are windows under the control of your computer's windowing system, you can move and resize figures as you would any other windows. MATLAB automatically updates the `Position` property to the new values.

Figure Position for Docked Figures

When a figure is docked in the MATLAB desktop, the `Position` property is defined with respect to the figure group container within the desktop. See "Docking Figures in the Desktop" on page 9-3 for more information.

Units

The figure's `Units` property determines the units of the values used to specify the position on the screen. Possible values for the `Units` property are

```
set(gcf,'Units')
[ inches | centimeters | normalized | points | {pixels} |
characters]
```

with `pixels` being the default. These choices allow you to specify the figure size and location in absolute units (such as inches) if you want the window always to be a certain size, or in units relative to the screen size (such as pixels).

Characters are units that enable you to define the location and size of the figure in units that are based on the size of the default system font. See "Example — Using Figure Panels" on page 8-90 for an example that uses character units.

Determining Screen Size

Whatever units you use, it is important to know the extent of the screen in those units. You can obtain this information from the root `CDataMapping` property. For example,

```
get(0,'ScreenSize')
ans =
    1 1 1152 900
```

In this case, the screen is 1152 by 900 pixels. MATLAB returns the `ScreenSize` in the units determined by the root `Units` property. For example,

```
set(0,'Units','normalized')
```

normalizes the values returned by `ScreenSize`.

```
get(0,'ScreenSize')
ans =
    0 0 1 1
```

Defining the figure `Position` in terms of the `ScreenSize` in normalized units makes the specification independent of variations in screen size. This is useful if you are writing an M-file that is to be used on different computer systems.

Example — Specifying Figure Position

Suppose you want to define two figure windows that occupy the upper third of the computer screen (e.g., one for uicontrols and the other to display data). To position the windows precisely, you must consider the window borders when calculating the size and offsets to specify for the `Position` properties.

1 The figure `Position` property does not include the window borders, so this example uses a width of 5 pixels on the sides and bottom and 30 pixels on the top.

```
bdwidth = 5;
topbdwidth = 30;
```

2 Ensure root units are pixels and get the size of the screen:

```
set(0,'Units','pixels')
scnsize = get(0,'ScreenSize');
```

3 Define the size and location of the figures:

```
pos1  = [bdwidth,...
 2/3*scnsize(4) + bdwidth,...
 scnsize(3)/2 - 2*bdwidth,...
 scnsize(4)/3 - (topbdwidth + bdwidth)];
pos2 = [pos1(1) + scnsize(3)/2,...
 pos1(2),...
 pos1(3),...
 pos1(4)];
```

4 Create the figures:

```
figure('Position',pos1)
figure('Position',pos2)
```

The two figures now occupy the top third of the screen.

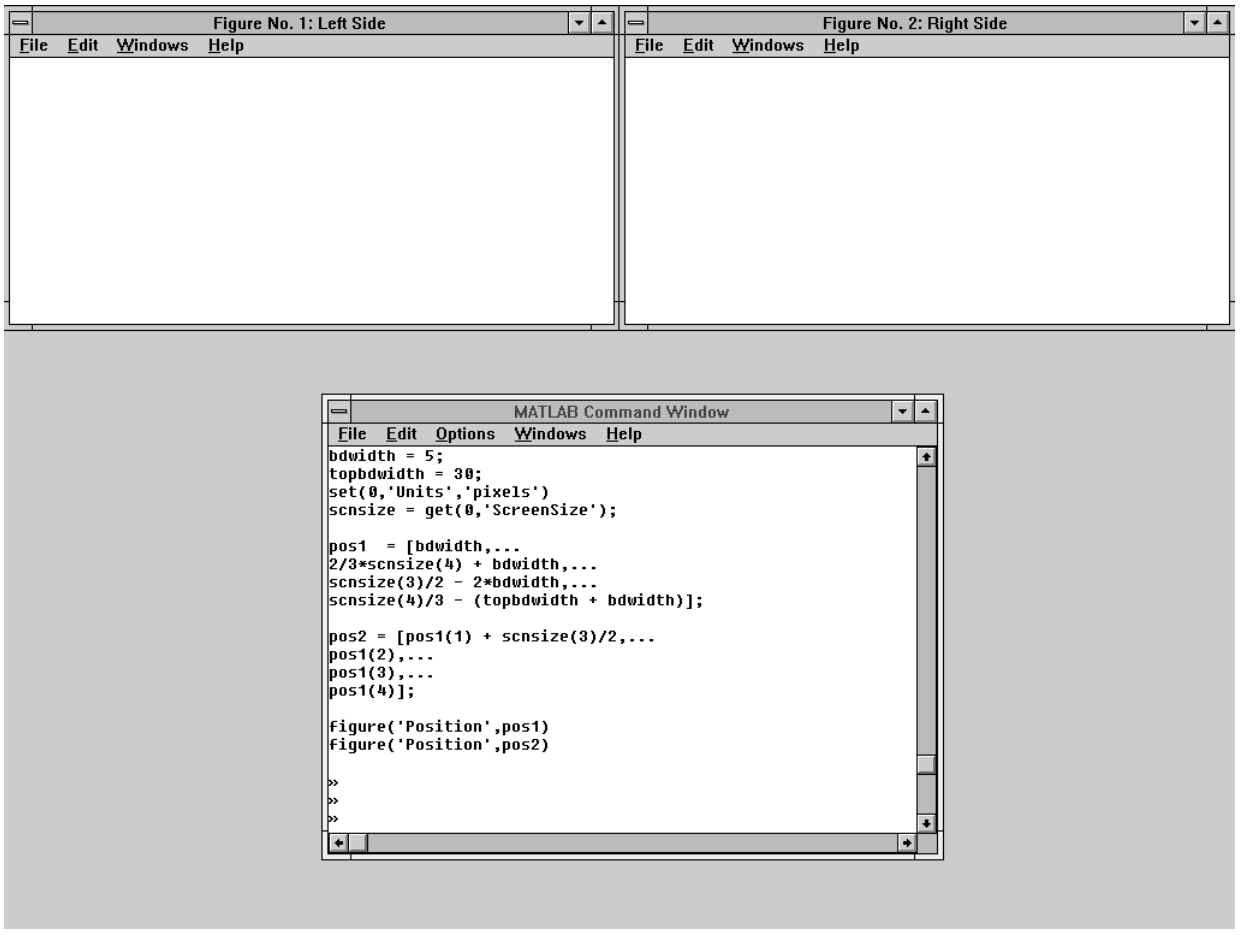


Figure No. 1: Left Side
File Edit Windows Help
Figure No. 2: Right Side
File Edit Windows Help
MATLAB Command Window
File Edit Options Windows Help
bdwidth = 5;
topbdwidth = 30;
set(0,'Units','pixels')
scnsize = get(0,'ScreenSize');
pos1 = [bdwidth,...
2/3*scnsize(4) + bdwidth,...
scnsize(3)/2 - 2*bdwidth,...
scnsize(4)/3 - (topbdwidth + bdwidth)];
pos2 = [pos1(1) + scnsize(3)/2,...
pos1(2),...
pos1(3),...
pos1(4)];
figure('Position',pos1)
figure('Position',pos2)
»
»
»

Figure Colormaps — The Colormap Property

In this section...

Introduction

MATLAB defines a colormap as a three-column array. Each row of the array defines a particular color by giving three values in the range [0...1]. These values specify the RGB values; the intensity of the red, green, and blue video components.

Colormaps enable you to control how MATLAB maps data values to colors in surfaces, patches, images, and plotting functions that are based on these objects. See the following sections for more information.

- "Coloring Mesh and Surface Plots" in 3-D Visualization
- "Specifying Patch Coloring" in 3-D Visualization
- "The Image Object and Its Properties" on page 6-27

Specifying the Figure Colormap

The figure `Colormap` property contains the colormap array. You can specify the figure colormap by setting this property to an *m*-by-3 array, where *m* is the number of colors in the colormap.

For example, the following statement creates a figure with a colormap having 128 random colors.

```
figure('Colormap',rand(128,3));
```

The `colormap` function is an easy way to specify the colormap. MATLAB also provides a number of functions that generate colormaps. For example,

```
colormap(hsv(96))
```

sets the colormap of the current figure to a 96 element version of the `hsv` colormap. See the `colormap` reference page for a list of predefined colormaps. Note that the default colormap is `jet(64)`.

Selecting Drawing Methods

In this section...

Double Buffering

Overview

Set `DoubleBuffer` to `on` when you are animating lines rendered in `painters` with `EraseMode` set to `normal`.

More Details

Double buffering is the process of drawing into an offscreen pixel buffer and then blitting the buffer contents to the screen once the drawing is complete (instead of drawing directly to the screen, where the process of drawing is visible as it progresses). Double buffering generally produces flash-free rendering for simple animations (such as those involving lines, as opposed to objects containing large numbers of polygons).

The figure `DoubleBuffer` property accepts the values `on` and `off`, with `on` being the default. You can select double buffering only when the figure `Renderer` property is set to `painters`. `zbuffer` and `opengl` always use double buffering and ignore this property.

Use double buffering with the animated object's `EraseMode` property set to `normal`.

Selecting a Renderer

Overview

MATLAB automatically selects the best renderer based on the complexity of the graphics objects and the options available on your system.

More Details

A renderer is the software that processes graphics data (such as vertex coordinates) into a form that MATLAB can use to draw into the figure. MATLAB supports three renderers:

- Painters
- Z-buffer
- OpenGL

Painters

Painters method is faster when the figure contains only simple or small graphics. It cannot be used with lighting.

Z-Buffer

Z-buffering is the process of determining how to render each pixel by drawing only the front-most object, as opposed to drawing all objects back to front, redrawing objects that obscure those behind. The pixel data is buffered and then blitted to the screen all at once.

Z-buffering is generally faster for more complex graphics, but can be slower for very simple graphics. You can set the `Renderer` property to whatever produces the fastest drawing (either `zbuffer` or `painters`), or let MATLAB decide which method to use by setting the `RendererMode` property to `auto` (the default).

Printing from Z-Buffer. You can select the resolution of the PostScript file produced by the `print` command using the `-r` option. By default, MATLAB prints Z-buffered figures at a medium resolution of 150 dpi (the default with `Renderer` set to `painters` is 864 dpi).

The size of the file generated from a Z-buffer figure does not depend on its contents, just the size of the figure. To decrease the file size, make the `PaperPosition` property smaller before printing (or set `PaperPositionMode` to `auto` and resize the figure window).

OpenGL

OpenGL is available on many computer systems. It is generally faster than either painters or Z-buffer and in some cases enables MATLAB to use the system's graphics hardware (which results in significant speed increase). See the figure `Renderer` property for more information.

Limitations of OpenGL. OpenGL has two limitations when compared to painters and Z-buffer:

- OpenGL does not interpolate colors within the figure colormap; all color interpolation is performed through the RGB color cube, which can produce unexpected results.
- OpenGL does not support Phong lighting.

Specifying the Figure Pointer

In this section...

Predefined Figure Pointer Symbols

MATLAB indicates the position of the pointer (cursor) within the figure window using a graphical symbol. You can select a pointer from 15 predefined symbols (see table below) or you can define your own symbol. By convention, each of the predefined symbols has a purpose associated with it (although MATLAB enforces no rules for the use of any symbols).

You specify the pointer symbol by setting the value of the figure `Pointer` property. For example, this statement sets the pointer in the current figure (`gcf`) to an arrow.

```
set(gcf,'Pointer','arrow')
```

The following table shows the predefined symbols, the associated specifier, and describes typical use.

Purpose	Specifier	Typical Symbol
Use for editing text	`ibeam`	
Locate a point on a graphics object	`crosshair`	
Select a point anywhere in the figure	`arrow`	
Indicate the system is busy	`watch`	
Resize an object from the top-left corner	`topl`	
Resize an object from the top-right corner	`topr`	

Purpose	Specifier	Typical Symbol
Resize an object from the bottom-left corner	`botl`	
Resize an object from the bottom-right corner	`botr`	
View the actual hot spot	`circle`	
Locate a point	`cross`	
Use as popular symbol	`fleur`	
Resize an object from the left side	`left`	
Resize an object from the right side	`right`	
Resize an object from the top	`top`	
Resize an object from the bottom	`bottom`	
Align a point with other objects on the display	`fullcross`	
See the next section for information on defining your own pointer shape	`custom`	

Defining Custom Pointers

When you set the `Pointer` property to `custom`, MATLAB displays the pointer you define using the `PointerShapeCData` and the `PointerShapeHotSpot` properties. Custom pointers are 16-by-16 pixels, where each pixel can be either black, white, or transparent.

Specify the pointer by creating a 16-by-16 matrix containing elements that are

- 1's where you want the pixel black
- 2's where you want the pixel white
- `NaN`s where you want the pixel transparent

Assign the matrix to the figure `PointerShapeCData` property. MATLAB displays the defined pointer whenever the pointer is in the figure window.

The `PointerShapeHotSpot` property specifies the pixel that indicates the pointer location. MATLAB then stores this location in the root `PointerLocation` property. Set the `PointerShapeHotSpot` property to a two-element vector specifying the row and column indices in the `PointerShapeCData` matrix that correspond to the pixel specifying the location. The default value for this property is [1 1], which corresponds to the upper left corner of the pointer.

Example — Two Custom Pointers

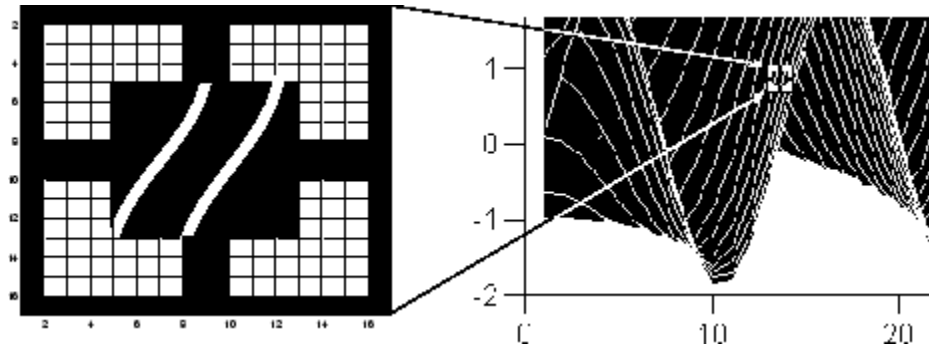
One way to create a custom pointer is to assign values to a 16-by-16 matrix by hand, as illustrated in the following example.

First, initialize the matrix, setting all values to 2. Create a black border 1 pixel wide. Add alignment marks.

```
P = ones(16)+1;
P(1,:) = 1; P(16,:) = 1;
P(:,1) = 1; P(:,16) = 1;
P(1:4,8:9) = 1; P(13:16,8:9) = 1;
P(8:9,1:4) = 1; P(8:9,13:16) = 1;
P(5:12,5:12) = NaN;     % Create a transparent region in the center
set(gcf,'Pointer','custom','PointerShapeCData',P,...
        'PointerShapeHotSpot',[9 9])
```

The last statement sets the `Pointer` property to `custom`, assigns the matrix to the `PointerShapeCData` property, and selects element (9,9) as the “hot spot.”

MATLAB now uses the custom pointer within the figure window.

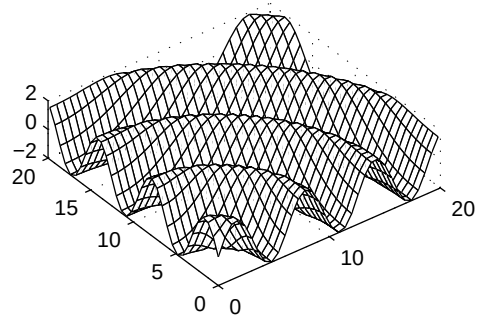


Creating Pointers from Functions. You can use a mathematical function to define the `PointerShapeCData` matrix. For example, evaluating the function

$$2(\sin(\sqrt{x^2 + y^2}))$$

```
g = 0:.2:20;
[X,Y] = meshgrid(g);
Z = 2*sin(sqrt(X.^2 + Y.^2));
mesh(Z);
```

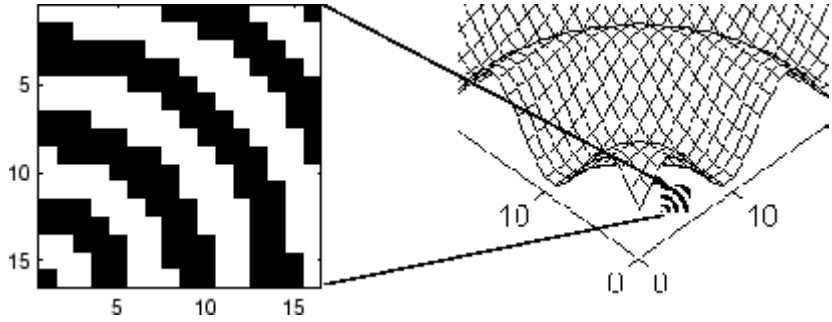
produces an interesting surface.



Use the values of `Z` to create a pointer sampling fewer points so that `Z` is a 16-by-16 matrix.

```
g = linspace(0,20,16);
[X,Y] = meshgrid(g);
Z = 2*sin(sqrt(X.^2 + Y.^2));
set(gcf,'Pointer','custom',...
        'PointerShapeCData',flipud((Z>0) + 1))
```

The statement `flipud((Z>0) + 1)` sets all values in `Z` that are greater than 0 to 2 (in MATLAB, true + 1 = 2), less than 0 to 1 (false + 1 = 1) and then flips the data around so that element (1,1) is the upper left corner.



Axes Properties

Axes Objects — Defining Coordinate Systems for Graphs

MATLAB uses graphics objects to create visual representations of data. For example, a two-dimensional array of numbers can be represented as lines connecting the data points defined by each column, as a surface constructed from a grid of rectangles whose vertices are defined by each element of the array, as a contour graph where equal values in the array are connected by lines, and so on.

In all these cases, there must be a frame of reference that defines where to place each data point on the graph. This frame of reference is the coordinate system defined by the axes. Axes orient and scale graphs to produce the view of the data that you see on screen.

MATLAB creates axes to define the coordinate system of each graph. Axes are always contained within a figure object and themselves contain the graphics objects that make up the graph.

Axes properties control many aspects of how MATLAB displays graphical information. This section discusses some of the features that are implemented through axes properties and provides examples of how to uses these features.

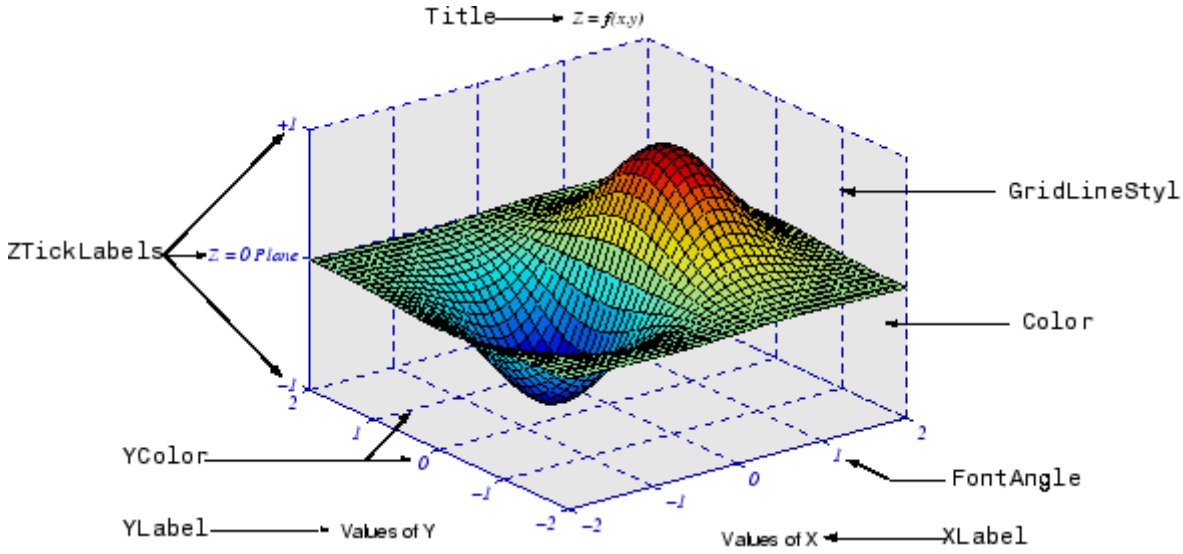
The Axes Properties list all axes properties and provide an overview of the characteristics that are affected by each property.

Labeling and Appearance Properties

In this section...

Introduction

MATLAB provides a number of properties for labeling and controlling the appearance of axes. For example, this surface plot shows some of the labeling possibilities and indicates the controlling property.



Creating Axes with Specific Characteristics

To create this axes, specify values for the indicated properties.

```
h = axes('Color',[.9 .9 .9],...
        'GridLineStyle','--',...
        'ZTickLabels','-1|Z = 0 Plane|+1',...
```

```
'FontName','times',...
'FontAngle','italic',...
'FontSize',14,...
'XColor',[0 0 .7],...
'YColor',[0 0 .7],...
'ZColor',[0 0 .7]);
```

Axis Labels

The individual axis labels are text objects whose handles are normally hidden from the command line (their `HandleVisibility` property is set to `callback`). You can use the `xlabel`, `ylabel`, `zlabel`, and `title` functions to create axis labels. However, these functions affect only the current axes. If you are labeling axes other than the current axes by referencing the axes handle, then you must obtain the text object handle from the corresponding axes property.

Getting the Text Object Handle

For example,

```
get(axes_handle,'XLabel')
```

returns the handle of the text object used as the *x*-axis label. Obtaining the text handle from the axes is useful in M-files and MATLAB-based applications where you cannot be sure the intended target is the current axes.

The following statements define the *x*- and *y*-axis labels and title for the axes above.

```
set(get(axes_handle,'XLabel'),'String','Values of X')
set(get(axes_handle,'YLabel'),'String','Values of Y')
set(get(axes_handle,'Title'),'String','\fontname{times}\itZ =
f(x,y)')
```

Because the labels are text, you must specify a value for the `String` property, which is initially set to the empty string (i.e., there are no labels).

MATLAB overrides many of the other text properties to control positioning and orientation of these labels. However, you can set the `Color`, `FontAngle`, `FontName`, `FontSize`, `FontWeight`, and `String` properties.

Specifying Axis Label Fonts

Note that both axes objects and text objects have font specification properties. The call to the `axes` function on the previous page set values for the `FontName`, `FontAngle`, and `FontSize` properties.

If you want to specify the font for the labels and title, set the font property values when defining their `String` property. For example, the *x*-axis label statement would be

```
set(get(h,'XLabel'),'String','Values of X',...
                    'FontName','times',...
                    'FontAngle','italic',...
                    'FontSize',14)
```

Bitmapped Vs. Truetype Fonts — Text Does Not Rotate

Bitmapped fonts (e.g., Courier) cannot be rotated on the display. Therefore, when you specify a bitmapped font with the `FontName` property, this text might not be rotated correctly, for example, when used as the *y*-axis label.

To avoid problems with bitmapped fonts, use TrueType fonts. For example, you might have a TrueType font named Courier New that you can use instead of Courier. See your system documentation for information on which fonts are installed on your system.

Positioning Axes

In this section...

Introduction

The axes `Position` property controls the size and location of an axes within a figure. The default axes has the same aspect ratio (ratio of width to height) as the default figure and fills most of the figure, leaving a border around the edges. However, you can define the axes position as any rectangle and place it wherever you want within a figure.

The Position Vector

MATLAB defines the axes `Position` property as a vector.

```
[left bottom width height]
```

`left` and `bottom` define a point in the figure that locates the lower left corner of the axes rectangle. `width` and `height` specify the dimensions of the axes rectangle. Viewing the axes in 2-D (azimuth = 0°, elevation = 90°) orients the *x*-axis horizontally and the *y*-axis vertically. From this angle, the plot box (the area used for plotting, exclusive of the axis labels) coincides with the axes rectangle.

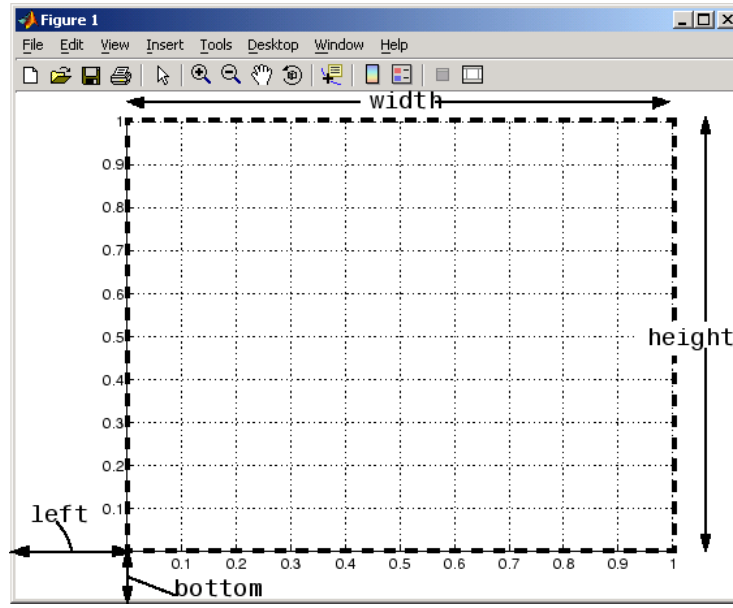


Figure 1
File Edit View Insert Tools Desktop Window Help
width
height
left
bottom
0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1

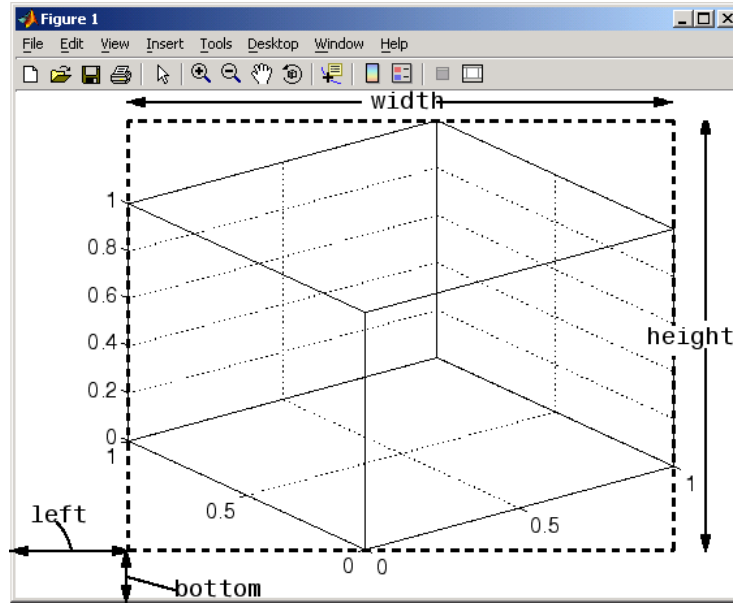


Figure 1
File Edit View Insert Tools Desktop Window Help
width
height
left
bottom
1
0.8
0.6
0.4
0.2
0
0.5
0.5
1
0 0

By default, MATLAB draws the plot box to fill the axes rectangle, regardless of its shape. However, axes properties enable control over the shape and scaling of the plot box.

Position Units

The axes `Units` property determines the units of measurement for the `Position` property. Possible values for this property are

```
set(gca,'Units')
[ inches | centimeters | {normalized} | points | pixels ]
```

with `normalized` being the default. Normalized units map the lower left corner of the figure to the point (0,0) and the upper right corner to (1.0,1.0), regardless of the size of the figure. Normalized units cause axes to resize automatically whenever you resize the figure. All other units are absolute measurements that remained fixed as you resize the figure.

Automatic Axes Resize

In this section...

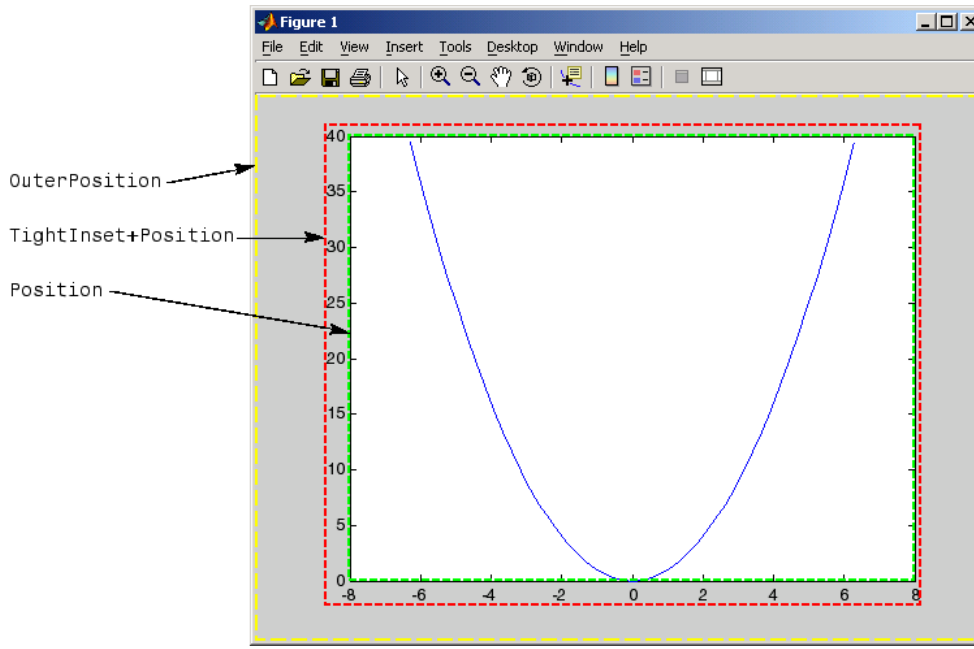
Properties Controlling Axes Size

When you create a graph, MATLAB automatically creates an axes to display the graph. The axes is sized to fit in the figure and automatically resizes as you resize the figure. Note, however, that MATLAB applies the automatic resize behavior only when the axes `Units` property is set to `normalized` (the default).

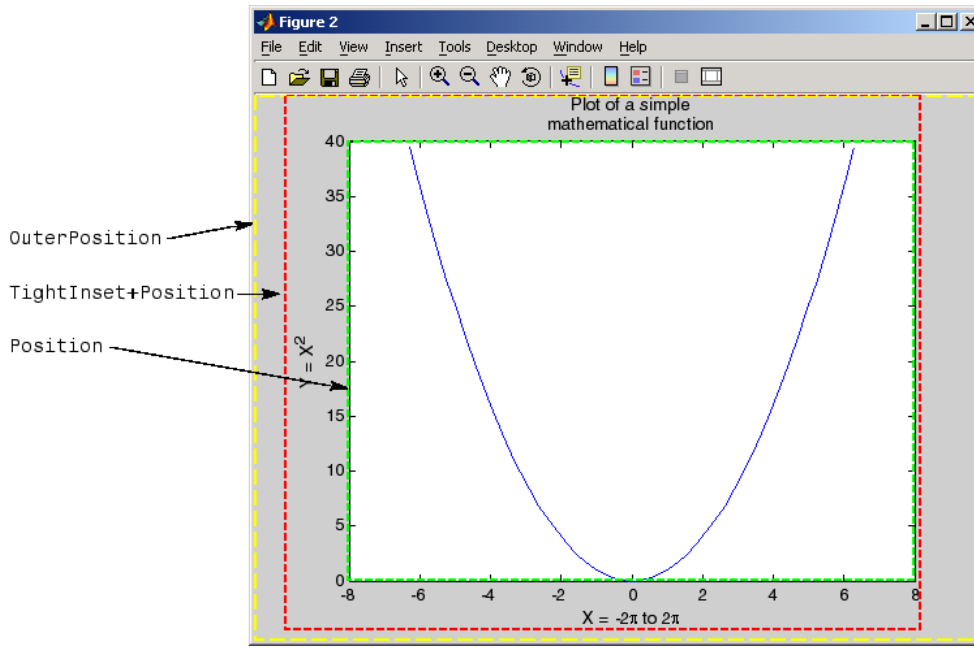
You can control the resize behavior of the axes using the following axes properties.

- `OuterPosition` — The boundary of the axes including the axis labels, title, and a margin. For figures with only one axes, this is the interior of the figure.
- `Position` — The boundary of the axes, excluding the tick marks and labels, title, and axis labels.
- `ActivePositionProperty` — Specifies whether to use the `OuterPosition` or the `Position` property as the size to preserve when resizing the figure containing the axes.
- `TightInset` — The margins added to the width and height of the `Position` property to include text labels, title, and axis labels.
- `Units` — Keep this property set to `normalized` to enable automatic axes resizing.

The following graph shows the areas defined by the `OuterPosition`, `TightInset + Position`, and `Position` properties.



When you add axis labels and a title, the `TightInset` changes to accommodate the additional text, as shown in the following graph.

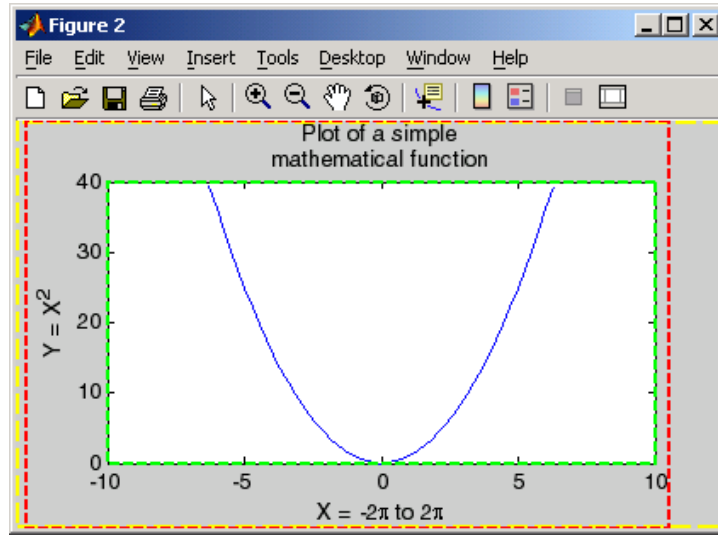


Now the size of the rectangle defined by the `TightInset + Position` properties includes all graph text. The `Position` and `OuterPosition` properties remain unchanged.

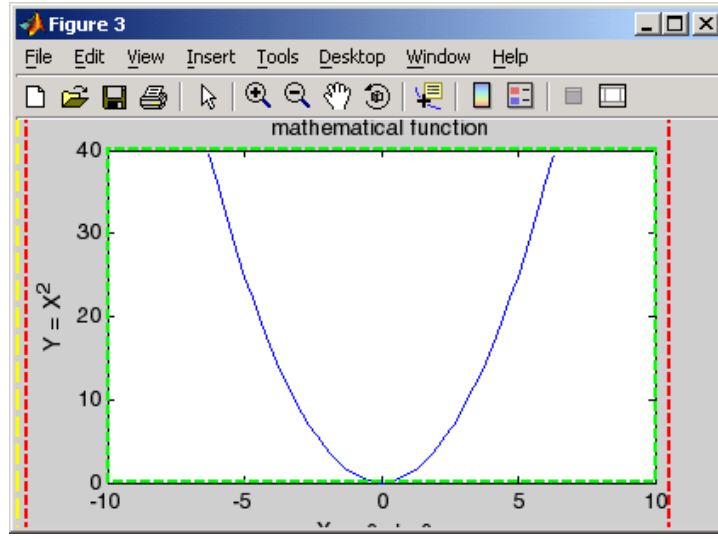
Using OuterPosition as the ActivePositionProperty

As you resize the figure, MATLAB maintains the area defined by the `TightInset + Position` so the test is not cut off. Compare the next two graphs, which have both been resized to the same figure size.

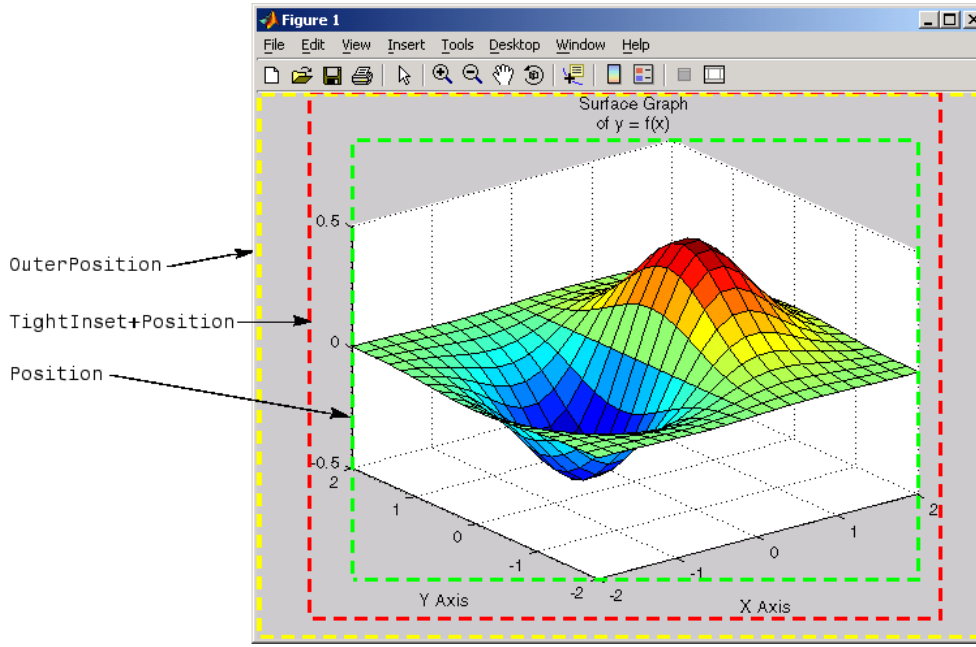
ActivePositionProperty = OuterPosition



ActivePositionProperty = Position



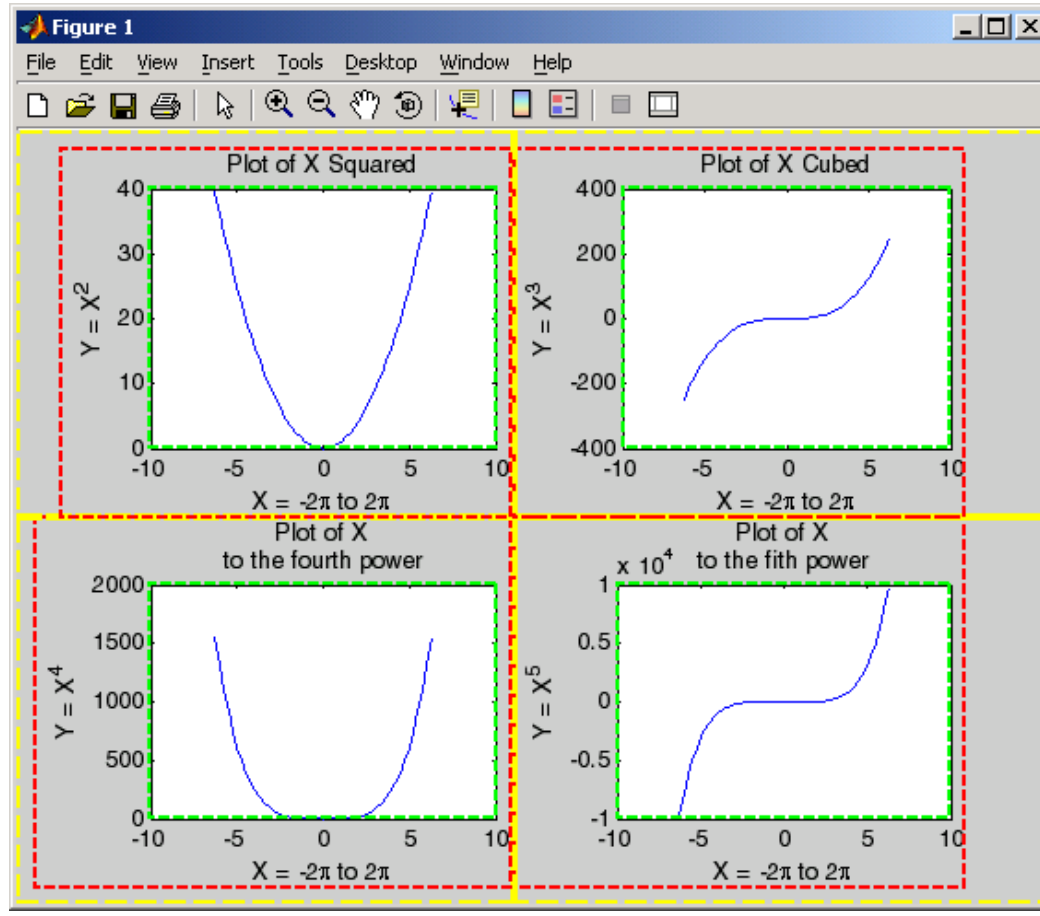
The following picture shows how these properties apply to 3-D graphs.



Axes Resizing in Subplots

Using the `OuterPosition` property as the `ActivePositionProperty` is an effective way to prevent titles and labels from being overwritten when there are multiple axes in a figure.

The following picture illustrates how MATLAB resizes the axes to accommodate the multiline titles on the lower two axes.



The default 3-D view is azimuth = -37.5°, elevation = 30°.

Multiple Axes per Figure

In this section...

Introduction

The `subplot` function creates multiple axes in one figure by computing values for `Position` that produce the specified number of axes.

The `subplot` function is useful for laying out a number of graphs equally spaced in the figure. However, overlapping axes can create some other useful effects. The following sections provide examples.

Placing Text Outside the Axes

MATLAB always displays text objects within an axes. If you want to create a graph and provide a description of the information alongside the graph, you must create another axes to position the text. If you create an axes that is the same size as the figure and then create a smaller axes to draw the graph, you can then display text anywhere independently of the graph.

For example, define two axes.

```
h = axes('Position',[0 0 1 1],'Visible','off');
axes('Position',[.25 .1 .7 .8])
```

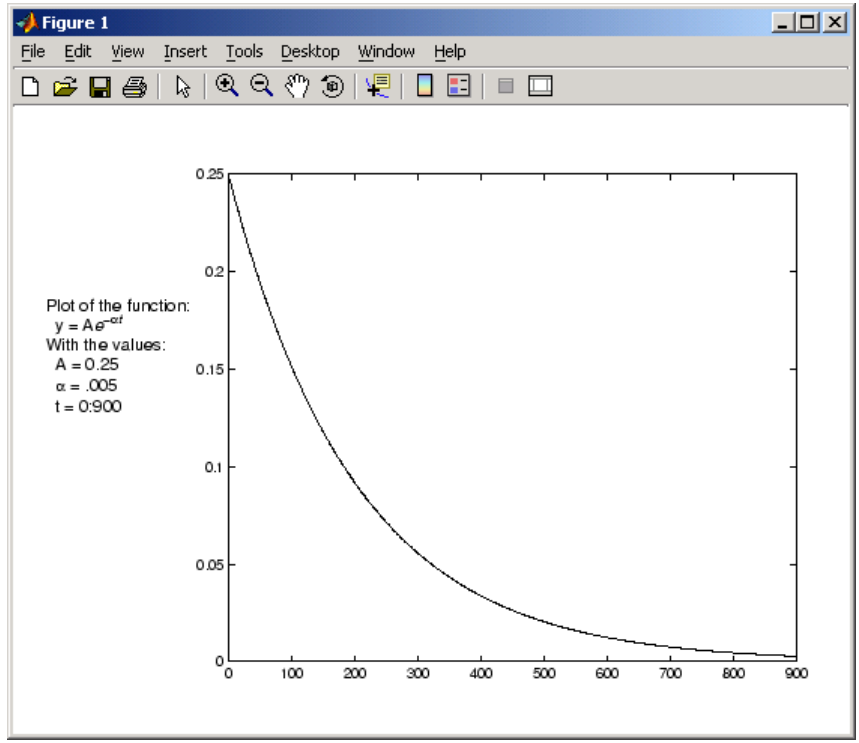
Because the axes units are normalized to the figure, specifying the `Position` as `[0 0 1 1]` creates an axes that encompasses the entire window.

Now plot some data in the current axes. The last axes created is the current axes, so MATLAB directs graphics output there.

```
t = 0:900;
plot(t,0.25*exp(-0.005*t))
```

Define the text and display it in the full-window axes.

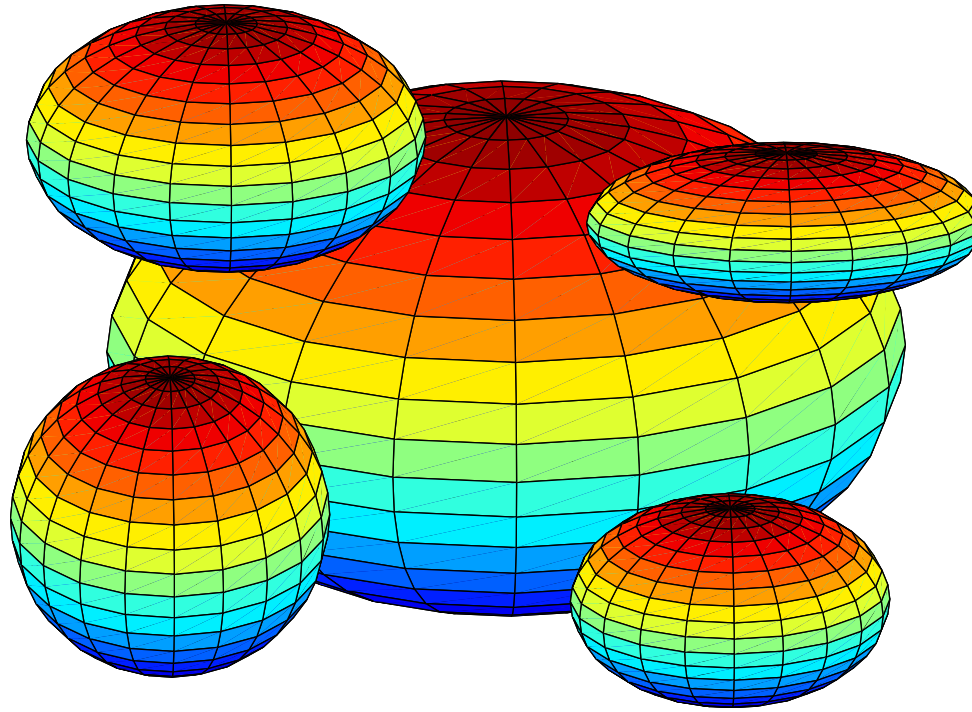
```
str(1) = {'Plot of the function:'};
str(2) = {' y = A{\ite}^{-\alpha{\itt}}'};
str(3) = {'With the values:'};
str(3) = {' A = 0.25'};
str(4) = {' \alpha = .005'};
str(5) = {' t = 0:900'};
set(gcf,'CurrentAxes',h)
text(.025,.6,str,'FontSize',12)
```



Multiple Axes for Different Scaling

You can create multiple axes to display graphics objects with different scaling without changing the data that defines these objects (which would be required to display them in a single axes).

```
h(1) = axes('Position',[0 0 1 1]);
sphere
h(2) = axes('Position',[0 0 .4 .6]);
sphere
h(3) = axes('Position',[0 .5 .5 .5]);
sphere
h(4) = axes('Position',[.5 0 .4 .4]);
sphere
h(5) = axes('Position',[.5 .5 .5 .3]);
sphere
set(h,'Visible','off')
```



Each sphere is defined by the same data. However, because the parent axes occupy regions of different size and location, the spheres appear to be different sizes and shapes.

Individual Axis Control

In this section...

Properties Controlling Axis Limits

MATLAB automatically determines axis limits, tick mark placement, and tick mark labels whenever you create a graph. However, you can specify these values manually by setting the appropriate property.

When you specify a value for a property controlled by a mode (e.g., the `XLim` property has an associated `XLimMode` property), MATLAB sets the mode to manual, enabling you to override automatic specification. Because the default values for these mode properties are automatic, calling high-level functions such as `plot` or `surf` resets these modes to `auto`.

This section discusses the following properties.

Property	Purpose
`XLim,YLim,ZLim`	Sets the axis range

`XLimMode,`
`YLimMode,`
`ZLimMode`
	Specifies whether axis limits are determined automatically by MATLAB or specified manually by the user

`XTick,`
`YTick,`
`ZTick`
	Sets the location of the tick marks along the axis

Property	Purpose
`XTickMode`, `YTickMode`, `ZTickMode`	Specifies whether tick mark locations are determined automatically by MATLAB or specified manually by the user
`XTickLabel`, `YTickLabel`, `ZTickLabel`	Specifies the labels for the axis tick marks
`XTickLabelMode` `YTickLabelMode` `ZTickLabelMode`	Specifies whether tick mark labels are determined automatically by MATLAB or specified manually by the user
`XDir,YDir,ZDir`	Sets the direction of increasing axis values

Setting Axis Limits

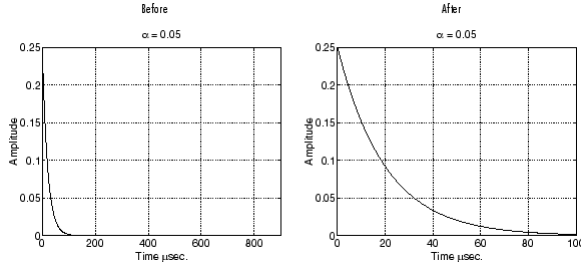
MATLAB determines the limits automatically for each axis based on the range of the data. You can override the selected limits by specifying the `XLim`, `YLim`, or `ZLim` property. For example, consider a plot of the function $Ae^{-\alpha t}$ evaluated with A = 0.25, α = 0.05, and t = 0 to 900.

```
t = 0:900;
plot(t,0.25*exp(-0.05*t))
```

The plot on the left shows the results. MATLAB selects axis limits that encompass the range of data in both x and y. However, because the plot contains little information beyond t = 100, changing the x-axis limits improves the usefulness of the plot. If the handle of the axes is `axes_handle`, then the following statement,

```
set(axes_handle,'XLim',[0 100])
```

creates the plot on the right.



You can use the `axis` command to set limits on the current axes only.

Semiautomatic Limits

You can specify either the minimum or maximum value for an axis limit and allow the other limit to autorange. Do this by setting an explicit value for the manual limit and `Inf` for the automatic limit. For example, the statement

```
set(axes_handle,'XLim',[0 Inf])
```

sets the `XLimMode` property to `auto` and allows MATLAB to determine the maximum value for `XLim`. Similarly, the statement

```
set(axes_handle,'XLim',[-Inf 800])
```

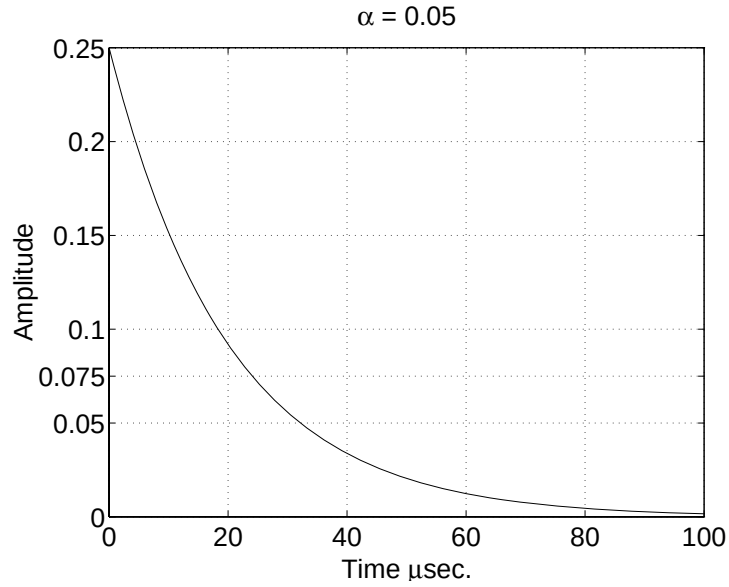
sets the `XLimMode` property to `auto` and allows MATLAB to determine the minimum value for `XLim`.

Setting Tick Mark Locations

MATLAB selects the tick mark location based on the data range to produce equally spaced ticks (for linear graphs). You can specify alternative locations for the tick marks by setting the `XTick`, `YTick`, and `ZTick` properties.

For example, if the value 0.075 is of interest for the amplitude of the function $Ae^{-\alpha t}$, specify tick marks to include that value.

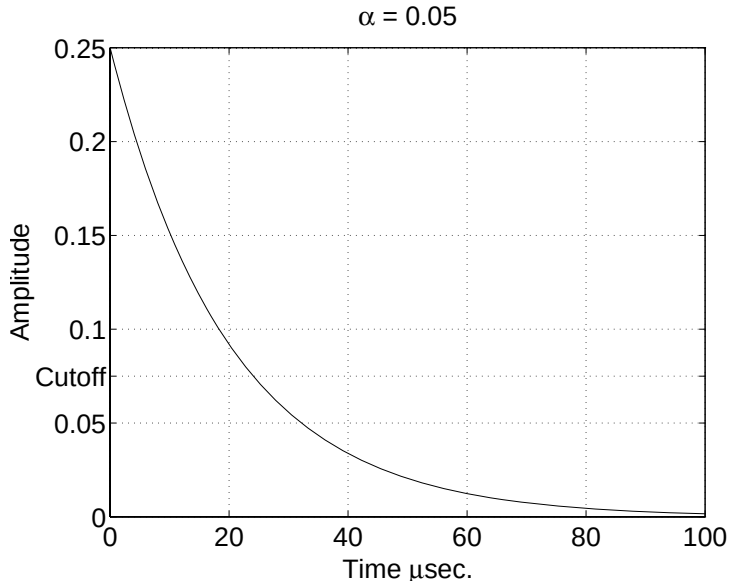
```
set(gca,'YTick',[0 0.05 0.075 0.1 0.15 0.2 0.25])
```



You can change tick labeling from numbers to strings using the `XTickLabel`, `YTickLabel`, and `ZTickLabel` properties.

For example, to label the *y*-axis value of 0.075 with the string `Cutoff`, you can specify all *y*-axis labels as a string, separating each label with the "|" character.

```
set(gca,'YTickLabel','0|0.05|Cutoff|0.1|0.15|0.2|0.25')
```



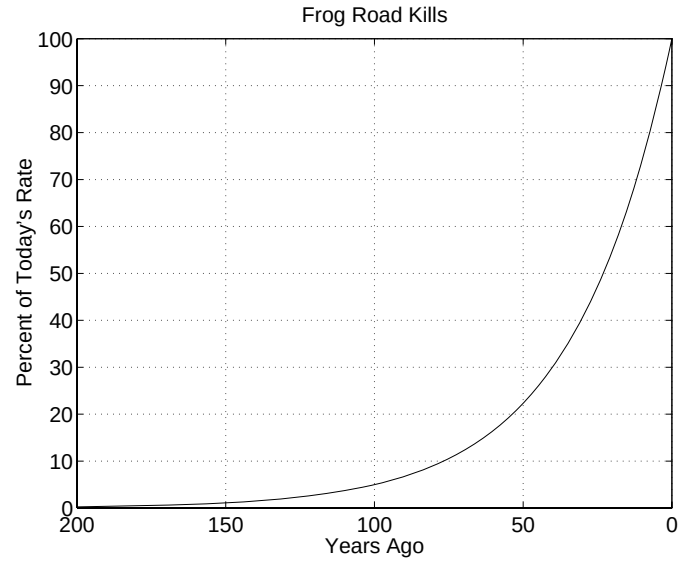
Changing Axis Direction

The `XDir`, `YDir`, and `ZDir` properties control the direction of increasing values on the respective axis. In the default 2-D view, the *x*-axis values increase from left to right and the *y*-axis values increase from bottom to top. The *z*-axis points out of the screen.

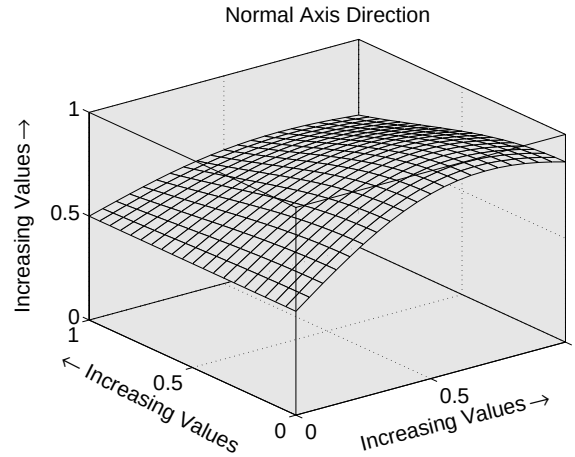
You can change the direction of increasing values by setting the associated property to `reverse`. For example, setting `XDir` to reverse,

```
set(gca,'XDir','reverse')
```

produces a plot whose *x*-axis decreases from left to right.



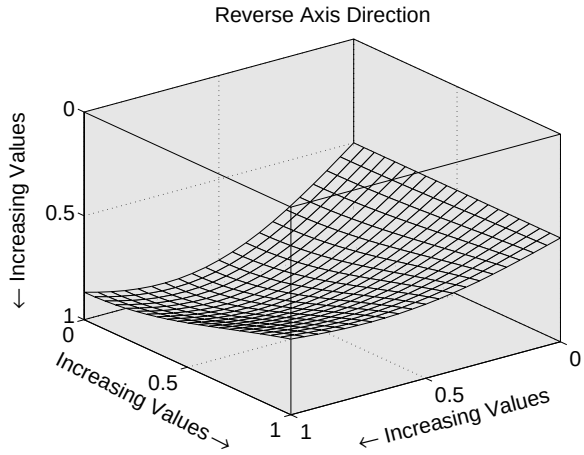
In the 3-D view, the *y*-axis increases from front to back and the *z*-axis increases from bottom to top.



Setting the *x*-, *y*-, and *z*-directions to reverse,

```
set(gca,'XDir','rev','YDir','rev','ZDir','rev')
```

yields



Using Multiple X- and Y-Axes

In this section...

Introduction

The `XAxisLocation` and `YAxisLocation` properties specify on which side of the graph to place the *x*- and *y*-axes. You can create graphs with two different *x*-axes and *y*-axes by superimposing two axes objects and using `XAxisLocation` and `YAxisLocation` to position each axis on a different side of the graph. This technique is useful to plot different sets of data with different scaling in the same graph.

Example — Double Axis Graphs

This example creates a graph to display two separate sets of data using the bottom and left sides as the *x*- and *y*-axis for one, and the top and right sides as the *x*- and *y*-axis for the other.

Suppose you have two sets of data having different *x*- and *y*-ranges:

```
x1 = [0:.1:40];
y1 = 4.*cos(x1)./(x1+2);
x2 = [1:.2:20];
y2 = x2.^2./x2.^3;
```

Using low-level `line` and `axes` routines allows you to superimpose objects easily. Plot the first data, making the color of the line and the corresponding *x*- and *y*-axis the same to more easily associate them.

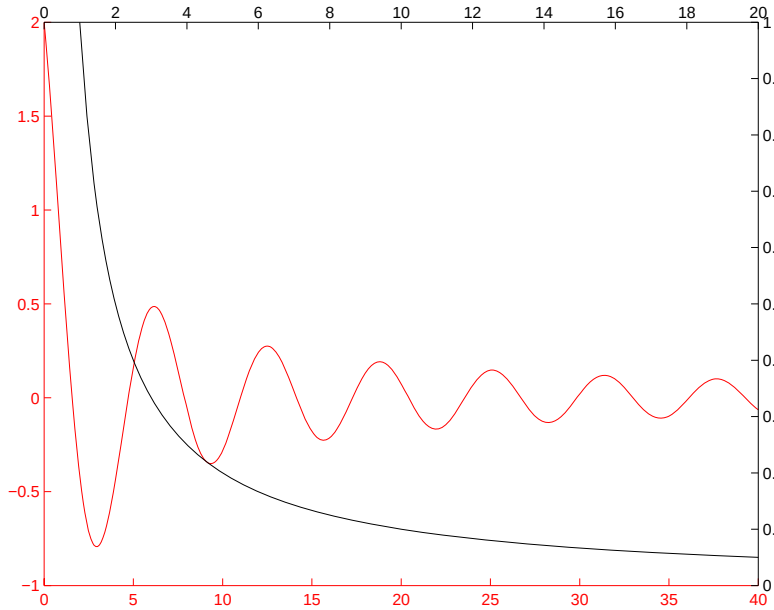
```
hl1 = line(x1,y1,'Color','r');
ax1 = gca;
set(ax1,'XColor','r','YColor','r')
```

Next, create another axes at the same location as the first, placing the *x*-axis on top and the *y*-axis on the right. Set the axes `Color` to `none` to allow the first axes to be visible and color code the *x*- and *y*-axis to match the data.

```
ax2 = axes('Position',get(ax1,'Position'),...
           'XAxisLocation','top',...
           'YAxisLocation','right',...
           'Color','none',...
           'XColor','k','YColor','k');
```

Draw the second set of data in the same color as the *x*- and *y*-axis.

```
hl2 = line(x2,y2,'Color','k','Parent',ax2);
```



Creating Coincident Grids

Since the two axes are completely independent, MATLAB determines tick mark locations according to the data plotted in each. It is unlikely the gridlines will coincide. This produces a somewhat confusing looking graph, even though the two grids are drawn in different colors. However, if you manually specify tick mark locations, you can make the grids coincide.

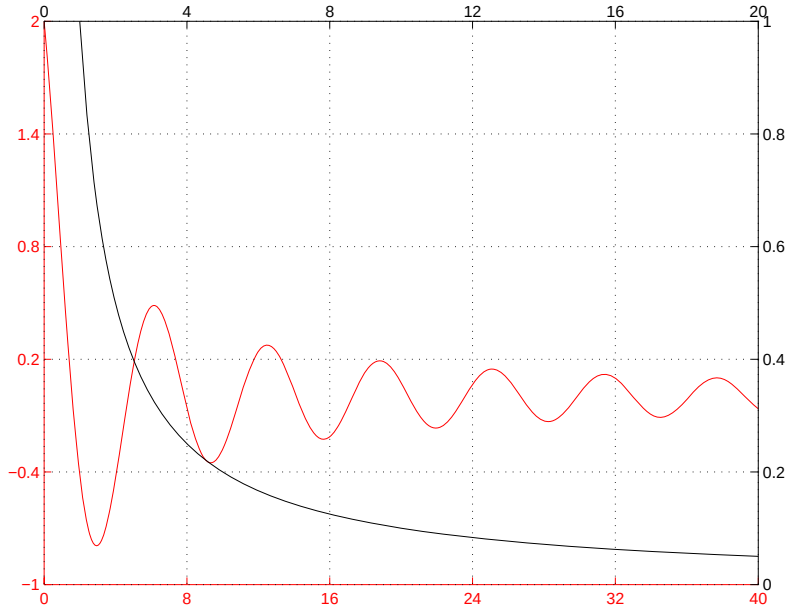
The key is to specify the same number of tick marks along corresponding axis lines (it is also necessary for both axes to be the same size). The following graph of the same data uses six tick marks per axis, equally spaced within the original limits. To calculate the tick mark locations, obtain the limits of each axis and calculate an increment.

```
xlimits = get(ax1,'XLim');
ylimits = get(ax1,'YLim');
xinc = (xlimits(2)-xlimits(1))/5;
yinc = (ylimits(2)-ylimits(1))/5;
```

Now set the tick mark locations.

```
set(ax1,'XTick',[xlimits(1):xinc:xlimits(2)],...
        'YTick',[ylimits(1):yinc:ylimits(2)])
```

Perform this calculation and set the axis limits for both axes. The resulting graph is visually simpler, even though the *y*-axis on the left has rather odd tick mark values.


0
4
8
12
16
20
2
1.4
0.8
0.2
−0.4
−1
1
0.8
0.6
0.4
0.2
0
0
8
16
24
32
40

Automatic-Mode Properties

While object creation routines that create axes children do not explicitly change axes properties, some axes properties are under automatic control when their associated mode property is set to `auto` (which is the default). The following table lists the automatic-mode properties.

Mode Property	What It Controls
`CameraPositionMode`	Positioning of the viewpoint
`CameraTargetMode`	Positioning of the camera target in the axes
`CameraUpVectorMode`	The direction of "up" in 2-D and 3-D views
`CameraViewAngleMode`	The size of the projected scene and stretch-to-fit behavior
`CLimMode`	Mapping of data values to colors
`DataAspectRatioMode`	Relative scaling of data units along x-, y-, and z-axes and stretch-to-fit behavior
`PlotBoxAspectRatioMode`	Relative scaling of plot box along x-, y-, and z-axes and stretch-to-fit behavior
`TickDirMode`	Direction of axis tick marks (in for 2-D, out for 3-D)

`XLimMode`
`YLimMode`
`ZLimMode`
	Limits of the respective x, y, and z axes

`XTickMode`
`YTickMode`
`ZTickMode`
	Tick mark spacing along the respective x-, y-, and z-axes

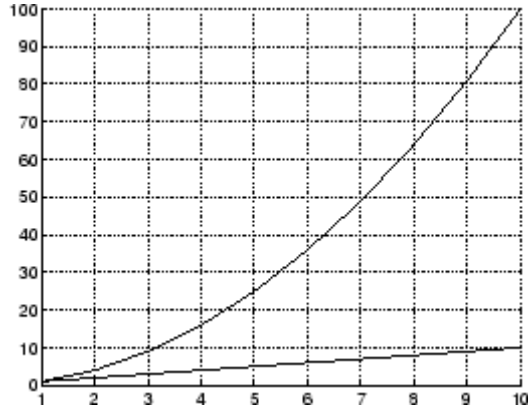
`XTickLabelMode`
`ZTickLabelMode`
`YTickLabelMode`
	Tick mark labels along the respective x-, y-, and z-axes

For example, if all property values are set to their defaults and you enter these statements,

```
line(1:10,1:10)
```

```
line(1:10,[1:10].^2)
```

the second `line` statement causes the `YLim` property to change from `[0 10]` to `[0 100]`.



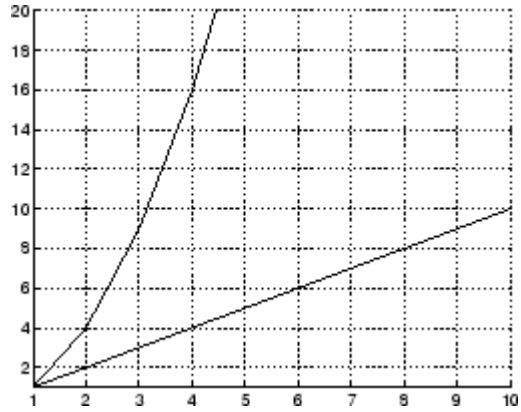
This is because `YLimMode` is `auto`, which always causes MATLAB to recompute the axis limits.

If you set the value controlled by an automatic-mode property, MATLAB sets the mode to `manual` and does not automatically recompute the value.

For example, in the statements

```
line(1:10,1:10)
set(gca,'XLim',[1 10],'YLim',[1 20])
line(1:10,[1:10].^2)
```

the `set` statement sets the *x*- and *y*-axis limits *and* changes the `XLimMode` and `YLimMode` properties to `manual`. The second `line` statement now draws a line that is clipped to the axis limits `[1 12]` instead of causing the axes to recompute its limits.


20
18
16
14
12
10
8
6
4
2
1
2
3
4
5
6
7
8
9
10

Colors Controlled by Axes

In this section...

Introduction

Axes properties specify the color of the axis lines, tick marks, labels, and the background. Properties also control the colors of the lines drawn by plotting routines and how image, patch, and surface objects obtain colors from the figure colormap.

The axes properties discussed in this section are listed in the following table.

Property	Characteristic it Controls
`Color`	Axes background color
`XColor`, `YColor`, `ZColor`	Color of the axis lines, tick marks, gridlines, and labels
`Title`	Title text object handles
`XLabel`, `YLabel`, `Zlabel`	Axis label text object handles
`CLim`	Controls mapping of graphic object `CData` to the figure colormap
`CLimMode`	Automatic or manual control of `CLim` property
`ColorOrder`	Line color autocycle order
`LineStyleOrder`	Line styles autocycle order (not a color, but related to `ColorOrder`)

Specifying Axes Colors

The default axes background color is set up by the `colordef` command, which is called in your startup file. However, you can easily define your own color scheme.

Changing the Color Scheme

Suppose you want an axes to use a “black-on-white” color scheme. First, change the background to white and the axis lines, grid, tick marks, and tick mark labels to black.

```
set(gca,'Color','w',...
        'XColor','k',...
        'YColor','k',...
        'ZColor','k')
```

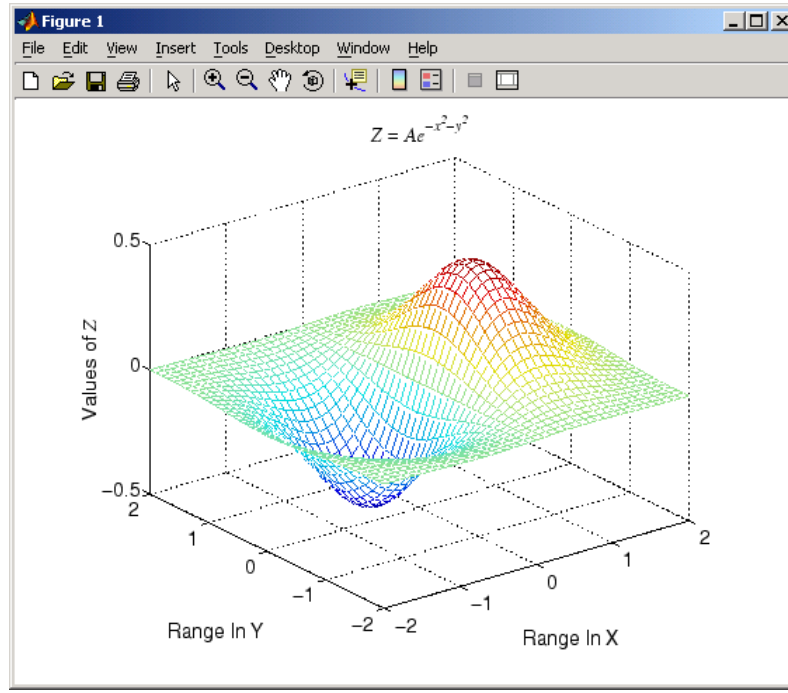
Next, change the color of the text objects used for the title and axis labels.

```
set(get(gca,'Title'),'Color','k')
set(get(gca,'XLabel'),'Color','k')
set(get(gca,'YLabel'),'Color','k')
set(get(gca,'ZLabel'),'Color','k')
```

Changing the figure background color to white completes the new color scheme.

```
set(gcf,'Color','w')
```

When you are done, a figure containing a mesh plot looks like the following figure.



You can define default values for the appropriate properties and put these definitions in your `startup.m` file. Titles and axis labels are text objects, so you must set a default color for all text objects, which is a good idea anyway because the default text color of white is not visible on the white background. Lines created with the low-level `line` function (but not the plotting routines) also have a default color of white, so you should change the default line color as well.

To set default values on the root level, use

```
set(0,'DefaultFigureColor','w'
      'DefaultAxesColor','w',...
      'DefaultAxesXColor','k',...
      'DefaultAxesYColor','k',...
      'DefaultAxesZColor','k',...
      'DefaultTextColor','k',...
      'DefaultLineColor','k')
```

MATLAB colors other axes children (i.e., image, patch, and surface objects) according to the values of their `CData` properties and the figure colormap.

Axes Color Limits — the CLim Property

In this section...

Introduction

Many 3-D plotting functions produce graphs that use color as another data dimension. For example, surface plots map surface height to color. The color limits control the limits of the color dimension in a way analogous to setting axis limits.

The axes `CLim` property controls the mapping of image, patch, and surface `CData` to the figure colormap. `CLim` is a two-element vector `[cmin cmax]` specifying the `CData` value to map to the first color in the colormap (`cmin`) and the `CData` value to map to the last color in the colormap (`cmax`). Data values in between are linearly transformed from the second to the penultimate color, using the expression

```
colormap_index = fix((CData-cmin)/(cmax-cmin)*cm_length)+1
```

`cm_length` is the length of the colormap. When the axes `CLimMode` property is `auto`, MATLAB sets `CLim` to the range of the `CData` of all graphics objects within the axes. However, you can set `CLim` to span any range of values. This enables individual axes within a single figure to use different portions of the figure's colormap. You can create colormaps with different regions, each used by a different axes.

See the `caxis` command for more information on color limits.

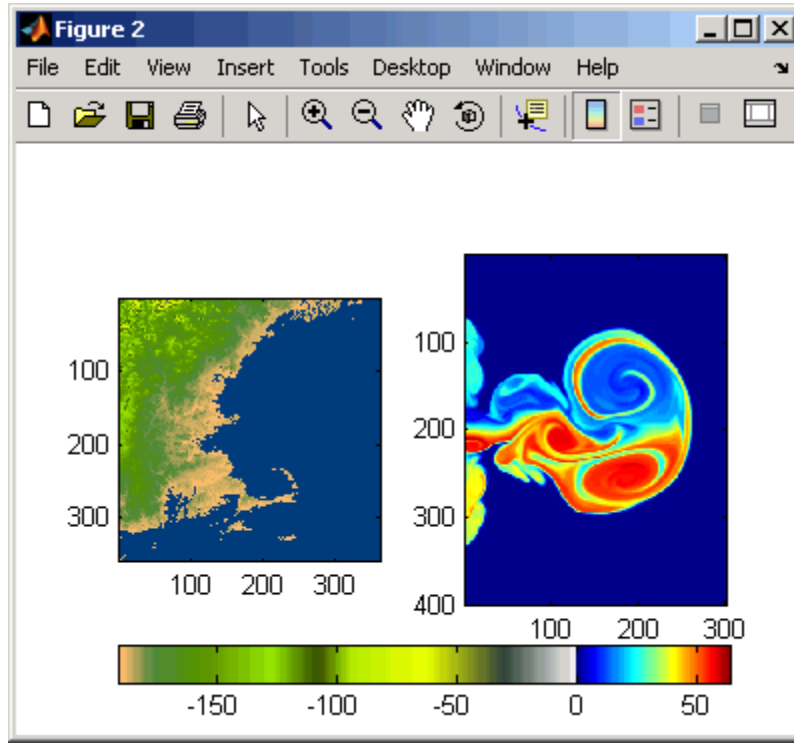
See "Introduction" on page 10-36 for an example that calculates color limits.

Simulating Multiple Colormaps in a Figure

Suppose you want to display two different images in the same figure. Images typically have their own colormaps, but you can specify only one colormap per figure. The solution is to concatenate the two colormaps and then setting the `CLim` property of each axes so that the two images map into different portions of the colormap.

Note Colormap Size Limit. On Windows platforms, the maximum length of the figure colormap is 256 colors (i.e., 256–by-3).

This example displays two images in one figure and maps the data in each image to the appropriate sections of the colormap, which has been created by concatenating the two colormaps together. The `colorbar` below the two images shows the entire colormap.



Complete Example Code

If you are using the MATLAB Help browser, you can:

- Run example
- Open the M-file in the editor

Calculating Color Limits

The key to this example is calculating values for `CLim` that cause each surface to use the section of the colormap containing the appropriate colors.

To calculate the new values for `CLim`, you need to know

- The total length of the colormap (`CmLength`)

- The beginning colormap slot to use for each axes (`BeginSlot`)
- The ending colormap slot to use for each axes (`EndSlot`)
- The minimum and maximum `CData` values of the graphic objects contained in the axes. That is, the values of the axes `CLim` property determined by MATLAB when `CLimMode` is `auto` (`CDmin` and `CDmax`).

First, define subplot regions and plot the surfaces.

```
im1 = load('cape.mat');
im2 = load('flujet.mat');
ax1 = subplot(1,2,1);
imagesc(im1.X)
axis(ax1,'image')
ax2 = subplot(1,2,2);
imagesc(im2.X)
axis(ax2,'image')
```

Concatenate two colormaps and install the new colormap.

```
colormap([im1.map;im2.map])
```

Obtain the data you need to calculate new values for `CLim`.

```
CmLength   = length(colormap);   % Colormap length
BeginSlot1 = 1;                  % Beginning slot
EndSlot1   = length(im1.map);    % Ending slot
BeginSlot2 = EndSlot1 + 1;
EndSlot2   = CmLength;
CLim1      = get(ax1,'CLim');  % CLim values for each axis
CLim2      = get(ax2,'CLim');
```

Defining a Function to Calculate CLim Values

Computing new values for `CLim` involves determining the portion of the colormap you want each axes to use relative to the total colormap size and scaling its `Clim` range accordingly. You can define a MATLAB function to do this.

```
function CLim = newclim(BeginSlot,EndSlot,CDmin,CDmax,CmLength)
   %     Convert slot number and range
```

```
    %     to percent of colormap
    PBeginSlot    = (BeginSlot - 1) / (CmLength - 1);
    PEndSlot      = (EndSlot - 1) / (CmLength - 1);
    PCmRange      = PEndSlot - PBeginSlot;
    %     Determine range and min and max
    %     of new CLim values
    DataRange     = CDmax - CDmin;
    ClimRange     = DataRange / PCmRange;
    NewCmin       = CDmin - (PBeginSlot * ClimRange);
    NewCmax       = CDmax + (1 - PEndSlot) * ClimRange;
    CLim          = [NewCmin,NewCmax];
end
```

The input arguments are identified in the bulleted list above. The function first computes the percentage of the total colormap you want to use for a particular axes (`PCmRange`) and then computes the `CLim` range required to use that portion of the colormap given the `CData` range in the axes. Finally, it determines the minimum and maximum values required for the calculated `CLim` range and returns these values. These values are the color limits for the given axes.

Using the Function

Use the `newclim` function to set the `CLim` values of each axes. The statement

```
set(ax1,'CLim',newclim(BeginSlot1,EndSlot1,CLim1(1),CLim1(2),CmLength))
```

sets the `CLim` values for the first axes so the surface uses color slots 65 to 120. The lit surface uses the lower 64 slots. You need to reset its `CLim` values as well.

```
set(ax2,'CLim',newclim(BeginSlot2,EndSlot2,CLim2(1),CLim2(2),CmLength))
```

How the Function Works

MATLAB enables you to specify any values for the axes `CLim` property, even if these values do not correspond to the `CData` of the graphics objects displayed in the axes. The minimum `CLim` value is always mapped to the first color in the colormap and the maximum `CLim` value is always mapped to the last color in the colormap, whether or not there are really any `CData` values corresponding to these colors. Therefore, if you specify values for `CLim` that

extend beyond the object's actual `CData` minimum or maximum, MATLAB colors the object with only a subset of the colormap.

The `newclim` function computes values for `CLim` that map the graphics object's actual `CData` values to the beginning and ending colormap slots that you specify. It does this by defining a "virtual" graphics object having the computed `CLim` values.

Defining the Color of Lines for Plotting

In this section...

Introduction

The axes `ColorOrder` property determines the color of the individual lines drawn by the `plot` and `plot3` functions. For multiline graphs, these functions cycle through the colors defined by `ColorOrder`, repeating the cycle when they reach the end of the list.

The `colordef` command defines various color order schemes for different background colors. `colordef` is typically called in the `matlabrc` file, which is executed during MATLAB startup.

Defining Your Own ColorOrder

You can redefine `ColorOrder` to be any *m*-by-3 matrix of RGB values, where *m* is the number of colors. However, high-level functions like `plot` and `plot3` reset most axes properties (including `ColorOrder`) to the defaults each time you call them. To use your own `ColorOrder` definition you must do one of the following three things:

- Define a default `ColorOrder` on the figure or root level
- Change the axes `NextPlot` property to `add` or `replacechildren`
- Use the informal form of the `line` function, which obeys the `ColorOrder` but does not clear the axes or reset properties

Changing the Default ColorOrder

You can define a new `ColorOrder` that MATLAB uses within a particular figure, for all axes within any figures created during the MATLAB session, or as a user-defined default that MATLAB always uses.

To change the ColorOrder for all plots in the current figure, set a default in that figure. For example, to set ColorOrder to the colors red, green, and blue, use the statement

```
set(gcf,'DefaultAxesColorOrder',[1 0 0;0 1 0;0 0 1])
```

To define a new ColorOrder that MATLAB uses for all plotting during your entire MATLAB session, set a default on the root level so axes created in any figure use your defaults.

```
set(0,'DefaultAxesColorOrder',[1 0 0;0 1 0;0 0 1])
```

To define a new ColorOrder that MATLAB always uses, place the previous statement in your startup.m file.

Setting the NextPlot Property

The axes NextPlot property determines how high-level graphics functions draw into an existing axes. You can use this property to prevent plot and plot3 from resetting the ColorOrder property each time you call them, but still clear the axes of any existing plots.

By default, NextPlot is set to replace, which is equivalent to a cla reset command (i.e., delete all axes children and reset all properties, except Position, to their defaults). If you set NextPlot to replacechildren,

```
set(gca,'NextPlot','replacechildren')
```

MATLAB deletes the axes children, but does not reset axes properties. This is equivalent to a cla command without the reset.

After setting NextPlot to replacechildren, you can redefine the ColorOrder property and call plot and plot3 without affecting the ColorOrder.

Setting NextPlot to add is the equivalent of issuing the hold on command. This setting prevents MATLAB from resetting the ColorOrder property, but it does not clear the axes children with each call to a plotting function.

Using the line Function

The behavior of the `line` function depends on its calling syntax. When you use the informal form (which does not include any explicit property definitions),

```
line(x,y,z)
```

`line` obeys the `ColorOrder` property, but does not clear the axes with each invocation or change the view to 3-D (as `plot3` does). However, `line` can be useful for creating your own plotting functions where you do not want the automatic behavior of `plot` or `plot3`, but you do want multiline graphs to use a particular `ColorOrder`.

Line Styles Used for Plotting — LineStyleOrder

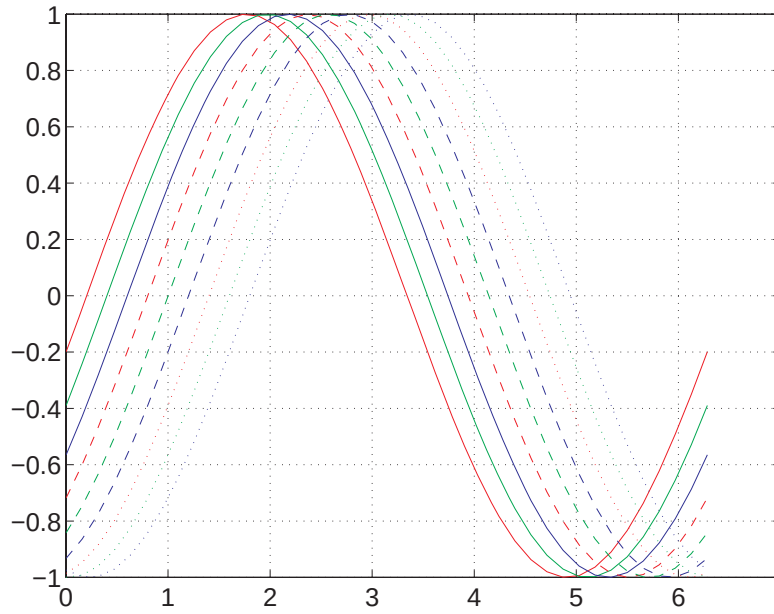
The axes `LineStyleOrder` property is analogous to the `ColorOrder` property. It specifies the line styles to use for multiline plots created with the `plot` and `plot3` functions. MATLAB increments the line style only after using all of the colors in the `ColorOrder` property. It then uses all the colors again with the second line style, and so on.

For example, define a default `ColoOrder` of red, green, and blue and a default `LineStyleOrder` of solid, dashed, and dotted lines.

```
set(0,'DefaultAxesColorOrder',[1 0 0;0 1 0;0 0 1],...
      'DefaultAxesLineStyleOrder','-|--|:')
```

Then plot some multiline data.

```
t = 0:pi/20:2*pi;
a = ones(length(t),9);
for i = 1:9
    a(:,i) = sin(t-i/5)';
end
plot(t,a)
```



MATLAB cycles through all colors for each line style.

