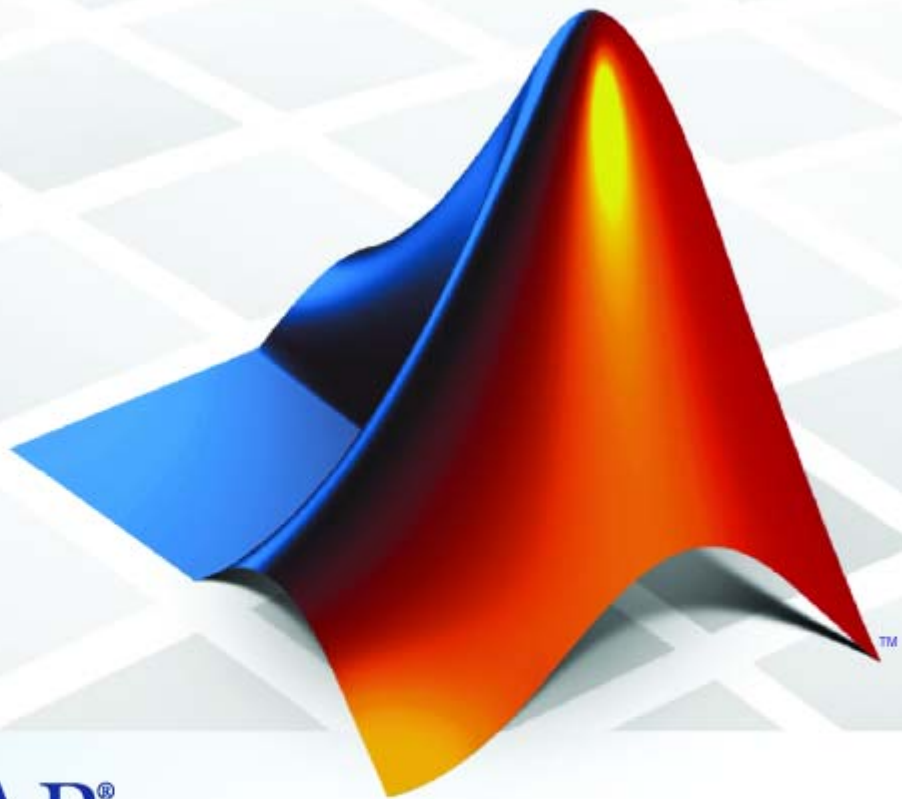


MATLAB® 7

Getting Started Guide



MATLAB®

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® Getting Started Guide

© COPYRIGHT 1984–2008 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

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)
March 2008	Eleventh printing	Minor revision for MATLAB 7.6 (Release 2008a)
October 2008	Twelfth printing	Minor revision for MATLAB 7.7 (Release 2008b)

Introduction

1

Product Overview	1-2
Overview of the MATLAB Environment	1-2
The MATLAB System	1-3
Documentation	1-5
Starting and Quitting the MATLAB Program	1-7
Starting a MATLAB Session	1-7
Quitting the MATLAB Program	1-8

Matrices and Arrays

2

Matrices and Magic Squares	2-2
About Matrices	2-2
Entering Matrices	2-4
sum, transpose, and diag	2-5
Subscripts	2-7
The Colon Operator	2-8
The magic Function	2-9
Expressions	2-11
Variables	2-11
Numbers	2-12
Operators	2-13
Functions	2-13
Examples of Expressions	2-15
Working with Matrices	2-16
Generating Matrices	2-16
The load Function	2-17

M-Files	2-17
Concatenation	2-18
Deleting Rows and Columns	2-19
More About Matrices and Arrays	2-20
Linear Algebra	2-20
Arrays	2-24
Multivariate Data	2-26
Scalar Expansion	2-27
Logical Subscripting	2-27
The find Function	2-28
Controlling Command Window Input and Output	2-30
The format Function	2-30
Suppressing Output	2-31
Entering Long Statements	2-32
Command Line Editing	2-32

Graphics

3

Overview of Plotting	3-2
Plotting Process	3-2
Graph Components	3-6
Figure Tools	3-7
Arranging Graphs Within a Figure	3-14
Choosing a Type of Graph to Plot	3-15
Editing Plots	3-19
Plot Edit Mode	3-19
Using Functions to Edit Graphs	3-24
Some Ways to Use Plotting Tools	3-25
Plotting Two Variables with Plotting Tools	3-25
Changing the Appearance of Lines and Markers	3-28
Adding More Data to the Graph	3-29
Changing the Type of Graph	3-32
Modifying the Graph Data Source	3-34

Preparing Graphs for Presentation	3-39
Annotating Graphs for Presentation	3-39
Printing the Graph	3-44
Exporting the Graph	3-48
Using Basic Plotting Functions	3-51
Creating a Plot	3-51
Plotting Multiple Data Sets in One Graph	3-52
Specifying Line Styles and Colors	3-53
Plotting Lines and Markers	3-54
Graphing Imaginary and Complex Data	3-56
Adding Plots to an Existing Graph	3-57
Figure Windows	3-58
Displaying Multiple Plots in One Figure	3-59
Controlling the Axes	3-61
Adding Axis Labels and Titles	3-62
Saving Figures	3-63
Creating Mesh and Surface Plots	3-65
About Mesh and Surface Plots	3-65
Visualizing Functions of Two Variables	3-65
Plotting Image Data	3-73
About Plotting Image Data	3-73
Reading and Writing Images	3-74
Printing Graphics	3-75
Overview of Printing	3-75
Printing from the File Menu	3-75
Exporting the Figure to a Graphics File	3-76
Using the Print Command	3-76
Understanding Handle Graphics Objects	3-78
Using the Handle	3-78
Graphics Objects	3-79
Setting Object Properties	3-81
Specifying the Axes or Figure	3-84
Finding the Handles of Existing Objects	3-85

4

Flow Control	4-2
Conditional Control – if, else, switch	4-2
Loop Control – for, while, continue, break	4-5
Error Control – try, catch	4-7
Program Termination – return	4-8
Other Data Structures	4-9
Multidimensional Arrays	4-9
Cell Arrays	4-11
Characters and Text	4-13
Structures	4-16
Scripts and Functions	4-20
Overview	4-20
Scripts	4-21
Functions	4-22
Types of Functions	4-24
Global Variables	4-26
Passing String Arguments to Functions	4-27
The eval Function	4-28
Function Handles	4-28
Function Functions	4-29
Vectorization	4-31
Preallocation	4-32
Object-Oriented Programming	4-33
MATLAB Classes and Objects	4-33
Learn About Defining MATLAB Classes	4-33

5

Introduction	5-2
Preprocessing Data	5-3

Overview	5-3
Loading the Data	5-3
Missing Data	5-3
Outliers	5-4
Smoothing and Filtering	5-6
Summarizing Data	5-10
Overview	5-10
Measures of Location	5-10
Measures of Scale	5-11
Shape of a Distribution	5-11
Visualizing Data	5-14
Overview	5-14
2-D Scatter Plots	5-14
3-D Scatter Plots	5-16
Scatter Plot Arrays	5-18
Exploring Data in Graphs	5-19
Modeling Data	5-27
Overview	5-27
Polynomial Regression	5-27
General Linear Regression	5-28

Creating Graphical User Interfaces

6

What Is GUIDE?	6-2
Laying Out a GUI	6-3
Starting GUIDE	6-3
The Layout Editor	6-4
Programming a GUI	6-7

Desktop Tools and Development Environment

7

Desktop Overview	7-2
Introduction to the Desktop	7-2
Arranging the Desktop	7-4
Start Button	7-4
Command Window and Command History	7-6
Command Window	7-6
Command History	7-7
Help	7-8
Help Browser	7-8
Other Forms of Help	7-11
Typographical Conventions	7-11
Current Directory, Search Path, and File Management	7-13
About the Current Directory and Search Path	7-13
Current Directory	7-13
Search Path	7-15
Workspace Browser and Variable Editor	7-17
Workspace Browser	7-17
Variable Editor	7-18
Editor	7-19
Editing M-Files	7-19
Publishing M-Files	7-21
Improving and Tuning M-Files	7-25
Finding Errors Using the M-Lint Code Check Report	7-25
Improving Performance Using the Profiler	7-27

Programming Interfaces	8-2
Call MATLAB Software from C and Fortran Programs ...	8-2
Call C and Fortran Programs from MATLAB Command Line	8-2
Call Sun Java Commands from MATLAB Command Line	8-3
Call Functions in Shared Libraries	8-3
Import and Export Data	8-3
Component Object Model Interface	8-4
Web Services	8-5
Serial Port Interface	8-6

Introduction

- “Product Overview” on page 1-2
- “Documentation” on page 1-5
- “Starting and Quitting the MATLAB Program” on page 1-7

Product Overview

In this section...
“Overview of the MATLAB Environment” on page 1-2
“The MATLAB System” on page 1-3

Overview of the MATLAB Environment

The MATLAB® high-performance language for technical computing 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. It 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. You can add on toolboxes for signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many other areas.

The MATLAB System

The MATLAB system consists of these main parts:

Desktop Tools and Development Environment

This part of MATLAB is the set of tools and facilities that help you use and become more productive with MATLAB functions and files. Many of these tools are graphical user interfaces. It includes: the MATLAB desktop and Command Window, an editor and debugger, a code analyzer, browsers for viewing help, the workspace, and files, and other tools.

Mathematical Function Library

This library 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 Language

The MATLAB language 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 programs you do not intend to reuse. You can also do “programming in the large” to create complex application programs intended for reuse.

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.

External Interfaces

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

Documentation

The MATLAB program 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, begin with this Getting Started guide. It covers all the primary MATLAB features at a high level, including many examples.

To view the online documentation, select **Help > Product Help** 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 and Related Resources”.

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 and become more productive with MATLAB
- Mathematics — Mathematical operations
- Data Analysis — Data analysis, including data fitting, Fourier analysis, and time-series tools
- Programming Fundamentals — The MATLAB language and how to develop MATLAB applications
- Object-Oriented Programming — Designing and implementing MATLAB classes
- Graphics — Tools and techniques for plotting, graph annotation, printing, and programming with Handle Graphics® objects
- 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 Sun Microsystems™ Java™ software, COM, and the serial port

There is reference documentation for all MATLAB functions:

- **Function Reference** — Lists all MATLAB functions, listed in categories or alphabetically
- **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 for current and recent previous releases
- **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 the MATLAB Program

In this section...
“Starting a MATLAB Session” on page 1-7
“Quitting the MATLAB Program” on page 1-8

Starting a MATLAB Session

On Microsoft® Windows® platforms, start the MATLAB program by double-clicking the MATLAB_R2008b shortcut  on your Windows desktop.

On Apple® Macintosh® platforms, start MATLAB by double-clicking the MATLAB_R2008b icon in the Applications folder.

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

When you start MATLAB, by default, MATLAB automatically loads all the program files provided by The MathWorks for MATLAB and other MathWorks products. You do not have to start each product you want to use.

There are alternative ways to start MATLAB, and you can customize MATLAB startup. For example, you can change the directory in which MATLAB starts or automatically execute MATLAB statements upon startup.

For More Information See “Startup and Shutdown” in the Desktop Tools and Development Environment documentation.

The Desktop

When you start MATLAB, the 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

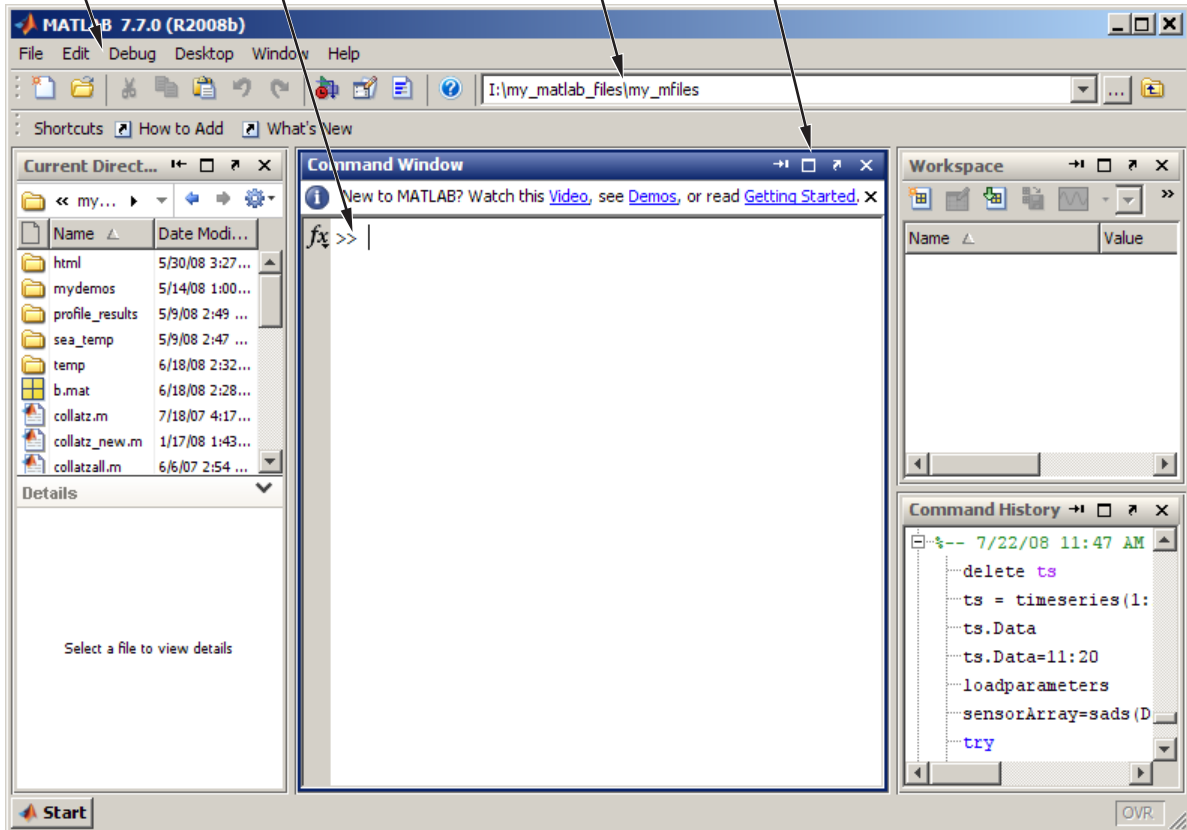
about the desktop tools, see Chapter 7, “Desktop Tools and Development Environment”.

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.



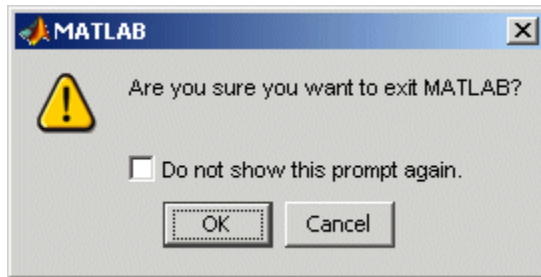
Quitting the MATLAB Program

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 the MATLAB Program” in the Desktop Tools and Development Environment documentation.

Matrices and Arrays

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

- “Matrices and Magic Squares” on page 2-2
- “Expressions” on page 2-11
- “Working with Matrices” on page 2-16
- “More About Matrices and Arrays” on page 2-20
- “Controlling Command Window Input and Output” on page 2-30

Matrices and Magic Squares

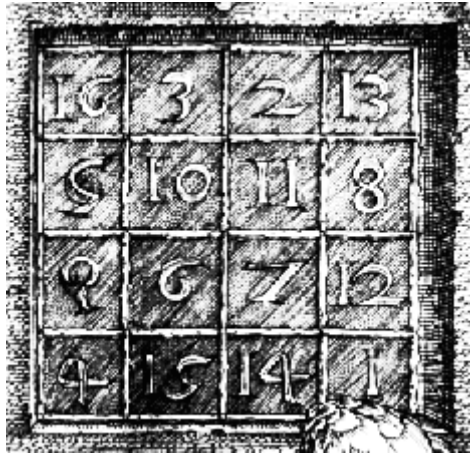
In this section...
“About Matrices” on page 2-2
“Entering Matrices” on page 2-4
“sum, transpose, and diag” on page 2-5
“Subscripts” on page 2-7
“The Colon Operator” on page 2-8
“The magic Function” on page 2-9

About Matrices

In the MATLAB environment, 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. 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 the 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 subscript 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, $A(k)$. A single subscript is the usual way of referencing row and column vectors. However, 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 the 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.
```

Conversely, 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 . Thus:

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

computes the sum of the fourth column. However, there is a better way to perform this computation. 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. Thus:

```
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 subscript indicates that—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
```


Expressions

In this section...

“Variables” on page 2-11

“Numbers” on page 2-12

“Operators” on page 2-13

“Functions” on page 2-13

“Examples of Expressions” on page 2-15

Variables

Like most other programming languages, the MATLAB language 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.

MATLAB software stores the real and imaginary parts of a complex number. It handles the magnitude of the parts in different ways depending on the context. For instance, the `sort` function sorts based on magnitude and resolves ties by phase angle.

```
sort([3+4i, 4+3i])
ans =
    4.0000 + 3.0000i    3.0000 + 4.0000i
```

This is because of the phase angle:

```
angle(3+4i)
ans =
```

```

0.9273
angle(4+3i)
ans =
0.6435

```

The “equal to” relational operator `==` requires both the real and imaginary parts to be equal. The other binary relational operators `>`, `<`, `>=`, and `<=` ignore the imaginary part of the number and consider the real part only.

Operators

Expressions use familiar arithmetic operators and precedence rules.

<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division
<code>\</code>	Left division (described in “Linear Algebra” in the MATLAB documentation)
<code>^</code>	Power
<code>'</code>	Complex conjugate transpose
<code>()</code>	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.

<code>pi</code>	3.14159265...
<code>i</code>	Imaginary unit, $\sqrt{-1}$
<code>j</code>	Same as <code>i</code>
<code>eps</code>	Floating-point relative precision, $\epsilon = 2^{-52}$
<code>realmin</code>	Smallest floating-point number, 2^{-1022}
<code>realmax</code>	Largest floating-point number, $(2 - \epsilon)2^{1023}$
<code>Inf</code>	Infinity
<code>NaN</code>	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” on page 2-16

“The load Function” on page 2-17

“M-Files” on page 2-17

“Concatenation” on page 2-18

“Deleting Rows and Columns” on page 2-19

Generating Matrices

MATLAB software provides four functions that generate basic matrices.

<code>zeros</code>	All zeros
<code>ones</code>	All ones
<code>rand</code>	Uniformly distributed random elements
<code>randn</code>	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 from 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” on page 2-20
 “Arrays” on page 2-24
 “Multivariate Data” on page 2-26
 “Scalar Expansion” on page 2-27
 “Logical Subscripting” on page 2-27
 “The find Function” on page 2-28

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
```

Because 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 sum results 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:

$$P = \begin{matrix} & \begin{matrix} 0.4706 & 0.0882 & 0.0588 & 0.3824 \\ 0.1471 & 0.2941 & 0.3235 & 0.2353 \\ 0.2647 & 0.1765 & 0.2059 & 0.3529 \\ 0.1176 & 0.4412 & 0.4118 & 0.0294 \end{matrix} \end{matrix}$$

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

$$P^5$$

is

$$\begin{matrix} & \begin{matrix} 0.2507 & 0.2495 & 0.2494 & 0.2504 \\ 0.2497 & 0.2501 & 0.2502 & 0.2500 \\ 0.2500 & 0.2498 & 0.2499 & 0.2503 \\ 0.2496 & 0.2506 & 0.2505 & 0.2493 \end{matrix} \end{matrix}$$

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

Finally, the coefficients in the characteristic polynomial

$$\text{poly}(A)$$

are

$$1 \quad -34 \quad -64 \quad 2176 \quad 0$$

These coefficients indicate 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. 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 the Statistics Toolbox™ software, 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:

$$B = \begin{array}{cccc} 7.5 & -5.5 & -6.5 & 4.5 \\ -3.5 & 1.5 & 2.5 & -0.5 \\ 0.5 & -2.5 & -1.5 & 3.5 \\ -4.5 & 6.5 & 5.5 & -7.5 \end{array}$$

sum(B)

$$\text{ans} = \begin{array}{cccc} 0 & 0 & 0 & 0 \end{array}$$

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:

$$B = \begin{array}{cccc} 7.5 & 0 & 0 & 4.5 \\ -3.5 & 0 & 0 & -0.5 \\ 0.5 & -2.5 & -1.5 & 3.5 \\ -4.5 & 6.5 & 5.5 & -7.5 \end{array}$$

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” on page 2-30
“Suppressing Output” on page 2-31
“Entering Long Statements” on page 2-32
“Command Line Editing” on page 2-32

The format Function

The format function controls the numeric format of the values displayed. The function affects only how numbers are displayed, not how MATLAB software 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
```

```
1.3333333333333333 0.00000123450000
```

```
format long e
    1.3333333333333333e+000    1.2345000000000000e-006

format long g
    1.3333333333333333        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 + sqrt(5))/2
```

You have misspelled sqrt. MATLAB responds with