Index

A

B

C

D

E

F

G

L

M

N

O

P

Q

R

S

T

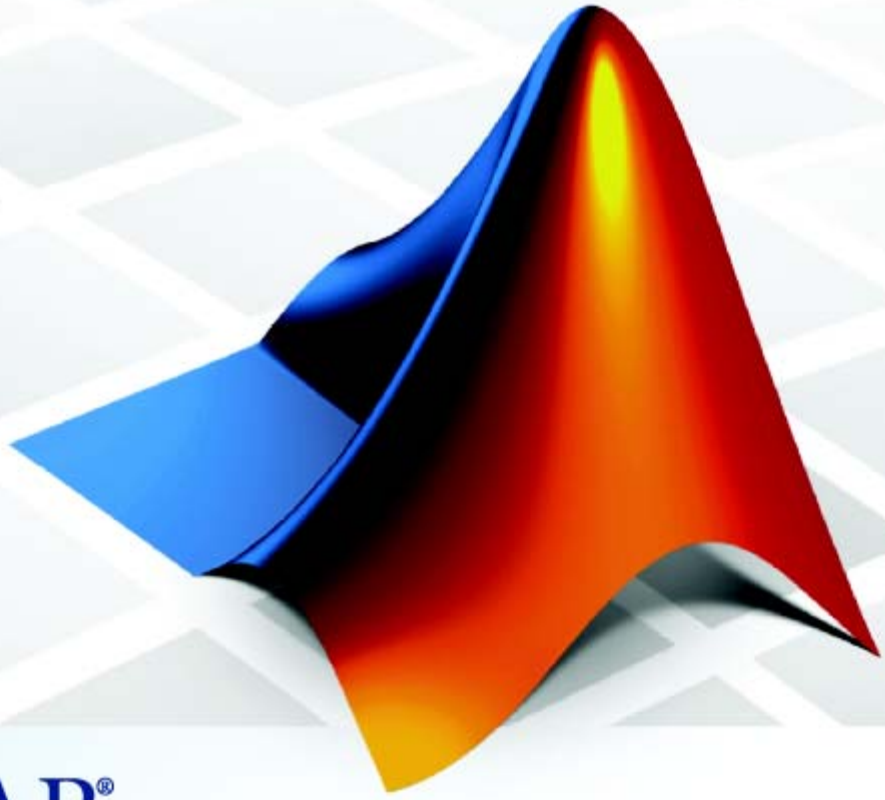
U

V

X

Z

MATLAB® 7
MAT-File Format



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks:

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MAT-File Format

Revision History

June 1999	Online only	New for MATLAB 5.3 (Release 11)
November 2000	PDF only	Revised for MATLAB 6.0 (Release 12)
June 2001	PDF only	Revised for MATLAB 6.1 (Release 12.1)
July 2002	PDF only	Revised for MATLAB 6.5 (Release 13)
January 2003	PDF only	Revised for MATLAB 6.5.1 (Release 13 SP1)
June 2004	PDF only	Revised for MATLAB 7.0 (Release 14)
October 2004	PDF only	Revised for MATLAB 7.0.1 (Release 14SP1)
September 2005	PDF only	Minor revision for MATLAB 7.1 (Release 14SP3)
September 2007	PDF only	Minor revision for MATLAB 7.5 (Release R2007b)

1

MAT-File Format

Introduction

This document describes the internal format of MATLAB® Level 4 and Level 5 MAT-files. Level 4 MAT-files are compatible with versions of MATLAB up to Version 4. Level 5 MAT-files are compatible with MATLAB Versions 5 and up. You can read and write Level 4 MAT-files with the later versions of MATLAB, but when writing a MAT-file under these circumstances, you need to specify a switch in the `save` or `matOpen` command line to tell MATLAB that the MAT-file is at Level 4.

A MAT-file stores data in binary (not human-readable) form. In MATLAB, you create MAT-files using the `save` function, which writes the arrays currently in memory to a file as a continuous byte stream. By convention, this file has the filename extension `.mat`; thus the name MAT-file. The `load` function reads the arrays from a MAT-file into the MATLAB workspace.

Most MATLAB users do not need to know the internal format of a MAT-file. Even users who must read and write MAT-files from C and Fortran programs do not need to know the MAT-file format if they use the MAT-file interface. This interface shields users from dependence on the details of the MAT-file format.

Note See "Importing and Exporting Data" in the MATLAB External Interfaces documentation for information on the MAT-file interface. See "C [or Fortran] MAT-File Functions" in the MATLAB External Interfaces Reference documentation for information on the functions available with this interface.

However, if you need to read or write MAT-files on a system that does not support the MAT-file interface, you must write your own read and write routines. The MAT-file interface is only available for platforms on which MATLAB is supported. This document provides the details about the MAT-file format you will need to read and write MAT-files on these systems.

Note Whenever possible, The MathWorks strongly advises you to use the MAT-file interface functions to read and write MAT-files. Any code you write that depends on the MAT-file format may need to be rewritten when the format changes in future releases.

MAT-File Formats

This document describes both Level 5 and Level 4 MAT-file formats. The Level 5 MAT-file format supports all the array types supported in MATLAB versions 5 and up, including multidimensional numeric arrays, character arrays, sparse arrays, cell arrays, structures, and objects. “Level 5 MAT-File Format” on page 1-4 describes this format.

The Level 4 MAT-file format is a simpler format but it only supports two-dimensional matrices and character strings. “Level 4 MAT-File Format” on page 1-36 describes this format.

Level 5 MAT-File Format

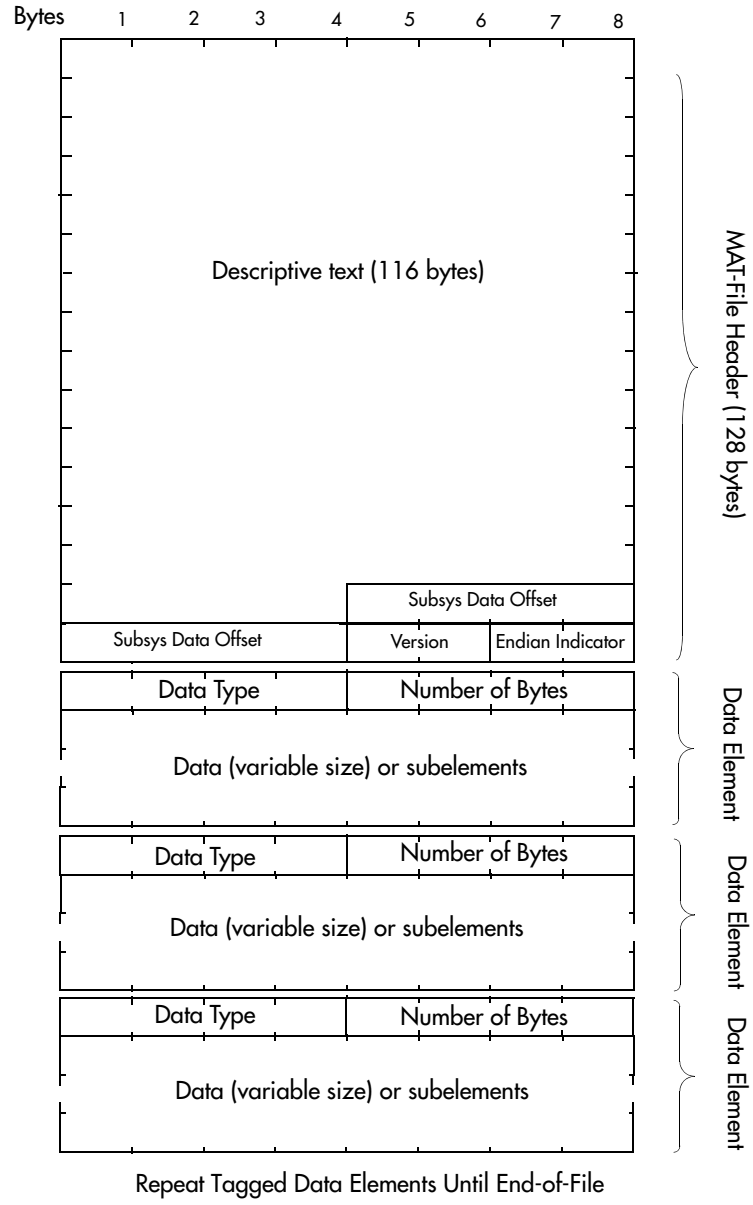
Level 5 MAT-files are made up of a 128-byte *header* followed by one or more *data elements*. Each data element is composed of an 8-byte *tag* followed by the data in the element. The tag specifies the number of bytes in the data element and how these bytes should be interpreted; that is, should the bytes be read as 16-bit values, 32-bit values, floating point values or some other data type.

By using tags, the Level 5 MAT-file format provides quick access to individual data elements within a MAT-file. You can move through a MAT-file by finding a tag and then skipping ahead the specified number of bytes until the next tag.

"MATLAB Level 5 MAT-File Format" on page 1-5 graphically illustrates this MAT-file format. The sections that follow provide more details about these MAT-file elements.

This section covers the following topics:

Figure 1-1: MATLAB Level 5 MAT-File Format



MAT-File Header Format

Level 5 MAT-files begin with a 128-byte header made up of a 124 byte text field and two, 16-bit flag fields.

This section covers the following topics:

Header Text Field

The first 116 bytes of the header can contain text data in human-readable form. This text typically provides information that describes how the MAT-file was created. For example, MAT-files created by MATLAB include the following information in their headers:

- Level of the MAT-file (value equals `1` for Level 5)
- Platform on which the file was created
- Date and time the file was created

You can view the text in a MAT-file header using the `cat` command on UNIX systems, or the `type` command on a PC. The output displays the text in this part of the header. (The display of the header is followed by unreadable characters representing the binary data in the file.)

```
cat my_matfile.mat
MATLAB 5.0 MAT-file, Platform: SOL2, Created on: Thu Nov 13
10:10:27 1997
```

Note When creating a MAT-file, you *must* write data in the first four bytes of this header. MATLAB uses these bytes to determine if a MAT-file uses a Level 5 format or a Level 4 format. If any of these bytes contain a zero, MATLAB will incorrectly assume the file is a Level 4 MAT-file.

Header Subsystem Data Offset Field

Bytes 117 through 124 of the header contain an offset to subsystem-specific data in the MAT-file. All zeros or all spaces in this field indicate that there is no subsystem-specific data stored in the file.

Header Flag Fields

The last four bytes in the header are divided into two, 16-bit flag fields (`int16`).

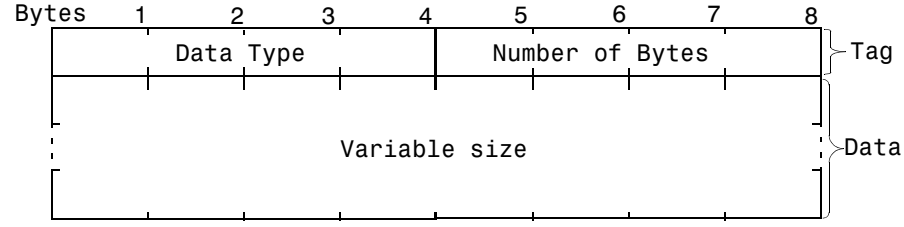
Field	Value
Version	When creating a MAT-file, set this field to `0x0100`.
Endian Indicator	Contains the two characters, `M` and `I`, written to the MAT-file in this order, as a 16-bit value. If, when read from the MAT-file as a 16-bit value, the characters appear in reversed order (`IM` rather than `MI`), it indicates that the program reading the MAT-file must perform byte-swapping to interpret the data in the MAT-file correctly.

Note Programs that create MAT-files always write data in their native machine format. Programs that read MAT-files are responsible for byte-swapping.

Data Element Format

Each data element begins with an 8-byte tag followed immediately by the data in the element. Figure 1-2 shows this format. (MATLAB also supports a compressed data element format. See page 1-10 for more information.)

Figure 1-2: MAT-File Data Element Format



This section covers the following topics:

The Tag Field

The 8-byte data element tag is composed of two, 32-bit fields:

- Data Type
- Number of Bytes

Data Type. The Data Type field specifies how the data in the element should be interpreted, that is, its size and format. The MAT-file format supports many data types including signed and unsigned, 8-bit, 16-bit, 32-bit, and 64-bit data types, a special data type that represents MATLAB arrays, Unicode encoded character data, and data stored in compressed format. Table 1-1 lists all these data types with the values used to specify them. The table also includes

symbols that are used to represent these data types in the examples in this document.

Table 1-1: MAT-File Data Types

Value	Symbol	MAT-File Data Type
1	`miINT8`	8 bit, signed
2	`miUINT8`	8 bit, unsigned
3	`miINT16`	16-bit, signed
4	`miUINT16`	16-bit, unsigned
5	`miINT32`	32-bit, signed
6	`miUINT32`	32-bit, unsigned
7	`miSINGLE`	IEEE 754 single format
8	`--`	Reserved
9	`miDOUBLE`	IEEE 754 double format
10	`--`	Reserved
11	`--`	Reserved
12	`miINT64`	64-bit, signed
13	`miUINT64`	64-bit, unsigned
14	`miMATRIX`	MATLAB array
15	`miCOMPRESSED`	Compressed Data
16	`miUTF8`	Unicode UTF-8 Encoded Character Data
17	`miUTF16`	Unicode UTF-16 Encoded Character Data
18	`miUTF32`	Unicode UTF-32 Encoded Character Data

The `UTF-16` and `UTF-32` encodings are in the byte order specified by the Endian Indicator (See “Header Flag Fields” on page 1-7). UTF-8 is byte order neutral.

For character data that is not Unicode encoded, the Data Type part of the Tag field should be set to `miUINT16`.

For more information about the `miMATRIX` data type, see "Level 5 MATLAB Array Data Element Formats" on page 1-14.

Number of Bytes. The Number of Bytes field is a 32-bit value that specifies the number of bytes of data in the element. This value does not include the eight bytes of the data element's tag.

If Data Type is `miCOMPRESSED`, then the Number of Bytes field contains the compressed MATLAB array size in bytes. (See "Data Compression" on page 1-12.)

The Data Field

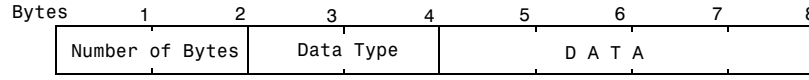
The data immediately follows the tag. All data that is uncompressed must be aligned on 64-bit boundaries. When writing a MAT-file, if the amount of data in a data element falls short of a 64-bit boundary, you must add bytes of padding to make sure the tag of the next data element falls on a 64-bit boundary. Likewise, when reading data from a MAT-file, be sure to account for these padding bytes.

Note For data elements representing MATLAB arrays, (type `miMATRIX`), the value of the Number of Bytes field includes padding bytes in the total. For all other MAT-file data types, the value of the Number of Bytes field does *not* include padding bytes.

Small Data Element Format

If a data element takes up only one to four bytes, MATLAB saves storage space by storing the data in an 8-byte format. In this format, the Data Type and Number of Bytes fields are stored as 16-bit values, freeing four bytes in the tag in which to store the data. Figure 4 illustrates this format.

Figure 1-3: Small Data Element Format

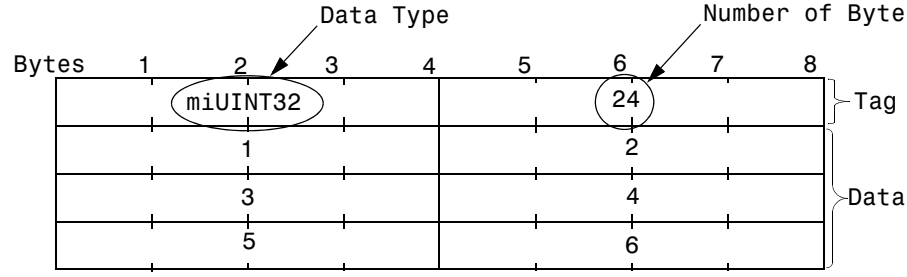


Note When reading a MAT-file, you can tell if you are processing a small data element by comparing the value of the first two bytes of the tag with the value zero (0). If these two bytes are not zero, the tag uses the small data element format. When writing to a MAT-file, use of the small data element format is optional.

Example Data Element

Figure 1-4 illustrates a data element representing an array of six 32-bit, unsigned integers: 1, 2, 3, 4, 5, 6. In the figure, the Data Type field contains the value from Table 1-1 that specifies unsigned, 32-bit integers (`miUINT32`). The Number of Bytes field in the data element tag contains the number of data values multiplied by the number of bytes used to represent each value. Note that this value does not include the eight bytes in the data element tag.

Figure 1-4: Example MAT-File Data Element



Data Compression

MATLAB compresses the data it saves to a MAT-file using buffered in-memory gzip compression. It compresses MATLAB variables transparently as they are written out to disk. This technique uses less memory than systems that compress an entire variable at once before writing it out to disk. Also, no temporary files are required in order to read or write compressed data.

Because it compresses each variable individually, MATLAB can read a compressed MAT-file like any other MAT-file. No code changes are required at either the C or M level to read a compressed MAT-file.

MATLAB compresses data for MAT-files (file I/O) only, not for sequential streams. The reason for this is that the size of the compressed variable is known only after it is compressed, but it must be written in the tag at the beginning of the variable. In a file it is possible to seek back and write the size, while in a stream this cannot be done.

MAT-files containing compressed variables are non-platform specific like any MAT-file, and such a file saved on any MATLAB supported platform can be loaded on any other supported platform.

To decompress the contents of a compressed variable in a MAT-file, you can use the `uncompress` function from the freeware `zlib-1.1.4` library available at the `gzip` web site, `http://www.gzip.org/zlib/`. Once a MATLAB array has been decompressed, you can ignore the `miCOMPRESSED` tag and process the data normally, as if it had not been compressed.

Note To disable data compression when writing to MAT-files, see "Saving and Loading MAT-files" in the "Data Import and Export" chapter of the MATLAB Programming documentation.

Storing Compressed Data

MATLAB stores compressed data in gzip-compressed MATLAB arrays. Each compressed variable is stored complete with its tags and data field in the same format as uncompressed variables, as described in "Data Element Format" on page 1-8. The difference is that the entire variable is compressed into a data buffer, and this buffer is preceded by an 8-byte tag named `miCOMPRESSED`. The tag contains the length of the compressed buffer.

Each variable in a MAT-file has a 56-byte header. Even if the data is stored in the header itself, as can be the case for variables containing 1 to 4 bytes of data, no variable in a MAT-file can be less than 56 bytes. Thus, regardless of how random the data in a variable may be, it is unlikely (but not impossible) that a compressed variable will take more space than its uncompressed counterpart. This is because the 56-byte header always compresses to a smaller size. Note that compression works best on nonrandom data. The more random the data, the less it will compress.

Level 5 MATLAB Array Data Element Formats

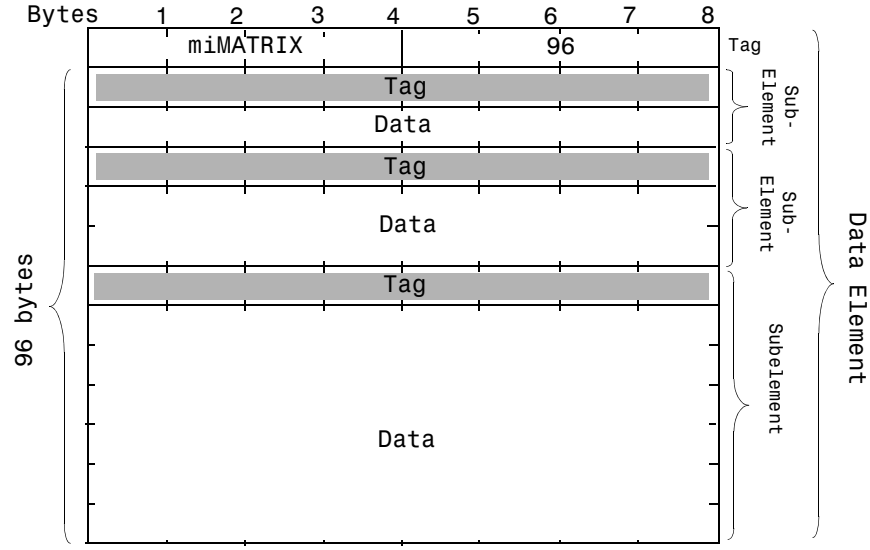
The MAT-file data type `miMATRIX` (`14`) is used to represent all types of MATLAB arrays, including:

- Numeric arrays
- Character arrays
- Sparse arrays
- Cell arrays
- Structures
- Objects

The `miMATRIX` data type is a compound data type. MAT-file data elements of this type are composed of multiple *subelements*. The subelements can be of any other MAT-file data type, including other `miMATRIX` data types.

Figure 1-5 shows a `miMATRIX` data element composed of three subelements. Note how each subelement is a data element with its own tag. The value of the Number of Bytes field (`96` in the figure) in the data element tag includes all the subelements.

Figure 1-5: MATLAB Array Data Element with Subelements



Each `miMATRIX` data element representing the different types of MATLAB arrays each has a specific set of subelements. Some of these subelements are common to all MATLAB arrays. Others subelements are unique to a particular type of array. The following sections detail the subelements for each MATLAB array type.

Numeric Array and Character Array Data Element Formats

A MAT-file data element representing a MATLAB numeric array or character array is composed of four subelements and one optional subelement. Table 1-2 lists the subelements in the order in which they appear in the data element. The table also includes the values of the Data Type and Number of Bytes fields you would use in the tag of each subelement. For an example, see “Examples of Numeric Array Data Elements” on page 1-19.

Table 1-2: Numeric and Character Array Subelements with Tag Data

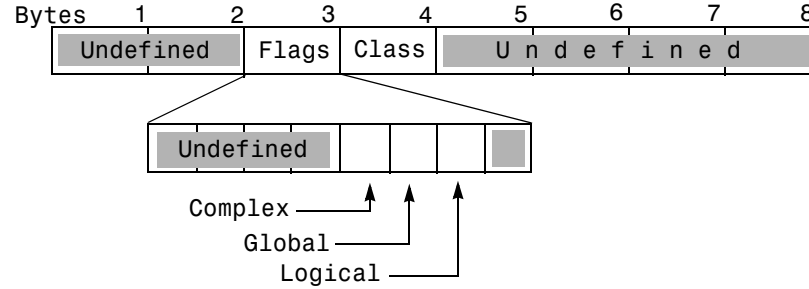
Subelement	Data Type	Number of Bytes
Array Flags	`miUINT32`	`2*size-of-Data-Type (8 bytes)`
Dimensions Array	`miINT32`	`number-of-dimensions*size-of-Data-Type` (To learn how to determine the number of dimensions, see “Dimensions Array Subelement” on page 1-17.)
Array Name	`miINT8`	`number-of-characters*size-of-Data-Type`
Real part (`pr`)	Any of the numeric data types.	`number-of-values*size-of-Data-Type`
Imaginary part (`pi`) (Optional)	Any of the numeric data types.	`number-of-values*size-of-Data-Type`

Array Flags Subelement

This subelement identifies the MATLAB array type (class) represented by the data element and provides other information about the array. The Array Flags subelement is common to all array types.

Figure 1-6 illustrates the format of the Array Flags subelement. (For sparse matrices, bytes 5 through 8 are used to store the maximum number of nonzero elements in the matrix. See “Sparse Array Data Element Format” on page 1-22 for more information.)

Figure 1-6: Array Flags Format



Flags. This field contains three, single-bit flags that indicate whether the numeric data is complex, global, or logical. If the complex bit is set, the data element includes an imaginary part (`pi`). If the global bit is set, MATLAB loads the data element as a global variable in the base workspace. If the logical bit is set, it indicates the array is used for logical indexing.

Class. This field contains a value that identifies the MATLAB array type (class) represented by the data element. Table 1-3 lists the MATLAB array types with the values you use to specify them. The table also includes symbols that are used to represent the MATLAB array type in the examples in this document.

Note The value of the Class field identifies the MATLAB data type. The value of the Data Type field in the data element tag identifies the data type used to store the data *in the MAT-file*. The MAT-file data types are listed in Table 1-1. The value of the Class and the Data Type fields do not need to be the same; for more information, see “Automatic Compression of Numeric Data” on page 1-19.

Table 1-3: MATLAB Array Types (Classes)

MATLAB Array Type (Class)	Value	Symbol
Cell array	1	mxCELL_CLASS
Structure	2	mxSTRUCT_CLASS
Object	3	mxOBJECT_CLASS
Character array	4	mxCHAR_CLASS
Sparse array	5	mxSPARSE_CLASS
Double precision array	6	mxDOUBLE_CLASS
Single precision array	7	mxSINGLE_CLASS
8-bit, signed integer	8	mxINT8_CLASS
8-bit, unsigned integer	9	mxUINT8_CLASS
16-bit, signed integer	10	mxINT16_CLASS
16-bit, unsigned integer	11	mxUINT16_CLASS
32-bit, signed integer	12	mxINT32_CLASS
32-bit unsigned, integer	13	mxUINT32_CLASS

For numeric arrays, Class can contain any of the numeric array types: mxDOUBLE_CLASS, mxSINGLE_CLASS, mxINT8_CLASS, mxUINT8_CLASS, mxINT16_CLASS, mxUINT16_CLASS, mxINT32_CLASS, or mxUINT32_CLASS.

For character arrays, Class contains mxCHAR_CLASS.

Dimensions Array Subelement

This subelement specifies the size of each dimension of an *n*-dimensional array in an *n*-sized array of 32-bit values (miINT32). All numeric arrays have at least two dimensions. The Dimensions Array subelement is common to all MATLAB array types.

For example, if a data element represents a 2-by-3-by-2 MATLAB array, the Dimensions Array subelement would contain three values: `2`, `3`, and `2`.

Note To calculate the number of dimensions in an array, divide the value stored in the Number of Bytes field in the Dimensions Array subelement tag by `4`, the number of bytes in the data type (`miINT32`) used in the subelement.

Array Name Subelement

This subelement specifies the name assigned to the array, as an array of signed, 8-bit values (`miINT8`). This subelement is common to all array types.

Real Part (pr) Subelement

This subelement contains the numeric data in the MATLAB array. If the array contains complex numbers (the complex bit in the Array Flags is set), this is the real part of the number.

The data type of the values can be any of the numeric data types listed in Table 1-1, MAT-File Data Types, on page 1-9.

For character data that is not Unicode encoded, the Data Type part of the Tag field should be set to `miUINT16`.

Imaginary Part (pi) Subelement

This subelement contains the imaginary part of the numeric data in the MATLAB array. This subelement is only present if one or more of the numeric values in the MATLAB array is a complex number (if the complex bit is set in Array Flags). The data type of the values can be any of the numeric data types listed in Table 1-1, MAT-File Data Types, on page 1-9.

Note When reading a MAT-file, check the value of the Data Type field in the tag of Real Part and Imaginary Part subelements to identify the data type used to store data. Also note that MATLAB reads and writes these values in column-major order.

Automatic Compression of Numeric Data

MATLAB stores the numeric data in an array in double precision format. When MATLAB writes a numeric (or sparse) array to a MAT-file, it uses the smallest possible data type to store the data, both the real and imaginary parts.

For example, if MATLAB determines that the data stored in double precision format can actually be stored in an 8-bit format, it will use an 8-bit data type to store it in a MAT-file. Note, however, that if any of the numeric values in the array requires a 64-bit representation, MATLAB stores all of the data in a 64-bit data type. See "Compressed Data Element" on page 1-21 for an example.

When you create a MAT-file, compressing data is optional.

Note When MATLAB uses a smaller data type to store data in a MAT-file, the value of the Class field in the Array Flags subelement identifies the original MATLAB data type.

Examples of Numeric Array Data Elements

This section uses examples to illustrate both the compressed and uncompressed numeric array data element formats.

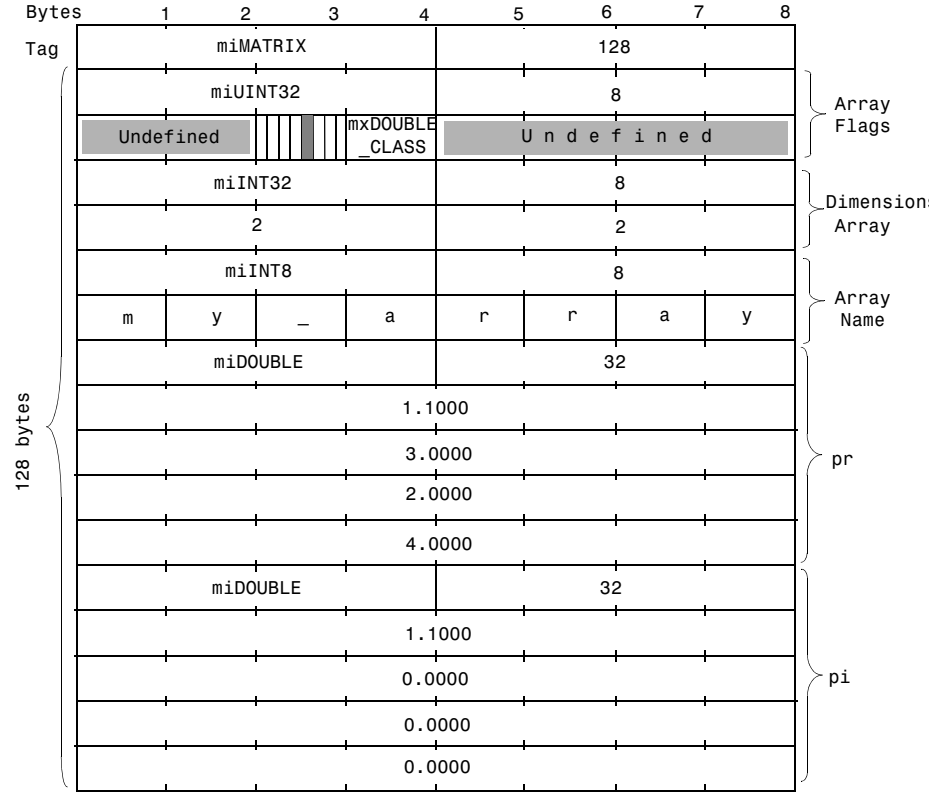
Uncompressed Data Element. Figure 1-7 shows how this 2-by-2 numeric array, `my_array`, is represented in a MAT-file.

```
my_array = [ 1.1+1.1i 2 ; 3 4 ]

my_array =

     1.1000 + 1.1000i     2.0000
     3.0000               4.0000
```

In the figure, note:

- The data element includes five subelements. Because one of the numeric values in the array is a complex number, the complex bit flag in the Array Flags subelement is set and the Imaginary Part (`pi`) subelement is included.
- The value of the Number of Bytes field in the data element tag includes all the subelements, but not the eight bytes of the tag itself.

Figure 1-7: Example Numeric Array MAT-File Data Element



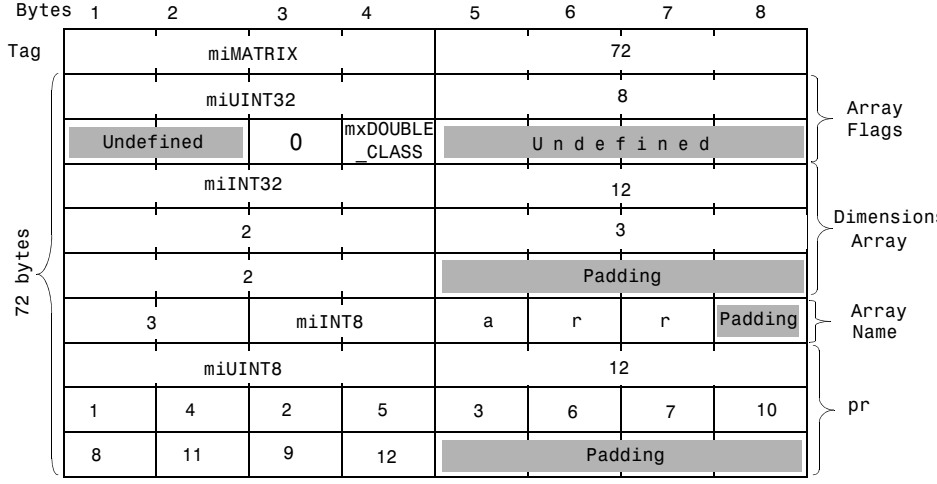
Compressed Data Element. Figure 1-8 shows how the three-dimensional numeric array in this example, `arr`, is represented in a MAT-file when compression is used to conserve storage space.

```
A = [ 1 2 3 ; 4 5 6 ];
B = [ 7 8 9 ; 10 11 12];
arr = cat(3,A,B)
arr(:,:,1) =
     1     2     3
     4     5     6

arr(:,:,2) =
     7     8     9
    10    11    12
```

In the figure, note:

- The Array Name subelement uses the compressed data element format.
- The numeric data in the array, stored in double precision format in MATLAB, is stored as 8-bit, unsigned values in the `pr` subelement. The Class field in the Array Flags subelement identifies the original MATLAB data type.

Figure 1-8: Example Numeric Array MAT-file Data Element (Compressed)



Sparse Array Data Element Format

A MAT-file data element representing a MATLAB sparse array is composed of six subelements and one optional subelement. Table 1-4 lists the subelements in the order in which they appear in the data element. The table lists the values of the Data Type and Number of Bytes fields of the tag for each subelement.

Table 1-4: Sparse Array Subelements with Tag Data

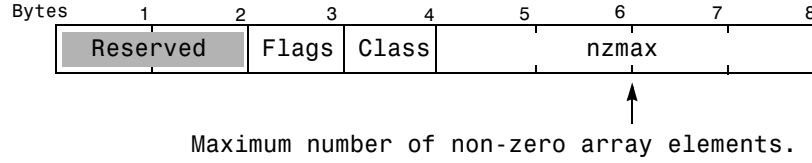
Subelement	Data Type	Number of Bytes
Array Flags	`miUINT32`	`2*size-of-Data-Type (8 bytes)`
Dimensions Array	`miINT32`	`number-of-dimensions*size-of-Data-Type` where `number-of-dimensions` can be 0, 1 or 2.
Array Name	`miINT8`	`number-of-characters*size-of-Data-Type`
Row Index (`ir`)	`miINT32`	`nzmax*size-of-Data-Type` (The `nzmax` value is stored in Array Flags.)
Column Index (`jc`)	`miINT32`	`(N+1)*sizeof(int32)` where `N` is the second element of the Dimensions array subelement.
Real part (`pr`)	Any numeric data type	`number-of-nonzero-values*size-of-Data-Type`
Imaginary part (`pi`) (Optional)	Any numeric data type	`number-of-nonzero-values*size-of-Data-Type`

Array Flags Subelement

This subelement identifies the MATLAB array type (class) represented by the data element and provides other information about the array. The Array Flags subelement is common to all array types.

Figure 1-9 shows the Array Flags format. For sparse arrays, this value also contains the maximum number of non-zero elements in the array (`nzmax`).

Figure 1-9: Array Flags Format for Sparse Arrays



Flag. For more information, see "Flags" on page 1-16.

Class. This field contains a value that identifies the MATLAB data type represented by the data element. For sparse arrays, Class contains the value `5` (`mxSPARSE_CLASS`). See "Class" on page 1-16 for more information.

Dimensions Array Subelement

This subelement specifies the size of each dimension of the array. This subelement is common to all array types. For more information, see "Dimensions Array Subelement" on page 1-17.

Note that MATLAB only supports two-dimensional sparse arrays.

Array Name Subelement

This subelement specifies the name assigned to the array. This subelement is common to all array types. For more information, see "Array Name Subelement" on page 1-18.

Row Index for Non-zero Values (ir) Subelement

This subelement specifies the row indices of the non-zero elements in the real part (`pr`) of the matrix data and the imaginary part (`pi`) of the matrix data, if present. This subelement is a series of 32-bit (`miINT32`) values.

Column Index for Non-Zero Values (jc) Subelement

This subelement contains column index information as a series of 32-bit (`miINT32`) values. For more information about what this subelement contains, see the *MATLAB Application Program Interface Guide*.

Real Part (pr) Subelement

This subelement contains the numeric data in the MATLAB array. If the array contains complex numbers (the complex bit in the Array Flags is set), this is the real part of the number.

Because MATLAB uses data compression to save storage space, the data type of the values can be any of the numeric data types listed in Table 1-1, MAT-File Data Types, on page 1-9. For more information, see "Automatic Compression of Numeric Data" on page 1-19.

Imaginary Part (pi) Subelement

This subelement contains the imaginary data in the array, if one or more of the numeric values in the MATLAB array is a complex number (if the complex bit is set in Array Flags).

Because MATLAB uses data compression to save storage space, the data type of the values can be any of the numeric data types listed in Table 1-1, MAT-File Data Types, on page 1-9. For more information, see "Automatic Compression of Numeric Data" on page 1-19.

Note You must check the value of the Data Type field in the tag of Real Part and Imaginary Part subelements to identify the type of the data. Also note that MATLAB reads and writes these values in column-major order.

Example Sparse Array

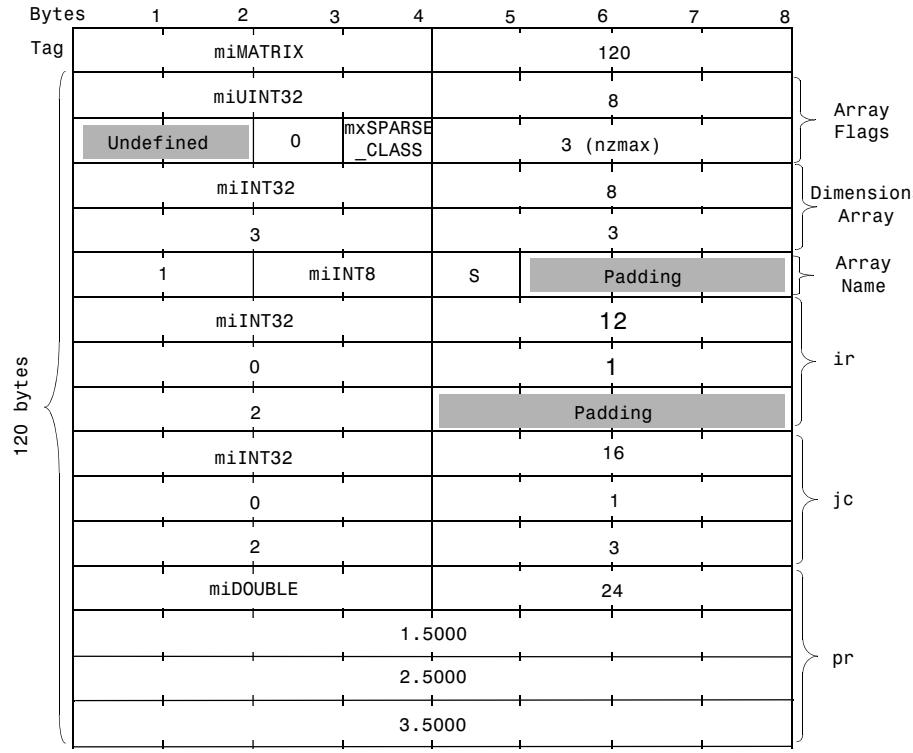
Figure 1-10 illustrates the MAT-file data element format of this 3-by-3 sparse matrix:

```
a = [ 1 2 3 ];
S = sparse(a,a,a+.5)

S =

   (1,1)        1.5000
   (2,2)        2.5000
   (3,3)        3.5000
```

In the figure, note:

- The data element contains six subelements.
- The value of the Number of Bytes field in the data element tag includes all the subelements, but not the eight bytes of the tag itself.
- Bytes 5 through 8 of the Array Flags subelement contain the maximum number of non-zero elements (`nzmax`) in the sparse array.
- The Array Name subelement uses the compressed data element format.

Figure 1-10: Example Sparse Array MAT-file Data Element



Cell Array Data Element Format

A MAT-file data element representing a MATLAB cell array is composed of four subelements. Table 1-5 lists the subelements in the order in which they appear in the data element. The table lists the values of the Data Type and Number of Bytes fields of the tag for each subelement.

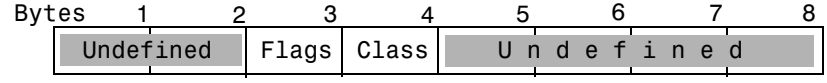
Table 1-5: Cell Array Subelements with Tag Data

Subelement	Data Type	Number of Bytes
Array Flags	`miUINT32`	`2*size-of-Data-Type (8 bytes)`
Dimensions Array	`miINT32`	`number-of-dimensions*size-of-Data-Type`
Array Name	`miINT8`	`number-of-characters*size-of-Data-Type`
Cells	Each cell is written in place as an `miMATRIX` element.	

Array Flags Subelement

This subelement identifies the MATLAB array type (class) represented by the data element and provides other information about the array. Figure 1-11 shows the Array Flags format. The Array Flags subelement is common to all array types.

Figure 1-11: Array Flags Format



Flags. See “Flags” on page 1-16 for more information.

Class. This field contains a value that identifies the MATLAB data type represented by the data element. For cell arrays, Class contains the value `1` (`mxCELL_CLASS`). For more information, see “Class” on page 1-16.

Dimensions Array Subelement

This subelement specifies the size of each dimension of the array. This subelement is common to all array types. For more information, see “Dimensions Array Subelement” on page 1-17.

Array Name Subelement

This subelement specifies the name assigned to the array. This subelement is common to all array types. For more information, see “Array Name Subelement” on page 1-18.

Cells Subelement

This subelement contains the value stored in a cell. These values are MATLAB arrays, represented using the `miMATRIX` format specific to the array type: numeric array, sparse array, structure, object or other cell array. See the appropriate section in this document for details about the MAT-file representation of a each of these array types. Cells are written in column-major order.

Example Cell Array

Figure 1-12 illustrates the MAT-file data element format of this cell array:

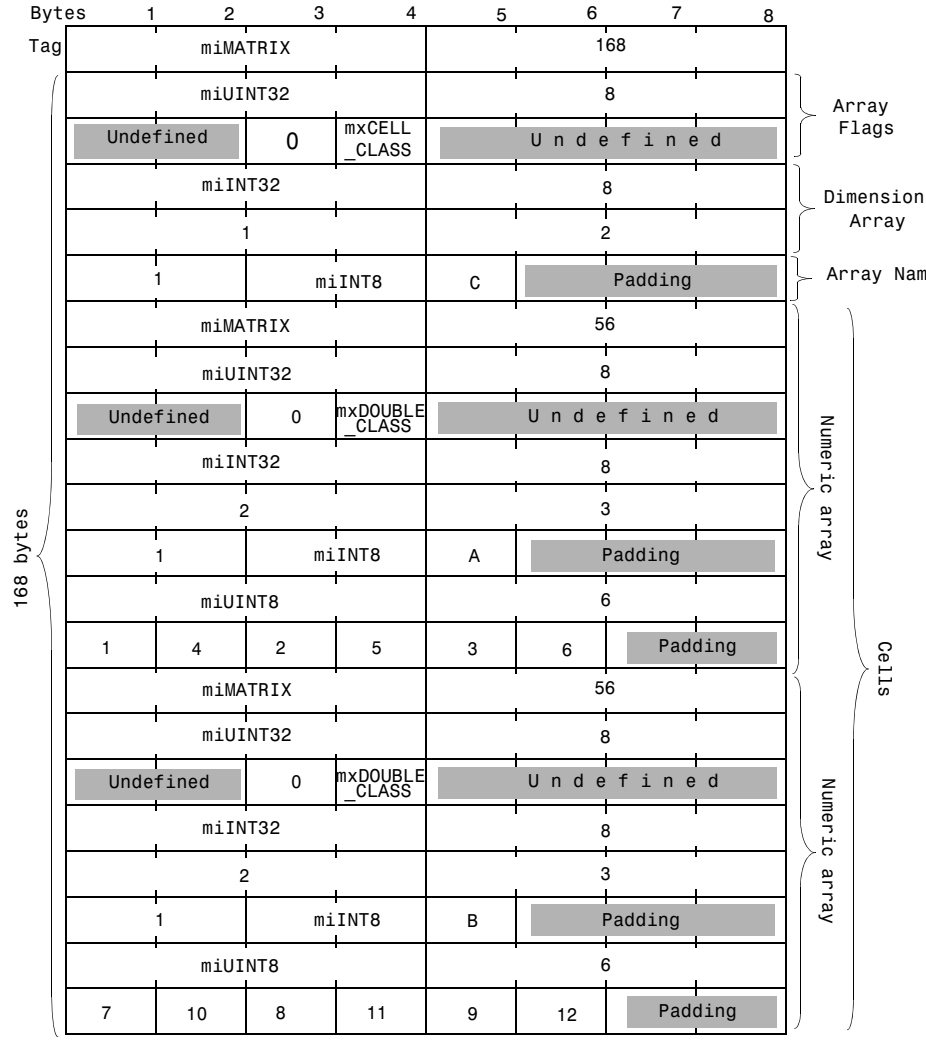
```
A = [ 1 2 3 ; 4 5 6 ]
A =
     1     2     3
     4     5     6

B = [ 7 8 9 ; 10 11 12 ]
B =
     7     8     9
    10    11    12

C = { A, B }
C =
   [2x3 double]     [2x3 double]
```

In the figure, note:

- The data element contains five subelements, the three common subelements; Array Flags, Dimensions and Array Name; and two cell subelements.
- The value of the Number of Bytes field in the data element tag includes all the subelements, but not the eight bytes of the tag itself.
- Each cell subelement is an `miMATRIX` type. In the example, each cell contains a numeric array. For more information about the format of these elements,

see “Numeric Array and Character Array Data Element Formats” on page 1-15.

Figure 1-12: Example Cell Array Data Element



Structure MAT-File Data Element Format

A MAT-file data element representing a MATLAB structure is composed of six subelements. Table 1-6 lists the subelements in the order in which they appear in the data element. The table lists the values of the Data Type and Number of Bytes fields of the tag for each subelement.

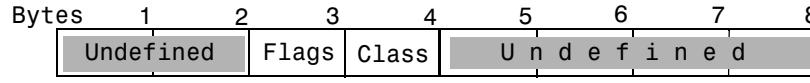
Table 1-6: Structure Subelements with Tag Data

Subelements	Data Type	Number of Bytes
Array Flags	`miUINT32`	`2*size-of-Data-Type (8 bytes)`
Dimensions Array	`miINT32`	`number-of-dimensions*size-of-Data-Type`
Array Name	`miINT8`	`number-of-characters*size-of-Data-Type`
Field Name Length	`miINT32`	`size-of-Data-Type (4 bytes)`
Field Names	`miINT8`	`number-of-fields*Field-Name-Length`
Fields	Each field is written in place as an array. Fields are written in column order.	

Array Flags Subelement

This subelement identifies the MATLAB array type (class) represented by the data element and provides other information about the array. Figure 1-13 shows the Array Flags format. The Array Flags subelement is common to all array types.

Figure 1-13: Array Flags Format



Flags. See “Flags” on page 1-16 for more information.

Class. This field contains a value that identifies the MATLAB data type represented by the data element. For structures, Class contains the value 2 (`mxSTRUCT_CLASS`). For more information, see “Class” on page 1-16.

Dimensions Array Subelement

This subelements Specifies the size of each dimension of the array. This subelement is common to all array types. For more information, see “Dimensions Array Subelement” on page 1-17.

Array Name Subelement

This subelement specifies the name assigned to the structure. This subelement is common to all array types. For more information, see “Array Name Subelement” on page 1-18.

Field Name Length Subelement

This subelement specifies the maximum length of a Field Name. MATLAB sets this limit to 32 (31 characters and a `NULL` terminator). In a MAT-file created by MATLAB, this subelement always uses the compressed data element format.

Field Names Subelement

This subelement specifies the name of each field in the structure as a series of 8-bit (`miINT8`) character arrays. The value of the Field Name Length subelement determines the length of each field name array (32 bytes). Field names must be `NULL`-terminated.

Fields Subelement

This subelement contains the value stored in a field. These values are MATLAB arrays, represented using the `miMATRIX` format specific to the array type: numeric array, sparse array, cell, object or other structure. See the appropriate section of this document for details about the MAT-file format of each of these array type. MATLAB reads and writes these fields in column-major order.

Example

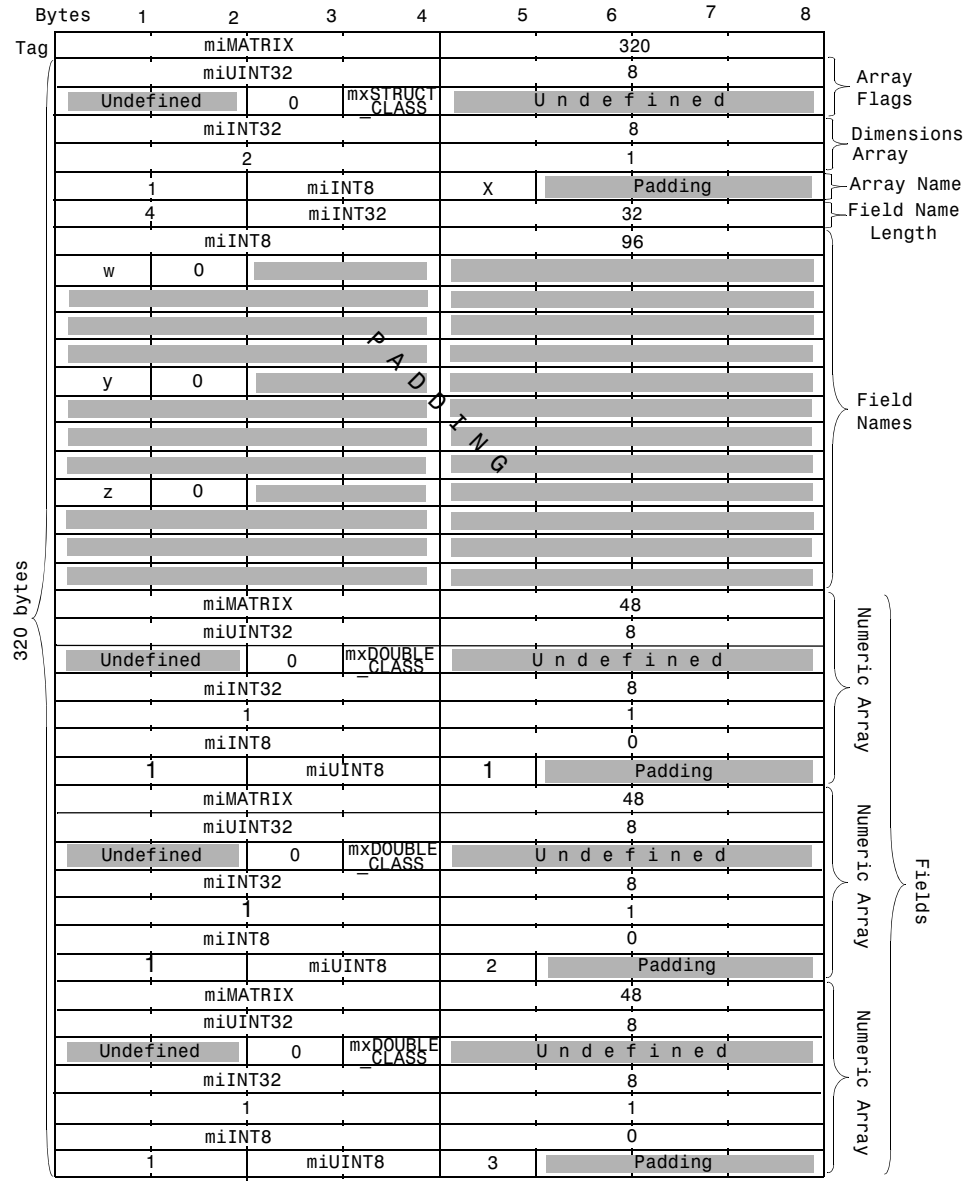
Figure 1-14 illustrates the MAT-file data element format for this MATLAB structure:

```
X.w = [1];
X.y = [2];
X.z = [3];
X
   X =
      w: 1
      y: 2
      z: 3
```

In the figure, note:

- The data element contains eight subelements: the three common subelements (Array Flags, Dimensions and Array Name) and five structure-specific subelements (Field Name Length, Field Names, and three Field subelements).
- The value of the Number of Bytes field in the data element tag includes all the subelements, but not the eight bytes of the tag itself.
- The Field Names subelement allocates 32 bytes of storage for each field name. A `NULL` terminator indicates the end of each field name.
- Each Field subelement is an `miMATRIX` data type. In the example, each field contains a numeric array. For more information about the format of these elements, see “Numeric Array and Character Array Data Element Formats” on page 1-15.
- Each of the numeric arrays contain zero-length Array Name subelements. The Field Names subelement contains the names of the numeric arrays.

Figure 1-14: Example Structure MAT-File Data Element



MATLAB Object MAT-File Data Element Format

A MAT-file data element representing a MATLAB object is composed of seven subelements. Table 1-7 lists the subelements in the order in which they appear in the data element. An object data element has the same subelements as a structure with the addition of the Class Name subelement.The table lists the values of the Data Type and Number of Bytes fields of the tag for each subelement.

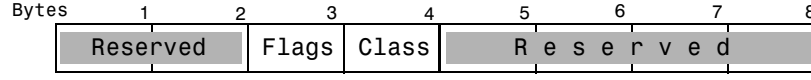
Table 1-7: MATLAB Object Subelements with Tag Data

Subelement	Data Type	Number of Bytes
Array Flags	`miUINT32`	`2*size-of-Data-Type (8 bytes)`
Dimensions Array	`miINT32`	`number-of-dimensions*size-of-Data-Type`
Array Name	`miINT8`	`number-of-characters*size-of-Data-Type`
Class Name	`miINT8`	`number-of-characters*size-of-Data-Type`
Field Name Length	`miINT32`	`size-of-Data-Type (4 bytes)`
Field Names	`miINT8`	`number-of-fields*Field-Name-Length`
Fields	Each field is written in place as an array.	

Array Flags Subelement

This subelement identifies the MATLAB array type (class) represented by the data element and provides other information about the array. Figure 1-15 shows the Array Flags format. The Array Flags subelement is common to all array types.

Figure 1-15: Array Flags Format



Flags. See “Flags” on page 1-16 for more information.

Class. This field contains a value that identifies the MATLAB data type represented by the data element. For objects, the Class byte has the value 3 (`mxOBJECT_CLASS`). For more information, see "Class" on page 1-16.

Dimensions Array Subelement

This subelement specifies the size of each dimension of the array. This subelement is common to all array types. For more information, see "Dimensions Array Subelement" on page 1-17.

Array Name Subelement

This subelement specifies the name assigned to the array. This subelement is common to all array types. For more information, see "Array Name Subelement" on page 1-18.

Class Name Subelement

This subelement specifies the name assigned to the object class. This subelement is an array of 8-bit characters (`miINT8`).

Field Name Length Subelement

This subelement specifies the maximum length of a Field Name. See "Field Name Length Subelement" on page 1-30 for more information.

Field Names Subelement

This subelement specifies the name of each field in the structure. See "Field Names Subelement" on page 1-30 for more information.

Fields Subelement

This subelement contains the value stored in a field. See "Fields Subelement" on page 1-30 for more information.

Example

Figure 1-16 illustrates how the MATLAB object in this example is represented in a MAT-file.

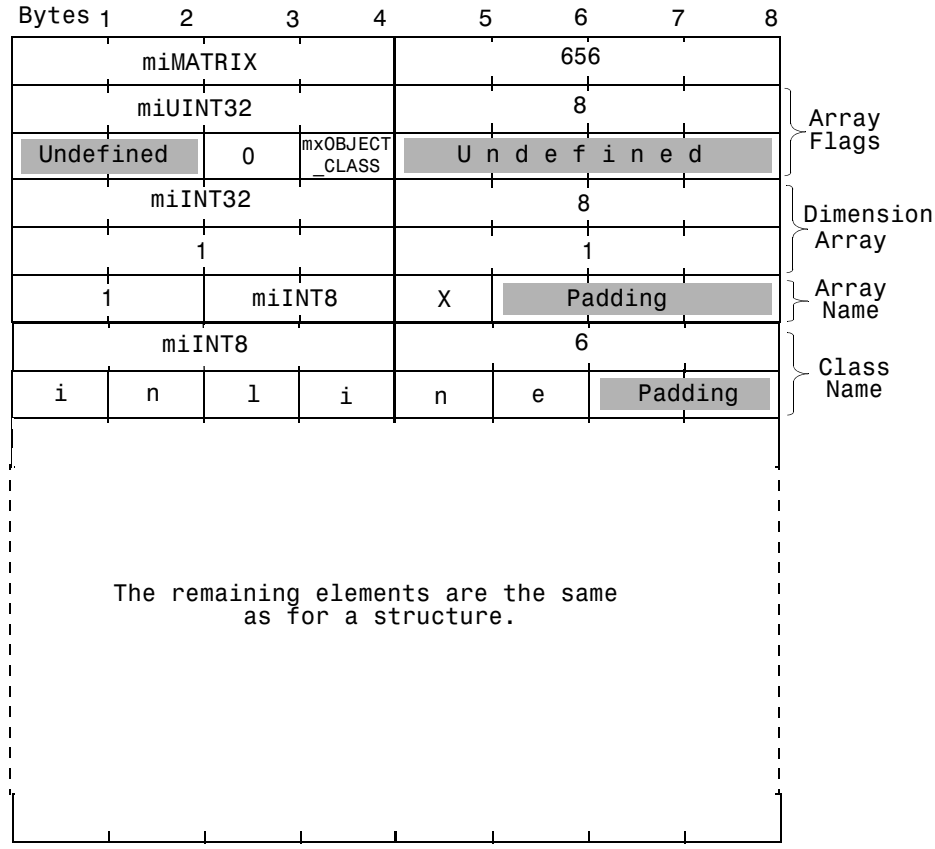
```
X = inline(`t^2');
```

The figure only shows the first four subelements of the object. For an example that shows the remaining subelements, see "Example" on page 1-31.

In the figure, note:

- The Array Flag Class byte is set to `mxOBJECT_CLASS`.
- The data element includes the Class Name subelement.

Figure 1-16: Example Object MAT-file Data Element



Level 4 MAT-File Format

Note This section is taken from the MATLAB V4.2 *External Interface Guide*, which is no longer available in printed form.

This section presents the internal structure of Level 4 MAT-files. This information is provided to enable users to read and write MAT-files on machines for which the MAT-file access routine library is not available. It is not needed when using the MAT-file subroutine library to read and write MAT-files, and we strongly advise that you do use the External Interface Library if it is available for all of the machines that you are working with.

A MAT-file may contain one or more matrices. The matrices are written sequentially on disk, with the bytes forming a continuous stream. Each matrix starts with a fixed-length 20-byte header that contains information describing certain attributes of the Matrix. The 20-byte header consists of five long (4-byte) integers:

Table 1-8: Level 4 MAT-File Matrix Header Format

Field	Description	
type	The `type` flag contains an integer whose decimal digits encode storage information. If the integer is represented as `MOPT` where `M` is the thousands digit, `O` is the hundreds digit, `P` is the tens digit, and `T` is the ones digit, then:	
	`M` indicates the numeric format of binary numbers on the machine that wrote the file. Use this table to determine the number to use for your machine:	
	0	IEEE Little Endian (PC, 386, 486, DEC Risc)
	1	IEEE Big Endian (Macintosh, SPARC, Apollo,SGI, HP 9000/300, other Motorola)
	2	VAX D-float
	3	VAX G-float
	4	Cray

Table 1-8: Level 4 MAT-File Matrix Header Format (Continued)

	`O` is always 0 (zero) and is reserved for future use.	
	`P` indicates which format the data is stored in according to the following table:	
	0	double-precision (64-bit) floating point numbers
	1	single-precision (32-bit) floating point numbers
	2	32-bit signed integers
	3	16-bit signed integers
	4	16-bit unsigned integers
	5	8-bit unsigned integers
	The precision used by the `save` command depends on the size and type of each matrix. Matrices with any noninteger entries and matrices with 10,000 or fewer elements are saved in floating point formats requiring 8 bytes per real element. Matrices with all integer entries and more than 10,000 elements are saved in the following formats, requiring fewer bytes per element.	
	Element range	Bytes per element
	[0:255]	1
	[0:65535]	2
	[-32767:32767]	2
	[-2^31+1:2^31-1]	4
	other	8
	`T` indicates the matrix type according to the following table:	
	0	Numeric (Full) matrix
	1	Text matrix
	2	Sparse matrix
	Note that the elements of a text matrix are stored as floating point numbers between 0 and 255 representing ASCII-encoded characters.	

Table 1-8: Level 4 MAT-File Matrix Header Format (Continued)

mrows	The row dimension contains an integer with the number of rows in the matrix.
`ncols`	The column dimension contains an integer with the number of columns in the matrix.
imagf	The imaginary flag is an integer whose value is either 0 or 1. If 1, then the matrix has an imaginary part. If 0, there is only real data.
namlen	The name length contains an integer with 1 plus the length of the matrix name.

Immediately following the fixed length header is the data whose length is dependent on the variables in the fixed length header:

Table 1-9: Level 4 MAT-File Matrix Data Format

Field	Description
name	The matrix name consists of `namlen` ASCII bytes, the last one of which must be a `null` character (`'\0'`).
real	Real part of the matrix consists of `mrows` * `ncols` numbers in the format specified by the `P` element of the type flag. The data is stored column-wise such that the second column follows the first column, etc.
imag	Imaginary part of the matrix, if any. If the imaginary flag `imagf` is nonzero, the imaginary part of a matrix is placed here. It is stored in the same manner as the real data.

This structure is repeated for each matrix stored in the file.

The following C language code demonstrates how to write a single matrix to disk in Level 1.0 MAT-file format.

```
#include <stdio.h>

main() {
typedef struct {
   long type;
   long mrows;
   long ncols;
   long imagf;
```

```
   long namelen;
   } Fmatrix;

char *pname;
double *pr;
double *pi;
Fmatrix x;
int mn;
FILE *fp;

double real_data = 1.0;
double imag_data = 2.0;

fp = fopen("mymatfile.mat", "wb");
if (fp != NULL) {
   pname = "x";
   x.type = 1000;
   x.mrows = 1;
   x.ncols = 1;
   x.imagf = 1;
   x.namelen = 2;

   pr = &real_data;
   pi = &imag_data;

   fwrite(&x, sizeof(Fmatrix), 1, fp);
   fwrite(pname, sizeof(char), x.namelen, fp);

   mn = x.mrows *x.ncols;
   fwrite(pr, sizeof(double), mn, fp);

   if(x.imagf)
      fwrite(pi, sizeof(double), mn, fp);
   }

else
   printf("File could not be opened.\n");

fclose(fp);
}
```

Again, we strongly advise against using this approach, and recommend that you instead use the MAT-file access routines provided in the External Interface Library. You will need to write your own C code as shown above only if you do not have the MAT-file access routines for the particular platform on which you need to read and write MAT-files.

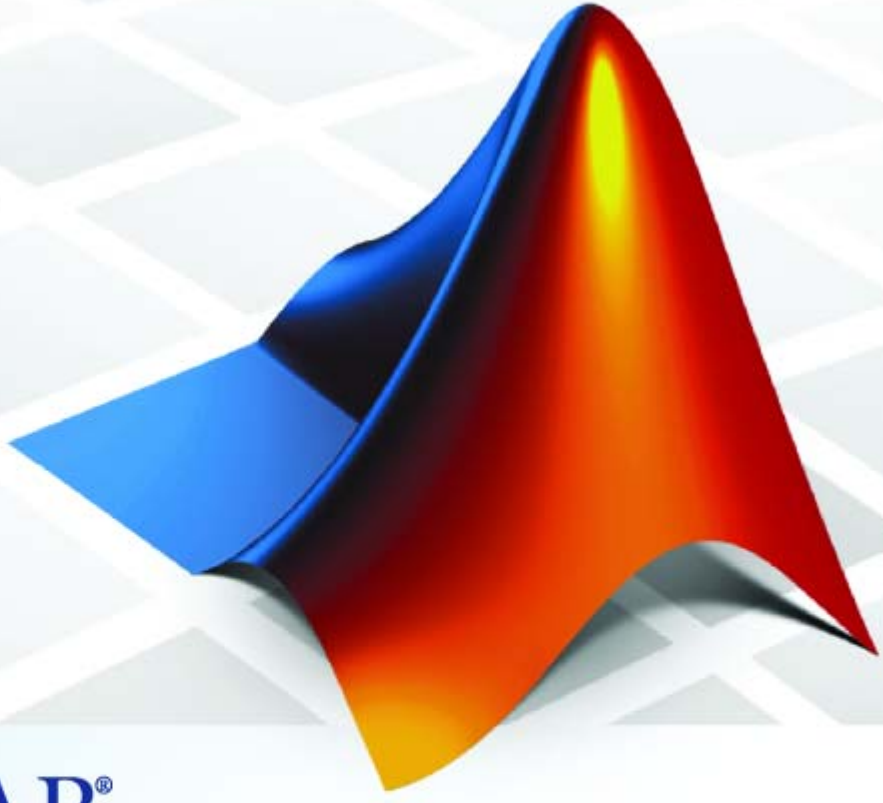
Index

T

U

V

MATLAB® 7
Mathematics



MATLAB®

The MathWorks™
Accelerating the pace of engineering and science

How to Contact The MathWorks

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

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

508-647-7000 (Phone)

508-647-7001 (Fax)

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

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

MATLAB Mathematics

Revision History

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

Contents

1 Matrices and Linear Algebra

2 Polynomials and Interpolation

3 Fast Fourier Transform (FFT)

4 Function Functions

5 Differential Equations

6 Sparse Matrices

1

Matrices and Linear Algebra

Function Summary

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

There are three columns in this table, but only four entries in the first column. The entries in the first column divide the functions into the categories of matrix analysis, linear equations, eigenvalues and singular values, and matrix functions.

Function Summary

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

Function Summary (Continued)

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

Function Summary (Continued)

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

Matrices in MATLAB

In this section...

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
```

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

Adding and Subtracting Matrices

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

```
A = pascal(3);
B = magic(3);
X = A + B

X =
       9     2     7
       4     7    10
       5    12     8

Y = X - A

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

Addition and subtraction require both matrices to have the same dimension, or one of them be a scalar. If the dimensions are incompatible, an error results:

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

w = v + s

w =
       9     7     6
```

Vector Products and Transpose

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

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

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

x =
       2

X = u*v

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

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

The example matrix `A` is *symmetric*, so `A'` is equal to `A`. But `B` is not symmetric:

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

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

Transposition turns a row vector into a column vector:

```
x = v'

x =
       2
       0
      -1
```

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

```
x'*y
```

and

```
y'*x
```

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

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

```
z = [1+2i 7-3i 3+4i; 6-2i 9i 4+7i]
z =
   1.0000 + 2.0000i   7.0000 - 3.0000i   3.0000 + 4.0000i
   6.0000 - 2.0000i        0 + 9.0000i   4.0000 + 7.0000i
```

then

```
z'
ans =
   1.0000 - 2.0000i   6.0000 + 2.0000i
   7.0000 + 3.0000i        0 - 9.0000i
   3.0000 - 4.0000i   4.0000 - 7.0000i
```

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

```
z.'
ans =
   1.0000 + 2.0000i   6.0000 - 2.0000i
   7.0000 - 3.0000i        0 + 9.0000i
   3.0000 + 4.0000i   4.0000 + 7.0000i
```

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

Multiplying Matrices

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

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

MATLAB uses a single asterisk to denote matrix multiplication. The next two examples illustrate the fact that matrix multiplication is not commutative; AB is usually not equal to BA:

```
X = A*B

X =
      15    15    15
      26    38    26
      41    70    39

Y = B*A

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

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

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

```
x = A*u

x =
       8
      17
      30

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

y =
      12    -7    10
```

Rectangular matrix multiplications must satisfy the dimension compatibility conditions:

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

X =
      17    19
      31    41
      51    70

Y = C*A

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

Anything can be multiplied by a scalar:

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

w =
      14     0    -7
```

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 uppercase 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

In this section...

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 $7^{-1} = 0.142857...$, 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 1-27.

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

```
pinv(A)*b

ans =
    0.3850
   -0.1103
    0.7066
```

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

t	y
1.1	0.60
1.6	0.55
2.3	0.50

Enter the data into MATLAB with the statements

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

Try modeling the data with a decaying exponential function:

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

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

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

E =
      1.0000    1.0000
      1.0000    0.7408
      1.0000    0.4493
      1.0000    0.3329
      1.0000    0.2019
      1.0000    0.1003
```

Use the backslash operator to get the least squares solution:

```
c = E\y

c =
      0.4760
```

```
    0.3413
```

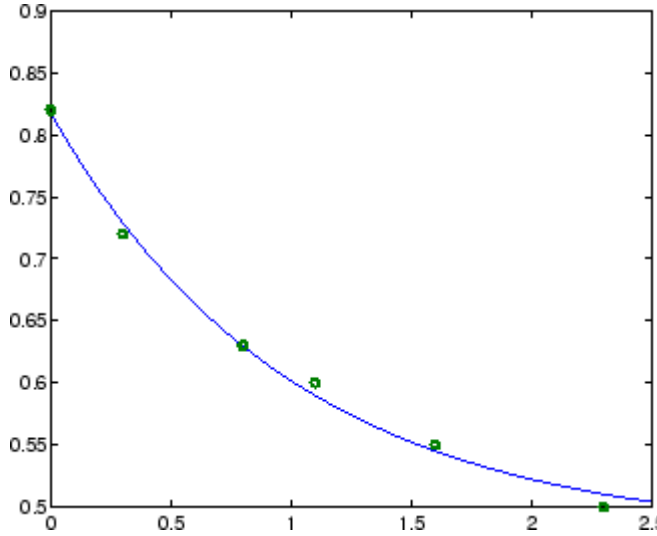
In other words, the least squares fit to the data is

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

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

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

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

In this section...

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

In this section...

About Matrix Factorizations

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

In this section...

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
```

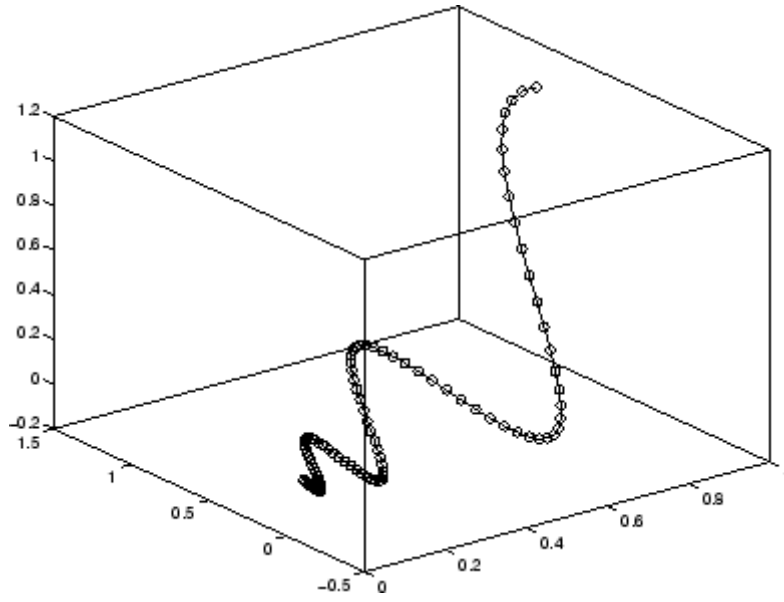
The matrix exponential is used to compute the solution, $x(t)$, to the differential equation at 101 points on the interval $0 \le t \le 1$ with

```
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

In this section...

Eigenvalue Decomposition

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

$$Av = \lambda v$$

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

$$AV = V\Lambda$$

If V is nonsingular, this becomes the eigenvalue decomposition

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

A good example is provided by the coefficient matrix of the ordinary differential equation in the previous section:

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

The statement

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

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

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

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

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

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

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

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

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

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

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

In this section...

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

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];
```

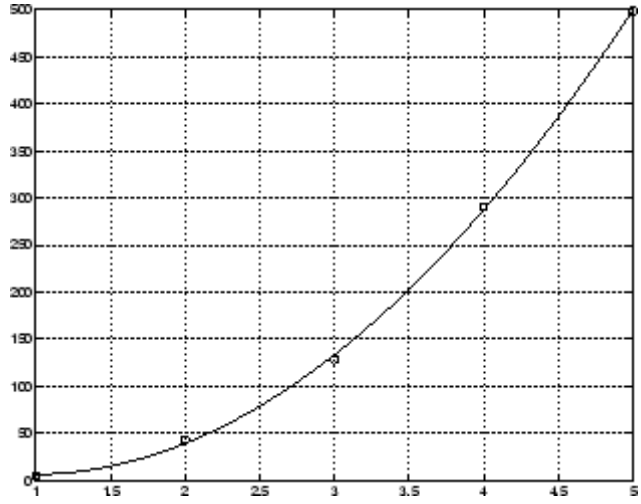
A third degree polynomial that approximately fits the data is

```
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 “Linear Regression Analysis” in MATLAB Data Analysis.

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

In this section...

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)

Interpolation Function Summary (Continued)

Function	Description
`interpft`	One-dimensional interpolation using FFT method
`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" on page 2-10
- "FFT-Based Interpolation" on page 2-12

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 nonuniform and some points are much closer together than others.

- Cubic interpolation requires more memory and execution time than either the nearest neighbor or linear methods. However, both the interpolated data and its derivative are continuous.

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.

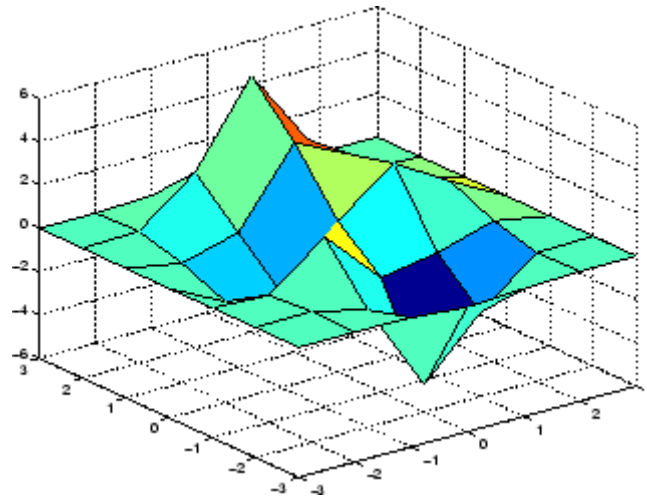
All of these methods require that `X` and `Y` be monotonic, that is, either always increasing or always decreasing from point to point. You should prepare these matrices using the `meshgrid` function, or else be sure that the "pattern" of the points emulates the output of `meshgrid`. In addition, each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

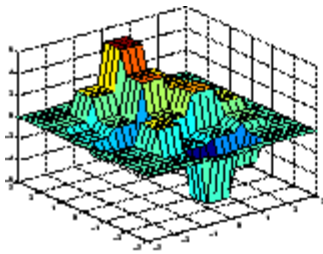
4 Interpolate using bilinear interpolation:

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

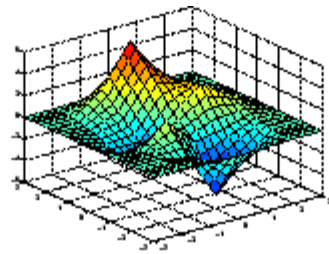
5 Interpolate using bicubic interpolation:

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

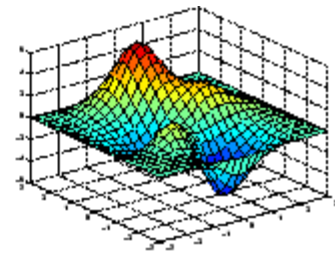
6 Compare the surface plots for the different interpolation methods.



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

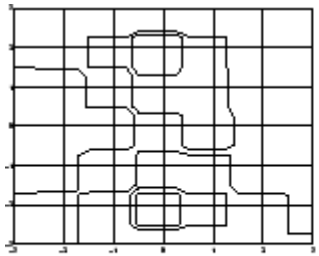


```
surf(xi,yi,zi2)
% bilinear
```

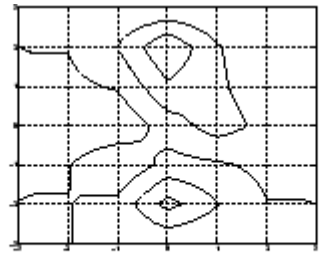


```
surf(xi,yi,zi3)
% bicubic
```

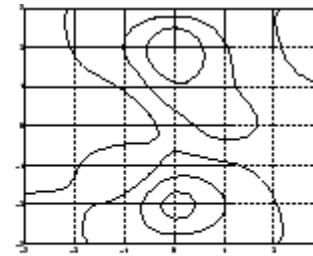
7 Compare the contour plots for the different interpolation methods.



```
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

The function `interp3` performs three-dimensional interpolation, finding interpolated values between points of a three-dimensional set of samples `V`. You must specify a set of known data points:

- `X`, `Y`, and `Z` matrices specify the points for which values of `V` are given.
- A matrix `V` contains values corresponding to the points in `X`, `Y`, and `Z`.

The most general form for `interp3` is

```
VI = interp3(X,Y,Z,V,XI,YI,ZI,method)
```

`XI`, `YI`, and `ZI` are the points at which `interp3` interpolates values of `V`. For out-of-range values, `interp3` returns `NaN`.

There are three different interpolation methods for three-dimensional data:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method chooses the value of the nearest point.
- *Trilinear interpolation* (`method = 'linear'`). This method uses piecewise linear interpolation based on the values of the nearest eight points.

- *Tricubic interpolation* (`method = 'cubic'`). This method uses piecewise cubic interpolation based on the values of the nearest sixty-four points.

All of these methods require that `X`, `Y`, and `Z` be *monotonic*, that is, either always increasing or always decreasing in a particular direction. In addition, you should prepare these matrices using the `meshgrid` function, or else be sure that the "pattern" of the points emulates the output of `meshgrid`.

Each method automatically maps the input to an equally spaced domain before interpolating. If `x` is already equally spaced, you can speed execution time by prepending an asterisk to the `method` string, for example, `'*cubic'`.

Interpolation of Higher Dimensional Data

The function `interpn` performs multidimensional interpolation, finding interpolated values between points of a multidimensional set of samples `V`. The most general form for `interpn` is

```
VI = interpn(X1,X2,X3...,V,Y1,Y2,Y3,...,method)
```

`1`, `2`, `3`, `...` are matrices that specify the points for which values of `V` are given. `V` is a matrix that contains the values corresponding to these points. `1`, `2`, `3`, `...` are the points for which `interpn` returns interpolated values of `V`. For out-of-range values, `interpn` returns `NaN`.

`Y1`, `Y2`, `Y3`, `...` must be either arrays of the same size, or vectors. If they are vectors of different sizes, `interpn` passes them to `ndgrid` and then uses the resulting arrays.

There are three different interpolation methods for multidimensional data:

- *Nearest neighbor interpolation* (`method = 'nearest'`). This method chooses the value of the nearest point.
- *Linear interpolation* (`method = 'linear'`). This method uses piecewise linear interpolation based on the values of the nearest two points in each dimension.
- *Cubic interpolation* (`method = 'cubic'`). This method uses piecewise cubic interpolation based on the values of the nearest four points in each dimension.

All of these methods require that `X1`, `X2`,`X3` be monotonic. In addition, you should prepare these matrices using the `ndgrid` function, or else be sure that the "pattern" of the points emulates the output of `ndgrid`.

Each method automatically maps the input to an equally spaced domain before interpolating. If `X` is already equally spaced, you can speed execution time by prepending an asterisk to the `method` string; for example, `'*cubic'`.

Multidimensional Data Gridding

The `ndgrid` function generates arrays of data for multidimensional function evaluation and interpolation. `ndgrid` transforms the domain specified by a series of input vectors into a series of output arrays. The `i`th dimension of these output arrays are copies of the elements of input vector x_i.

The syntax for `ndgrid` is

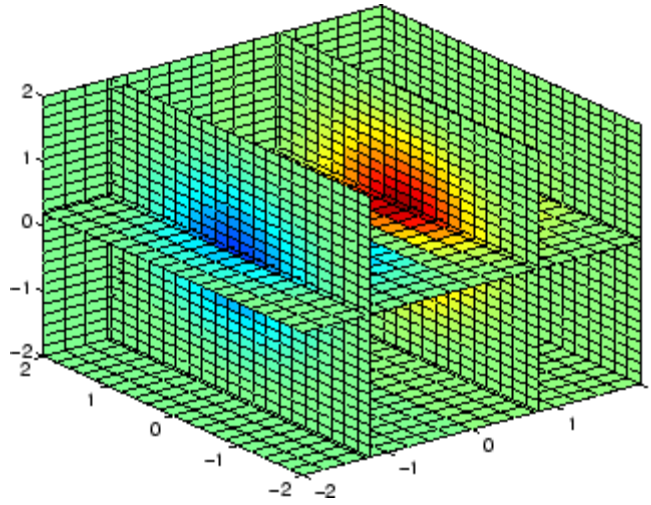
```
[X1,X2,X3,...] = ndgrid(x1,x2,x3,...)
```

For example, assume that you want to evaluate a function of three variables over a given range. Consider the function

$$z = x_2 e^{(-x_1^2 - x_2^2 - x_3^2)}$$

for $-2\pi \le x_1 \le 0$, $2\pi \le x_2 \le 4\pi$, and $0 \le x_3 \le 2\pi$. To evaluate and plot this function,

```
x1 = -2:0.2:2;
x2 = -2:0.25:2;
x3 = -2:0.16:2;
[X1,X2,X3] = ndgrid(x1,x2,x3);
z = X2.*exp(-X1.^2 -X2.^2 -X3.^2);
slice(X2,X1,X3,z,[-1.2 0.8 2],2,[-2 0.2])
```



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
`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

Functions for Analysis of Closest-Point Problems and Geometric Analysis (Continued)

Function	Description
`tsearch`	Closest triangle search
`voronoi`	Voronoi diagram

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

- "Convex Hulls" on page 2-20
- "Delaunay Triangulation" on page 2-21
- "Voronoi Diagrams" on page 2-26

See also "Tessellation and Interpolation of Scattered Data in Higher Dimensions" on page 2-27.

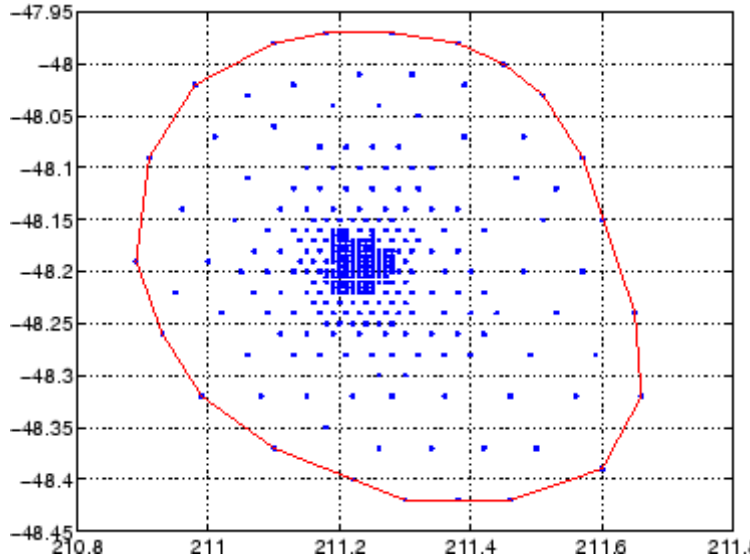
Note Examples in this section use the MATLAB `seamount` data set. Seamounts are underwater mountains. They are valuable sources of information about marine geology. The `seamount` data set represents the surface, in 1984, of the seamount designated LR148.8W located at 48.2°S, 148.8°W on the Louisville 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

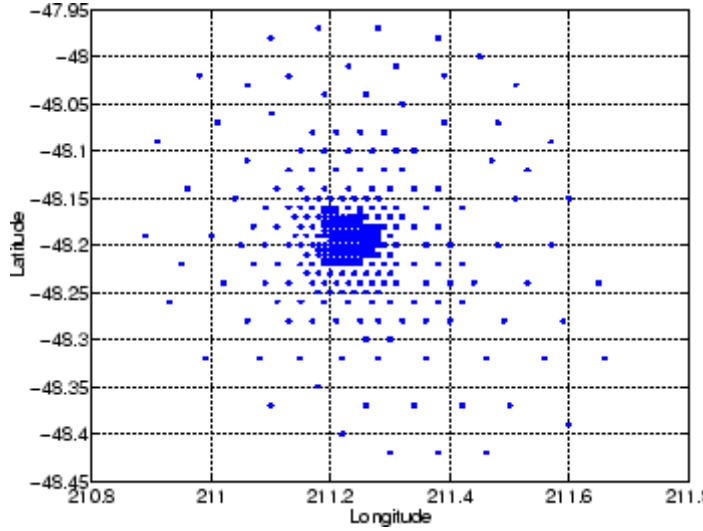
Given a set of coplanar data points, *Delaunay triangulation* is a set of lines connecting each point to its natural neighbors. The `delaunay` function returns a Delaunay triangulation as a set of triangles having the property that, for each triangle, the unique circle circumscribed about the triangle 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:

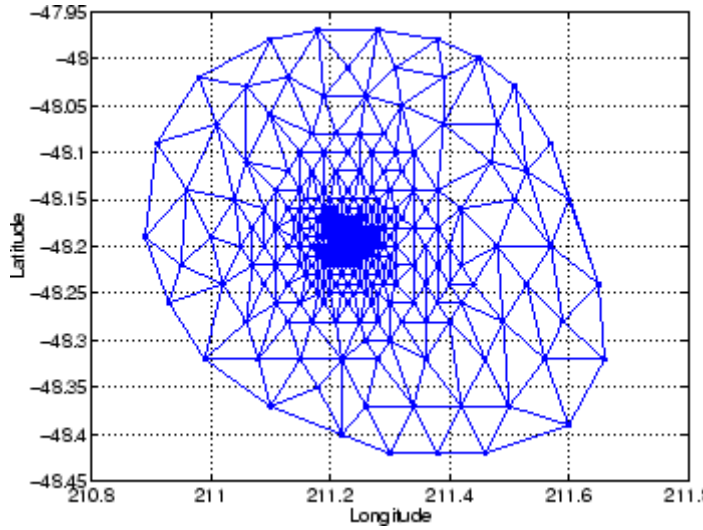
```
load seamount
```

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



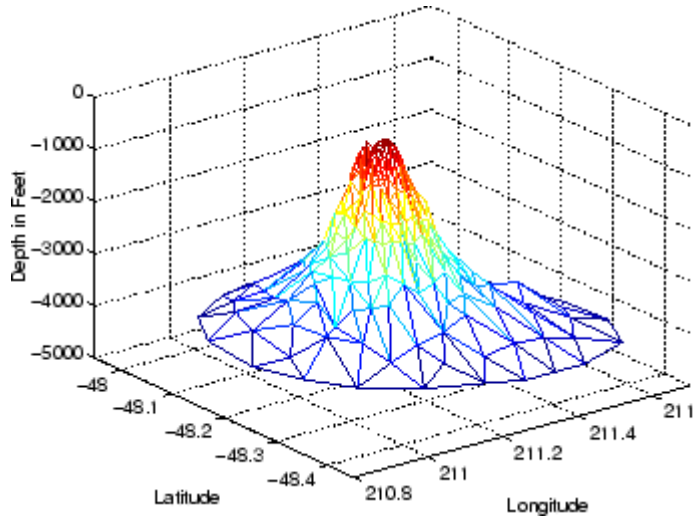
Apply Delaunay triangulation and use `triplot` to overplot the resulting triangles on the scatter plot:

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



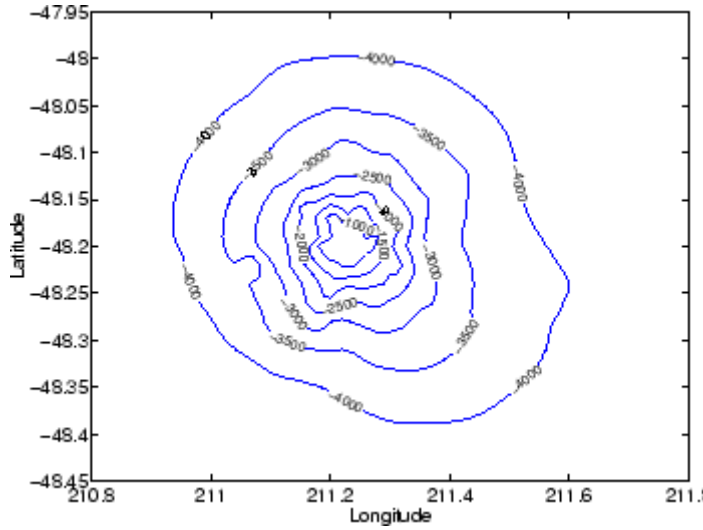
Mesh and Surface Plots. Add the depth data (`z`) from `seamount`, to the Delaunay triangulation, and use `trimesh` to produce a mesh in three-dimensional space. Similarly, you can use `trisurf` to produce a surface:

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

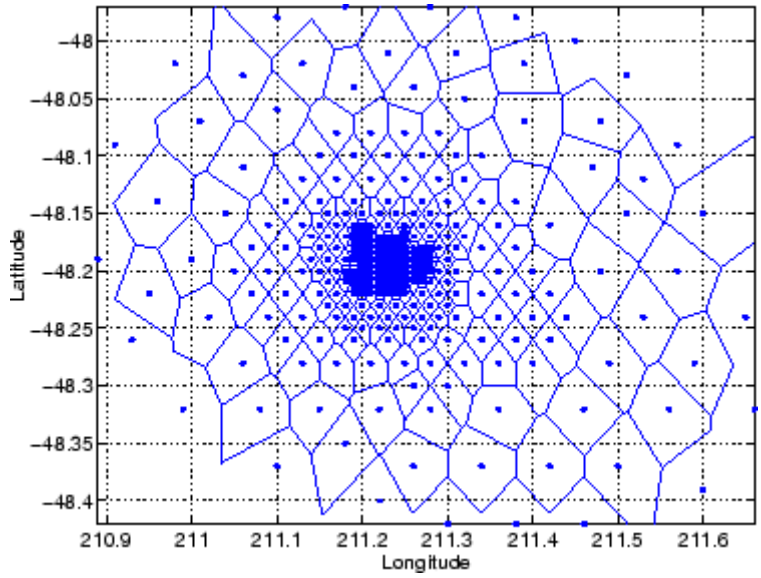
Voronoi Diagrams

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

Functions for Multidimensional Geometrical Analysis (Continued)

Function	Description
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

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
```

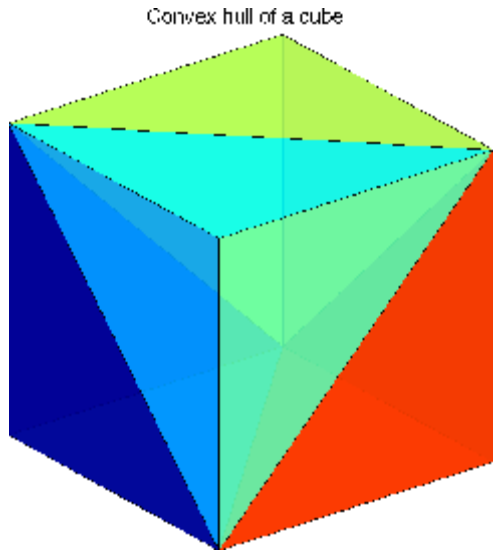
```
     6     5     1
     4     6     2
     6     4     8
     6     7     5
     7     6     8
```

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     3     9     1
```

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

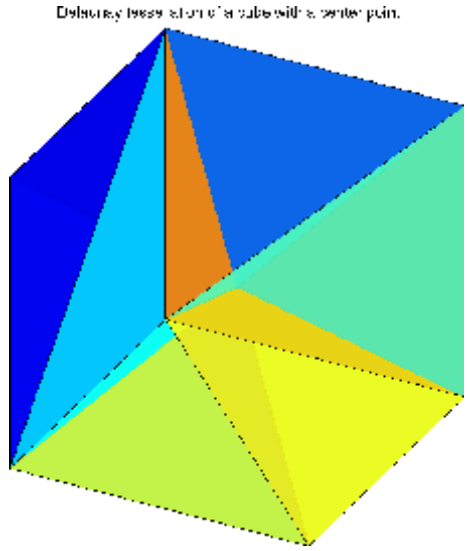
The 12 rows of `T` represent the 12 simplices, in this case irregular tetrahedrons, that partition the cube. Each row represents one tetrahedron, and the row elements are indices of points in `X`.

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.

rand('twister', 5489);     % Initialize the random number generator.
X = X+0.01*rand(size(X));  % Make the cube less regular.

[V,C] = voronoin(X)

V =
       Inf       Inf       Inf
    0.0029   -1.4858    0.0079
    0.1702   -0.0719  193.1848
    0.0018    0.0089    1.5064
    0.0067    0.0040    1.5064
    0.0060    2.9397    0.0073
    0.0033    1.5095    0.0119
    0.0117    1.5095    0.0035
   -1.4873    0.0055    0.0107
   -1.6607    0.0051    0.0100
   -1.4873    0.0050    0.0101
    0.0054   -1.4858    0.0054
    4.6864   -0.0017    0.0080
    1.5105    0.0107    0.0022
    1.5105    0.0025    0.0104
    0.0108    0.0120   -1.4850
    0.0032    0.0070   -3.1326
    0.0043    0.0056   -1.4849

C =
    [1x7  double]
    [1x10 double]
    [1x8  double]
    [1x7  double]
    [1x8  double]
```

```
    [1x7  double]
    [1x7  double]
    [1x10 double]
    [1x12 double]
```

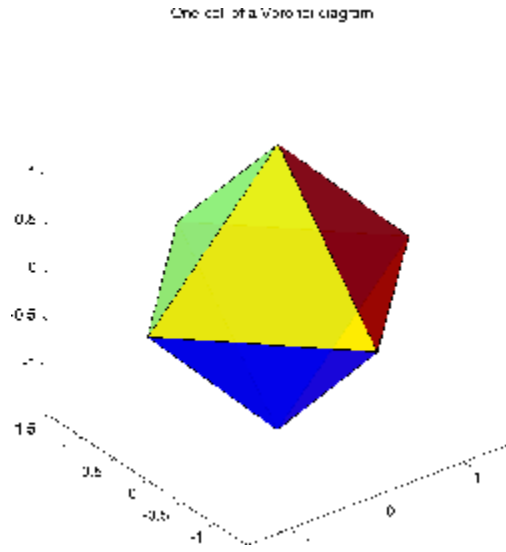
In this example, `V` is a 13-by-3 matrix, the 13 rows are the coordinates of the 13 Voronoi vertices. The first row of `V` is a point at infinity. `C` is a 9-by-1 cell array, where each cell in the array contains an index vector into `V` corresponding to one of the 9 Voronoi cells. For example, the 9th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

```
C{9} = 2   4   5   7   8   9  11  12  14  15  16  18
```

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:

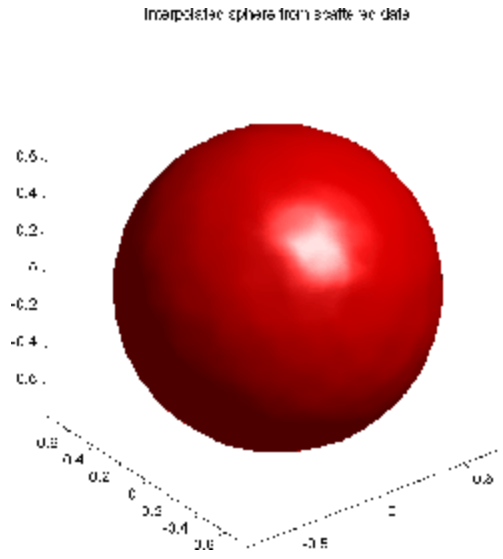
```
delta = 0.05;
```

```
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.



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)

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

In this section...

Finding an FFT

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

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

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`.

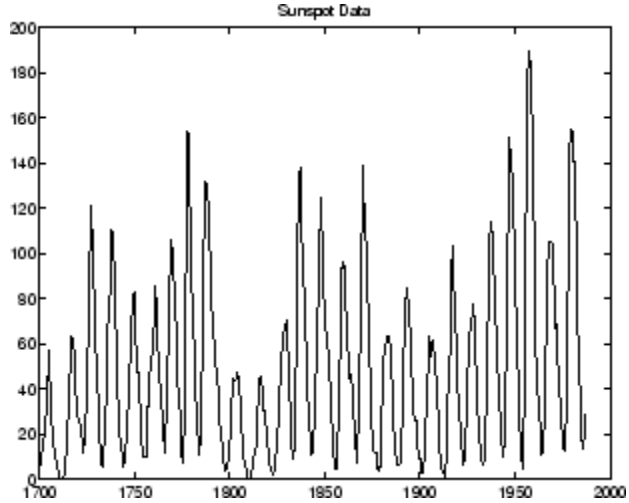
Example: Using FFT to Calculate Sunspot Periodicity

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')
```

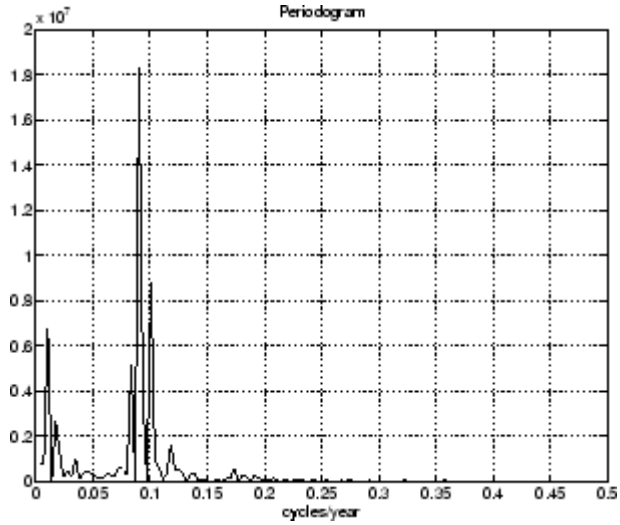


Now take the FFT of the sunspot data:

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

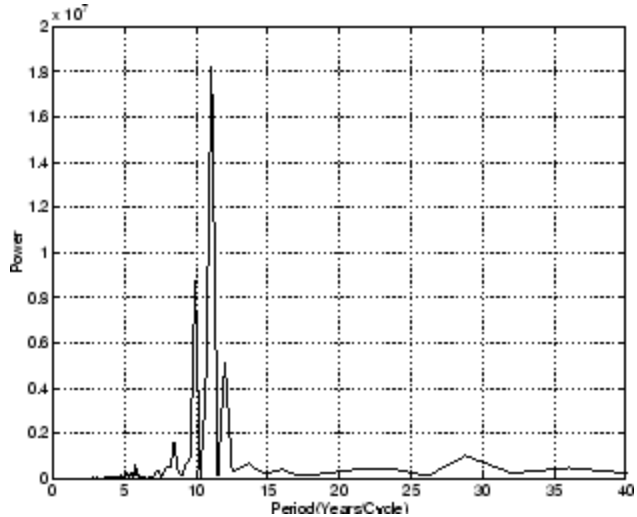
The result of this transform is the complex vector, `Y`. The magnitude of `Y` squared is called the *power* and a plot of power versus frequency is a *periodogram*. Remove the first component of `Y`, which is simply the sum of the data, and plot the results:

```
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);
```

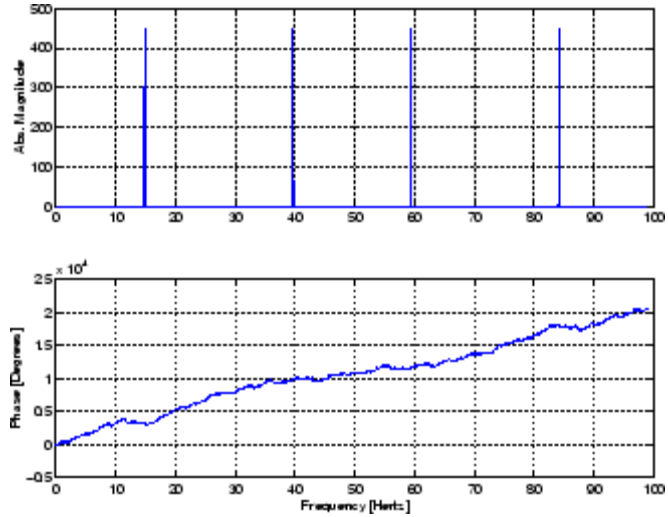
Now use the `fft` function to compute the DFT of the sequence. The code below calculates the magnitude and phase of the transformed sequence. It uses the `abs` function to obtain the magnitude of the data, the `angle` function to obtain the phase information, and `unwrap` to remove phase jumps greater 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
500
400
300
200
100
0
0 10 20 30 40 50 60 70 80 90 100
x 10^4
Phase [Degrees]
2.5
2
1.5
1
0.5
0
-0.5
0 10 20 30 40 50 60 70 80 90 100
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, “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

See the Chapter 5, “Differential Equations” and Chapter 6, “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 .

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.

This is a three column by eight row table. The entries in the first column span several rows. The first entry, Plotting, has only one function, `fplot`. The second entry, Optimization and zero finding, covers the functions `fminbnd`, `fminsearch`, and `fzero`. The third entry, numerical integration, covers the functions `quad`, `quad1`, `dblquad`, and `triplequad`.

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

Function Summary (Continued)

Category	Function	Description
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

In this section...

MATLAB Functions

MATLAB can represent mathematical functions by expressing them as MATLAB functions in M-files or as "Anonymous Functions" in . 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.

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
```

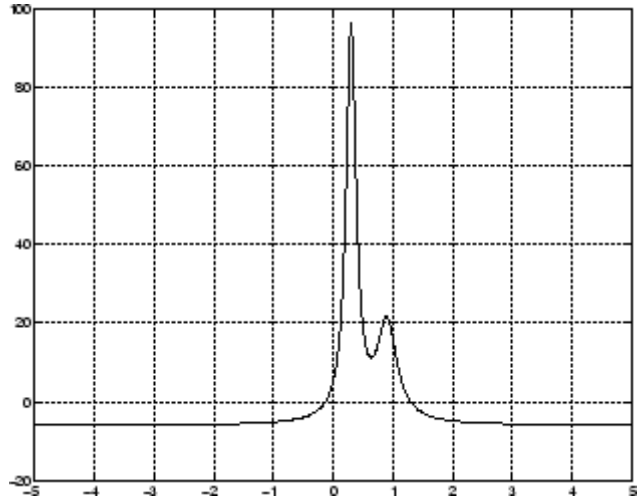
You can also create anonymous functions of more than one argument. The following function has two input arguments `x` and `y`.

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

Plotting Mathematical Functions

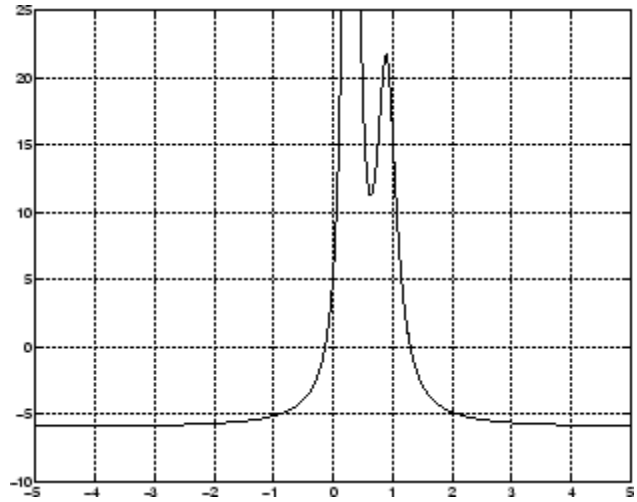
The `fplot` function plots a mathematical function between a given set of axes limits. You can control the *x*-axis limits only, or both the *x*- and *y*-axis limits. For example, to plot the `humps` function over the *x*-axis range `[-5 5]`, use

```
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

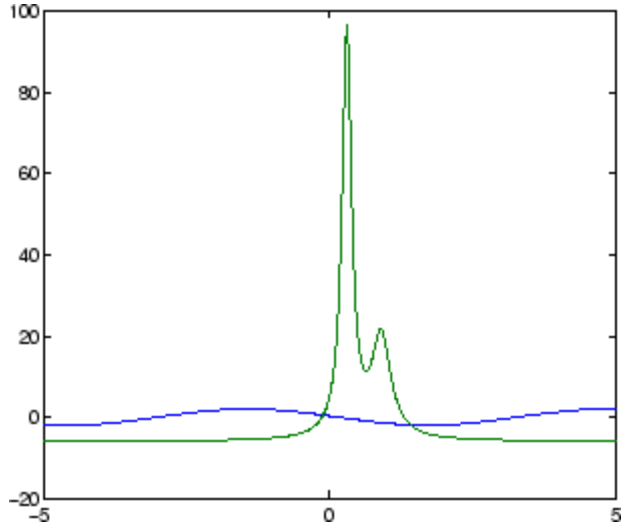
```
fplot(@(x)2*sin(x+3),[-1 1]);
```

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

In this section...

MATLAB Optimization Functions

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 User's Guide*.

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.

• "Plotting the Results" on page 4-13

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
```

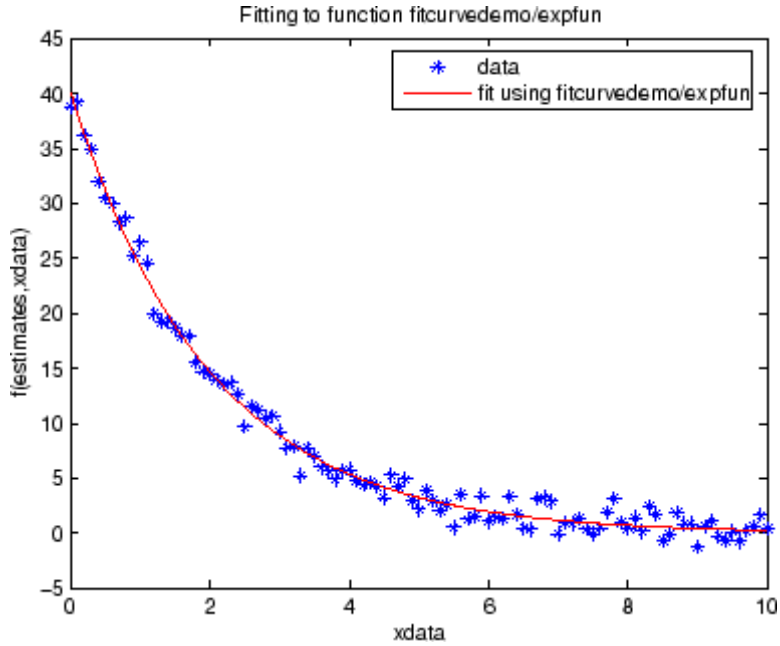
and a function handle, `model`, to the function that computes the exponential function `A*exp(-lambda*t)`.

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 an input to the optimization function, for example, by calling `fminbnd` with the syntax

```
x = fminbnd(fun,x1,x2,options)
```

or `fminsearch` with the syntax

```
x = fminsearch(fun,x0,options)
```

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

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`.

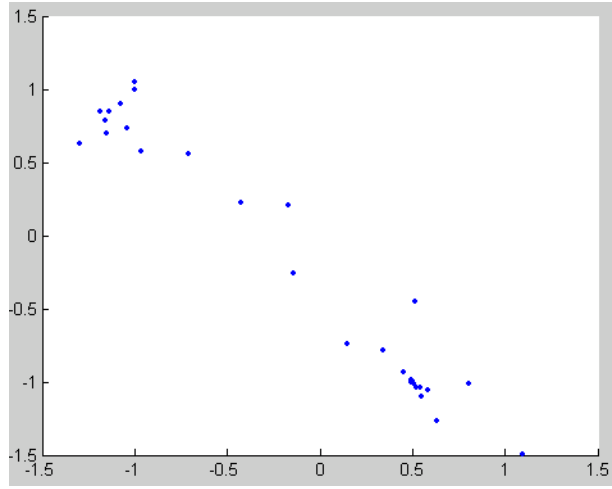
3 Enter the following commands:

```
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 4-22.
- `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 4-20 describes the structure in detail.
- `state` is the current state of the algorithm. "States of the Algorithm" on page 4-21 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 4-16 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 MATLAB Programming 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,:) =

   -1.0000    1.0000
   -1.0000    1.0000
   -1.0750    0.9000
   -1.0125    0.8500
```

The final row of points is the same as the optimal point, `x`.

```
history(end,:)

ans =

    0.1290   -0.5323

objfun(history(end,:))

ans =

   -0.5689
```

Fields in optimValues

The following table lists the fields of the `optimValues` structure that are provided by all three optimization functions, `fminbnd`, `fminsearch`, and `fzero`. The 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
```

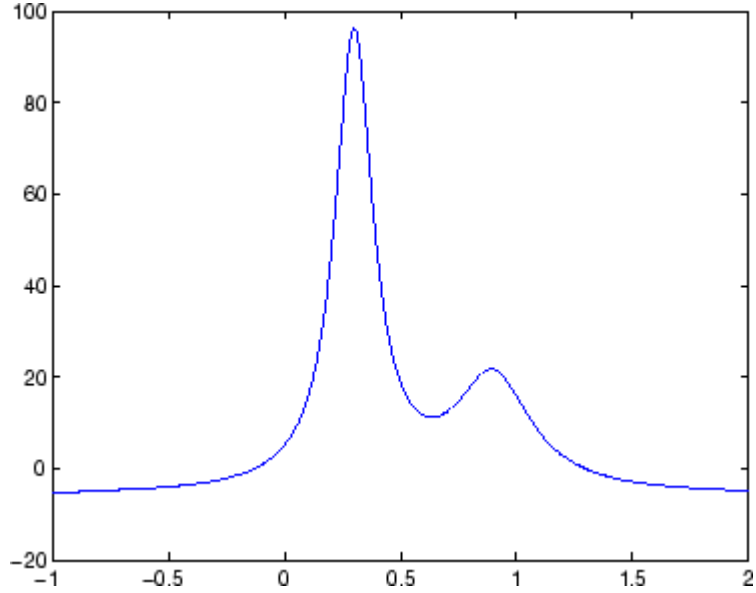
Stopping an Optimization Based on GUI Input. If you design a GUI to perform optimizations, you can make the output function stop an optimization when a user clicks a **Stop** button on the GUI. The following code shows how to do this, assuming that the **Stop** button callback stores the value `true` 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 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 is shown in the output. At the tenth step, the sign of `humps` is positive at the endpoint, `-0.10949`, but negative at the endpoint, `-0.264`. From this point on, the algorithm continues to narrow down the interval `[-0.10949 -0.264]`, as described in the previous section, until it reaches the value `-0.1316`.

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.

Problem	Recommendation
Sometimes an optimization problem has values of `x` for which it is impossible to evaluate `f`.	Modify your function to include a penalty function to give a large positive value to `f` when infeasibility is encountered.
The minimization routine appears to enter an infinite loop or returns a solution that is not a minimum (or not a zero in the case of `fzero`).	Your objective function (`fun`) may be returning `NaN` or complex values. The optimization routines expect only real numbers to be returned. Any other values may cause unexpected results. To determine whether this is the case, set `options = optimset('FunValCheck', 'on')` and call the optimization function with `options` as an input argument. This displays an error when the objective function returns `NaN` or complex values.

Numerical Integration (Quadrature)

In this section...

The area beneath a section of a function $F(x)$ can be determined by numerically integrating $F(x)$, a process referred to as *quadrature*. The MATLAB quadrature functions are:

`quad`	Use adaptive Simpson quadrature
`quadl`	Use adaptive Lobatto quadrature
`quadgk`	Use adaptive Gauss-Kronrod quadrature
`quadv`	Vectorized quadrature
`dblquad`	Numerically evaluate double integral
`triplequad`	Numerically evaluate triple integral

To integrate the function defined by `humps.m` from `0` to `1`, use

```
q = quad(@humps,0,1)

q =
    29.8583
```

Both `quad` and `quadl` operate recursively. If either method detects a possible singularity, it prints a warning.

You can include a fourth argument for `quad` or `quadl` that specifies an absolute error tolerance for the integration. If a nonzero fifth argument is passed to `quad` or `quadl`, the function evaluations are traced.

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

In this section...

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 4-33, 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 4-4 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 4-33.

5

Differential Equations

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

In this section...

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

Solver	Solves These Kinds of Problems	Method
`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 $y_1, y_2, \cdots, y_n$.

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

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

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 5-11 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 5-5. 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.

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 5-17.

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 5-19.)

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 5-5.
`options`	Structure of optional parameters that change the default integration properties. "Changing ODE Integration Properties" on page 5-19 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 5-76 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 5-11 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, μ = 1000 (Stiff)" on page 5-14 demonstrates the solution of a stiff problem.
- "Evaluating the Solution at Specific Points" on page 5-16 tells you how to evaluate the solution at specific points.

Note See "Basic ODE Solver Syntax" on page 5-9 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 5-6 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 5-11 and the `@` 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`.

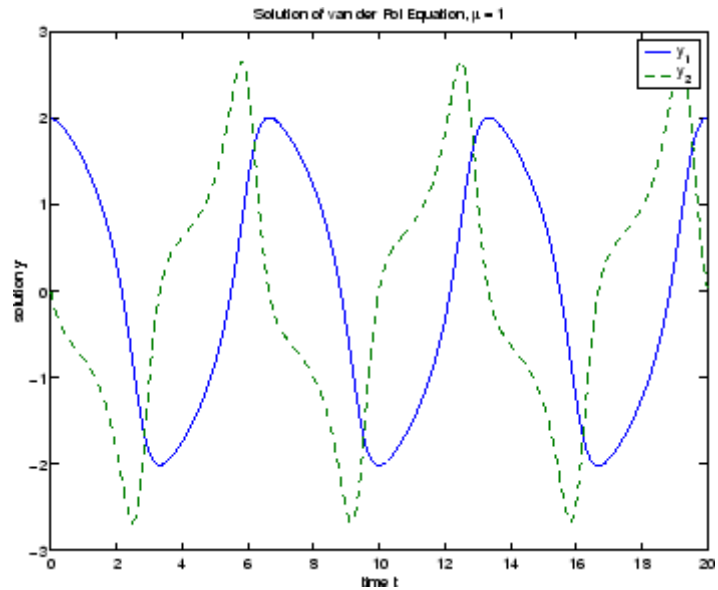
```
[t,y] = ode45(@vdp1,[0 20],[2; 0]);
```

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" section of 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.

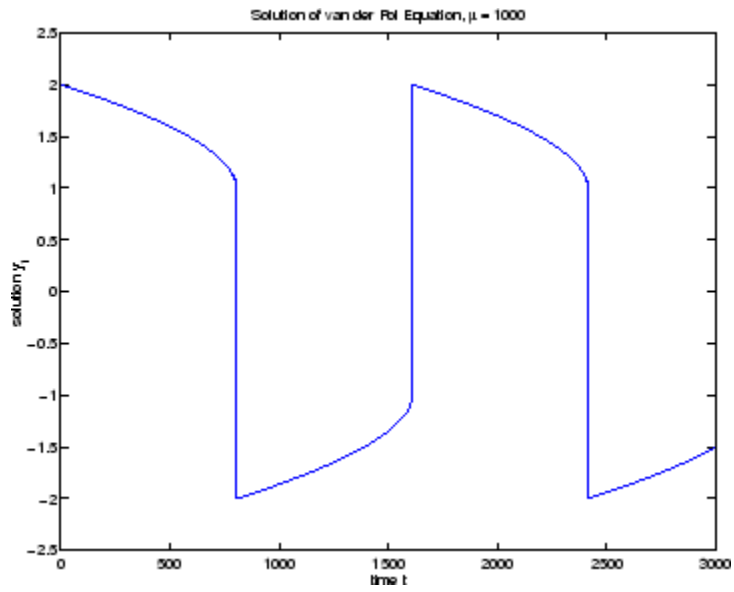
The `vdp1000` function evaluates the van der Pol system from the previous example, but with μ = 1000.

```
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 5-11.

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 μ = 1 and μ=1000 are hard-coded in the ODE functions. If you are solving an ODE for several different parameter values, it might be more convenient to include the parameter in the ODE function and assign a value to the parameter each time you run the ODE solver. This section explains how to do this for the van der Pol equation.

One way to provide parameter values to the ODE function is to write 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 5-11, enter

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

To run the code for μ = 1000, as in "Example: The van der Pol Equation, μ = 1000 (Stiff)" on page 5-14, 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 5-11 by calling `ode45` with a single output argument `sol`,

```
sol = ode45(@vdp1,[0 20],[2; 0]);
```

`ode45` returns the solution as a structure. You can then evaluate the approximate solution at points in the vector `xint = 1:5` as follows:

```
xint = 1:5;
Sxint = deval(sol,xint)

Sxint =

    1.5081    0.3235   -1.8686   -1.7407   -0.8344
   -0.7803   -1.8320   -1.0220    0.6260    1.3095
```

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

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 5-18 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 5-76 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.

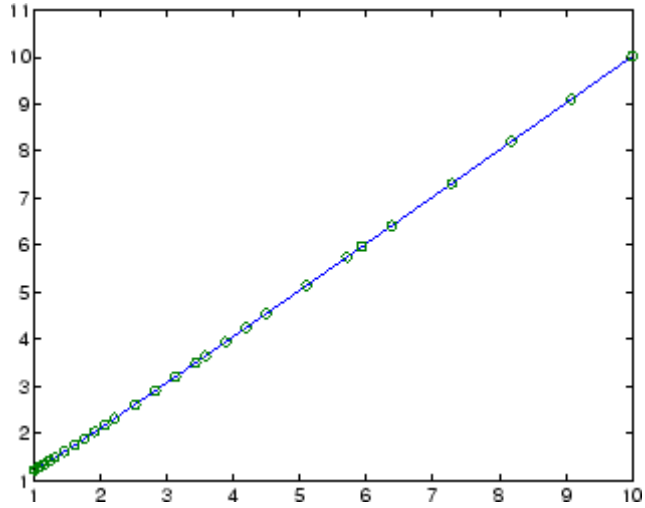
Before calling `ode15i`, the example uses a helper function `decic` to compute a consistent initial value for $y'(t_0)$. In the following call, the given initial

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 5-22. For a complete description of the available options, see the reference page for `odeset`.

Examples: Applying the ODE Initial Value Problem Solvers

Running the Examples

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.

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.

$$y_1' = y_2 y_3$$

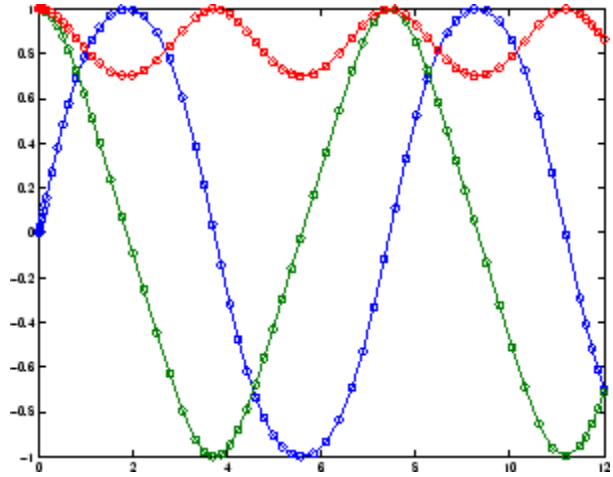
$$y_2' = -y_1 y_3$$

$$y_3' = -0.51 y_1 y_2$$

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, μ = 1000 (Stiff)" on page 5-14. The differential equations

$$y_1' = y_2$$

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

involve a constant parameter μ.

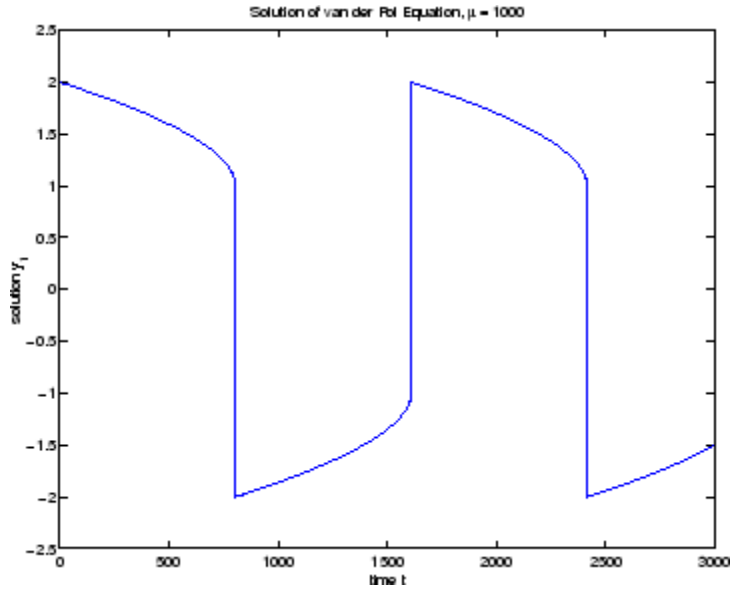
As μ increases, the problem becomes more stiff, and the period of oscillation becomes larger. When μ is 1000 the equation is in relaxation oscillation and the problem is very stiff. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with 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 μ = 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

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

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

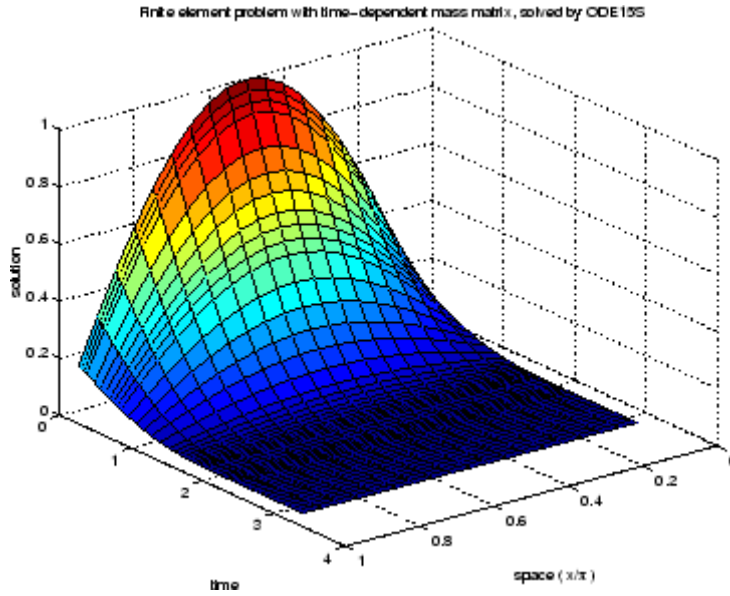
```
options = odeset('Mass',@mass,'MStateDep','none','Jacobian',J)
```

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

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

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

In the call `brussode(N)`, where `N` corresponds to N, the parameter `N` ≥ 2 specifies the number of grid points. The resulting system consists of `2N` equations. By default, `N` is `20`. The problem becomes increasingly stiff and the Jacobian increasingly sparse as `N` increases.

The nested function `f(t,y)` returns the derivatives vector for the Brusselator problem. The subfunction `jpattern(N)` returns a sparse matrix of `1`s and `0`s showing the locations of nonzeros in the Jacobian $\partial f/\partial y$. The example assigns this matrix to the property `JPattern`, and the solver uses the sparsity pattern to generate the Jacobian numerically as a sparse matrix. Providing a sparsity pattern can significantly reduce the number of function evaluations required to generate the Jacobian and can accelerate integration.

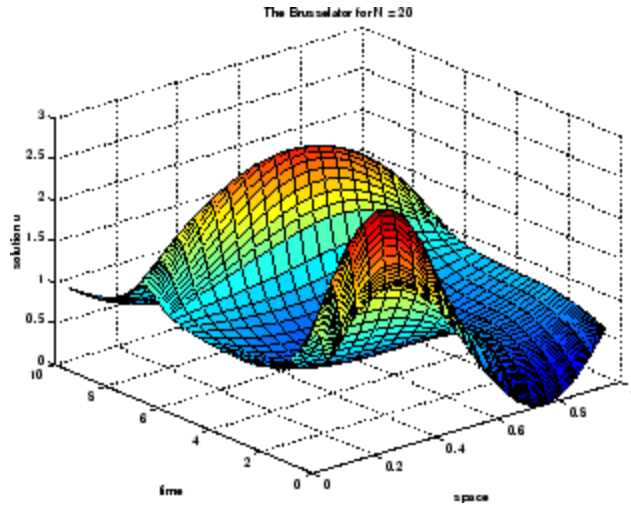
For the Brusselator problem, if the sparsity pattern is not supplied, `2N` evaluations of the function are needed to compute the `2N`-by-`2N` Jacobian matrix. If the sparsity pattern is supplied, only four evaluations are needed, regardless of the value of `N`.

To run this example, click on the example name, or type `brussode` at the command line. From the command line, you can specify a value of N as an argument to `brussode`. The default is $N = 20$.

```
function brussode(N)
%BRUSSODE  Stiff problem modeling a chemical reaction

if nargin < 1
  N = 20;
```

```
end

tspan = [0; 10];
y0 = [1+sin((2*pi/(N+1))*(1:N));
repmat(3,1,N)];

options = odeset('Vectorized','on','JPattern',jpattern(N));

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

u = y(:,1:2:end);
x = (1:N)/(N+1);
surf(x,t,u);
view(-40,30);
xlabel('space');
ylabel('time');
zlabel('solution u');
title(['The Brusselator for N = ' num2str(N)]);
% ------------------------------------------------------------
function dydt = f(t,y)
c = 0.02 * (N+1)^2;
dydt = zeros(2*N,size(y,2));      % preallocate dy/dt
% Evaluate the two components of the function at one edge of
% the grid (with edge conditions).
i = 1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(1-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(3-2*y(i+1,:)+y(i+3,:));
% Evaluate the two components of the function at all interior
% grid points.
i = 3:2:2*N-3;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(y(i-2,:)-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(y(i-1,:)-2*y(i+1,:)+y(i+3,:));
% Evaluate the two components of the function at the other edge
% of the grid (with edge conditions).
i = 2*N-1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
```

```
            c*(y(i-2,:)-2*y(i,:)+1);
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(y(i-1,:)-2*y(i+1,:)+3);
end   % End nested function f
end   % End function brussode
% ------------------------------------------------------------------
function S = jpattern(N)
B = ones(2*N,5);
B(2:2:2*N,2) = zeros(N,1);
B(1:2:2*N-1,4) = zeros(N,1);
S = spdiags(B,-2:2,2*N,2*N);
end;
```



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 5-33).

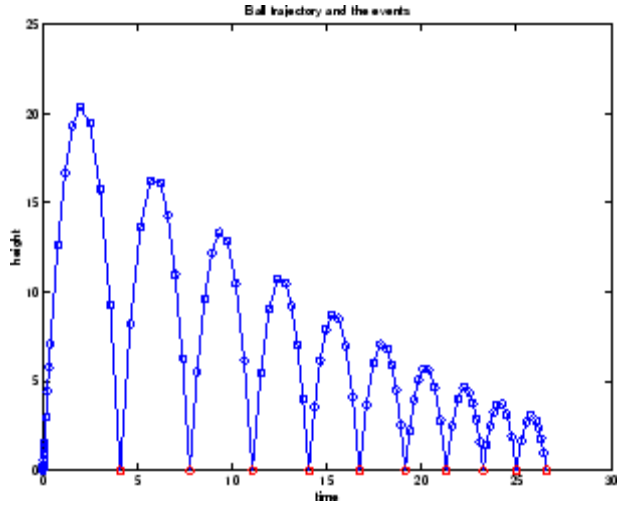
In `ballode`, setting the `Events` property to `@events` causes the solver to stop the integration (`isterminal = 1`) when the ball hits the ground (the height `y(1)` is `0`) during its fall (`direction = -1`). The example then restarts the integration with initial conditions corresponding to a ball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

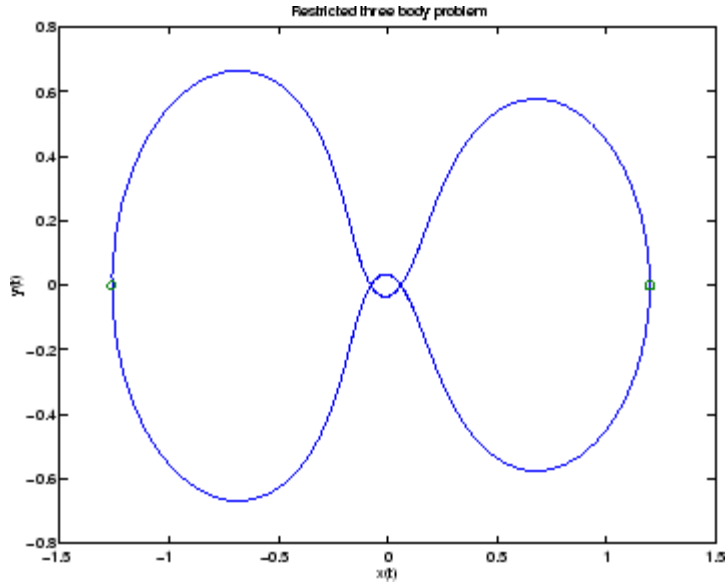
The nested `events` function includes event functions that locate the point of maximum distance from the starting point and the time the spaceship returns to the starting point. Note that the events are located accurately, even though the step sizes used by the integrator are *not* determined by the 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

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

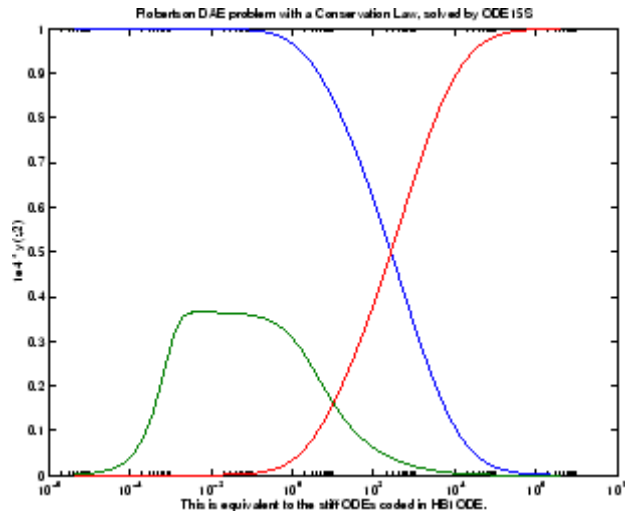
M is singular, but `hb1dae` does not inform the solver of this. The solver must recognize that the problem is a DAE, not an ODE. Similarly, although consistent initial conditions are obvious, the example uses an inconsistent value $y_3(0) = 10^{-3}$ to illustrate computation of consistent initial conditions.

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`, or for implicit solvers (`ode15s`, `ode23t`, `ode23tb`) applied to problems where there is a mass matrix.

Imposing nonnegativity is not always a trivial task. We suggest that you use this option only when necessary, for example in instances in which the application of a solution or integration will fail otherwise.

Consider the following initial value problem solved on the interval `[0, 40]`:

```
y' = - |y|, y(0) = 1
```

The solution of this problem decays to zero. If a solver produces a negative approximate solution, it begins to track the solution of the ODE through this

value, the solution goes off to minus infinity, and the computation fails. Using the `NonNegative` property prevents this from happening.

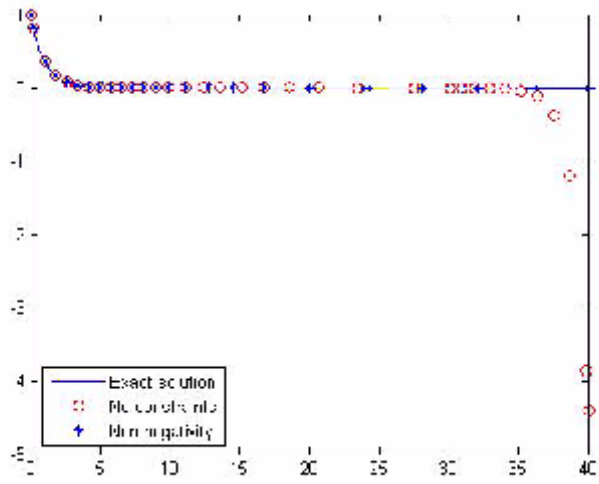
In this example, the first call to `ode45` uses the defaults for the solver parameters:

```
ode = @(t,y) -abs(y);
[t0,y0] = ode45(ode,[0, 40], 1);
```

The second uses options to impose nonnegativity conditions:

```
options = odeset('NonNegative',1);
[t1,y1] = ode45(ode,[0, 40], 1, options);
```

This plot compares the numerical solution to the exact solution.



Here is a more complete view of the code used to obtain this plot:

```
ode = @(t,y) -abs(y);
options = odeset('Refine',1);
[t0,y0] = ode45(ode,[0, 40], 1,options);
options = odeset(options,'NonNegative',1);
```

```
[t1,y1] = ode45(ode,[0, 40], 1, options);
t = linspace(0,40,1000);
y = exp(-t);
plot(t,y,'b-',t0,y0,'ro',t1,y1,'b*');
legend('Exact solution','No constraints','Nonnegativity', ...
       'Location','SouthWest')
```

The 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

$$\epsilon y' = (1-x)y - y^2, \quad y(0) = 1$$

For $0 < \epsilon < 1$, the solution of this problem approaches null isoclines $y = 1 - x$ and $y = 0$ for $x < 1$ and $x > 1$, respectively. The numerical solution, when computed with default tolerances, follows the $y = 1 - x$ isocline for the whole interval of integration. Imposing nonnegativity constraints results in the correct solution.

Here is the code that makes up the `kneeode` demo:

```
function kneeode
%KNEEODE  The "knee problem" with Nonnegativity constraints.

% Problem parameter
epsilon = 1e-6;

y0 = 1;
xspan = [0, 2];

% Solve without imposing constraints
options = [];
[x1,y1] = ode15s(@odefcn,xspan,y0,options);

% Impose nonnegativity constraint
options = odeset('NonNegative',1);
[x2,y2] = ode15s(@odefcn,xspan,y0,options);

figure
plot(x1,y1,'b.-',x2,y2,'g-')
```

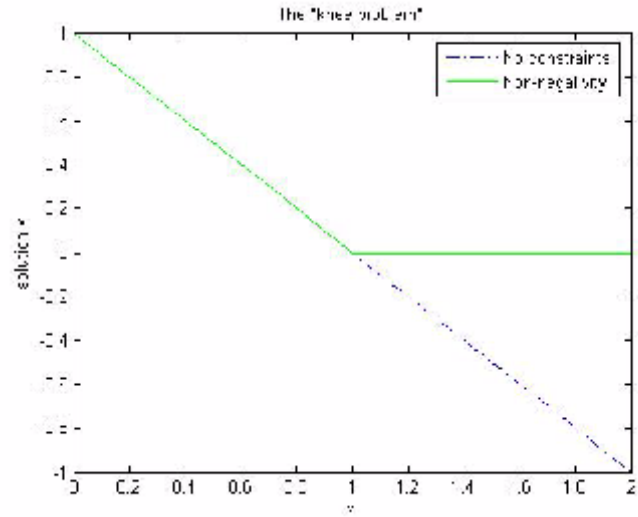
```
axis([0,2,-1,1]);
title('The "knee problem"');
legend('No constraints','nonnegativity')
xlabel('x');
ylabel('solution y')

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

The derivative function is defined within nested function `odefcn`. The value of epsilon used in `odefcn` is obtained from the outer function:

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

The demo solves the problem using the `ode15s` function, first with the default options, and then by imposing a nonnegativity constraint. To run the demo, type `kneeode` at the MATLAB command prompt.

Here is the output plot. The plot confirms correct solution behavior after imposing constraints.



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 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)

Example	Description
`burgersode`	ODE with strongly state-dependent mass matrix — Burgers' equation solved using a moving mesh technique
`fem1ode`	Stiff problem with a time-dependent mass matrix — finite element method
`fem2ode`	Stiff problem with a constant mass matrix — finite element method
`hb1ode`	Stiff ODE problem solved on a very long interval — Robertson chemical reaction
`hb1dae`	Robertson problem — stiff, linearly implicit DAE from a conservation law
`ihb1dae`	Robertson problem — stiff, fully implicit DAE
`iburgersode`	Burgers' equation solved as implicit ODE system
`kneeode`	The "knee problem" with nonnegativity constraints
`orbitode`	Advanced event location — restricted three body problem
`rigidode`	Nonstiff problem — Euler equations of a rigid body without external forces
`vdpode`	Parameterizable van der Pol equation (stiff for large μ)

Questions and Answers, and Troubleshooting

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

- Solving Different Kinds of Problems on page 5-49
- Troubleshooting on page 5-51

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.

Problem Size, Memory Use, and Computation Speed (Continued)

Question	Answer
I'm solving a very large system, but only care about a couple of the components of `y`. Is there any way to avoid storing all of the elements?	Yes. The user-installable output function capability is designed specifically for this purpose. When you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history. Instead, the solver calls `OutputFcn(t,y,flag)` at each time step. To keep the history of specific elements, write an output function that stores or plots only the elements you care about.
What is the startup cost of the integration and how can I reduce it?	The biggest startup cost occurs as the solver attempts to find a step size appropriate to the scale of the problem. If you happen to know an appropriate step size, use the `InitialStep` property. For example, if you repeatedly call the integrator in an event location loop, the last step that was taken before the event is probably on scale for the next integration. See `ballode` for an example.

Time Steps for Integration

Question	Answer
The first step size that the integrator takes is too large, and it misses important behavior.	You can specify the first step size with the `InitialStep` property. The integrator tries this value, then reduces it if necessary.
Can I integrate with fixed step sizes?	No.

Error Tolerance and Other Options

Question	Answer
How do I choose `RelTol` and `AbsTol`?	`RelTol`, the relative accuracy tolerance, controls the number of correct digits in the answer. `AbsTol`, the absolute error tolerance, controls the difference between the answer and the solution. At each step, the error `e` in component `i` of the solution satisfies `|e(i)| ≤max(RelTol*abs(y(i)),AbsTol(i))` Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.

Error Tolerance and Other Options (Continued)

Question	Answer
I want answers that are correct to the precision of the computer. Why can't I simply set `RelTol` to `eps`?	You can get close to machine precision, but not that close. The solvers do not allow `RelTol` near `eps` because they try to approximate a continuous function. At tolerances comparable to `eps`, the machine arithmetic causes all functions to look discontinuous.
How do I tell the solver that I don't care about getting an accurate answer for one of the solution components?	You can increase the absolute error tolerance corresponding to this solution component. If the tolerance is bigger than the component, this specifies no correct digits for the component. The solver may have to get some correct digits in this component to compute other components accurately, but it generally handles this automatically.

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 5-27. 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 5-93.
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`.
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`.

Troubleshooting (Continued)

Question	Answer
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

In this section...

DDE Function Summary

DDE Initial Value Problem Solvers

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

DDE Helper Functions

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

DDE Solver Properties Handling

Use these functions to create, alter, or access an options structure. An options structure contains named properties, the values of which are passed to `dde23` or `ddesd`, thus affecting the solution of the problem.

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

DDE Initial Value Problem Examples

These examples illustrate the kind of problems you can solve using dde23 or ddesd. 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 the link to invoke the browser, or type odeexamples('dde') at the command line.

The first two of these examples demonstrate ways that you can use the dde23 solver. The last example demonstrates the ddesd solver.

Example	Description
ddex1	Straightforward example
ddex2	Cardiovascular model with discontinuities
ddex3	Problem involving state-dependent delays

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

Introduction to Initial Value DDE Problems

The DDE dde23 solver can solve systems of ordinary differential equations, such as

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

where t is the independent variable, y is the dependent variable, and y' represents (derivative of y with respect to t)dy/dt. The delays (lags) $\tau_1, \ldots, \tau_k$ are positive constants. The solver `ddesd` allows delays that depend on t and y.

Using a History to Specify the Solution of Interest

In an *initial value problem*, you seek the solution on an interval $[t_0, t_f]$ with $t_0 < t_f$. The DDE shows that $y'(t)$ depends on values of the solution at times prior to t. In particular, $y'(t_0)$ depends on $y(t_0 - \tau_1), \ldots, y(t_0 - \tau_k)$. Because of this, a solution on $[t_0, t_f]$ depends on its values for $t \le t_0$, i.e., its *history* $S(t)$.

Propagation of Discontinuities with DDE Solvers

Generally, the solution $y(t)$ of an IVP for a system of DDEs has a jump in its first derivative at the initial point t_0 because the first derivative of the history function does not satisfy the DDE there. A discontinuity in any derivative propagates into the future at spacings of $\tau_1, \tau_2, \ldots, \tau_k$ when the delays are constant, and in a more complicated way when they are not. For the DDEs solved by `dde23` and `ddesd`, the solution becomes smoother as the integration proceeds.

DDE Solvers

This section describes:

The basic syntax for the two solvers is shown in the function reference pages for `dde23` and `ddesd`.

DDE Solver dde23

The function `dde23` solves initial value problems for DDEs with constant delays. It integrates a system of first-order differential equations

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

on the interval $[t_0, t_f]$, with $t_0 < t_f$ and given history $y(t) = S(t)$ for $t \le t_0$.

dde23 produces a solution that is continuous on $[t_0, t_f]$. You can use the function deval and the output of dde23 to evaluate the solution at specific points on the interval of integration.

dde23 tracks low-order discontinuities and integrates the differential equations with the explicit Runge-Kutta (2,3) pair and interpolant used by ode23. The Runge-Kutta formulas are implicit for step sizes longer than the delays. When the solution is smooth enough that steps this big are justified, the implicit formulas are evaluated by a predictor-corrector iteration.

DDE Solver ddesd

The function ddesd solves initial value problems for DDEs with general delays. It integrates a system of first-order differential equations

$$y'(t) = f(t, y(t), y(d(1)), \ldots, y(d(k)))$$

on the interval $[t_0, t_f]$, with $t_0 < t_f$, where delays $d(j)$ can depend on both t and $y(t)$. Use the function deval and the output of ddesd to evaluate the solution at specific points on the interval of integration.

ddesd integrates with the classic four-stage, fourth-order explicit Runge-Kutta method, and controls the size of the residual of a natural interpolant. It uses iteration to take steps that are longer than the delays. For further details, see "Solving ODEs and DDEs with Residual Control," L.F. *Shampine, Applied Numerical Mathematics*, 52 (2005), pp 113-127.

Solving DDE Problems

This section uses examples to describe

- Using dde23 and ddesd to solve initial value problems for 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.

---

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 5-11). 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`.

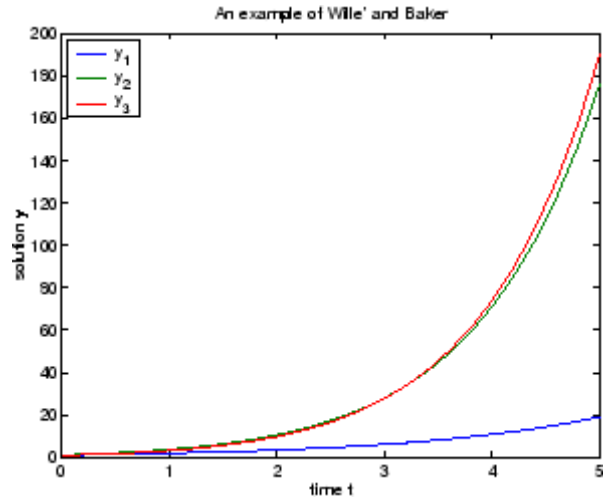
```
sol = dde23(@ddex1de,lags,@ddex1hist,[0,5]);
```

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 `i`th column of `Sint` approximates the solution $y(\text{tint(i)})$.

Using the output `sol` from the previous example, this code evaluates the numerical solution at 100 equally spaced points in the interval [0,5] and plots the result.

```
tint = linspace(0,5);
Sint = deval(sol,tint);
plot(tint,Sint);
```

Discontinuities

`dde23` performs better if it is informed of discontinuities in the history and at known locations. Discontinuities may be specified by event functions. There

is a property with which you can specify a solution that is different from the value given by the history function.

Discontinuity	Property	Comments
At the initial value $t = t_0$	`InitialY`	Generally the initial value $y(t_0)$ is the value $S(t_0)$ returned by the history function, which is to say that the solution is continuous at the initial point. However, if this is not the case, supply a different initial value using the `InitialY` property.
In the history, i.e., the solution at $t < t_0$, or in the equation coefficients for $t > t_0$	`Jumps`	Provide the known locations t of the discontinuities in a vector as the value of the `Jumps` property. Applies only to `dde23`.
State-dependent	`Events`	`dde23` and `ddesd` use the events function you supply to locate these discontinuities. When the solver finds such a discontinuity, restart the integration to continue. Specify the solution structure for the current integration as the history for the new integration. The solver extends each element of the solution structure after each restart so that the final structure 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.

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

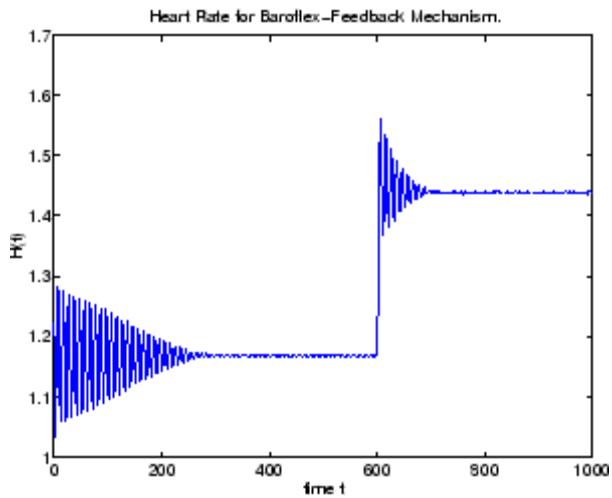
```
sol = dde23(@ddex2de,tau,history,[0, 600]);
sol = dde23(@ddex2de,tau,sol,[600, 1000]);
```

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. This is not an option with `ddesd`, which handles discontinuities in a different way. When you use event functions to locate such discontinuities, you must restart the integration at discontinuities.



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`.

Example of a State-Dependent Delay

This example solves a system of two DDEs with state-dependent delay that was used as a test problem by W.H. Enright and H. Hayashi [10] because it has an analytical solution. The differential equations are

$$y'_1(t) = y_2(t)$$
$$y'_2(t) = -y_2(e^{(1-y_2(t))}) \times y_2(t)^2 \times e^{(1-y_2(t))}$$

The analytical solution

$$y_1(t) = \log(t)$$
$$y_2(t) = 1/t$$

is used as the history for $t \le 0.1$ and the equations are solved on `[0.1, 5]`. The only thing different about solving this example is that it must be solved with `ddesd` rather than `dde23`. This is because the first factor in the second equation has the form $y_2(d(y))$ with a delay that depends on the second component of the solution. The delay is provided to `ddesd` with a function like

```
function d = ddex3delay(t,y)
% State dependent delay function for DDEX3
d = exp(1 - y(2));
```

Note The demo `ddex3` contains the complete code for this example. To see the code in an editor, click the example name, or type `edit ddex3` at the command line. To run the example type `ddex3` at the command line.

Boundary Value Problems for ODEs

In this section...

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 the derivative of y with respect to x dy/dx.

See “What Is an Ordinary Differential Equation?” on page 5-5 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

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 5-75.

`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, (see "BVP Solver Options" on page 5-70).

`bcfun` — Handle to a function that evaluates the residual in the boundary conditions. It has the basic form

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

where `ya` and `yb` are column vectors representing `y(a)` and `y(b)`, and `res` is a column vector of the residual in satisfying the boundary conditions. `bcfun` can also accept a vector of unknown parameters and a variable number of known parameters, (see "BVP Solver Options" on page 5-70).

`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 5-59.

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 5-76 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

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 lambdaλ of Mathieu's equation

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

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

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

Note The demo `mat4bvp` contains the complete code for this example. The demo uses nested functions to place all functions required by `bvp4c` in a single M-file. To run this example type `mat4bvp` at the command line. See "BVP Solver Basic Syntax" on page 5-69 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 5-75 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" section of MATLAB Programming 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`.

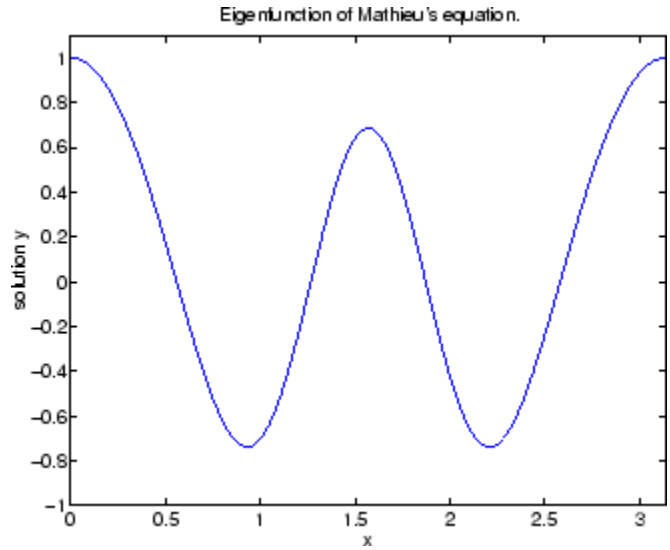
```
fprintf('The fourth eigenvalue is approximately %7.3f.\n',...
        sol.parameters)
```

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 5-76 for information about using `deval`.

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



Finding Unknown Parameters

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

$$y' = f(x, y, p)$$
$$bc(y(a), y(b), p) = 0$$

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

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

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

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

While solving the differential equations, `bvp4c` adjusts the value of unknown parameters to satisfy the boundary conditions. The solver returns the final values of these unknown parameters in `sol.parameters`. See "Example: Mathieu's Equation" on page 5-72.

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 `i`th column of `Sxint` approximates the solution $y(\text{xint(i)})$.

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 5-69 and "Solving BVP Problems" on page 5-72 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.

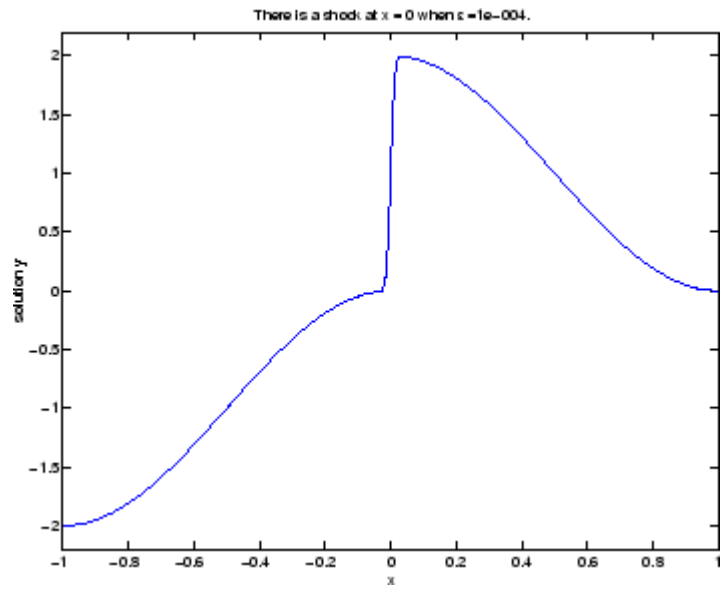
```
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
```

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 5-69 and "Solving BVP Problems" on page 5-72 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]:

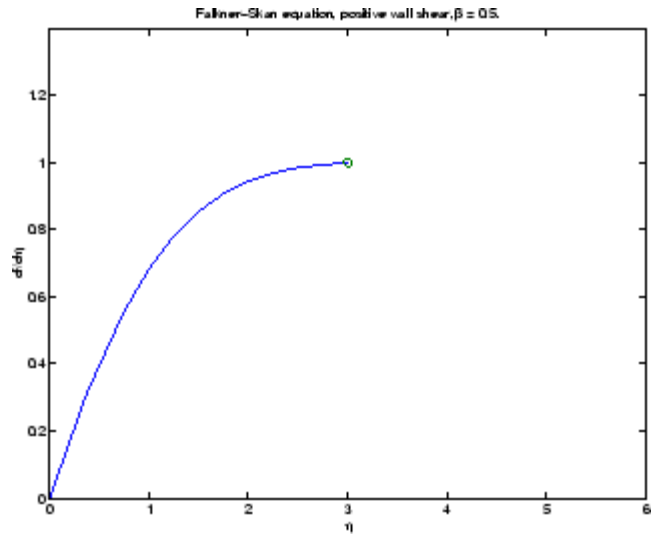
```
sol = bvp4c(@fsode,@fsbc,solinit);
eta = sol.x;
f = sol.y;

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
```



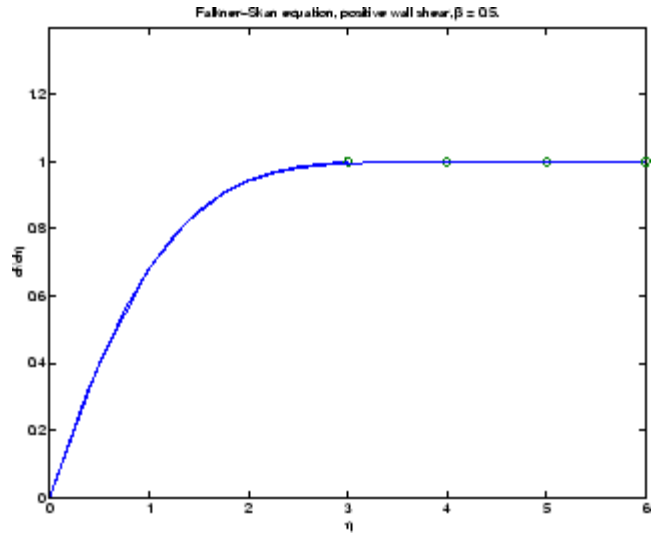
5 **Use continuation to solve the problem and plot subsequent solutions.** The example then solves the problem for `infinity = 4, 5, 6`. It uses `bvpinit` to extrapolate the solution `sol` for one value of `infinity` as an initial guess for the next value of `infinity`. For each iteration, the example 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

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

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 5-69 and "Solving BVP Problems" on page 5-72 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.

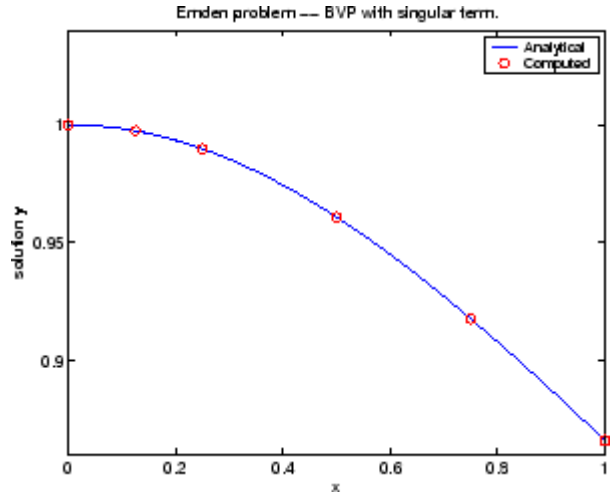
```
sol = bvp4c(@emdenode,@emdenbc,solinit,options);
```

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 you are solving a nonsmooth problem. The following equations are solved on $0 \le x \le \lambda$ for constant parameters `n`, κ, $\lambda > 1$, and $\eta = \lambda^2/(n \times \kappa^2)$. These are subject to boundary conditions `v(0) = 0` and $C(\lambda) = 1$:

```
v' = (C - 1)/n
C' = (v * C - min(x,1))/η
```

The term `min(x,1)` is not smooth at `xc = 1`, and this can affect the solver's efficiency. By introducing an interface point at `xc = 1`, smooth solutions can be obtained on `[0,1]` and `[1,λ]`. To get a continuous solution over the entire interval `[0,λ]`, the example imposes matching conditions at the interface.

Note The demo `threebvp` contains the complete code for this example and solves the problem for λ = 2, n = 0.05, and several values of κ. The demo uses nested functions to place all functions required by `bvp4c` in a single 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 `k`th column `YL(:,k)` represents the solution at the left boundary of the `k`th region. Similarly, `YR(:,k)` represents the solution at the right boundary of the `k`th region.

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

```
function res = bc(YL,YR)
   res = [YL(1,1)               % v(0) = 0
          YR(1,1) - YL(1,2)     % Continuity of v(x) at x=1
          YR(2,1) - YL(2,2)     % Continuity of C(x) at x=1
          YR(2,end) - 1];       % C(λ) = 1
end                             % End nested function bc
```

5 **Create an initial guess.** For multipoint BVPs, when creating an initial guess using `bvpinit`, use double entries in `xinit` for the interface point `xc`. This example uses a constant guess `yinit = [1;1]`:

```
xc = 1;
xinit = [0, 0.25, 0.5, 0.75, xc, xc, 1.25, 1.5, 1.75, 2];
solinit = bvpinit(xinit,yinit)
```

For multipoint BVPs, you can use different guesses in different regions. To do that, you specify the initial guess for `y` as a function using the following syntax:

```
solinit = bvpinit(xinit,@yinitfcn)
```

The initial guess function must have the following general form:

```
function y = yinitfcn(x,region)
   switch region
   case 1            % x in [0, 1]
      y = [1;1];     % initial guess for y(x) 0≤x≤1
   case 2 % x in [1, λ]
      y = [1;1];     % initial guess for y(x), 1≤x≤λ
end
```

6 **Apply the solver.** The `bvp4c` function uses the same syntax for multipoint BVPs as it does for two-point BVPs:

```
sol = bvp4c(@f,@bc,solinit);
```

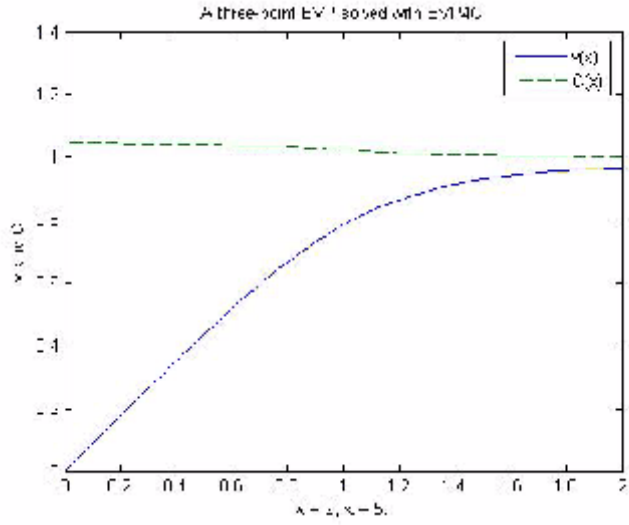
The mesh points returned in `sol.x` are adapted to the solution behavior, but the mesh still includes a double entry for the interface point `xc = 1`. Corresponding columns of `sol.y` represent the left and right solution at `xc`.

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

Note that, with the left and right values computed at the interface, the solution is not uniquely defined at `xc = 1`. When evaluating the solution exactly at the interface, `deval` issues a warning and returns the average of the left and right solution values. Call `deval` at `xc-eps(xc)` and `xc+eps(xc)` to get the limit values at `xc`.

The example plots the solution approximated at the mesh points selected by the solver:

```
plot(sol.x,sol.y(1,:),sol.x,sol.y(2,:),'--')
legend('v(x)','C(x)')
title('A three-point BVP solved with BVP4C')
xlabel(['\lambda = ',num2str(lambda), ...
        ', \kappa = ',num2str(kappa),'.'])
ylabel('v and C')
```



Partial Differential Equations

In this section...

PDE Function Summary

MATLAB PDE Solver

This is the MATLAB PDE solver.

PDE Initial-Boundary Value Problem Solver	Description
`pdepe`	Solve initial-boundary value problems for systems of parabolic and elliptic PDEs in one space variable and time.

PDE Helper Function

PDE Helper Function	Description
`pdeval`	Evaluate the numerical solution of a PDE using the output of `pdepe`.

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 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

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

The PDEs hold for $t_0 \le t \le t_f$ and $a \le x \le b$. The interval [a,b][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 xx 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 f rather than partial derivative of u with respect to x$\partial u/\partial x$. Also, of the two coefficients, only p can depend on u.

MATLAB Partial Differential Equation Solver

The PDE Solver

The MATLAB PDE solver, `pdepe`, solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable x and time t. There must be at least one parabolic equation in the system.

The pdepe solver converts the PDEs to ODEs using a second-order accurate spatial discretization based on a fixed set of 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 5-94.

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 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 5-104 for more information.

Solving PDE Problems

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 ≥ 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 5-96 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 5-94 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 5-95). 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 `@` section of MATLAB Programming for information about function handles.

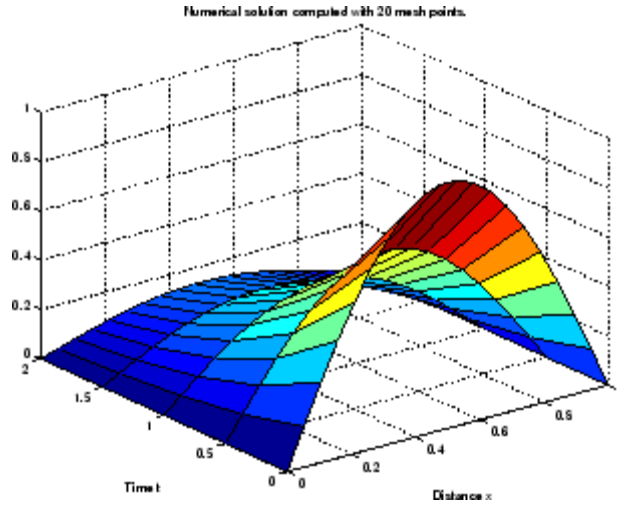
---

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

a Extract and display the first solution component. In this example, the solution u has only one component, but for illustrative purposes, the

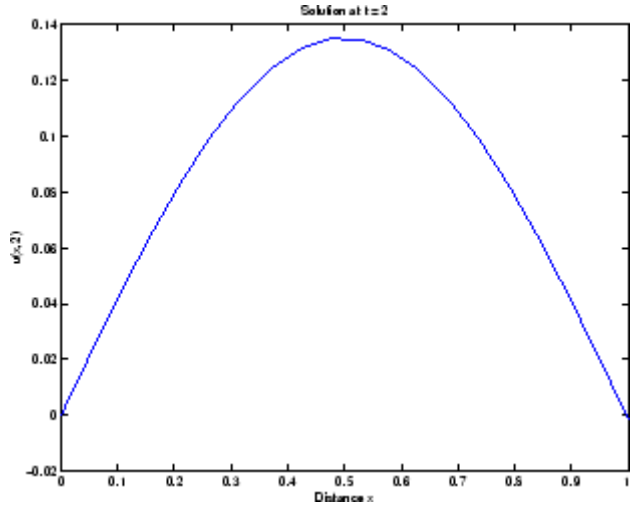
example “extracts” it from the three-dimensional array. The surface plot shows the behavior of the solution.

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

surf(x,t,u)
title('Numerical solution computed with 20 mesh points')
xlabel('Distance x')
ylabel('Time t')
```



b Display a solution profile at t_f, the final value of t. In this example, t_f = t = 2. See “Evaluating the Solution at Specific Points” on page 5-104 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 5-19 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 less than or equal to x less than or equal to $10 \le x \le 1$ 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 5-96 and "Solving PDE Problems" on page 5-99 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.

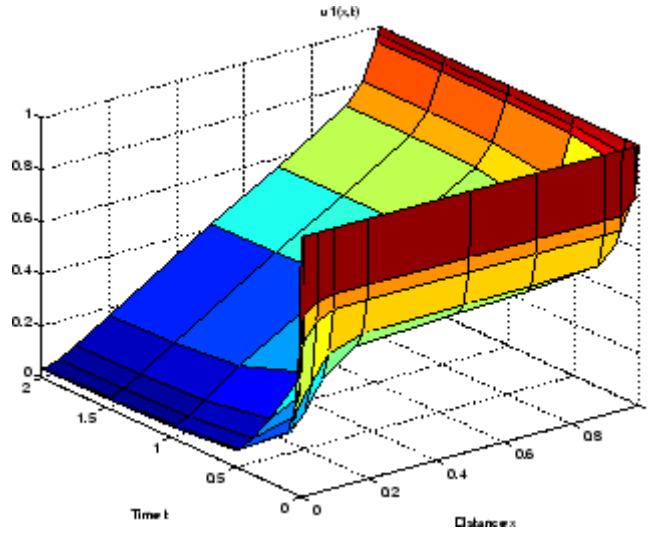
```
x = [0 0.005 0.01 0.05 0.1 0.2 0.5 0.7 0.9 0.95 0.99 0.995 1];
t = [0 0.005 0.01 0.05 0.1 0.5 1 1.5 2];
```

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)`.

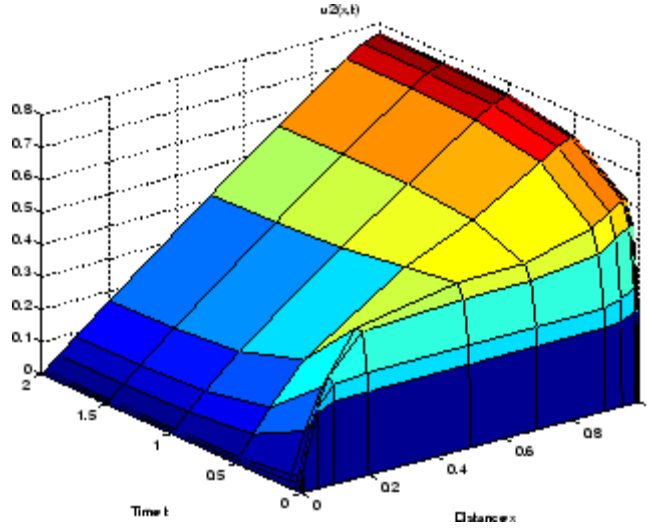
```
m = 0;
sol = pdepe(m,@pdex4pde,@pdex4ic,@pdex4bc,x,t);
```

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')
```



Selected Bibliography

[1] Ascher, U., R. Mattheij, and R. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, SIAM, Philadelphia, PA, 1995, p. 372.

[2] Cebeci, T. and H. B. Keller, "Shooting and Parallel Shooting Methods for Solving the Falkner-Skan Boundary-layer Equation," *J. Comp. Phys.*, Vol. 7, 1971, pp. 289-300.

[3] Hairer, E., and G. Wanner, *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems*, Springer-Verlag, Berlin, 1991, pp. 5-8.

[4] Hindmarsh, A. C., "LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers," *SIGNUM Newsletter*, Vol. 15, 1980, pp. 10-11.

[5] Hindmarsh, A. C., and G. D. Byrne, "Applications of EPISODE: An Experimental Package for the Integration of Ordinary Differential Equations," *Numerical Methods for Differential Systems*, L. Lapidus and W. E. Schiesser eds., Academic Press, Orlando, FL, 1976, pp 147-166.

[6] Ottesen, J. T., "Modelling of the Baroflex-Feedback Mechanism with Time-Delay," *J. Math. Biol.*, Vol. 36, 1997.

[7] Shampine, L. F., *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall Mathematics, 1994.

[8] Shampine, L. F., and M. K. Gordon, *Computer Solution of Ordinary Differential Equations*, W.H. Freeman & Co., 1975.

[9] Skeel, R. D. and M. Berzins, "A Method for the Spatial Discretization of Parabolic Equations in One Space Variable," *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, 1990, pp. 1-32.

[10] W.H. Enright and H. Hayashi, "The Evaluation of Numerical Software for Delay Differential Equations," R. Boisvert (Ed.), *The Quality of Numerical Software: Assessment and Enhancement*, Chapman & Hall, London, 1997, pp. 179-192.

6

Sparse Matrices

Function Summary

In this section...

Categories of Functions That Support Sparse Matrices

Sparse matrix functions are located in the MATLAB `sparfun` directory. These functions fall into the following categories:

Elementary Sparse Matrices

Function	Description
`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

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

Working with Sparse Matrices

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

Graph Theory

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

Reordering Algorithms

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

Linear Algebra

Function	Description
`eigs`	A few eigenvalues
`svds`	A few singular values
`luinc`	Incomplete LU factorization
`ilu`	Incomplete LU factorization
`cholinc`	Incomplete Cholesky factorization
`normest`	Estimate the matrix 2-norm
`condest`	1-norm condition number estimate
`sprank`	Structural rank

Linear Equations (Iterative Methods)

Function	Description
`bicg`	Biconjugate gradients method
`bicgstab`	Biconjugate gradients stabilized method
`cgs`	Conjugate gradients squared method

Function	Description
gmres	Generalized minimum residual method
lsqr	LSQR method
minres	Minimum residual method
pcg	Preconditioned conjugate gradients method
qmr	Quasi-minimal residual method
symmlq	Symmetric LQ method

Other Miscellaneous Functions

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

Categories of Functions That Do Not Support Sparse Matrices

- "Histogram Plotting Functions" on page 6-9

These built-in functions do not accept sparse matrices as input. M-file functions that depend on these built-ins will also not work with sparse matrices.