```
Undefined function or variable 'sqrt'.
```

Instead of retyping the entire line, simply press the \uparrow key. The statement you typed is redisplayed. Use the \leftarrow key to move the cursor over and insert the missing r. Repeated use of the \uparrow key recalls earlier lines. Typing a few characters and then the \uparrow 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 Preferences” 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 <i>previous</i> line. Works only at command line.
↓	Ctrl+N	Recall <i>next</i> line. Works only at the prompt if you previously used the up arrow or Ctrl+P .
←	Ctrl+B	Move <i>back</i> one character.
→	Ctrl+F	Move <i>forward</i> one character.
Ctrl+→	None	Move <i>right</i> one word.
Ctrl+←	None	Move <i>left</i> one word.
Home	Ctrl+A	Move to beginning of current statement.
End	Ctrl+E	Move to end 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 (<i>kill</i>) 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.
Ctrl+C or Ctrl+Break	None	Stop execution of whatever is currently running, if feasible.

Graphics

- “Overview of Plotting” on page 3-2
- “Editing Plots” on page 3-19
- “Some Ways to Use Plotting Tools” on page 3-25
- “Preparing Graphs for Presentation” on page 3-39
- “Using Basic Plotting Functions” on page 3-51
- “Creating Mesh and Surface Plots” on page 3-65
- “Plotting Image Data” on page 3-73
- “Printing Graphics” on page 3-75
- “Understanding Handle Graphics Objects” on page 3-78

Overview of Plotting

In this section...
“Plotting Process” on page 3-2
“Graph Components” on page 3-6
“Figure Tools” on page 3-7
“Arranging Graphs Within a Figure” on page 3-14
“Choosing a Type of Graph to Plot” on page 3-15

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

Plotting Process

The MATLAB environment 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. You can choose from many predefined graph types, such as line, bar, histogram, and pie graphs as well as 3-D graphs, such as surfaces, slice planes, and streamlines.

There are two basic ways to create MATLAB graphs:

- Use plotting tools to create graphs interactively.

See “Some Ways to Use Plotting Tools” on page 3-25.

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

See “Using Basic Plotting Functions” on page 3-51.

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

After 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. You can also identify individual graph observations with `Datatip` and `Data` brushing tools and trace them back to their data sources in the MATLAB workspace.

For More Information See “Data Exploration Tools” in the MATLAB Graphics documentation and “Marking Up Graphs with Data Brushing”, “Making Graphs Responsive with Data Linking”, “Interacting with Graphed Data”, 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 The data values that a line graph (for example) displays are copied into and become properties of the (lineseries) graphics object. You can, therefore, change the data in the workspace without creating a new graph. You can also add data to a graph. See “Editing Plots” on page 3-19 for more information.

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 box 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 3-29 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 for Reuse

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 when you run it on a different type of computer. When you load a FIG-file, a new MATLAB figure opens in the same state as the one you saved.

Note The states of any figure tools on the toolbars are not saved in a FIG-file; only the contents of the graph and its annotations are saved. The contents of datatips (created by the Data Cursor tool), are also saved. This means that FIG-files always open in the default figure mode.

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 using MATLAB.

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

Graph Components

MATLAB graphs display 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.

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

See “Understanding Handle Graphics Objects” on page 3-78 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 3-39.

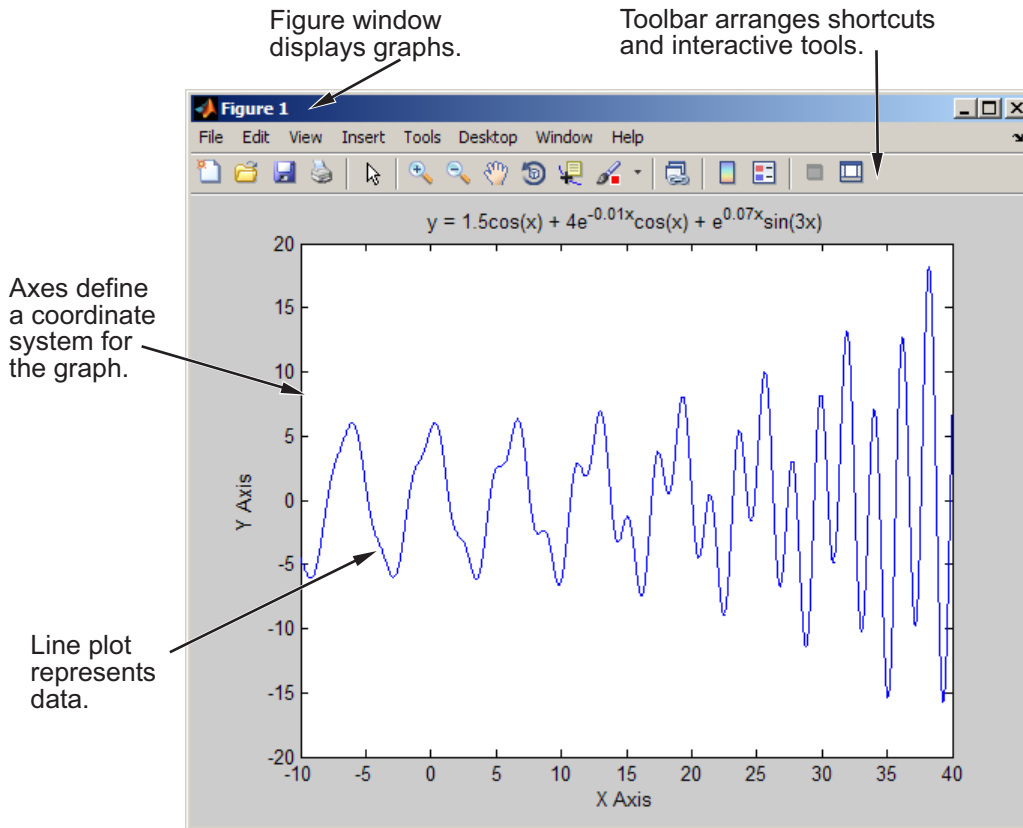
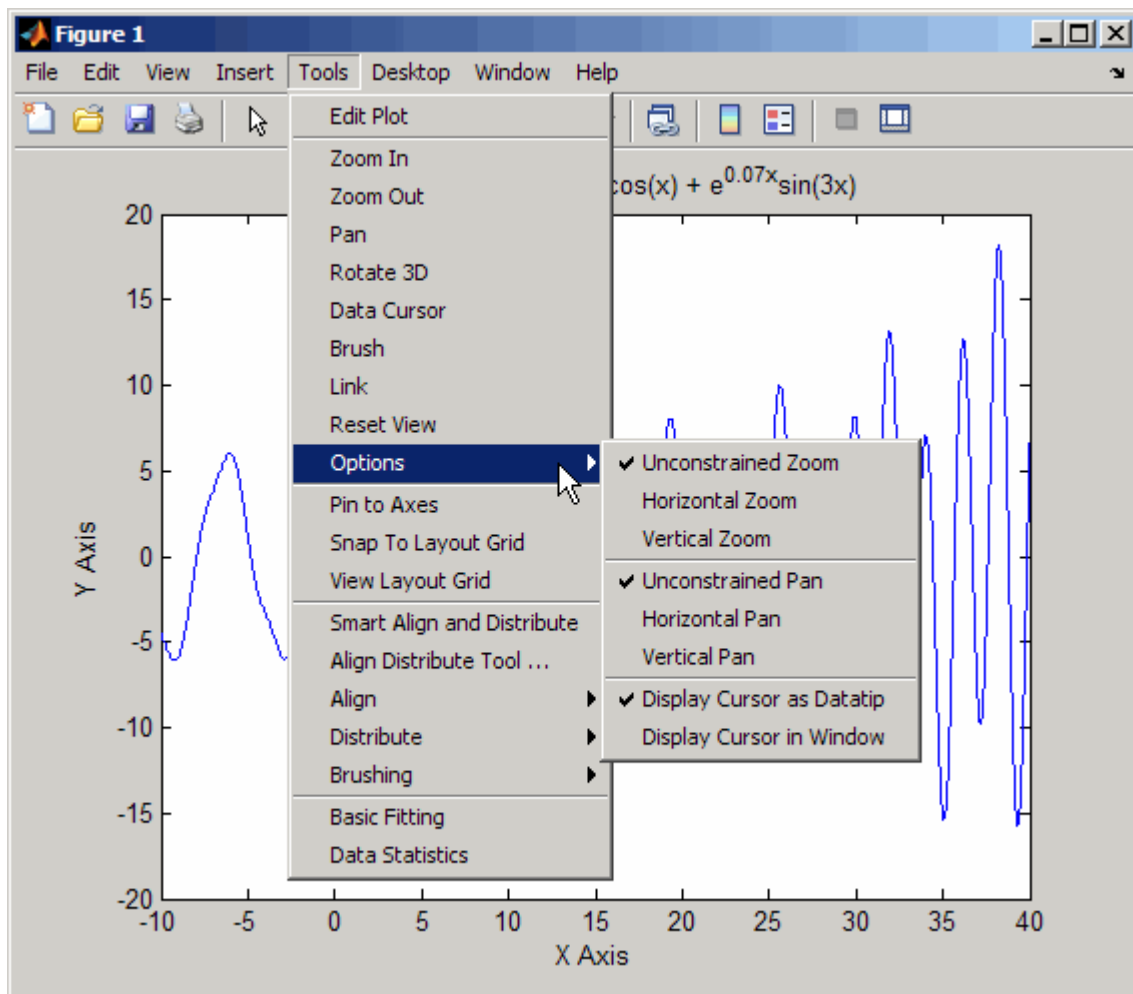


Figure Tools

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 in this figure also appear 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 3-9.



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.

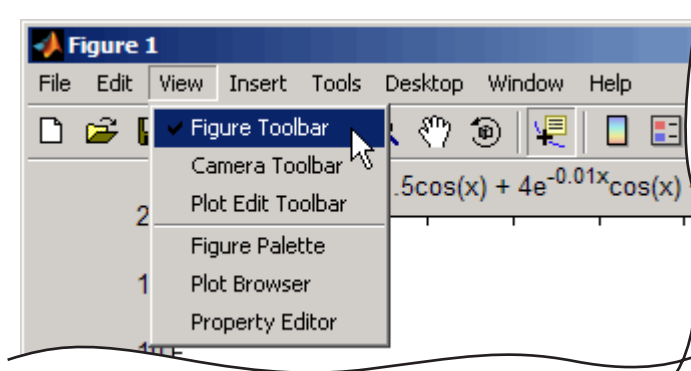
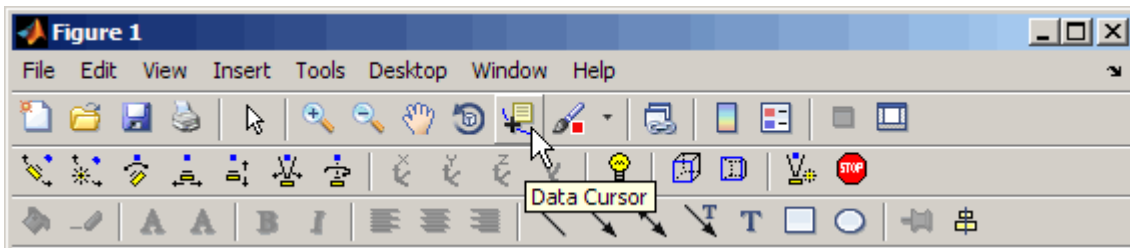


Figure Toolbars

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.



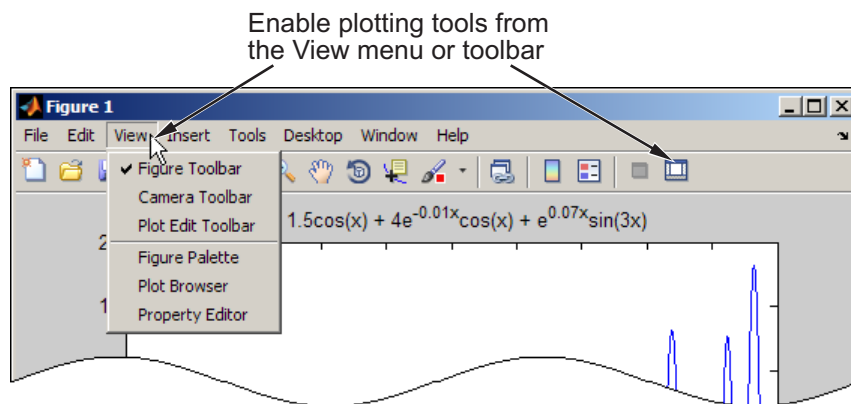
For More Information See “Anatomy of a Graph” in the MATLAB Graphics documentation.

Plotting Tools

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

- 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.

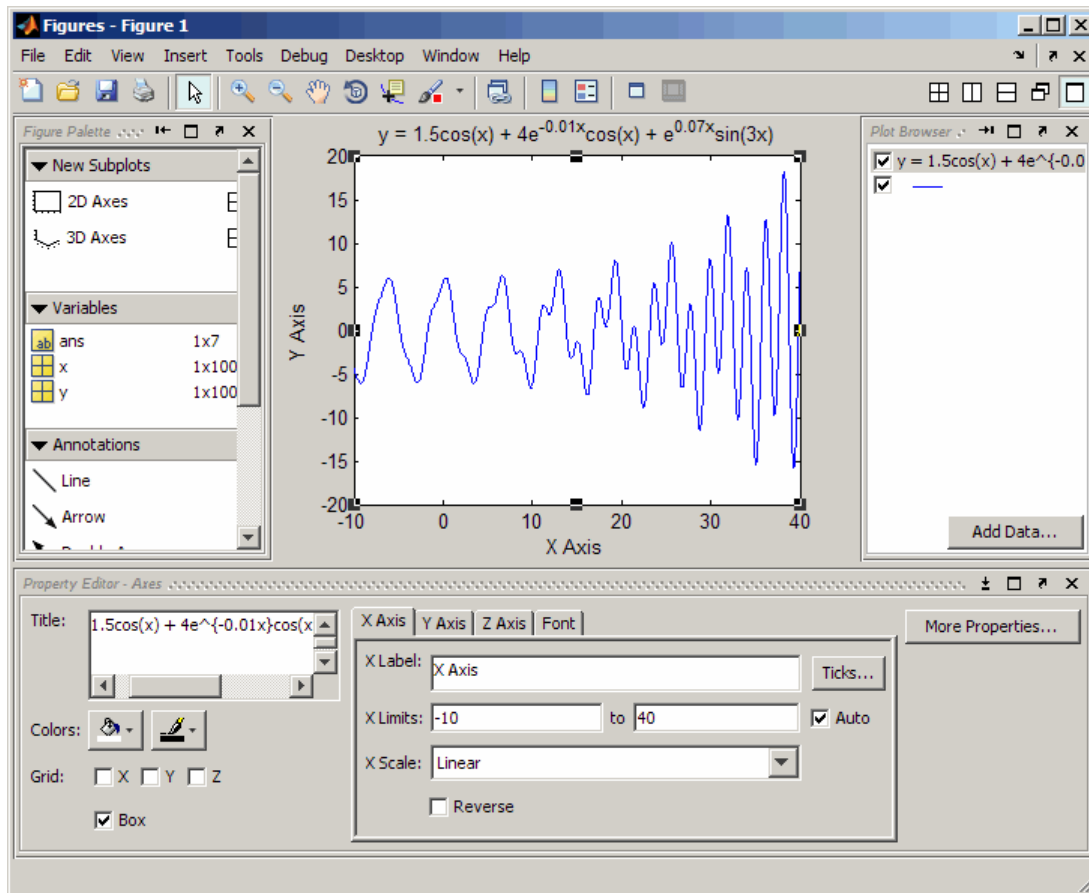
- Plot Browser — Select objects in the graphics hierarchy, control visibility, and add data to axes.
- Property Editor — Change key properties of the selected object. Click **More Properties** to access all object properties with the Property Inspector.

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