Elementary Matrices and Arrays

Function	Description
`rand`	Uniformly distributed pseudorandom numbers

Elementary Math Functions

Complex Functions.

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

Real Array Exponential Functions.

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

Bit-wise Functions

Function	Description
`bitand`	Bitwise AND
`bitcmp`	Bitwise complement
`bitget`	Bit at specified position
`bitmax`	Maximum double-precision floating-point integer

Function	Description
bitor	Bitwise OR
bitset	Set bit at specified position
bitshift	Shift bits specified number of places
bitxor	Bitwise XOR

Eigenvalue and Singular Value Functions

Function	Description
hess	Hessenberg form of matrix
ordeig	Eigenvalues of quasitriangular matrices
ordqz	Reorder eigenvalues in QZ factorization
ordschur	Reorder eigenvalues in Schur factorization
schur	Schur decomposition
svd	Singular value decomposition

Matrix Analysis Functions

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

Factorization Functions

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

Linear Equation Functions

Function	Description
`linsolve`	Solve linear system of equations
`lsqnonneg`	Solve nonnegative least-squares constraints problem
`pinv`	Moore-Penrose pseudoinverse of matrix

Specialized Math Functions

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

Function	Description
gammaln	Gamma function
psi	Psi (polygamma) function

Filtering and Convolution Functions

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

Fourier Transform Functions

Function	Description
fft	Discrete Fourier transform
fftn	N-D discrete Fourier transform
ifft	Inverse discrete Fourier transform
ifftn	N-D inverse discrete Fourier transform

Histogram Plotting Functions

Function	Description
histc	Histogram count

Sparse-Supported Replacement Functions

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

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

Reducing Memory and Efficiency with Sparse Matrices

In this section...

Storing Sparse Matrices

Using sparse matrices to store data that contains a large number of zero-valued elements can both save a significant amount of memory and also speed up the processing of that data. `sparse` is an attribute that you can assign to any two-dimensional MATLAB matrix that is composed of `double` or `logical` elements.

The `sparse` attribute allows MATLAB to:

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

For full matrices, MATLAB stores 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.

Comparing Storage for Sparse and Full Matrices

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 fewer in the sparse case, because zero-valued elements are not stored.

Creating and Importing Sparse Matrices

In this section...

Creating Sparse Matrices

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

The *density* of a matrix is the number of nonzero elements divided by the total number of matrix elements. For matrix `M`, this would be

```
nnz(M) / prod(size(M));
```

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

In this section...

Obtaining 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`.

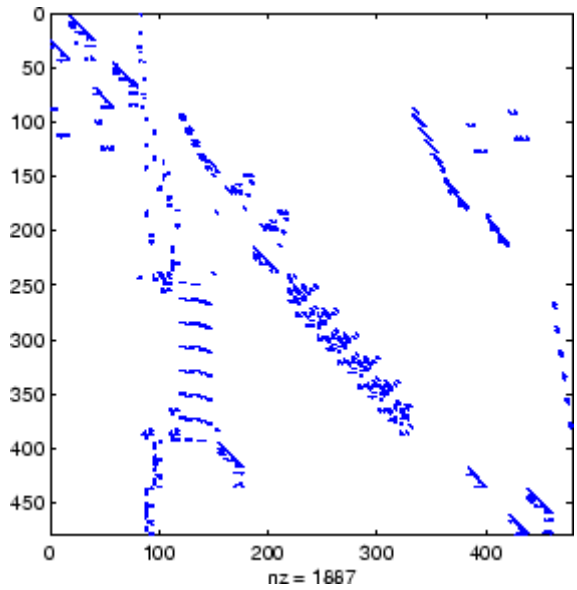
However, you can add as many nonzero elements to the matrix as desired. You are not constrained by the original value of `nzmax`.

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)
```



Finding Indices and Values of 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)
```

Operating on Sparse Matrices

In this section...

Considering Computational Complexity and Standard Mathematical Operations

Computational Complexity

The computational complexity of sparse operations is proportional to `nnz`, the number of nonzero elements in the matrix. Computational complexity also depends linearly on the row size `m` and column size `n` of the matrix, but is independent of the product `m*n`, the total number of zero and nonzero elements.

The complexity of fairly complicated operations, such as the solution of sparse linear equations, involves factors like ordering and fill-in, which are discussed in the previous section. In general, however, the computer time required for a sparse matrix operation is proportional to the number of arithmetic operations on nonzero quantities.

Operating Principals for Sparse Matrices

Sparse matrices propagate through computations according to these rules:

- Functions that accept a matrix and return a scalar or constant-size vector always produce output in full storage format. For example, the `size` function always returns a full vector, whether its input is full or sparse.

- Functions that accept scalars or vectors and return matrices, such as `zeros`, `ones`, `rand`, and `eye`, always return full results. This is necessary to avoid introducing sparsity unexpectedly. The sparse analog of `zeros(m,n)` is simply `sparse(m,n)`. The sparse analogs of `rand` and `eye` are `sprand` and `speye`, respectively. There is no sparse analog for the function `ones`.
- Unary functions that accept a matrix and return a matrix or vector preserve the storage class of the operand. If `S` is a sparse matrix, then `chol(S)` is also a sparse matrix, and `diag(S)` is a sparse vector. Columnwise functions such as `max` and `sum` also return sparse vectors, even though these vectors can be entirely nonzero. Important exceptions to this rule are the `sparse` and `full` functions.
- Binary operators yield sparse results if both operands are sparse, and full results if both are full. For mixed operands, the result is full unless the operation preserves sparsity. If `S` is sparse and `F` is full, then `S+F`, `S*F`, and `F\S` are full, while `S.*F` and `S&F` are sparse. In some cases, the result might be sparse even though the matrix has few zero elements.
- Matrix concatenation using either the `cat` function or square brackets produces sparse results for mixed operands.
- Submatrix indexing on the right side of an assignment preserves the storage format of the operand unless the result is a scalar. `T = S(i,j)` produces a sparse result if `S` is sparse and either `i` or `j` is a vector. It produces a full scalar if both `i` and `j` are scalars. Submatrix indexing on the left, as in `T(i,j) = S`, does not change the storage format of the matrix on the left.

Performing Permutations and Reordering

- "Reordering for Sparsity" on page 6-26
- "Reordering to Reduce Bandwidth" on page 6-26
- "Approximate Minimum Degree Ordering" on page 6-26

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 6-29 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].

Factorizing

This section discusses four important factorization techniques for sparse matrices:

LU Factorization

If `S` is a sparse matrix, the following command returns three sparse matrices `L`, `U`, and `P` such that `P*S = L*U`.

```
[L,U,P] = lu(S)
```

`lu` obtains the factors by Gaussian elimination with partial pivoting. The permutation matrix `P` has only `n` nonzero elements. As with dense matrices, the statement `[L,U] = lu(S)` returns a permuted unit lower triangular matrix and an upper triangular matrix whose product is `S`. By itself, `lu(S)` returns `L` and `U` in a single matrix without the pivot information.

The three-output syntax

```
[L,U,P] = lu(S)
```

selects `P` via numerical partial pivoting, but does not pivot to improve sparsity in the `LU` factors. On the other hand, the four-output syntax

```
[L,U,P,Q]=lu(S)
```

selects `P` via threshold partial pivoting, and selects `P` and `Q` to improve sparsity in the `LU` factors.

You can control pivoting in sparse matrices using

```
lu(S,thresh)
```

where `thresh` is a pivot threshold in [0,1]. Pivoting occurs when the diagonal entry in a column has magnitude less than `thresh` times the magnitude of any sub-diagonal entry in that column. `thresh = 0` forces diagonal pivoting. `thresh = 1` is the default. (The default for `thresh` is `0.1` for the four-output syntax).

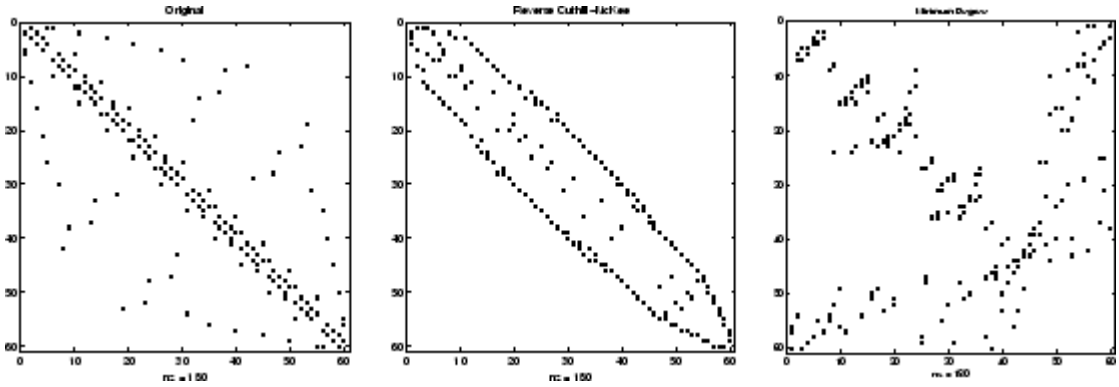
When you call `lu` with three or less outputs, MATLAB automatically allocates the memory necessary to hold the sparse `L` and `U` factors during the factorization. Except for the four-output syntax, MATLAB does not use any symbolic LU prefactorization to determine the memory requirements and set up the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

Reordering and Factorization. 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.

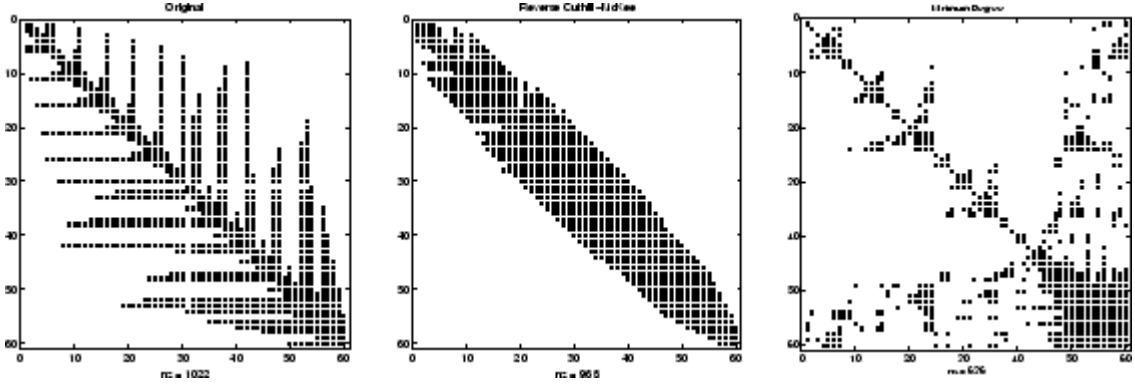
To see the fill-in generated in the LU factorization of the Bucky ball, use `speye(n,n)`, the sparse identity matrix, to insert -3s on the diagonal of `B`.

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

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

For example:

```
A = gallery('neumann', 1600) + speye(1600);
nnz(A)
ans =
        7840

nnz(lu(A))
ans =
      126478
```

shows that `A` has 7840 zeros, and its complete LU factorization has 126478 nonzeros:

```
nnz(luinc(A,'0'))
ans =
        7840

setup.type = 'nofill';
nnz(ilu(A,setup))
ans =
        7840

nnz(luinc(A,1e-6))
ans =
       51541

setup.type = 'ilutp';
setup.droptol = 1e-6;
```

```
nnz(luinc(A,setup))
ans =
       51541
```

These calculations, with both the `luinc` and `ilu` functions, show that with level 0 fill-in it has 7840 zeros, and with a drop tolerance of `1e-6` it has 51541 nonzeros. See the `luinc` and `ilu` reference pages for more options and details.

The `cholinc` function provides drop tolerance and level 0 fill-in Cholesky factorizations of symmetric, positive definite sparse matrices. See the `cholinc` reference page for more information.

Solving Simultaneous Linear Equations

- "Direct Methods" on page 6-34
- "Iterative Methods" on page 6-35

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 can 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
`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 can be approximated by one with constant coefficients, for example. Incomplete matrix factorizations can be used in the absence of natural preconditioners.

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

```
A = delsq(numgrid('S',50));
b = ones(size(A,1),1);
tol = 1.e-3;
maxit = 10;
R = cholinc(A,tol);
[x,flag,err,iter,res] = pcg(A,b,tol,maxit,R',R);
```

Only four iterations are required to achieve the prescribed accuracy.

Background information on these iterative methods and incomplete factorizations is available in [2] and [7].

Solving Eigenvalues and Singular Values

Two functions are available that compute a few specified eigenvalues or singular values. `svds` is based on `eigs` that uses ARPACK [6].

Functions to Compute a Few Eigenvalues or Singular Values

Function	Description
`eigs`	Few eigenvalues
`svds`	Few singular values

These functions are most frequently used with sparse matrices, but they can be used with full matrices or even with linear operators defined by M-files.

The statement

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

finds the `k` eigenvalues and corresponding eigenvectors of the matrix `A` that are nearest the "shift" `sigma`. If `sigma` is omitted, the eigenvalues largest in magnitude are found. If `sigma` is zero, the eigenvalues smallest in magnitude

are found. A second matrix, B, can be included for the generalized eigenvalue problem:

$$Av = \lambda Bv$$

The statement

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

finds the k largest singular values of A and

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

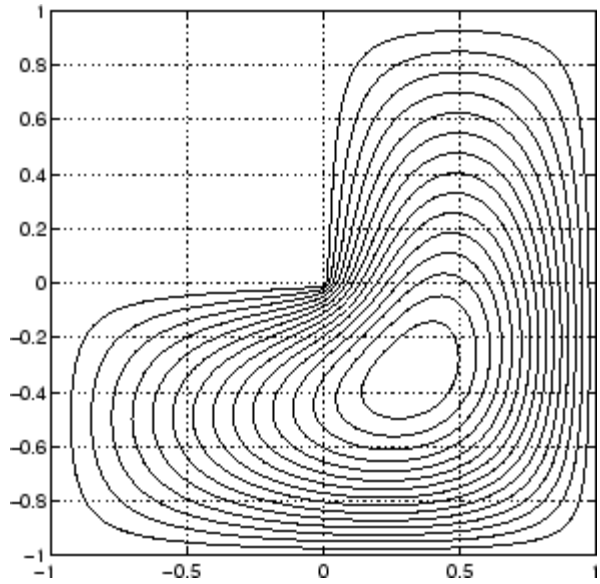
The statement

```
[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].

Identifying 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 could prevent you from transposing a sparse matrix having a large number of rows. This might occur even when the number of nonzero values is small.

Using linear indexing to access or assign an element in a large sparse matrix will fail if the linear index exceeds `intmax`. To access an element whose linear index is greater than `intmax`, use array indexing:

```
S = spalloc(216^2, 216^2, 2)
S(1) = 1
S(end) = 1
S(216^2,216^2) = 1
```

Selected Bibliography

[1] Amestoy, P. R., T. A. Davis, and I. S. Duff, "An Approximate Minimum Degree Ordering Algorithm," *SIAM Journal on Matrix Analysis and Applications*, Vol. 17, No. 4, Oct. 1996, pp. 886-905.

[2] Barrett, R., M. Berry, T. F. Chan, et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.

[3] Davis, T.A., Gilbert, J. R., Larimore, S.I., Ng, E., Peyton, B., "A Column Approximate Minimum Degree Ordering Algorithm," *Proc. SIAM Conference on Applied Linear Algebra*, Oct. 1997, p. 29.

[4] Gilbert, John R., Cleve Moler, and Robert Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM J. Matrix Anal. Appl.*, Vol. 13, No. 1, January 1992, pp. 333-356.

[5] Larimore, S. I., *An Approximate Minimum Degree Column Ordering Algorithm*, MS Thesis, Dept. of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, 1998, available at `http://www.cise.ufl.edu/submit/files/file_281.ps` .

[6] Lehoucq, R. B., D. C. Sorensen, C. Yang, *ARPACK Users' Guide*, SIAM, Philadelphia, 1998.

[7] Saad, Yousef, *Iterative Methods for Sparse Linear Equations*. PWS Publishing Company, 1996.

Index

A

B

C

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

Z

ISSN 1360-1725

THE UNIVERSITY
of MANCHESTER

The Test Matrix Toolbox for MATLAB (Version 3.0)

N. J. Higham

Numerical Analysis Report No. 276

September 1995

Manchester Centre for Computational Mathematics
Numerical Analysis Reports

DEPARTMENTS OF MATHEMATICS

Reports available from:
Department of Mathematics
University of Manchester
Manchester M13 9PL
England

And over the World-Wide Web from URLs
`http://www.ma.man.ac.uk/MCCM/MCCM.html`
`ftp://ftp.ma.man.ac.uk/pub/narep`

The Test Matrix Toolbox for MATLAB (Version 3.0)

Nicholas J. Higham*

September 22, 1995

Abstract

We describe version 3.0 of the Test Matrix Toolbox for MATLAB 4.2. The toolbox contains a collection of test matrices, routines for visualizing matrices, routines for direct search optimization, and miscellaneous routines that provide useful additions to MATLAB's existing set of functions. There are 58 parametrized test matrices, which are mostly square, dense, nonrandom, and of arbitrary dimension. The test matrices include ones with known inverses or known eigenvalues; ill-conditioned or rank deficient matrices; and symmetric, positive definite, orthogonal, defective, involutary, and totally positive matrices. The visualization routines display surface plots of a matrix and its (pseudo-) inverse, the field of values, Gershgorin disks, and two- and three-dimensional views of pseudospectra. The direct search optimization routines implement the alternating directions method, the multidirectional search method and the Nelder–Mead simplex method. We explain the need for collections of test matrices and summarize the features of the collection in the toolbox. We give examples of the use of the toolbox and explain some of the interesting properties of the Frank matrix and magic square matrices. The leading comment lines from all the toolbox routines are listed.

Key words. test matrix, MATLAB, pseudospectrum, visualization, Frank matrix, magic square matrix, random matrix, direct search optimization.

AMS subject classifications. primary 65F05

Contents

*Department of Mathematics, University of Manchester, Manchester, M13 9PL, England (na.nhigham@na-net.ornl.gov). This work was supported by Science and Engineering Research Council grant GR/H52139.

1 Distribution

If you wish to distribute the toolbox please give exact copies of it, not selected routines.

2 Installation

The Test Matrix Toolbox is available by anonymous ftp from The MathWorks with URL

```
ftp://ftp.mathworks.com/pub/contrib/linalg/testmatrix
```

This document is `testmatrix.ps` in the same location. The MathWorks ftp server provides information on how to download a complete directory as one file.

The toolbox is also available from the URL

```
ftp://ftp.ma.man.ac.uk/pub/higham/testmatrix.tar.Z
```

This document is `narep276.ps.Z` in the same location. To install the toolbox from this location, download the tar file (in binary mode) into a `testmatrix` directory (`matlab/testmatrix` is recommended). Then uncompress the tar file and untar it:

```
uncompress testmatrix.tar.Z
tar xvf testmatrix
```

To try the toolbox from within MATLAB, change to the `testmatrix` directory and run the demonstration script by typing `tmtdemo`. For serious use it is best to put the `testmatrix` directory on the MATLAB path *before* the `matlab/toolbox` entries—this is because several toolbox routines have the same name as MATLAB routines and are intended to replace them (namely, `compan`, `cond`, `hadamard`, `hilb`, and `pascal`).

This document describes version 3.0 of the toolbox, dated September 19, 1995.

3 Release History

The first release of this toolbox (version 1.0, July 4 1989) was described in a technical report [19]. The collection was subsequently published as ACM Algorithm 694 [21]. Prior to the current version, version 3.0, the most recent release was version 2.0 (November 14 1993) [24]. Version 2.0 incorporated many additions and improvements over version 1.3 and took full advantage of the features of MATLAB 4.

The major changes in version 3.0 are as follows.

- New routines: `cgs` (classical Gram–Schmidt), `mgs` (modified Gram–Schmidt), `gj` (Gauss–Jordan elimination), `diagpiv` (diagonal pivoting factorization with partial pivoting for a symmetric matrix); `adsmax`, `mdsmax` and `nmsmax` for direct search optimization.
- Bug in `eigsens` corrected. Minor bugs in other routines corrected.

Version 3.0 of the toolbox was developed in conjunction with the book *Accuracy and Stability of Numerical Algorithms* [26]. The book contains a chapter *A Gallery of Test Matrices* which has sections

- The Hilbert and Cauchy Matrices
- Random Matrices
- "Randsvd" Matrices
- The Pascal Matrix
- Tridiagonal Toeplitz Matrices
- Companion Matrices
- Notes and References
- LAPACK
- Problems

Users of the toolbox should consult [26] for further information not contained in this document.

4 Quick Reference Tables

This section contains quick reference tables for the Test Matrix Toolbox. All the M-files in the toolbox are listed by category, with a short description. More detailed documentation is given in Section 10, or can be obtained on-line by typing `help M-file_name`.

Demonstration	
`tmtdemo`	Demonstration of Test Matrix Toolbox.

Test Matrices, A–K	
`augment`	Augmented system matrix.
`cauchy`	Cauchy matrix.
`chebspec`	Chebyshev spectral differentiation matrix.
`chebvand`	Vandermonde-like matrix for the Chebyshev polynomials.
`chow`	Chow matrix—a singular Toeplitz lower Hessenberg matrix.
`circul`	Circulant matrix.
`clement`	Clement matrix—tridiagonal with zero diagonal entries.
`compan`	Companion matrix.
`condex`	"Counterexamples" to matrix condition number estimators.
`cycol`	Matrix whose columns repeat cyclically.
`dingdong`	Dingdong matrix—a symmetric Hankel matrix.
`dorr`	Dorr matrix—diagonally dominant, ill conditioned, tridiagonal.
`dramadah`	A $(0,1)$ matrix whose inverse has large integer entries.
`fiedler`	Fiedler matrix—symmetric.
`forsythe`	Forsythe matrix—a perturbed Jordan block.
`frank`	Frank matrix—ill conditioned eigenvalues.
`gallery`	Famous, and not so famous, test matrices.
`gearm`	Gear matrix.
`gfpp`	Matrix giving maximal growth factor for Gaussian elimination with partial pivoting.
`grcar`	Grcar matrix—a Toeplitz matrix with sensitive eigenvalues.
`hadamard`	Hadamard matrix.
`hanowa`	A matrix whose eigenvalues lie on a vertical line in the complex plane.
`hilb`	Hilbert matrix.
`invhess`	Inverse of an upper Hessenberg matrix.
`invol`	An involutory matrix.
`ipjfact`	A Hankel matrix with factorial elements.
`jordbloc`	Jordan block.
`kahan`	Kahan matrix—upper trapezoidal.
`kms`	Kac–Murdock–Szegö Toeplitz matrix.
`krylov`	Krylov matrix.

Test Matrices, L–Z	
`lauchli`	Lauchli matrix—rectangular.
`lehmer`	Lehmer matrix—symmetric positive definite.
`lesp`	A tridiagonal matrix with real, sensitive eigenvalues.
`lotkin`	Lotkin matrix.
`makejcf`	A matrix with given Jordan canonical form.
`minij`	Symmetric positive definite matrix $\min(i,j)$.
`moler`	Moler matrix—symmetric positive definite.
`neumann`	Singular matrix from the discrete Neumann problem (sparse).
`ohess`	Random, orthogonal upper Hessenberg matrix.
`orthog`	Orthogonal and nearly orthogonal matrices.
`parter`	Parter matrix—a Toeplitz matrix with singular values near π.
`pascal`	Pascal matrix.
`pdtoep`	Symmetric positive definite Toeplitz matrix.
`pei`	Pei matrix.
`pentoep`	Pentadiagonal Toeplitz matrix (sparse).
`poisson`	Block tridiagonal matrix from Poisson's equation (sparse).
`prolate`	Prolate matrix—symmetric, ill-conditioned Toeplitz matrix.
`rando`	Random matrix with elements -1, 0 or 1.
`randsvd`	Random matrix with pre-assigned singular values.
`redheff`	A (0,1) matrix of Redheffer associated with the Riemann hypothesis.
`riemann`	A matrix associated with the Riemann hypothesis.
`rschur`	An upper quasi-triangular matrix.
`smoke`	Smoke matrix—complex, with a "smoke ring" pseudospectrum.
`tridiag`	Tridiagonal matrix (sparse).
`triw`	Upper triangular matrix discussed by Wilkinson and others.
`vand`	Vandermonde matrix.
`wathen`	Wathen matrix—a finite element matrix (sparse, random entries).
`wilk`	Various specific matrices devised/discussed by Wilkinson.

Visualization	
`fv`	Field of values (or numerical range).
`gersh`	Gershgorin disks.
`ps`	Dot plot of a pseudospectrum.
`pscont`	Contours and colour pictures of pseudospectra.
`see`	Pictures of a matrix and its (pseudo-) inverse.

Decompositions and Factorizations	
`cgs`	Classical Gram–Schmidt QR factorization.
`cholp`	Cholesky factorization with pivoting of a positive semidefinite matrix.
`cod`	Complete orthogonal decomposition.
`diagpiv`	Diagonal pivoting factorization with partial pivoting.
`ge`	Gaussian elimination without pivoting.
`gecp`	Gaussian elimination with complete pivoting.
`gj`	Gauss–Jordan elimination to solve $Ax = b$.
`mgs`	Modified Gram–Schmidt QR factorization.
`poldec`	Polar decomposition.
`signm`	Matrix sign decomposition.

Direct Search Optimization	
`adsmax`	Alternating directions direct search method.
`mdsmax`	Multidirectional search method for direct search optimization.
`mmsmax`	Nelder–Mead simplex method for direct search optimization.

Miscellaneous	
`bandred`	Band reduction by two-sided unitary transformations.
`chop`	Round matrix elements.
`comp`	Comparison matrices.
`cond`	Matrix condition number in 1, 2, Frobenius, or ∞-norm.
`cpltaxes`	Determine suitable `axis` for plot of complex vector.
`dual`	Dual vector with respect to Hölder p-norm.
`eigsens`	Eigenvalue condition numbers.
`house`	Householder matrix.
`matrix`	Test Matrix Toolbox information and matrix access by number.
`matsignt`	Matrix sign function of a triangular matrix.
`pnorm`	Estimate of matrix p-norm ($1 \leq p \leq \infty$).
`qmult`	Pre-multiply by random orthogonal matrix.
`rq`	Rayleigh quotient.
`seqa`	Additive sequence.
`seqcheb`	Sequence of points related to Chebyshev polynomials.
`seqm`	Multiplicative sequence.
`show`	Display signs of matrix elements.
`skewpart`	Skew-symmetric (skew-Hermitian) part.
`sparsify`	Randomly sets matrix elements to zero.
`sub`	Principal submatrix.
`symmpart`	Symmetric (Hermitian) part.
`trap2tri`	Unitary reduction of trapezoidal matrix to triangular form.

5 Test Matrices

Numerical experiments are an indispensable part of research in numerical analysis. We do them for several reasons:

- To gain insight and understanding into an algorithm that is only partially understood theoretically.
- To verify the correctness of a theoretical analysis and to see if the analysis completely explains the practical behaviour.
- To compare rival methods with regard to accuracy, speed, reliability, and so on.
- To tune parameters in algorithms and codes, and to test heuristics.

One of the difficulties in designing experiments is finding good test problems—ones that reveal extremes of behaviour, cover a wide range of difficulty, are representative of practical problems, and (ideally) have known solutions. In many areas of numerical analysis good test problems have been identified, and several collections of such problems have been published. For example, collections are available in the areas of nonlinear optimization [33], linear programming [13], [31], ordinary differential equations [10], and partial differential equations [34].

Probably the most prolific devisers of test problems have been workers in matrix computations. Indeed, in the 1950s and 1960s it was common for a whole paper to be devoted to a particular test matrix: typically its inverse or eigenvalues would be obtained in closed form. An early survey of test matrices was given by Rutishauser [36]; most of the matrices he discusses come from continued fractions or moment problems. Two well-known books present collections of test matrices. Gregory and Karney [15] deal exclusively with the topic, while Westlake [43] gives an appendix of test matrices. In the 24 years since these books appeared several interesting matrices have been discovered (and in fact both books omit some worthy test matrices that were known at the time).

The Test Matrix Toolbox contains an up-to-date, well documented and readily accessible collection of test matrices. The matrices are given in the form of self-documenting MATLAB M-files. For some of the matrices we give mathematical formulas for the matrix elements in comment lines; in other cases the formulas can be reconstructed from the MATLAB code. We do not give exhaustive descriptions of matrix properties, or proofs of these properties; instead, in the comment lines we list a few key properties and give references where further details can be found.

With a few exceptions each of the 58 matrices satisfies the following requirements:

- It is a square matrix with one or more variable parameters, one of which is the dimension. Thus it is actually a parametrized family of matrices of arbitrary dimension.
- It is dense.
- It has some property that makes it of interest as a test matrix.

The first criterion is enforced because it is often desirable to explore the behaviour of a numerical method as parameters such as the matrix dimension vary. The third criterion is somewhat subjective, and the matrices presented here represent the author's personal choice. Note that we have omitted plausible matrices that we thought not "sufficiently different" from others in the collection. Although all but two of our test matrices are usually real, those with an arbitrary parameter can be made complex by choosing a non-real value for the parameter.

As well as their obvious application to research in matrix computations we hope that the matrices presented here will be useful for constructing test problems in other areas, such as optimization (see, for example, [3]) and ordinary differential equations.

We mention some other collections of test matrices that complement ours. The Harwell-Boeing collection of sparse matrices, largely drawn from practical problems, is presented by Duff, Grimes and Lewis [8], [9]. Bai [2] is building a collection of test matrices for the large-scale nonsymmetric eigenvalue problem. Zielke [46] gives various parametrized rectangular matrices of fixed dimension with known generalized inverses. Demmel and McKenney [7] present a suite of Fortran 77 codes for generating random square and rectangular matrices with prescribed singular values, eigenvalues, band structure, and other properties. This suite is part of the testing code for LAPACK [1]. Our focus is primarily on non-random matrices but we include a class of random matrices `randsvd` that has some of the features of the Demmel and McKenney test set.

Where possible, we have chosen the names of the test matrices eponymously, since it is easier to remember, for example, "the Kahan matrix", than "Example 3.8". For portability reasons we restrict all M-file names in the toolbox to eight characters (since this is the limit in the MSDOS operating system, under which the Microsoft Windows version of MATLAB runs). We have written a routine `matrix` that accesses the matrices by number rather than by name; this makes it easy to run experiments on the whole collection of matrices (with parameters other than the matrix dimension set to their default values.)

The matrices described here can be modified in various ways while still retaining some or all of their interesting properties. Among the many ways of constructing new test matrices from old are:

- Similarity transformations $A \leftarrow X^{-1}AX$.
- Unitary transformations $A \leftarrow UAV$, where $U^*U = V^*V = I$.
- Kronecker products $A \leftarrow A \otimes B$ or $B \otimes A$ (for which MATLAB has a routine `kron`).
- Powers $A \leftarrow A^k$.

For a discussion of these techniques, and others, see [15, Chapter 2]. Techniques for obtaining a triangular, orthogonal, or symmetric positive definite matrix that is related to a given matrix include

- Bandwidth reduction using unitary transformations (see toolbox routine `bandred`).
- LU, Cholesky, QR and polar decompositions (see `lu`, `chol`, `qr` and, from the toolbox, `cholp`, `ge`, `gecp` and `poldec`.)

See [14] for details of these techniques.

Another way to generate a new matrix is to perturb an existing one. One approach is to add a random perturbation. Another is to round the matrix elements to a certain number of binary places; this can be done using the toolbox routine `chop`.

Our programming style is as follows. Each M-file `foo` begins with comment lines that are displayed when the user types `help foo`. The first comment line, the H1 line, is a self-contained statement of the purpose of the routine; the H1 lines are searched and displayed by MATLAB's `lookfor` command (e.g., `lookfor toeplitz`). Any further comments and references follow a blank line and so are not displayed by `help`. As far as possible, every routine sets default values for any arguments that are not specified. In particular, for most test matrix routines `testmat`, `A = testmat(n)` is a valid way to generate an $n \times n$ matrix. In general we have strived for

conciseness, modularity, speed, and minimal use of temporary storage in our MATLAB codes. Hence, where possible, we used matrix or vector constructs instead of `for` loops and have used calls to existing M-files.

Some of those matrices that are banded with a small bandwidth are given the sparse storage format, to allow large matrices to be generated. The `full` function can be used to convert to non-sparse storage (e.g., `A = full(tridiag(32))`). We check for errors in parameters in some, but not all, cases. A few of the test matrix routines do not properly handle the dimension $n = 1$ (for example, they halt with an error, or return an empty matrix). We decided not to add extra code for this case, since the routines are unlikely to be called with $n = 1$.

Tables 5.1 and 5.2 provide a summary of the properties of the test matrices. The column headings have the following meanings:

Inverse: the inverse of the matrix is known explicitly.

Ill-cond: the matrix is ill-conditioned for some values of the parameters.

Rank: the matrix is rank-deficient for some values of the parameters (we exclude "trivial" examples such as `vand`, which is singular if its vector argument contains repeated points). Note that there are some matrices that are mathematically rank-deficient but behave as ill-conditioned full rank matrices in the presence of rounding errors; these are listed only as rank-deficient (for example, `chebspec`).

Symm: the matrix is symmetric for some values of the parameters.

Pos Def: the matrix is symmetric positive definite for some values of the parameters.

Orth: the matrix is orthogonal, or a diagonal scaling of an orthogonal matrix, for some values of the parameters.

Eig: something is known about the eigensystem (or the singular values), ranging from bounds or qualitative knowledge of the eigenvalues to explicit formulas for some or all eigenvalues and eigenvectors.

We summarise further interesting properties possessed by some of the matrices. Recall that A is a *Hankel matrix* if the anti-diagonals are constant ($a_{ij} = r_{i+j}$), *idempotent* if $A^2 = A$, *normal* if $A^*A = AA^*$ (or, equivalently, A is unitarily diagonalizable), *nilpotent* if $A^k = 0$ for some k, *involutary* if $A^2 = I$, *totally positive (nonnegative)* if the determinant of every submatrix is positive (nonnegative), and a *Toeplitz matrix* if the diagonals are constant ($a_{ij} = r_{j-i}$). A totally positive matrix has distinct, real and positive eigenvalues and its ith eigenvector (corresponding to the ith largest eigenvalue) has exactly $i - 1$ sign changes [12, Theorem 13, p. 105]; this property is important in testing regularization algorithms [16], [17]. See [28] for further details of these matrix properties.

defective: `chebspec, gallery, gear, jordbloc, triw`

Hankel: `dingdong, hilb, ipjfact`

Hessenberg: `chow, frank, grcar, ohess, randsvd`

idempotent: `invol`

involutary: `invol, orthog, pascal`

normal (but not symmetric or orthogonal): `circul`

Matrix	Inverse	Ill-cond	Rank	Symm	Pos Def	Orth	Eig
augment		√	√				
cauchy	√	√		√	√		
chebspec			√				√
chebvand		√				√	
chow			√				√
circul				√	√		√
clement	√		√	√			√
compan	√		√				√
condex		√					
cycol			√				
dingdong				√			√
dorr		√					
dramadah		√					
fiedler	√			√			√
forsythe	√	√					√
frank		√					√
gallery	√	√	√	√	√		√
gearm			√				√
gfpp	√	√					
grcar							√
hadamard	√					√	√
hanowa							√
hilb	√	√		√	√		
invhess	√	√		√	√		√
invol	√	√					√
ipjfact		√		√			
jordbloc	√	√	√				√
kahan	√	√	√				
kms	√	√		√	√		
krylov		√					

Table 5.1: Properties of the test matrices, A–K.

Matrix	Inverse	Ill-cond	Rank	Symm	Pos Def	Orth	Eig
lauchli		√					
lehmer	√			√	√		
lesp							√
lotkin	√	√					√
minij	√			√	√		√
moler	√	√		√	√		
neumann			√				√
ohess	√					√	√
orthog	√					√	√
parter							√
pascal	√	√		√	√		√
pdtoep	√	√	√	√	√		
pei	√	√		√	√		√
pentoep		√	√	√	√		
poisson	√			√	√		√
prolate		√		√	√		√
rando							
randsvd	√	√		√	√	√	
redheff							√
riemann							√
rschur		√					√
smoke	√						√
tridiag	√	√	√	√	√		√
triw	√	√					
vand	√	√					
wathen				√	√		√
wilk		√		√	√		√

Table 5.2: Properties of the test matrices, L–Z.

nilpotent: `chebspec, gallery`

rectangular: `chebvand, cycol, kahan, krylov, lauchli, rando, randsvd, triw, vand`

Toeplitz: `chow, dramadah, grcar, kms, parter, pentoep, prolate`

totally positive or totally nonnegative: `cauchy`[1], `hilb, lehmer, pascal, vand`[2]

tridiagonal: `clement, dorr, gallery, lesp, randsvd, tridiag, wilk`

inverse of a tridiagonal matrix: `kms, lehmer, minij`

triangular: `dramadah, jordbloc, kahan, pascal, triw`

Finally, we note that several of the test matrices are related to those supplied with MATLAB. The functions `hadamard` and `pascal` were in the first release of the toolbox and were subsequently included by The MathWorks in the MATLAB distribution. The toolbox version of `hadamard` is the same as the one in MATLAB 4.2 except for the addition of an H1 line, whereas the toolbox version of `pascal` contains more informative comment lines than the MATLAB 4.2 version and produces a different `pascal(n,2)` matrix[3] (but one that is still a cube root of the identity). The toolbox routine `compan` is more versatile than the MATLAB 4.2 version. Similarly, the toolbox routine `vand` is more versatile than MATLAB 4.2's `vander`. The toolbox version of `hilb` is coded differently and contains more informative comments than the one in MATLAB 4.2. The toolbox routine `augment` is similar to MATLAB 4.2's `spaugment`, but produces a non-sparse matrix instead of a sparse one. The toolbox function `cond` supports the 1, 2, ∞ and Frobenius norms, whereas MATLAB 4.2's `cond` supports only the 2-norm.

6 Visualization

The toolbox contains five routines for visualizing matrices. The routines can give insight into the properties of a matrix that is not easy to obtain by looking at the numerical entries. They also provide an easy way to generate pretty pictures!

The routine `see` displays a figure with four subplots (strictly speaking four "axes", in MATLAB terminology) in the format

`mesh(A)`	`mesh(pinv(A))`
`semilogy(svd(A))`	`fv(A)`

An example for the `chebvand` matrix is given in Figure 6.1. MATLAB's `mesh` command plots a three-dimensional, coloured, wire-frame surface, by regarding the entries of a matrix as specifying heights above a plane. We use `axis('ij')`, so that the coordinate system for the plot matches the (i, j) matrix element numbering. `pinv(A)` is the Moore–Penrose pseudo-inverse A^+ of A, which is the usual inverse when A is square and nonsingular. `semilogy(svd(A))` plots the singular values of A (ordered in decreasing size) on a logarithmic scale; the singular values are denoted by circles, which are joined by a solid line to emphasise the shape of the distribution. From Figure 6.1 we can see that `chebvand(8)` has a 2-norm condition number of about 10^5 and that the largest elements of its inverse are in the lower triangle. For a sparse MATLAB matrix, `see` simply displays a `spy` plot, which shows the sparsity pattern of the matrix. The user could,

[1] `cauchy(x,y)` is totally positive if $0 < x_1 < \cdots < x_n$ and $0 < y_1 < \cdots < y_n$ [39, p. 295].

[2] `vand(p)` is totally positive if the p_i satisfy $0 < p_1 < \cdots < p_n$ [12, p. 99].

[3] The new `pascal(n,2)` is generated by a call to `rot90` and is "reverse upper triangular" instead of "reverse lower triangular" as in the MATLAB 4.2 version.

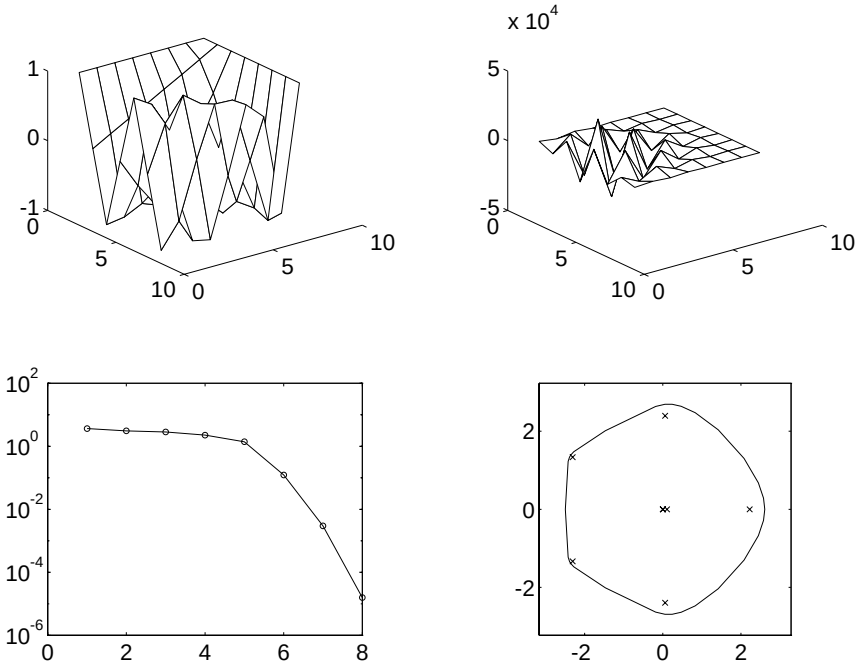


Figure 6.1: `see(chebvand(8))`.

of course, try `see(full(A))` for a sparse matrix, but for large dimensions the storage and time required would be prohibitive. Figure 6.2 displays the result of applying `see` to the Wathen matrix—a symmetric positive definite sparse matrix that comes from a finite element problem.

The routine `fv` plots the field of values of a square matrix $A \in \mathbf{C}^{n\times n}$ (also called the numerical range), which is the set of all Rayleigh quotients,

$$\left\{ \frac{x^*Ax}{x^*x} : 0 \neq x \in \mathbf{C}^n \right\};$$

the eigenvalues of A are plotted as crosses. The field of values is a convex set that contains the eigenvalues. It is the convex hull of the eigenvalues when A is a normal matrix. If A is Hermitian, the field of values is just a segment of the real line. For non-Hermitian A the field of values is usually two-dimensional and its shape and size gives some feel for the behaviour of the matrix. Trefethen [41] notes that the field of values is the largest reasonable answer to the question "Where in $\mathbf{C}$ does a matrix A 'live'?" and the spectrum is the smallest reasonable answer.

Some examples of field of values plots are given in Figure 6.3. The `circul` matrix is normal, hence its field of values is the convex hull of the eigenvalues. For an example of how the field of values gives insight into the problem of finding a nearest normal matrix see [35]. An excellent reference for the theory of the field of values is [29, Chapter 1].

The routine `gersh` plots the Gershgorin disks for an $A \in \mathbf{C}^{n\times n}$, which are the n disks

$$D_i = \{ z \in \mathbf{C} : |z - a_{ii}| \leq \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \}$$

in the complex plane. Gershgorin's theorem tells us that the eigenvalues of A lie in the union of the disks, and an extension of the theorem states that if k disks form a connected region that is isolated from the other disks, then there are precisely k eigenvalues in this region. Thus the size of the disks gives a feel for how nearly diagonal A is, and their locations give information

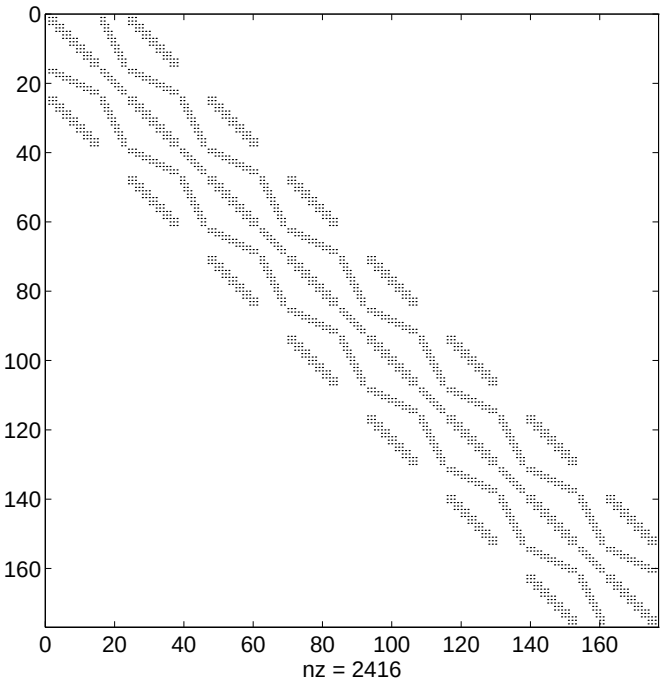


Figure 6.2: `see(wathen(7,7))`.

on where the eigenvalues lie in the complex plane. Four examples of Gershgorin disk plots are given in Figure 6.4; Gershgorin's theorem provides nontrivial information only for the third matrix, `ipjfact(8,1)`.

The last two routines, `ps` and `pscont`, are concerned with pseudospectra. The ϵ-pseudospectrum of a matrix $A \in \mathbf{C}^{n\times n}$ is defined, for a given $\epsilon > 0$, to be the set

$$\Lambda_\epsilon(A) = \{\, z : z \text{ is an eigenvalue of } A + E \text{ for some } E \text{ with } \|E\|_2 \le \epsilon \,\}.$$

In other words, it is the set of all complex numbers that are eigenvalues of $A + E$ for some perturbation E of 2-norm at most ϵ. For a normal matrix A the ϵ-pseudospectrum is the union of the balls of radius ϵ around the eigenvalues of A. For nonnormal matrices the ϵ-pseudospectrum can take a wide variety of shapes and sizes, depending on the matrix and how nonnormal it is. Pseudospectra play an important role in many numerical problems. For full details see the work of Trefethen—in particular, [40] and [41].

The routine `ps` plots an approximation to the ϵ-pseudospectrum $\Lambda_\epsilon(A)$, which it obtains by computing the eigenvalues of a given number of random perturbations of A. The eigenvalues are plotted as crosses and the pseudo-eigenvalues as dots. Arguments to `ps` control the number and type of perturbations. Figure 6.5 gives four examples of 10^{-3}-pseudospectra, all of which involve the pentadiagonal Toeplitz matrix `pentoep`.

Another characterization of $\Lambda_\epsilon(A)$, in terms of the resolvent $(zI - A)^{-1}$, is

$$\Lambda_\epsilon(A) = \{\, z : \|(zI - A)^{-1}\|_2 \ge \epsilon^{-1} \,\}.$$

An alternative way of viewing the pseudospectrum is to plot the function

$$f(z) = \|(zI - A)^{-1}\|_2^{-1} = \sigma_{\min}(zI - A)$$

over the complex plane, where $\sigma_{\min}$ denotes the smallest singular value [41]. The routine `pscont` plots $\log_{10} f(z)^{-1}$ and offers several ways to view the surface: by its contour lines alone, or as

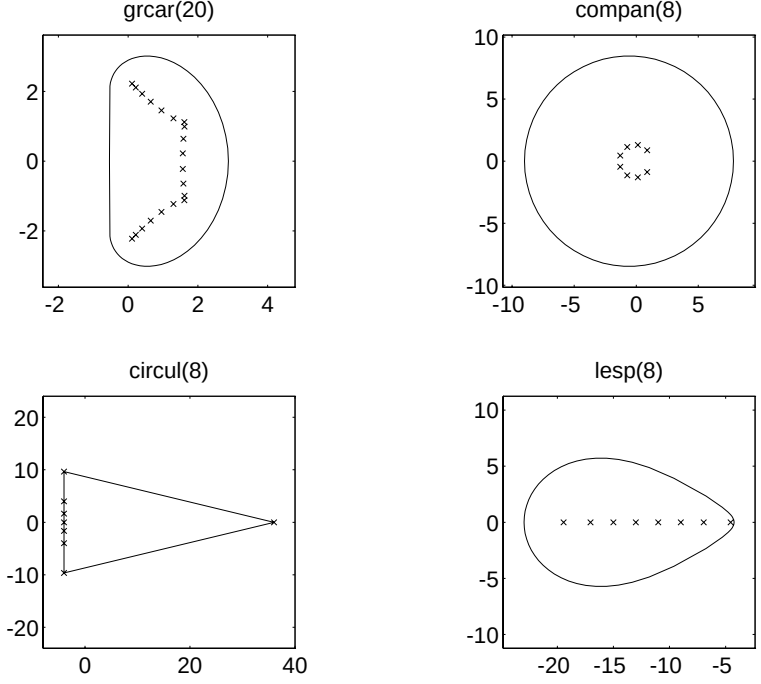


Figure 6.3: Fields of values (`fv`).

a coloured surface plot in two or three dimensions, with or without contour lines. (The two-dimensional plot is the view from directly above the surface.) Two different `pscont` views of the pseudospectra of the triangular matrix `triw(11)` are given in Figures 6.6 and 6.7. Since all the eigenvalues of this matrix are equal to 1, there is a single point where the resolvent is unbounded in norm—this is the "bottomless pit" in the pictures. The spike in Figure 6.7 should be infinitely deep; since `pscont` evaluates $f(z)$ on a finite grid, the spike has a finite depth dependent on the grid spacing. Also because of the grid spacing chosen, the contours are a little jagged. Various aspects of the plots can be changed from the MATLAB command line upon return from `pscont`; for example, the colour map (`colormap`), the shading (`shading`), and the viewing angle (`view`). For Figure 6.6 we set `shading interp` and `colormap copper`.

Both pseudospectrum routines are computationally intensive, so the defaults for the arguments are chosen to produce a result in a reasonable time (under 20 seconds on a SPARC-2 processor or equivalent); for plots that reveal reasonable detail it is usually necessary to override the defaults.

7 Direct Search Optimization

The toolbox contains three multivariate direct search maximization routines `mdsmax`, `adsmax` and `nmsmax`, together with a demonstration function `fdemo` on which to try them. The routines are competitors to `fmins`, which is supplied with MATLAB (but `fmins` minimizes rather than maximizes). `nmsmax` is actually a modified version of `fmins` with the same interface as `mdsmax` and `adsmax`.

`mdsmax`, `adsmax` and `nmsmax` are direct search methods (as is `fmins`), that is, they attempt to maximize a real function f of a vector argument x using function values only. `mdsmax` uses the multidirectional search method, `adsmax` uses the method of alternating directions, and `nmsmax` uses the Nelder–Mead simplex method. In general, `mdsmax` and `nmsmax` can be expected to perform better than `adsmax` since they use a more sophisticated method.

These routines were developed during the work described in [23].

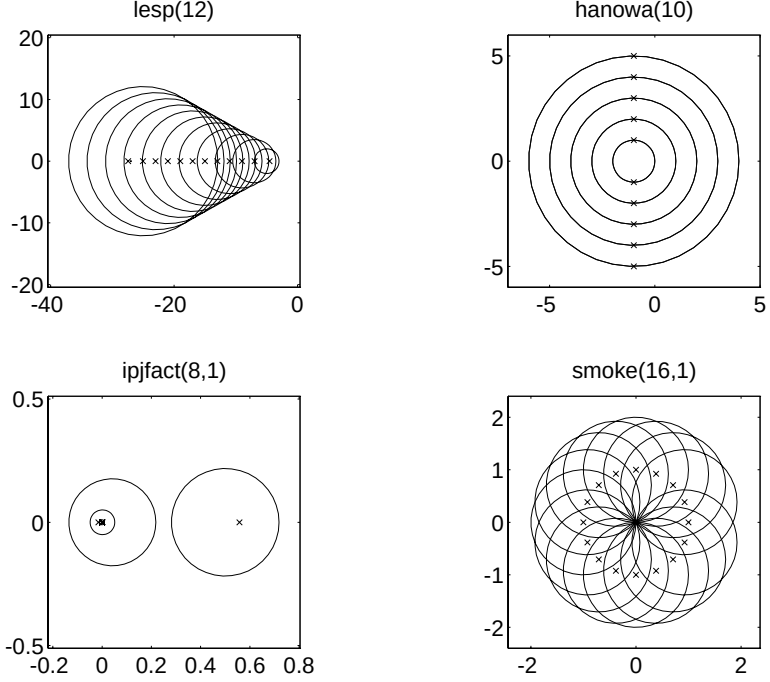


Figure 6.4: Gershgorin disks (`gersh`).

Note: These routines, like `fmins`, are not competitive with more sophisticated methods such as (quasi-)Newton methods when applied to smooth problems. They are at their best when applied to non-smooth problems such as the one in the example below.

The routines are fully documented in their leading comment lines, but it is appropriate to add here a few comments about the format of the output and the use of the `savit` argument.

`mdsmax` produces output to the screen (this can be suppressed by setting the input argument `stopit(5) = 0`). The output is illustrated by

```
Iter. 10,  inner =  2,  size = -4,  nf = 401,  f = 4.7183e+001  (51.0%)
```

The means that on the tenth iteration, two inner iterations were required, and at the end of the iteration the simplex edges were 2^{-4} times the length of those of the initial simplex. Further, `nf` is the total number of function evaluations so far, `f` is the current highest function value, and the percentage increase in function value over the tenth iteration is 51%.

The output produced by `adsmax` is similar to that of `mdsmax` and is illustrated by the following extract from the start of the second outer iteration:

```
Iter  2  (nf = 146)
Comp. =  1,  steps = 12,  f = 1.5607e+000  (0.4%)
```

`Comp` denotes the component of x being varied on the current stage and steps is the number of steps in the crude line search for this stage.

The output from `nmsmax` is also similar to that from `mdsmax`, but only iterations on which an increase in the function value is achieved are reported.

In all three routines, if a non-empty fourth input argument string `savit` is present then at the end of each iteration the following "snapshot" is written to the file specified by `savit`: the largest function value found so far, `fmax`, the point at which it is achieved, `x`, and the total number of function evaluations, `nf`. This option enables the user to abort an optimization, load and examine `x`, `fmax` and nf using MATLAB's `load` command, and then possibly restart the optimization at `x`.

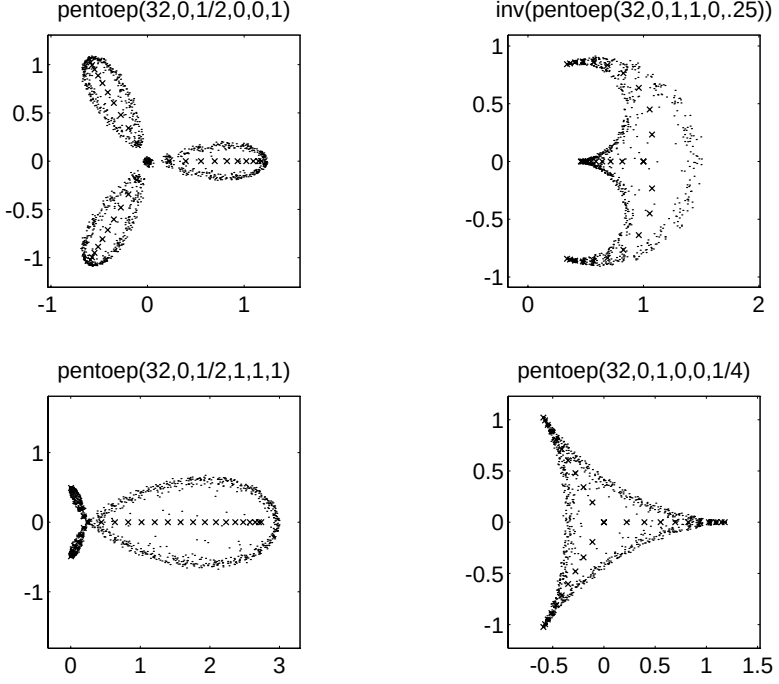


Figure 6.5: Pseudospectra (`ps`).

One further point worth mentioning is that `mdsmax`, `adsmax` and `nmsmax` always call the function `f` to be maximized with an argument of the same dimensions (or shape) as the starting value `x0`. Similarly, the output argument `x` and the variable `x` in `savit` saves have the same shape as `x0`. This feature is very convenient when `f` is a function of a matrix, as in the example below.

To facilitate a quick test of the routines the toolbox includes a function `fdemo`, which takes a square matrix argument `A` and evaluates the ratio of `rcond(A)` to the exact 1-norm condition number of the matrix `A`. (`rcond` is MATLAB's built-in condition estimator.) Making `fdemo` large corresponds to finding a matrix where `rcond` returns a poor condition number estimate.

Here is an extract from output produced by MATLAB 4.2b on a 486DX PC; similar output should be obtained on other machines (a machine that uses IEEE arithmetic will probably produce identical results). In this example MDSMAX rapidly achieves a function value of over 100, but ADSMAX and NMSMAX make only slow progress and terminate when the default convergence tests are satisfied.

```
>> A = hilb(5);  % Starting matrix.
>> B = mdsmax('fdemo', A);

f(x0) = 1.3596e+000
Iter.  1,  inner =  0,  size =  0,  nf =  26,  f = 1.3648e+000  (0.4%)
Iter.  2,  inner =  1,  size =  1,  nf =  76,  f = 1.4493e+000  (6.2%)
Iter.  3,  inner =  1,  size =  1,  nf = 126,  f = 1.4954e+000  (3.2%)
Iter.  4,  inner =  1,  size =  2,  nf = 176,  f = 1.5935e+000  (6.6%)
Iter.  5,  inner =  1,  size =  2,  nf = 226,  f = 1.7589e+000  (10.4%)
Iter.  6,  inner =  1,  size =  2,  nf = 276,  f = 3.7883e+000  (115.4%)
Iter.  7,  inner =  1,  size =  3,  nf = 326,  f = 3.1601e+001  (734.2%)
Iter.  8,  inner =  1,  size =  4,  nf = 376,  f = 5.3514e+001  (69.3%)
Iter.  9,  inner =  2,  size =  3,  nf = 476,  f = 5.3888e+001  (0.7%)
```

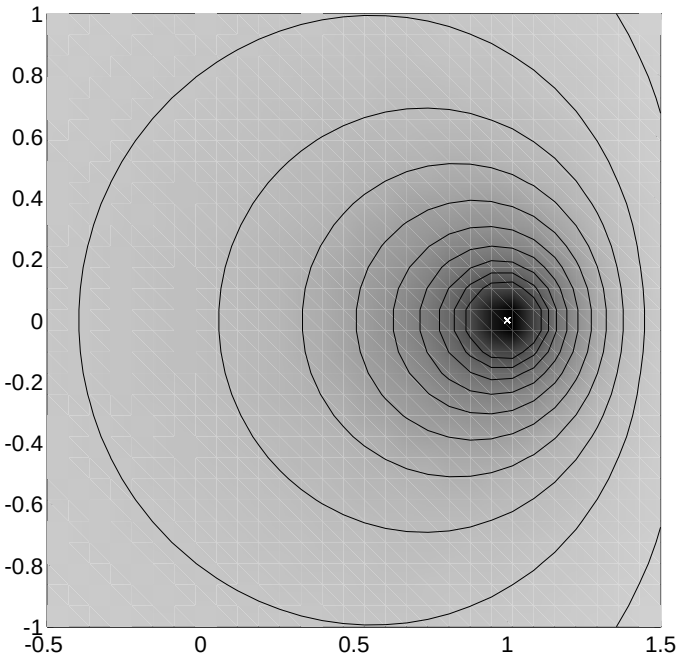


Figure 6.6: `pscont(triw(11), 0, 30, [-0.5 1.5 -1 1])`.

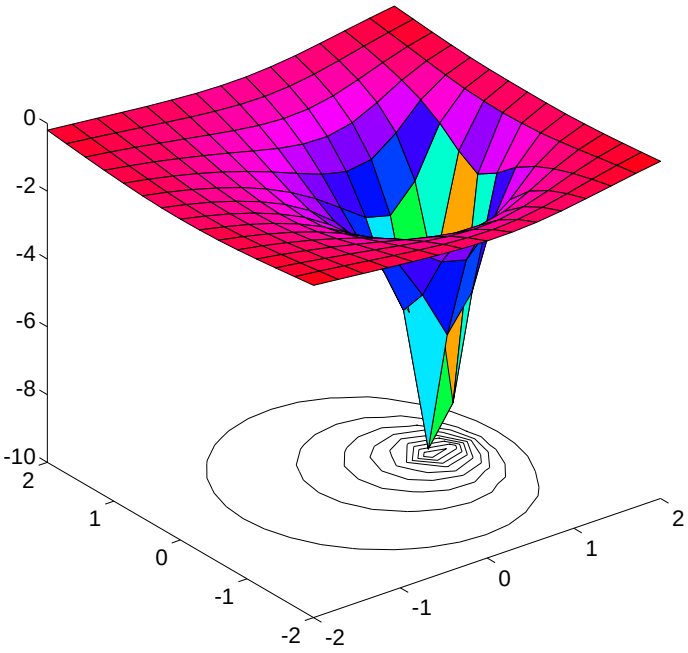


Figure 6.7: `pscont(triw(11), 2, 15, [-2 2 -2 2])`.

```
Iter. 10,  inner =  1,  size =  3,  nf = 526,  f = 9.9432e+001  (84.5%)
Iter. 11,  inner =  3,  size =  1,  nf = 676,  f = 1.0414e+002  (4.7%)
Iter. 12,  inner =  1,  size =  0,  nf = 726,  f = 1.0462e+002  (0.5%)
Iter. 13,  inner =  1,  size = -1,  nf = 776,  f = 1.0579e+002  (1.1%)
Iter. 14,  inner =  1,  size =  0,  nf = 826,  f = 1.0872e+002  (2.8%)
Iter. 15,  inner =  1,  size =  0,  nf = 876,  f = 1.0981e+002  (1.0%)
Iter. 16,  inner =  1,  size = -1,  nf = 926,  f = 1.1082e+002  (0.9%)
Iter. 17,  inner =  1,  size = -1,  nf = 976,  f = 1.1238e+002  (1.4%)
Iter. 18,  inner =  1,  size = -1,  nf = 1026,  f = 1.1334e+002  (0.9%)
Iter. 19,  inner =  1,  size = -1,  nf = 1076,  f = 1.1389e+002  (0.5%)
Iter. 20,  inner =  1,  size = -1,  nf = 1126,  f = 1.1470e+002  (0.7%)
Iter. 21,  inner =  1,  size = -1,  nf = 1176,  f = 1.1773e+002  (2.6%)
Iter. 22,  inner =  1,  size =  0,  nf = 1226,  f = 1.2174e+002  (3.4%)
Iter. 23,  inner =  1,  size = -1,  nf = 1276,  f = 1.2317e+002  (1.2%)
Iter. 24,  inner =  1,  size =  0,  nf = 1326,  f = 1.2682e+002  (3.0%)
Iter. 25,  inner =  2,  size = -1,  nf = 1426,  f = 1.2794e+002  (0.9%)
Iter. 26,  inner =  1,  size = -2,  nf = 1476,  f = 1.2830e+002  (0.3%)
Iter. 27,  inner =  1,  size = -1,  nf = 1526,  f = 1.3185e+002  (2.8%)
Iter. 28,  inner =  1,  size = -1,  nf = 1576,  f = 1.3553e+002  (2.8%)
Iter. 29,  inner =  2,  size = -3,  nf = 1676,  f = 1.3665e+002  (0.8%)
Iter. 30,  inner =  2,  size = -4,  nf = 1776,  f = 1.3749e+002  (0.6%)
Iter. 31,  inner =  1,  size = -5,  nf = 1826,  f = 1.3761e+002  (0.1%)
Iter. 32,  inner =  1,  size = -4,  nf = 1876,  f = 1.3833e+002  (0.5%)
Iter. 33,  inner =  1,  size = -5,  nf = 1926,  f = 1.3852e+002  (0.1%)
Simplex size 5.7156e-004 <= 1.0000e-003...quitting

>> format short e, format compact
>> B

B =
  3.4019e+000  2.0181e+000  7.5303e+000  1.7681e+000 -3.5852e+000
 -6.8208e+000  1.8514e+000  1.7681e+000  1.7181e+000  1.6847e+000
 -6.2348e-001  1.7681e+000  1.6960e+000  1.6847e+000 -1.1421e+001
  3.2707e+000  1.7181e+000  1.6847e+000  1.6609e+000  1.6431e+000
  3.2477e+001  1.6847e+000  1.8156e+000  1.6431e+000  1.6292e+000

>> % Confirm that the returned B defines a matrix where RCOND does badly.
>> [1/rcond(B) cond(B,1) rcond(B)*cond(B,1)]

ans =
  1.4329e+001  1.9849e+003  1.3852e+002

>> B = adsmax('fdemo', A);

f(x0) = 1.3596e+000
Iter  1  (nf =  1)
Comp. =  1,  steps = 10,  f = 1.3609e+000  (0.1%)
Comp. =  2,  steps =  2,  f = 1.3609e+000  (0.0%)
Comp. =  3,  steps =  0,  f = 1.3609e+000  (0.0%)
Comp. =  4,  steps =  0,  f = 1.3609e+000  (0.0%)
```

```
Comp. =  5,  steps =  0,  f = 1.3609e+000  (0.0%)
Comp. =  6,  steps =  8,  f = 1.3617e+000  (0.1%)
Comp. =  7,  steps =  1,  f = 1.3617e+000  (0.0%)
Comp. =  8,  steps =  0,  f = 1.3617e+000  (0.0%)
Comp. =  9,  steps =  0,  f = 1.3617e+000  (0.0%)
Comp. = 10,  steps =  0,  f = 1.3617e+000  (0.0%)
Comp. = 11,  steps =  7,  f = 1.3618e+000  (0.0%)
Comp. = 12,  steps =  2,  f = 1.3618e+000  (0.0%)
Comp. = 13,  steps =  0,  f = 1.3618e+000  (0.0%)
Comp. = 14,  steps =  0,  f = 1.3618e+000  (0.0%)
Comp. = 15,  steps =  0,  f = 1.3618e+000  (0.0%)
Comp. = 16,  steps = 10,  f = 1.3944e+000  (2.4%)
Comp. = 17,  steps =  5,  f = 1.4223e+000  (2.0%)
Comp. = 18,  steps =  2,  f = 1.4395e+000  (1.2%)
Comp. = 19,  steps =  7,  f = 2.2811e+000  (58.5%)
Comp. = 20,  steps =  2,  f = 2.3077e+000  (1.2%)
Comp. = 21,  steps =  4,  f = 2.3166e+000  (0.4%)
Comp. = 22,  steps =  0,  f = 2.3166e+000  (0.0%)
Comp. = 23,  steps =  0,  f = 2.3166e+000  (0.0%)
Comp. = 24,  steps =  5,  f = 2.5066e+000  (8.2%)
Comp. = 25,  steps =  0,  f = 2.5066e+000  (0.0%)
Iter  2  (nf = 108)
...
Iter  7  (nf = 394)
Comp. =  1,  steps =  0,  f = 4.5918e+000  (0.0%)
Comp. =  2,  steps =  0,  f = 4.5918e+000  (0.0%)
Comp. =  3,  steps =  0,  f = 4.5918e+000  (0.0%)
Comp. =  4,  steps =  1,  f = 4.5936e+000  (0.0%)
Comp. =  5,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. =  6,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. =  7,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. =  8,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. =  9,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 10,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 11,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 12,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 13,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 14,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 15,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 16,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 17,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 18,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 19,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 20,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 21,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 22,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 23,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 24,  steps =  0,  f = 4.5936e+000  (0.0%)
Comp. = 25,  steps =  0,  f = 4.5936e+000  (0.0%)
Function values 'converged'...quitting
```

```
>> B = nmsmax('fdemo', A);

f(x0) = 1.3596e+000
Iter.  1,  how = initial    nf =  26,  f = 1.3648e+000  (0.4%)
Iter.  2,  how = shrink,    nf =  53,  f = 1.4331e+000  (5.0%)
Iter. 10,  how = shrink,    nf = 141,  f = 1.4421e+000  (0.6%)
Iter. 11,  how = shrink,    nf = 168,  f = 1.5257e+000  (5.8%)
Iter. 12,  how = shrink,    nf = 195,  f = 1.6796e+000  (10.1%)
Iter. 16,  how = contract,  nf = 200,  f = 1.7651e+000  (5.1%)
Iter. 20,  how = contract,  nf = 205,  f = 1.8961e+000  (7.4%)
Iter. 35,  how = shrink,    nf = 277,  f = 1.9934e+000  (5.1%)
Iter. 36,  how = reflect,   nf = 279,  f = 2.0306e+000  (1.9%)
Iter. 37,  how = reflect,   nf = 281,  f = 2.0734e+000  (2.1%)
Iter. 56,  how = shrink,    nf = 331,  f = 2.1540e+000  (3.9%)
Simplex size 6.5338e-004 <= 1.0000e-003...quitting
```

For more on the use of direct search in "automatic error analyis" see [23] or [26, Ch. 24].

8 Miscellaneous Routines

In addition to the test matrices and visualization routines, the Test Matrix Toolbox provides several routines that can be used to manipulate matrices or compute matrix functions or decompositions.

The decomposition functions offered are as follows.

• `cgs` and `mgs` apply the classical and modified Gram–Schmidt methods, respectively, to a matrix $A \in \mathbf{C}^{m\times n}$ of full rank n, to produce the factorization $A = QR$, where $Q \in \mathbf{C}^{m\times n}$ has orthonormal columns and $R \in \mathbf{C}^{n\times n}$ is upper triangular. The methods are identical in exact arithmetic but produce different results in the presence of rounding errors [26].

• `cholp` computes the Cholesky factorization with pivoting $\Pi^T A \Pi = R^* R$ of a Hermitian positive semi-definite matrix $A \in \mathbf{C}^{n\times n}$. Here, R is upper triangular with nonnegative diagonal elements and Π is a permutation matrix chosen to permute the largest diagonal element to the pivot position at each stage of the reduction (see [14, Section 4.2.9]). For the usual Cholesky factorization, as computed by `chol`, Π is the identity. Whereas `chol` can break down when presented with a Hermitian positive semi-definite matrix that is singular, `cholp` will always succeed.

• `cod` computes the complete orthogonal decomposition of a rank r matrix $A \in \mathbf{C}^{m\times n}$,

$$A = U \begin{bmatrix} R & 0 \\ 0 & 0 \end{bmatrix} V,$$

where $R \in \mathbf{C}^{r\times r}$ is upper triangular and $U \in \mathbf{C}^{m\times m}$ and $V \in \mathbf{C}^{n\times n}$ are unitary. The criterion used to define the numerical rank is a simple one based on the diagonal elements of the upper triangular matrix from the QR factorization with column pivoting. The complete orthogonal decomposition is an important tool in rank-deficient least squares problems [14, Sec. 5.5.2], [37].

• `diagpiv` implements the diagonal pivoting factorization with partial pivoting of a symmetric matrix A. It produces the factorization $PAP^T = LDL^T$, where P is a permutation, L is unit lower triangular, and D is block diagonal with 1×1 and 2×2 diagonal blocks. The Bunch–Kaufman partial pivoting strategy is used [4].

• `ge` implements Gaussian elimination without pivoting. This routine is similar to `lu`, except that no row interchanges are done. Thus the routine computes, if possible, the LU factorization

$A = LU$ of $A \in \mathbf{C}^{n\times n}$. The routine is of pedagogical interest, but also of practical interest because there are certain classes of $Ax = b$ problem where a more accurate solution is obtained from LU factorization when there are no row or column interchanges [20], [26].

• `gecp` implements Gaussian elimination with complete pivoting. Thus it computes the factorization $PAQ = LU$ of $A \in \mathbf{C}^{n\times n}$, where P and Q are permutation matrices and L and U are lower and upper triangular, respectively. At the kth stage of the reduction of A to triangular form, row and column interchanges are used to bring the element of largest absolute value in the active submatrix to the pivot position (k,k) [14, Sec. 3.4.8].

• `gj` implements Gauss–Jordan elimination for solving a nonsingular linear system $Ax = b$.

• `poldec` computes the polar decomposition $A = UH \in \mathbf{C}^{m\times n}$, where $H \in \mathbf{C}^{n\times n}$ is Hermitian positive semi-definite and $U \in \mathbf{C}^{m\times n}$ has orthonormal columns or rows, according as $m \geq n$ or $m \leq n$. The polar decomposition is a generalization of the polar representation $z = re^{i\theta}$ for complex numbers. The factor U has the property that when $m \geq n$ it is the nearest matrix with orthonormal columns to A for both the 2-norm and the Frobenius norm:

$$\|A - U\| = \min\{\, \|A - Q\| : Q^*Q = I,\ Q \in \mathbf{C}^{m\times n} \,\}.$$

For more details see [18] or [28].

• `signm` computes the matrix sign decomposition $A = SN \in \mathbf{C}^{n\times n}$, where $S = \mathrm{sign}(A)$ is the matrix sign function [25]. If A has the Jordan canonical form

$$A = XJX^{-1} = X \begin{bmatrix} J_1 & 0 \\ 0 & J_2 \end{bmatrix} X^{-1},$$

where the eigenvalues of J_1 lie in the open left half-plane and those of J_2 lie in the open right half-plane, then

$$\mathrm{sign}(A) = X \begin{bmatrix} -I & 0 \\ 0 & I \end{bmatrix} X^{-1}.$$

(The sign function is not defined if A has any pure imaginary eigenvalues). The matrix sign function has several applications and is the subject of much recent research; see [25] for details and further references. Since $\mathrm{sign}(A)^2 = I$, `signm` provides one way to generate involutary matrices.

The toolbox contains further miscellaneous routines, including the following ones.

`bandred`: bandwidth reduction by unitary transformation (called by `randsvd`).

`comp`: forms comparison matrices.

`cond`: generalizes the `cond` function supplied with MATLAB 4.0 to work with the 1, ∞ and Frobenius norms (for square matrices) as well as the 2-norm.

`qmult`: Premultiplies a matrix by a random real orthogonal matrix from the Haar distribution (called by `randsvd`).

`seqa`, `seqm`: form additive or multiplicative sequences.

`sparsify`: randomly sets elements of a matrix to zero.

`pnorm`: estimates the p-norm of a matrix for $1 \leq p \leq \infty$ (MATLAB's `norm` works only for $p = 1, 2, \infty$, `'fro'`).

`eigsens`: evaluates the Wilkinson condition numbers for the eigenvalues of a matrix.

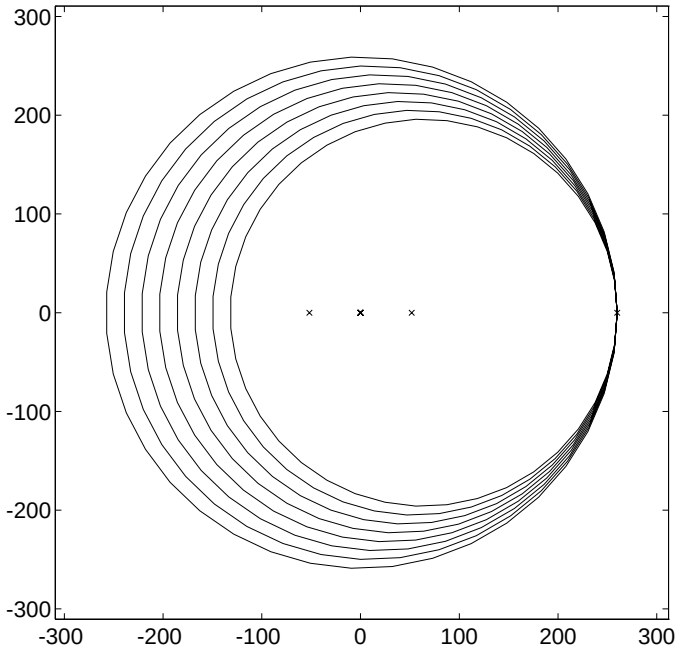


Figure 9.1: Gershgorin disks for `magic(8)`.

9 Examples

In this section we give examples of the use of the toolbox and explain some of the interesting properties of magic squares and the Frank matrix.

9.1 Magic Squares

In the winter 1993 MathWorks Newsletter, Moler described some of the fascinating properties of magic squares, as embodied in MATLAB's `magic` function [32]. Some further properties can be illustrated with the aid of the toolbox. Recall that a magic square is an $n \times n$ matrix containing the integers from 1 to n^2 whose row and column sums are all the same. Let μ_n denote the magic sum of `magic(n)` (thus, $\mu_n = n(n^2+1)/2$).

Moler pointed out that the largest singular value of `A = magic(n)` (namely `max(svd(A))`) is μ_n, but left the proof as an exercise. The largest singular value of A is its 2-norm, so the problem is to prove that $\|A\|_2 = \mu_n$. This leads naturally to the question of what is the p-norm of a magic square, for any p between 1 and ∞. The Hölder p-norm of an $m \times n$ matrix A is defined by

$$\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}, \tag{9.1}$$

where $p \geq 1$ and $\|x\|_p = (\sum_{i=1}^{n} |x_i|^p)^{1/p}$. We can investigate the p-norm of a magic square using the toolbox function `pnorm`, which computes an *estimate* of $\|A\|_p$ using a generalization of the power method.

```
for p = [1 1.5 2 exp(1) pi 10 inf]
    fprintf(' %9.4f    %9.4f\n', p, pnorm(magic(10),p))
end
    1.0000     505.0000
    1.5000     504.9968
    2.0000     504.9968
```

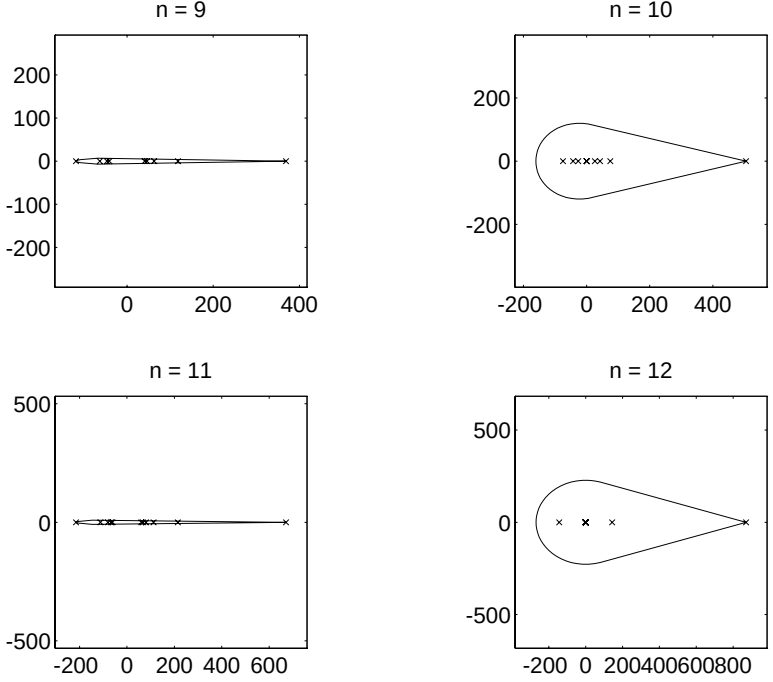


Figure 9.2: Field of values for `magic(n)`.

```
    2.7183      504.9971
    3.1416      504.9997
   10.0000      504.9988
       Inf      505.0000
```

All the p-norms in this example are very close to $\mu_{10} = 505$. Since the default convergence tolerance for `pnorm` is 10^{-4}, the exact p-norms could all be 505, as far as we can tell from the estimates. In fact, $\|A\|_p \equiv \mu_n$ for all $1 \leq p \leq \infty$. The proof relies on the convexity of the p-norm, which yields the inequality (see, [22], for example)

$$\|A\|_p \leq \|A\|_1^{1/p} \|A\|_\infty^{1-1/p}.$$

(This inequality is well-known for $p = 2$.) For a magic square, $\|A\|_1 = \|A\|_\infty = \mu_n$, so the inequality gives $\|A\|_p \leq \mu_n$. But by taking x in (9.1) to be the vector of all ones, we see that $\|A\|_p \geq \mu_n$, and so it follows that $\|A\|_p = \mu_n$. This result is actually a special case of an apparently little-known 1962 result of Stoer and Witzgall, which says that the norm of a doubly stochastic matrix is 1 for any norm subordinate to a permutation-invariant absolute vector norm [38].

To estimate the eigenvalues of `magic(n)` we can apply Gershgorin's theorem. Unfortunately, the results are not very informative because the Gershgorin disks are all approximately the same, as is clear from the structure of the matrix; see Figure 9.1.

In his article, Moler pointed out that the function `magic` uses different algorithms for odd n, even n divisible by 4, and even n not divisible by 4. He gave four `mesh` plots to illustrate the difference. Another approach is to look at the fields of values—see Figure 9.2. The plot for $n = 10$ reflects the fact that `magic(n)` has rank $n/2 + 2$ when n is even and not divisible by 4—there are only 6 eigenvalues away from the origin (`magic(10)` is diagonalizable). For n divisible by 4 the rank is only 3.

9.2 The Frank Matrix

A famous test matrix for eigensolvers is the $n \times n$ upper Hessenberg matrix F_n introduced by Frank in 1958 [11], illustrated for $n = 8$ by

```
F = frank(8)

F =

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

In evaluating three eigenvalue algorithms Frank found that this matrix "gives our selected procedures difficulties", and that "accuracy was lost in the smaller roots". The difficulties encountered by Frank's codes were shown by Wilkinson [44, Section 8], [45, pp. 92–93] to be caused by the inherent sensitivity of the eigenvalues to perturbations in the matrix.

The Frank matrix is interesting to analyze using MATLAB. The toolbox function `eigsens` evaluates the Wilkinson eigenvalue condition numbers, which are the reciprocals of the cosines of the angles between the left and right eigenvectors:

```
F = frank(10);
[V, D, s] = eigsens(F); d = diag(D); [x, k] = sort(d);
[d(k) s(k)]  % Eigenvalue followed by its condition number.
ans =

  3.9100e-002  1.4082e+005
  6.7743e-002  2.5897e+005
  1.2426e-001  1.4103e+005
  2.5692e-001  2.4028e+004
  6.1859e-001  1.1837e+003
  1.6166e+000  3.1920e+001
  3.8922e+000  2.2871e+000
  8.0476e+000  1.8287e+000
  1.4762e+001  3.0303e+000
  2.5575e+001  2.3393e+000
```

The output shows that the condition numbers grow, almost monotonically, as the eigenvalues decrease in size—in other words, the smallest eigenvalues are the most sensitive to perturbations in the matrix. The varying eigenvalue sensitivities can also be seen from pseudospectral plots. Figure 9.3 shows the 0.1-pseudospectrum, which shows that perturbations to F_{10} of 2-norm at most 0.1 have the greatest effect on the smallest eigenvalues. Another view is provided by Figure 9.4, for which we set `colormap hot`.