```
figurepalette  
plotbrowser  
propertyeditor
```

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

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

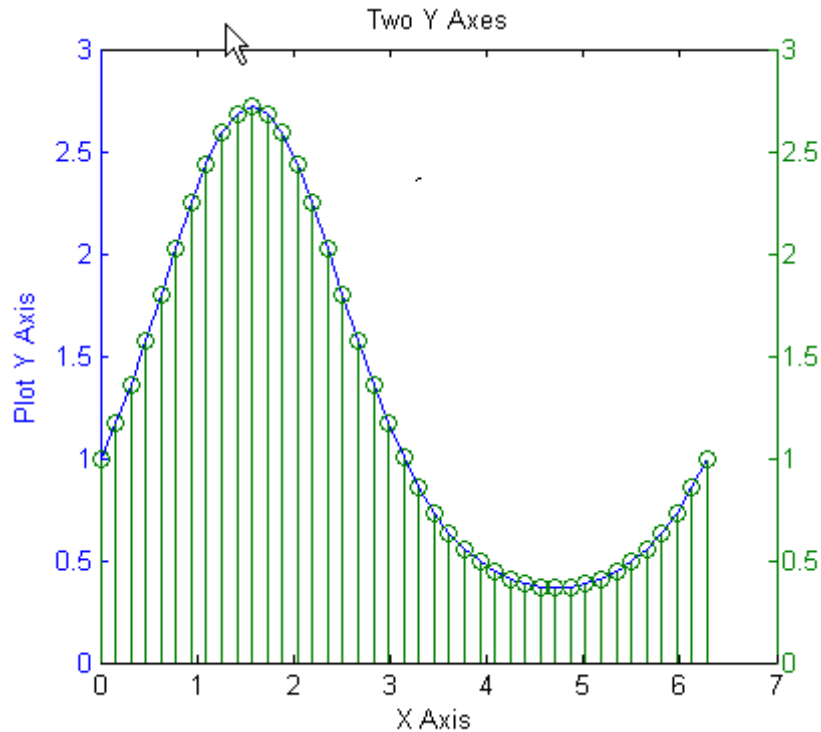


Using Plotting Tools and MATLAB Code

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:

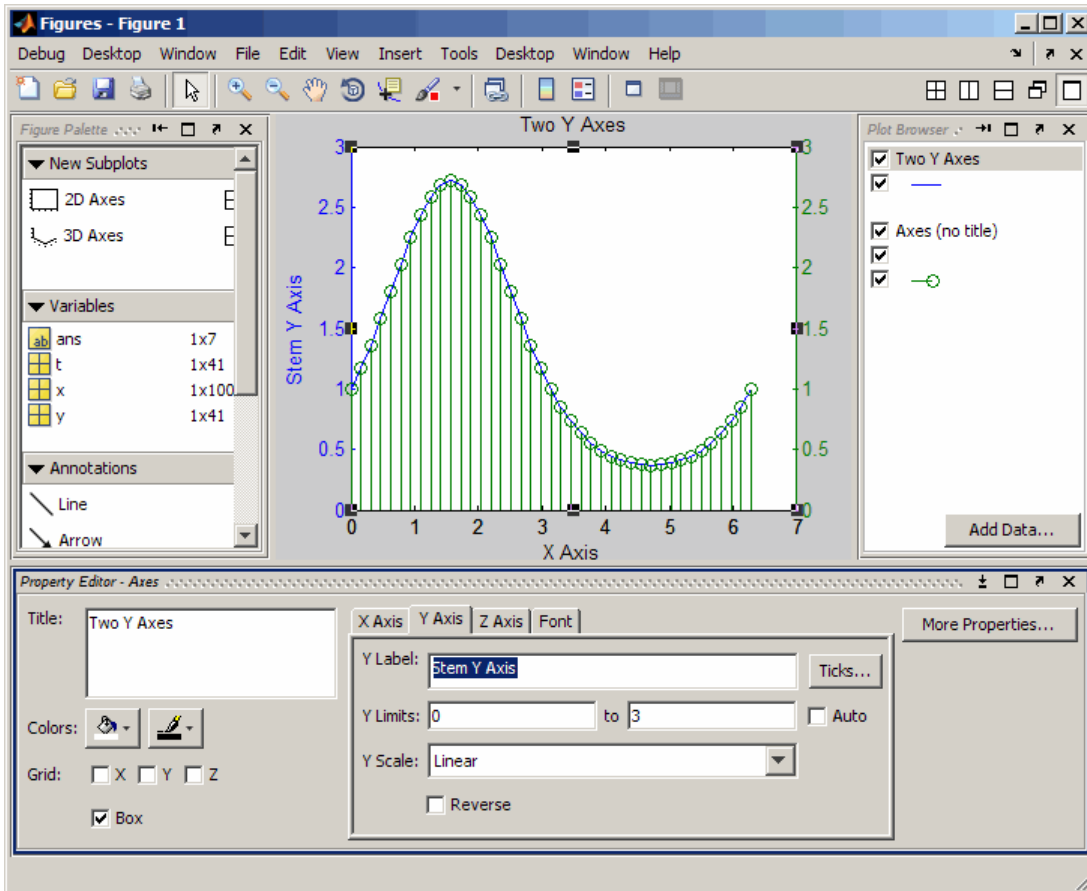
```
t = 0:pi/20:2*pi;
y = exp(sin(t));
plotyy(t,y,t,y,'plot','stem')
xlabel('X Axis')
ylabel('Plot Y Axis')
```

```
title('Two Y Axes')
```



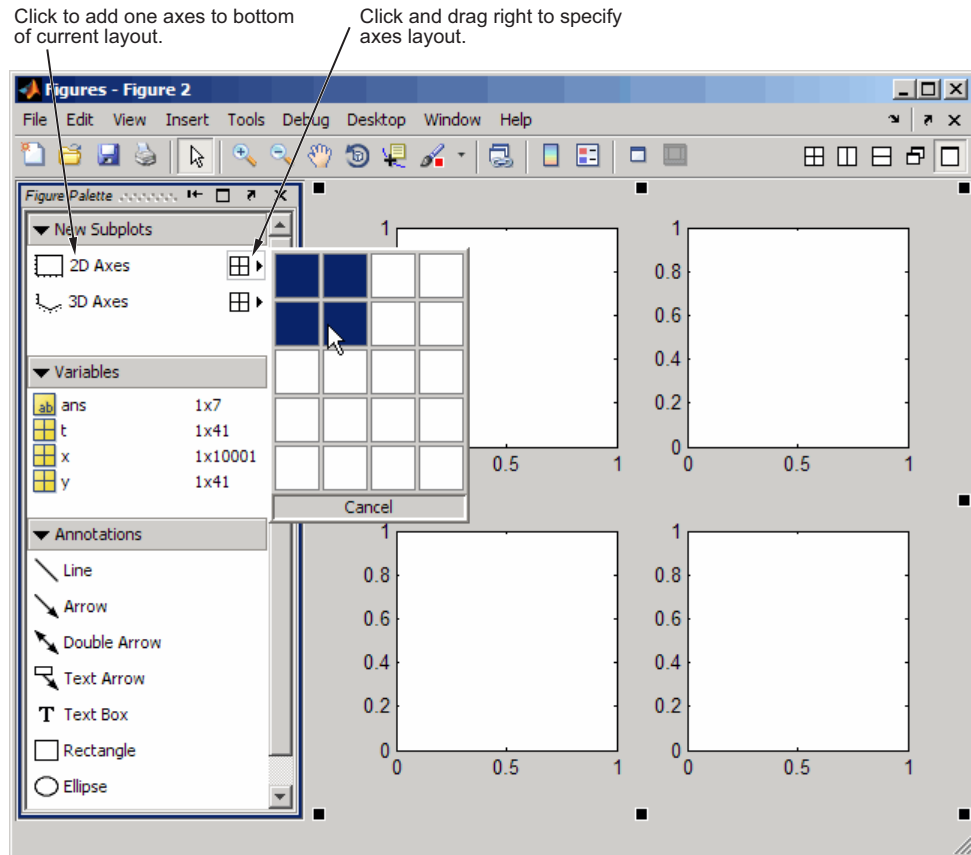
This graph contains two y-axes, one for each plot type (a lineseries and a stemseries). The plotting tools make it easy to select any of the objects that the graph contains and modify their properties.

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.

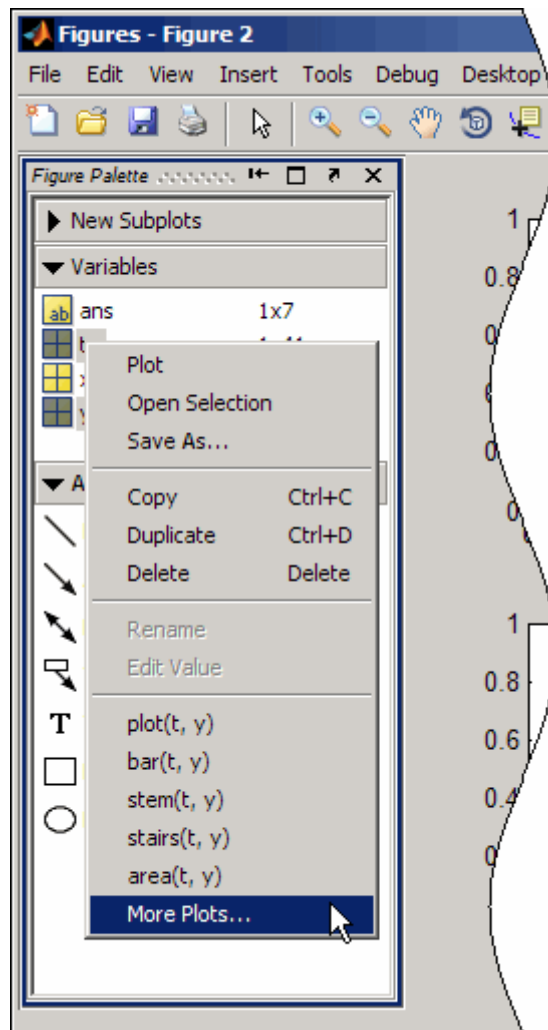
Choosing a Type of Graph to Plot

The many kinds of 2-D and 3-D graphs you can make are described in “Types of MATLAB Plots” 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 Variable 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 you select no data or nonnumeric data.

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



The Plot Catalog opens 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	plot	<p>2-D Line Graph</p> <p>2-D line graph using linear axes</p> <p>Vectors create a single line; matrices create one line per column.</p> <p>Plotted Variables</p> <ul style="list-style-type: none"> • 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. <p>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.</p> <p>More Information</p>
Stem and Stair Plots	ploty	
Bar Plots	semilogx	
Scatter Plots	semilogy	
Pie Charts	loglog	
Histograms	area	
Polar Plots	errorbar	
Contour Plots	plot3	
Images		
3D Surfaces		
Volumetrics		
Vector Fields		
Analytic Plots		

Select a category of graphs and then choose a specific type.

Editing Plots

In this section...

“Plot Edit Mode” on page 3-19

“Using Functions to Edit Graphs” on page 3-24

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:

Plot edit mode enabled

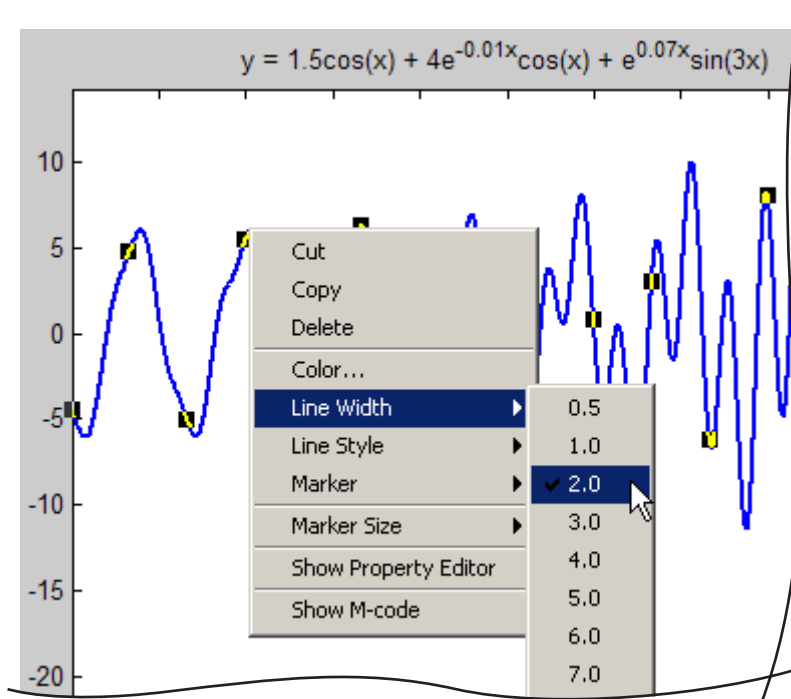


You can also select **Edit Plot** from the figure **Tools** menu.

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.

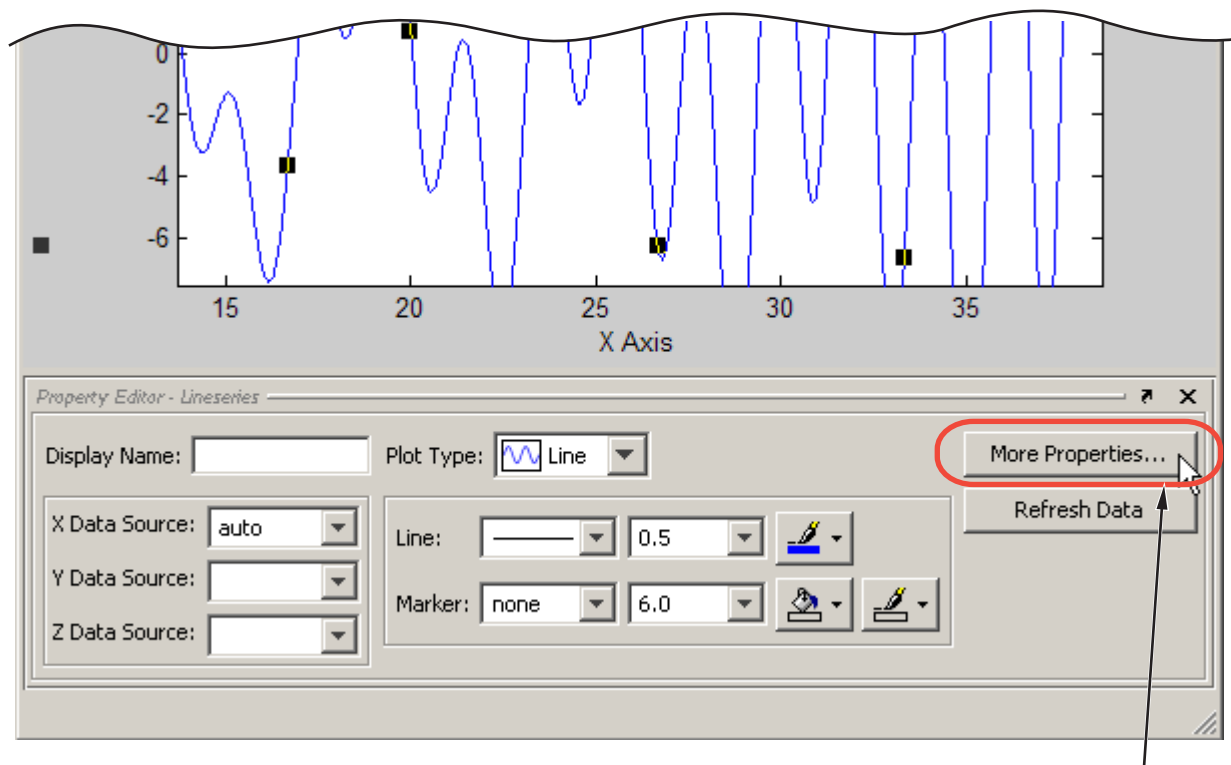
Right-click with the pointer over the selected object to display the object's context menu:



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. When you select an object, it updates to display the properties of whatever object you select.



Click to display Property Inspector

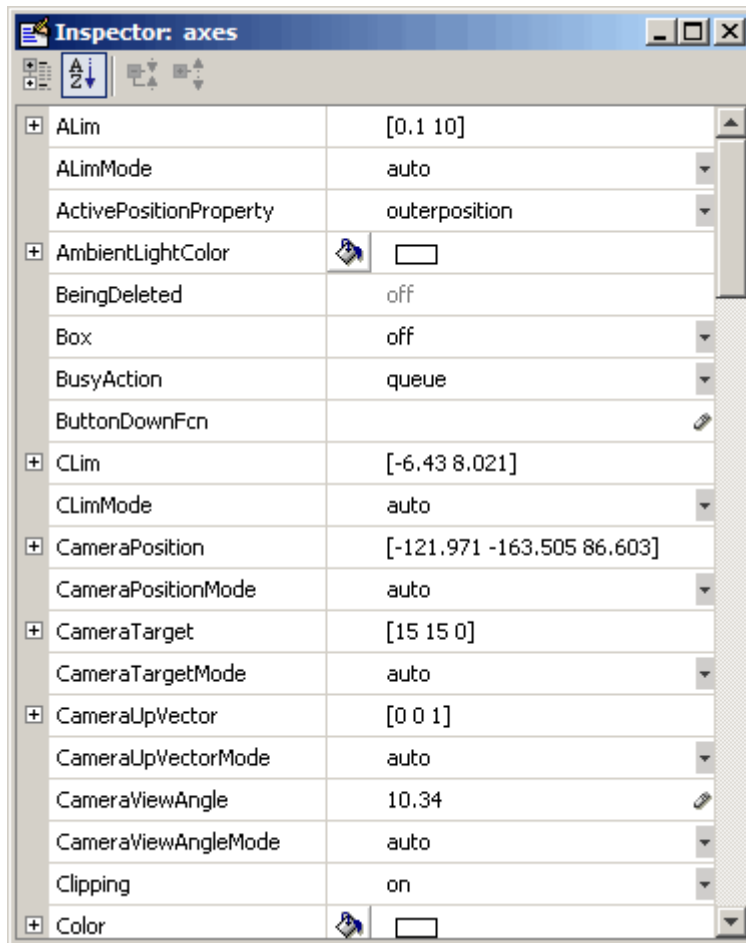
Accessing Properties with the Property Inspector


The *Property Inspector* is a tool that enables you to access most Handle Graphics properties 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

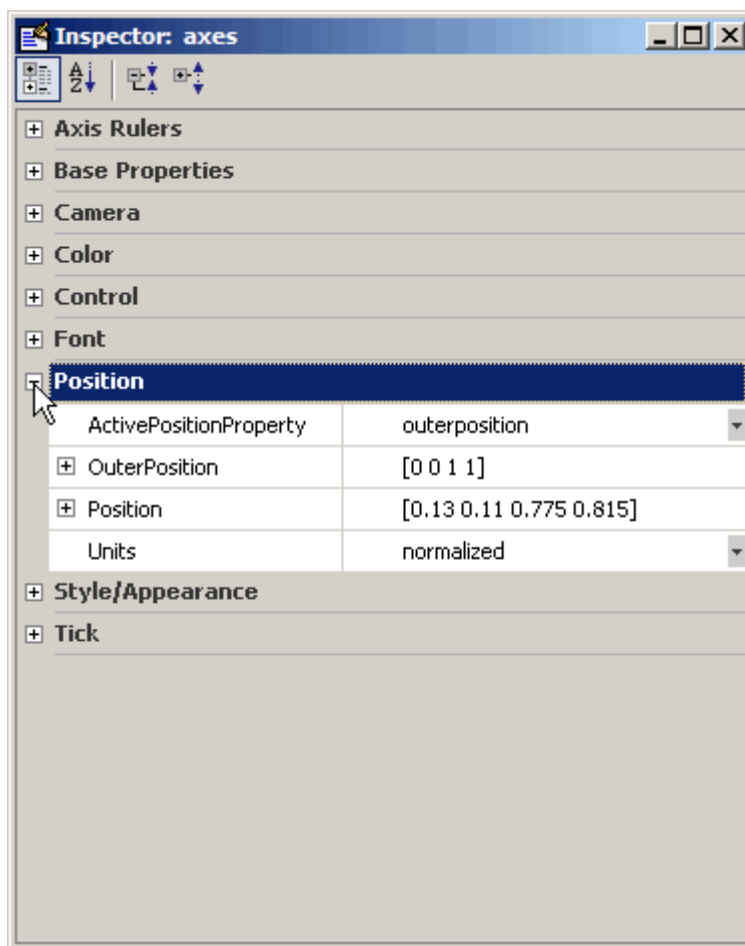
```
inspect(gca)
```

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 updates 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 so, 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 “Understanding Handle Graphics Objects” on page 3-78.

Some Ways to Use Plotting Tools

In this section...

“Plotting Two Variables with Plotting Tools” on page 3-25

“Changing the Appearance of Lines and Markers” on page 3-28

“Adding More Data to the Graph” on page 3-29

“Changing the Type of Graph” on page 3-32

“Modifying the Graph Data Source” on page 3-34

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 like this because the MATLAB software can distribute arithmetic operations over all elements of a multivalued variable.

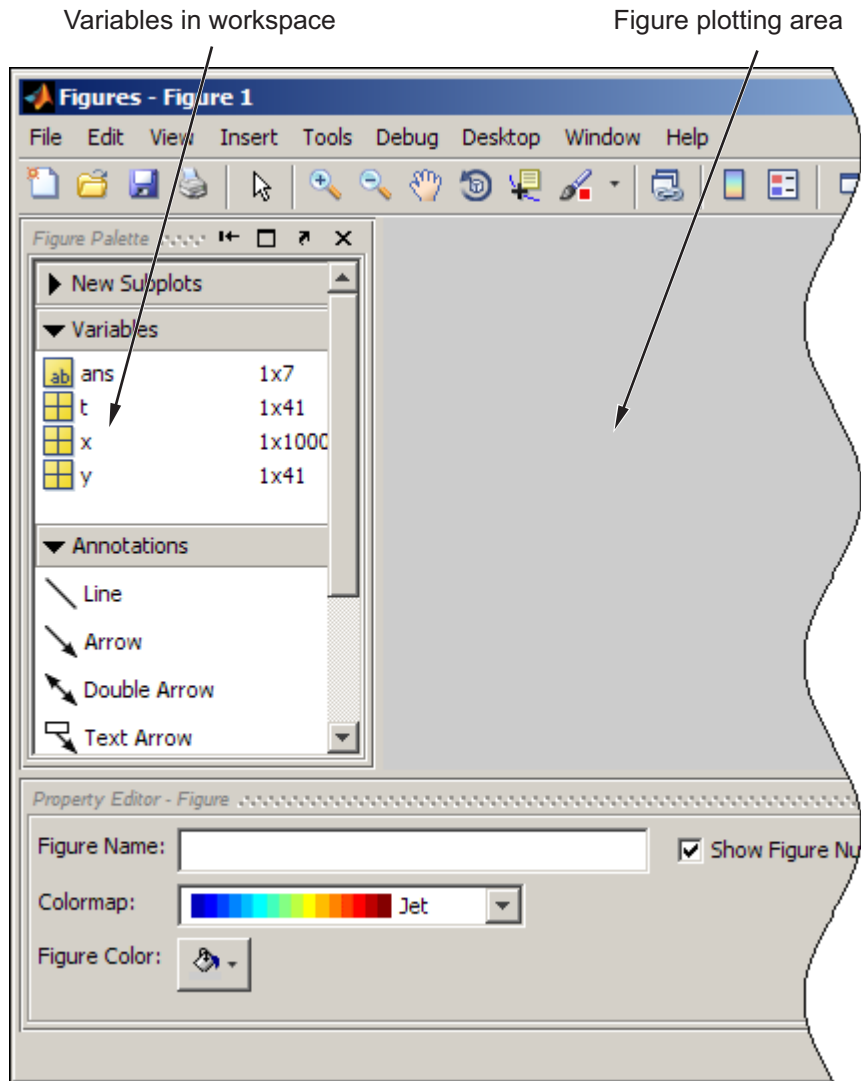
For example, the following statement creates a variable x that contains values ranging from -1 to 1 in increments of 0.1 (you could also use the `linspace` function to generate data for x). The second statement raises each value in x to the third power and stores these values in y :