Further insight into the eigenvalues of F_n can be obtained by looking at its characteristic polynomial:

```
poly(F)
```

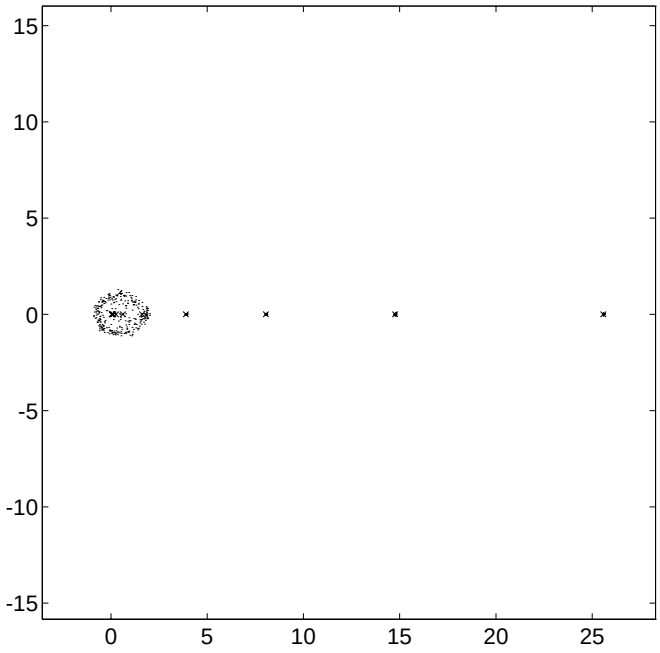


Figure 9.3: `ps(frank(10), 50, 1e-1, 0, 1)`.

```
ans =

  1.0e+004 *

  Columns 1 through 7

    0.0001   -0.0055    0.1035   -0.8310    2.9505   -4.5297    2.9505

  Columns 8 through 11

   -0.8310    0.1035   -0.0055    0.0001
```

The coefficients seem to be palindromic! As a check we use the function `charpoly` from MATLAB 4's Maple Symbolic Toolbox [5] to compute the characteristic polynomial exactly:

```
charpoly(F)

ans =

1-55*x+1035*x^2-8310*x^3+29505*x^4-45297*x^5+29505*x^6-8310*x^7+1035*x^8-55*x^9+x^10
```

Any matrix whose characteristic polynomial π_n has a palindromic coefficient vector has eigenvalues occurring in reciprocal pairs, since $\pi_n(\lambda) = \lambda^n \pi_n(1/\lambda)$. In particular, it has determinant 1, and 1 is an eigenvalue when n is odd. We can check the determinant property numerically:

```
F = frank(20); [det(F) det(F')]

ans =
```

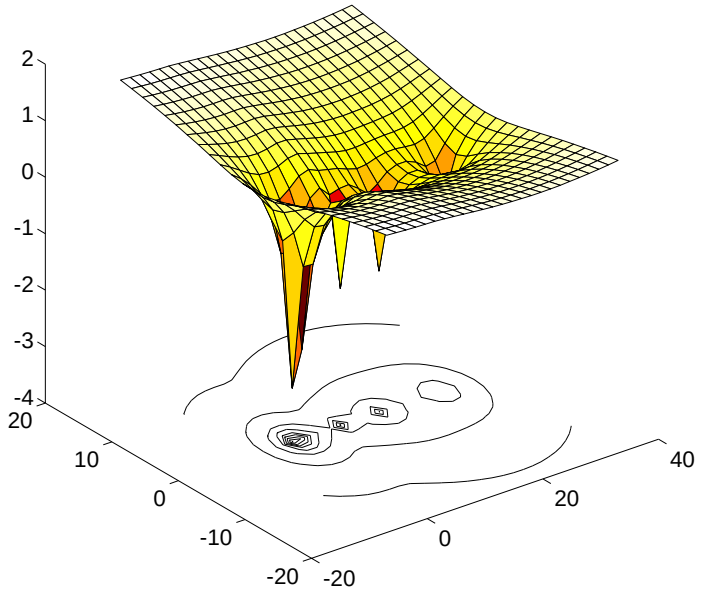


Figure 9.4: `pscont(frank(10), 2, 25, [-7 33 -20 20])`.

```
     1   -14

F = frank(25); [det(F) det(F')]

ans =

            1   -48886168
```

Since $\det(A) = \det(A^T)$ for any matrix A, the output is mathematically incorrect. The reason is that rounding errors influence MATLAB's evaluation of $\det(F_n^T)$ much more than its evaluation of $\det(F_n)$; an illuminating discussion of this phenomenon is given by Frank [11] and Wilkinson [44, Section 8], [45, pp. 92–93]. The extreme sensitivity of $\det(F_n)$ to perturbations in F_n is easy to see: if we change the $(1,n)$ element from 1 to $1+\epsilon$, then $\det(F_n)$ changes from 1 to $1 + (-1)^n(n-1)!\epsilon$.

The inverse of F_n is lower Hessenberg. This can be seen using the following representation of F_n noted by Rutishauser [36, Sec. 9]:

$$F_n = PC_nP,$$

where P is the identity with the order of its columns reversed (`I = eye(n); P = I(:, n:-1:1)` in MATLAB notation) and

$$C_n = \begin{bmatrix} 1 & & & & \\ -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & -1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 & 1 & & & \\ & 1 & 2 & & \\ & & 1 & \ddots & \\ & & & \ddots & n-1 \\ & & & & 1 \end{bmatrix}.$$

(By manipulating the identity $\det(F_n - \lambda I) = \det(C_n - \lambda I)$ the reciprocal pair property of the eigenvalues can be proved; cf. [42].) As an illustration, here is the exact inverse as returned

by the Maple Symbolic Toolbox (the MATLAB function `inv` produces nonzero, but tiny, upper triangular elements because of rounding errors):

```
inverse(frank(8))

ans =

[    1,    -1,    0,    0,   0,  0,  0,  0]
[   -7,     8,   -1,    0,   0,  0,  0,  0]
[   42,   -48,    7,   -1,   0,  0,  0,  0]
[ -210,   240,  -35,    6,  -1,  0,  0,  0]
[  840,  -960,  140,  -24,   5, -1,  0,  0]
[-2520,  2880, -420,   72, -15,  4, -1,  0]
[ 5040, -5760,  840, -144,  30, -8,  3, -1]
[-5040,  5760, -840,  144, -30,  8, -3,  2]
```

In his 1958 paper, Frank commented

> "At the moment, the largest matrices resolved on the [Univac] 1103A are two 20-order matrices, one real symmetric and one complex. In both cases computing time was approximately one hour, and 6–8 places of accuracy were obtained."

The complete eigensystem of a complex 20×20 matrix A is found in under a second by the MATLAB command `eig(A)` on the workstation used for the examples reported here! This improvement over Frank's timing is attributable not only to hardware advances but also to an algorithmic breakthrough: `eig` uses the QR algorithm, which was not available to Frank.

9.3 Numerical Linear Algebra

The Test Matrix Toolbox M-files embody some well-known and other not so well-known results from numerical linear algebra.

The function `gfpp` generates $n \times n$ matrices that produce the maximum growth of 2^{n-1} for Gaussian elimination with partial pivoting; these include Wilkinson's classic example [45, p. 212]

```
gfpp(7)

ans =

     1     0     0     0     0     0     1
    -1     1     0     0     0     0     1
    -1    -1     1     0     0     0     1
    -1    -1    -1     1     0     0     1
    -1    -1    -1    -1     1     0     1
    -1    -1    -1    -1    -1     1     1
    -1    -1    -1    -1    -1    -1     1
```

as well as all members of the "2^{n-1} class" described by Higham and Higham [27]. The following extract uses the toolbox routine `gecp` to evaluate the growth factor for complete pivoting on the Wilkinson matrix.

```
n = 20; A = gfpp(n);
[L, U] = lu(A);    % Partial pivoting.
[max(max(abs(U))) / max(max(abs(A)))  2^(n-1)] % Approximation to growth factor.

ans =

      524288      524288

% Complete pivoting using toolbox routine GECP.
[L, U, P, Q, rho] = gecp(A); rho

rho =

     2
```

As the output shows, complete pivoting is perfectly stable for these matrices. However, several of the matrices produced by `orthog` yield relatively large growth for complete pivoting: growth of order $n/2$ for real data, or n for a particular complex matrix [27].

```
n = 50;
for k = [-2 -1 1 2 3]
    A = orthog(n, k);
    [L, U, P, Q, rho] = gecp(A);
    fprintf(' %g\n', rho)
end
 25.3116
 24.7028
 25.6214
 25.3296
 50

A = hadamard(64);
[L, U, P, Q, rho] = gecp(A); rho

rho =

    64
```

It is easy to show that complete pivoting suffers growth of at least n for an $n \times n$ Hadamard matrix. However, Hadamard matrices do not exist for all n.

The MATLAB function `rcond` computes an upper bound for $\kappa_1(A)^{-1} = (\|A\|_1\|A^{-1}\|_1)^{-1}$ using the LINPACK condition estimation algorithm. Although this algorithm is very reliable in general, parametrized matrices are known for which it can perform arbitrarily badly [6]. Here are two examples, from the toolbox routine `condex`. The underestimation ratio is approximately the same in both examples, but the second is probably the more serious because `rcond` does not detect any ill-conditioning whatsoever.

```
A = condex(4, 1, 1e8);  format short e
% True         estimate     true/estimate
[cond(A,1)  1/rcond(A)  cond(A,1)*rcond(A)]

ans =
```

```
  8.0000e+016   5.6000e+008   1.4286e+008


A = condex(3, 2, 1e8);
[cond(A,1)   1/rcond(A)   cond(A,1)*rcond(A)]

ans =

  6.0000e+008   7.5000e+000   8.0000e+007
```

The QR decomposition with column pivoting of $A \in \mathbf{C}^{m\times n}$ $(m \geq n)$ is a decomposition $A\Pi = Q \begin{bmatrix} R \\ 0 \end{bmatrix}$ where Π is a permutation matrix chosen according to a certain pivoting strategy, Q is orthogonal, and R is upper triangular [14, Section 5.4.1]. This decomposition is often used to estimate the rank of A; in particular, $|r_{nn}|$ provides an upper bound for the smallest singular value $\sigma_{\min}(A)$ of A that is usually at most a factor of 10 too big. Kahan [30] designed a matrix for which $|r_{nn}|$ can be approximately 2^{n-1} times bigger than $\sigma_{\min}(A)$, and which thus shows the fallibility of the QR decomposition with column pivoting for revealing rank. The toolbox function `kahan` generates Kahan's matrix:

```
n = 25;
A = kahan(n, 0.6);
[Q, R, Pi] = qr(A);
norm(Pi-eye(n),1), R(n,n)/min(svd(A))

ans =

     0

ans =

  1.0638e+006
```

Kahan's matrix $A(\theta)$ is upper triangular and is designed so that Π is the identity in the QR decomposition with column pivoting. In practice, rounding errors can cause Π to differ from the identity for the Kahan matrix, thus nullifying the example (the test on $\Pi - I$ in the above example confirms that Π is indeed the identity here). The toolbox routine adds a small perturbation to the diagonal elements of $A(\theta)$ so that $\Pi = I$ for a range of choices of n and θ. If we set the perturbation in the above example to zero, this is what happens:

```
A = kahan(n, 0.6, 0);  % Third parameter is the diagonal perturbation.
[Q, R, Pi] = qr(A);
norm(Pi-eye(n),1), R(n,n)/min(svd(A))

ans =

     2

ans =

    1.1953
```

The toolbox contains two matrices, one a Toeplitz matrix and the other a Hankel matrix, whose eigenvalues or singular values are related to π:

```
A = parter(10); format long
e = svd(A); [e  e-pi]

ans =

   3.14159265358968  -0.00000000000011
   3.14159265356666  -0.00000000002313
   3.14159265139317  -0.00000000219663
   3.14159252749873  -0.00000012609106
   3.14158778157056  -0.00000487201924
   3.14145930586226  -0.00013334772753
   3.13895248060091  -0.00264017298888
   3.10410768313639  -0.03748497045341
   2.78691548240413  -0.35467717118566
   1.30096907002970  -1.84062358356009


A = dingdong(10);
e = eig(A); [e  abs(e)-pi/2]

ans =

  -1.57079632679484  -0.00000000000006
  -1.57079632569658  -0.00000000109831
  -1.57079389078528  -0.00000243600962
   1.57079632678333  -0.00000000001157
   1.57079626374937  -0.00000006304553
   1.57072965293113  -0.00006667386377
  -1.56947624030045  -0.00132008649444
   1.55205384156819  -0.01874248522670
  -1.39345774120206  -0.17733858559283
   0.65048453501485  -0.92031179178005
```

Finally, here is an example of the use of the function `matrix` to access the test matrices sequentially, by number. The following piece of code steps through all the square matrices of arbitrary dimension, setting A to each 10×10 matrix in turn (any matrix parameters are at their default values). It evaluates the 2-norm condition number and the ratio $\mu(A) = \max_i |\lambda_i(A)| / \min_i |\lambda_i(A)|$ of the largest to smallest eigenvalue in absolute value.

```
c = []; e = [];
j = 1;
for i=1:matrix(0)
    A = full(matrix(i, 10));
    if norm(skewpart(A),1)  % If not Hermitian...
       c1 = cond(A);
       eg = eig(A);
       e1 = max(abs(eg)) / min(abs(eg));
       % Filter out extremely ill-conditioned matrices.
```

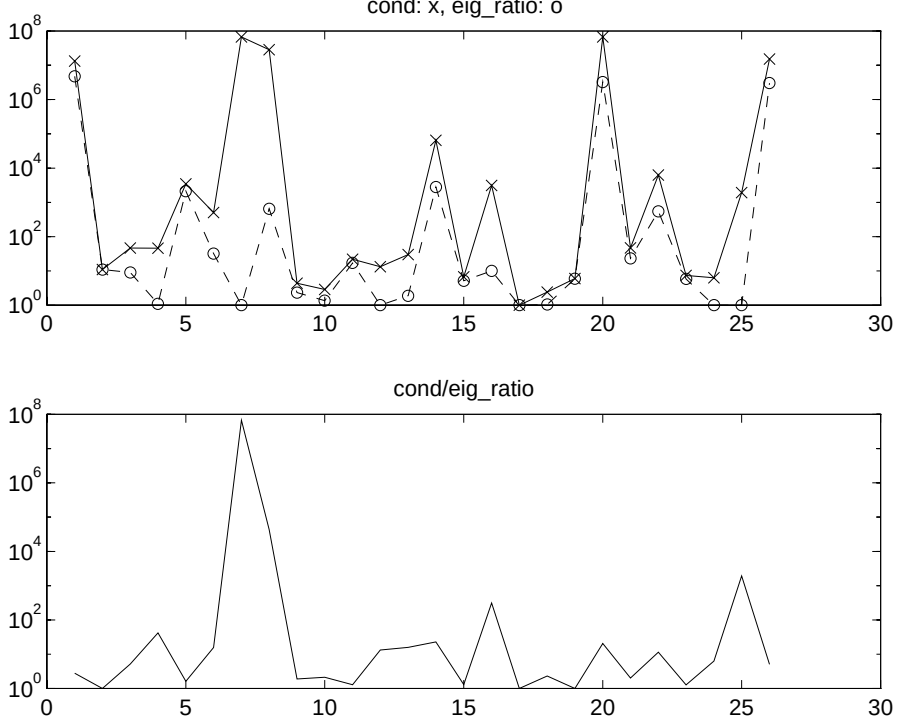


Figure 9.5: Comparison of condition number with extremal eigenvalue ratio.

```
        if c1 <= 1e10, c(j) = c1; e(j) = e1; j = j + 1; end
    end
end
```

As is well known, $\kappa_2(A)$ can be arbitrarily larger than $\mu(A)$. The plots in Figure 9.5, produced from the vectors `c` and `e` from the above code, confirm that $\kappa_2(A)/\mu(A)$ can be large.

10 M-File Leading Comment Lines

The demonstration file `tmtdemo` is not listed here.

```
function [x, fmax, nf] = adsmax(f, x, stopit, savit, P)
%ADSMAX  Alternating directions direct search method.
%        [x, fmax, nf] = ADSMAX(f, x0, STOPIT, SAVIT, P) attempts to
%        maximize the function specified by the string f, using the starting
%        vector x0.  The alternating directions direct search method is used.
%        Output arguments:
%               x    = vector yielding largest function value found,
%               fmax = function value at x,
%               nf   = number of function evaluations.
%        The iteration is terminated when either
%               - the relative increase in function value between successive
%                 iterations is <= STOPIT(1) (default 1e-3),
%               - STOPIT(2) function evaluations have been performed
%                 (default inf, i.e., no limit), or
%               - a function value equals or exceeds STOPIT(3)
%                 (default inf, i.e., no test on function values).
%        Progress of the iteration is not shown if STOPIT(5) = 0 (default 1).
%        If a non-empty fourth parameter string SAVIT is present, then
%        'SAVE SAVIT x fmax nf' is executed after each inner iteration.
%        By default, the search directions are the co-ordinate directions.
%        The columns of a fifth parameter matrix P specify alternative search
%        directions (P = EYE is the default).
%        NB: x0 can be a matrix.  In the output argument, in SAVIT saves,
%            and in function calls, x has the same shape as x0.

%     Reference:
%     N.J. Higham, Optimization by direct search in matrix computations,
%     SIAM J. Matrix Anal. Appl, 14(2): 317-333, April 1993.
```

```
function C = augment(A, alpha)
%AUGMENT  Augmented system matrix.
%         AUGMENT(A, ALPHA) is the square matrix
%         [ALPHA*EYE(m) A; A' ZEROS(n)] of dimension m+n, where A is m-by-n.
%         It is the symmetric and indefinite coefficient matrix of the
%         augmented system associated with a least squares problem
%         minimize NORM(A*x-b).  ALPHA defaults to 1.
%         Special case: if A is a scalar, n say, then AUGMENT(A) is the
%                       same as AUGMENT(RANDN(p,q)) where n = p+q and
%                       p = ROUND(n/2), that is, a random augmented matrix
%                       of dimension n is produced.
%         The eigenvalues of AUGMENT(A) are given in terms of the singular
%         values s(i) of A (where m>n) by
%                  1/2 +/- SQRT( s(i)^2 + 1/4 ),  i=1:n  (2n eigenvalues),
%                  1,  (m-n eigenvalues).
%         If m < n then the first expression provides 2m eigenvalues and the
```

```
%         remaining n-m eigenvalues are zero.
%
%         See also SPAUGMENT.

%         Reference:
%         G.H. Golub and C.F. Van Loan, Matrix Computations, Second
%         Edition, Johns Hopkins University Press, Baltimore, Maryland,
%         1989, sec. 5.6.4.
```

```
function A = bandred(A, kl, ku)
%BANDRED  Band reduction by two-sided unitary transformations.
%         B = BANDRED(A, KL, KU) is a matrix unitarily equivalent to A
%         with lower bandwidth KL and upper bandwidth KU
%         (i.e. B(i,j) = 0 if i > j+KL or j > i+KU).
%         The reduction is performed using Householder transformations.
%         If KU is omitted it defaults to KL.

%         Called by RANDSVD.
%         This is a 'standard' reduction.  Cf. reduction to bidiagonal form
%         prior to computing the SVD.  This code is a little wasteful in that
%         it computes certain elements which are immediately set to zero!
%
%         Reference:
%         G.H. Golub and C.F. Van Loan, Matrix Computations, second edition,
%         Johns Hopkins University Press, Baltimore, Maryland, 1989.
%         Section 5.4.3.
```

```
function C = cauchy(x, y)
%CAUCHY  Cauchy matrix.
%        C = CAUCHY(X, Y), where X, Y are N-vectors, is the N-by-N matrix
%        with C(i,j) = 1/(X(i)+Y(j)).   By default, Y = X.
%        Special case: if X is a scalar CAUCHY(X) is the same as CAUCHY(1:X).
%        Explicit formulas are known for DET(C) (which is nonzero if X and Y
%        both have distinct elements) and the elements of INV(C).
%        C is totally positive if 0 < X(1) < ... < X(N) and
%        0 < Y(1) < ... < Y(N).

%        References:
%        N.J. Higham, Accuracy and Stability of Numerical Algorithms,
%           Society for Industrial and Applied Mathematics, Philadelphia, PA,
%           USA, 1996; sec. 26.1.
%        D.E. Knuth, The Art of Computer Programming, Volume 1,
%           Fundamental Algorithms, second edition, Addison-Wesley, Reading,
%           Massachusetts, 1973, p. 36.
%        E.E. Tyrtyshnikov, Cauchy-Toeplitz matrices and some applications,
%           Linear Algebra and Appl., 149 (1991), pp. 1-18.
%        O. Taussky and M. Marcus, Eigenvalues of finite matrices, in
%           Survey of Numerical Analysis, J. Todd, ed., McGraw-Hill, New York,
%           pp. 279-313, 1962. (States the totally positive property on p. 295.)
```

```
function [Q, R] = cgs(A)
%CGS     Classical Gram-Schmidt QR factorization.
%        [Q, R] = cgs(A) uses the classical Gram-Schmidt method to compute the
%        factorization A = Q*R for m-by-n A of full rank,
%        where Q is m-by-n with orthonormal columns and R is n-by-n.
```

```
function C = chebspec(n, k)
%CHEBSPEC  Chebyshev spectral differentiation matrix.
%          C = CHEBSPEC(N, K) is a Chebyshev spectral differentiation
%          matrix of order N.  K = 0 (the default) or 1.
%          For K = 0 (‘no boundary conditions’), C is nilpotent, with
%              C^N = 0 and it has the null vector ONES(N,1).
%              C is similar to a Jordan block of size N with eigenvalue zero.
%          For K = 1, C is nonsingular and well-conditioned, and its eigenvalues
%              have negative real parts.
%          For both K, the computed eigenvector matrix X from EIG is
%              ill-conditioned (MESH(REAL(X)) is interesting).

%          References:
%          C. Canuto, M.Y. Hussaini, A. Quarteroni and T.A. Zang, Spectral
%             Methods in Fluid Dynamics, Springer-Verlag, Berlin, 1988; p. 69.
%          L.N. Trefethen and M.R. Trummer, An instability phenomenon in
%             spectral methods, SIAM J. Numer. Anal., 24 (1987), pp. 1008-1023.
%          D. Funaro, Computing the inverse of the Chebyshev collocation
%             derivative, SIAM J. Sci. Stat. Comput., 9 (1988), pp. 1050-1057.
```

```
function C = chebvand(m,p)
%CHEBVAND  Vandermonde-like matrix for the Chebyshev polynomials.
%          C = CHEBVAND(P), where P is a vector, produces the (primal)
%          Chebyshev Vandermonde matrix based on the points P,
%          i.e., C(i,j) = T_{i-1}(P(j)), where T_{i-1} is the Chebyshev
%          polynomial of degree i-1.
%          CHEBVAND(M,P) is a rectangular version of CHEBVAND(P) with M rows.
%          Special case: If P is a scalar then P equally spaced points on
%                        [0,1] are used.

%           Reference:
%           N.J. Higham, Stability analysis of algorithms for solving confluent
%           Vandermonde-like systems, SIAM J. Matrix Anal. Appl., 11 (1990),
%           pp. 23-41.
```

```
function [R, P, I] = cholp(A, pivot)
%CHOLP  Cholesky factorization with pivoting of a pos. semidefinite matrix.
%       [R, P] = CHOLP(A) returns R and a permutation matrix P such that
%       R'*R = P'*A*P.  Only the upper triangular part of A is used.
%       [R, P, I] = CHOLP(A) returns in addition the index I of the last
%       positive diagonal element of R.  The first I rows of R contain
%       the Cholesky factor of A.
```

```
%       [R, I] = CHOLP(A, 0) forces P = EYE(SIZE(A)), and therefore produces
%       the same output as R = CHOL(A) when A is positive definite (to
%       within roundoff).

%       This routine is based on the LINPACK routine CCHDC.  It works
%       for both real and complex matrices.
%
%       Reference:
%       G.H. Golub and C.F. Van Loan, Matrix Computations, Second
%       Edition, Johns Hopkins University Press, Baltimore, Maryland,
%       1989, sec. 4.2.9.
```

```
function c = chop(x, t)
%CHOP    Round matrix elements.
%        CHOP(X, t) is the matrix obtained by rounding the elements of X
%        to t significant binary places.
%        Default is t = 24, corresponding to IEEE single precision.
```

```
function A = chow(n, alpha, delta)
%CHOW    Chow matrix - a singular Toeplitz lower Hessenberg matrix.
%        A = CHOW(N, ALPHA, DELTA) is a Toeplitz lower Hessenberg matrix
%        A = H(ALPHA) + DELTA*EYE, where H(i,j) = ALPHA^(i-j+1).
%        H(ALPHA) has p = FLOOR(N/2) zero eigenvalues, the rest being
%        4*ALPHA*COS( k*PI/(N+2) )^2, k=1:N-p.
%        Defaults: ALPHA = 1, DELTA = 0.

%        References:
%        T.S. Chow, A class of Hessenberg matrices with known
%           eigenvalues and inverses, SIAM Review, 11 (1969), pp. 391-395.
%        G. Fairweather, On the eigenvalues and eigenvectors of a class of
%           Hessenberg matrices, SIAM Review, 13 (1971), pp. 220-221.
```

```
function C = circul(v)
%CIRCUL  Circulant matrix.
%        C = CIRCUL(V) is the circulant matrix whose first row is V.
%        (A circulant matrix has the property that each row is obtained
%        from the previous one by cyclically permuting the entries one step
%        forward; it is a special Toeplitz matrix in which the diagonals
%        'wrap round'.)
%        Special case: if V is a scalar then C = CIRCUL(1:V).
%        The eigensystem of C (N-by-N) is known explicitly.   If t is an Nth
%        root of unity, then the inner product of V with W = [1 t t^2 ... t^N]
%        is an eigenvalue of C, and W(N:-1:1) is an eigenvector of C.

%        Reference:
%        P.J. Davis, Circulant Matrices, John Wiley, 1977.
```

```
function A = clement(n, k)
%CLEMENT   Clement matrix - tridiagonal with zero diagonal entries.
%          CLEMENT(N, K) is a tridiagonal matrix with zero diagonal entries
%          and known eigenvalues.  It is singular if N is odd.  About 64
%          percent of the entries of the inverse are zero.  The eigenvalues
%          are plus and minus the numbers N-1, N-3, N-5, ..., (1 or 0).
%          For K = 0 (the default) the matrix is unsymmetric, while for
%          K = 1 it is symmetric.
%          CLEMENT(N, 1) is diagonally similar to CLEMENT(N).

%          Similar properties hold for TRIDIAG(X,Y,Z) where Y = ZEROS(N,1).
%          The eigenvalues still come in plus/minus pairs but they are not
%          known explicitly.
%
%          References:
%          P.A. Clement, A class of triple-diagonal matrices for test
%             purposes, SIAM Review, 1 (1959), pp. 50-52.
%          A. Edelman and E. Kostlan, The road from Kac's matrix to Kac's
%             random polynomials. In John~G. Lewis, editor, Proceedings of
%             the Fifth SIAM Conference on Applied Linear Algebra Society
%             for Industrial and Applied Mathematics, Philadelphia, 1994,
%             pp. 503-507.
%          O. Taussky and J. Todd, Another look at a matrix of Mark Kac,
%             Linear Algebra and Appl., 150 (1991), pp. 341-360.
```

```
function [U, R, V] = cod(A, tol)
%COD    Complete orthogonal decomposition.
%       [U, R, V] = COD(A, TOL) computes a decomposition A = U*T*V,
%       where U and V are unitary, T = [R 0; 0 0] has the same dimensions as
%       A, and R is upper triangular and nonsingular of dimension rank(A).
%       Rank decisions are made using TOL, which defaults to approximately
%       MAX(SIZE(A))*NORM(A)*EPS.
%       By itself, COD(A, TOL) returns R.

%       Reference:
%       G.H. Golub and C.F. Van Loan, Matrix Computations, Second
%       Edition, Johns Hopkins University Press, Baltimore, Maryland,
%       1989, sec. 5.4.2.
```

```
function C = comp(A, k)
%COMP    Comparison matrices.
%        COMP(A) is DIAG(B) - TRIL(B,-1) - TRIU(B,1), where B = ABS(A).
%        COMP(A, 1) is A with each diagonal element replaced by its
%        absolute value, and each off-diagonal element replaced by minus
%        the absolute value of the largest element in absolute value in
%        its row.  However, if A is triangular COMP(A, 1) is too.
%        COMP(A, 0) is the same as COMP(A).
%        COMP(A) is often denoted by M(A) in the literature.
```

```
%        Reference (e.g.):
%        N.J. Higham, A survey of condition number estimation for
%        triangular matrices, SIAM Review, 29 (1987), pp. 575-596.
```

```
function A = compan(p)
%COMPAN  Companion matrix.
%        COMPAN(P) is a companion matrix.  There are three cases.
%        If P is a scalar then COMPAN(P) is the P-by-P matrix COMPAN(1:P+1).
%        If P is an (n+1)-vector, COMPAN(P) is the n-by-n companion matrix
%           whose first row is -P(2:n+1)/P(1).
%        If P is a square matrix, COMPAN(P) is the companion matrix
%           of the characteristic polynomial of P, computed as
%           COMPAN(POLY(P)).

%        References:
%        J.H. Wilkinson, The Algebraic Eigenvalue Problem,
%           Oxford University Press, 1965, p. 12.
%        G.H. Golub and C.F. Van Loan, Matrix Computations, second edition,
%           Johns Hopkins University Press, Baltimore, Maryland, 1989,
%           sec 7.4.6.
%        C. Kenney and A.J. Laub, Controllability and stability radii for
%          companion form systems, Math. Control Signals Systems, 1 (1988),
%          pp. 239-256. (Gives explicit formulas for the singular values of
%          COMPAN(P).)
```

```
function y = cond(A, p)
%COND   Matrix condition number in 1, 2, Frobenius, or infinity norm.
%       For p = 1, 2, 'fro', inf,  COND(A,p) = NORM(A,p) * NORM(INV(A),p).
%       If p is omitted then p = 2 is used.
%       A may be a rectangular matrix if p = 2; in this case COND(A)
%       is the ratio of the largest singular value of A to the smallest
%       (and hence is infinite if A is rank deficient).
```

```
function A = condex(n, k, theta)
%CONDEX   'Counterexamples' to matrix condition number estimators.
%         CONDEX(N, K, THETA) is a 'counterexample' matrix to a condition
%         estimator.  It has order N and scalar parameter THETA (default 100).
%         If N is not equal to the 'natural' size of the matrix then
%         the matrix is padded out with an identity matrix to order N.
%         The matrix, its natural size, and the estimator to which it applies
%         are specified by K (default K = 4) as follows:
%             K = 1:   4-by-4,     LINPACK (RCOND)
%             K = 2:   3-by-3,     LINPACK (RCOND)
%             K = 3:   arbitrary,  LINPACK (RCOND) (independent of THETA)
%             K = 4:   N >= 4,     SONEST (Higham 1988)
%         (Note that in practice the K = 4 matrix is not usually a
%          counterexample because of the rounding errors in forming it.)
```

```
%         References:
%         A.K. Cline and R.K. Rew, A set of counter-examples to three
%            condition number estimators, SIAM J. Sci. Stat. Comput.,
%            4 (1983), pp. 602-611.
%         N.J. Higham, FORTRAN codes for estimating the one-norm of a real or
%            complex matrix, with applications to condition estimation
%            (Algorithm 674), ACM Trans. Math. Soft., 14 (1988), pp. 381-396.
```

```
function x = cpltaxes(z)
%CPLTAXES   Determine suitable AXIS for plot of complex vector.
%           X = CPLTAXES(Z), where Z is a complex vector,
%           determines a 4-vector X such that AXIS(X) sets axes for a plot
%           of Z that has axes of equal length and leaves a reasonable amount
%           of space around the edge of the plot.

%           Called by FV, GERSH, PS and PSCONT.
```

```
function A = cycol(n, k)
%CYCOL   Matrix whose columns repeat cyclically.
%        A = CYCOL([M N], K) is an M-by-N matrix of the form A = B(1:M,1:N)
%        where B = [C C C...] and C = RANDN(M, K).  Thus A's columns repeat
%        cyclically, and A has rank at most K.   K need not divide N.
%        K defaults to ROUND(N/4).
%        CYCOL(N, K), where N is a scalar, is the same as CYCOL([N N], K).
%
%        This type of matrix can lead to underflow problems for Gaussian
%        elimination: see NA Digest Volume 89, Issue 3 (January 22, 1989).
```

```
function [L, D, P, rho] = diagpiv(A)
%DIAGPIV     Diagonal pivoting factorization with partial pivoting.
%            Given a symmetric matrix A,
%            [L, D, P, rho] = diagpiv(A) computes a permutation P,
%            a unit lower triangular L, and a block diagonal D
%            with 1x1 and 2x2 diagonal blocks, such that
%            P*A*P' = L*D*L'.
%            The Bunch-Kaufman partial pivoting strategy is used.
%            Rho is the growth factor.

%            Reference:
%            J.R. Bunch and L. Kaufman, Some stable methods for calculating
%            inertia and solving symmetric linear systems, Math. Comp.,
%            31(137):163-179, 1977.
```

```
function A = dingdong(n)
%DINGDONG  Dingdong matrix - a symmetric Hankel matrix.
%          A = DINGDONG(N) is the symmetric N-by-N Hankel matrix with
%                          A(i,j) = 0.5/(N-i-j+1.5).
```

```
%          The eigenvalues of A cluster around PI/2 and -PI/2.

%          Invented by F.N. Ris.
%
%          Reference:
%          J.C. Nash, Compact Numerical Methods for Computers: Linear
%          Algebra and Function Minimisation, second edition, Adam Hilger,
%          Bristol, 1990 (Appendix 1).
```

```
function [c, d, e] = dorr(n, theta)
%DORR  Dorr matrix - diagonally dominant, ill conditioned, tridiagonal.
%      [C, D, E] = DORR(N, THETA) returns the vectors defining a row diagonally
%      dominant, tridiagonal M-matrix that is ill conditioned for small
%      values of the parameter THETA >= 0.
%      If only one output parameter is supplied then
%      C = FULL(TRIDIAG(C,D,E)), i.e., the matrix iself is returned.
%      The columns of INV(C) vary greatly in norm.  THETA defaults to 0.01.
%      The amount of diagonal dominance is given by (ignoring rounding errors):
%            COMP(C)*ONES(N,1) = THETA*(N+1)^2 * [1 0 0 ... 0 1]'.

%      Reference:
%      F.W. Dorr, An example of ill-conditioning in the numerical
%      solution of singular perturbation problems, Math. Comp., 25 (1971),
%      pp. 271-283.
```

```
function A = dramadah(n, k)
%DRAMADAH  A (0,1) matrix whose inverse has large integer entries.
%          An anti-Hadamard matrix A is a matrix with elements 0 or 1 for
%          which MU(A) := NORM(INV(A),'FRO') is maximal.
%          A = DRAMADAH(N, K) is an N-by-N (0,1) matrix for which MU(A) is
%          relatively large, although not necessarily maximal.
%          Available types (the default is K = 1):
%          K = 1: A is Toeplitz, with ABS(DET(A)) = 1, and MU(A) > c(1.75)^N,
%                 where c is a constant.
%          K = 2: A is upper triangular and Toeplitz.
%          The inverses of both types have integer entries.
%
%          Another interesting (0,1) matrix:
%          K = 3: A has maximal determinant among (0,1) lower Hessenberg
%          matrices: det(A) = the n'th Fibonacci number.  A is Toeplitz.
%          The eigenvalues have an interesting distribution in the complex
%          plane.

%          References:
%          R.L. Graham and N.J.A. Sloane, Anti-Hadamard matrices,
%             Linear Algebra and Appl., 62 (1984), pp. 113-137.
%          L. Ching, The maximum determinant of an nxn lower Hessenberg
%             (0,1) matrix, Linear Algebra and Appl., 183 (1993), pp. 147-153.
```

```
function y = dual(x, p)
%DUAL    Dual vector with respect to Holder p-norm.
%        Y = DUAL(X, p), where 1 <= p <= inf, is a vector of unit q-norm
%        that is dual to X with respect to the p-norm, that is,
%        norm(Y, q) = 1 where 1/p + 1/q = 1 and there is
%        equality in the Holder inequality: X'*Y = norm(X, p)*norm(Y, q).
%        Special case: DUAL(X), where X >= 1 is a scalar, returns Y such
%                      that 1/X + 1/Y = 1.

%        Called by PNORM.
```

```
function [X, D, s] = eigsens(A)
%EIGSENS   Eigenvalue condition numbers.
%          EIGSENS(A) is a vector of condition numbers for the eigenvalues
%          of A (reciprocals of the Wilkinson s(lambda) numbers).
%          These condition numbers are the reciprocals of the cosines of the
%          angles between the left and right eigenvectors.
%          [V, D, s] = EIGSENS(A) is equivalent to
%                      [V, D] = EIG(A); s = EIGSENS(A);

%          Reference:
%          G.H. Golub and C.F. Van Loan, Matrix Computations, Second
%          Edition, Johns Hopkins University Press, Baltimore, Maryland,
%          1989, sec. 7.2.2.
```

```
function f = fdemo(A)
%FDEMO   Demonstration function for direct search maximizers.
%        FDEMO(A) is the reciprocal of the underestimation ratio for RCOND
%        applied to the square matrix A.
%        Demonstration function for ADSMAX, MDSMAX and NMSMAX.
```

```
function A = fiedler(c)
%FIEDLER  Fiedler matrix - symmetric.
%         A = FIEDLER(C), where C is an n-vector, is the n-by-n symmetric
%         matrix with elements ABS(C(i)-C(j)).
%         Special case: if C is a scalar, then A = FIEDLER(1:C)
%                       (i.e. A(i,j) = ABS(i-j)).
%         Properties:
%           FIEDLER(N) has a dominant positive eigenvalue and all the other
%                      eigenvalues are negative (Szego, 1936).
%           Explicit formulas for INV(A) and DET(A) are given by Todd (1977)
%           and attributed to Fiedler.  These indicate that INV(A) is
%           tridiagonal except for nonzero (1,n) and (n,1) elements.
%           [I think these formulas are valid only if the elements of
%           C are in increasing or decreasing order---NJH.]

%           References:
%           G. Szego, Solution to problem 3705, Amer. Math. Monthly,
```

```
%              43 (1936), pp. 246-259.
%           J. Todd, Basic Numerical Mathematics, Vol. 2: Numerical Algebra,
%              Birkhauser, Basel, and Academic Press, New York, 1977, p. 159.
```

```
function A = forsythe(n, alpha, lambda)
%FORSYTHE  Forsythe matrix - a perturbed Jordan block.
%          FORSYTHE(N, ALPHA, LAMBDA) is the N-by-N matrix equal to
%          JORDBLOC(N, LAMBDA) except it has an ALPHA in the (N,1) position.
%          It has the characteristic polynomial
%                  DET(A-t*EYE) = (LAMBDA-t)^N - (-1)^N ALPHA.
%          ALPHA defaults to SQRT(EPS) and LAMBDA to 0.
```

```
function F = frank(n, k)
%FRANK   Frank matrix---ill conditioned eigenvalues.
%        F = FRANK(N, K) is the Frank matrix of order N.  It is upper
%        Hessenberg with determinant 1.  K = 0 is the default; if K = 1 the
%        elements are reflected about the anti-diagonal (1,N)--(N,1).
%        F has all positive eigenvalues and they occur in reciprocal pairs
%        (so that 1 is an eigenvalue if N is odd).
%        The eigenvalues of F may be obtained in terms of the zeros of the
%        Hermite polynomials.
%        The FLOOR(N/2) smallest eigenvalues of F are ill conditioned,
%        the more so for bigger N.

%        DET(FRANK(N)') comes out far from 1 for large N---see Frank (1958)
%        and Wilkinson (1960) for discussions.
%
%        This version incorporates improvements suggested by W. Kahan.
%
%        References:
%        W.L. Frank, Computing eigenvalues of complex matrices by determinant
%           evaluation and by methods of Danilewski and Wielandt, J. Soc.
%           Indust. Appl. Math., 6 (1958), pp. 378-392 (see pp. 385, 388).
%        G.H. Golub and J.H. Wilkinson, Ill-conditioned eigensystems and the
%           computation of the Jordan canonical form, SIAM Review, 18 (1976),
%             pp. 578-619 (Section 13).
%        H. Rutishauser, On test matrices, Programmation en Mathematiques
%           Numeriques, Editions Centre Nat. Recherche Sci., Paris, 165,
%           1966, pp. 349-365.  Section 9.
%        J.H. Wilkinson, Error analysis of floating-point computation,
%           Numer. Math., 2 (1960), pp. 319-340 (Section 8).
%        J.H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University
%           Press, 1965 (pp. 92-93).
%        The next two references give details of the eigensystem, as does
%        Rutishauser (see above).
%        P.J. Eberlein, A note on the matrices denoted by B_n, SIAM J. Appl.
%           Math., 20 (1971), pp. 87-92.
%        J.M. Varah, A generalization of the Frank matrix, SIAM J. Sci. Stat.
%           Comput., 7 (1986), pp. 835-839.
```

```
function [f, e] = fv(B, nk, thmax, noplot)
%FV     Field of values (or numerical range).
%       FV(A, NK, THMAX) evaluates and plots the field of values of the
%       NK largest leading principal submatrices of A, using THMAX
%       equally spaced angles in the complex plane.
%       The defaults are NK = 1 and THMAX = 16.
%       (For a 'publication quality' picture, set THMAX higher, say 32.)
%       The eigenvalues of A are displayed as 'x'.
%       Alternative usage: [F, E] = FV(A, NK, THMAX, 1) suppresses the
%       plot and returns the field of values plot data in F, with A's
%       eigenvalues in E.   Note that NORM(F,INF) approximates the
%       numerical radius,
%                 max {abs(z): z is in the field of values of A}.

%       Theory:
%       Field of values FV(A) = set of all Rayleigh quotients. FV(A) is a
%       convex set containing the eigenvalues of A.  When A is normal FV(A) is
%       the convex hull of the eigenvalues of A (but not vice versa).
%               z = x'Ax/(x'x),  z' = x'A'x/(x'x)
%               => REAL(z) = x'Hx/(x'x),   H = (A+A')/2
%       so      MIN(EIG(H)) <= REAL(z) <= MAX(EIG(H))
%       with equality for x = corresponding eigenvectors of H.  For these x,
%       RQ(A,x) is on the boundary of FV(A).
%
%       Based on an original routine by A. Ruhe.
%
%       References:
%       R.A. Horn and C.R. Johnson, Topics in Matrix Analysis, Cambridge
%            University Press, 1991, Section 1.5.
%       A.S. Householder, The Theory of Matrices in Numerical Analysis,
%            Blaisdell, New York, 1964, Section 3.3.
%       C.R. Johnson, Numerical determination of the field of values of a
%            general complex matrix, SIAM J. Numer. Anal., 15 (1978),
%            pp. 595-602.
```

```
function [A, e] = gallery(n)
%GALLERY    Famous, and not so famous, test matrices.
%       A = GALLERY(N) is an N-by-N matrix with some special property.
%       The following values of N are currently available:
%         N = 3 is badly conditioned.
%         N = 4 is the Wilson matrix.  Symmetric pos def, integer inverse.
%         N = 5 is an interesting eigenvalue problem: defective, nilpotent.
%         N = 8 is the Rosser matrix, a classic symmetric eigenvalue problem.
%               [A, e] = GALLERY(8) returns the exact eigenvalues in e.
%         N = 21 is Wilkinson's tridiagonal W21+, another eigenvalue problem.

%       Original version supplied with MATLAB.  Modified by N.J. Higham.
%
%       References:
```

```
%       J.R. Westlake, A Handbook of Numerical Matrix Inversion and Solution
%          of Linear Equations, John Wiley, New York, 1968.
%       J.H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University
%          Press, 1965.
```

```
function [L, U, rho] = ge(A)
%GE     Gaussian elimination without pivoting.
%       [L, U, RHO] = GE(A) computes the factorization A = L*U,
%       where L is unit lower triangular and U is upper triangular.
%       RHO is the growth factor.
%       By itself, GE(A) returns the final reduced matrix from the
%       elimination containing both L and U.
```

```
function A = gearm(n, i, j)
%GEARM   Gear matrix.
%        A = GEARM(N,I,J) is the N-by-N matrix with ones on the sub- and
%        super-diagonals, SIGN(I) in the (1,ABS(I)) position, SIGN(J)
%        in the (N,N+1-ABS(J)) position, and zeros everywhere else.
%        Defaults: I = N, j = -N.
%        All eigenvalues are of the form 2*COS(a) and the eigenvectors
%        are of the form [SIN(w+a), SIN(w+2a), ..., SIN(w+Na)].
%        The values of a and w are given in the reference below.
%        A can have double and triple eigenvalues and can be defective.
%        GEARM(N) is singular.

%        (GEAR is a Simulink function, hence GEARM for Gear matrix.)
%        Reference:
%        C.W. Gear, A simple set of test matrices for eigenvalue programs,
%        Math. Comp., 23 (1969), pp. 119-125.
```

```
function [L, U, P, Q, rho] = gecp(A)
%GECP   Gaussian elimination with complete pivoting.
%       [L, U, P, Q, RHO] = GECP(A) computes the factorization P*A*Q = L*U,
%       where L is unit lower triangular, U is upper triangular,
%       and P and Q are permutation matrices.  RHO is the growth factor.
%       By itself, GECP(A) returns the final reduced matrix from the
%       elimination containing both L and U.
```

```
function  [G, e] = gersh(A, noplot)
%GERSH    Gershgorin disks.
%         GERSH(A) draws the Gershgorin disks for the matrix A.
%         The eigenvalues are plotted as crosses 'x'.
%         Alternative usage: [G, E] = GERSH(A, 1) suppresses the plot
%         and returns the data in G, with A's eigenvalues in E.
%
%         Try GERSH(LESP(N)) and GERSH(SMOKE(N,1)).
```

```
function A = gfpp(T, c)
%GFPP   Matrix giving maximal growth factor for Gaussian elim. with pivoting.
%       GFPP(T) is a matrix of order N for which Gaussian elimination
%       with partial pivoting yields a growth factor 2^(N-1).
%       T is an arbitrary nonsingular upper triangular matrix of order N-1.
%       GFPP(T, C) sets all the multipliers to C  (0 <= C <= 1)
%       and gives growth factor (1+C)^(N-1).
%       GFPP(N, C) (a special case) is the same as GFPP(EYE(N-1), C) and
%       generates the well-known example of Wilkinson.

%       Reference:
%       N.J. Higham and D.J. Higham, Large growth factors in
%       Gaussian elimination with pivoting, SIAM J. Matrix Analysis and
%       Appl., 10 (1989), pp. 155-164.
```

```
function x = gj(A, b, piv)
%GJ        Gauss-Jordan elimination to solve Ax = b.
%          x = GJ(A, b, PIV) solves Ax = b by Gauss-Jordan elimination,
%          where A is a square, nonsingular matrix.
%          PIV determines the form of pivoting:
%              PIV = 0:           no pivoting,
%              PIV = 1 (default): partial pivoting.
```

```
function G = grcar(n, k)
%GRCAR     Grcar matrix - a Toeplitz matrix with sensitive eigenvalues.
%          GRCAR(N, K) is an N-by-N matrix with -1s on the
%          subdiagonal, 1s on the diagonal, and K superdiagonals of 1s.
%          The default is K = 3.  The eigenvalues of this matrix form an
%          interesting pattern in the complex plane (try PS(GRCAR(32))).

%          References:
%          J.F. Grcar, Operator coefficient methods for linear equations,
%               Report SAND89-8691, Sandia National Laboratories, Albuquerque,
%               New Mexico, 1989 (Appendix 2).
%          N.M. Nachtigal, L. Reichel and L.N. Trefethen, A hybrid GMRES
%               algorithm for nonsymmetric linear systems, SIAM J. Matrix Anal.
%               Appl., 13 (1992), pp. 796-825.
```

```
function H = hadamard(n)
%HADAMARD  Hadamard matrix.
%          HADAMARD(N) is a Hadamard matrix of order N, that is,
%          a matrix H with elements 1 or -1 such that H*H' = N*EYE(N).
%          An N-by-N Hadamard matrix with N>2 exists only if REM(N,4) = 0.
%          This function handles only the cases where N, N/12 or N/20
%          is a power of 2.

%          Reference:
%          S.W. Golomb and L.D. Baumert, The search for Hadamard matrices,
%          Amer. Math. Monthly, 70 (1963) pp. 12-17.
```

```
function A = hanowa(n, d)
%HANOWA  A matrix whose eigenvalues lie on a vertical line in the complex plane.
%        HANOWA(N, d) is the N-by-N block 2x2 matrix (thus N = 2M must be even)
%                        [d*EYE(M)   -DIAG(1:M)
%                         DIAG(1:M)   d*EYE(M)]
%        It has complex eigenvalues lambda(k) = d +/- k*i  (1 <= k <= M).
%        Parameter d defaults to -1.

%        Reference:
%        E. Hairer, S.P. Norsett and G. Wanner, Solving Ordinary
%        Differential Equations I: Nonstiff Problems, Springer-Verlag,
%        Berlin, 1987. (pp. 86-87)
```

```
function H = hilb(n)
%HILB   Hilbert matrix.
%       HILB(N) is the N-by-N matrix with elements 1/(i+j-1).
%       It is a famous example of a badly conditioned matrix.
%       COND(HILB(N)) grows like EXP(3.5*N).
%       See INVHILB (standard MATLAB routine) for the exact inverse, which
%       has integer entries.
%       HILB(N) is symmetric positive definite, totally positive, and a
%       Hankel matrix.

%       References:
%       M.-D. Choi, Tricks or treats with the Hilbert matrix, Amer. Math.
%           Monthly, 90 (1983), pp. 301-312.
%       N.J. Higham, Accuracy and Stability of Numerical Algorithms,
%           Society for Industrial and Applied Mathematics, Philadelphia, PA,
%           USA, 1996; sec. 26.1.
%       M. Newman and J. Todd, The evaluation of matrix inversion
%           programs, J. Soc. Indust. Appl. Math., 6 (1958), pp. 466-476.
%       D.E. Knuth, The Art of Computer Programming,
%           Volume 1, Fundamental Algorithms, second edition, Addison-Wesley,
%           Reading, Massachusetts, 1973, p. 37.
```

```
function [v, beta] = house(x)
%HOUSE   Householder matrix.
%        If [v, beta] = HOUSE(x) then H = EYE - beta*v*v' is a Householder
%        matrix such that Hx = -sign(x(1))*norm(x)*e_1.
%        NB: If x = 0 then v = 0, beta = 1 is returned.
%            x can be real or complex.
%            sign(x) := exp(i*arg(x)) ( = x./abs(x) when x ~= 0).

%        Theory: (textbook references Golub & Van Loan 1989, 38-43;
%                 Stewart 1973, 231-234, 262; Wilkinson 1965, 48-50).
%        Hx = y: (I - beta*v*v')x = -s*e_1.
%        Must have |s| = norm(x), v = x+s*e_1, and
%        x'y = x'Hx =(x'Hx)' real => arg(s) = arg(x(1)).
%        So take s = sign(x(1))*norm(x) (which avoids cancellation).
```

```
%        v'v = (x(1)+s)^2 + x(2)^2 + ... + x(n)^2
%            = 2*norm(x)*(norm(x) + |x(1)|).
%
%        References:
%        G.H. Golub and C.F. Van Loan, Matrix Computations, second edition,
%           Johns Hopkins University Press, Baltimore, Maryland, 1989.
%        G.W. Stewart, Introduction to Matrix Computations, Academic Press,
%           New York, 1973,
%        J.H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University
%           Press, 1965.
```

```
function A = invhess(x, y)
%INVHESS  Inverse of an upper Hessenberg matrix.
%         INVHESS(X, Y), where X is an N-vector and Y an N-1 vector,
%         is the matrix whose lower triangle agrees with that of
%         ONES(N,1)*X' and whose strict upper triangle agrees with
%         that of [1 Y]*ONES(1,N).
%         The matrix is nonsingular if X(1) ~= 0 and X(i+1) ~= Y(i)
%         for all i, and its inverse is an upper Hessenberg matrix.
%         If Y is omitted it defaults to -X(1:N-1).
%         Special case: if X is a scalar INVHESS(X) is the same as
%         INVHESS(1:X).

%         References:
%         F.N. Valvi and V.S. Geroyannis, Analytic inverses and
%             determinants for a class of matrices, IMA Journal of Numerical
%             Analysis, 7 (1987), pp. 123-128.
%         W.-L. Cao and W.J. Stewart, A note on inverses of Hessenberg-like
%             matrices, Linear Algebra and Appl., 76 (1986), pp. 233-240.
%         Y. Ikebe, On inverses of Hessenberg matrices, Linear Algebra and
%             Appl., 24 (1979), pp. 93-97.
%         P. Rozsa, On the inverse of band matrices, Integral Equations and
%             Operator Theory, 10 (1987), pp. 82-95.
```

```
function A = invol(n)
%INVOL   An involutory matrix.
%        A = INVOL(N) is an N-by-N involutory (A*A = EYE(N)) and
%        ill-conditioned matrix.
%        It is a diagonally scaled version of HILB(N).
%        NB: B = (EYE(N)-A)/2 and B = (EYE(N)+A)/2 are idempotent (B*B = B).

%        Reference:
%        A.S. Householder and J.A. Carpenter, The singular values
%        of involutory and of idempotent matrices, Numer. Math. 5 (1963),
%        pp. 234-237.
```

```
function [A, detA] = ipjfact(n, k)
%IPJFACT   A Hankel matrix with factorial elements.
```

```
%          A = IPJFACT(N, K) is the matrix with
%                    A(i,j) = (i+j)!    (K = 0, default)
%                    A(i,j) = 1/(i+j)!  (K = 1)
%          Both are Hankel matrices.
%          The determinant and inverse are known explicitly.
%          If a second output argument is present, d = DET(A) is returned:
%          [A, d] = IPJFACT(N, K);

%          Suggested by P. R. Graves-Morris.
%
%          Reference:
%          M.J.C. Gover, The explicit inverse of factorial Hankel matrices,
%          Dept. of Mathematics, University of Bradford, 1993.
```

```
function J = jordbloc(n, lambda)
%JORDBLOC  Jordan block.
%          JORDBLOC(N, LAMBDA) is the N-by-N Jordan block with eigenvalue
%          LAMBDA.  LAMBDA = 1 is the default.
```

```
function U = kahan(n, theta, pert)
%KAHAN  Kahan matrix - upper trapezoidal.
%       KAHAN(N, THETA) is an upper trapezoidal matrix
%       that has some interesting properties regarding estimation of
%       condition and rank.
%       The matrix is N-by-N unless N is a 2-vector, in which case it
%       is N(1)-by-N(2).
%       The parameter THETA defaults to 1.2.
%       The useful range of THETA is 0 < THETA < PI.
%
%       To ensure that the QR factorization with column pivoting does not
%       interchange columns in the presence of rounding errors, the diagonal
%       is perturbed by PERT*EPS*diag( [N:-1:1] ).
%       The default is PERT = 25, which ensures no interchanges for KAHAN(N)
%       up to at least N = 90 in IEEE arithmetic.
%       KAHAN(N, THETA, PERT) uses the given value of PERT.

%       The inverse of KAHAN(N, THETA) is known explicitly: see
%       Higham (1987, p. 588), for example.
%       The diagonal perturbation was suggested by Christian Bischof.
%
%       References:
%       W. Kahan, Numerical linear algebra, Canadian Math. Bulletin,
%          9 (1966), pp. 757-801.
%       N.J. Higham, A survey of condition number estimation for
%          triangular matrices, SIAM Review, 29 (1987), pp. 575-596.
```

```
function A = kms(n, rho)
%KMS   Kac-Murdock-Szego Toeplitz matrix.
```

```
%       A = KMS(N, RHO) is the N-by-N Kac-Murdock-Szego Toeplitz matrix with
%       A(i,j) = RHO^(ABS((i-j))) (for real RHO).
%       If RHO is complex, then the same formula holds except that elements
%       below the diagonal are conjugated.
%       RHO defaults to 0.5.
%       Properties:
%          A has an LDL' factorization with
%                   L = INV(TRIW(N,-RHO,1)'),
%                   D(i,i) = (1-ABS(RHO)^2)*EYE(N) except D(1,1) = 1.
%          A is positive definite if and only if 0 < ABS(RHO) < 1.
%          INV(A) is tridiagonal.

%        Reference:
%        W.F. Trench, Numerical solution of the eigenvalue problem
%        for Hermitian Toeplitz matrices, SIAM J. Matrix Analysis and Appl.,
%        10 (1989), pp. 135-146 (and see the references therein).
```

```
function B = krylov(A, x, j)
%KRYLOV    Krylov matrix.
%          KRYLOV(A, x, j) is the Krylov matrix
%               [x, Ax, A^2x, ..., A^(j-1)x],
%          where A is an n-by-n matrix and x is an n-vector.
%          Defaults: x = ONES(n,1), j = n.
%          KRYLOV(n) is the same as KRYLOV(RANDN(n)).

%          Reference:
%          G.H. Golub and C.F. Van Loan, Matrix Computations, second edition,
%          Johns Hopkins University Press, Baltimore, Maryland, 1989, p. 369.
```

```
function A = lauchli(n, mu)
%LAUCHLI   Lauchli matrix - rectangular.
%          LAUCHLI(N, MU) is the (N+1)-by-N matrix [ONES(1,N); MU*EYE(N))].
%          It is a well-known example in least squares and other problems
%          that indicates the dangers of forming A'*A.
%          MU defaults to SQRT(EPS).

%          Reference:
%          P. Lauchli, Jordan-Elimination und Ausgleichung nach
%          kleinsten Quadraten, Numer. Math, 3 (1961), pp. 226-240.
```

```
function A = lehmer(n)
%LEHMER  Lehmer matrix - symmetric positive definite.
%        A = LEHMER(N) is the symmetric positive definite N-by-N matrix with
%                          A(i,j) = i/j for j >= i.
%        A is totally nonnegative.  INV(A) is tridiagonal, and explicit
%        formulas are known for its entries.
%        N <= COND(A) <= 4*N*N.
```

```
%        References:
%        M. Newman and J. Todd, The evaluation of matrix inversion
%           programs, J. Soc. Indust. Appl. Math., 6 (1958), pp. 466-476.
%        Solutions to problem E710 (proposed by D.H. Lehmer): The inverse
%           of a matrix, Amer. Math. Monthly, 53 (1946), pp. 534-535.
%        J. Todd, Basic Numerical Mathematics, Vol. 2: Numerical Algebra,
%           Birkhauser, Basel, and Academic Press, New York, 1977, p. 154.
```

```
function T = lesp(n)
%LESP   A tridiagonal matrix with real, sensitive eigenvalues.
%       LESP(N) is an N-by-N matrix whose eigenvalues are real and smoothly
%       distributed in the interval approximately [-2*N-3.5, -4.5].
%       The sensitivities of the eigenvalues increase exponentially as
%       the eigenvalues grow more negative.
%       The matrix is similar to the symmetric tridiagonal matrix with
%       the same diagonal entries and with off-diagonal entries 1,
%       via a similarity transformation with D = diag(1!,2!,...,N!).

%       References:
%       H.W.J. Lenferink and M.N. Spijker, On the use of stability regions in
%            the numerical analysis of initial value problems,
%            Math. Comp., 57 (1991), pp. 221-237.
%       L.N. Trefethen, Pseudospectra of matrices, in Numerical Analysis 1991,
%            Proceedings of the 14th Dundee Conference,
%            D.F. Griffiths and G.A. Watson, eds, Pitman Research Notes in
%            Mathematics, volume 260, Longman Scientific and Technical, Essex,
%            UK, 1992, pp. 234-266.
```

```
function A = lotkin(n)
%LOTKIN  Lotkin matrix.
%        A = LOTKIN(N) is the Hilbert matrix with its first row altered to
%        all ones.  A is unsymmetric, ill-conditioned, and has many negative
%        eigenvalues of small magnitude.
%        The inverse has integer entries and is known explicitly.

%        Reference:
%        M. Lotkin, A set of test matrices, MTAC, 9 (1955), pp. 153-161.
```

```
function A = makejcf(n, e, m, X)
%MAKEJCF   A matrix with given Jordan canonical form.
%          MAKEJCF(N, E, M) is a matrix having the Jordan canonical form
%          whose i'th Jordan block is of dimension M(i) with eigenvalue E(i),
%          and where N = SUM(M).
%          Defaults: E = 1:N, M = ONES(SIZE(E)) with M(1) so that SUM(M) = N.
%          The matrix is constructed by applying a random similarity
%          transformation to the Jordan form.
%          Alternatively, the matrix used in the similarity transformation
%          can be specified as a fifth parameter.
```

```
%          In particular, MAKEJCF(N, E, M, EYE(N)) returns the Jordan form
%          itself.
%          NB: The JCF is very sensitive to rounding errors.
```

```
function A = matrix(k, n)
%MATRIX    Test Matrix Toolbox information and matrix access by number.
%          MATRIX(K, N) is the N-by-N instance of the matrix number K in
%          the Test Matrix Toolbox (including some of the matrices provided
%          with MATLAB), with all other parameters set to their default.
%          N.B. Only those matrices which take an arbitrary dimension N
%               are included (thus GALLERY is omitted, for example).
%          MATRIX(K) is a string containing the name of the K'th matrix.
%          MATRIX(0) is the number of matrices, i.e. the upper limit for K.
%          Thus to set A to each N-by-N test matrix in turn use a loop like
%                for k=1:matrix(0)
%                    A = matrix(k, N);
%                    Aname = matrix(k);   % The name of the matrix
%                end
%          MATRIX(-1) returns the version number and date of the toolbox.
%          MATRIX with no arguments lists the names of the M-files in the
%          collection.

%          References:
%          N.J. Higham. The Test Matrix Toolbox for Matlab (version 3.0),
%             Numerical Analysis Report No. 276, Manchester Centre for
%             Computational Mathematics, Manchester, England, September 1995.
%          N.J. Higham, Algorithm 694: A collection of test matrices in
%             MATLAB, ACM Trans. Math. Soft., 17 (1991), pp. 289-305.
%
%          Matrices omitted are: gallery, hadamard, hanowa, lauchli,
%          neumann, wathen, wilk.
%          Matrices provided with MATLAB that are included here: invhilb,
%          magic.
```

```
function S = matsignt(T)
%MATSIGNT    Matrix sign function of a triangular matrix.
%            S = MATSIGN(T) computes the matrix sign function S of the
%            upper triangular matrix T using a recurrence.

%            Adapted from FUNM.  Called by SIGNM.
```

```
function [x, fmax, nf] = mdsmax(fun, x, stopit, savit)
%MDSMAX  Multidirectional search method for direct search optimization.
%        [x, fmax, nf] = MDSMAX(fun, x0, STOPIT, SAVIT) attempts to
%        maximize the function specified by the string fun, using the
%        starting vector x0.  The method of multidirectional search is used.
%        Output arguments:
%               x    = vector yielding largest function value found,
```

```
%               fmax = function value at x,
%               nf   = number of function evaluations.
%       The iteration is terminated when either
%               - the relative size of the simplex is <= STOPIT(1)
%                 (default 1e-3),
%               - STOPIT(2) function evaluations have been performed
%                 (default inf, i.e., no limit), or
%               - a function value equals or exceeds STOPIT(3)
%                 (default inf, i.e., no test on function values).
%       The form of the initial simplex is determined by STOPIT(4):
%          STOPIT(4) = 0: regular simplex (sides of equal length, the default)
%          STOPIT(4) = 1: right-angled simplex.
%       Progress of the iteration is not shown if STOPIT(5) = 0 (default 1).
%       If a non-empty fourth parameter string SAVIT is present, then
%       'SAVE SAVIT x fmax nf' is executed after each inner iteration.
%       NB: x0 can be a matrix.  In the output argument, in SAVIT saves,
%           and in function calls, x has the same shape as x0.

% References:
% [1] V.J. Torczon, Multi-directional search: A direct search algorithm for
%     parallel machines, Ph.D. Thesis, Rice University, Houston, Texas, 1989.
% [2] V.J. Torczon, On the convergence of the multidirectional search
%     algorithm, SIAM J. Optimization, 1 (1991), pp. 123-145.
% [3] N.J. Higham, Optimization by direct search in matrix computations,
%     SIAM J. Matrix Anal. Appl, 14(2): 317-333, April 1993.
```

```
function [Q, R] = mgs(A)
%MGS     Modified Gram-Schmidt QR factorization.
%       [Q, R] = mgs(A) uses the modified Gram-Schmidt method to compute the
%       factorization A = Q*R for m-by-n A of full rank,
%       where Q is m-by-n with orthonormal columns and R is n-by-n.
```

```
function A = minij(n)
%MINIJ   Symmetric positive definite matrix MIN(i,j).
%       A = MINIJ(N) is the N-by-N symmetric positive definite matrix with
%       A(i,j) = MIN(i,j).
%       Properties, variations:
%       INV(A) is tridiagonal: it is minus the second difference matrix
%                   except its (N,N) element is 1.
%       2*A-ONES(N) (Givens' matrix) has tridiagonal inverse and
%                   eigenvalues .5*sec^2([2r-1)PI/4N], r=1:N.
%       (N+1)*ONES(N)-A also has a tridiagonal inverse.

%       References:
%       J. Todd, Basic Numerical Mathematics, Vol. 2: Numerical Algebra,
%          Birkhauser, Basel, and Academic Press, New York, 1977, p. 158.
%       D.E. Rutherford, Some continuant determinants arising in physics and
%          chemistry---II, Proc. Royal Soc. Edin., 63, A (1952), pp. 232-241.
%          (For the eigenvalues of Givens' matrix.)
```

```
function A = moler(n, alpha)
%MOLER   Moler matrix - symmetric positive definite.
%        A = MOLER(N, ALPHA) is the symmetric positive definite N-by-N matrix
%        U'*U where U = TRIW(N, ALPHA).
%        For ALPHA = -1 (the default) A(i,j) = MIN(i,j)-2, A(i,i) = i.
%        A has one small eigenvalue.

%        Nash (1990) attributes the ALPHA = -1 matrix to Moler.
%
%        Reference:
%        J.C. Nash, Compact Numerical Methods for Computers: Linear
%        Algebra and Function Minimisation, second edition, Adam Hilger,
%        Bristol, 1990 (Appendix 1).
```

```
function [A, T] = neumann(n)
%NEUMANN  Singular matrix from the discrete Neumann problem (sparse).
%         NEUMANN(N) is the singular, row diagonally dominant matrix resulting
%         from discretizing the Neumann problem with the usual five point
%         operator on a regular mesh.
%         It has a one-dimensional null space with null vector ONES(N,1).
%         The dimension N should be a perfect square, or else a 2-vector,
%         in which case the dimension of the matrix is N(1)*N(2).

%         Reference:
%         R.J. Plemmons, Regular splittings and the discrete Neumann
%         problem, Numer. Math., 25 (1976), pp. 153-161.
```

```
function [x, fmax, nf] = nmsmax(fun, x, stopit, savit)
%NMSMAX  Nelder-Mead simplex method for direct search optimization.
%        [x, fmax, nf] = NMSMAX(fun, x0, STOPIT, SAVIT) attempts to
%        maximize the function specified by the string fun, using the
%        starting vector x0.  The Nelder-Mead direct search method is used.
%        Output arguments:
%               x    = vector yielding largest function value found,
%               fmax = function value at x,
%               nf   = number of function evaluations.
%        The iteration is terminated when either
%               - the relative size of the simplex is <= STOPIT(1)
%                 (default 1e-3),
%               - STOPIT(2) function evaluations have been performed
%                 (default inf, i.e., no limit), or
%               - a function value equals or exceeds STOPIT(3)
%                 (default inf, i.e., no test on function values).
%        The form of the initial simplex is determined by STOPIT(4):
%           STOPIT(4) = 0: regular simplex (sides of equal length, the default)
%           STOPIT(4) = 1: right-angled simplex.
%        Progress of the iteration is not shown if STOPIT(5) = 0 (default 1).
%        If a non-empty fourth parameter string SAVIT is present, then
%        'SAVE SAVIT x fmax nf' is executed after each inner iteration.
```

```
%        NB: x0 can be a matrix.  In the output argument, in SAVIT saves,
%            and in function calls, x has the same shape as x0.

% References:
% J.E. Dennis, Jr., and D.J. Woods, Optimization on microcomputers:
%     The Nelder-Mead simplex algorithm, in New Computing Environments:
%     Microcomputers in Large-Scale Computing, A. Wouk, ed., Society for
%     Industrial and Applied Mathematics, Philadelphia, 1987, pp. 116-122.
% N.J. Higham, Optimization by direct search in matrix computations,
%     SIAM J. Matrix Anal. Appl, 14(2): 317-333, April 1993.
```

```
function H = ohess(x)
%OHESS  Random, orthogonal upper Hessenberg matrix.
%       H = OHESS(N) is an N-by-N real, random, orthogonal
%       upper Hessenberg matrix.
%       Alternatively, H = OHESS(X), where X is an arbitrary real
%       N-vector (N > 1) constructs H non-randomly using the elements
%       of X as parameters.
%       In both cases H is constructed via a product of N-1 Givens rotations.

%       Note: See Gragg (1986) for how to represent an N-by-N (complex)
%       unitary Hessenberg matrix with positive subdiagonal elements in terms
%       of 2N-1 real parameters (the Schur parametrization).
%       This M-file handles the real case only and is intended simply as a
%       convenient way to generate random or non-random orthogonal Hessenberg
%       matrices.
%
%       Reference:
%       W.B. Gragg, The QR algorithm for unitary Hessenberg matrices,
%       J. Comp. Appl. Math., 16 (1986), pp. 1-8.
```

```
function Q = orthog(n, k)
%ORTHOG Orthogonal and nearly orthogonal matrices.
%       Q = ORTHOG(N, K) selects the K'th type of matrix of order N.
%       K > 0 for exactly orthogonal matrices, K < 0 for diagonal scalings of
%       orthogonal matrices.
%       Available types: (K = 1 is the default)
%       K = 1:  Q(i,j) = SQRT(2/(n+1)) * SIN( i*j*PI/(n+1) )
%               Symmetric eigenvector matrix for second difference matrix.
%       K = 2:  Q(i,j) = 2/SQRT(2*n+1)) * SIN( 2*i*j*PI/(2*n+1) )
%               Symmetric.
%       K = 3:  Q(r,s) = EXP(2*PI*i*(r-1)*(s-1)/n) / SQRT(n)  (i=SQRT(-1))
%               Unitary, the Fourier matrix.  Q^4 is the identity.
%               This is essentially the same matrix as FFT(EYE(N))/SQRT(N)!
%       K = 4:  Helmert matrix: a permutation of a lower Hessenberg matrix,
%               whose first row is ONES(1:N)/SQRT(N).
%       K = 5:  Q(i,j) = SIN( 2*PI*(i-1)*(j-1)/n ) + COS( 2*PI*(i-1)*(j-1)/n ).
%               Symmetric matrix arising in the Hartley transform.
%       K = -1: Q(i,j) = COS( (i-1)*(j-1)*PI/(n-1) )
```

```
%               Chebyshev Vandermonde-like matrix, based on extrema of T(n-1).
%       K = -2: Q(i,j) = COS( (i-1)*(j-1/2)*PI/n) )
%               Chebyshev Vandermonde-like matrix, based on zeros of T(n).

%       References:
%       N.J. Higham and D.J. Higham, Large growth factors in Gaussian
%            elimination with pivoting, SIAM J. Matrix Analysis and  Appl.,
%            10 (1989), pp. 155-164.
%       P. Morton, On the eigenvectors of Schur's matrix, J. Number Theory,
%            12 (1980), pp. 122-127. (Re. ORTHOG(N, 3))
%       H.O. Lancaster, The Helmert Matrices, Amer. Math. Monthly, 72 (1965),
%            pp. 4-12.
%       D. Bini and P. Favati, On a matrix algebra related to the discrete
%            Hartley transform, SIAM J. Matrix Anal. Appl., 14 (1993),
%            pp. 500-507.
```

```
function A = parter(n)
%PARTER    Parter matrix - a Toeplitz matrix with singular values near PI.
%          PARTER(N) is the matrix with (i,j) element 1/(i-j+0.5).
%          It is a Cauchy matrix and a Toeplitz matrix.

%          At the Second SIAM Conference on Linear Algebra, Raleigh, N.C.,
%          1985, Cleve Moler noted that most of the singular values of
%          PARTER(N) are very close to PI.  An explanation of the phenomenon
%          was given by Parter; see also the paper by Tyrtyshnikov.
%
%          References:
%          The MathWorks Newsletter, Volume 1, Issue 1, March 1986, page 2.
%          S.V. Parter, On the distribution of the singular values of Toeplitz
%               matrices, Linear Algebra and Appl., 80 (1986), pp. 115-130.
%          E.E. Tyrtyshnikov, Cauchy-Toeplitz matrices and some applications,
%               Linear Algebra and Appl., 149 (1991), pp. 1-18.
```

```
function P = pascal(n, k)
%PASCAL  Pascal matrix.
%        P = PASCAL(N) is the Pascal matrix of order N: a symmetric positive
%        definite matrix with integer entries taken from Pascal's
%        triangle.
%        The Pascal matrix is totally positive and its inverse has
%        integer entries.  Its eigenvalues occur in reciprocal pairs.
%        COND(P) is approximately 16^N/(N*PI) for large N.
%        PASCAL(N,1) is the lower triangular Cholesky factor (up to signs
%        of columns) of the Pascal matrix.   It is involutary (is its own
%        inverse).
%        PASCAL(N,2) is a transposed and permuted version of PASCAL(N,1)
%        which is a cube root of the identity.

%        References:
%        R. Brawer and M. Pirovino, The linear algebra of the Pascal matrix,
```

```
%           Linear Algebra and Appl., 174 (1992), pp. 13-23 (this paper
%           gives a factorization of L = PASCAL(N,1) and a formula for the
%           elements of L^k).
%       N.J. Higham, Accuracy and Stability of Numerical Algorithms,
%           Society for Industrial and Applied Mathematics, Philadelphia, PA,
%           USA, 1996; sec. 26.4.
%       S. Karlin, Total Positivity, Volume 1, Stanford University Press,
%           1968.  (Page 137: shows i+j-1 choose j is TP (i,j=0,1,...).
%                   PASCAL(N) is a submatrix of this matrix.)
%       M. Newman and J. Todd, The evaluation of matrix inversion programs,
%           J. Soc. Indust. Appl. Math., 6(4):466--476, 1958.
%       H. Rutishauser, On test matrices, Programmation en Mathematiques
%           Numeriques, Editions Centre Nat. Recherche Sci., Paris, 165,
%           1966, pp. 349-365.  (Gives an integral formula for the
%           elements of PASCAL(N).)
%       J. Todd, Basic Numerical Mathematics, Vol. 2: Numerical Algebra,
%           Birkhauser, Basel, and Academic Press, New York, 1977, p. 172.
%       H.W. Turnbull, The Theory of Determinants, Matrices, and Invariants,
%           Blackie, London and Glasgow, 1929.  (PASCAL(N,2) on page 332.)
```

```
function T = pdtoep(n, m, w, theta)
%PDTOEP   Symmetric positive definite Toeplitz matrix.
%         PDTOEP(N, M, W, THETA) is an N-by-N symmetric positive (semi-)
%         definite (SPD) Toeplitz matrix, comprised of the sum of M rank 2
%         (or, for certain THETA, rank 1) SPD Toeplitz matrices.
%         Specifically,
%                 T = W(1)*T(THETA(1)) + ... + W(M)*T(THETA(M)),
%         where T(THETA(k)) has (i,j) element COS(2*PI*THETA(k)*(i-j)).
%         Defaults: M = N, W = RAND(M,1), THETA = RAND(M,1).

%         Reference:
%         G. Cybenko and C.F. Van Loan, Computing the minimum eigenvalue of
%         a symmetric positive definite Toeplitz matrix, SIAM J. Sci. Stat.
%         Comput., 7 (1986), pp. 123-131.
```

```
function P = pei(n, alpha)
%PEI    Pei matrix.
%       PEI(N, ALPHA), where ALPHA is a scalar, is the symmetric matrix
%       ALPHA*EYE(N) + ONES(N).
%       If ALPHA is omitted then ALPHA = 1 is used.
%       The matrix is singular for ALPHA = 0, -N.

%       Reference:
%       M.L. Pei, A test matrix for inversion procedures,
%       Comm. ACM, 5 (1962), p. 508.
```

```
function P = pentoep(n, a, b, c, d, e)
%PENTOEP   Pentadiagonal Toeplitz matrix (sparse).
```

```
%          P = PENTOEP(N, A, B, C, D, E) is the N-by-N pentadiagonal
%          Toeplitz matrix with diagonals composed of the numbers
%          A =: P(3,1), B =: P(2,1), C =: P(1,1), D =: P(1,2), E =: P(1,3).
%          Default: (A,B,C,D,E) = (1,-10,0,10,1) (a matrix of Rutishauser).
%                    This matrix has eigenvalues lying approximately on
%                    the line segment 2*cos(2*t) + 20*i*sin(t).
%
%          Interesting plots are
%          PS(FULL(PENTOEP(32,0,1,0,0,1/4)))  - 'triangle'
%          PS(FULL(PENTOEP(32,0,1/2,0,0,1)))  - 'propeller'
%          PS(FULL(PENTOEP(32,0,1/2,1,1,1)))  - 'fish'

%          References:
%          R.M. Beam and R.F. Warming, The asymptotic spectra of
%             banded Toeplitz and quasi-Toeplitz matrices, SIAM J. Sci.
%             Comput. 14 (4), 1993, pp. 971-1006.
%          H. Rutishauser, On test matrices, Programmation en Mathematiques
%             Numeriques, Editions Centre Nat. Recherche Sci., Paris, 165,
%             1966, pp. 349-365.
```

```
function [est, x, k] = pnorm(A, p, tol, noprint)
%PNORM   Estimate of matrix p-norm (1 <= p <= inf).
%        [EST, x, k] = PNORM(A, p, TOL) estimates the Holder p-norm of a
%        matrix A, using the p-norm power method with a specially
%        chosen starting vector.
%        TOL is a relative convergence tolerance (default 1E-4).
%        Returned are the norm estimate EST (which is a lower bound for the
%        exact p-norm), the corresponding approximate maximizing vector x,
%        and the number of power method iterations k.
%        A nonzero fourth argument causes trace output to the screen.
%        If A is a vector, this routine simply returns NORM(A, p).
%
%        See also NORM, NORMEST.

%        Note: The estimate is exact for p = 1, but is not always exact for
%        p = 2 or p = inf.  Code could be added to treat p = 2 and p = inf
%        separately.
%
%        Calls DUAL and SEQA.
%
%        Reference:
%        N.J. Higham, Estimating the matrix p-norm,
%        Numer. Math., 62 (1992), pp. 539-555.
```

```
function A = poisson(n)
%POISSON   Block tridiagonal matrix from Poisson's equation (sparse).
%          POISSON(N) is the block tridiagonal matrix of order N^2
%          resulting from discretizing Poisson's equation with the
%          5-point operator on an N-by-N mesh.
```

```
%          Reference:
%          G.H. Golub and C.F. Van Loan, Matrix Computations, second edition,
%          Johns Hopkins University Press, Baltimore, Maryland, 1989
%          (Section 4.5.4).
```

```
function [U, H] = poldec(A)
%POLDEC   Polar decomposition.
%         [U, H] = POLDEC(A) computes a matrix U of the same dimension
%         as A, and a Hermitian positive semi-definite matrix H,
%         such that A = U*H.
%         U has orthonormal columns if m>=n, and orthonormal rows if m<=n.
%         U and H are computed via an SVD of A.
%         U is a nearest unitary matrix to A in both the 2-norm and the
%         Frobenius norm.

%         Reference:
%         N.J. Higham, Computing the polar decomposition---with applications,
%         SIAM J. Sci. Stat. Comput., 7(4):1160--1174, 1986.
%
%         (The name 'polar' is reserved for a graphics routine.)
```

```
function A = prolate(n, w)
%PROLATE   Prolate matrix - symmetric, ill-conditioned Toeplitz matrix.
%          A = PROLATE(N, W) is the N-by-N prolate matrix with parameter W.
%          It is a symmetric Toeplitz matrix.
%          If 0 < W < 0.5 then
%             - A is positive definite
%             - the eigenvalues of A are distinct, lie in (0, 1), and
%               tend to cluster around 0 and 1.
%          W defaults to 0.25.

%          Reference:
%          J.M. Varah. The Prolate matrix. Linear Algebra and Appl.,
%          187:269--278, 1993.
```

```
function y = ps(A, m, tol, rl, marksize)
%PS     Dot plot of a pseudospectrum.
%       PS(A, M, TOL, RL) plots an approximation to a pseudospectrum
%       of the square matrix A, using M random perturbations of size TOL.
%       M defaults to a SIZE(A)-dependent value and TOL to 1E-3.
%       RL defines the type of perturbation:
%         RL =  0 (default): absolute complex perturbations of 2-norm TOL.
%         RL =  1:           absolute real perturbations of 2-norm TOL.
%         RL = -1:           componentwise real perturbations of size TOL.
%       The eigenvalues of A are plotted as crosses 'x'.
%       PS(A, M, TOL, RL, MARKSIZE) uses the specified marker size instead
%       of a size that depends on the figure size, the matrix order, and M.
```

```
%       If MARKSIZE < 0, the plot is suppressed and the plot data is returned
%       as an output argument.
%       PS(A, 0) plots just the eigenvalues of A.

%       For a given TOL, the pseudospectrum of A is the set of
%       pseudo-eigenvalues of A, that is, the set
%       { e : e is an eigenvalue of A+E, for some E with NORM(E) <= TOL }.
%
%       Reference:
%       L.N. Trefethen, Pseudospectra of matrices, in D.F. Griffiths and
%            G.A. Watson, eds, Numerical Analysis 1991, Proceedings of the 14th
%            Dundee Conference, vol. 260, Pitman Research Notes in Mathematics,
%            Longman Scientific and Technical, Essex, UK, 1992, pp. 234-266.
```

```
function [x, y, z, m] = pscont(A, k, npts, ax, levels)
%PSCONT   Contours and colour pictures of pseudospectra.
%         PSCONT(A, K, NPTS, AX, LEVELS) plots LOG10(1/NORM(R(z))),
%         where R(z) = INV(z*I-A) is the resolvent of the square matrix A,
%         over an NPTS-by-NPTS grid.
%         NPTS defaults to a SIZE(A)-dependent value.
%         The limits are AX(1) and AX(2) on the x-axis and
%                        AX(3) and AX(4) on the y-axis.
%         If AX is omitted, suitable limits are guessed based on the
%         eigenvalues of A.
%         The eigenvalues of A are plotted as crosses 'x'.
%         K determines the type of plot:
%             K = 0 (default) PCOLOR and CONTOUR
%             K = 1           PCOLOR only
%             K = 2           SURFC (SURF and CONTOUR)
%             K = 3           SURF only
%             K = 4           CONTOUR only
%         The contours levels are specified by the vector LEVELS, which
%         defaults to -10:-1 (recall we are plotting log10 of the data).
%         Thus, by default, the contour lines trace out the boundaries of
%         the epsilon pseudospectra for epsilon = 1e-10, ..., 1e-1.
%         [X, Y, Z, NPTS] = PSCONT(A, ...) returns the plot data X, Y, Z
%         and the value of NPTS used.
%
%         After calling this function you may want to change the
%         color map (e.g., type COLORMAP HOT - see HELP COLOR) and the
%         shading (e.g., type SHADING INTERP - see HELP INTERP).
%         For an explanation of the term 'pseudospectra' see PS.M.
%         When A is real and the grid is symmetric about the x-axis, this
%         routine exploits symmetry to halve the computational work.

%         Colour pseduospectral pictures of this type are referred to as
%         'spectral portraits' by Godunov, Kostin, and colleagues.
%         References:
%         V. I. Kostin, Linear algebra algorithms with guaranteed accuracy,
```

```
%            Technical Report TR/PA/93/05, CERFACS, Toulouse, France, 1993.
%         L.N. Trefethen, Pseudospectra of matrices, in D.F. Griffiths and
%            G.A. Watson, eds, Numerical Analysis 1991, Proceedings of the 14th
%            Dundee Conference, vol. 260, Pitman Research Notes in Mathematics,
%            Longman Scientific and Technical, Essex, UK, 1992, pp. 234-266.
```

```
function B = qmult(A)
%QMULT  Pre-multiply by random orthogonal matrix.
%       QMULT(A) is Q*A where Q is a random real orthogonal matrix from
%       the Haar distribution, of dimension the number of rows in A.
%       Special case: if A is a scalar then QMULT(A) is the same as
%                     QMULT(EYE(A)).

%       Called by RANDSVD.
%
%       Reference:
%       G.W. Stewart, The efficient generation of random
%       orthogonal matrices with an application to condition estimators,
%       SIAM J. Numer. Anal., 17 (1980), 403-409.
```

```
function A = rando(n, k)
%RANDO   Random matrix with elements -1, 0 or 1.
%        A = RANDO(N, K) is a random N-by-N matrix with elements from
%        one of the following discrete distributions (default K = 1):
%          K = 1:  A(i,j) =  0 or 1    with equal probability,
%          K = 2:  A(i,j) = -1 or 1    with equal probability,
%          K = 3:  A(i,j) = -1, 0 or 1 with equal probability.
%        N may be a 2-vector, in which case the matrix is N(1)-by-N(2).
```

```
function A = randsvd(n, kappa, mode, kl, ku)
%RANDSVD  Random matrix with pre-assigned singular values.
%      RANDSVD(N, KAPPA, MODE, KL, KU) is a (banded) random matrix of order N
%      with COND(A) = KAPPA and singular values from the distribution MODE.
%      N may be a 2-vector, in which case the matrix is N(1)-by-N(2).
%      Available types:
%             MODE = 1:   one large singular value,
%             MODE = 2:   one small singular value,
%             MODE = 3:   geometrically distributed singular values,
%             MODE = 4:   arithmetically distributed singular values,
%             MODE = 5:   random singular values with unif. dist. logarithm.
%      If omitted, MODE defaults to 3, and KAPPA defaults to SQRT(1/EPS).
%      If MODE < 0 then the effect is as for ABS(MODE) except that in the
%      original matrix of singular values the order of the diagonal entries
%      is reversed: small to large instead of large to small.
%      KL and KU are the lower and upper bandwidths respectively; if they
%      are omitted a full matrix is produced.
%      If only KL is present, KU defaults to KL.
%      Special case: if KAPPA < 0 then a random full symmetric positive
```

```
%                   definite matrix is produced with COND(A) = -KAPPA and
%                   eigenvalues distributed according to MODE.
%                   KL and KU, if present, are ignored.

%      Reference:
%      N.J. Higham, Accuracy and Stability of Numerical Algorithms,
%         Society for Industrial and Applied Mathematics, Philadelphia, PA,
%         USA, 1996; sec. 26.3.
```

```
function A = redheff(n)
%REDHEFF    A (0,1) matrix of Redheffer associated with the Riemann hypothesis.
%           A = REDHEFF(N) is an N-by-N matrix of 0s and 1s defined by
%               A(i,j) = 1 if j = 1 or if i divides j,
%               A(i,j) = 0 otherwise.
%           It has N - FLOOR(LOG2(N)) - 1 eigenvalues equal to 1,
%           a real eigenvalue (the spectral radius) approximately SQRT(N),
%           a negative eigenvalue approximately -SQRT(N),
%           and the remaining eigenvalues are provably ``small''.
%           Barrett and Jarvis (1992) conjecture that
%             ``the small eigenvalues all lie inside the unit circle
%               ABS(Z) = 1'',
%           and a proof of this conjecture, together with a proof that some
%           eigenvalue tends to zero as N tends to infinity, would yield
%           a new proof of the prime number theorem.
%           The Riemann hypothesis is true if and only if
%           DET(A) = O( N^(1/2+epsilon) ) for every epsilon > 0
%                                           (`!' denotes factorial).
%           See also RIEMANN.

%           Reference:
%           W.W. Barrett and T.J. Jarvis,
%           Spectral Properties of a Matrix of Redheffer,
%           Linear Algebra and Appl., 162 (1992), pp. 673-683.
```

```
function A = riemann(n)
%RIEMANN    A matrix associated with the Riemann hypothesis.
%           A = RIEMANN(N) is an N-by-N matrix for which the
%           Riemann hypothesis is true if and only if
%           DET(A) = O( N! N^(-1/2+epsilon) ) for every epsilon > 0
%                                           (`!' denotes factorial).
%           A = B(2:N+1, 2:N+1), where
%           B(i,j) = i-1 if i divides j and -1 otherwise.
%           Properties include, with M = N+1:
%              Each eigenvalue E(i) satisfies ABS(E(i)) <= M - 1/M.
%              i <= E(i) <= i+1 with at most M-SQRT(M) exceptions.
%              All integers in the interval (M/3, M/2] are eigenvalues.
%
%           See also REDHEFF.
```

```
%           Reference:
%           F. Roesler, Riemann's hypothesis as an eigenvalue problem,
%           Linear Algebra and Appl., 81 (1986), pp. 153-198.
```

```
function z = rq(A,x)
%RQ      Rayleigh quotient.
%        RQ(A,x) is the Rayleigh quotient of A and x, x'*A*x/(x'*x).

%        Called by FV.
```

```
function A = rschur(n, mu, x, y)
%RSCHUR   An upper quasi-triangular matrix.
%         A = RSCHUR(N, MU, X, Y) is an N-by-N matrix in real Schur form.
%         All the diagonal blocks are 2-by-2 (except for the last one, if N
%         is odd) and the k'th has the form [x(k) y(k); -y(k) x(k)].
%         Thus the eigenvalues of A are x(k) +/- i*y(k).
%         MU (default 1) controls the departure from normality.
%         Defaults: X(k) = -k^2/10, Y(k) = -k, i.e., the eigenvalues
%                   lie on the parabola x = -y^2/10.

%         References:
%         F. Chatelin, Eigenvalues of Matrices, John Wiley, Chichester, 1993;
%            Section 4.2.7.
%         F. Chatelin and V. Fraysse, Qualitative computing: Elements
%            of a theory for finite precision computation, Lecture notes,
%            CERFACS, Toulouse, France and THOMSON-CSF, Orsay, France,
%            June 1993.
```

```
function see(A, k)
%SEE    Pictures of a matrix and its (pseudo-) inverse.
%       SEE(A) displays MESH(A), MESH(PINV(A)), SEMILOGY(SVD(A),'o'),
%       and (if A is square) FV(A) in four subplot windows.
%       SEE(A, 1) plots an approximation to the pseudospectrum in the
%       third window instead of the singular values.
%       SEE(A, -1) plots only the eigenvalues in the fourth window,
%       which is much quicker than plotting the field of values.
%       If A is complex, only real parts are used for the mesh plots.
%       If A is sparse, just SPY(A) is shown.
```

```
function y = seqa(a, b, n)
%SEQA   Additive sequence.
%       Y = SEQA(A, B, N) produces a row vector comprising N equally
%       spaced numbers starting at A and finishing at B.
%       If N is omitted then 10 points are generated.
```

```
function x = seqcheb(n, k)
%SEQCHEB   Sequence of points related to Chebyshev polynomials.
%          X = SEQCHEB(N, K) produces a row vector of length N.
%          There are two choices:
%              K = 1:  zeros of T_N,          (the default)
%              K = 2:  extrema of T_{N-1},
%          where T_k is the Chebsyhev polynomial of degree k.
```

```
function y = seqm(a, b, n)
%SEQM   Multiplicative sequence.
%       Y = SEQM(A, B, N) produces a row vector comprising N
%       logarithmically equally spaced numbers, starting at A ~= 0
%       and finishing at B ~= 0.
%       If A*B < 0 and N > 2 then complex results are produced.
%       If N is omitted then 10 points are generated.
```

```
function show(x)
%SHOW   Display signs of matrix elements.
%       SHOW(X) displays X in 'FORMAT +' form, that is,
%       with '+', '-' and  blank representing positive, negative
%       and zero elements respectively.
```

```
function [S, N] = signm(A)
%SIGNM   Matrix sign decomposition.
%        [S, N] = SIGNM(A) is the matrix sign decomposition A = S*N,
%        computed via the Schur decomposition.
%        S is the matrix sign function, sign(A).

%        Reference:
%        N.J. Higham, The matrix sign decomposition and its relation to the
%        polar decomposition, Linear Algebra and Appl., 212/213:3-20, 1994.
```

```
function S = skewpart(A)
%SKEWPART  Skew-symmetric (skew-Hermitian) part.
%          SKEWPART(A) is the skew-symmetric (skew-Hermitian) part of A,
%          (A - A')/2.
%          It is the nearest skew-symmetric (skew-Hermitian) matrix to A in
%          both the 2- and the Frobenius norms.
```

```
function A = smoke(n, k)
%SMOKE     Smoke matrix - complex, with a 'smoke ring' pseudospectrum.
%          SMOKE(N) is an N-by-N matrix with 1s on the
%          superdiagonal, 1 in the (N,1) position, and powers of
%          roots of unity along the diagonal.
%          SMOKE(N, 1) is the same except for a zero (N,1) element.
%          The eigenvalues of SMOKE(N, 1) are the N'th roots of unity;
```

```
%          those of SMOKE(N) are the N'th roots of unity times 2^(1/N).
%
%          Try PS(SMOKE(32)).  For SMOKE(N, 1) the pseudospectrum looks
%          like a sausage folded back on itself.
%          GERSH(SMOKE(N, 1)) is interesting.

%          Reference:
%          L. Reichel and L.N. Trefethen, Eigenvalues and pseudo-eigenvalues of
%          Toeplitz matrices, Linear Algebra and Appl., 162-164:153-185, 1992.
```

```
function A = sparsify(A, p)
%SPARSIFY   Randomly sets matrix elements to zero.
%           S = SPARSIFY(A, P) is A with elements randomly set to zero
%           (S = S' if A is square and A = A', i.e. symmetry is preserved).
%           Each element has probability P of being zeroed.
%           Thus on average 100*P percent of the elements of A will be zeroed.
%           Default: P = 0.25.
```

```
function S = sub(A, i, j)
%SUB     Principal submatrix.
%        SUB(A,i,j) is A(i:j,i:j).
%        SUB(A,i)  is the leading principal submatrix of order i,
%        A(1:i,1:i), if i>0, and the trailing principal submatrix
%        of order ABS(i) if i<0.
```

```
function S = symmpart(A)
%SYMMPART  Symmetric (Hermitian) part.
%          SYMMPART(A) is the symmetric (Hermitian) part of A, (A + A')/2.
%          It is the nearest symmetric (Hermitian) matrix to A in both the
%          2- and the Frobenius norms.
```

```
function [Q, T] = trap2tri(L)
%TRAP2TRI  Unitary reduction of trapezoidal matrix to triangular form.
%          [Q, T] = TRAP2TRI(L), where L is an m-by-n lower trapezoidal
%          matrix with m >= n, produces a unitary Q such that QL = [T; 0],
%          where T is n-by-n and lower triangular.
%          Q is a product of Householder transformations.

%          Called by RANDSVD.
%
%          Reference:
%          G.H. Golub and C.F. Van Loan, Matrix Computations, second edition,
%          Johns Hopkins University Press, Baltimore, Maryland, 1989.
%          P5.2.5, p. 220.
```

```
function T = tridiag(n, x, y, z)
%TRIDIAG  Tridiagonal matrix (sparse).
%         TRIDIAG(X, Y, Z) is the tridiagonal matrix with subdiagonal X,
%         diagonal Y, and superdiagonal Z.
%         X and Z must be vectors of dimension one less than Y.
%         Alternatively TRIDIAG(N, C, D, E), where C, D, and E are all
%         scalars, yields the Toeplitz tridiagonal matrix of order N
%         with subdiagonal elements C, diagonal elements D, and superdiagonal
%         elements E.   This matrix has eigenvalues (Todd 1977)
%                  D + 2*SQRT(C*E)*COS(k*PI/(N+1)), k=1:N.
%         TRIDIAG(N) is the same as TRIDIAG(N,-1,2,-1), which is
%         a symmetric positive definite M-matrix (the negative of the
%         second difference matrix).

%         References:
%         J. Todd, Basic Numerical Mathematics, Vol. 2: Numerical Algebra,
%           Birkhauser, Basel, and Academic Press, New York, 1977, p. 155.
%         D.E. Rutherford, Some continuant determinants arising in physics and
%           chemistry---II, Proc. Royal Soc. Edin., 63, A (1952), pp. 232-241.
```

```
function t = triw(n, alpha, k)
%TRIW   Upper triangular matrix discussed by Wilkinson and others.
%       TRIW(N, ALPHA, K) is the upper triangular matrix with ones on
%       the diagonal and ALPHAs on the first K >= 0 superdiagonals.
%       N may be a 2-vector, in which case the matrix is N(1)-by-N(2) and
%       upper trapezoidal.
%       Defaults: ALPHA = -1,
%                 K = N - 1     (full upper triangle).
%       TRIW(N) is a matrix discussed by Kahan, Golub and Wilkinson.
%
%       Ostrowski (1954) shows that
%         COND(TRIW(N,2)) = COT(PI/(4*N))^2,
%       and for large ABS(ALPHA),
%         COND(TRIW(N,ALPHA)) is approximately ABS(ALPHA)^N*SIN(PI/(4*N-2)).
%
%       Adding -2^(2-N) to the (N,1) element makes TRIW(N) singular,
%       as does adding -2^(1-N) to all elements in the first column.

%       References:
%       G.H. Golub and J.H. Wilkinson, Ill-conditioned eigensystems and the
%          computation of the Jordan canonical form, SIAM Review,
%          18(4), 1976, pp. 578-619.
%       W. Kahan, Numerical linear algebra, Canadian Math. Bulletin,
%          9 (1966), pp. 757-801.
%       A.M. Ostrowski, On the spectrum of a one-parametric family of
%          matrices, J. Reine Angew. Math., 193 (3/4), 1954, pp. 143-160.
%       J.H. Wilkinson, Singular-value decomposition---basic aspects,
%          in D.A.H. Jacobs, ed., Numerical Software---Needs and Availability,
%          Academic Press, London, 1978, pp. 109-135.
```

```
function V = vand(m, p)
%VAND   Vandermonde matrix.
%       V = VAND(P), where P is a vector, produces the (primal)
%       Vandermonde matrix based on the points P, i.e. V(i,j) = P(j)^(i-1).
%       VAND(M,P) is a rectangular version of VAND(P) with M rows.
%       Special case: If P is a scalar then P equally spaced points on [0,1]
%                     are used.

%       Reference:
%       N.J. Higham, Stability analysis of algorithms for solving
%       confluent Vandermonde-like systems, SIAM J. Matrix Anal. Appl.,
%       11 (1990), pp. 23-41.
```

```
function A = wathen(nx, ny, k)
%WATHEN  Wathen matrix - a finite element matrix (sparse, random entries).
%        A = WATHEN(NX,NY) is a sparse random N-by-N finite element matrix
%        where N = 3*NX*NY + 2*NX + 2*NY + 1.
%        A is precisely the 'consistent mass matrix' for a regular NX-by-NY
%        grid of 8-node (serendipity) elements in 2 space dimensions.
%        A is symmetric positive definite for any (positive) values of
%        the 'density', RHO(NX,NY), which is chosen randomly in this routine.
%        In particular, if D = DIAG(DIAG(A)), then
%             0.25 <= EIG(INV(D)*A) <= 4.5
%        for any positive integers NX and NY and any densities RHO(NX,NY).
%        This diagonally scaled matrix is returned by WATHEN(NX,NY,1).

%        Reference:
%        A.J. Wathen, Realistic eigenvalue bounds for the Galerkin
%        mass matrix, IMA J. Numer. Anal., 7 (1987), pp. 449-457.
```

```
function [A, b] = wilk(n)
%WILK   Various specific matrices devised/discussed by Wilkinson.
%       [A, b] = WILK(N) is the matrix or system of order N.
%       N = 3: upper triangular system Ux=b illustrating inaccurate solution.
%       N = 4: lower triangular system Lx=b, ill-conditioned.
%       N = 5: HILB(6)(1:5,2:6)*1.8144.  Symmetric positive definite.
%       N = 21: W21+, tridiagonal.   Eigenvalue problem.

%       References:
%       J.H. Wilkinson, Error analysis of direct methods of matrix inversion,
%          J. Assoc. Comput. Mach., 8 (1961),  pp. 281-330.
%       J.H. Wilkinson, Rounding Errors in Algebraic Processes, Notes on Applied
%          Science No. 32, Her Majesty's Stationery Office, London, 1963.
%       J.H. Wilkinson, The Algebraic Eigenvalue Problem, Oxford University
%          Press, 1965.
```