```
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

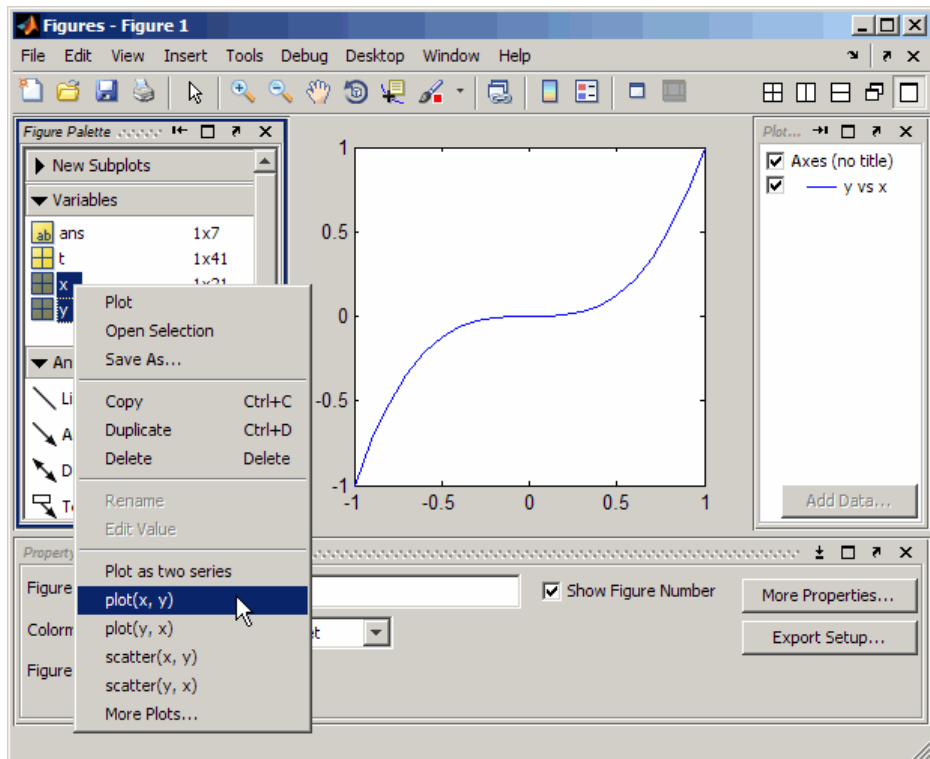
```
plottools
```

A figure displays with plotting tools attached.



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 or **Ctrl**+click if variables are not contiguous to select again), and then right-click to display the context menu.

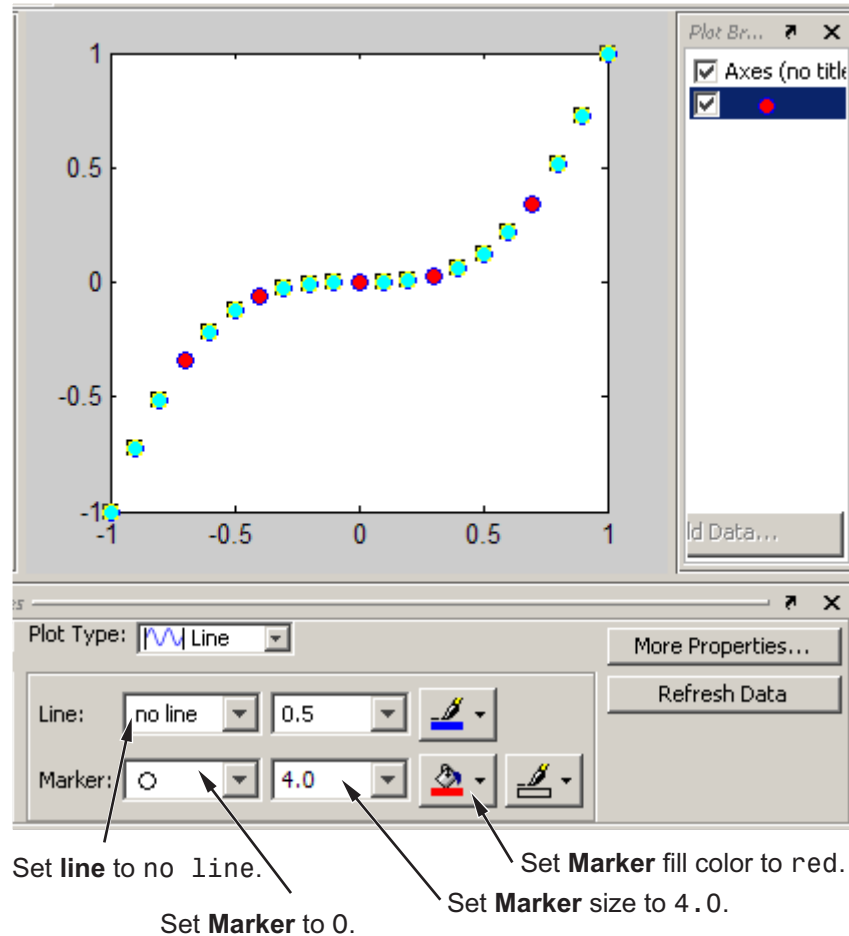


Select **plot(x, y)** from the menu. The line graph plots in the figure area. The black squares indicate that the line is selected and you can edit its properties with the Property Editor.