Acknowledgements

In preparing the earlier test matrix collections I benefited from the helpful suggestions of people too numerous to mention. While working on version 2.0 of the toolbox I received valuable advice from Cleve Moler and Nick Trefethen, and Per Christian Hansen offered helpful comments on a draft version of the manual accompanying version 2.0.

References

[1] E. Anderson, Z. Bai, C. H. Bischof, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992. ISBN 0-89871-294-7. xv+118+listings pp.

[2] Zhaojun Bai. A collection of test matrices for large scale nonsymmetric eigenvalue problems (version 1.0). Manuscript, July 1994.

[3] Richard Bartels and Barry Joe. On generating discrete linear l_∞ test problems. *SIAM J. Sci. Stat. Comput.*, 10(3):550–561, 1989.

[4] James R. Bunch and Linda Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.*, 31(137):163–179, 1977.

[5] Denise Chen and Cleve Moler. *Symbolic Math Toolbox: User's Guide*. The MathWorks, Inc., Natick, MA, USA, 1993.

[6] A. K. Cline and R. K. Rew. A set of counter-examples to three condition number estimators. *SIAM J. Sci. Stat. Comput.*, 4(4):602–611, 1983.

[7] James W. Demmel and A. McKenney. A test matrix generation suite. Preprint MCS-P69-0389, Mathematics and Computer Science Division, Argonne National Laboratory, IL, USA, March 1989. 16 pp. LAPACK Working Note 9.

[8] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Software*, 15(1):1–14, 1989.

[9] Iain S. Duff, Roger G. Grimes, and John G. Lewis. Users' guide for the Harwell–Boeing sparse matrix collection (release 1). Report RAL-92-086, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon, UK, December 1992. 84 pp.

[10] W. H. Enright and J. D. Pryce. Two FORTRAN packages for assessing initial value methods. *ACM Trans. Math. Soft.*, 13(1):1–27, 1987.

[11] Werner L. Frank. Computing eigenvalues of complex matrices by determinant evaluation and by methods of Danilewski and Wielandt. *J. Soc. Indust. Appl. Math.*, 6:378–392, 1958.

[12] F. R. Gantmacher. *The Theory of Matrices*, volume two. Chelsea, New York, 1959. ISBN 0-8284-0133-0. ix+276 pp.

[13] David M. Gay. Electronic mail distribution of linear programming test problems. *Mathematical Programming Society COAL Newsletter*, December:10–12, 1985.

[14] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, USA, second edition, 1989. ISBN 0-8018-3772-3 (hardback), 0-8018-3739-1 (paperback). xix+642 pp.

[15] Robert T. Gregory and David L. Karney. *A Collection of Matrices for Testing Computational Algorithms*. Wiley, New York, 1969. ISBN 0-88275-649-4. ix+154 pp. Reprinted with corrections by Robert E. Krieger, Huntington, New York, 1978.

[16] Per Christian Hansen. Regularization tools. A Matlab package for analysis and solution of discrete ill-posed problems. Report UNIC-92-03, UNI·C, Technical University of Denmark, DK-2800 Lyngby, Denmark, June 1992.

[17] Per Christian Hansen. Test matrices for regularization methods. *SIAM J. Sci. Comput.*, 16(2):506–512, 1995.

[18] Nicholas J. Higham. Computing the polar decomposition—with applications. *SIAM J. Sci. Stat. Comput.*, 7(4):1160–1174, October 1986.

[19] Nicholas J. Higham. A collection of test matrices in MATLAB. Numerical Analysis Report No. 172, University of Manchester, Manchester, England, July 1989.

[20] Nicholas J. Higham. How accurate is Gaussian elimination? In D. F. Griffiths and G. A. Watson, editors, *Numerical Analysis 1989, Proceedings of the 13th Dundee Conference*, volume 228 of *Pitman Research Notes in Mathematics*, pages 137–154. Longman Scientific and Technical, Essex, UK, 1990.

[21] Nicholas J. Higham. Algorithm 694: A collection of test matrices in MATLAB. *ACM Trans. Math. Software*, 17(3):289–305, September 1991.

[22] Nicholas J. Higham. Estimating the matrix p-norm. *Numer. Math.*, 62:539–555, 1992.

[23] Nicholas J. Higham. Optimization by direct search in matrix computations. *SIAM J. Matrix Anal. Appl.*, 14(2):317–333, April 1993.

[24] Nicholas J. Higham. The Test Matrix Toolbox for Matlab. Numerical Analysis Report No. 237, Manchester Centre for Computational Mathematics, Manchester, England, December 1993. 76 pp.

[25] Nicholas J. Higham. The matrix sign decomposition and its relation to the polar decomposition. *Linear Algebra and Appl.*, 212/213:3–20, 1994.

[26] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996. ISBN 0-89871-355-2. Approx xxiv+690 pp. In press.

[27] Nicholas J. Higham and Desmond J. Higham. Large growth factors in Gaussian elimination with pivoting. *SIAM J. Matrix Anal. Appl.*, 10(2):155–164, April 1989.

[28] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985. ISBN 0-521-30586-1. xiii+561 pp.

[29] Roger A. Horn and Charles R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, 1991. ISBN 0-521-30587-X. viii+607 pp.

[30] W. Kahan. Numerical linear algebra. *Canadian Math. Bulletin*, 9:757–801, 1966.

[31] I. J. Lustig. An analysis of an available set of linear programming test problems. *Computers and Operations Research*, 16:173–184, 1989.

[32] Cleve B. Moler. MATLAB's magical mystery tour. *The MathWorks Newsletter*, 7(1):8–9, 1993.

[33] J. J. Moré, B. S. Garbow, and K. E. Hillstrom. Testing unconstrained optimization software. *ACM Trans. Math. Software*, 7:17–41, 1981.

[34] John R. Rice and R. E. Boisvert. *Solving Elliptic Problems using ELLPACK*. Springer-Verlag, New York, 1985.

[35] A. Ruhe. Closest normal matrix finally found! *BIT*, 27:585–598, 1987.

[36] H. Rutishauser. On test matrices. In *Programmation en Mathématiques Numériques, Besançon, 1966*, volume 7 (no. 165) of *Éditions Centre Nat. Recherche Sci., Paris*, pages 349–365, 1968.

[37] G. W. Stewart. Updating a rank-revealing ULV decomposition. *SIAM J. Matrix Anal. Appl.*, 14(2):494–499, 1993.

[38] J. Stoer and C. Witzgall. Transformations by diagonal matrices in a normed space. *Numer. Math.*, 4:158–171, 1962.

[39] Olga Taussky and Marvin Marcus. Eigenvalues of finite matrices. In John Todd, editor, *Survey of Numerical Analysis*, pages 279–313. McGraw-Hill, New York, 1962.

[40] Lloyd N. Trefethen. Pseudospectra of matrices. In D. F. Griffiths and G. A. Watson, editors, *Numerical Analysis 1991, Proceedings of the 14th Dundee Conference*, volume 260 of *Pitman Research Notes in Mathematics*, pages 234–266. Longman Scientific and Technical, Essex, UK, 1992.

[41] Lloyd N. Trefethen. *Spectra and Pseudospectra: The Behavior of Non-Normal Matrices and Operators*. Book in preparation.

[42] J. M. Varah. A generalization of the Frank matrix. *SIAM J. Sci. Stat. Comput.*, 7(3): 835–839, 1986.

[43] Joan R. Westlake. *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations*. Wiley, New York, 1968.

[44] J. H. Wilkinson. Error analysis of floating-point computation. *Numer. Math.*, 2:319–340, 1960.

[45] J. H. Wilkinson. *The Algebraic Eigenvalue Problem*. Oxford University Press, 1965. ISBN 0-19-853403-5 (hardback), 0-19-853418-3 (paperback). xviii+662 pp.

[46] G. Zielke. Report on test matrices for generalized inverses. *Computing*, 36:105–162, 1986.

IMPLICITLY RESTARTED ARNOLDI/LANCZOS METHODS FOR LARGE SCALE EIGENVALUE CALCULATIONS

D. C. Sorensen*
Department of Computational and Applied Mathematics
Rice University
Houston, Texas 77251-1829

October 25, 1995

Abstract

This report provides an introductory overview of the numerical solution of large scale algebraic eigenvalue problems. The main focus is on a class of methods called Krylov subspace projection methods. The Lanczos method is the premier member of this class and the Arnoldi method is a generalization to the nonsymmetric case. A recently developed and very promising variant of the Arnoldi/Lanczos scheme called the Implicitly Restarted Arnoldi Method is presented here in detail. This method is highlighted because of its suitability as a basis for software development. It may be viewed as a truncated form of the implicitly shifted QR-algorithm that is appropriate for very large problems. Based on this technique, a public domain software package called ARPACK has been developed in Fortran 77 for finding a few eigenvalues and eigenvectors of large scale symmetric, nonsymmetric, standard or generalized problems. This package has performed well on workstations, parallel-vector supercomputers, distributed memory parallel systems and clusters of workstations. The important features of this package are presented along with a discussion some applications and performance indicators.

AMS classification: Primary 65F15; Secondary 65G05

Key words and phrases: Large scale eigenvalue problems, Arnoldi methods, Lanczos methods, Krylov subspace projection, Implicit restarting,

*This work was supported in part by the National Science Foundation contract ASC-9408795, National Science Foundation cooperative agreement CCR-9120008 and by DARPA through U.S. Army ORA7453.01.

IMPLICITLY RESTARTED ARNOLDI/LANCZOS METHODS FOR LARGE SCALE EIGENVALUE CALCULATIONS

D.C. SORENSEN
Department of Computational and Applied Mathematics
Rice University
P.O. Box 1892
Houston, TX 77251
sorensen@rice.edu

1 Introduction

Eigenvalues and eigenfunctions of linear operators are important to many areas of applied mathematics. The ability to approximate these quantities numerically is becoming increasingly important in a wide variety of applications. This increasing demand has fueled interest in the development of new methods and software for the numerical solution of large scale algebraic eigenvalue problems. In turn, the existence of these new methods and software, along with the dramatically increased computational capabilities now available, has enabled the solution of problems that would not have even been posed five or ten years ago. Until very recently, software for large scale nonsymmetric problems was virtually non-existent. Fortunately, the situation is improving rapidly.

The purpose of this article is to provide an overview of the numerical solution of large scale algebraic eigenvalue problems. The focus will be on a class of methods called Krylov subspace projection methods. The well known Lanczos method is the premier member of this class. The Arnoldi method generalizes the Lanczos method to the non-symmetric case. A recently developed variant of the Arnoldi/Lanczos scheme called the Implicitly Restarted Arnoldi Method [44] is presented here in some depth. This method is highlighted because of its suitability as a basis for software development.

The discussion begins with a brief synopsis of the theory and the basic iterations suitable for large scale problems to motivate the introduction of Krylov subspaces. Then the Lanczos/Arnoldi factorization is introduced along with a discussion of its important approximation properties. Spectral transformations are presented as a means to improve these approximation properties and to enhance convergence of the basic methods. Restarting is introduced as a way to overcome intractable storage and computational requirements in the

original Arnoldi method. Implicit restarting is a new sophisticated variant of restarting. This new technique may be viewed as a truncated form of the powerful implicitly shifted QR technique that is suitable for large scale problems. Implicit restarting provides a means to approximate a few eigenvalues with user specified properties in space proportional to nk where k is the number of eigenvalues sought.

Generalized eigenvalue problems are discussed in some detail. They arise naturally in PDE applications and they have a number of subtleties with respect to numerically stable implementation of spectral transformations.

Software issues and considerations for implementation on vector and parallel computers are introduced in the later sections. Implicit restarting has provided a means to develop very robust and efficient software for a wide variety of large scale eigen-problems. A public domain software package called ARPACK has been developed in Fortran 77. This package has performed well on workstations, parallel-vector supercomputers, distributed memory parallel systems and clusters of workstations. The features of this package along with some applications and performance indicators occupy the final section of this paper.

2 Eigenvalues, Power Iterations, and Spectral Transformations

A brief discussion of the mathematical structure of the eigenvalue problem is necessary to fix notation and introduce ideas that lead to an understanding of the behavior, strengths and limitations of the algorithms. In this discussion, the real and complex number fields are denoted by $\mathbf{R}$ and $\mathbf{C}$ respectively. The standard n-dimensional real and complex vectors are denoted by $\mathbf{R}^n$ and $\mathbf{C}^n$ and the symbols $\mathbf{R}^{m\times n}$ and $\mathbf{C}^{m\times n}$ will denote the real and complex matrices m rows and n columns. Scalars are denoted by lower case Greek letters, vectors are denoted by lower case Latin letters and matrices by capital Latin letters. The transpose of a matrix A is denoted by A^T and the conjugate-transpose by A^H. The symbol, $\|\cdot\|$ will denote the Euclidean or 2-norm of a vector. The standard basis of $\mathbf{C}^n$ is denoted by the set $\{e_j\}_{j=1}^n$.

The set of numbers $\sigma(A) \equiv \{\lambda \in \mathbf{C} : rank(A-\lambda I) < n)\}$ is called the *spectrum* of A. The elements of this discrete set are the *eigenvalues* of A and they may be characterized as the n roots of the *characteristic polynomial* $p_A(\lambda) \equiv det(\lambda I - A)$. Corresponding to each distinct eigenvalue $\lambda \in \sigma(A)$ is at least one nonzero vector x such that $Ax = x\lambda$. This vector is called a *right eigenvector* of A corresponding to the eigenvalue λ. The pair (x, λ) is called an *eigenpair*. A nonzero vector y such that $y^H A = \lambda y^H$ is called a *left eigenvector*. The multiplicity $n_a(\lambda)$ of λ as a root of the characteristic polynomial is the *algebraic* multiplicity and the dimension $n_g(\lambda)$ of $Null(\lambda I - A)$ is the *geometric* multiplicity of λ. A matrix is *defective* if $n_g(\lambda) < n_a(\lambda)$ and otherwise A is *non-defective*. The eigenvalue λ is *simple* if $n_a(\lambda) = 1$.

A subspace $\mathcal{S}$ of $\mathbf{C}^{n\times n}$ is called an *invariant subspace* of A if $A\mathcal{S} \subset \mathcal{S}$. It is straightforward to show if $A \in \mathbf{C}^{n\times n}$, $X \in \mathbf{C}^{n\times k}$ and $B \in \mathbf{C}^{k\times k}$ satisfy

$$AX = XB, \tag{1}$$

then $\mathcal{S} \equiv \mathrm{R}ange(X)$ is an invariant subspace of A. Moreover, if X has full column rank k then the columns of X form a basis for this subspace and $\sigma(B) \subset \sigma(A)$. If $k = n$ then $\sigma(B) = \sigma(A)$ and A is said to be *similar* to B under the *similarity transformation* X. A is *diagonalizable* if it is similar to a diagonal matrix and this property is equivalent to A being non-defective.

An extremely important theorem to the study of numerical algorithms for eigenproblems is the Schur decomposition. It states that every square matrix is *unitarily similar* to an upper triangular matrix. In other words, for any linear operator on $\mathbf{C}^n$, there is a unitary basis in which the operator has an upper triangular matrix representation.

Theorem 2.1 *(Schur Decomposition). Let $A \in \mathbf{C}^{n\times n}$. Then there is a unitary matrix Q and an upper triangular matrix R such that*

$$AQ = QR. \tag{2}$$

The diagonal elements of R are the eigenvalues of A.

From the Schur decomposition, the fundamental structure of Hermitian and normal matrices is easily exposed:

Lemma 2.2 *A matrix $A \in \mathbf{C}^{n\times n}$ is normal ($AA^H = A^HA$) if and only if $A = Q\Lambda Q^H$ with $Q \in \mathbf{C}^{n\times n}$ unitary and $\Lambda \in \mathbf{C}^{n\times n}$ diagonal. A matrix $A \in \mathbf{C}^{n\times n}$ is Hermitian ($A = A^H$) if and only if $A = Q\Lambda Q^H$ with $Q \in \mathbf{C}^{n\times n}$ unitary and $\Lambda \in \mathbf{R}^{n\times n}$ diagonal. In either case, the diagonal entries of Λ are the eigenvalues of A and the columns of Q are the corresponding eigenvectors.*

The proof follows easily through substitution of the Schur decomposition in place of A in each of the defining relationships. The columns of Q are called Schur vectors in general and these are eigenvectors of A if and only if A is normal.

For purposes of algorithmic development this structure is fundamental. In fact, the well known Implicitly Shifted QR-Algorithm [16] is designed to produce a sequence of unitary similarity transformations Q_j that iteratively reduce A to upper triangular form. This algorithm begins with an initial unitary similarity transformation V of A to the condensed form $AV = VH$ where H is upper Hessenberg (tridiagonal in case $A = A^H$). Then the following iteration is performed:

Algorithm 1: Implicitly Shifted QR-iteration

Input: (A, V, H) with $AV = VH, V^HV = I$, H upper Hessenberg;

For $j = 1, 2, 3, \ldots$ until *convergence*,

(a1.1) Select a shift $\mu \leftarrow \mu_j$

(a1.2) Factor $[Q, R] = \mathrm{qr}(H - \mu I)$;

(a1.3) $H \leftarrow Q^HHQ$; $V \leftarrow VQ$;

End_For

where Q is unitary and R is upper triangular (i.e. the QR factorization of $H - \mu I$). It is easy to see that H is unitarily similar to A throughout the course of this iteration. The

iteration is continued until the subdiagonal elements of H converge to zero, i.e. until a Schur decomposition has been (approximately) obtained. In the standard implicitly shifted QR-iteration, the unitary matrix Q is never actually formed. it is computed indirectly as a product of 2×2 Givens or 3×3 Householder transformations through a "bulge chase" process. The elegant details of an efficient and stable implementation would be too much of a digression here. They may be found in [18]. The convergence behavior of this iteration is fascinating. The columns of V converge to Schur vectors at various rates. These rates are fundamentally linked to the simple power method and its rapidly convergent variant, inverse iteration (see [51]).

Despite the extremely fast rate of convergence and the efficient use of storage, the implicitly shifted QR method is not suitable for large scale problems and it has proved to be extremely difficult to parallelize. Large scale problems are typically sparse or structured so that a matrix-vector product $w \leftarrow Av$ may be computed with time and storage proportional to n rather than n^2 . A method based upon full similarity transformations quickly destroys this structure. Storage and operation counts become order n^2. Hence, there is considerable motivation for methods that only require matrix-vector products with the original A.

2.1 SINGLE VECTOR POWER ITERATIONS

Probably the oldest algorithm for approximating eigenvalues and corresponding eigenvectors of a matrix is the power method. This method is an important tool in its own right when conditions are appropriate. It is very simple and only requires matrix-vector products along with two vectors of storage. In addition to its role as an algorithm, the method is central to the development, understanding, and convergence analysis of all of the iterative methods discussed here.

Algorithm 2: The Power Method

Input: (A, v_o)

Put $v = v_o / \|v_o\|_\infty$;

For $j = 1, 2, 3, \ldots$ until $convergence$,

(a2.1) $w \leftarrow Av$;

(a2.2) $\lambda = \frac{v^H w}{v^H v}$;

(a2.3) $i =$ i_max (w);

(a2.4) $v \leftarrow v/(e_i^T w)$;

End_For

At Step (a2.3), i is the index of the element of w with largest absolute value. It is easily seen that the contents of v after k-steps of this iteration will be the vector

$$v_k = (\frac{1}{e_i^T A^k v_o}) A^k v_o = (\frac{\rho_k}{e_i^T A^k v_o})(\frac{1}{\rho_k} A^k v_o)$$

for any nonzero scalar ρ_k. In particular, this iteration may be analyzed as if the vectors had been scaled by $\rho_k = \lambda_1^k$ at each step, with λ_1 an eigenvalue of A with largest magnitude.

If A is diagonalizable with eigenpairs $\{(x_j, \lambda_j), 1 \leq j \leq n\}$ and v_o has the expansion $v_o = \sum_{j=1}^{n} x_j \gamma_j$ in this basis then

$$\frac{1}{\lambda_1^k} A^k v_o = \frac{1}{\lambda_1^k} \sum_{j=1}^{n} A^k x_j \gamma_j = \sum_{j=1}^{n} x_j (\lambda_j / \lambda_1)^k \gamma_j. \tag{3}$$

If λ_1 is a simple eigenvalue then

$$\left(\frac{\lambda_j}{\lambda_1} \right)^k \to 0, \quad 2 \leq j \leq n.$$

It follows that $v_k \to x_1 / (e_i^T x_1)$, where $i =$ i_max (x_1), at a linear rate with a convergence factor of $|\frac{\lambda_2}{\lambda_1}|$.

While the power method is useful, it has two obvious drawbacks. Convergence may be arbitrarily slow or may not happen at all. Only one eigenvalue and corresponding vector can be found.

2.2 SPECTRAL TRANSFORMATIONS

The basic power iteration may be modified to overcome these difficulties. The most fundamental modification is to employ a *spectral transformation*. Spectral transformations are generally based upon the following:

Let $A \in \mathbf{C}^{n \times n}$ have an eigenvalue λ with corresponding eigenvector x.

1. Let $p(\tau) = \gamma_0 + \gamma_1 \tau + \gamma_2 \tau^2 + \ldots + \gamma_k \tau^k$. Then $p(\lambda)$ is an eigenvalue of the matrix $p(A) = \gamma_0 I + \gamma_1 A + \gamma_2 A^2 + \ldots + \gamma_k A^k$ with corresponding eigenvector x (i.e. $p(A)x = x p(\lambda)$).

2. If $r(\tau) = \frac{p(\tau)}{q(\tau)}$ where p and q are polynomials with $q(A)$ nonsingular, define $r(A) = [q(A)]^{-1} p(A)$. Then $r(\lambda)$ is an eigenvalue of $r(A)$ with corresponding eigenvector x.

It is often possible to construct a polynomial or rational function $\phi(\tau)$ such that

$$|\phi(\lambda_i)| \ll |\phi(\lambda_j)| \quad \text{for} \quad 1 \leq j \leq n, \quad j \neq i,$$

where λ_i is an eigenvalue of particular interest. This is called a spectral transformation since the eigenvectors of the transformed matrix $\phi(A)$ remain the same, but the corresponding eigenvalues λ_j are transformed to $\phi(\lambda_j)$. Applying the power method with $\phi(A)$ in place of A will then produce the eigenvector $q \equiv x_i$ corresponding to λ_i at a linear a convergence rate with a convergence factor of $|\frac{\phi(\lambda_j)}{\phi(\lambda_i)}| << 1$. Once the eigenvector has been found, the eigenvalue $\lambda \equiv \lambda_i$ may be calculated directly from a Rayleigh quotient $\lambda = q^H A q / q^H q$.

2.3 INVERSE ITERATION

Spectral transformation can lead to dramatic enhancement of the convergence of the power method. Polynomial transformations may be applied using only matrix-vector products.

Rational transformations require the solution of linear systems with the transformed matrix as the coefficient matrix. The simplest rational transformation turns out to be very powerful and is almost exclusively used for this purpose. If $\mu \notin \sigma(A)$ then $A - \mu I$ is invertible and $\sigma([A - \mu I]^{-1}) = \{1/(\lambda - \mu) : \lambda \in \sigma(A)\}$. This transformation is very successful since eigenvalues near the shift μ are transformed to extremal eigenvalues which are well separated from the other ones while the original extremal eigenvalues are transformed near the origin. Hence under this transformation the eigenvector q corresponding to the eigenvalue of A that is closest to μ may be readily found and the corresponding eigenvalue may obtained either through the formula $\lambda = \theta + 1/\mu$, where θ is the eigenvalue of the transformed matrix, or it may be calculated directly from a Rayleigh quotient.

Algorithm 3: The Inverse Power Method

Input: $(A, v_o, \mu\ )$

Put $v = v_o / \|v_o\|_\infty$;

For $j = 1, 2, 3, ...$ until $convergence$,

(a3.1) Solve $(A - \mu I)w = v$;

(a3.2) $\lambda = \mu + \frac{v^H w}{w^H w}$;

(a3.3) $i =$ i_max (w);

(a3.4) $v \leftarrow v/(e_i^T w)$;

End_For

Observe that the formula for λ at Step (a3.2) is equivalent to forming $\lambda = (w^H A w)/(w^H w)$ so an additional matrix vector product is not necessary to obtain the Rayleigh quotient estimate. The analysis of convergence remains entirely in tact. This iteration converges linearly with the convergence factor

$$\frac{|\lambda_1 - \mu|}{|\lambda_2 - \mu|}$$

where the eigenvalues of A have been re-indexed so that $|\lambda_1 - \mu| < |\lambda_2 - \mu| \leq |\lambda_3 - \mu| \leq ... \leq |\lambda_n - \mu|$. Hence, the convergence becomes faster as μ gets closer to λ_1.

This result is encouraging but still leaves us wondering how to select the shift μ to be close to the unknown eigenvalue we are trying to compute. In many applications the choice is apparent from the requirements of the problem. It is also possible to change the shift at each iteration at the expense of a new matrix factorization at each step. An obvious choice would be to replace the shift with the current Rayleigh quotient estimate. This method, called Rayleigh Quotient Iteration, has very impressive convergence rates indeed. Rayleigh Quotient Iteration converges at a quadratic rate in general and at a cubic rate on Hermitian problems. For a more detailed discussion of the eigenvalue problem and basic algorithms see [52, 46, 18].

3 Krylov Subspaces and Projection Methods

Although the rate of convergence can be improved to an acceptable level through spectral transformations, power iterations are only able to find one eigenvector at a time. If more vectors are sought, then various deflation techniques (such as orthogonalizing against previously converged eigenvectors) and shift strategies must be introduced. One alternative is to introduce a block form of the simple power method which is often called subspace iteration. This important class of algorithms has been developed and investigated in [46]. Several software efforts have been based upon this approach [3, 47, 12]. However, there is another class of algorithms called Krylov subspace projection methods that are based upon the intricate structure of the sequence of vectors naturally produced by the power method.

An examination of the behavior of the power sequence as exposed in equation (3) hints that the successive vectors produced by a power iteration may contain considerable information along eigenvector directions corresponding to eigenvalues other than the one with largest magnitude. The expansion coefficients of the vectors in the power sequence evolve in a very structured way. Therefore, linear combinations of the these vectors might well be devised to expose additional eigenvectors. A single vector power iteration simply ignores this additional information, but more sophisticated techniques may be employed to extract it.

If one hopes to obtain additional information through various linear combinations of the power sequence, it is natural to formally consider the *Krylov* subspace

$$\mathcal{K}_k(A, v_1) = \text{Span } \{v_1, Av_1, A^2v_1, \ldots, A^{k-1}v_1\}$$

and to attempt to formulate the best possible approximations to eigenvectors from this subspace.

It is reasonable to construct approximate eigenpairs from this subspace by imposing a Galerkin condition: A vector $x \in \mathcal{K}_k(A, v_1)$ is called a *Ritz vector* with corresponding *Ritz value* θ if the Galerkin condition

$$< w, Ax - x\theta >= 0 \ , \quad \text{for all} \quad w \in \mathcal{K}_k(A, v_1)$$

is satisfied. There are some immediate consequences of this definition: Let W be a matrix whose columns form an orthonormal basis for $\mathcal{K}_k \equiv \mathcal{K}_k(A, v_1)$. Let $\mathcal{P} = WW^H$ denote the related orthogonal projector onto $\mathcal{K}_k$ and define $\hat{A} \equiv \mathcal{P}A\mathcal{P} = WBW^H$ where $B \equiv W^HAW$. It can be shown that

Lemma 3.1 *For the quantities defined above:*

1. (x, θ) *is a Ritz-pair if and only if* $x = Wy$ *with* $By = y\theta$.
2. $\|(I - \mathcal{P})AW\| = \|(A - \hat{A})W\| \leq \|(A - M)W\|$
   for all $M \in \mathbf{C}^{n\times n}$ *such that* $M\mathcal{K}_k \subset \mathcal{K}_k$.
3. *The Ritz-pairs* (x, θ) *and the minimum value* $\|(I - \mathcal{P})AW\|$ *are independent of the choice of orthonormal basis* W.

Item (1) follows immediately from the Galerkin condition since it implies that $0 = W^H(AWy -$

$Wy\theta) = By - y\theta$. Item (2) is easily shown using invariance of $\|\cdot\|$ under unitary transformations. Item (3) follows from the fact that V is an orthonormal basis for $\mathcal{K}_k$ if and only if $V = WQ$ for some $k \times k$ unitary matrix Q. With this change of basis $\hat{A} = VHV^H$, where $H = V^H AV = Q^H BQ$. Since H is unitarily similar to B, the Ritz-values remain the same and the Ritz-vectors are of the form $x = Wy = V\hat{y}$ where $\hat{y} = Q^H y$.

These facts are actually valid for any k dimensional subspace $\mathcal{S}$ in place of $\mathcal{K}_k$. The following properties are consequences of the fact that every $w \in \mathcal{K}_k$ is of the form $w = \phi(A)v_1$ for some polynomial ϕ of degree less than k.

Lemma 3.2 *For the quantities defined above:*

1. *If q is a polynomial of degree less than k then*

   $$q(A)v_1 = q(\hat{A})v_1 = Wq(B)z_1$$

   where $v_1 = Wz_1$, and if degree of q is k then

   $$\mathcal{P}q(A)v_1 = q(\hat{A})v_1.$$

2. *If $\hat{p}(\lambda) \equiv det(\lambda I - B)$ is the characteristic polynomial of B then $\hat{p}(\hat{A}) = 0$ and $\|\hat{p}(A)v_1\| \leq \|q(A)v_1\|$ for all monic polynomials of degree k.*

3. *If y is any vector in $\mathbf{C}^k$ then $AWy - WBy = \gamma\hat{p}(A)v_1$ for some scalar γ.*

4. *If (x, θ) is any Ritz-pair for A with respect to $\mathcal{K}_k$ then*

   $$Ax - x\theta = \gamma\hat{p}(A)v_1$$

   for some scalar γ.

This discussion follows the treatment given by Saad in [40] and in his earlier papers. While these facts may seem esoteric, they have important algorithmic consequences. First, it should be noted that $\mathcal{K}_k$ is an invariant subspace for A if and only if $v_1 = Vy$, where $AV = VR$ with $V^H V = I_k$ and R is $k \times k$ upper triangular. Also, $\mathcal{K}_k$ is an invariant subspace for A if $v_1 = Xy$, where $X \in \mathbf{C}^{n\times k}$ and $AX = X\Lambda$ with Λ diagonal . This follows from items (2) and (3) since there is a k degree monic polynomial q such that $q(R) = 0$ and hence $\|\hat{p}(A)v_1\| \leq \|q(A)v_1\| = \|Vq(R)y\| = 0$ (A similar argument holds when $v_1 = Xy$).

Secondly, there is some algorithmic motivation to seek a convenient orthonormal basis $V = WQ$ that will provide a means to successively construct these basis vectors. It is possible to construct a $k \times k$ unitary Q using standard Householder transformations such that $v_1 = Ve_1$ and $H = Q^H BQ$ is upper Hessenberg with non-negative subdiagonal elements. It is also possible to show using item (3) that in this basis,

$$AV = VH + fe_k^T, \quad \text{where} \quad f = \gamma\hat{p}(A)v_1$$

and $V^H f = 0$ follows from the Galerkin condition.

The first observation shows that if it is possible to obtain a v_1 as a linear combination of k eigenvectors of A then $f = 0$ and V is an orthonormal basis for an invariant subspace of A and that the Ritz values $\sigma(H) \subset \sigma(A)$ and corresponding Ritz vectors are eigenpairs for A. The second observation leads to the Lanczos/Arnoldi process [23, 1].

4 The Arnoldi Factorization

Definition : If $A \in \mathbf{C}^{n\times n}$ then a relation of the form

$$AV_k = V_k H_k + f_k e_k^T$$

where $V_k \in \mathbf{C}^{n\times k}$ has orthonormal columns, $V_k^H f_k = 0$ and $H_k \in \mathbf{C}^{k\times k}$ is upper Hessenberg with non-negative subdiagonal elements is called a $k-$*step Arnoldi Factorization* of A. If A is Hermitian then H_k is real, symmetric and tridiagonal and the relation is called a $k-$*step Lanczos Factorization* of A. The columns of V_k are referred to as the *Arnoldi vectors* or *Lanczos vectors* respectively.

The development of this factorization has been purely through the consequences of the orthogonal projection imposed by the Galerkin conditions. A more straightforward but less illuminating derivation is to simply truncate the reduction of A to Hessenberg form that precedes the implicitly shifted QR-iteration by equating the first k columns on both sides of the complete reduction $AV = VH$. An alternative way to write this factorization is

$$AV_k = (V_k, v_{k+1}) \begin{pmatrix} H_k \\ \beta_k e_k^T \end{pmatrix} \quad \text{where} \quad \beta_k = \|f_k\| \quad \text{and} \quad v_{k+1} = \frac{1}{\beta_k} f_k \ .$$

This factorization may be used to obtain approximate solutions to a linear system $Ax = b$ if $b = v_1\beta_o$ and this underlies the GMRES method [41]. However, the purpose here is to investigate the use of this factorization to obtain approximate eigenvalues and eigenvectors. The discussion of the previous section implies that Ritz pairs satisfying the Galerkin condition are immediately available from the eigenpairs of the small projected matrix H.

If $H_k y = y\theta$ then the vector $x = V_k y$ satisfies

$$\|Ax - x\theta\| = \|(AV_k - V_k H_k)y\| = |\beta_k e_k^T y|.$$

The number $|\beta_k e_k^T y|$ is called the *Ritz estimate* for this the Ritz pair (x, θ) as an approximate eigenpair for A. Observe that if (x, θ) is a Ritz pair then

$$\theta = y^H H_k y = (V_k y)^H A (V_k y) = x^H A x$$

is a Rayleigh Quotient (assuming $\|y\| = 1$) and the associated Rayleigh Quotient residual $r(x) = Ax - x\theta$ satisfies

$$\|r(x)\| = |\beta_k e_k^T y|.$$

When A is Hermitian, this relation may be used to provide computable rigorous bounds on the accuracy of the eigenvalues of H as approximations to eigenvalues of A (see [34]). When A is non-Hermitian the possibility of non-normality precludes such bounds and one can only say that the RQ-residual is small if $|\beta_k e_k^T y|$ is small. However, in either case, if

$f_k = 0$ these the Ritz pairs become exact eigenpairs of A.

This factorization may be advanced one step at the cost of a (sparse) matrix-vector product involving A and two dense matrix vector products involving V_k^T and V_k.

The explicit steps needed to form a k-Step Arnoldi Factorization are:

Algorithm 4: The k-Step Arnoldi Factorization

Input: (A, v)

Put $v_1 = v/\|v\|$; $w = Av_1$;$\alpha_1 = v_1^H w$;

Put $f_1 \leftarrow w - v_1\alpha_1$; $V \leftarrow (v_1)$; $H \leftarrow (\alpha_1)$;

For $j = 1, 2, 3, ...$k-1,

(a4.1) $\beta_j = \|f_j\|$; $v_{j+1} \leftarrow f_j/\beta_j$;

(a4.2) $V_{j+1} \leftarrow (V_j, v_{j+1})$; $\hat{H}_j \leftarrow \begin{pmatrix} H_j \\ \beta_j e_j^T \end{pmatrix}$;

(a4.3) $z \leftarrow Av_{j+1}$;

(a4.4) $h \leftarrow V_{j+1}^T z$; $f_{j+1} \leftarrow z - V_{j+1}h$;

(a4.5) $H_{j+1} \leftarrow (\hat{H}_j, h)$;

End_For

In exact arithmetic, the columns of V form an orthonormal basis for the Krylov subspace and H is the orthogonal projection of A onto this space. In finite precision arithmetic, care must be taken to assure that the computed vectors are orthogonal to working precision. The method proposed by Daniel, Gragg, Kaufman and Stewart (DGKS) in [9] provides an excellent way to construct a vector f_{j+1} that is numerically orthogonal to V_{j+1}. It amounts to computing a correction

$$s = V_{j+1}^T f_{j+1}; \quad f_{j+1} \leftarrow f_{j+1} - V_{j+1}s; \quad h \leftarrow h + s;$$

just after Step (a4.4) if necessary. A simple test can be devised to avoid this DGKS correction if it is not needed.

The dense matrix-vector products at Step (a4.4) and also the correction may be accomplished using Level 2 BLAS. This is quite important for performance on vector, and parallel-vector supercomputers. The BLAS operation _GEMV is easily parallelized and vectorized and has a much better ratio of floating point computation to data movement [10, 11]. The Modified Gram-Schmidt Process (MGS) is often used in the construction of Arnoldi factorizations. However, MGS will definitely not produce numerically orthogonal basis vectors in practice. Moreover, MGS cannot be formulated in terms of Level 2 BLAS unless all of the vectors to be orthogonalized are known in advance and this is not the case in the Arnoldi process. For these reasons, classical Gram-Schmidt orthogonalization with the DGKS correction step is highly recommended.

The information obtained through this process is completely determined by the choice of the starting vector. Eigen-information of interest may not appear until k gets very large.

In this case it becomes intractable to maintain numerical orthogonality of the basis vectors V_k. Moreover, extensive storage will be required and repeatedly finding the eigensystem of H will become prohibitive at a cost of $\mathcal{O}(k^3)$ flops.

Failure to maintain orthogonality leads to several numerical difficulties. In a certain sense, the computation (or approximation) of the projection indicated at Step (a4.4) in a way that overcomes these difficulties has been the main source of research activity in these Krylov subspace projection methods. The computational difficulty stems from the fact that $\|f_k\| = 0$ if and only if the columns of V_k span an invariant subspace of A. When V_k "nearly" spans such a subspace $\|f_k\|$ will be small. Typically, in this situation, a loss of significant digits will take place at Step (a4.4) through numerical cancellation unless special care is taken (i.e. use of the DGKS correction).

It is desirable for $\|f_k\|$ to become small because this indicates that the eigenvalues of H are accurate approximations to the eigenvalues of A. However, this "convergence" will indicate a probable loss of numerical orthogonality in V. Moreover, if subsequent Arnoldi vectors are not forced to be orthogonal to the converged ones then components along these directions re-enter the basis via round-off effects and quickly cause a spurious copy of the previously computed eigenvalue to appear repeatedly in the spectrum of the projected matrix H. The identification of this phenomenon in the symmetric case and the first rigorous numerical treatment is due to Paige [31]. There have been several approaches to overcome this problem in the symmetric case. They include: (1) complete re-orthogonalization, which may be accomplished through maintaining V in product Householder form [50, 17] or through the Modified Gram-Schmidt processes with re-orthogonalization [9]. (2) Selective re-orthogonalization, which has been proposed by Parlett and has been heavily researched by him and his students. Most notably, the theses and subsequent papers and computer codes of Scott and of Simon have developed this idea [34, 33, 43]. (3) No re-orthogonalization, which has been developed by Cullum and her colleagues. This last option introduces the almost certain possibility of introducing spurious eigenvalues. Various techniques have been developed to detect and deal with the presence of spurious eigenvalues [6, 8].

The appearance of spurious eigenvalues may be avoided through complete orthogonalization of the Arnoldi (or Lanczos) vectors using the DGKS correction. Computational cost has been cited as the reason for not employing this option . However, the cost will be reasonable if one is able to fix k at a modest size and then update the starting vector $v_1 = V_k e_1$ while repeatedly doing k-Arnoldi steps. This approach was introduced in [21] and developed further by [7] for the symmetric case. Saad [38, 39, 40] has developed explicit restarting for the nonsymmetric case. Restarting has proven to have important consequences for the development of numerical software based upon Arnoldi's method and this will be explored in the following section.

5 Restarting the Arnoldi Method

An unfortunate aspect of the Lanczos/Arnoldi process is that one cannot know in advance how many steps will be required before eigenvalues of interest are well approximated by Ritz

values. This is particularly true when the problem has a wide range of eigenvalues but the eigenvalues of interest are clustered. For example, in computational chemistry, problems are usually symmetric and positive definite and there is a wide range of eigenvalues varying over many orders of magnitude. Only the smallest eigenvalues are physically interesting and they are typically clustered at the low end of the spectrum. Shift and invert is usually not an option because of fill in from the factorizations. Without a spectral transformation, many Lanczos steps are required to obtain the smallest eigenvalues. In order to recover eigenvectors, one is obliged to store all of the Lanczos basis vectors (usually on a peripheral device) and to solve very large tridiagonal eigenvalue subproblems at each step. In the Arnoldi process that is used in the non-Hermitian case, not only do the basis vectors have to be stored, but the cost of the Hessenberg eigenvalue subproblem is $\mathcal{O}(k^3)$ at the k-th step.

5.1 EXPLICIT RESTARTING

An alternative has been proposed by Saad based upon the polynomial acceleration scheme developed by Manteuffel [28] for the iterative solution of linear systems. Saad [39] proposed to restart the iteration with a vector that has been preconditioned so that it is more nearly in a k-dimensional invariant subspace of interest. This preconditioning takes the form of a polynomial applied to the starting vector that is constructed to damp unwanted components from the eigenvector expansion. The resulting algorithm takes the form:

Algorithm 5: An Explicitly Restarted Arnoldi Method

Input: (A, v)

Put $v_1 = v/\|v\|$;

For $j = 1, 2, 3, \ldots$ until *convergence*

(a5.1) Compute an m-step Arnoldi factorization

$AV_m = V_m H_m + f_m e_m^T$ with $V_m e_1 = v_1$;

(a5.2) Compute $\sigma(H_m)$ and corresponding Ritz estimates

and halt if desired eigenvalues are well approximated.

(a5.3) Construct a polynomial ψ based upon $\sigma(H_m)$ to damp unwanted components.

(a5.4) $v_1 \leftarrow \psi(A)v_1$; $v_1 \leftarrow v_1/\|v_1\|$;

End_For

The construction of the polynomial at Step (a5.3) may be guided by *a priori* information about the spectrum of A or solely by information gleaned from $\sigma(H_m)$. A typical scheme is to sort the spectrum of H_m into two disjoint sets Ω_w and Ω_u, with $\sigma(H_m) = \Omega_w \cup \Omega_u$. The Ritz values in the set Ω_w are to be regarded as approximations to the "wanted" eigenvalues of A and an open convex set $\mathcal{C}_u$ containing Ω_u with $\Omega_w \cap \mathcal{C}_u = \emptyset$ is to be regarded as a region that approximately encloses the "unwanted" portion of the spectrum of A. The polynomial ψ is then constructed to be as small in magnitude as possible on $\mathcal{C}_u$ when normalized, for example, to take the value 1 at an element of Ω_w closest to $\partial\mathcal{C}_u$. Chebyshev polynomials

are appropriate when $\mathcal{C}_u$ is taken to be an ellipse and this was the original proposal of Saad when he adapted the Manteuffel idea to eigenvalue calculations. Another possibility explored by Saad has been to take $\mathcal{C}_u$ to be the convex hull of Ω_u and to construct the polynomial ψ that best approximates 0 on this set in the least squares sense. Both of these are based upon well known theory of polynomial approximation. The problem of constructing an optimal ellipse for this problem has been studied by Chatelin and Ho. The reader is referred to [5] for details of constructing these polynomials.

The reasoning behind this type of algorithm is that that if v_1 is a linear combination of precisely k eigenvectors of A then Arnoldi factorization terminates in k steps (i.e. $f_k = 0$). The columns of V_k will form an orthonormal basis for the invariant subspace spanned by those eigenvectors, and the Ritz values $\sigma(H_k)$ will be the corresponding eigenvalues of A. The update of the starting vector v_1 is designed to enhance the components of this vector in the directions of the wanted eigenvectors and damp its components in the unwanted directions. This effect is achieved at Step (a5.4) since

$$v_1 = \sum_{j=1}^{n} x_j \gamma_j \Rightarrow \psi(A)v_1 = \sum_{j=1}^{n} x_j \psi(\lambda_j)\gamma_j.$$

If the same polynomial were applied each time, then after M iterations, the j-th original expansion coefficient would be essentially attenuated by a factor

$$\left(\frac{\psi(\lambda_j)}{\psi(\lambda_1)}\right)^M ,$$

where the eigenvalues have been ordered according decreasing values $|\psi(\lambda_j))|$. The eigenvalues inside the region $\mathcal{C}_u$ become less and less significant as the iteration proceeds. Hence, the wanted eigenvalues are approximated increasingly well as the iteration proceeds.

Another restarting strategy proposed by Saad is to replace the starting vector with a linear combination of Ritz vectors corresponding to wanted Ritz values. If the eigenvalues and corresponding vectors are re-indexed so that the first k are wanted and $(\hat{x}_j, \theta_j)$ is the the Ritz pair approximating the eigenpair (x_j, λ_j) then

$$v_1^+ \leftarrow \sum_{j=1}^{k} \hat{x}_j \gamma_j \tag{4}$$

is taken as the new starting vector. Again, the motivation here is that the Arnoldi residual f_k would vanish if these k Ritz vectors were actually eigenvectors of A and the Ritz vectors are the best available approximations to these eigenvectors. A heuristic choice for the coefficients γ_j has also been suggested by Saad [38]. It is to weight the j-th Ritz vector with the value of its Ritz estimate and then normalize so that the new starting vector has norm 1. This has the effect of favoring the Ritz vectors that have least converged. Additional aspects of explicit restarting are developed thoroughly in Chapter VII of [40]. In any case, this restarting mechanism is actually polynomial restarting in disguise. Since $\hat{x}_j \in \mathcal{K}_m(A, v_1)$ implies $\hat{x}_j = \phi_j(A)v_1$ for some polynomial ϕ_j the formula for v_1^+ in (4) is

of the form

$$v_1^+ \leftarrow \phi(A)v_1 \equiv \sum_{j=1}^{k} \gamma_j \phi_j(A)v_1. \tag{5}$$

The technique just described is referred to as explicit (polynomial) restarting. When Chebyshev polynomials are used it is called an Arnoldi-Chebyshev method. The cost in terms of matrix-vector products $w \leftarrow Av$ is $M*(m+deg(\psi))$ for M major iterations. The cost of the arithmetic in the Arnoldi factorization is $M*(2n*m^2+O(m^3))$ Flops (floating point operations). Tradeoffs must be made in terms of cost of the Arnoldi factorization vs. cost of the matrix-vector products Av and also in terms of storage $(nm+O(m^2))$.

5.2 IMPLICIT RESTARTING

There is another approach to restarting that offers a more efficient and numerically stable formulation. This approach called *implicit restarting* is a technique for combining the implicitly shifted QR mechanism with a k-step Arnoldi or Lanczos factorization to obtain a truncated form of the implicitly shifted QR-iteration. The numerical difficulties and storage problems normally associated with Arnoldi and Lanczos processes are avoided. The algorithm is capable of computing a few (k) eigenvalues with user specified features such as largest real part or largest magnitude using $2nk+O(k^2)$storage. No auxiliary storage is required. The computed Schur basis vectors for the desired k-dimensional eigen-space are numerically orthogonal to working precision. This method is well suited to the development of mathematical software and this will be discussed in Section 7.

Implicit restarting provides a means to extract interesting information from very large Krylov subspaces while avoiding the storage and numerical difficulties associated with the standard approach. It does this by continually compressing the interesting information into a fixed size k-dimensional subspace. This is accomplished through the implicitly shifted QR mechanism. An Arnoldi factorization of length $m = k+p$

$$AV_m = V_m H_m + f_m e_m^T, \tag{6}$$

is compressed to a factorization of length k that retains the eigen-information of interest. This is accomplished using QR steps to apply p shifts implicitly. The first stage of this shift process results in

$$AV_m^+ = V_m^+ H_m^+ + f_m e_m^T Q, \tag{7}$$

where $V_m^+ = V_m Q$, $H_m^+ = Q^T H_m Q$, and $Q = Q_1 Q_2 \cdots Q_p$, with Q_j the orthogonal matrix associated with the shift μ_j. It may be shown that the first $k-1$ entries of the vector $e_m^T Q$ are zero (i.e. $e_m^T Q = (\sigma e_k^T, \hat{q}^T)$). Equating the first k columns on both sides yields an updated $k-$step Arnoldi factorization

$$AV_k^+ = V_k^+ H_k^+ + f_k^+ e_k^T, \tag{8}$$

with an updated residual of the form $f_k^+ = V_{k+p}^+ e_{k+1}\hat{\beta}_k + f_{k+p}\sigma$. Using this as a starting point it is possible to apply p additional steps of the Arnoldi process to return to the original m-step form.

Each of these shift cycles results in the implicit application of a polynomial in A of degree p to the starting vector.

$$v_1 \leftarrow \psi(A)v_1 \quad \text{with} \quad \psi(\lambda) = \prod_1^p (\lambda - \mu_j).$$

The roots of this polynomial are the shifts used in the QR process and these may be selected to filter unwanted information from the starting vector and hence from the Arnoldi factorization. Full details may be found in [44]. The basic iteration is given here in Algorithm 6 and the diagrams in Figures 1-3 describe how this iteration proceeds schematically. In Algorithm 6 and in the discussion below, the notation $M_{(1:n,1:k)}$ denotes the leading $n \times k$ submatrix of M.

Algorithm 6: An Implicitly Restarted Arnoldi Method

Input: (A, V, H, f) with $AV_m = V_m H_m + f_m e_m^T$, an m-Step Arnoldi Factorization;

For $\ell = 1, 2, 3, \ldots$ until *convergence*

(a6.2) Compute $\sigma(H_m)$ and select set of p shifts $\mu_1, \mu_2, \ldots \mu_p$
based upon $\sigma(H_m)$ or perhaps other information;

(a6.3) $q^T \leftarrow e_m^T$;

(a6.4) For $j = 1, 2, \ldots, p$,
Factor $[Q_j, R_j] = \mathrm{qr}(H_m - \mu_j I)$;
$H_m \leftarrow Q_j^H H_m Q_j$; $V_m \leftarrow V_m Q_j$;
$q \leftarrow q^H Q_j$;
End_For

(a6.5) $f_k \leftarrow v_{k+1}\hat{\beta}_k + f_m \sigma_k$; $V_k \leftarrow V_{m(1:n,1:k)}$; $H_k \leftarrow H_{m(1:k,1:k)}$;

(a6.6) Beginning with the k-step Arnoldi factorization
$AV_k = V_k H_k + f_k e_k^T$,
apply p additional steps of the Arnoldi process
to obtain a new m-step Arnoldi factorization
$AV_m = V_m H_m + f_m e_m^T$.

End_For

The diagrams in Figures 1-3 indicate how this iteration proceeds schematically.

Observe that if $m = n$ then $f = 0$ and this iteration is precisely the same as the Implicitly Shifted QR iteration. Even for $m < n$, the first k columns of V and the Hessenberg submatrix $H_{(1:k,1:k)}$ are mathematically equivalent to the matrices that would appear in the full Implicitly Shifted QR iteration using the same shifts μ_j. In this sense, the Implicitly Restarted Arnoldi method may be viewed as a truncation of the Implicitly Shifted QR iteration. The fundamental difference is that the standard Implicitly Shifted QR iteration selects shifts to drive subdiagonal elements of H to zero from the bottom up while the shift selection in the Implicitly Restarted Arnoldi method is made to drive subdiagonal

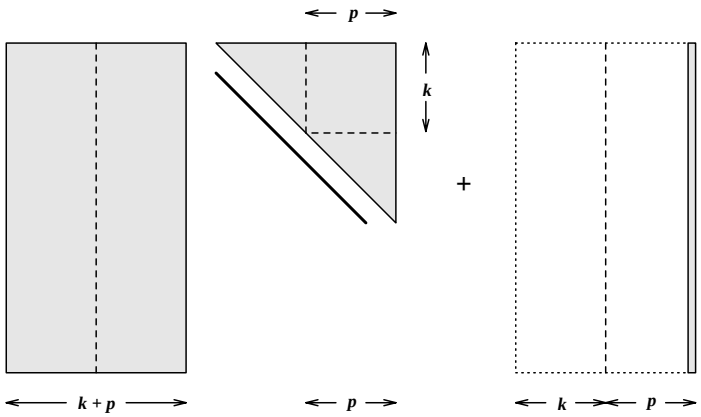


Figure 1: Representation of $V_{k+p}H_{k+p} + f_{k+p}e_{k+p}^T$. Shaded regions denote nonzeros.

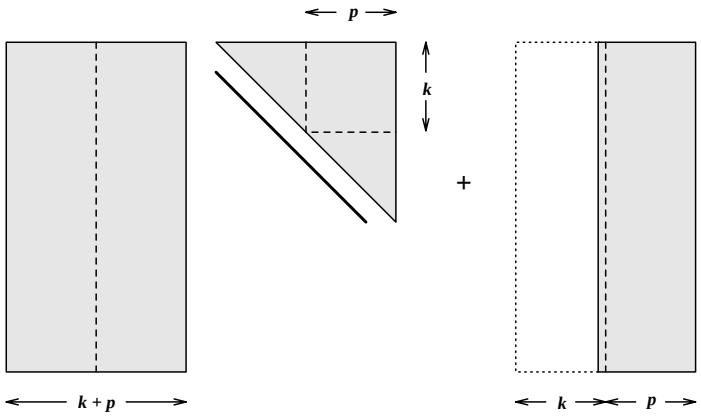


Figure 2: $V_{k+p}QQ^TH_{k+p}Q + f_{k+p}e_{k+p}^TQ$ after p implicitly shifted QR steps.

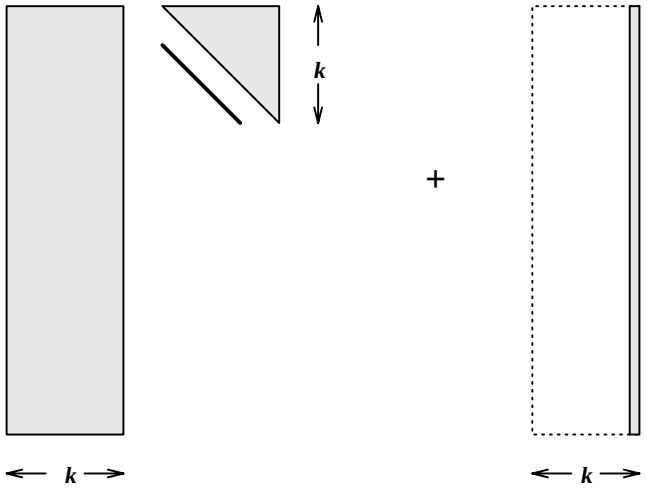


Figure 3: Leading k columns $V_kH_k + f_ke_k^T$ form a length k Arnoldi factorization after discarding the last p columns.

elements of H to zero from the top down. Important implementation details concerning the deflation (setting to zero) of subdiagonal elements of H and the purging of unwanted but converged Ritz values are beyond the scope of this discussion. However, these details are extremely important to the success of this iteration in difficult cases. Complete details of these numerical refinements may be found in [26, 24]

The above iteration can be used to apply any known polynomial restart. If the roots of the polynomial are not known there is an alternative implementation that only requires one to compute $q_1 = \psi(H)e_1$ where ψ is the desired degree p polynomial. A sequence of Householder transformations may developed to form a unitary matrix Q such that $Qe_1 = q_1$ and $H \leftarrow Q^H H Q$ is upper Hessenberg. The details which follow standard developments for the Implicitly Shifted QR iteration will be omitted here.

A shift selection strategy that has proved successful in practice is called the "Exact Shift Strategy". In this strategy, one computes $\sigma(H)$ and sorts this into two disjoint sets Ω_w and Ω_u. The k Ritz values in the set Ω_w are regarded as approximations to the "wanted" eigenvalues of A , and the p Ritz values in the set Ω_u are taken as the shifts μ_j. An interesting consequence (in exact arithmetic) is that after Step (a6.4) above, the spectrum of H_k in Step (a6.5) is $\sigma(H_k) = \Omega_w$ and the updated starting vector v_1 is a particular linear combination of the k Ritz vectors associated with these Ritz values. In other words, the implicit restarting scheme with exact shifts provides a specific selection of the coefficients γ_j in the formula (4) and this implicit scheme costs p rather than the $k+p$ matrix-vector products the explicit scheme would require. Thus the exact shift strategy can be viewed both as a means to damp unwanted components from the starting vector and also as directly forcing the starting vector to be a linear combination of wanted eigenvectors. The exact shift strategy has two additional interesting theoretical properties.

Lemma 5.1 *If H is unreduced and diagonalizable then:*

1. *The polynomial ϕ in (5) satisfies $\phi(\lambda) = \psi(\lambda)\rho(\lambda)$, where ψ is the exact shift polynomial and ρ is some polynomial of degree at most $k-1$.*

2. *The updated Krylov subspace generated by the new starting vector satisfies*
   $\mathcal{K}_m(A, v_1^+) = Span\{\hat{x}_1, \hat{x}_2, \cdots, \hat{x}_k, A\hat{x}_j, A^2\hat{x}_j, \cdots, A^p\hat{x}_j\}$
   for $j = 1, 2, \cdots, k$.

The first property $\phi(\lambda) = \psi(\lambda)\rho(\lambda)$ indicates that the linear combination selected by the exact shift scheme is somehow minimal while the second property indicates that each of the subspaces $\mathcal{K}_p(A, \hat{x}_j) \subset \mathcal{K}_m(A, v_1^+)$ so that each sequence of "wanted" Ritz vectors is represented equally in the updated subspace. The first property was established in [24] along with an extensive analysis of the numerical properties of implicit restarting. The surprising second property was established by Morgan in [30] along with some compelling numerical results indicating superior performance of implicit over explicit restarting.

6 The Generalized Eigenvalue Problem

A typical source of large scale eigenproblems is through a discrete form of a continuous problem. The resulting finite dimensional problems become large due to accuracy requirements and spatial dimensionality. Typically this takes the form

$$\begin{aligned} \mathcal{L}u &= u\lambda \text{ in } \Omega, \\ u &\text{ satisfies } \mathcal{B} \text{ on } \partial\Omega, \end{aligned} \tag{9}$$

where $\mathcal{L}$ is some linear differential operator. A number of techniques may be used to discretize $\mathcal{L}$. The finite element method provides an elegant discretization. If $\mathcal{W}$ is a space of functions in which the solution to (9) may be found and $\mathcal{W}_n \subset \mathcal{W}$ is an n-dimensional subspace with basis functions $\{\phi_j\}$ then an approximate solution u_n is expanded in the form

$$u_n = \sum_{j=1}^{n} \phi_j \xi_j.$$

A variational or a Galerkin principle is applied depending on whether or not $\mathcal{L}$ is self-adjoint, leading to a weak form of (9)

$$\mathcal{A}(v,u) = \lambda < v, u >, \tag{10}$$

where $\mathcal{A}(v,u)$ is a bilinear form. Substituting the expanded form of $u = u_n$ and requiring (10) to hold for each trial function $v = \phi_i$ gives a set of algebraic equations

$$\mathcal{A}(\phi_i, \sum_{j=1}^{n} \phi_j \xi_j) = \lambda < \phi_i, \sum_{j=1}^{n} \phi_j \xi_j >,$$

where $< \cdot, \cdot >$ is an inner product in $\mathcal{W}_n$. This leads to the following systems of equations

$$\sum_{j=1}^{n} \mathcal{A}(\phi_i, \phi_j)\xi_j = \lambda \sum_{j=1}^{n} < \phi_i, \phi_j > \xi_j, \tag{11}$$

for $1 \le i \le n$. We may rewrite (11) and obtain the matrix equation

$$Ax = \lambda M x,$$

where

$$A_{i,j} = \mathcal{A}(\phi_i, \phi_j), \quad M_{i,j} =< \phi_i, \phi_j >, \quad x^T = [\xi_1, \ldots, \xi_n]^T,$$

for $1 \le i, j \le n$. Typically the basis functions are chosen so that few entries in a row of A or M are nonzero. In structures problems A is called the "stiffness" matrix and M is called the "mass" matrix. In chemistry and physics M is often referred to as the "overlap" matrix. A nice feature of this approach to discretization is that boundary conditions are naturally incorporated into the discrete problem. Moreover, in the self-adjoint case, the Rayleigh principle is preserved from the continuous to the discrete problem. In particular, since Ritz values are Rayleigh quotients, this assures the smallest Ritz value is greater than the smallest eigenvalue of the original problem.

Thus, it is natural for large scale eigenproblems to arise as generalized rather than standard problems. If $\mathcal{L}$ is self-adjoint the discrete problems are symmetric or Hermitian and if not the matrix A is nonsymmetric but the matrix M is symmetric and at least positive

semi-definite. There are a number of ways to convert the generalized problem to standard form. There is always motivation to preserve symmetry when it is present.

If M is positive definite then factor $M = LL^T$ and the eigenvalues of $\hat{A} \equiv L^{-1}AL^{-T}$ are the eigenvalues of (A, M) and the eigenvectors are obtained by solving $L^T x = \hat{x}$ where $\hat{x}$ is an eigenvector of $\hat{A}$. This standard transformation is fine if one wants the eigenvalues of largest magnitude and it preserves symmetry if A is symmetric. However, when M is ill-conditioned this can be a dangerous transformation leading to numerical difficulties. Since a matrix factorization will have to be done anyway, one may as well formulate a spectral transformation.

6.1 STRUCTURE OF THE SPECTRAL TRANSFORMATION

A convenient way to provide a spectral transformation is to note that

$$Ax = \lambda Mx \iff (A - \mu M)x = (\lambda - \mu)Mx$$

Thus

$$(A - \mu M)^{-1}Mx = x\theta, \quad \text{where} \quad \theta = \frac{1}{\lambda - \mu}.$$

If A is symmetric then one can maintain symmetry in the Arnoldi/Lanczos process by taking the inner product to be

$$< x, y >= x^T My.$$

It is easy to verify that the operator $(A - \mu M)^{-1}M$ is symmetric with respect to this inner product if A is symmetric. In the Arnoldi/Lanczos process the matrix-vector product $w \leftarrow Av$ is replaced by $w \leftarrow (A - \mu M)^{-1}Mv$ and the step $h \leftarrow V^T f$ is replaced by $h \leftarrow V^T(Mf)$. If A is symmetric then the matrix H is symmetric and tridiagonal. Moreover, this process is well defined even when M is singular and this can have important consequences even if A is nonsymmetric. We shall refer to this process as the M-Arnoldi process.

If M is singular then the operator $S \equiv (A - \mu M)^{-1}M$ has a non-trivial null space and the bilinear function $< x, y >= x^T My$ is a semi-inner product and $\|x\|_M \equiv < x, y >^{1/2}$ is a semi-norm. Since $(A - \mu M)$ is assumed to be nonsingular, $\mathcal{N} \equiv Null(S) = Null(M)$. Vectors in $\mathcal{N}$ are generalized eigenvectors corresponding to *infinite* eigenvalues. Typically, one is only interested in the finite eigenvalues of (A, M) and these will correspond to the non-zero eigenvalues of S. The invariant subspace corresponding to these non-zero eigenvalues is easily corrupted by components of vectors from $\mathcal{N}$ during the Arnoldi process. However, using the M-Arnoldi process with some refinements can provide a solution.

In order to better understand the situation, it is convenient to note that since M is positive semi-definite, there is an orthogonal matrix Q such that

$$M = Q \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix} Q^T$$

where D is a positive definite diagonal matrix of order n, say. Thus

$$\hat{S} \equiv Q^T S Q = \begin{bmatrix} S_1 & 0 \\ S_2 & 0 \end{bmatrix},$$

where S_1 is a square matrix of order n and S_2 is an $m \times n$ matrix with the original A, M being of order $m+n$. Observe now that a non-zero eigenvalue λ of $\hat{S}$ satisfies $\hat{S}x = x\lambda$, i.e.

$$\begin{bmatrix} S_1 x_1 \\ S_2 x_1 \end{bmatrix} = \begin{bmatrix} x_1 \lambda \\ x_2 \lambda \end{bmatrix}$$

so that $x_2 = \frac{1}{\lambda} S_2 x_1$ must hold. Note also that for any eigenvector $x^H = (x_1^H, x_2^H)$, the leading vector x_1 must be an eigenvector of S_1. Since $\hat{S}$ is block triangular, $\sigma(\hat{S}) = \sigma(S_1) \cup \sigma(0_m)$. Assuming S_2 has full rank, it follows that if S_1 has a zero eigenvalue then there is no corresponding eigenvector (since $S_2 x_1 = 0$ would be implied). Thus if zero is an eigenvalue of S_1 with algebraic multiplicity m_o then zero is an eigenvalue of $\hat{S}$ of algebraic multiplicity $m + m_o$ and with geometric multiplicity m. Of course, since, S is similar to $\hat{S}$ all of these statements hold for S as well.

6.2 EIGENVECTOR/NULL-SPACE PURIFICATION

With these observations in hand, it is possible to see the virtue of using M-Arnoldi on S. After k-steps of M-Arnoldi,

$$SV = VH + fe_k^T \quad \text{with} \quad V^T M V = I_k, V^T M f = 0.$$

Introducing the similarity transformation Q gives

$$\hat{S}\hat{V} = \hat{V}H + \hat{f}e_k^T \quad \text{with} \quad \hat{V}^T Q^T M Q \hat{V} = I_k, V^T Q^T M Q \hat{f} = 0,$$

where $\hat{V} = Q^T V$ and $\hat{f} = Q^T f$. Partitioning $\hat{V}^T = (V_1^T V_2^T)$ and $\hat{f}^T = (f_1^T, f_2^T)$ consistent with the blocking of $\hat{S}$ gives

$$S_1 V_1 = V_1 H + f_1 e_k^T \quad \text{with} \quad V_1^T D V_1 = I_k, V_1^T D f_1 = 0.$$

Moreover, the side condition $S_2 V_1 = V_2 H + f_2 e_k^T$ holds, so that in exact arithmetic a zero eigenvalue should not appear as a converged Ritz value of H. This argument shows that M-Arnoldi on S is at the same time doing D-Arnoldi on S_1 while avoiding convergence to zero eigenvalues.

Round-off error due to finite precision arithmetic will cloud the situation, as usual. It is clear that the goal is to prevent components in $\mathcal{N}$ from corrupting the vectors V Thus to begin, the starting vector v_1 should be of the form $v_1 = Sv$. If a final approximate eigenvector x has components in $\mathcal{N}$ they may be purged by replacing $x \leftarrow Sx$ and then normalizing. To see the effect of this, note that if $x = Q \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

$$Sx = Q \begin{bmatrix} S_1 x_1 \\ S_2 x_1 \end{bmatrix}$$

and all components in $\mathcal{N}$ which are of the form $Q \begin{bmatrix} 0 \\ p \end{bmatrix}$ will have been purged. This final application of S may be done implicitly in two ways. One is to note that if $x = Vy$ with $Hy = y\theta$ then $Sx = VHy + fe_k^T y = x\theta + fe_k^T y$ and this is the correction suggested by [32]. Another recent suggestion due to Meerbergen and Spence is to use implicit restarting with a zero shift [29]. Recall that implicit restarting with ℓ zero shifts is equivalent to starting the M-Arnoldi process with a starting vector of $S^\ell v_1$ and all the resulting Ritz vectors will be multiplied by S^ℓ as well. After applying the implicit shifts to H, the leading submatrix of order $k - \ell$ will provide the updated Ritz values. No additional explicit matrix-vector products with S are required.

The ability to apply ℓ zero shifts (i.e. to multiply by S^ℓ implicitly) is very important when S_1 has zero eigenvalues. If $S_1 x_1 = 0$ then

$$\begin{bmatrix} S_1 & 0 \\ S_2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ S_2 x_1 \end{bmatrix} \in \mathcal{N}.$$

Thus to completely eradicate components from $\mathcal{N}$ one must multiply by S^ℓ where ℓ is equal to the dimension of the largest Jordan block corresponding to a zero eigenvalue of S_1.

Spectral transformations were studied extensively by Ericsson and Ruhe [14] and the first eigenvector purification strategy was developed in [32]. Shift and invert techniques play an essential role in the block Lanczos code developed by Grimes, Lewis, and Simon. The many nuances of this technique in practical applications is discussed thoroughly in [19]. The development presented here and the eigenvector purification through implicit restarting is due to Meerbergen and Spence [29].

6.3 AN EXAMPLE

This discussion is illustrated with the following example.

$$A = \begin{bmatrix} K & C \\ C^T & 0 \end{bmatrix} \quad \text{and} \quad M = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix},$$

with A an order 225 matrix approximation to a convection-diffusion operator and C a structured random matrix. This example was chosen because it has the block structure of a typical steady-state Navier-Stokes linear stability analysis (see [29]). The following MATLAB code was used to generate the example:

```
rand('seed',0);
n = 225;m=100;
K = lapc(n,100);
C = [rand(m,m) ; zeros(n-m,m)];
M = [eye(n) zeros(n,m) ; zeros(m,n) zeros(m,m)];
A = [K C ; C' zeros(m,m)];
mu = 7.0;
S = (A - mu*M)\M;
```

$\|P_{\mathcal{N}}V\|$	$\|P_{\mathcal{N}}V^+\|$	$\|P_{\mathcal{G}}V\|$	$\|P_{\mathcal{G}}V^+\|$
3.70	1.48(-11)	1.32(-11)	2.85(-12)

Table 1: Projection of V onto $\mathcal{N}$ and $\mathcal{G}$

j	$\|Ax_j - Mx_j\lambda_j\|$	$\|(Ax_j - Mx_j\lambda_j)^+\|$
1	1.50(-03)	9.93(-06)
2	1.11(-02)	6.77(-05)

Table 2: Residuals before and after purging components from $\mathcal{N}$ and $\mathcal{G}$

The matrices `K, C, M, A` correspond to the matrices in the equations above. The funcion `lapc` computes a finite difference approximation to $\Delta u + \rho u_x$ on a 15×15 regular grid in the unit square with $\rho = 100$. Any matrix pencil (A, M) with this block structure (assuming C full rank and $A - \mu * M$ nonsingular) will produce an S of the form

$$S = \begin{bmatrix} 0 & 0 & 0 \\ 0 & S_{22} & 0 \\ S_{31} & S_{32} & 0 \end{bmatrix},$$

with S_{22} nonsingular and order $n - m$. From the above discussion one may conclude that S has an eigenvalue 0 with algebraic multiplicity $2m$ and geometric multiplicity m. There are three important subspaces associated with S. They are $\mathcal{N}$, $\mathcal{G}$ and $\mathcal{R}$ and these spaces satisfy

$$S\mathcal{N} = \{0\} \quad , \quad S\mathcal{G} \subset \mathcal{N} \quad , \quad S\mathcal{R} \subset \mathcal{R}.$$

All of $\mathbf{C}^n$ may be represented as a direct sum of these three spaces. The (oblique) projectors associated with these spaces shall be denoted by $P_{\mathcal{N}}$, $P_{\mathcal{G}}$, and $P_{\mathcal{R}}$ respectively. Explicit formulas are:

$$P_{\mathcal{N}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -S_{32}S_{22}^{-1} & I \end{bmatrix} \quad P_{\mathcal{G}} = \begin{bmatrix} I & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad P_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & S_{22} & \\ S_{31} & S_{32} & 0 \end{bmatrix}$$

The following table shows the norms of the projections of the basis vectors V onto the spaces $\mathcal{N}$ and $\mathcal{G}$ where V was computed with 20 steps of M-Arnoldi starting with a vector $v_1 = Sv$ (v a vector with all entries equal to 1) . The norms of the projections are taken before and after purging by applying two zero shifts using implicit restarting. The + symbol denotes the updated basis after purging.

The next table shows the residual norms for the two approximate eigenvalues that are closest to the shift μ before and after purging.

Clearly, there is considerable merit to doing this purging. This generalizes the purging proposed by [32] and seems to be quite promising. Further testing is needed but some form of this process is essential to the construction of numerical software to implement shift-invert strategies.

7 SOFTWARE, PERFORMANCE, and PARALLEL COMPUTATION

The Implicitly Restarted Arnoldi Method has been implemented and a package of Fortran 77 subroutines has been developed. This software, called ARPACK [27], provides several features which are not present in other codes based upon a single-vector Arnoldi process. One of the most important features from the software standpoint is the reverse communication interface. This feature provides a convenient way to interface with application codes without imposing a structure on the users matrix or the way a matrix-vector product is accomplished. In the parallel setting, this reverse communication interface enables efficient memory and communication management for massively parallel MIMD and SIMD machines. The important features of ARPACK are:

- A reverse communication interface.
- Ability to return k eigenvalues which satisfy a user specified criterion such as largest real part, largest absolute value, largest algebraic value (symmetric case), etc.
- A fixed pre-determined storage requirement suffices throughout the computation. Usually this is $n*O(2k)+O(k^2)$ where k is the number of eigenvalues to be computed and n is the order of the matrix. No auxiliary storage or interaction with such devices is required during the course of the computation.
- Eigenvectors may be computed on request. The Arnoldi basis of dimension k is always computed. The Arnoldi basis consists of vectors which are numerically orthogonal to working accuracy. Computed eigenvectors of symmetric matrices are also numerically orthogonal.
- The numerical accuracy of the computed eigenvalues and vectors is user specified. Residual tolerances may be set to the level of working precision. At working precision, the accuracy of the computed eigenvalues and vectors is consistent with the accuracy expected of a dense method such as the implicitly shifted QR iteration.
- Multiple eigenvalues offer no theoretical or computational difficulty other than additional matrix-vector products required to expose the multiple instances. This is made possible through the implementation of deflation techniques similar to those employed to make the implicitly shifted QR-algorithm robust and practical. A block method is not required and hence one does not need to "guess" the correct blocksize that would be needed to capture multiple eigenvalues.

7.1 REVERSE COMMUNICATION INTERFACE

As mentioned above, the reverse communication interface is one of the most important aspects of the design of ARPACK. In the serial code, a typical usage of this interface is illustrated with the following example:

```
10  continue
    call snaupd (ido, bmat, n, which,...,V,...,lworkl, info)
```

```
if (ido .eq. newprod) then
   call matvec ('A', n, workd(ipntr(1)), workd(ipntr(2)))
else
   return
endif
go to 10
```

As usual, with reverse communication, control is returned to the calling program when interaction with the matrix A is required. The action requested of the calling program is to simply perform the action indicated by the reverse communication parameter `ido` (in this case multiply the vector held in the array `workd` beginning at location `ipntr(1)` and put the result in the array `workd` beginning at location `ipntr(2)`). Note that call to the subroutine `matvec` in this code segment is simply meant to indicate that this matrix-vector operation is taking place. The user is free to use any available mechanism or subroutine to accomplish this task. In particular, no specific data structure is imposed and indeed, no explicit representation of the matrix is even required. One only needs to supply the action of the matrix on the specified vector.

There are several reasons for supplying this interface. It is more convenient to use with large application codes. The alternative is to put the user supplied matrix-vector product in a subroutine with a pre-specified calling sequence. This may be quite cumbersome and is especially so in those cases where the action of the matrix on a vector is known only through a lengthy computation that doesn't involve the matrix A explicitly. Typically, if the matrix-vector product must be provided in the form of a subroutine with a fixed calling sequence, then named `common` or some other means must be used to pass data to the routine. This is incompatible with efficient memory management for massively parallel MIMD and SIMD machines.

This has been implemented on a number of parallel machines including the CRAY-C90, Thinking Machines CM-200 and CM-5, Intel Delta, and CRAY T3D. Parallel performance on the C90 is obtained through the BLAS operations without any modification to the serial code. SIMD performance on the CM-200 is also relatively straightforward. All of the BLAS operations were expressed using Fortran 90 array constructs and hence were automatically compiled for execution on the SIMD array instead of the frontend. Operations on the projected matrix H were not encoded with these array constructs and hence were automatically scheduled for the frontend. The only additional complication was to define the data layouts of the V array and the work arrays for efficient execution. In the distributed memory implementations, the reverse communication interface provided a natural way to parallelize the ARPACK codes internally without imposing a fixed parallel decomposition on the user supplied matrix-vector product.

7.2 DATA DISTRIBUTION AND GLOBAL OPERATIONS

The parallelization strategy for distributed memory machines consists of providing the user with an Single Program Multiple Data (SPMD) template. The array V is blocked and distributed across the processors. The projected matrix H is replicated. The SPMD

program looks essentially like the serial code except that the local block `Vloc` is passed in place of V. The work space is partitioned consistently with the partition of V and each section of the work space is distributed to the node processors. Thus the SPMD parallel code looks very similar to that of the serial code. Assuming a parallel version of the subroutine matvec, an example of the application of the distributed interface is illustrated as the follows:

```
10  continue
    call snaupd (ido, bmat, nloc, which, ...,
   *                      Vloc , ... lworkl, info)
    if (ido .eq. newprod) then
       call matvec ('A', nloc,workd(ipntr(1)), workd(ipntr(2)))
    else
       return
    endif
    go to 10
```

Where, `nloc` is the number of rows in the block `Vloc` of V that has been assigned to this node process.

Typically, the blocking of V is commensurate with the parallel decomposition of the matrix A as well as with the configuration of the distributed memory and interconnection network. Logically, the V matrix be partitioned by blocks

$$V^T = ({V^{(1)}}^T, {V^{(2)}}^T, ..., {V^{(nproc)}}^T)$$

with one block per processor and with H replicated on each processor.

The explicit steps of the process responsible for the j block are:

1. $\beta_k = gnorm(f_k^{(*)})$; $v_{k+1}^{(j)} \leftarrow f_k^{(j)}/\beta$;
2. $V_{k+1}^{(j)} \leftarrow (V_k, v_{k+1})^{(j)}$; $\hat{H}_k \leftarrow \begin{pmatrix} H_k \\ \beta_k e_k^T \end{pmatrix}$.
3. $z \leftarrow (Aloc)v_{k+1}$;
4. $h^{(j)} \leftarrow {V_k^{(j)}}^T z$; $h \leftarrow gsum(h^{(*)})$ $f_{k+1} \leftarrow z - V_{k+1}h$;
5. $H_{k+1} \leftarrow (\hat{H}_k, h)$;

Note that the function $gnorm$ at Step 1 is meant to represent the global reduction operation of computing the norm of the distributed vector f_k from the norms of the local segments $f_k^{(j)}$ and the function $gsum$ at Step 4 is meant to represent the global sum of the local vectors $h^{(j)}$ so that the quantity $h = \sum_{j=1}^{nproc} h^{(j)}$ is available to each process on completion. These are the only two communication points within this algorithm. The remainder is perfectly parallel. Additional communication will typically occur at Step 3. Here the operation $(Aloc)v$ is meant to indicate that the user supplied matrix-vector product is able to compute the local segment of the matrix-vector product Av that is consistent with the

partition of V. Ideally, this would only involve nearest neighbor communication among the processes.

Since H is replicated on each processor, the parallelization of the implicit restart mechanism described by Algorithm(6) remains untouched. The only difference is that the local block $V^{(j)}$ is in place of the full matrix V. All operations on the matrix H are replicated on each processor. Thus there is no communication overhead but there is a "serial bottleneck" here due to the redundant work. If k is small relative to n this bottleneck is insignificant. However, it becomes a very important latency issue as k grows and will prevent scalabilty if k grows with n as the problem size increases.

The main benefit of this approach is that the changes to the serial version of ARPACK are very minimal. Since the change of dimension from matrix order n to its local distributed blocksize `nloc` is invoked through the calling sequence of the subroutine `snaupd`, there is no essential change to the code. Only six routines were effected in a minimal way. These routines either required a change in norm calculation for distributed vectors (Step 1) or for the distributed dense matrix-vector product (Step 4). Since the vectors are distributed, norms had to be done via partial (scaled) dot products for the local vector segments and then a global sum operation was used to complete the sum of the squared norms of these segments on all processors. More specifically, the commands are changed from

```
   rnorm = sdot (n, resid, 1, workd, 1)
   rnorm = sqrt(abs(rnorm))
```

to

```
   rnorm0 = sdot (n, resid, 1, workd, 1)
   call gssum(rnorm0,1,tmp)
   rnorm0 = sqrt(abs(rnorm0))
   rnorm  = rnorm0
```

Similarly, the computation of the matrix-vector product operation $h \leftarrow V^T w$ requires a change from

```
   call sgemv ('T', n, j, one, v, ldv, workd(ipj), 1,
  *            zero, h(1,j), 1)
```

to

```
   call sgemv ('T', n, j, one, v, ldv, workd(ipj), 1,
  *            zero, h(1,j), 1)
   call gssum(h(1,j),j,h(1,j+1))
```

so the global sum operation `gssum` was sufficient to implement all of the global operations.

7.3 DISTRIBUTED MEMORY PARALLEL PERFORMANCE

To get an idea of the potential performance of ARPACK on distributed memory machines some examples have been run on the Intel Touchstone DELTA. The examples involved have been designed to test the performance of the software, the matrix structure and the

Touchstone DELTA machine architecture, and the speedup behavior of the software on DELTA.

The user's implementation of the matrix-vector product $w \leftarrow Av$ can have considerable effect upon the parallel performance. Moreover, there is a fundamental difficulty in testing how the performance scales as the problem size increases. The difficulty is that the problem often becomes increasingly difficult to solve as the size increases due to clustering of eigenvalues. The tests reported here attempt to isolate and measure the performance of the parallelization of the ARPACK routines independently of the matrix-vector product.

In order to isolate the performance of the ARPACK routines from the performance of the user's matrix-vector product and also to isolate effects of a changing problem characteristics as the size increases, a test was comprised of replicating the same matrix repeatedly to obtain a block diagonal matrix. Each diagonal block corresponded to a corresponding block of the partitioned and distributed matrix V. This is, of course, a completely contrived situation that allows the workload to increase linearly with the number of processors. Since the each diagonal block of the matrix is identical the algorithm should behave as if $nproc$ identical problems are being solved simultaneously as long as the initial distributed segments of v_1 are generated the same. Thus, the only things that could prevent ideal speedup are the communication involved in the global operations and the "serial bottleneck" associated with the replicated operations on the projected matrix H. If neither of these were present then one would expect the execution time to remain constant as the problem size and the number of processors increase.

In this first example, each diagonal block is of order 3,000 which is identical to the vector segment size on each node. The matrix-vector product operation $z^{(j)} \leftarrow (Aloc)v_{k+1}^{(j)}$ is executed locally on each node processor upon the distributed vector segments $v_{k+1}^{(j)}$, and there is no communication among processors involved in this operation. As described above, the problem size in increased linearly with the the number of processors by adjoining an additional identical diagonal block to the A matrix for each additional processor. The global sum operation `gssum` is essentially a ring algorithm and thus has a linear dependence with respect to the number of nodes. Since the diagonal blocks are identical, the replicated operations on H should remain the same as the problem size increases and hence linear speed up is expected, i.e. as the problem size increases the execution time should remain constant. This ideal speedup is very nearly achieved as is clearly reflected in Table 3.

The second example is obtained from the similar numerical model of the eigenproblem of the Laplacian operator defined on the unit square with square with Dirichlet boundary conditions on three sides and a Neuman boundary condition on the fourth side. This leads to a mildly nonsymmetric matrix with the same 5-diagonal structure as the standard 2-D discrete Laplacian on a 5 point stencil. The unit square $\{(x, y)|0 \leq x, y \leq 1\}$ was discretized with x-direction mesh size and y-direction mesh size $1/(n+1)$ and $1/(m+1)$, respectively. Thus the matrix A is block tridiagonal and of order $N = nm$. The order of each diagonal block is n, and the number of diagonal blocks is m.

A natural way to carry out the matrix-vector product operation $w \leftarrow Av$ is described as the follows. A standard domain decomposition partitioning of the unit square into sub-

Problem size	Number of nodes	Total Time (s)
3000*1	1	22.96
3000*2	2	23.22
3000*4	4	23.98
3000*8	8	24.08
3000*16	16	24.39
3000*32	32	24.95
3000*64	64	25.50
3000*128	128	27.13
3000*256	256	28.65

Table 3: Parallel ARPACK test on DELTA, matrix order 3,000 on each node

Problem size	Number of nodes	Total Time (s)
2500*1	1	19.63
2500*2	2	20.71
2500*4	4	21.97
2500*8	8	22.47
2500*16	16	22.50
2500*32	32	23.13
2500*64	64	23.68
2500*128	128	24.78
2500*256	256	28.16

Table 4: Parallel ARPACK test on DELTA, matrix order 2,500 on each node

rectangles leads to a parallel matrix-vector product that only exchanges boundary data across the boundaries of the sub-domains and hence only needs nearest neighbor connections. The subdomains are naturally chosen so that the blocking of the matrix is commensurate with the blocking and distribution of the V array. The reverse communication interface allows the user supplied matrix-vector product to take advantage of the matrix structure. Simple send and receive operations using the native Intel *isend* and *irecv* were used to carry out the nearest neighbor communication operation.

The results of these tests are given in Table 4 and demonstrate nearly the same speedup as to Table 3. The relatively minor communication to receive boundary data from nearest neighbors effected the speedup properties somewhat.

The final example shows how dramatically an inefficient matrix-vector product operation $w \leftarrow Av$ and also how problem size can effect performance. A naive way to perform the matrix-vector product would be to collect the segments of the vector v from all nodes before the operation, and then distribute the segments of the result vector w to each node after the operation. The performance of this scheme is shown in Table 5. No advantage of the matrix structure was taken in computing the matrix-vector product. The matrix size was fixed at

Nodes	Time (s)	Iters.	Ave. $\frac{Time}{Iter}$	$\frac{OP*x\ Time}{Total\ Time}$
1	1809.07	173	10.46	0.84 %
2	1073.36	189	5.679	1.48 %
4	732.72	213	3.440	2.65 %
8	449.95	225	2.000	5.24 %
16	201.27	192	1.048	8.90 %
32	114.98	154	0.747	13.3 %
64	161.24	260	0.620	18.0 %
128	128.28	210	0.611	25.9 %

Table 5: Parallel ARPACK test run, matrix order 3,200

n = 3,200. The parallel ARPACK software was then used to compute the eigenvalues and eigenvectors. A residual tolerance of (10^{-8}) was imposed.

Table 5 shows the total time and the number of iterations required to solve this fixed problem with a different number of processors. The number of iterations varied with different processor configurations and this was attributed to different initial random vectors being generated as the number of processors changed. However, the corresponding result eigenvalues and eigenvectors are identical for all of the runs.

The speedup caused by increasing the number of processors can be observed by checking the average run time per iterate for each individual test. The third column in Table 5, demonstrates deteriorated speedup after the number of processors exceeds 32. Column four shows that the reason for this deterioration lies with the inefficient matrix-vector product.

7.4 GENERAL APPLICATIONS OF ARPACK

ARPACK has been used in a variety of challenging applications, and has proven to be useful both in symmetric and nonsymmetric problems. It is of particular interest when there is no opportunity to factor the matrix and employ a "shift and invert" form of spectral transformation,

$$\hat{A} \leftarrow (A - \sigma I)^{-1} \ . \tag{12}$$

Existing codes often rely upon this transformation to enhance convergence. Extreme eigenvalues $\{\mu\}$ of the matrix $\hat{A}$ are found very rapidly with the Arnoldi/Lanczos process and the corresponding eigenvalues $\{\lambda\}$ of the original matrix A are recovered from the relation $\lambda = 1/\mu + \sigma$. Implementation of this transformation generally requires a matrix factorization. In many important applications this is not possible due to storage requirements and computational costs. The implicit restarting technique used in ARPACK is often successful without this spectral transformation.

One of the most important classes of application arise in computational fluid dynamics. Here the matrices are obtained through discretization of the Navier-Stokes equations. A typical application involves linear stability analysis of steady state solutions. Here one lin-

earizes the nonlinear equation about a steady state and studies the stability of this state through the examination of the spectrum. Usually this amounts to determining if the eigenvalues of the discrete operator lie in the left halfplane. Typically these are parametrically dependent problems and the analysis consists of determining phenomena such as simple bifurcation, Hopf bifurcation (an imaginary complex pair of eigenvalues cross the imaginary axis), turbulence, and vortex shedding as this parameter is varied. ARPACK is well suited to this setting as it is able to track a specified set of eigenvalues while they vary as functions of the parameter. Our software has been used to find the leading eigenvalues in a Couette-Taylor wavy vortex instability problem involving matrices of order 4000. One interesting facet of this application is that the matrices are not available explicitly and are logically dense. The particular discretization provides efficient matrix-vector products through Fourier transform. Details may be found in [13].

Very large symmetric generalized eigenproblems arise in structural analysis. One example that we have worked with at Cray Research through the courtesy of Ford Motor Company involves an automobile engine model constructed from 3D solid elements. Here the interest is in a set of modes to allow solution of a forced frequency response problem $(K - \lambda M)x = f(t)$, where $f(t)$ is a cyclic forcing function which is used to simulate expanding gas loads in the engine cylinder as well as bearing loads from the piston connecting rods. This model has over 250,000 degrees of freedom. The smallest eigenvalues are of interest and the ARPACK code appears to be very competitive with the best commercially available codes on problems of this size. For details see [45].

The Singular Value Decomposition (SVD) may also be computed using ARPACK and the SVD has a many large scale applications. Two SVD applications occur in computational biology. The first of these is the 3-D image reconstruction of biological macromolecules from 2-D projections obtained through electron micrographs. The second is an application to molecular dynamical simulation of the motions of proteins. The SVD may be used to compress the data required to represent the simulation and more importantly to provide an analytical tool to help in understanding the function of the protean. See [35] for further details of the molecular dynamics application. The underlying algorithm for reconstructing 3-D image reconstruction of biological macromolecules from 2-D projections [48] is based upon the statistical technique of principal component analysis [49]. In this algorithm, a singular value decomposition (SVD) of the data set is performed to extract the largest singular vectors which are then used in a classification procedure. Our initial effort has been to replace the existing algorithm for computing the SVD with ARPACK which has increased the speed of the analysis by a factor of 7 on an Iris workstation. The accuracy of the results were also increased dramatically. Details are reported in [15].

Computational chemistry provides a rich source of problems. ARPACK is being used in two applications currently and holds promise for a variety of challenging problems in this area. We are collaborating with researchers at Ohio State on large scale three-dimensional reactive scattering problems. The governing equation is the Schroedinger equation and the computational technique for studying the physical phenomena relies upon repeated eigenanalysis of a Hamiltonian operator consisting of a Laplacian operator discretized in spherical co-ordinates plus a surface potential. The discrete operator has a tensor product

Nprocs	MFLOPS
2	172.50
4	322.03
8	586.29
16	1006.60
32	1412.73

Table 6: Parallel ARPACK on T3D Shared Memory

structure from the discrete Laplacian plus a diagonal matrix from the potential. The resulting matrix has a block structure consisting of $m \times m$ blocks of order n . The diagonal blocks are dense and the off diagonal blocks are scalar multiples of the order n identity matrix. It is virtually impossible to factor this matrix directly because the factors are dense in any ordering. We are using a distributed memory parallel version of ARPACK together with some preconditioning ideas to solve these problems on distributed memory machines. Encouraging computational results have been obtained on Cray Y-MP machines and also on the Intel Delta and the CM-5. The code has recently been ported to the CRAY T3D with very promising results. On a matrix of order 12800 computing the smallest eight eigenvalues using a Chebyshev polynomial preconditioner of degree eight the CRAY YMP executed at a rate of 290.66 Mflops while the T3D using the distributed-shared memory model executed at a peak rate of 1412 Mflops (See Table 6). For details about the method and experimental results, see [20], [45].

Nonsymmetric problems also arise in quantum chemistry. Researchers at University of Washington have used the code to investigate the effects of the electric field on InAs/GaSb and GaAs/Al_xGa_{1-x} as quantum wells. ARPACK was used to find highly accurate solutions to these nonsymmetric problems which couldn't be solved by other means. See [25] for details. Researchers at U. Massachusetts have used ARPACK to solve the eignvalue problems arising in their FEM quantum well Kp model for strained layer superlattices [4].

A final example of non-symmetric eigenproblems to be discussed here arises in magneto-hydrodynamics (MHD) involving the study of the interaction of a plasma and a magnetic field. The MHD equations describe the macroscopic behavior of the plasma in the magnetic field. These equations form a system of coupled nonlinear PDE. Linear stability analysis of the linearized MHD equations leads to a complex eigenvalue problem. Researchers at the Institute for Plasma Physics and Utrecht University in the Netherlands have modified the codes in ARPACK to work in complex arithmetic and are using the resulting code to obtain very accurate approximations to the eigenvalues lying on the Alfven curve. The code is not only computes extremely accurate solutions, it does so very efficiently in comparison to other methods that have been tried. See [22] for details.

There are many other applications. It is hoped that the examples that have been briefly discussed here will provide an indication of the versatility of the ARPACK software as well a the wide variety of eigenvalue problems that arise.

8 Conclusions

This paper has attempted to give an overview of the numerical solution of large scale eigenvalue problems. Basic theory and algorithms were introduced to motivate Krylov subspace projection methods. The focus has been on a particular variant, the Implicitly Restarted Arnoldi Method which has been developed into a substantial software package ARPACK.

There are a number of competing methods that have not been discussed here in any detail. Two notable methods that have not been discussed are methods based on the nonsymmetric two-sided Lanczos process and methods based upon subspace iteration. At this point, no single method appears to be viable for all problems. Certainly in the nonsymmetric case there is no "black box" technique and it is questionable that there is one in the symmetric case either. A block method called ABLE based upon two-sided nonsymmetric Lanczos is being developed by Bai, Day and Ye [2]. Software based upon subspace iteration with Chbeychev acceleration has been developed by Duff and Scott [12]. Jennifer Scott has also developed software based upon an explicitly restarted Chebyshev-Arnoldi method [42]. Finally, the Rational Krylov method being developed by Ruhe [36, 37] is very promising for the nonsymmetric problem when a factorization of the matrix is possible.

9 Acknowledgements

The computational results presented in Section 7 are due to Zdenko Tomasic and Dan Hu. I would like to thank Rich Lehoucq for producing Figures 1-3 and for constructive comments and discussions about this work.

Financial support for this work was provided in part by the National Science Foundation cooperative agreement CCR-912008, by ARPA contract number DAAL03-91-C-0047 (administered by the U.S. Army Research Office), and by the National Science Foundation project ASC-9408795.

References

[1] W.E. Arnoldi, The principle of minimized iterations in the solution of the matrix eigenvalue problem, *Quart. Appl. Math.* **9** , 17–29 (1951) .

[2] Z. Bai, D. Day and Q. Ye, ABLE: an Adaptive Block Lanczos Method for Non-Hermitian Eigenvalue Problems, *Tech. Rept. 95-04*, U. Kentucky, Lexington (1995).

[3] Z.Bai and G.W. Stewart, SRRIT - A FORTRAN subroutine to calculate the dominant invariant subspace of a nonsymmetric matrix, *Tech. Rept. 2908*, Dept of Computer Science, U. Maryland (1992).

[4] A. Baliga, D. Trifedi, N.G. Anderson, Tensile-strain effects in quantum-well and superlattice band structures *Phys. Rev. B* (1994).

[5] F. Chatelin and D. Ho, Arnoldi-Tchebychev procedure for large scale nonsymmetric matrices, *Math. Modeling and Num. Analysis* , **24**,53–65 (1990).

[6] J. Cullum, The simultaneous computation of a few of the algebraically largest and smallest eigenvalues of a large, symmetric, sparse matrix, *BIT* **18**, 265–275 (1978).

[7] J. Cullum and W.E. Donath, A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace for large, sparse symmetric matrices, in *Proc. 1974 IEEE Conference on Decision and Control*, IEEE Press, New York, 505–509 (1974).

[8] J. Cullum and R.A. Willoughby, Computing eigenvalues of very large symmetric matrices - an implementation of a Lanczos algorithm with no reorthogonalization, *J. Comput. Phys.* **434**, 329–358 (1981).

[9] J. Daniel, W.B. Gragg, L. Kaufman, G.W. Stewart, Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization, *Math. Comp.*,**30**, 772–795 (1976).

[10] J.J. Dongarra, J. Du Croz, S. Hammarling, and R.J. Hanson, Algorithm 656 An extended set of fortran basic linear algebra subprograms: Model implementation and test programs, *ACM Trans. Math. Soft.* **14**, 18–32 (1988).

[11] J.J. Dongarra, I.S. Duff, D.C. Sorensen and H.A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Publications, Philadelphia (1991).

[12] I.S. Duff and J Scott, Computing selected eigenvalues of large sparse unsymmetric matrices using subspace iteration, *ACM Transactions on Mathematical Software*, **19**, 137–159 (1993).

[13] W.S. Edwards, L.S. Tuckerman, R.A. Friesner and D.C. Sorensen, Krylov Methods for the Incompressible Navier-Stokes Equations, *Journal of Computational Physics*, **110**,82–102 (1994).

[14] T. Ericsson and A. Ruhe, The spectral transformation Lanczos method for the numerical solution of large sparse generalized symmetric eigenvalue problems, *Math. Comp.* **35**, 1251–1268 (1980).

[15] L. Feinswog, M. Sherman, W. Chiu, D.C. Sorensen, Improved Computational Methods for 3-Dimensional Image Reconstruction, *CRPC Tech. Rept.*, Rice University (in preparation).

[16] J.G.F. Francis, The QR transformation: A unitary analogue to the LR transformation, Parts I and II, *Comp. J.* **4**, 265–272, 332–345 (1961).

[17] G.H. Golub, R. Underwood, and J.H. Wilkinson, The Lanczos algorithm for the symmetric $Ax = \lambda Bx$ problem, *Report STAN-CS-72-270*, Department of Computer Science, Stanford U. Stanford, California ,(1972).

[18] G.H. Golub and C.F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, Maryland (1983).

[19] R.G. Grimes, J.G. Lewis and H.D. Simon, A shifted block Lanczos algorithm for solving sparse symmetric generalized eigenproblems, *SIAM J. Matrix Anal. Appl.* **15**, 228–272 (1994).

[20] P. Pendergast. Z. Darakjian, E. F. Hayes, D.C. Sorensen, Scalable Algorithms for Three-dimensional Reactive Scattering: Evaluation of a New Algorithm for Obtaining Surface Functions, *J. Comp. Phys.* , **113**,201–214 (1994).

[21] W. Karush, An iterative method for finding characteristic vectors of a symmetric matrix, *Pacific J. Math.* **1**, 233–248 (1951).

[22] M.N. Kooper, H.A. van der Vorst, S. Poedts, and J.P. Goedbloed, Application of the Implicitly Updated Arnoldi Method with a Complex Shift and Invert Strategy in MHD, *Tech. Rept., Institute for Plasmaphysics*, FOM Rijnhuizen, Nieuwegin, The Netherlands (Sep. 1993) (submitted to *Journal of Computational Physics*).

[23] C. Lanczos, An iteration method for the solution of the eigenvalue problem of linear differential and integral operators, *J. Res. Nat. Bur. Stand.* , **45**, 255–282 (1950).

[24] R.B. Lehoucq, *Analysis and Implementation of an Implicitly Restarted Arnoldi Iteration* Ph.D. Thesis, Rice U. (1995) (Available as *CAAM Tech. Rept. TR95-13*, Rice U., Houston)

[25] T.L. Li, K.J. Kuhn FEM solution to quantum wells by irreducible formulation *Dept. Elec. Eng. Tech. Rept.* U. Wash. (1993).

[26] R.B. Lehoucq and D.C. Sorensen, Deflation Techniques for an Implicitly Re-started Arnoldi Iteration, *CAAM-TR 94-13*, Rice U. , Houston (1994).

[27] R. Lehoucq, D.C. Sorensen, P.A. Vu, ARPACK: Fortran subroutines for solving large scale eigenvalue problems, Release 2.1, available from `netlib@ornl.gov` in the `scalapack` directory (1994).

[28] T.A. Manteuffel, Adaptive procedure for estimating parameters for the nonsymmetric Tchebychev iteration, *Numer. Math.* **31**, 183–208 (1978).

[29] K. Meerbergen and A. Spence, Implicitly restarted Arnoldi with purification for the shift-invert transformation, *Tech. Rept. TW225*, Katholieke Universitet Leuven, Belgium (1995).

[30] R.B. Morgan, On restarting the Arnoldi method for large scale eigenvalue problems, *Math. of Comp.* (to appear).

[31] C.C. Paige, *The Computation of Eigenvalues and Eigenvectors of Very Large Sparse Matrices,* Ph.D. thesis, Univ. of London (1971).

[32] B. Nour-Omid, B.N. Parlett, T. Ericsson, and P.S. Jensen, How to implement the spectral transformation, *Math. of Comp.*, **48**, 663–673 (1987).

[33] B.N. Parlett and D. S. Scott, The Lanczos algorithm with selective orthogonalization, *Math. Comp.* **33**, 311–328 (1979).

[34] B.N. Parlett, *The Symmetric Eigenvalue Problem* , Prentice-Hall, Englewood Cliffs, NJ. (1980).

[35] T.D. Romo, J.B. Clarage, D.C. Sorensen, and G.N. Phillips, Jr., Automatic Identification of Discrete Substates in Proteins: Singular Value Decomposition Analysis of Time Averaged Crystallographic Refinements, *CRPC-TR 94481*, Rice University (Oct. 1994).

[36] A. Ruhe, Rational Krylov sequence methods for eigenvalue computation, *Linear Algebra Apps.*, **58**, 391–405 (1984).

[37] A. Ruhe, Rational Krylov sequence methods for eigenvalue computation II, *Linear Algebra Apps.*, **197,198**, 283–295 (1994).

[38] Y. Saad, Variations on Arnoldi's method for computing eigenelements of large unsymmetric matrices, *Linear Algebra Apps.*, **34**, 269–295 (1980).

[39] Y. Saad, Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems, *Math. Comp.*, **42**, 567–588 (1984).

[40] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, Halsted Press-John Wiley & Sons Inc., New York (1992).

[41] Y. Saad and M. Schultz, GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems, *SIAM J. Scientific and Stat. Comp.*, **7**, 856–869 (1986).

[42] J.A. Scott. An Arnoldi code for computing selected eigenvalues of sparse real unsymmetric matrices, *Tech. Rept. RAL-93-097*, Rutherford Appleton Laboratory (1993).

[43] H. Simon, Analysis of the symmeteric Lanczos algorithm with reorthogonalization methods, *Linear Algebra and Its Applications* **61**, 101–131 (1984).

[44] D. C. Sorensen, Implicit application of polynomial filters in a k-step Arnoldi method, *SIAM J. Matrix Anal. Appl.*, **13**, pp. 357–385, 1992.

[45] D.C. Sorensen, P.A. Vu, Z. Tomasic, Algorithms and Software for Large Scale Eigenproblems on High Performance Computers, *High Performance Computing 1993 - Grand Challenges in Computer Simulation*,Adrian Tentner ed. , Proceedings 1993 Simulation Multiconference, Society for Computer Simulation, 149–154 (1993).

[46] G.W. Stewart, *Introduction to Matrix Computations,* Academic Press, New York, 1973.

[47] W.J. Stewart and A. Jennings, ALGORITHM 570: LOPSI a simultaneous iteratin method for real matrices [F2], *ACM Transactions on Mathematical Software,* **7**, 184–198 (1981).

[48] M. Van Heel, J. Frank, Use of Multivariate Statistics in Analysing the Images of Biological Macromolecules, *Ultramicroscopy*, **6** 187–194 (1981).

[49] S. Van Huffel and J. Vandewalle, *The Total Least Squares Provblem: Computational Aspects and Analysis* , Frontiers in Applied Mathematics **9**, SIAM Press, Philadelphia (1991).

[50] H.F. Walker, Implementation of the GMRES method using Householder transformations, *SIAM J. Scientific and Stat. Comp.* **9**,152–163 (1988).

[51] D.S. Watkins and L. Elsner, Convergence of algorithms of decomposition type for the eigenvalue problem, *Linear Algebra and Its Applications,* **143**, 19–47 (1991).

[52] J.H. Wilkinson, *The Algebraic Eigenvalue Problem,* Claredon Press, Oxford, England (1965).

SPARSE MATRICES IN MATLAB: DESIGN AND IMPLEMENTATION

JOHN R. GILBERT*, CLEVE MOLER†, AND ROBERT SCHREIBER‡

Dedicated to Gene Golub on the occasion of his 60th birthday.

Abstract. We have extended the matrix computation language and environment MATLAB to include sparse matrix storage and operations. The only change to the outward appearance of the MATLAB language is a pair of commands to create full or sparse matrices. Nearly all the operations of MATLAB now apply equally to full or sparse matrices, without any explicit action by the user. The sparse data structure represents a matrix in space proportional to the number of nonzero entries, and most of the operations compute sparse results in time proportional to the number of arithmetic operations on nonzeros.

Key words. MATLAB, mathematical software, matrix computation, sparse matrix algorithms.

AMS subject classifications. 65–04, 65F05, 65F20, 65F50, 68N15, 68R10.

1. Introduction. MATLAB is an interactive environment and programming language for numeric scientific computation [18]. One of its distinguishing features is the use of matrices as the only data type. In MATLAB, a matrix is a rectangular array of real or complex numbers. All quantities, even loop variables and character strings, are represented as matrices, although matrices with only one row, or one column, or one element are sometimes treated specially.

The part of MATLAB that involves computational linear algebra on dense matrices is based on direct adaptations of subroutines from LINPACK and EISPACK [5, 23]. An $m \times n$ real matrix is stored as a full array of mn floating point numbers. The computational complexity of basic operations such as addition or transposition is proportional to mn. The complexity of more complicated operations such as triangular factorization is proportional to mn^2. This has limited the applicability of MATLAB to problems involving matrices of order a few hundred on contemporary workstations and perhaps a few thousand on contemporary supercomputers.

We have now added sparse matrix storage and operations to MATLAB. This report describes our design and implementation.

Sparse matrices are widely used in scientific computation, especially in large-scale optimization, structural and circuit analysis, computational fluid dynamics, and, generally, the numerical solution of partial differential equations. Several effective Fortran subroutine packages for solving sparse linear systems are available, including SPARSPAK [11], the Yale Sparse Matrix Package [9], and some of the routines in the Harwell Subroutine Library [25].

Our work was facilitated by our knowledge of the techniques used in the Fortran sparse matrix packages, but we have not directly adapted any of their code. MATLAB

* Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304.

† The MathWorks, 325 Linfield Place, Menlo Park, California 94025.

‡ Research Institute for Advanced Computer Science, MS T045-1, NASA Ames Research Center, Moffett Field, CA 94035. This author's work was supported by the NAS Systems Division and DARPA via Cooperative Agreement NCC 2-387 between NASA and the University Space Research Association (USRA).

TABLE 1
Operations with the 4096 by 4096 discrete Laplacian.

	Sparse	Full
Memory	0.25 megabyte	128 megabytes
Compute Dx	0.2 seconds	30 seconds
Solve $Dx = b$	10 seconds	$>$ 12 hours

is written in C and we wished to take advantage of the data structures and other programming features of C that would not be used in a simple translation of Fortran code. We also wanted to implement the full range of matrix operations that MATLAB provides; the Fortran packages do not generally have routines for simply adding or transposing sparse matrices, for example. And, finally, we wanted to incorporate some recent algorithmic ideas that are not used in the Fortran packages.

J. H. Wilkinson's informal working definition of a sparse matrix was "any matrix with enough zeros that it pays to take advantage of them." So sparsity is an economic issue. By avoiding arithmetic operations on zero elements, sparse matrix algorithms require less computer time. And, perhaps more importantly, by not storing many zero elements, sparse matrix data structures require less computer memory. In a sense, we have not added any new functionality to MATLAB; we've merely made some existing functionality more efficient in terms of both time and storage.

An important descriptive parameter of a sparse matrix S is $\mathrm{nnz}(S)$, the number of nonzero elements in S. Computer storage requirements are proportional to nnz. The computational complexity of simple array operations should also be should be proportional to nnz, and perhaps also depend linearly on m or n, but be independent of the product mn. The complexity of more complicated operations involves such factors as ordering and fill-in, but an objective of a good sparse matrix algorithm should be:

> The time required for a sparse matrix operation should be proportional to number of arithmetic operations on nonzero quantities.

We call this the "time is proportional to flops" rule; it is a fundamental tenet of our design.

With sparse techniques, it is practical to handle matrices involving tens of thousands of nonzero elements on contemporary workstations. As one example, let D be the matrix representation of the discrete 5-point Laplacian on a square 64×64 grid with a nested dissection ordering. This is a 4096×4096 matrix with 20224 nonzeros. Table 1 gives the memory requirements for storing D as a MATLAB sparse matrix and as a traditional Fortran or MATLAB full matrix, as well as the execution time on a Sun SPARCstation-1 workstation for computing a matrix-vector product and solving a linear system of equations by elimination.

Band matrices are special cases of sparse matrices whose nonzero elements all happen to be near the diagonal. It would be somewhat more efficient, in both time and storage, to provide a third data structure and collection of operations for band matrices. We have decided against doing this because of the added complexity, particular in cases involving mixtures of full, sparse and band matrices. We suspect that solving linear systems with matrices which are dense within a narrow band might be twice as fast with band storage as it is with sparse matrix storage, but that linear systems with matrices that are sparse within the band (such as those obtained

from two-dimensional grids) are more efficiently solved with general sparse matrix technology. However, we have not investigated these tradeoffs in any detail.

In this paper, we concentrate on elementary sparse matrix operations, such as addition and multiplication, and on direct methods for solving sparse linear systems of equations. These operations are now included in the "core" of MATLAB. Except for a few short examples, we will not discuss higher level sparse matrix operations, such as iterative methods for linear systems. We intend to implement such operations as interpreted programs in the MATLAB language, so-called "m-files," outside the MATLAB core.

2. The user's view of sparse MATLAB.

2.1. Sparse matrix storage. We wish to emphasize the distinction between a matrix and what we call its *storage class*. A given matrix can conceivably be stored in many different ways—fixed point or floating point, by rows or by columns, real or complex, full or sparse—but all the different ways represent the same matrix. We now have two matrix storage classes in MATLAB, full and sparse.

Two MATLAB variables, `A` and `B`, can have different storage classes but still represent the same matrix. They occupy different amounts of computer memory, but in most other respects they are the same. Their elements are equal, their determinants and their eigenvalues are equal, and so on. The crucial question of which storage class to choose for a given matrix is the topic of Section 2.5.

Even though MATLAB is written in C, it follows its LINPACK and Fortran predecessors and stores full matrices by columns [5, 19]. This organization has been carried over to sparse matrices. A sparse matrix is stored as the concatenation of the sparse vectors representing its columns. Each sparse vector consists of a floating point array of nonzero entries (or two such arrays for complex matrices), together with an integer array of row indices. A second integer array gives the locations in the other arrays of the first element in each column. Consequently, the storage requirement for an $m \times n$ real sparse matrix with nnz nonzero entries is nnz reals and $nnz + n$ integers. On typical machines with 8-byte reals and 4-byte integers, this is $12nnz + 4n$ bytes. Complex matrices use a second array of nnz reals. Notice that m, the number of rows, is almost irrelevant. It is not involved in the storage requirements, nor in the operation counts for most operations. Its primary use is in error checks for subscript ranges. Similar storage schemes, with either row or column orientation, are used in the Fortran sparse packages.

2.2. Converting between full and sparse storage. Initially, we contemplated schemes for automatic conversion between sparse and full storage. There is a MATLAB precedent for such an approach. Matrices are either real or complex and the conversion between the two is automatic. Computations such as square roots and logarithms of negative numbers and eigenvalues of nonsymmetric matrices generate complex results from real data. MATLAB automatically expands the data structure by adding an array for the imaginary parts.

Moreover, several of MATLAB's functions for building matrices produce results that might effectively be stored in the sparse organization. The function `zeros(m,n)`, which generates an $m \times n$ matrix of all zeros, is the most obvious candidate. The functions `eye(n)` and `diag(v)`, which generate the $n \times n$ identity matrix and a diagonal matrix with the entries of vector v on the main diagonal, are also possibilities. Even `tril(A)` and `triu(A)`, which take the lower and upper triangular parts of a matrix A, might be considered. But this short list begins to demonstrate a difficulty—how far

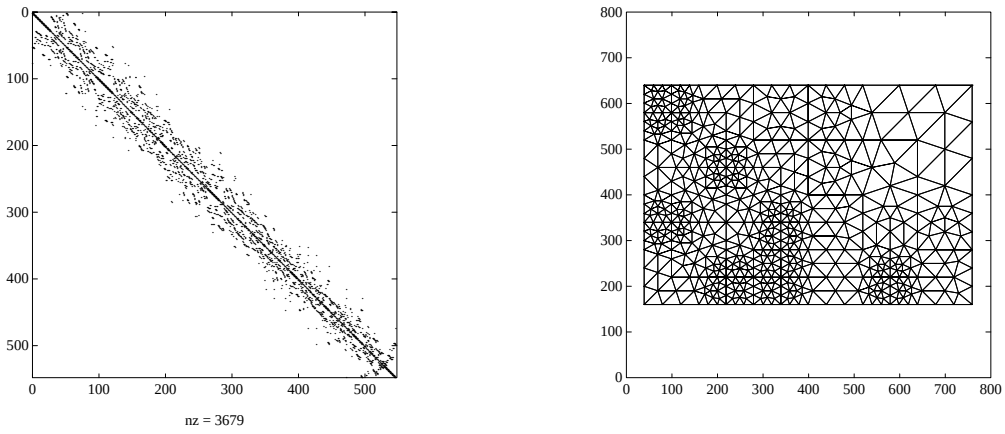


FIG. 1. *The Eppstein mesh as plotted by* `spy(A)` *and* `gplot(A,xy)`.

should "automatic sparsification" be carried? Is there some threshold value of sparsity where the conversion should be done? Should the user provide the value for such a sparsification parameter? We don't know the answers to these questions, so we decided to take another approach, which we have since found to be quite satisfactory.

No sparse matrices are created without some overt direction from the user. Thus, the changes we have made to MATLAB do not affect the user who has no need for sparsity. Operations on full matrices continue to produce full matrices. But once initiated, sparsity propagates. Operations on sparse matrices produce sparse matrices. And an operation on a mixture of sparse and full matrices produces a sparse result unless the operator ordinarily destroys sparsity. (Matrix addition is an example; more on this later.)

There are two new built-in functions, `full` and `sparse`. For any matrix A, `full(A)` returns A stored as a full matrix. If A is already full, then A is returned unchanged. If A is sparse, then zeros are inserted at the appropriate locations to fill out the storage. Conversely, `sparse(A)` removes any zero elements and returns A stored as a sparse matrix, regardless of how sparse A actually is.

2.3. Displaying sparse matrices. Sparse and full matrices print differently. The statement

```
A = [0 0 11; 22 0 0; 0 33 0]
```

produces a conventional MATLAB full matrix that prints as

```
A =
     0     0    11
    22     0     0
     0    33     0
```

The statement `S = sparse(A)` converts A to sparse storage, and prints

```
S =
   (2,1)       22
   (3,2)       33
   (1,3)       11
```

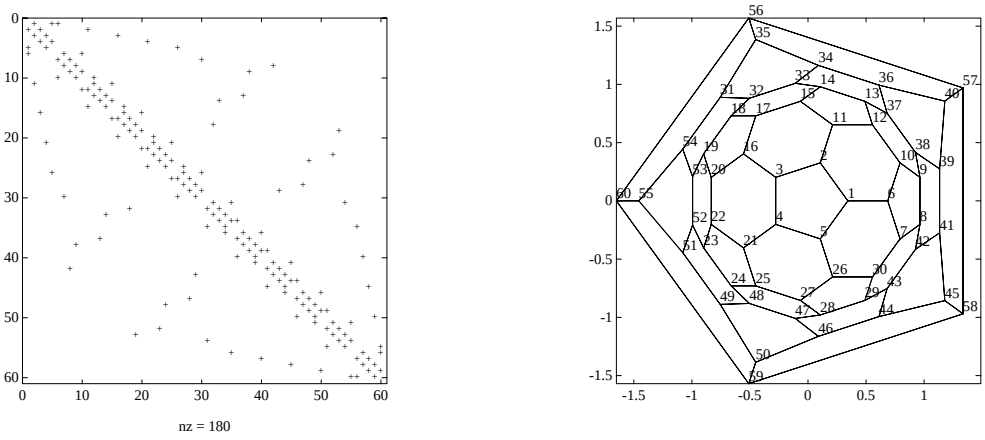


FIG. 2. *The buckyball as rendered by* `spy` *and* `gplot`.

As this illustrates, sparse matrices are printed as a list of their nonzero elements (with indices), in column major order.

The function `nnz(A)` returns the number of nonzero elements of A. It is implemented by scanning full matrices, and by access to the internal data structure for sparse matrices. The function `nzmax(A)` returns the number of storage locations for nonzeros allocated for A.

Graphic visualization of the structure of a sparse matrix is often a useful tool. The function `spy(A)` plots a silhouette of the nonzero structure of A. Figure 1 illustrates such a plot for a matrix that comes from a finite element mesh due to David Eppstein. A picture of the graph of a matrix is another way to visualize its structure. Laying out an arbitrary graph for display is a hard problem that we do not address. However, some sparse matrices (from finite element applications, for example) have spatial coordinates associated with their rows or columns. If xy contains such coordinates for matrix A, the function `gplot(A,xy)` draws its graph. The second plot in Figure 1 shows the graph of the sample matrix, which in this case is just the same as the finite element mesh. Figure 2 is another example: The `spy` plot is the 60×60 adjacency matrix of the graph of a Buckminster Fuller geodesic dome, a soccer ball, and a C_{60} molecule, and the `gplot` shows the graph itself.

Section 3.3.4 describes another function for visualizing the elimination tree of a matrix.

2.4. Creating sparse matrices. Usually one wants to create a sparse matrix directly, without first having a full matrix A and then converting it with `S = sparse(A)`. One way to do this is by simply supplying a list of nonzero entries and their indices. Several alternate forms of `sparse` (with more than one argument) allow this. The most general is

```
S = sparse(i,j,s,m,n,nzmax)
```

Ordinarily, i and j are vectors of integer indices, s is a vector of real or complex entries, and m, n, and $nzmax$ are integer scalars. This call generates an $m \times n$ sparse matrix, having one nonzero for each entry in the vectors i, j, and s, with $S(i(k), j(k)) = s(k)$, and with enough space allocated for S to have $nzmax$ nonzeros. The indices in i and j need not be given in any particular order.

If a pair of indices occurs more than once in i and j, `sparse` adds the corresponding values of s together. Then the sparse matrix S is created with one nonzero for each nonzero in this modified vector s. The argument s and one of the arguments i and j may be scalars, in which case they are expanded so that the first three arguments all have the same length.

There are several simplifications of the full six-argument call to `sparse`.

`S = sparse(i,j,s,m,n)` uses $nzmax = \text{length}(s)$.

`S = sparse(i,j,s)` uses $m = \max(i)$ and $n = \max(j)$.

`S = sparse(m,n)` is the same as `S = sparse([],[],[],m,n)`, where `[]` is MATLAB's empty matrix. It produces the ultimate sparse matrix, an $m \times n$ matrix of all zeros.

Thus for example

```
S = sparse([1 2 3], [3 1 2], [11 22 33])
```

produces the sparse matrix S from the example in Section 2.3, but does not generate any full 3 by 3 matrix during the process.

MATLAB's function `k = find(A)` returns a list of the positions of the nonzeros of A, counting in column-major order. For sparse MATLAB we extended the definition of `find` to extract the nonzero elements together with their indices. For any matrix A, full or sparse, `[i,j,s] = find(A)` returns the indices and values of the nonzeros. (The square bracket notation on the left side of an assignment indicates that the function being called can return more than one value. In this case, `find` returns three values, which are assigned to the three separate variables i, j, and s.) For example, this dissects and then reassembles a sparse matrix:

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

So does this, if the last row and column have nonzero entries:

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

Another common way to create a sparse matrix, particularly for finite difference computations, is to give the values of some of its diagonals. Two functions `diags` and `blockdiags` can create sparse matrices with specified diagonal or block diagonal structure.

There are several ways to read and write sparse matrices. The MATLAB `save` and `load` commands, which save the current workspace or load a saved workspace, have been extended to accept sparse matrices and save them efficiently. We have written a Fortran utility routine that converts a file containing a sparse matrix in the Harwell-Boeing format [6] into a file that MATLAB can load.

2.5. The results of sparse operations. What is the result of a MATLAB operation on sparse matrices? This is really two fundamental questions: what is the value of the result, and what is its storage class? In this section we discuss the answers that we settled on for those questions.

A function or subroutine written in MATLAB is called an *m-file*. We want it to be possible to write m-files that produce the same results for sparse and for full inputs. Of course, one could ensure this by converting all inputs to full, but that would defeat the goal of efficiency. A better idea, we decided, is to postulate that

> The value of the result of an operation does not depend on the storage class of the operands, although the storage class of the result may.

The only exception is a function to inquire about the storage class of an object: `issparse(A)` returns 1 if A is sparse, 0 otherwise.

Some intriguing notions were ruled out by our postulate. We thought, for a while, that in cases such as `A ./ S` (which denotes the pointwise quotient of A and S) we ought not to divide by zero where S is zero, since that would not produce anything useful; instead we thought to implement this as if it returned $A(i,j)/S(i,j)$ wherever $S(i,j) \neq 0$, leaving A unchanged elsewhere. All such ideas, however, were dropped in the interest of observing the rule that the result does not depend on storage class.

The second fundamental question is how to determine the storage class of the result of an operation. Our decision here is based on three ideas. First, the storage class of the result of an operation should depend only on the storage classes of the operands, not on their values or sizes. (Reason: it's too risky to make a heuristic decision about when to sparsify a matrix without knowing how it will be used.) Second, sparsity should not be introduced into a computation unless the user explicitly asks for it. (Reason: the full matrix user shouldn't have sparsity appear unexpectedly, because of the performance penalty in doing sparse operations on mostly nonzero matrices.) Third, once a sparse matrix is created, sparsity should propagate through matrix and vector operations, concatenation, and so forth. (Reason: most m-files should be able to do sparse operations for sparse input or full operations for full input without modification.)

Thus full inputs always give full outputs, except for functions like `sparse` whose purpose is to create sparse matrices. Sparse inputs, or mixed sparse and full inputs, follow these rules (where S is sparse and F is full):

- Functions from matrices to scalars or fixed-size vectors, like `size` or `nnz`, always return full results.
- Functions from scalars or fixed-size vectors to matrices, like `zeros`, `ones`, and `eye`, generally return full results. Having `zeros(m,n)` and `eye(m,n)` return full results is necessary to avoid introducing sparsity into a full user's computation; there are also functions `spzeros` and `speye` that return sparse zero and identity matrices.
- The remaining unary functions from matrices to matrices or vectors generally return a result of the same storage class as the operand (the main exceptions are `sparse` and `full`). Thus, `chol(S)` returns a sparse Cholesky factor, and `diag(S)` returns a sparse vector (a sparse $m \times 1$ matrix). The vectors returned by `max(S)`, `sum(S)`, and their relatives (that is, the vectors of column maxima and column sums respectively) are sparse, even though they may well be all nonzero.
- Binary operators yield sparse results if both operands are sparse, and full results if both are full. In the mixed case, the result's storage class depends on the operator. For example, `S + F` and `F \ S` (which solves the linear system $SX = F$) are full; `S .* F` (the pointwise product) and `S & F` are sparse.
- A block matrix formed by concatenating smaller matrices, like

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix},$$

is written as `[A B ; C D]` in MATLAB. If all the inputs are full, the result is full, but a concatenation that contains any sparse matrix is sparse. Submatrix indexing on the right counts as a unary operator; `A = S(i,j)` produces a sparse result (for sparse S) whether i and j are scalars or vectors. Submatrix indexing on the left, as in `A(i,j) = S`, does not change the storage class of the matrix being modified.

These decisions gave us some difficulty. Cases like `~S` and `S >= T`, where the result has many ones when the operands are sparse, made us consider adding more exceptions to the rules. We discussed the possibility of "sparse" matrices in which all the values not explicitly stored would be some scalar (like 1) rather than zero. We rejected these ideas in the interest of simplicity.

3. Implementation. This section describes the algorithms for the sparse operations in MATLAB in some detail. We begin with a discussion of fundamental data structures and design decisions.

3.1. Fundamentals.

3.1.1. Data structure. A most important implementation decision is the choice of a data structure. The internal representation of a sparse matrix must be flexible enough to implement all the MATLAB operations. For simplicity, we ruled out the use of different data structures for different operations. The data structure should be compact, storing only nonzero elements, with a minimum of overhead storage for integers or pointers. Wherever possible, it should support matrix operations in time proportional to flops. Since MATLAB is an interpreted, high-level matrix language, efficiency is more important in matrix arithmetic and matrix-vector operations than in accessing single elements of matrices.

These goals are met by a simple column-oriented scheme that has been widely used in sparse matrix computation. A sparse matrix is a C record structure with the following constituents. The nonzero elements are stored in a one-dimensional array of double-precision reals, in column major order. (If the matrix is complex, the imaginary parts are stored in another such array.) A second array of integers stores the row indices. A third array of $n+1$ integers stores the index into the first two arrays of the leading entry in each of the n columns, and a terminating index whose value is nnz. Thus a real $m \times n$ sparse matrix with nnz nonzeros uses nnz reals and $nnz + n + 1$ integers.

This scheme is not efficient for manipulating matrices one element at a time: access to a single element takes time at least proportional to the logarithm of the length of its column; inserting or removing a nonzero may require extensive data movement. However, element-by-element manipulation is rare in MATLAB (and is expensive even in full MATLAB). Its most common application would be to create a sparse matrix, but this is more efficiently done by building a list $[i, j, s]$ of matrix elements in arbitrary order and then using `sparse(i,j,s)` to create the matrix.

The sparse data structure is allowed to have unused elements after the end of the last column of the matrix. Thus an algorithm that builds up a matrix one column at a time can be implemented efficiently by allocating enough space for all the expected nonzeros at the outset.

3.1.2. Storage allocation. Storage allocation is one of the thorniest parts of building portable systems. MATLAB handles storage allocation for the user, invisibly allocating and deallocating storage as matrices appear, disappear, and change size.

Sometimes the user can gain efficiency by preallocating storage for the result of a computation. One does this in full MATLAB by allocating a matrix of zeros and filling it in incrementally. Similarly, in sparse MATLAB one can preallocate a matrix (using `sparse`) with room for a specified number of nonzeros. Filling in the sparse matrix a column at a time requires no copying or reallocation.

Within MATLAB, simple "allocate" and "free" procedures handle storage allocation. (We will not discuss how MATLAB handles its free storage and interfaces to the operating system to provide these procedures.) There is no provision for doing storage allocation within a single matrix; a matrix is allocated as a single block of storage, and if it must expand beyond that block it is copied into a newly allocated larger block.

MATLAB must allocate space to hold the results of operations. For a full result, MATLAB allocates mn elements at the start of the computation. This strategy could be disastrous for sparse matrices. Thus, sparse MATLAB attempts to make a reasonable choice of how much space to allocate for a sparse result.

Some sparse matrix operations, like Cholesky factorization, can predict in advance the exact amount of storage the result will require. These operations simply allocate a block of the right size before the computation begins. Other operations, like matrix multiplication and LU factorization, have results of unpredictable size. These operations are all implemented by algorithms that compute one column at a time. Such an algorithm first makes a guess at the size of the result. If more space is needed at some point, it allocates a new block that is larger by a constant factor (typically 1.5) than the current block, copies the columns already computed into the new block, and frees the old block.

Most of the other operations compute a simple upper bound on the storage required by the result to decide how much space to allocate—for example, the pointwise product `S .* T` uses the smaller of $\mathrm{nnz}(S)$ and $\mathrm{nnz}(T)$, and `S + T` uses the smaller of $\mathrm{nnz}(S) + \mathrm{nnz}(T)$ and mn.

3.1.3. The sparse accumulator. Many sparse matrix algorithms use a dense working vector to allow random access to the currently "active" column or row of a matrix. The sparse MATLAB implementation formalizes this idea by defining an abstract data type called the sparse accumulator, or SPA. The SPA consists of a dense vector of real (or complex) values, a dense vector of true/false "occupied" flags, and an unordered list of the indices whose occupied flags are true.

The SPA represents a column vector whose "unoccupied" positions are zero and whose "occupied" positions have values (zero or nonzero) specified by the dense real or complex vector. It allows random access to a single element in constant time, as well as sequencing through the occupied positions in constant time per element. Most matrix operations allocate the SPA (with appropriate dimension) at their beginning and free it at their end. Allocating the SPA takes time proportional to its dimension (to turn off all the occupied flags), but subsequent operations take only constant time per nonzero.

In a sense the SPA is a register and an instruction set in an abstract machine architecture for sparse matrix computation. MATLAB manipulates the SPA through some thirty-odd access procedures. About half of these are operations between the SPA and a sparse or dense vector, from a "spaxpy" that implements SPA := SPA+ax (where a is a scalar and x is a column of a sparse matrix) to a "spaeq" that tests elementwise equality. Other routines load and store the SPA, permute it, and access individual elements. The most complicated SPA operation is a depth-first search on

an acyclic graph, which marks as "occupied" a topologically ordered list of reachable vertices; this is used in the sparse triangular solve described in Section 3.4.2.

The SPA simplifies data structure manipulation, because all fill occurs in the SPA; that is, only in the SPA can a zero become nonzero. The "spastore" routine does not store exact zeros, and in fact the sparse matrix data structure never contains any explicit zeros. Almost all real arithmetic operations occur in SPA routines, too, which simplifies MATLAB's tally of flops. (The main exceptions are in certain scalar-matrix operations like `2*A`, which are implemented without the SPA for efficiency.)

3.1.4. Asymptotic complexity analysis. A strong philosophical principle in the sparse MATLAB implementation is that it should be possible to analyze the complexity of the various operations, and that they should be efficient in the asymptotic sense as well as in practice. This section discusses this principle, in terms of both theoretical ideals and engineering compromises.

Ideally all the matrix operations would use time proportional to flops, that is, their running time would be proportional to the number of nonzero real arithmetic operations performed. This goal cannot always be met: for example, `[0 1] + [1 0]` does no nonzero arithmetic. A more accurate statement is that time should be proportional to flops or data size, whichever is larger. Here "data size" means the size of the output and that part of the input that is used nontrivially; for example, in `A*b` only those columns of A corresponding to nonzeros in b participate nontrivially.

This more accurate ideal can be realized in almost all of MATLAB. The exceptions are some operations that do no arithmetic and cannot be implemented in time proportional to data size. The algorithms to compute most of the reordering permutations described in Section 3.3 are efficient in practice but not linear in the worst case. Submatrix indexing is another example: if i and j are vectors of row and column indices, `B = A(i,j)` may examine all the nonzeros in the columns $A(:, j)$, and `B(i,j) = A` can at worst take time linear in the total size of B.

The MATLAB implementation actually violates the "time proportional to flops" philosophy in one systematic way. The list of occupied row indices in the SPA is not maintained in numerical order, but the sparse matrix data structure does require row indices to be ordered. Sorting the row indices when storing the SPA would theoretically imply an extra factor of $O(\log n)$ in the worst-case running times of many of the matrix operations. All our algorithms could avoid this factor—usually by storing the matrix with unordered row indices, then using a linear-time transposition sort to reorder all the rows of the final result at once—but for simplicity of programming we included the sort in "spastore".

The idea that running time should be susceptible to analysis helps the user who writes programs in MATLAB to choose among alternative algorithms, gives guidance in scaling up running times from small examples to larger problems, and, in a general-purpose system like MATLAB, gives some insurance against an unexpected worst-case instance arising in practice. Of course complete *a priori* analysis is impossible—the work in sparse LU factorization depends on numerical pivoting choices, and the efficacy of a heuristic reordering such as minimum degree is unpredictable—but we feel it is worthwhile to stay as close to the principle as we can.

In a technical report [14] we present some experimental evidence that sparse MATLAB operations require time proportional to flops and data size in practice.

3.2. Factorizations. The LU and Cholesky factorizations of a sparse matrix yield sparse results. MATLAB does not yet have a sparse QR factorization. Section 3.6 includes some remarks on sparse eigenvalue computation in MATLAB.

3.2.1. LU **Factorization.** If A is a sparse matrix, `[L,U,P] = lu(A)` returns three sparse matrices such that $PA = LU$, as obtained by Gaussian elimination with partial pivoting. The permutation matrix P uses only $O(n)$ storage in sparse format. As in dense MATLAB, `[L,U] = lu(A)` returns a permuted unit lower triangular and an upper triangular matrix whose product is A.

Since sparse LU must behave like MATLAB's full LU, it does not pivot for sparsity. A user who happens to know a good column permutation Q for sparsity can, of course, ask for `lu(A*Q')`, or `lu(A(:,q))` where q is an integer permutation vector. Section 3.3 describes a few ways to find such a permutation. The matrix division operators \ and / do pivot for sparsity by default; see Section 3.4.

We use a version of the `GPLU` algorithm [15] to compute the LU factorization. This computes one column of L and U at a time by solving a sparse triangular system with the already-finished columns of L. Section 3.4.2 describes the sparse triangular solver that does most of the work. The total time for the factorization is proportional to the number of nonzero arithmetic operations (plus the size of the result), as desired.

The column-oriented data structure for the factors is created as the factorization progresses, never using any more storage for a column than it requires. However, the total size of L or U cannot be predicted in advance. Thus the factorization routine makes an initial guess at the required storage, and expands that storage (by a factor of 1.5) whenever necessary.

3.2.2. Cholesky factorization. As in full MATLAB, `R = chol(A)` returns the upper triangular Cholesky factor of a Hermitian positive definite matrix A. Pivoting for sparsity is not automatic, but minimum degree and profile-limiting permutations can be computed as described in Section 3.3.

Our current implementation of Cholesky factorization emphasizes simplicity and compatibility with the rest of sparse MATLAB; thus it does not use some of the more sophisticated techniques such as the compressed index storage scheme [11, Sec. 5.4.2], or supernodal methods to take advantage of the clique structure of the chordal graph of the factor [2]. It does, however, run in time proportional to arithmetic operations with little overhead for data structure manipulation.

We use a slightly simplified version of an algorithm from the Yale Sparse Matrix Package [9], which is described in detail by George and Liu [11]. We begin with a combinatorial step that determines the number of nonzeros in the Cholesky factor (assuming no exact cancellation) and allocates a large enough block of storage. We then compute the lower triangular factor R^T one column at a time. Unlike YSMP and SPARSPAK, we do not begin with a symbolic factorization; instead, we create the sparse data structure column by column as we compute the factor. The only reason for the initial combinatorial step is to determine how much storage to allocate for the result.

3.3. Permutations. A permutation of the rows or columns of a sparse matrix A can be represented in two ways. A permutation matrix P acts on the rows of A as `P*A` or on the columns as `A*P'`. A permutation vector p, which is a full vector of length n containing a permutation of `1:n`, acts on the rows of A as `A(p,:)` or on the columns as `A(:,p)`. Here p could be either a row vector or a column vector.

Both representations use $O(n)$ storage, and both can be applied to A in time proportional to $\mathrm{nnz}(A)$. The vector representation is slightly more compact and efficient, so the various sparse matrix permutation routines all return vectors—full row vectors, to be precise—with the exception of the pivoting permutation in LU factorization.

Converting between the representations is almost never necessary, but it is simple. If I is a sparse identity matrix of the appropriate size, then P is `I(p,:)` and P^T is `I(:,p)`. Also p is `(P*(1:n)')'` or `(1:n)*P'`. (We leave to the reader the puzzle of using `find` to obtain p from P without doing any arithmetic.) The inverse of P is `P'`; the inverse r of p can be computed by the "vectorized" statement `r(p) = 1:n`.

3.3.1. Permutations for sparsity: Asymmetric matrices. Reordering the columns of a matrix can often make its LU or QR factors sparser. The simplest such reordering is to sort the columns by increasing 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 MATLAB function `p = colperm(A)` computes this column-count permutation. It is implemented as a two-line m-file:

```
[i,j] = find(A);
[ignore,p] = sort(diff(find(diff([0 j' inf]))));
```

The vector j is the column indices of all the nonzeros in A, in column major order. The inner `diff` computes first differences of j to give a vector with ones at the starts of columns and zeros elsewhere; the `find` converts this to a vector of column-start indices; the outer `diff` gives the vector of column lengths; and the second output argument from `sort` is the permutation that sorts this vector.

The symmetric reverse Cuthill-McKee ordering described in Section 3.3.2 can be used for asymmetric matrices as well; the function `symrcm(A)` actually operates on the nonzero structure of $A + A^T$. This is sometimes a good ordering for matrices that come from one-dimensional problems or problems that are in some sense long and thin.

Minimum degree is an ordering that often performs better than `colperm` or `symrcm`. The sparse MATLAB function `p = colmmd(A)` computes a minimum degree ordering for the columns of A. This column ordering is the same as a symmetric minimum degree ordering for the matrix $A^T A$, though we do not actually form $A^T A$ to compute it.

George and Liu [10] survey the extensive development of efficient and effective versions of symmetric minimum degree, most of which is reflected in the symmetric minimum degree codes in SPARSPAK, YSMP, and the Harwell Subroutine Library. The MATLAB version of minimum degree uses many of these ideas, as well as some ideas from a parallel symmetric minimum degree algorithm by Gilbert, Lewis, and Schreiber [13]. We sketch the algorithm briefly to show how these ideas are expressed in the framework of column minimum degree. The reader who is not interested in all the details can skip to Section 3.3.2.

Although most column minimum degree codes for asymmetric matrices are based on a symmetric minimum degree code, our organization is the other way around: MATLAB's symmetric minimum degree code (described in Section 3.3.2) is based on its column minimum degree code. This is because the best way to represent a symmetric matrix (for the purposes of minimum degree) is as a union of cliques, or full submatrices. When we begin with an asymmetric matrix A, we wish to reorder its columns by using a minimum degree order on the symmetric matrix $A^T A$—but each row of A induces a clique in $A^T A$, so we can simply use A itself to represent $A^T A$ instead of forming the product explictly. Speelpenning [24] called such a clique representation of a symmetric graph the "generalized element" representation; George and

Liu [10] call it the "quotient graph model." Ours is the first column minimum degree implementation that we know of whose data structures are based directly on A, and which does not need to spend the time and storage to form the structure of A^TA. The idea for such a code is not new, however—George and Liu [10] suggest it, and our implementation owes a great deal to discussions between the first author and Esmond Ng and Barry Peyton of Oak Ridge National Laboratories.

We simulate symmetric Gaussian elimination on A^TA, using a data structure that represents A as a set of vertices and a set of cliques whose union is the graph of A^TA. Initially, each column of A is a vertex and each row is a clique. Elimination of a vertex j induces fill among all the (so far uneliminated) vertices adjacent to j. This means that all the vertices in cliques containing j become adjacent to one another. Thus all the cliques containing vertex j merge into one clique. In other words, all the rows of A with nonzeros in column j disappear, to be replaced by a single row whose nonzero structure is their union. Even though fill is implicitly being added to A^TA, the data structure for A gets smaller as the rows merge, so no extra storage is required during the elimination.

Minimum degree chooses a vertex of lowest degree (the sparsest remaining column of A^TA, or the column of A having nonzero rows in common with the fewest other columns), eliminates that vertex, and updates the remainder of A by adding fill (i.e. merging rows). This whole process is called a "stage"; after n stages the columns are all eliminated and the permutation is complete. In practice, updating the data structure after each elimination is too slow, so several devices are used to perform many eliminations in a single stage before doing the update for the stage.

First, instead of finding a single minimum-degree vertex, we find an entire "independent set" of minimum-degree vertices with no common nonzero rows. Eliminating one such vertex has no effect on the others, so we can eliminate them all at the same stage and do a single update. George and Liu call this strategy "multiple elimination". (They point out that the resulting permutation may not be a strict minimum degree order, but the difference is generally insignificant.)

Second, we use what George and Liu call "mass elimination": After a vertex j is eliminated, its neighbors in A^TA form a clique (a single row in A). Any of those neighbors whose own neighbors all lie within that same clique will be a candidate for elimination at the next stage. Thus, we may as well eliminate such a neighbor during the same stage as j, immediately after j, delaying the update until afterward. This often saves a tremendous number of stages because of the large cliques that form late in the elimination. (The number of stages is reduced from the height of the elimination tree to approximately the height of the clique tree; for many two-dimensional finite element problems, for example, this reduces the number of stages from about $\sqrt{n}$ to about $\log n$.) Mass elimination is particularly simple to implement in the column data structure: after all rows with nonzeros in column j are merged into one row, the columns to be eliminated with j are those whose only remaining nonzero is in that new row.

Third, we note that any two columns with the same nonzero structure will be eliminated in the same stage by mass elimination. Thus we allow the option of combining such columns into "supernodes" (or, as George and Liu call them, "indistinguishable nodes"). This speeds up the ordering by making the data structure for A smaller. The degree computation must account for the sizes of supernodes, but this turns out to be an advantage for two reasons. The quality of the ordering actually improves slightly if the degree computation does not count neighbors within the same

supernode. (George and Liu observe this phenomenon and call the number of neighbors outside a vertex's supernode its "external degree.") Also, supernodes improve the approximate degree computation described below. Amalgamating columns into supernodes is fairly slow (though it takes time only proportional to the size of A). Supernodes can be amalgamated at every stage, periodically, or never; the current default is every third stage.

Fourth, we note that the structure of A^TA is not changed by dropping any row of A whose nonzero structure is a subset of that of another row. This row reduction speeds up the ordering by making the data structure smaller. More significantly, it allows mass elimination to recognize larger cliques, which decreases the number of stages dramatically. Duff and Reid [8] call this strategy "element absorption." Row reduction takes time proportional to multiplying AA^T in the worst case (though the worst case is rarely realized and the constant of proportionality is very small). By default, we reduce at every third stage; again the user can change this.

Fifth, to achieve larger independent sets and hence fewer stages, we relax the minimum degree requirement and allow elimination of any vertex of degree at most $\alpha d+\beta$, where d is the minimum degree at this stage and α and β are parameters. The choice of threshold can be used to trade off ordering time for quality of the resulting ordering. For problems that are very large, have many right-hand sides, or factor many matrices with the same nonzero structure, ordering time is insignificant and the tightest threshold is appropriate. For one-off problems of moderate size, looser thresholds like $1.5d+2$ or even $2d+10$ may be appropriate. The threshold can be set by the user; its default is $1.2d+1$.

Sixth and last, our code has the option of using an "approximate degree" instead of computing the actual vertex degrees. Recall that a vertex is a column of A, and its degree is the number of other columns with which it shares some nonzero row. Computing all the vertex degrees in A^TA takes time proportional to actually computing A^TA, though the constant is quite small and no extra space is needed. Still, the exact degree computation can be the slowest part of a stage. If column j is a supernode containing $n(j)$ original columns, we define its approximate degree as

$$d(j) = \sum_{a_{ij}\neq 0} (\mathrm{nnz}(A(i,:)) - n(j)).$$

This can be interpreted as the sum of the sizes of the cliques containing j, except that j and the other columns in its supernode are not counted. This is a fairly good approximation in practice; it errs only by overcounting vertices that are members of at least three cliques containing j. George and Liu call such vertices "outmatched nodes," and observe that they tend to be rare in the symmetric algorithm. Computing approximate degrees takes only time proportional to the size of A.

Column minimum degree sometimes performs poorly if the matrix A has a few very dense rows, because then the structure of A^TA consists mostly of the cliques induced by those rows. Thus `colmmd` will withhold from consideration any row containing more than a fixed proportion (by default, 50%) of nonzeros.

All these options for minimum degree are under the user's control, though the casual user of MATLAB never needs to change the defaults. The default settings use approximate degrees, row reduction and supernode amalgamation every third stage, and a degree threshold of $1.2d+1$, and withhold rows that are at least 50% dense.

3.3.2. Permutations for sparsity: Symmetric matrices. Preorderings for Cholesky factorization apply symmetrically to the rows and columns of a symmetric

positive definite matrix. Sparse MATLAB includes two symmetric preordering permutation functions. The `colperm` permutation can also be used as a symmetric ordering, but it is usually not the best choice.

Bandwidth-limiting and profile-limiting orderings are useful for matrices whose structure is "one-dimensional" in a sense that is hard to make precise. The reverse Cuthill-McKee ordering is an effective and inexpensive profile-limiting permutation. MATLAB function `p = symrcm(A)` returns a reverse Cuthill-McKee permutation for symmetric matrix A. The algorithm first finds a "pseudo-peripheral" vertex of the graph of A, then generates a level structure by breadth-first search and orders the vertices by decreasing distance from the pseudo-peripheral vertex. Our implementation is based closely on the SPARSPAK implementation as described in the book by George and Liu [11].

Profile methods like reverse Cuthill-McKee are not the best choice for most large matrices arising from problems with two or more dimensions, or problems without much geometric structure, because such matrices typically do not have reorderings with low profile. The most generally useful symmetric preordering in MATLAB is minimum degree, obtained by the function `p = symmmd(A)`. Our symmetric minimum degree implementation is based on the column minimum degree described in Section 3.3.1. In fact, `symmmd` just creates a nonzero structure K with a column for each column of A and a row for each above-diagonal nonzero in A, such that K^TK has the same nonzero structure as A; it then calls the column minimum degree code on K.

3.3.3. Nonzero diagonals and block triangular form. A square nonsingular matrix A always has a row permutation p such that $A(p,:)$ has nonzeros on its main diagonal. The MATLAB function `p = dmperm(A)` computes such a permutation. With two output arguments, the function `[p,q] = dmperm(A)` gives both row and column permutations that put A into block upper triangular form; that is, $A(p,q)$ has a nonzero main diagonal and a block triangular structure with the largest possible number of blocks. Notice that the permutations p returned by these two calls are likely to be different.

The most common application of block triangular form is to solve a reducible system of linear equations by block back substitution, factoring only the diagonal blocks of the matrix. Figure 9 is an m-file that implements this algorithm. The m-file illustrates the call `[p,q,r] = dmperm(A)`, which returns p and q as before, and also a vector r giving the boundaries of the blocks of the block upper triangular form. To be precise, if there are b blocks in each direction, then r has length $b+1$, and the i-th diagonal block of $A(p,q)$ consists of rows and columns with indices from $r(i)$ through $r(i+1)-1$.

Any matrix, whether square or not, has a form called the "Dulmage-Mendelsohn decomposition" [4, 20], which is the same as ordinary block upper triangular form if the matrix is square and nonsingular. The most general form of the decomposition, for arbitrary rectangular A, is `[p,q,r,s] = dmperm(A)`. The first two outputs are permutations that put $A(p,q)$ into block form. Then r describes the row boundaries of the blocks and s the column boundaries: the i-th diagonal block of $A(p,q)$ has rows $r(i)$ through $r(i+1)-1$ and columns $s(i)$ through $s(i+1)-1$. The first diagonal block may have more columns than rows, the last diagonal block may have more rows than columns, and all the other diagonal blocks are square. The subdiagonal blocks are all zero. The square diagonal blocks have nonzero diagonal elements. All the diagonal blocks are irreducible; for the non-square blocks, this means that they have the "strong Hall property" [4]. This block form can be used to solve least squares

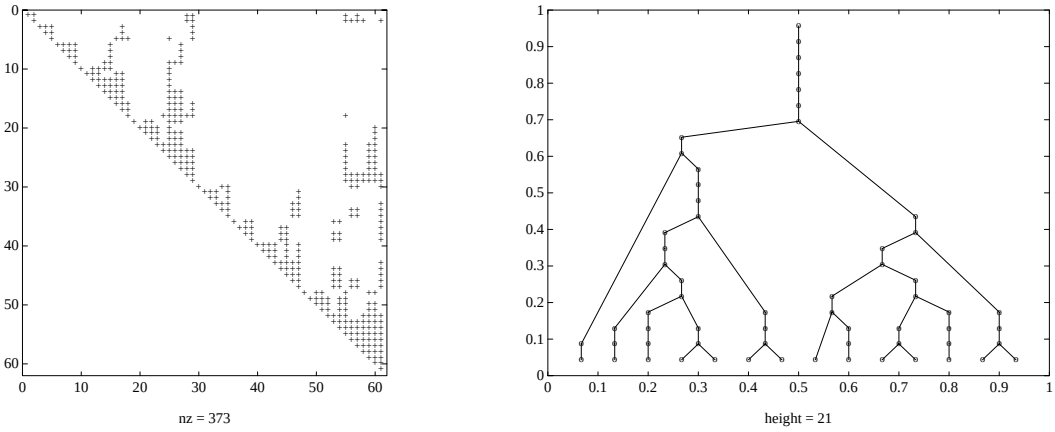


FIG. 3. *The Cholesky factor of a matrix and its elimination tree.*

problems by a method analogous to block back-substitution; see the references for more details.

3.3.4. Elimination trees. The elimination tree [21] of a symmetric positive definite matrix describes the dependences among rows or columns in Cholesky factorization. Liu [16] surveys applications of the elimination tree in sparse factorization. The nodes of the tree are the integers 1 through n, representing the rows of the matrix and of its upper triangular Cholesky factor. The parent of row i is the smallest $j > i$ such that the (i, j) element of the upper triangular Cholesky factor of the matrix is nonzero; if row i of the factor is zero after the diagonal, then i is a root. If the matrix is irreducible then its only root is node n.

Liu describes an algorithm to find the elimination tree without forming the Cholesky factorization, in time almost linear in the size of the matrix. That algorithm is implemented as the MATLAB function `[t,q] = etree(A)`. The resulting tree is represented by a row vector t of parent pointers: $t(i)$ is the parent of node i, or zero if i is a root.

The optional second output q is a permutation vector which gives a postorder permutation of the tree, or of the rows and columns of A. This permutation reorders the tree vertices so that every subtree is numbered consecutively, with the subtree's root last. This is an "equivalent reordering" of A, to use Liu's terminology: the Cholesky factorization of $A(q, q)$ has the same fill, operation count, and elimination tree as that of A. The permutation brings together the "fundamental supernodes" of A, which are full blocks in the Cholesky factor whose structure can be exploited in vectorized or parallel supernodal factorization [2, 17].

The postorder permutation can also be used to lay out the vertices for a picture of the elimination tree. The function `tspy(A)` plots a picture of the elimination tree of A, as shown in Figure 3.

3.4. Matrix division. The usual way to solve systems of linear equations in MATLAB is not by calling `lu` or `chol`, but with the matrix division operators / and \. If A is square, the result of `X = A\B` is the solution to the linear system $AX = B$; if A is not square then a least squares solution is computed. The result of `X = A/B` is the solution to $A = XB$, which is `(B'\A')'`. Full MATLAB computes $A\backslash B$ by LU

factorization with partial pivoting if A is square, or by QR factorization with column pivoting if not.

3.4.1. The sparse linear equation solver. Like full MATLAB, sparse MATLAB uses direct factorization methods to solve linear systems. The philosophy behind this is that iterative linear system solvers are best implemented as MATLAB m-files, which can use the sparse matrix data structures and operations in the core of MATLAB.

If A is sparse, MATLAB chooses among a sparse triangular solve, sparse Cholesky factorization, and sparse LU factorization, with optional preordering by minimum degree in the last two cases. The result returned has the same storage class as B. The outline of sparse $A\backslash B$ is as follows.

- If A is not square, solve the least squares problem.
- Otherwise, if A is triangular, perform a sparse triangular solve for each column of B.
- Otherwise, if A is a permutation of a triangular matrix, permute it and then perform a sparse triangular solve for each column of B.
- Otherwise, if A is Hermitian and has positive real diagonal elements, find a symmetric minimum degree order p and attempt to compute the Cholesky factorization of $A(p,p)$. If successful, finish with two sparse triangular solves for each column of B.
- Otherwise (if A is not Hermitian with positive diagonal or if Cholesky factorization fails), find a column minimum degree order p, compute the LU factorization with partial pivoting of $A(:,p)$, and perform two sparse triangular solves for each column of B.

Section 3.5 describes the sparse least squares method we currently use.

For a square matrix, the four possibilities are tried in order of increasing cost. Thus, the cost of checking alternatives is a small fraction of the total cost. The test for triangular A takes only $O(n)$ time if A is n by n; it just examines the first and last row indices in each column. (Notice that a test for triangularity would take $O(n^2)$ time for a full matrix.) The test for a "morally triangular" matrix, which is a row and column permutation of a nonsingular triangular matrix, takes time proportional to the number of nonzeros in the matrix and is in practice very fast. (A Dulmage-Mendelsohn decomposition would also detect moral triangularity, but would be slower.) These tests mean that, for example, the MATLAB sequence

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

will use triangular solves for both matrix divisions, since L is morally triangular and U is triangular.

The test for Hermitian positive diagonal is an inexpensive guess at when to use Cholesky factorization. Cholesky is quite a bit faster than LU, both because it does half as many operations and because storage management is simpler. (The time to look at every element of A in the test is insignificant.) Of course it is possible to construct examples in which Cholesky fails only at the last column of the reordered matrix, wasting significant time, but we have not seen this happen in practice.

The function `spparms` can be used to turn the minimum degree preordering off if the user knows how to compute a better preorder for the particular matrix in question.