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

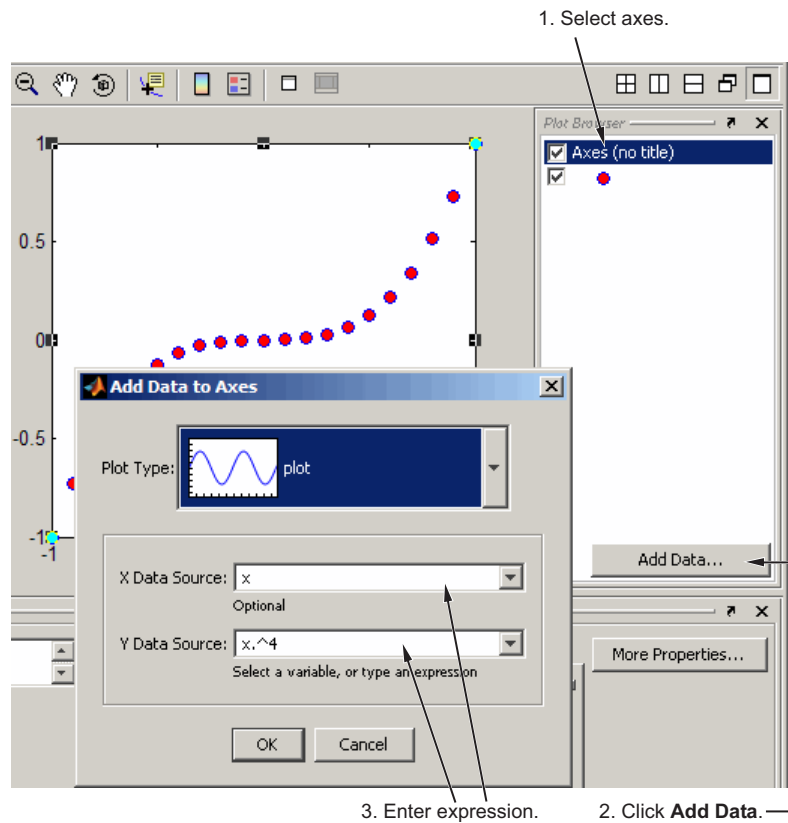
You can add more data to the graph by defining more variables or by specifying a MATLAB expression to generate data for the plot. This second approach makes it easy to explore variations of the data already plotted.

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, new data plots are always added to the existing graph, instead of replacing the graph, as it

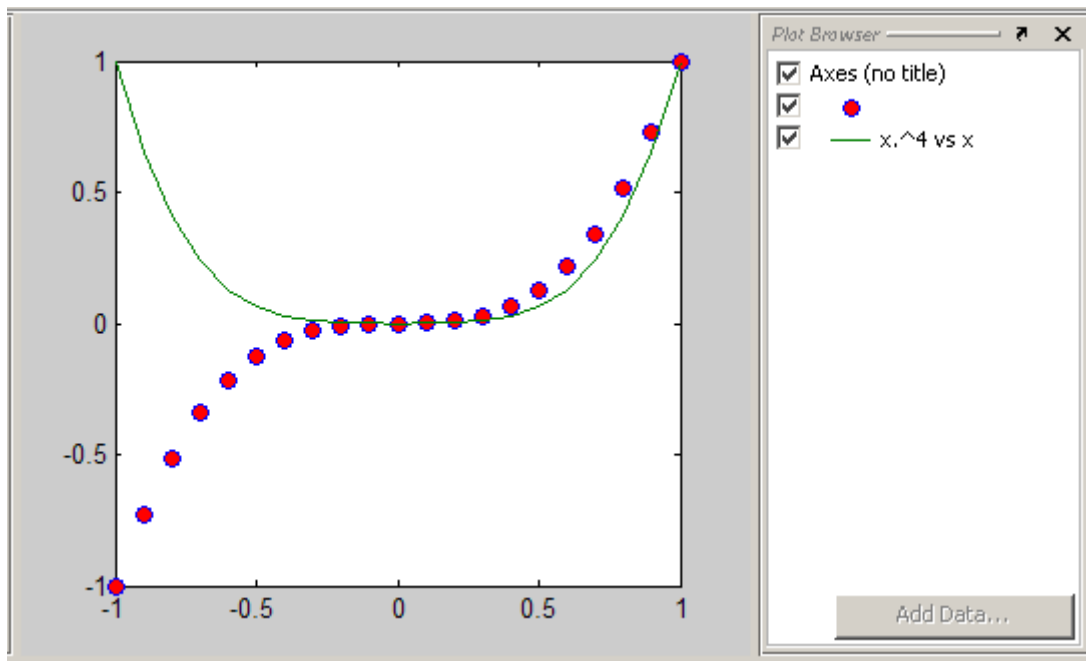
would if you issued repeated plotting commands. Thus, the plotting tools are in a hold on state.

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.



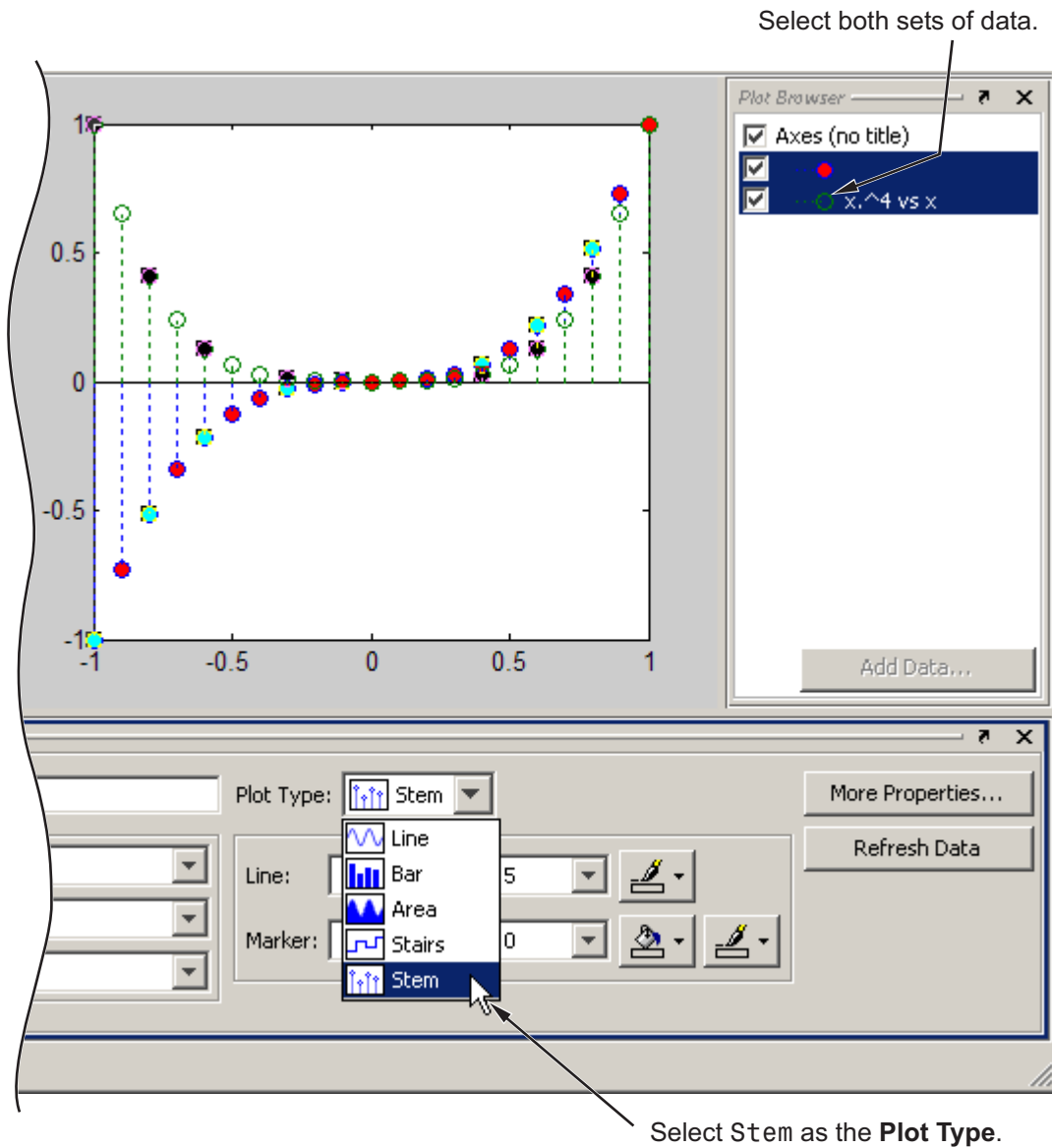
The preceding figure shows how to use the Add Data to Axes dialog box to create a line plot of $y = x^4$, which is added to the existing plot of $y = x^3$. The resulting plot appears with the Plot Browser, as shown in the following figure:



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.



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.)