MATLAB's matrix division does not have a block-triangular preordering built in, unlike (for example) the Harwell `MA28` code. Block triangular preordering and solution can be implemented easily as an m-file using the `dmperm` function; see Section 4.3.

Full MATLAB uses the LINPACK condition estimator and gives a warning if the denominator in matrix division is nearly singular. Sparse MATLAB should do the same, but the current version does not yet implement it.

3.4.2. Sparse triangular systems. The triangular linear system solver, which is also the main step of LU factorization, is based on an algorithm of Gilbert and Peierls [15]. When A is triangular and b is a sparse vector, $x = A\backslash b$ is computed in two steps. First, the nonzero structures of A and b are used (as described below) to make a list of the nonzero indices of x. This list is also the list of columns of A that participate nontrivially in the triangular solution. Second, the actual values of x are computed by using each column on the list to update the sparse accumulator with a "spaxpy" operation (Section 3.1.3). The list is generated in a "topological" order, which is one that guarantees that x_i is computed before column i of A is used in a spaxpy. Increasing order is one topological order of a lower triangular matrix, but any topological order will serve.

It remains to describe how to generate the topologically ordered list of indices efficiently. Consider the directed graph whose vertices are the columns of A, with an edge from j to i if $a_{ij} \neq 0$. (No extra data structure is needed to represent this graph—it is just an interpretation of the standard column data structure for A.) Each nonzero index of b corresponds to a vertex of the graph. The set of nonzero indices of x corresponds to the set of all vertices of b, plus all vertices that can be reached from vertices of b via directed paths in the graph of A. (This is true even if A is not triangular [12].) Any graph-searching algorithm could be used to identify those vertices and find the nonzero indices of x. A depth-first search has the advantage that a topological order for the list can be generated during the search. We add each vertex to the list at the time the depth-first search backtracks from that vertex. This creates the list in the reverse of a topological order; the numerical solution step then processes the list backwards, in topological order.

The reason to use this "reverse postorder" as the topological order is that there seems to be no way to generate the list in increasing or decreasing order, and the time wasted in sorting it would often be more than the number of arithmetic operations. However, the depth-first search examines just once each nonzero of A that participates nontrivially in the solve. Thus generating the list takes time proportional to the number of nonzero arithmetic operations in the numerical solve. This means that LU factorization can run in time proportional to arithmetic operations.

3.5. Least squares and the augmented system. We have not yet written a sparse QR factorization for the core of MATLAB. Instead, linear least squares problems of the form

$$\min \|b - Ax\|$$

are solved via the augmented system of equations

$$\begin{aligned} r + Ax &= b \\ A^T r &= 0. \end{aligned}$$

Introducing a residual scaling parameter α this can be written

$$\begin{pmatrix} \alpha I & A \\ A^T & 0 \end{pmatrix} \begin{pmatrix} r/\alpha \\ x \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}.$$

The augmented matrix, which inherits any sparsity in A, is symmetric, but clearly not positive definite. We ignore the symmetry and solve the linear system with a general sparse LU factorization, although a symmetric, indefinite factorization might be twice as fast.

A recent note by Björck [3] analyzes the choice of the parameter α by bounding the effect of roundoff errors on the error in the computed solution x. The value of α which minimizes the bound involves two quantities, $\|r\|$ and the smallest singular value of A, which are too expensive to compute. Instead, we use an apparently satisfactory substitute,

$$\alpha = \max|a_{ij}|/1000.$$

This approach has been used by several other authors, including Arioli et al. [1], who do use a symmetric factorization and a similar heuristic for choosing α.

It is not clear whether augmented matrices, orthogonal factorizations, or iterative methods are preferable for least squares problems, from either an efficiency or an accuracy point of view. We have chosen the augmented matrix approach because it is competitive with the other approaches, and because we could use exisiting code.

3.6. Eigenvalues of sparse matrices. We expect that most eigenvalue computations involving sparse matrices will be done with iterative methods of Lanczos and Arnoldi type, implemented outside the core of MATLAB as m-files. The most time-consuming portion will be the computation of Ax for sparse A and dense x, which can be done efficiently using our core operations.

However, we do provide one almost-direct technique for computing all the eigenvalues (but not the eigenvectors) of a real symmetric or complex Hermitian sparse matrix. The reverse Cuthill-McKee algorithm is first used to provide a permutation which reduces the bandwidth. Then an algorithm of Schwartz [22] provides a sequence of plane rotations which further reduces the bandwidth to tridiagonal. Finally, the symmetric tridiagonal QR algorithm from dense MATLAB yields all the eigenvalues.

4. Examples. This section gives the flavor of sparse MATLAB by presenting several examples. First, we show the effect of reorderings for sparse factorization by illustrating a Cholesky factorization with several different permutations. Then we give two examples of m-files, which are programs written in the MATLAB language to provide functionality that is not implemented in the "core" of MATLAB. These sample m-files are simplified somewhat for the purposes of presentation. They omit some of the error-checking that would be present in real implementations, and they could be written to contain more flexible options than they do.

4.1. Effect of permutations on Cholesky factors. This sequence of examples illustrates the effect of reorderings on the computation of the Cholesky factorization of one symmetric test matrix. The matrix is $S = WW^T$ where W is the Harwell-Boeing matrix `WEST0479` [6], a model due to Westerberg of an eight-stage chemical distillation column.

There are four figures. Each figure shows two `spy` plots, first a particular symmetric permutation of S and then the Cholesky factor of the permuted matrix. The

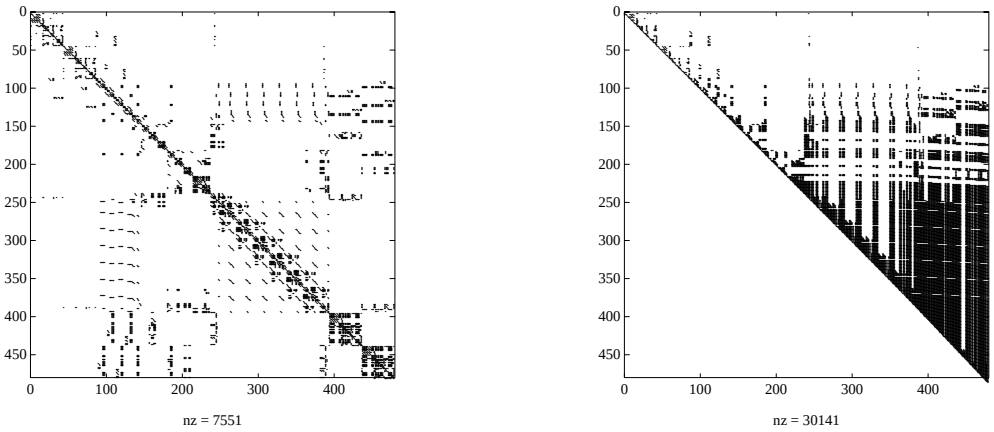


FIG. 4. *The structure of S and its Cholesky factor.*

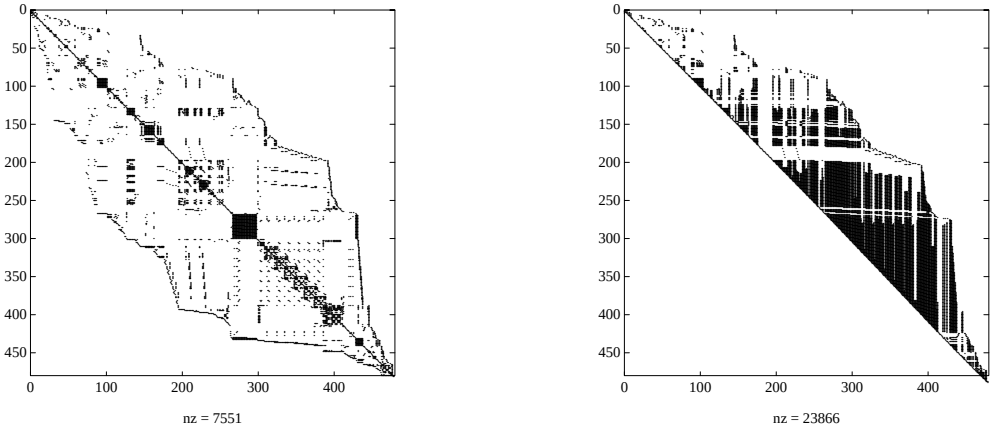


FIG. 5. *Matrix S and its Cholesky factor after reverse Cuthill-McKee reordering.*

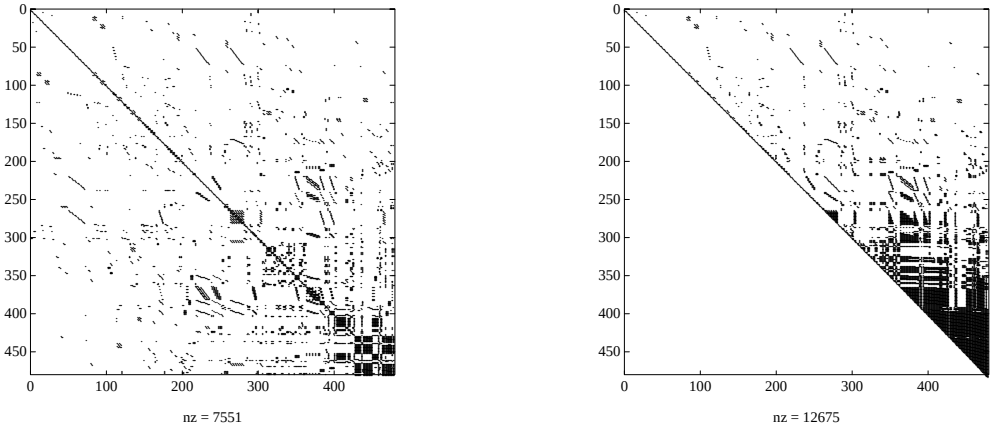


FIG. 6. *Matrix S and its Cholesky factor after column count reordering.*

TABLE 2
Effect of permutations on Cholesky factorization.

	nnz	time
Original order	30141	5.64
Reverse Cuthill-McKee	23866	4.26
Column count	12675	1.91
Minimum degree	12064	1.75

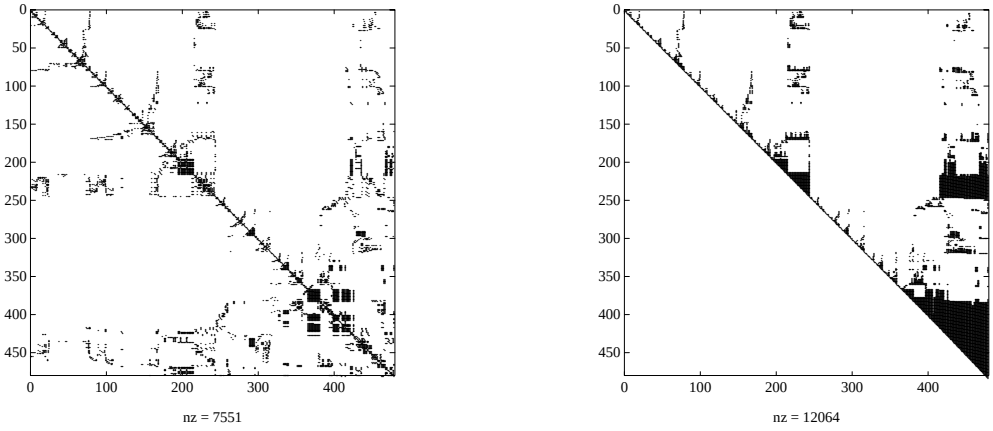


FIG. 7. *Matrix S and its Cholesky factor after minimum degree reordering.*

first figure is the original ordering; the second uses symmetric reverse Cuthill-McKee, `symrcm`; the third uses the column count permutation, `colperm`; the fourth uses symmetric minimum degree, `symmmd`. Each of the `spy` plots shows a matrix profile that is typical for the underlying permutation: Cuthill-McKee shows an envelope; column count shows all the mass in the later rows and columns; and minimum degree shows a recursive pattern curiously similar to divide-and-conquer orderings like nested dissection.

The matrix S is of order 479 and has 7551 nonzeros. Table 2 shows the number of nonzeros and the execution time in seconds (on a Sun SPARCstation-1) required to compute the Cholesky factors for each of the permutations. The behavior of `symrcm` and `symmmd` is typical; both produce significant reductions in nnz and in the execution time. The behavior of `colperm` is less typical; its reductions are not usually this significant.

4.2. The conjugate gradient method. Iterative techniques like the conjugate gradient method are often attractive for solving large sparse systems of linear equations. Figure 8 is an m-file for a conjugate gradient method. The code is somewhat simplified—a real code might use a more complicated criterion for termination, might compute Ap in a subroutine call in case A is not held explicitly, and might provide for preconditioning—but it illustrates an important point. Sparsity is never mentioned explicitly in the code. If the argument A is sparse then `Ap = A*p` will be computed as a sparse operation; if A is full then all the operations will be full.

In contrast with sparse direct methods, most iterative methods operate on matri-

```
function x = cgsolve (A,b,tol)

% Solve A*x = b by the conjugate gradient method.
% Iterate until norm(A*x-b) / norm(b) <= tol.

    x = zeros(size(b));
    r = b;
    rtr = r'*r;
    p = zeros(size(b));
    beta = 0;
    while ( norm(r) > tol * norm(b) )
        p = r + beta * p;
        Ap = A * p;
        alpha = rtr / ( p' * Ap );
        x = x + alpha * p;
        r = r - alpha * Ap;
        rtrold = rtr;
        rtr = r'*r;
        beta = rtr / rtrold;
    end
```

FIG. 8. *Solving $Ax = b$ by conjugate gradients.*

ces and vectors at a high level, typically using the coefficient matrix only in matrix-vector multiplications. This is the reason for our decision not to build an iterative linear solver into the core of MATLAB; such solvers can be more easily and flexibly written as m-files that make use of the basic sparse operations.

4.3. Solving reducible systems. If A is a reducible matrix, the linear system $Ax = b$ can be solved by permuting A to block upper triangular form (with irreducible diagonal blocks) and then performing block back-substitution. Only the diagonal blocks of the permuted matrix need to be factored, saving fill and arithmetic in the above-diagonal blocks. This strategy is incorporated in some existing Fortran sparse matrix packages, most notably Duff and Reid's code `MA28` in the Harwell Subroutine Library [7]. Figure 9 is an implementation as a MATLAB m-file. This function is a good illustration of the use of permutation vectors.

The call `[p,q,r] = dmperm(A)` returns a row permutation p and a column permutation q to put A in block triangular form. The third output argument r is an integer vector describing the boundaries of the blocks: the k-th block of $A(p,q)$ includes indices from $r(k)$ to $r(k+1)-1$. The loop has one iteration for each diagonal block; note that i and j are vectors of indices. The code resembles an ordinary triangular backsolve, but at each iteration the statement `x(j) = A(j,j) \ x(j)` solves for an entire block of x at once by sparse LU decomposition (with column minimum degree ordering) of one of the irreducible diagonal blocks of A.

Again this code is simplified a bit. A real code would merge every sequence of adjacent 1×1 diagonal blocks into a single triangular block, thus reducing the number of iterations of the main loop.

```
function x = dmsolve (A,b)

% Solve A*x = b by permuting A to block
% upper triangular form and then performing
% block back substitution.

    % Permute A to block form.
    [p,q,r] = dmperm(A);
    nblocks = length(r)-1;
    A = A(p,q);
    x = b(p);

    % Block backsolve.
    for k = nblocks : -1 : 1

        % Indices above the k-th block.
        i = 1 : r(k)-1;

        % Indices of the k-th block.
        j = r(k) : r(k+1)-1;

        x(j) = A(j,j) \ x(j);
        x(i) = x(i) - A(i,j) * x(j);

    end;

    % Undo the permutation of x.
    x(q) = x;
```

FIG. 9. *Solving $Ax = b$ by block triangular back-substitution.*

REFERENCES

[1] M. ARIOLI, I. S. DUFF, AND P. P. M. DE RIJK, *On the augmented system approach to least-squares problems*, Numerische Mathematik, 55 (1989), pp. 667–684.

[2] C. ASHCRAFT, R. GRIMES, J. LEWIS, B. PEYTON, AND H. SIMON, *Recent progress in sparse matrix methods for large linear systems*, International Journal of Supercomputer Applications, (1987), pp. 10–30.

[3] A. BJÖRCK, *A note on scaling in the augmented system methods (unpublished manuscript)*, 1991.

[4] T. F. COLEMAN, A. EDENBRANDT, AND J. R. GILBERT, *Predicting fill for sparse orthogonal factorization*, Journal of the Association for Computing Machinery, 33 (1986), pp. 517–532.

[5] J. DONGARRA, J. BUNCH, C. MOLER, AND G. STEWART, *LINPACK Users Guide*, Philadelphia, PA, 1978.

[6] I. S. DUFF, R. G. GRIMES, AND J. G. LEWIS, *Sparse matrix test problems*, ACM Transactions on Mathematical Software, 15 (1989), pp. 1–14.

[7] I. S. DUFF AND J. K. REID, *Some design features of a sparse matrix code*, ACM Transactions on Mathematical Software, 5 (1979), pp. 18–35.

[8] ———, *The multifrontal solution of indefinite sparse symmetric linear equations*, ACM Transactions on Mathematical Software, 9 (1983), pp. 302–325.

[9] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman, *Algorithms and data structures for sparse symmetric Gaussian elimination*, SIAM Journal on Scientific and Statistical Computing, 2 (1981), pp. 225–237.

[10] A. George and J. Liu, *The evolution of the minimum degree ordering algorithm*, SIAM Review, 31 (1989), pp. 1–19.

[11] A. George and J. W. H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, 1981.

[12] J. R. Gilbert, *Predicting structure in sparse matrix computations*, Tech. Report 86–750, Cornell University, 1986. To appear in *SIAM Journal on Matrix Analysis and Applications*.

[13] J. R. Gilbert, C. Lewis, and R. Schreiber, *Parallel preordering for sparse matrix factorization*. In preparation.

[14] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in Matlab: Design and implementation*, Tech. Report CSL 91–4, Xerox Palo Alto Research Center, 1991.

[15] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 862–874.

[16] J. W. H. Liu, *The role of elimination trees in sparse factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 134–172.

[17] J. W. H. Liu, E. Ng, and B. W. Peyton, *On finding supernodes for sparse matrix computations*, Tech. Report ORNL/TM-11563, Oak Ridge National Laboratory, 1990.

[18] The MathWorks, *Pro-Matlab User's Guide*, South Natick, MA, 1990.

[19] C. Moler, *Matrix computations with Fortran and paging*, Communications of the ACM, 15 (1972), pp. 268–270.

[20] A. Pothen and C.-J. Fan, *Computing the block triangular form of a sparse matrix*, ACM Transactions on Mathematical Software, 16 (1990), pp. 303–324.

[21] R. Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Transactions on Mathematical Software, 8 (1982), pp. 256–276.

[22] H. Schwartz, *Tridiagonalization of a symmetric band matrix*, Numer. Math., 12 (1968), pp. 231–241. Also in [26, pages 273–283].

[23] B. Smith, J. Boyle, Y. Ikebe, V. Klema, and C. Moler, *Matrix Eigensystem Routines: EISPACK Guide*, Springer-Verlag, New York, NY, second ed., 1970.

[24] B. Speelpenning, *The generalized element method*, Tech. Report UIUCDCS–R–78–946, University of Illinois, 1978.

[25] United Kingdom Atomic Energy Authority, *Harwell subroutine library: A catalogue of subroutines*, Tech. Report AERE R 9185, Harwell Laboratory, Oxfordshire OX11 0RA, Great Britain, 1988.

[26] J. Wilkinson and C. Reinsch, eds., *Linear Algebra*, vol. 2 of Handbook for Automatic Computation, Springer-Verlag, New York, NY, 1971.

THE MATLAB ODE SUITE

LAWRENCE F. SHAMPINE* AND MARK W. REICHELT†

Abstract. This paper describes mathematical and software developments for a suite of programs for solving ordinary differential equations in MATLAB.

Key words. ordinary differential equations, stiff systems, BDF, Gear method, Rosenbrock method, non-stiff systems, Runge-Kutta method, Adams method, software

AMS subject classifications. 65L06, 65L05, 65Y99, 34A65

1. Introduction. This paper presents mathematical and software developments that are the basis for a suite of programs for the solution of initial value problems

$$y' = F(t, y)$$

on a time interval $[t_0, t_f]$, given initial values $y(t_0) = y_0$. The solvers for stiff problems allow the more general form

$$M(t)\, y' = f(t, y)$$

with a mass matrix $M(t)$ that is non-singular and (usually) sparse. The programs have been developed for MATLAB [29], a widely used environment for scientific computing. This influenced the choice of methods and how they were implemented.

As in many environments, the typical problem in MATLAB is solved interactively and the results displayed graphically. Generally functions defining the differential equations are not expensive to evaluate. The typical stiff problem is either of modest size or has a highly structured Jacobian. In MATLAB, linear algebra and the built-in array operations are relatively fast, and the language provides for sparse arrays. MATLAB handles storage dynamically and retains copies of arrays.

A new family of formulas for the solution of stiff problems called the numerical differentiation formulas, NDF's, are devised in §2. They are more efficient than the backward differentiation formulas, BDF's, though a couple of the higher order formulas are somewhat less stable. These formulas are conveniently implemented with backward differences. A way of changing step size in this representation is developed that is both compact and efficient in MATLAB. In §3 we devise a new linearly implicit one-step method for solving stiff systems, specifically a modified Rosenbrock method, and also a continuous extension of the method. §4 describes briefly how to modify these methods so as to solve conveniently and efficiently problems involving a mass matrix. In §5, we discuss briefly three methods for non-stiff problems. The two based on explicit Runge-Kutta methods are more efficient than those previously available in MATLAB and have free interpolants.

MATLAB has features, some of which are available in C and FORTRAN 90, that make possible an interesting and powerful user interface. §6 explains how the language was exploited to devise an interface that is unobtrusive, powerful, and extendable. Using the scheme of §7 for the numerical approximation of Jacobians, the design makes it possible for all the codes of the suite to be used in *exactly* the same manner. Options

* Mathematics Department, Southern Methodist University, Dallas, TX 75275. (shampine@na-net.ornl.gov)

† The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760. (mwr@mathworks.com)

allow users to supply more information that makes the solution of stiff problems more reliable and/or efficient. In particular, it is easy to exploit sparse Jacobians.

Examples in §8 show the relative performance of codes in the suite and the value of some of the options. The availability of the codes is described in §9.

2. Implicit Formulas for Stiff Systems. The BDF's are very popular for solving stiff problems. When the step size is a constant h and backward differences are used, the formula of order k, BDFk, for a step from (t_n, y_n) to (t_{n+1}, y_{n+1}) is

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m y_{n+1} - hF(t_{n+1}, y_{n+1}) = 0 \tag{1}$$

The algebraic equation for y_{n+1} is solved with a simplified Newton (chord) iteration. The iteration is started with the predicted value

$$y_{n+1}^{(0)} = \sum_{m=0}^{k} \nabla^m y_n \tag{2}$$

The leading term of the BDFk truncation error can be conveniently approximated as

$$\frac{1}{k+1} h^{k+1} y^{(k+1)} \approx \frac{1}{k+1} \nabla^{k+1} y_{n+1}. \tag{3}$$

The typical implementation of a general-purpose BDF code is quasi-constant step size. This means that the formulas used are those for a constant step size and the step size is held constant during an integration unless there is good reason to change it. General-purpose BDF codes also vary the order during an integration.

2.1. The Numerical Differentiation Formulas. Noting that the predictor (2) has a longer memory than (1), Klopfenstein [25] and Reiher [31] considered how to exploit this to obtain better stability. Klopfenstein studied methods of the form

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m y_{n+1} - hF(t_{n+1}, y_{n+1}) - \kappa\gamma_k \left(y_{n+1} - y_{n+1}^{(0)} \right) = 0 \tag{4}$$

that he called numerical differentiation formulas, NDF's. Here κ is a scalar parameter and the coefficients γ_k are given by $\gamma_k = \sum_{j=1}^{k} \frac{1}{j}$. The role of the term added to BDFk is illuminated by the identity

$$y_{n+1} - y_{n+1}^{(0)} = \nabla^{k+1} y_{n+1}$$

and the approximation (3) to the truncation error of BDFk. It follows easily that for any value of the parameter κ, the method is of order (at least) k and the leading term of its truncation error is

$$\left(\kappa\gamma_k + \frac{1}{k+1} \right) h^{k+1} y^{(k+1)} \tag{5}$$

For orders 3-6, Klopfenstein and Reiher found numerically the κ that maximizes the angle of A(α)-stability. Because BDF2 is already A-stable, Klopfenstein considered how to choose κ so as to reduce the truncation error as much as possible whilst still retaining A-stability. The optimal choice is $\kappa = -1/9$, yielding a truncation error

TABLE 1
The Klopfenstein-Shampine NDF's and their efficiency and A(α)-stability relative to the BDF's.

order	NDF coeff	step ratio	stability angle		percent
k	κ	percent	BDF	NDF	change
1	-0.1850	26%	90°	90°	0%
2	-1/9	26%	90°	90°	0%
3	-0.0823	26%	86°	80°	-7%
4	-0.0415	12%	73°	66°	-10%
5	0	0%	51°	51°	0%

coefficient half that of BDF2. This implies that for sufficiently small step sizes, NDF2 can achieve the same accuracy as BDF2 with a step size about 26% bigger.

The formulas derived by Klopfenstein and Reiher at orders higher than 2 are less successful because the price of improved stability is reduced efficiency. Taking the opposite tack, we sought values of κ that would make the NDF's more accurate than the BDF's and not much less stable. Of course the leading term of the truncation error cannot be made too small, else it would not dominate and the formula would not behave as expected at realistic step sizes. Because Klopfenstein's second order formula optimally improves accuracy while retaining L-stability, it serves as the order 2 method of our NDF family. Correspondingly, we sought to obtain the same improvement in efficiency (26%) at orders 3-5. This comes at the price of reduced stability and we were not willing to reduce the stability angle by more than 10%. The search was carried out numerically. Our choices and the compromises made in balancing efficiency and stability are shown in Table 1. The stability of BDF5 is so poor that we were not willing to reduce it at all.

The first-order formula NDF1 has the form

$$y_{n+1} - y_n - \kappa\,(y_{n+1} - 2y_n + y_{n-1}) = hF(t_{n+1}, y_{n+1})$$

The boundary of the stability region of a linear multistep method consists of those points z for which the characteristic equation $\rho(\theta) - z\sigma(\theta) = 0$ has a root θ of magnitude 1. The root-locus method obtains the boundary as a subset of $z = \rho(\theta)/\sigma(\theta)$ as $\theta = \exp(i\psi)$ ranges over all numbers of magnitude 1. For NDF1 it is found that

$$Re(z) = 1 - (1 - 2\kappa)\cos(\psi) - 2\kappa\cos^2(\psi)$$

and that a sufficient condition for the formula to be A-stable is $1 - 2\kappa \geq 0$. As at other orders, we chose an improvement in efficiency of 26%, leading to $\kappa = -0.1850$.

2.2. Changing the Step Size. Backward differences are very suitable for implementing the NDF's in MATLAB because the basic algorithms can be coded compactly and efficiently. We develop here a way of changing step size that is also well-suited to the language.

When the integration reaches t_n, there are available solution values $y(t_{n-j})$ at $t_{n-j} = t_n - jh$ for $j = 0, 1, ..., k$. The interpolating polynomial is

$$P(t) = y(t_n) + \sum_{j=1}^{k} \nabla^j y(t_n) \frac{1}{j!h^j} \prod_{m=0}^{j-1} (t - t_{n-m})\,.$$

By definition $\nabla^j P(t_n) = \nabla^j y(t_n)$. In this representation the solution is held in the form of the current value $y(t_n)$ and a table of backward differences

$$D = \left[\nabla P(t_n), \nabla^2 P(t_n), ..., \nabla^k P(t_n)\right].$$

Changing to a new step size $h^* \neq h$ amounts to evaluating $P(t)$ at $t^* = t_n - jh^*$ for $j = 0, 1, ..., k$ and then forming

$$D^* = \left[\nabla^* P(t_n), \nabla^{*2} P(t_n), ..., \nabla^{*k} P(t_n)\right].$$

Here the asterisk on the backward difference ∇^* indicates that it is for step size h^*.

Equating the two representations of $P(t)$ leads to the identity

$$\sum_{j=1}^{k} \nabla^{*j} P(t_n) \frac{1}{j! h^{*j}} \prod_{m=0}^{j-1} \left(t - t^*_{n-m}\right) = \sum_{j=1}^{k} \nabla^j P(t_n) \frac{1}{j! h^j} \prod_{m=0}^{j-1} \left(t - t_{n-m}\right).$$

Evaluating this identity at $t = t^*_{n-r}$ for $r = 1, ..., k$ leads to the system of equations

$$\sum_{j=1}^{k} \nabla^{*j} P(t_n) U_{jr} = \sum_{j=1}^{k} \nabla^j P(t_n) R_{jr}$$

which is in matrix terms $D^* U = DR$. The entries of the $k \times k$ matrix U are

$$U_{jr} = \frac{1}{j! h^{*j}} \prod_{m=0}^{j-1} \left(t^*_{n-r} - t^*_{n-m}\right) = \frac{1}{j!} \prod_{m=0}^{j-1} (m - r).$$

Matrix U satisfies $U^2 = I$. This implies that $D^* = D\,(RU)$, the scheme we use for changing the step size. The entries of U are integers that do not depend on h nor on k. In terms of $\rho = h^*/h \neq 1$, the entries of the $k \times k$ matrix R are

$$R_{jr} = \frac{1}{j!} \prod_{m=0}^{j-1} (m - r\rho)$$

R must be formed each time the step size is changed. This is done in a single line of MATLAB code using the cumprod function. Likewise, changing the representation by means of matrix multiplication is done in a single line. Accordingly, in MATLAB this way of changing the step size is both compact and efficient.

2.3. The `ode15s` Program. The code `ode15s` is a quasi-constant step size implementation of the NDF's in terms of backward differences. Options allow integration with the BDF's and integration with a maximum order less than the default of 5.

The identity

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m y_{n+1} = \gamma_k \left(y_{n+1} - y^{(0)}_{n+1}\right) + \sum_{m=1}^{k} \gamma_m \nabla^m y_n$$

shows that equation (4) is equivalent to

$$(1 - \kappa)\,\gamma_k \left(y_{n+1} - y^{(0)}_{n+1}\right) + \sum_{m=1}^{k} \gamma_m \nabla^m y_n - hF(t_{n+1}, y_{n+1}) = 0.$$

In the simplified Newton iteration, the correction to the current iterate

$$y_{n+1}^{(i+1)} = y_{n+1}^{(i)} + \Delta^{(i)}$$

is obtained by solving

$$\left(I - \frac{h}{(1-\kappa)\,\gamma_k} J\right) \Delta^{(i)} = \frac{h}{(1-\kappa)\,\gamma_k} F(t_{n+1}, y_{n+1}^{(i)}) - \Psi - \left(y_{n+1}^{(i)} - y_{n+1}^{(0)}\right).$$

Here J is an approximation to the Jacobian of $F(t,y)$ and

$$\Psi = \frac{1}{(1-\kappa)\,\gamma_k} \sum_{m=1}^{k} \gamma_m \nabla^m y_n$$

is a quantity that is fixed during the computation of y_{n+1}. Scaling the equation to remove the scalar multiplying J offers some advantages [36]. It is much more accurate to obtain the fundamental quantity $\nabla^{k+1} y_{n+1}$ as the limit of

$$d^{(i)} = y_{n+1}^{(i)} - y_{n+1}^{(0)}$$

computed from

$$\begin{aligned} d^{(i+1)} &= d^{(i)} + \Delta^{(i)} \\ y_{n+1}^{(i+1)} &= y_{n+1}^{(0)} + d^{(i+1)} \end{aligned}$$

Many of the tactics adopted in the code resemble those found in the well-known codes DIFSUB [17], DDRIV2 [24], LSODE [22], and VODE [7]. In particular, local extrapolation is not done. The selection of the initial step size follows Curtis [10] who observes that by forming partial derivatives of $F(t,y)$ at t_0, it is possible to estimate the *optimal* initial step size.

In the context of MATLAB it is natural to retain a copy of the Jacobian matrix. Of the codes cited, only VODE exploits this possibility. It is also natural to form and factor the iteration matrix every time the step size or order is changed. The rate of convergence is monitored [34] and the iteration terminated if it is predicted that convergence will not be achieved in four iterations. Should this happen and the Jacobian not be current, a new Jacobian is formed. Otherwise the step size is reduced.

Our scheme for reusing Jacobians means that when the Jacobian is constant, `ode15s` will normally form a Jacobian just once in the whole integration. Also, the code will form very few Jacobians when applied to a problem that is not stiff. `ode15s` competes rather well with the codes for non-stiff problems because of this and the efficient linear algebra of MATLAB.

3. Linearly Implicit Formulas for Stiff Systems. In this section it is convenient at first to consider differential equations in autonomous form, $y' = F(y)$. Rosenbrock methods have the form

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i,$$

where the k_i are obtained for $i = 1, 2, ..., s$ by solving

$$W k_i = F\left(y_n + h \sum_{j=1}^{i-1} a_{ij} k_j\right) + hJ \sum_{j=1}^{i-1} d_{ij} k_j.$$

Here $W = I - hdJ$ and $J = \partial F(y_n)/\partial y$. Such methods are said to be linearly implicit because the computation of y_{n+1} requires the solution of systems of linear equations.

A number of authors have explored formulas of this form with J that only approximate $\partial F(y_n)/\partial y$. A second order formula due to Wolfbrandt [40] is

$$\begin{aligned} Wk_1 &= F\left(y_n\right) \\ Wk_2 &= F\left(y_n + \frac{2}{3}hk_1\right) - \frac{4}{3}hdJk_1 \\ y_{n+1} &= y_n + \frac{h}{4}\left(k_1 + 3k_2\right) \end{aligned}$$

Here the parameter $d = 1/\left(2+\sqrt{2}\right)$. It is characteristic of such W-formulas that the order of the formula does not depend on J. The stability does depend on J, and if $J = \partial F/\partial y$, the formula is L-stable.

One of the attractions of Rosenbrock and W methods is that they are one-step. However, it is not easy to retain this property when estimating the error, and the error estimate of [40] for Wolfbrandt's formula gives it up. Scraton [33] achieved a one-step error estimate without additional evaluations of F by assuming that

$$J = \frac{\partial F}{\partial y}\left(t_n, y_n\right) + hB + O\left(h^2\right)$$

We must assume that $J \approx \partial F/\partial y$ if we are to apply the usual linear stability theory to conclude the good stability that leads us to consider such formulas in the first place. We describe formulas based on Scraton's assumption as modified Rosenbrock formulas. Scraton's error estimate is inefficient for very stiff problems because the companion formula of order 3 is not stable at infinity. Zedan [43], [42] derived a companion that is A-stable and also requires no additional evaluations of F.

3.1. A Modified Rosenbrock Triple. In a step from t_n to t_{n+1} with Wolfbrandt's formula and any of the error estimates cited, F is evaluated only at t_n and $t_n + 2h/3$. It is possible that if a solution component changes sharply somewhere in $(t_n + 2h/3, t_n + h)$, a poor approximation to this component at t_{n+1} will be produced and accepted. Very sharp changes in a solution component are common when solving stiff problems and robust software ought to be able to deal with them. For this reason, many authors attempt to derive pairs that evaluate at both ends of a step.

Wolfbrandt's formula is a member of a family of second order, L-stable W methods [40] that involve two evaluations of F. By making the *first* evaluation of F for the next step the *same as* the *last* of the current step (FSAL), we have at our disposal a function evaluation that is usually "free" because most steps are successful. Exploiting this we derived a pair of formulas that we present in a form that avoids unnecessary matrix multiplications [36]. When advancing a step from (t_n, y_n) to $t_{n+1} = t_n + h$, the modified Rosenbrock pair is

$$\begin{aligned} F_0 &= F\left(t_n, y_n\right) \\ k_1 &= W^{-1}\left(F_0 + hdT\right) \\ F_1 &= F\left(t_n + 0.5h, y_n + 0.5hk_1\right) \\ k_2 &= W^{-1}\left(F_1 - k_1\right) + k_1 \\ y_{n+1} &= y_n + hk_2 \\ F_2 &= F\left(t_{n+1}, y_{n+1}\right) \end{aligned}$$

$$k_3 = W^{-1}\left[F_2 - e_{32}\left(k_2 - F_1\right) - 2\left(k_1 - F_0\right) + hdT\right]$$
$$error \approx \frac{h}{6}\left(k_1 - 2k_2 + k_3\right)$$

Here $W = I - hdJ$ with $d = 1/\left(2+\sqrt{2}\right)$ and $e_{32} = 6+\sqrt{2}$,

$$J \approx \frac{\partial F}{\partial y}\left(t_n, y_n\right) \quad \text{and} \quad T \approx \frac{\partial F}{\partial t}\left(t_n, y_n\right).$$

If the step is a success, the F_2 of the current step is the F_0 of the next. If $J = \partial F/\partial y$, the second order formula is L-stable. Because the pair samples at both ends of the step, it has a chance of recognizing very sharp changes that occur within the span of a step.

On p. 129 of his dissertation, Zedan describes briefly a scheme for interpolating his modified Rosenbrock method. On $[t_n, t_n + h]$ he approximates the solution with the quadratic polynomial interpolating y_n, y_{n+1}, and $F_0 = F\left(t_n, y_n\right)$. Interpolating F_0 can be unsatisfactory when the problem is very stiff. A standard way to obtain a continous extension to a one-step method is to derive a family of formulas, each of which provides an approximation at $t_n + h^*$ for a given $h^* = sh$. For any s, it is easy to derive a formula that has the same form as that of the basic step to t_{n+1}; the trick is to reuse the computations of the basic step as much as possible. In the new formula the matrix $W^* = I - h^*d^*J$, so if we take the parameter $d^* = d/s$, then $W^* = W$. It is easily found that with similar definitions, it is possible to obtain a second order W method for the intermediate value that requires no major computations not already available from the basic step itself. Specifically, it is found that a second order approximation to $y\left(t_n + sh\right)$ is provided by

$$y_{n+s} = y_n + h\left[\frac{s\left(1-s\right)}{1-2d}k_1 + \frac{s\left(s-2d\right)}{1-2d}k_2\right].$$

Though derived as a family of formulas depending on a parameter s, the continuous extension turns out to be a quadratic polynomial in s. It interpolates both y_n and y_{n+1}, hence connects continuously with approximations on adjacent steps. This interpolant behaves better for stiff problems than the one depending on F_0, in essence because of the W^{-1} implicit in the k_i.

3.2. The `ode23s` Program. The code `ode23s` based on the scheme derived here provides an alternative to `ode15s` for the solution of stiff problems. It is especially effective at crude tolerances, when a one-step method has advantages over methods with memory, and when Jacobians have eigenvalues near the imaginary axis. It is a fixed order method of such simple structure that the overhead is low except for the linear algebra, which is relatively fast in MATLAB. The integration is advanced with the lower order formula, so `ode23s` does not do local extrapolation. To achieve the same L-stability in `ode15s`, the maximum order would have to be restricted to 2.

We have considered algorithms along the lines of [11] for recognizing when a new Jacobian is needed and we have also considered tactics like those of [40] and [42] for this purpose. This is promising and we may revisit the matter, but the current version of `ode23s` forms a new Jacobian at every step for several reasons. A formula of order 2 is most appropriate at crude tolerances. At such tolerances solution components often change significantly in the course of a single step, so it is often appropriate to form a new Jacobian. In MATLAB the Jacobian is typically of modest size or sparse and its

evaluation is not very expensive compared with evaluating F. Lastly, evaluating the Jacobian at every step enhances the reliability and robustness of the code.

For non-autonomous problems `ode23s` requires an approximation to $\partial F/\partial t$ in addition to the approximation to $\partial F/\partial y$. For the convenience of the user and to make the use of all the codes the same, we have chosen always to approximate this partial derivative numerically.

4. Stiff Systems of More General Form. A stiff problem $M(t)\,y' = f(t,y)$ with a non-singular mass matrix $M(t)$ can always be solved by transforming the equation to the equivalent form $y' = F(t,y) = M^{-1}(t)\,f(t,y)$. Small modifications to the methods avoid the inconvenience and expense of this transformation.

Because the modified Rosenbrock method was derived for autonomous systems, it is awkward to accomodate matrices M that depend on t. Accordingly, we allow only constant mass matrices in `ode23s`. The modification is derived by applying the method to the transformed equation and then rewriting the computations in terms of M and $f(t,y)$. The first stage k_1 is obtained from

$$W k_1 = (I - hdJ)k_1 = F_0 + hdT$$

Here $F_0 = M^{-1}f(t_0,y_0) = M^{-1}f_0$,

$$J \approx \frac{\partial F}{\partial y} = M^{-1}\frac{\partial f}{\partial y} \quad \text{and} \quad T \approx \frac{\partial F}{\partial t} = M^{-1}\frac{\partial f}{\partial t}.$$

Scaling the equation for k_1 by M leads to

$$\left(M - hd\frac{\partial f}{\partial y}\right) k_1 = f_0 + hd\frac{\partial f}{\partial t}$$

The remaining computations are treated similarly. The usual form of the method is recovered when $M = I$. This modification of the method allows the user to pose the problem in terms of M and $f(t,y)$. It avoids the solution of linear equations that arise when the equation is transformed.

The `ode15s` code allows the mass matrix to depend on t. This causes only one difference in the modification of the methods of this code. The simplified Newton method involves solution of linear systems with the iteration matrix

$$I - \frac{h}{(1-\kappa)\gamma_k}J$$

and right hand sides involving

$$F(t_{n+1}, y_{n+1}^{(i)}) = M^{-1}(t_{n+1})f(t_{n+1}, y_{n+1}^{(i)})$$

Here

$$J \approx \frac{\partial F}{\partial y}\,(t_m, y_m) = M^{-1}(t_m)\frac{\partial f}{\partial y}\,(t_m, y_m)$$

for some $m \le n$. Because M depends on t, it is not possible to remove all the inverses simply by scaling with $M(t_{n+1})$. We approximate the scaled iteration matrix by

$$M(t_m) - \frac{h}{(1-\kappa)\gamma_k}J \quad \text{where} \quad J \approx \frac{\partial f}{\partial y}\,(t_m, y_m)\,.$$

With this approximation, the computations can be written in terms of $M(t)$ and $f(t,y)$. The modified method reduces to the usual one when $M(t) = I$.

5. Explicit Formulas for Non-Stiff Systems. The two explicit Runge-Kutta codes, `ode23` and `ode45`, in previous versions of MATLAB have been replaced by codes with the same names that remedy some deficiencies in design and take advantage of developments in the theory and practice of Runge-Kutta methods. A new code, `ode113`, is a PECE implementation of Adams-Bashforth-Moulton methods.

The new `ode23` is based on the Bogacki-Shampine $(2, 3)$ pair [3] (see also [37]) and the new `ode45` is based on the Dormand-Prince $(4, 5)$ pair [12]. Workers in the field employ a number of quantitative measures for evaluating the quality of a pair of formulas. In the standard measures these pairs are of high quality and significantly more efficient than those used in the earlier codes. Both pairs are FSAL and constructed for local extrapolation.

Because solution components can change substantially in the course of a single step, the values computed at the end of each natural step may not provide adequate resolution of the solution for plotting. This phenomenon is exacerbated when plotting in the phase plane. A good way to deal with this is to form additional values by means of a continuous extension (interpolant). A continuous extension also makes possible event location, a valuable capability in ODE codes. We selected pairs for which continuous extensions were available. In the case of the $(2, 3)$ pair, accurate solution values can be obtained for "free" (no additional evaluations of F) by cubic Hermite interpolation to the values and slopes computed at the ends of the step. Dormand and Prince obtained a number of inexpensive interpolants for their pair in [13]; they communicated to us another interpolant of order 4 that is of high quality and "free". The default in `ode45` is to use this interpolant to compute solution values at four points spaced evenly within the span of each natural step. Figure 1 shows the importance of this development. We return to this issue in §6.

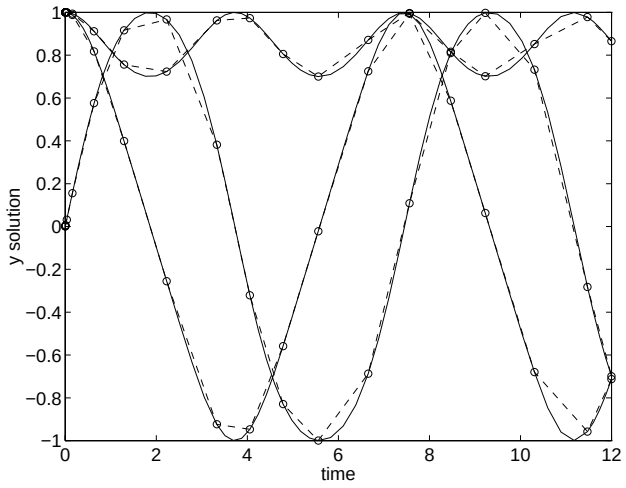


FIG. 1. *The* `rigid` *solution as computed by* `ode45`*. The continuous curves result from the default output of* `ode45`*. The discrete values are the results at the end of each step. The dashed curves show what happens when the new interpolation capability is disabled.*

`ode113` is a descendant of ODE/STEP, INTRP [39]. Though the code differs considerably in detail, its basic algorithms follow closely those of STEP. In particular, it does local extrapolation. The authors of ODE/STEP, INTRP tried to obtain as cheaply and reliably as possible solutions of moderate to high accuracy to problems

involving functions F that are expensive to evaluate. This was accomplished by monitoring the integration very closely and by providing formulas of quite high orders. In the present context the overhead of this monitoring is comparatively expensive. Although more than graphical accuracy is necessary for adequate resolution of solutions of moderately unstable problems, the high order formulas available in `ode113` are not nearly as helpful in the present context as they are in general scientific computation.

6. User Interface. Every author of an ODE code wants to make it as easy as possible to use. At the same time the code must be able to solve typical problems. It is not easy to reconcile these goals. In the MATLAB environment we considered it essential that it be possible to use all the codes in *exactly* the same way. However, we also considered it essential to provide codes for the solution of stiff problems. Methods for stiff problems make use of Jacobians. If a code for solving stiff problems is to "look" like a code for non-stiff problems, it is necessary to approximate these Jacobians inside the code. Unfortunately, it is difficult to do this reliably. Moreover, when it is not inconvenient to supply some information about the structure of the Jacobian, it can be quite advantageous. Indeed, this information is crucial to the solution of "large" systems. Clearly a user interface to a code for stiff problems must allow for the provision of additional information and this without complication when users do not have the additional information or do not think it worth the trouble of supplying. The same is true of other optional information, so a key issue is how to make options unobtrusive. Further, the design must be extendable. Indeed, we have already extended the functionality of our original design three times.

It is possible to use all the codes in the suite in precisely the same manner, so in the examples that follow `ode15s` is generic. An initial value problem can be solved by

```
[t,y] = ode15s('ydot',tspan,y0);
```

The results can then be displayed with the usual plotting tools of MATLAB, e.g., by `plot(t,y)`. Here `ydot` is the name of a function that defines the differential equation. Figure 2 shows an example of the van der Pol equation coded as function `vdpnsode`. The interval of integration is `tspan=[t0,tfinal]` and the initial conditions are `y0`. The code obtains the number of equations by measuring the length of the vector `y0`.

```
function dy = vdpnsode(t,y)
dy = zeros(2,1);          % preallocate column vector dy
dy(1) = y(2);
dy(2) = (1-y(1)^2)*y(2)-y(1);
```

FIG. 2. *The* MATLAB *code for the initial value problem* `vdpnsode`.

The suite exploits the possibility of a variable number of arguments. For instance, the codes monitor several measures of cost, such as the number of steps and the number of Jacobian evaluations, and return them in an optional output argument, viz. `[t,y,stats]`. It also exploits the possibility of empty arrays. For example, it is possible to define the initial value problem in one file. Figure 3 illustrates this for the CHM6 [14] stiff test problem of [28]. With the function coded as shown, if `ode15s` is invoked with empty or missing arguments for `tspan` and `y0`, it will call `chm6ode` with an empty argument for `t` to obtain the information not supplied in the call list.

Returning the independent variable and approximate solution at each step in arrays `[t,y]` is natural in MATLAB because it facilitates plotting and the sizes of

```
function [out1,out2,out3] = chm6ode(t,y)
if isempty(t)             % return default tspan, y0, options
  out1 = [0; 1000];
  out2 = [761; 0; 600; 0.1];
  out3 = odeset('atol',1e-13);
  return;
end
dy = zeros(4,1);          % preallocate column vector dy
K = exp(20.7 - 1500/y(1));
dy(1) = 1.3*(y(3) - y(1)) + 10400*K*y(2);
dy(2) = 1880 * (y(4) - y(2) * (1+K));
dy(3) = 1752 - 269*y(3) + 267*y(1);
dy(4) = 0.1 + 320*y(2) - 321*y(4);
out1 = dy;
```

FIG. 3. *The* MATLAB *code for the initial value problem* `chm6ode`.

output arrays do not have to be specified in advance. The steps chosen by a code tend to cluster where solution components change rapidly, so this design generally results in satisfactory plots. However, this is not always the case, a point made by Polking [30] for the old `ode45` and illustrated by Figure 1 for the new. All the methods implemented in the suite have free interpolants that can be evaluated at additional points to obtain a smooth plot. In our design there is an option `refine` for specifying the number of answers to be computed at points equally spaced in the span of each step. By increasing `refine`, it is always possible to get a smooth plot. The additional output is obtained inexpensively via a continuous extension of the formula.

To deal with output at specific points, we overload the definition of `tspan` and use the length of this vector to dictate how its values are to be interpreted. An input `tspan` with two entries means that output at the natural steps is desired. If `tspan` contains more than two entries, the code is to produce output at the values of the independent variable specified in `tspan` and only these values. Because output is obtained by evaluating a polynomial, the number and placement of specified output points has little effect on the cost of the integration.

It is difficult to accomodate all the possibilities for optional input without complicating the interface to the point that users despair. A traditional approach is to use an options vector. We do this too, but with some innovations. The options vector is optional. When it is employed, the syntax of a call to the solver is

```
[t,y] = ode15s('ydot',tspan,y0,options);
```

The vector `options` is built by means of the function `odeset` that accepts name-value pairs. We make use of keywords and exploit the fact that in MATLAB we can specify values for the options that are of different data types. Indeed, an option can have more than one data type as a value. `odeset` allows options to be set in any order and default values are used for any quantity not explicitly set by the user. A number of things are done to make the interface more convenient, e.g., the name alone instructs `odeset` to assign the value "true" to a Boolean variable.

The most commonly used options are `rtol` and `atol`, tolerances associated with the error control. Specification of the error control is a difficult matter discussed in

[37]. There it is explained how the simplifying assumptions made in the old `ode23` and `ode45` can lead to unsatisfactory results. Polking [30] makes the same point. Unfortunately, there seems to be no way to provide automatically a default control of the error that is completely satisfactory.

In the suite, the local error e_i in y_i is estimated in each step and made to satisfy

$$|e_i| \leq r\,|y_i| + a_i$$

where $r =$ `rtol` and $a_i =$ `atol(i)`. The scalar relative error tolerance `rtol` has a default value of 10^{-3}. The vector of absolute error tolerances `atol` has by default all its values equal to 10^{-6}. If a scalar absolute error tolerance is input, the code understands that the value is to be assigned to all entries of the `atol` vector.

As another example of setting options, suppose that we wish to solve a stiff problem with a constant Jacobian, an absolute error tolerance of 10^{-20} is appropriate for all components, and we wish to impose a maximum step size of 3500 to assure that the code will recognize phenomena occurring on this time scale. This is done by

```
options = odeset('constantJ','atol',1e-20,'hmax',3500);
```

An illuminating example is provided by the `chm6ode` function shown in Figure 3. Its solution is discussed in [27]. Figure 5 is a log-log plot of $y_2(t)$. A fundamental difficulty is that with an initial value of 0, there is no natural measure of scale for $y_2(t)$. It turns out that the component never gets bigger than about 7×10^{-10}, so the default absolute error tolerance of 10^{-6} is inappropriate. After an integration that revealed the general size of this solution component, we solved the problem again with the default relative tolerance of 10^{-3} and an optional absolute error tolerance of 10^{-13}. This is accomplished by `ode15s` in only 139 steps. The step sizes ranged from 5×10^{-14} to 10^2! This is mainly due to $y_2(t)$ changing rapidly on a very short time scale. On plotting the output it is seen that a logarithmic scale in t would be more appropriate. Because all the solution values are provided to users in our design and they are retained in MATLAB, they can be displayed in a more satisfactory way without having to recompute them.

In order that all codes in the suite have the same appearance to the user, the codes intended for stiff problems by default compute internally the necessary partial derivatives by differences. Users are given the option of providing a function for the analytical evaluation of the Jacobian. They are also given the option of specifying that the Jacobian is constant, a special case that leads to significant savings in `ode23s`. The default is to treat the Jacobian as a full matrix. To take advantage of a sparse Jacobian, the code must be informed of the sparsity pattern. The distinction between banded Jacobians and the much more complicated case of general sparse Jacobians that is important in other codes is absent in the new suite. All that a user must do is provide a (sparse) matrix S of zeros and ones that represents the sparsity pattern of the Jacobian. There are a number of ways to create matrix S. If there are `neq` equations, an `neq` $\times$ `neq` sparse matrix of zeros can be created by

```
S = sparse(neq,neq);
```

Then for each equation i in $F(t, y)$, if y_j appears in the equation, the (i, j) entry of S is set to 1, i.e. `S(i,j) = 1`. These quantities can be set in any order. If the Jacobian has a regular structure, it may be possible to define S more compactly. For example, if the Jacobian is banded with bandwidth $2m + 1$, it can be defined in a single line

```
S = spdiags(ones(neq,2m+1),-m:m,neq,neq)
```

After defining the sparsity pattern, the value S is assigned to the option `sparseJ` with `odeset`. No further action is required of the user.

As discussed in §4, the two codes for stiff problems permit a more general form of the differential equation, namely $M(t)\,y' = f(t,y)$, with a mass matrix $M(t)$. In other computing environments a mass matrix raises awkward questions about specification of M and its structure and how its structure relates to that of the Jacobian of $f(t,y)$. In our interface, the codes are informed of the presence of a mass matrix by means of the `mass` option. The value of this option is the name of a function that returns $M(t)$. Or, if the mass matrix is constant, the matrix itself can be provided as the value of the option. The language deals automatically with the structures of the matrices that arise in the specification and solution of the problem [19].

7. Numerical Partial Derivatives. Methods for the solution of stiff problems involve partial derivatives of the function defining the differential equation. The popular codes allow users to provide a routine for evaluating these derivatives analytically. However, this is so much trouble for users and so prone to error that the default is to approximate them internally by numerical differentiation. The new suite follows this approach, using a function `numjac` to compute the numerical approximation.

The scaling difficulties that are possible when approximating partial derivatives by differences are well-known [37]. The `numjac` code implements a scheme of D. E. Salane [32] that takes advantage of experience gained at one step to select good increments for difference quotients at the next step. If it appears that a column might consist mainly of roundoff, the increment is adjusted and the column recomputed.

The solvers invoke `numjac` by

```
[dFdy,fac,g] = numjac('F',t,y,Fty,thresh,fac,vectorized,S,g);
```

where the argument `'F'` is a string naming the function that defines the differential equation and `Fty` is the result of `'F'` evaluated at the current point in the integration `(t,y)`. The vector `thresh` provides a threshold of significance for `y`, i.e. the exact value of a component `y(i)` with magnitude less than `thresh(i)` is not important.

One aspect of the formation of partial derivatives special to the suite is the Boolean option `vectorized`. It is generally easy to code `'F'` so that it can return an array of function values. A vectorized version of the van der Pol example is shown in Figure 4. If `'F'` is coded so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]` and `vectorized` is set true, `numjac` will approximate all columns of the Jacobian with a single call to `'F'`. This avoids the overhead of repeatedly calling the function and it may reduce the cost of the evaluations themselves.

```
function dy = vdpex(t,y)
dy = zeros(size(y));      % preallocate column vector dy
dy(1,:) = y(2,:);
dy(2,:) = 1000*(1 - y(1,:).^2).*y(2,:) - y(1,:);
```

FIG. 4. *The vectorized* MATLAB *code for the* `vdpex` *problem.*

Another special aspect of `numjac` is that it computes sparse Jacobians as well as full ones. The structure of the Jacobian is provided by means of a sparse matrix `S` of

zeros and ones. The first time that a solver calls `numjac`, the function finds groups of columns of `dFdy` that can be approximated with a single call to `'F'`. This is done only once and the grouping is saved in `g`. Two schemes are tried (first-fit and first-fit after reverse column minimum-degree ordering [19]) and the more efficient grouping is adopted. This may not result in an optimal grouping because finding the smallest packing is an NP-complete problem equivalent to K-coloring a graph [9].

The modified Rosenbrock code requires the partial derivative `dFdt` every time it requires `dFdy`. On reaching t, the step size h provides a measure of scale for the approximation of `dFdt` by a forward difference. The computation is so simple that it is done in the solver itself.

8. Examples. The first question a user must ask is, which of the five codes in the suite should I use and which, if any, of the options? By design it is very easy simply to try the most promising codes, but it is useful to have some insight. The method implemented suggests circumstances in which a code might be particularly efficient or not. This section presents some experiments with test problems from classic collections that illustrate the performance of the solvers and the effects of using certain options. We report how much it costs to solve a problem for a given tolerance, usually the default tolerance. This is ordinarily what users of MATLAB want to know. A discussion of some of the issues that arise in comparing solvers is found in [37] where this approach is called the *first measure of efficiency*. Implicit in the use of this measure is the assumption that for routine problems, the codes compared will produce solutions with accuracy at least comparable to the tolerance. No credit is given for producing solutions with much more accuracy than requested. Because the solvers control local errors, the true, or global, error can be sensitive to the method and the details of its implementation. We have tuned the step size selection algorithms so that for a wide range of test problems, consistent global accuracies comparable to the tolerance are delivered by the various solvers applied to the same problem.

From our experience with writing solvers in other computing environments, we believe that our implementations of the methods are comparable in quality to popular codes based on the same or related methods. Naturally we tested this by comparing the new Runge-Kutta codes to the old MATLAB ones. Also, we report here tests showing that the NDF code is comparable to a popular BDF code written in FORTRAN.

The experiments reported here and others we have made suggest that except in special circumstances, `ode45` should be the code tried first. If there is reason to believe the problem to be stiff, or if the problem turns out to be unexpectedly difficult for `ode45`, the `ode15s` code should be tried. When solving stiff problems, it is important to keep in mind the options that improve the efficiency of forming Jacobians.

8.1. Stiff Examples. In MATLAB it is advantageous to vectorize computations whenever possible. Accordingly, all stiff problems were coded to use the `vectorized` option when computing numerical Jacobians. Also, the advantages of the `constantJ` option are so obvious that we used it when solving the stiff problems with constant Jacobians, namely `a2ex`, `a3ex`, `b5ex`, and `hb3ex`.

The `ode15s` code was developed for the NDF's. It is such an easy matter to provide for the BDF's in this code that they were allowed as an option. Some experiments show the consequences of exercising this option. Table 2 gives the number of steps and the real time required for the two choices when applied to a set of 13 stiff problems. For all but one problem the default NDF's result in fewer steps than the BDF's (an average of 10.9% fewer), and for *all* problems, the code is faster when using the NDF's (an average of 8.2% faster).

TABLE 2
Comparison of the NDF's and BDF's in `ode15s`*. Times are measured as seconds on a Sparc2.*

	BDF steps	NDF steps	percent fewer	BDF time	NDF time	percent faster
`a2ex`	118	101	14.4	3.60	3.14	12.8
`a3ex`	134	130	3.0	3.96	3.87	2.4
`b5ex`	1165	936	19.7	32.58	25.95	20.4
`buiex`	57	52	8.8	2.05	1.92	6.4
`chm6ex`	152	139	8.6	4.05	3.63	10.3
`chm7ex`	57	39	31.6	1.82	1.48	18.4
`chm9ex`	910	825	9.3	30.53	29.38	3.8
`d1ex`	67	62	7.5	2.35	2.29	2.5
`gearex`	20	19	5.0	1.12	1.08	3.5
`hb1ex`	197	179	9.1	5.57	5.09	8.5
`hb2ex`	555	577	-4.0	13.49	13.45	0.3
`hb3ex`	766	690	9.9	19.79	17.77	10.2
`vdpex`	708	573	19.1	20.75	19.33	6.9

To verify that the performance of `ode15s` is comparable to that of a modern BDF code, we have compared `ode15s` using the NDF's to DDRIV2 [24] on some relatively difficult problems. DDRIV2 is an easy-to-use driver for a more complex code with an appearance not too different from `ode15s`. It is a quasi-constant step size implementation of the BDF's of orders 1-5 that computes answers at specified points by interpolation. It approximates Jacobians internally by differences with an algorithm related to that of `ode15s`.

It is not possible to compare DDRIV2 and `ode15s` in detail because they cannot be used to solve exactly the same computational problem. For one thing, the error controls are different. DDRIV2 uses a root-mean-square norm to measure the error in a solution component relative to the larger of the magnitude of the component and a threshold. We made the controls roughly equivalent by taking the threshold to be equal to the desired absolute error and dividing the tolerances given to DDRIV2 by the square root of the number of equations. In addition the two codes handle output differently. DDRIV2 provides answers wherever requested, but only at those points. We asked the codes to produce 150 answers equally spaced within the interval of integration. This is inadequate for some of the examples, but asking for more answers could increase the cost in DDRIV2 because it has an internal maximum step size that is twice the distance between output points. Accepting a possible reduction in efficiency in `ode15s`, we used an option to impose the same maximum step size.

Table 3 compares DDRIV2 to `ode15s` using the NDF's. We interpret these comparisons as showing that `ode15s` is an effective code for the solution of stiff problems. DDRIV2 does not save Jacobians and the numerical results indicate that to a degree `ode15s` is trading linear algebra for a smaller number of approximate Jacobians. This is appropriate in MATLAB, but because these examples involve just a few equations, the benefits of reusing Jacobians are masked.

The chemical reaction problem `chm6ex` is given in §6. A plot of one component is displayed in Figure 5. `ode15s` is able to solve the problem effectively using a remarkably small number of Jacobians. Problem `chm9ex` is a scaled version of the Belousov oscillating chemical reaction [14]. A discussion of this problem and plots are

TABLE 3
Comparison of DDRIV2 to `ode15s` *using the NDF's. The table shows the number of successful steps, the number of failed steps, the number of function calls, the number of partial derivative evaluations, the number of LU decompositions, and the number of linear system solutions. Note that the integration parameters of* `ode15s` *were changed from their default values.*

example	code	time steps	failed steps	f evals	$\partial f/\partial y$ evals	LU's	linear solves
`chm6ex`	DDRIV2	218	6	404	33	33	271
	`ode15s`	177	4	224	2	29	213
`chm9ex`	DDRIV2	1073	217	2470	220	220	1802
	`ode15s`	750	227	2366	83	322	2033
`hb2ex`	DDRIV2	1370	162	2675	176	176	2316
	`ode15s`	939	70	1321	4	165	1308
`vdpex`	DDRIV2	871	185	1836	167	167	1497
	`ode15s`	724	197	1965	33	261	1865

found in [1], p. 49 ff. The limit solution is periodic and exhibits regions of very sharp change. We chose the interval $[0, 650]$ so as to include two complete periods and part of another. `ode15s` is trading some linear algebra for a reduction in the number of Jacobians. Because there are only three solution components, reducing the number of Jacobians does not have much effect on the number of function evaluations.

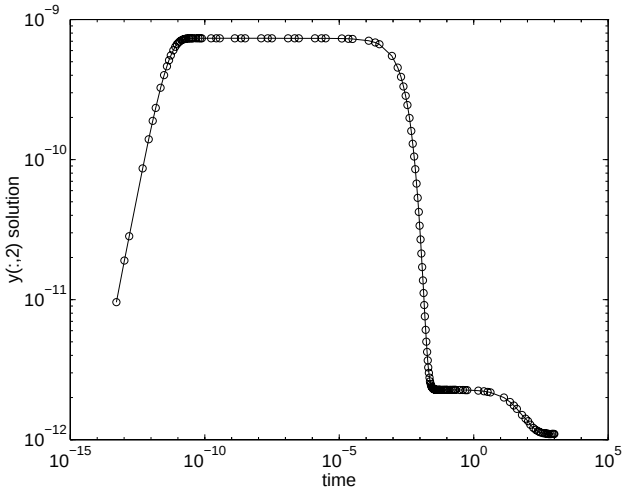


FIG. 5. *A log-log plot of the second component of the solution of* `chm6ex`.

Hindmarsh and Byrne [23] present the non-autonomous problem coded in `hb2ex` that arises in a diurnal kinetics model and they discuss its solution with EPISODE. The scaling of one component is such that an absolute error tolerance of 10^{-20} is needed. The problem is also discussed at length in [24] where it is solved with DDRIV2. As the measures of cost given in Table 3 show, `ode15s` performs quite well in comparison to a good BDF code. For this problem, reuse of Jacobians proved to be quite advantageous. With only two equations, numerical evaluation of a Jacobian is so cheap that the difference in cost cannot be due entirely to saving Jacobians.

Example `vdpex` is the van der Pol equation in relaxation oscillation in the form

specified in [35]. The tradeoffs in the tuning of `ode15s` as compared to DDRIV2 are clear. Because this problem involves only two solution components, forming Jacobians more often would probably have been a bargain in `ode15s`.

Now we compare the modified Rosenbrock code `ode23s` to `ode15s`. Table 4 provides statistics for two stiff problems. Note that `ode15s` was run with both the NDF's and the BDF's. The fact that NDF1 and NDF2 are more efficient than BDF1 and BDF2 is evident for the `b5ex` experiments in which the maximum order was limited.

TABLE 4
Comparison of `ode23s` *to* `ode15s`. *For* `vdpex`, *the relative accuracy was changed from the default of 0.1% to 1%. Times are measured as seconds on a Sparc2.*

example	code	time	time steps	failed steps	f evals	$\partial f/\partial y$ evals	LU's	linear solves
`vdpex`	`ode15s` (BDF)	17.06	525	203	1594	45	266	1458
	`ode15s` (NDF)	15.83	490	179	1514	46	249	1375
	`ode23s`	14.21	302	96	1706	303	398	1194
`b5ex`	`ode15s` (BDF)	32.58	1165	124	2586	1	319	2578
	`ode15s` (NDF)	25.95	936	97	2074	1	263	2066
	`ode23s`	15.38	549	17	1689	1	566	1698

As a first example we solved `vdpex` with a relative accuracy tolerance of 1.0%. At this tolerance `ode23s` was faster than `ode15s`, even though it made many more Jacobian evaluations, and the plot of $y(t)$ in Figure 6 obtained with `ode23s` is notably better than that of `ode15s`. However, when the tolerance is tightened to the default of 0.1%, `ode15s` is faster than `ode23s` and the plot of $y(t)$ is just as good.

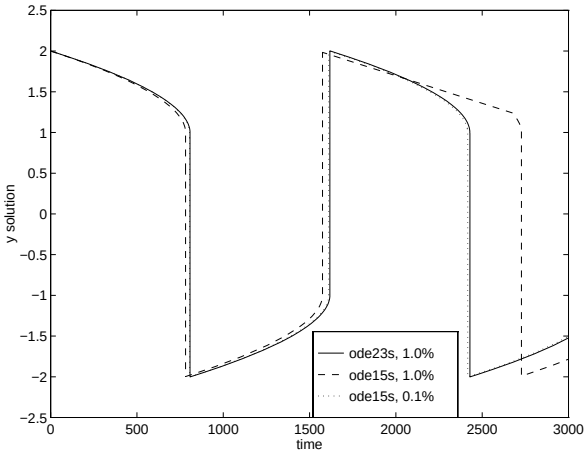


FIG. 6. *The* `vdpex` *solution computed by* `ode23s` *and* `ode15s` *at 1.0% accuracy, and by* `ode15s` *at 0.1% accuracy. The latter indicates that the 1.0% solution by* `ode15s` *is less accurate than* `ode23s`.

Problem `b5ex` [15] has a Jacobian with eigenvalues near the imaginary axis. The popular variable-order BDF codes do not deal with this well. Gaffney [16] compares BDF and other kinds of codes on this and similar problems. `ode15s` recognizes automatically that this problem has a constant Jacobian. For a fair comparison, we used

the option of telling both codes that the Jacobian is constant. We also restricted the maximum order of the NDF's used by `ode15s` to 2, so that both codes would be L-stable. Evidently `ode23s` is to be preferred for the solution of this problem.

Supplying a function for evaluating the Jacobian can be quite advantageous, both with respect to reliability and cost. Table 5 shows the effects of using the `analyticJ` option for some of the most expensive of our examples. Because `ode23s` evaluates the Jacobian at every step, reducing this cost by means of an analytical Jacobian has an important effect on the overall cost. It is much less significant when using `ode15s` because it makes comparatively few evaluations of the Jacobian.

TABLE 5

The solutions of four problems by `ode23s` *and* `ode15s` *showing the effect of using the* `analyticJ` *option to supply a function for evaluation of the Jacobian (time2). The* `brussex` *problem is* 100×100 *and its Jacobian function returns a sparse matrix. Times are measured as seconds on a Sparc2.*

	`ode23s`		`ode15s`	
problem	time	time2	time	time2
`brussex`	80.00	13.82	23.37	9.70
`chm9ex`	47.12	27.32	29.38	25.86
`hb2ex`	140.78	87.29	13.45	13.40
`vdpex`	28.61	16.36	19.33	19.21

Next, we examine the role of Jacobian sparsity. The `brussex` example is the classic "Brusselator" system modelling diffusion in a chemical reaction [21],

$$\begin{aligned} u_i' &= 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1}) \\ v_i' &= 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1}) \end{aligned}$$

and is solved on the time interval $[0, 10]$ with $\alpha = 1/50$ and

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

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

For progressively larger values of N, Table 6 shows the number of steps taken and compares the number of seconds required to solve the `brussex` problem by `ode45`, `ode23s` and `ode15s`. The `ode45` results indicate that the system becomes quite stiff for the larger N. The first columns of the results for `ode23s` and `ode15s` were produced using the default numerical approximation of Jacobians. As the second columns show, the `sparseJ` option makes a tremendous difference. Until N becomes large, `ode15s` is efficient even without the `sparseJ` option because it forms relatively few Jacobians.

The `fem2ex` example involves a mass matrix. The system of ODE's, found in [41], 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 $1/(N+1)$, and the solution of the partial differential equation is approximated at $x_k = k\pi h$ for $k = 0, 1, \ldots, N+1$ by

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

TABLE 6
The solutions of various size `brussex` *problems by* `ode45`, `ode23s`, `ode23s` *using the* `sparseJ` *option (time2),* `ode15s`, *and* `ode15s` *using the* `sparseJ` *option (time2). Times are measured as seconds on a Sparc2.*

	`ode45`		`ode23s`			`ode15s`		
size	steps	time	steps	time	time2	steps	time	time2
100	629	143.95	59	80.00	15.04	82	23.37	10.36
200	2458	4052.99	59	499.44	24.50	82	104.49	17.78
400	NA	NA	59	3398.47	43.00	85	574.42	32.19
600	NA	NA	59	NA	62.84	85	1703.68	49.21
800	NA	NA	59	NA	83.91	85	NA	63.51
1000	NA	NA	59	NA	105.93	85	NA	80.74

Here $\phi_k(x)$ is a piecewise linear function that is 1 at x_k and 0 at all the other x_j. The Galerkin discretization leads to the system of ODE's

$$A(t)\,c' = R\,c \quad \text{where} \quad c(t) = \begin{bmatrix} c_1(t) \\ \vdots \\ c_N(t) \end{bmatrix}$$

and the tridiagonal matrices $A(t)$ and R are given by

$$A_{ij} = \begin{cases} \exp(-t)2h/3 & \text{if } i = j \\ \exp(-t)h/6 & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad R_{ij} = \begin{cases} -2/h & \text{if } i = j \\ 1/h & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

The initial values $c(0)$ are taken from the initial condition for the partial differential equation. The problem is solved on the time interval $[0, \pi]$.

Because the mass matrix A depends on t, this equation cannot be solved directly with `ode23s`. However, scaling by $\exp(t)$ results in an equivalent system with a constant mass matrix that can be solved with both codes. As is typical of the method of lines, the mass matrix is sparse, but in this instance we have followed [41] in taking $N = 9$, which is too small to take advantage of the sparsity. The solution of `fem2ex` is shown in Figure 7 and statistics are presented in Table 7.

TABLE 7
Comparison of `ode23s` *to* `ode15s` *for a problem with a constant mass matrix,* `fem2ex` *with* $N = 9$. *Times are measured as seconds on a Sparc2.*

example	code	time	time steps	failed steps	f evals	$\partial f/\partial y$ evals	LU's	linear solves
`fem2ex`	`ode15s`	4.71	46	14	175	5	21	124
	`ode23s`	5.35	40	1	493	41	41	123

8.2. Non-stiff Examples. In this section we consider four non-stiff examples drawn from the collections [15, 39]. `vdpns` is the van der Pol equation with $\mu = 1$. `rigid` is the Euler equations of a rigid body without external forces as proposed by Krogh. The solution is displayed in Figure 1. `twobody`, D5 of [15], is the two body problem with an elliptical orbit of eccentricity 0.9. `r3body` describes a periodic orbit for a restricted three body problem [39]. Because the problems are non-stiff, they can be solved with all the MATLAB ODE solvers. Table 8 compares the costs of solution.

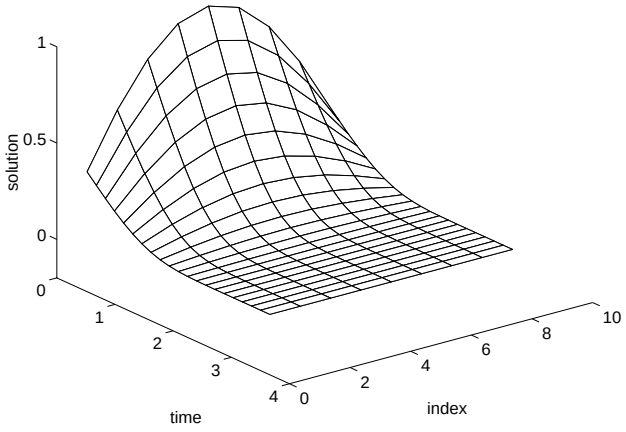


FIG. 7. *The* `fem2ex` *solution with* $N = 9$ *as computed by* `ode15s`.

9. Conclusion. Source code for the solvers and examples may be obtained gratis by ftp on ftp.mathworks.com in the pub/mathworks/toolbox/matlab/funfun directory. The solvers require MATLAB version 4.2 or later.

We have had the pleasure of correspondence and discussions with many experts whose publications and advice have been crucial to a number of aspects of this project. C. B. Moler helped us with just about every aspect. We have had the benefit of advice from a number of leading experts in explicit Runge-Kutta methods, viz. J. Dormand and P. Prince; M. Calvo, L. Randez, and J. Montijano; and P. Sharp and J. Verner. D. Salane provided us with FORTRAN versions of his algorithm for computing numerical partial derivatives along with helpful comments. T. Coleman provided advice about column grouping strategies. I. Gladwell provided advice about mathematical software for ordinary differential equations. H. Zedan provided copies of his publications that were essential to the development of our modified Rosenbrock method. J. Polking's experience teaching the solution of ordinary differential equations using the previous generation of codes in MATLAB influenced the new generation in a number of ways.

REFERENCES

[1] R. C. AIKEN, ed., *Stiff Computation*, Oxford Univ. Press, Oxford, 1985.
[2] G. BADER AND P. DEULFHARD, *A semi-implicit mid-point rule for stiff systems of ordinary differential equations*, Tech. Report 114, Institut für Angewandte Mathematik, Universität Heidelberg, Germany, 1981.
[3] P. BOGACKI AND L. F. SHAMPINE, *A 3(2) pair of Runge-Kutta formulas*, Appl. Math. Letters, 2 (1989), pp. 1–9.
[4] R. W. BRANKIN, I. GLADWELL, AND L. F. SHAMPINE, *RKSUITE: A suite of Runge-Kutta codes for the initial value problem for ODEs*, Tech. Report 92-S1, Math. Dept., Southern Methodist Univ., Dallas, 1992.
[5] R. K. BRAYTON, F. G. GUSTAVSON, AND G. D. HACHTEL, *A new efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas*, Proc. IEEE, 60 (1972), pp. 98–108.
[6] K. E. BRENAN, S. L. CAMPBELL, AND L. R. PETZOLD, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elsevier Science Publishing Co., New York, 1989.

TABLE 8

Comparison of all five of the ODE solvers on a set of four non-stiff problems. Times are measured as seconds on a Sparc2.

example	code	time	time steps	failed steps	f evals	$\partial f/\partial y$ evals	LU's	linear solves
`rigid`	`ode23s`	2.76	58	10	373	59	68	204
	`ode15s`	2.08	82	17	184	1	30	179
	`ode113`	2.12	65	4	135	0	0	0
	`ode23`	0.92	55	13	205	0	0	0
	`ode45`	0.57	19	2	127	0	0	0
`r3body`	`ode23s`	22.79	372	1	2612	373	373	1119
	`ode15s`	8.74	321	48	575	1	87	569
	`ode113`	10.72	237	20	495	0	0	0
	`ode23`	6.19	301	4	916	0	0	0
	`ode45`	3.84	73	27	601	0	0	0
`twobody`	`ode23s`	44.45	871	1	6105	872	872	2616
	`ode15s`	13.66	584	64	963	2	135	952
	`ode113`	18.46	396	29	822	0	0	0
	`ode23`	11.51	727	0	2182	0	0	0
	`ode45`	4.98	133	35	1009	0	0	0
`vdpns`	`ode23s`	6.65	158	21	836	159	179	537
	`ode15s`	4.48	192	35	426	1	60	422
	`ode113`	5.33	162	12	337	0	0	0
	`ode23`	2.10	146	19	496	0	0	0
	`ode45`	1.43	51	11	373	0	0	0

[7] P. N. BROWN, G. D. BYRNE, AND A. C. HINDMARSH, *VODE: a variable-coefficient ODE solver*, SIAM J. Sci. Comput., 10 (1989), pp. 1038–1051.

[8] G. D. BYRNE AND A. C. HINDMARSH, *A polyalgorithm for the numerical solution of ordinary differential equations*, ACM Transactions on Mathematical Software, 1 (1975), pp. 71–96.

[9] T. F. COLEMAN, B. S. GARBOW, AND J. J. MORE, *Software for estimating sparse Jacobian matrices*, ACM Transactions on Mathematical Software, 11 (1984), pp. 329–345.

[10] A. R. CURTIS, *The FACSIMILE numerical integrator for stiff initial value problems*, in Computational Techniques for Ordinary Differential Equations, I. Gladwell and D. K. Sayers, eds., Academic, London, 1980, pp. 47–82.

[11] P. DEUFLHARD, *Recent progress in extrapolation methods for ordinary differential equations*, SIAM Review, 27 (1985), pp. 505–535.

[12] J. R. DORMAND AND P. J. PRINCE, *A family of embedded Runge-Kutta formulae*, J. Comp. Appl. Math., 6 (1980), pp. 19–26.

[13] ———, *Runge-Kutta triples*, Comp. & Maths. with Appls., 12A (1986), pp. 1007–1017.

[14] W. H. ENRIGHT AND T. E. HULL, *Comparing numerical methods for the solution of stiff systems of ODE's arising in chemistry*, in Numerical Methods for Differential Systems, L. Lapidus and W. Schiesser, eds., Academic, New York, 1976, pp. 45–66.

[15] W. H. ENRIGHT, T. E. HULL, AND B. LINDBERG, *Comparing numerical methods for stiff systems of ODE's*, BIT, 15 (1975), pp. 10–48.

[16] P. W. GAFFNEY, *A performance evaluation of some FORTRAN subroutines for the solution of stiff oscillatory ordinary differential equations*, ACM Trans. Math. Software, 10 (1984), pp. 58–72.

[17] C. W. GEAR, *Numerical Initial Value Problems in Ordinary Differential Equations*, Automatic Computation, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[18] C. W. GEAR AND D. S. WATANABE, *Stability and convergence of variable order multistep methods*, SIAM J. Numer. Anal., 11 (1974), pp. 1044–1058.

[19] J. R. GILBERT, C. MOLER, AND R. SCHREIBER, *Sparse matrices in MATLAB: design and implementation*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 333–356.

[20] H. Gollwitzer, *Differential Systems User Manual*, Dept. Math. & Comp. Sci., Drexel Univ., Philadelphia, 1991.
[21] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II*, Springer, 1991.
[22] A. C. Hindmarsh, *LSODE and LSODI, two new initial value ordinary differential equation solvers*, ACM SIGNUM Newsletter, 15 (1980), pp. 10–11.
[23] A. C. Hindmarsh and G. D. Byrne, *Applications of EPISODE: an experimental package for the integration of systems of ordinary differential equations*, in Numerical Methods for Differential Systems, L. Lapidus and W. Schiesser, eds., Academic, New York, 1976, pp. 147–166.
[24] D. Kahaner, C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
[25] R. W. Klopfenstein, *Numerical differentiation formulas for stiff systems of ordinary differential equations*, RCA Review, 32 (1971), pp. 447–462.
[26] F. T. Krogh, *Algorithms for changing the step size*, SIAM J. Numer. Anal., 10 (1973), pp. 949–965.
[27] L. Lapidus, R. C. Aiken, and Y. A. Liu, *The occurrence and numerical solution of physical and chemical systems having widely varying time constants*, in Stiff Differential Systems, R. Willoughby, ed., Plenum Press, New York, 1974, pp. 187–200.
[28] D. Luss and N. R. Amundson, *Stability of batch catalytic fluidized beds*, AIChE J., 14 (1968), pp. 211–221.
[29] The MathWorks, Inc., *MATLAB 4.2*, 24 Prime Park Way, Natick MA, 1994.
[30] J. C. Polking, *MATLAB Manual for Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
[31] T. Reiher, *Statilitätsuntersuchungen bei rückwärtigen Differentiationsformeln in Abhängigkeit von einem Parameter*, Tech. Report #11, Sektion Mathematik, Humboldt–Universität zu Berlin, 1978.
[32] D. E. Salane, *Adaptive routines for forming Jacobians numerically*, Tech. Report SAND86–1319, Sandia National Laboratories, Albuquerque, NM, 1986.
[33] R. E. Scraton, *Some L-stable methods for stiff differential equations*, Intern. J. Computer Maths., 9 (1981), pp. 81–87.
[34] L. F. Shampine, *Implementation of implicit formulas for the solution of ODE's*, SIAM J. Sci. Statist. Comput., 1 (1980), pp. 103–118.
[35] ———, *Evaluation of a test set for stiff ODE solvers*, ACM Trans. Math. Software, 7 (1981), pp. 409–420.
[36] ———, *Implementation of Rosenbrock methods*, ACM Trans. Math. Software, 8 (1982), pp. 93–113.
[37] ———, *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.
[38] L. F. Shampine and L. S. Baca, *Error estimators for stiff differential equations*, J. Comp. Appl. Math., 11 (1984), pp. 197–207.
[39] L. F. Shampine and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.
[40] T. Steihaug and A. Wolfbrandt, *An attempt to avoid exact Jacobian and non–linear equations in the numerical solution of stiff differential equations*, Math. Comp., 33 (1979), pp. 521–534.
[41] Visual Numerics, Inc., *IMSL MATH/LIBRARY. FORTRAN subroutines for mathematical applications*, Suite 440, 9990 Richmond, Houston, TX, 1994.
[42] H. Zedan, *A variable order/variable-stepsize Rosenbrock-type algorithm for solving stiff systems of ODE's*, Tech. Report YCS114, Dept. Comp. Sci., Univ. of York, York, England, 1989. (to appear in ACM Trans. Math. Software).
[43] ———, *Avoiding the exactness of the Jacobian matrix in Rosenbrock formulae*, Computers Math. Applic., 19 (1990), pp. 83–89.