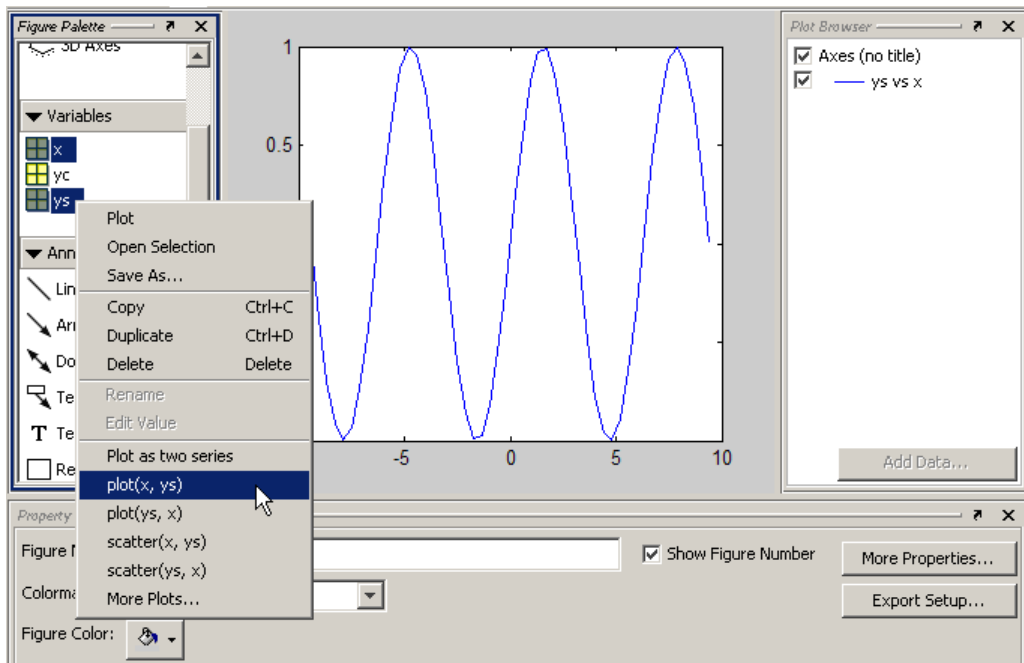
- 1 Define 50 points between -3π and 3π and compute their sines and cosines:

```
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)`:

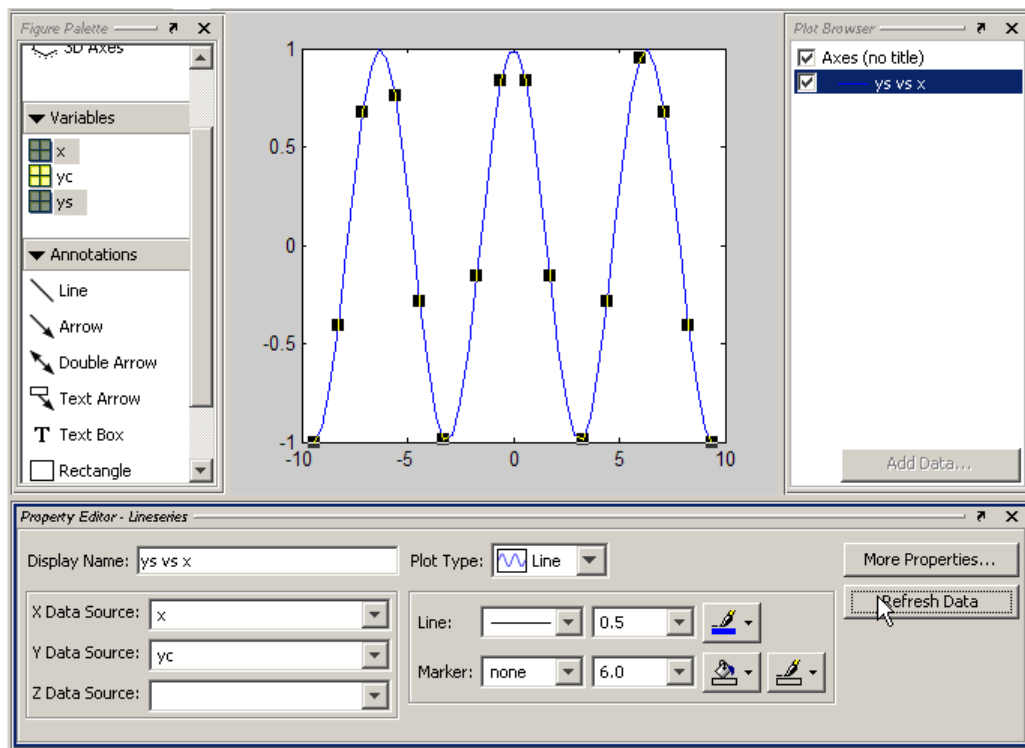
```
figure  
plottools
```

- 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 the following figure shows.



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

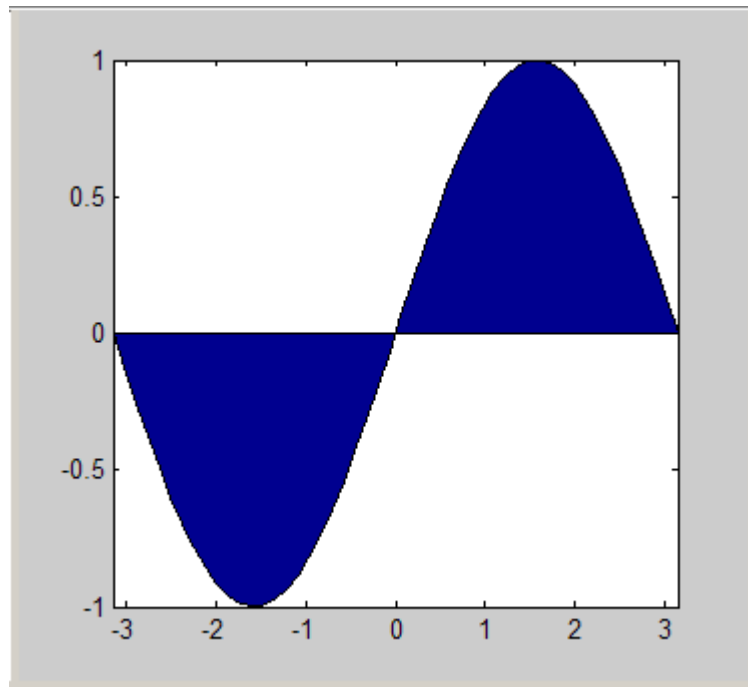
- 1 Select the line ys vs x in the Plot Browser or by clicking it.
- 2 In the Property Editor, select yc in the **Y Data Source** drop-down menu.
- 3 Click the **Refresh Data** button;
the plot will change to display a plot of yc vs x .



Providing New Values for the Data Source

Data values that define the graph are copies of 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.

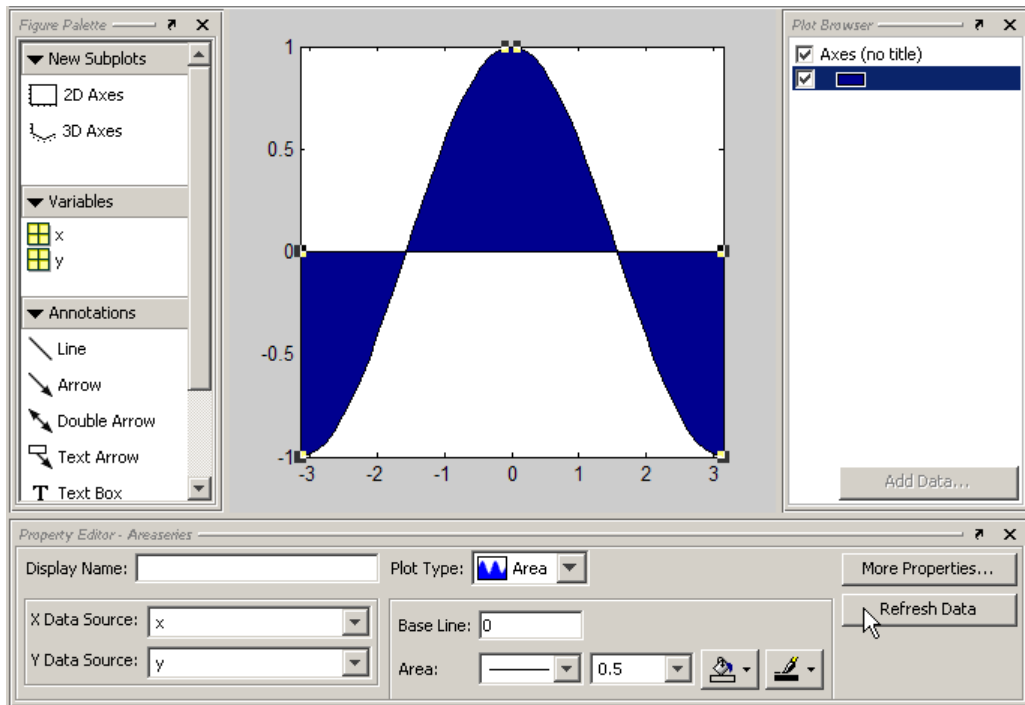
```
x = linspace(-pi,pi,50); % Define 50 points between - $\pi$  and  $\pi$ 
y = sin(x);
area(x,y) % Make an area plot of x and y
```



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.



Preparing Graphs for Presentation

In this section...

“Annotating Graphs for Presentation” on page 3-39

“Printing the Graph” on page 3-44

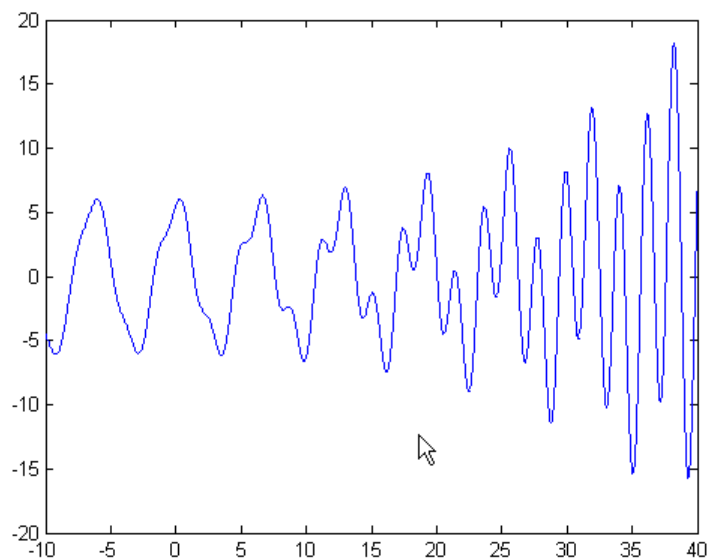
“Exporting the Graph” on page 3-48

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 figure shows the graph created by the previous code.

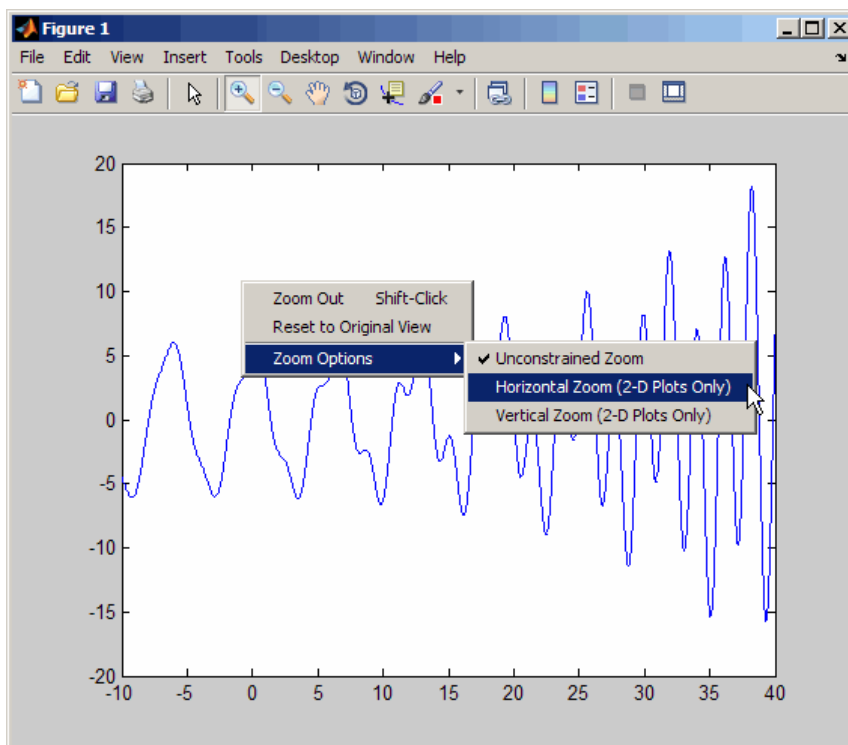



Now, suppose you want to save copies of the graph by


- 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

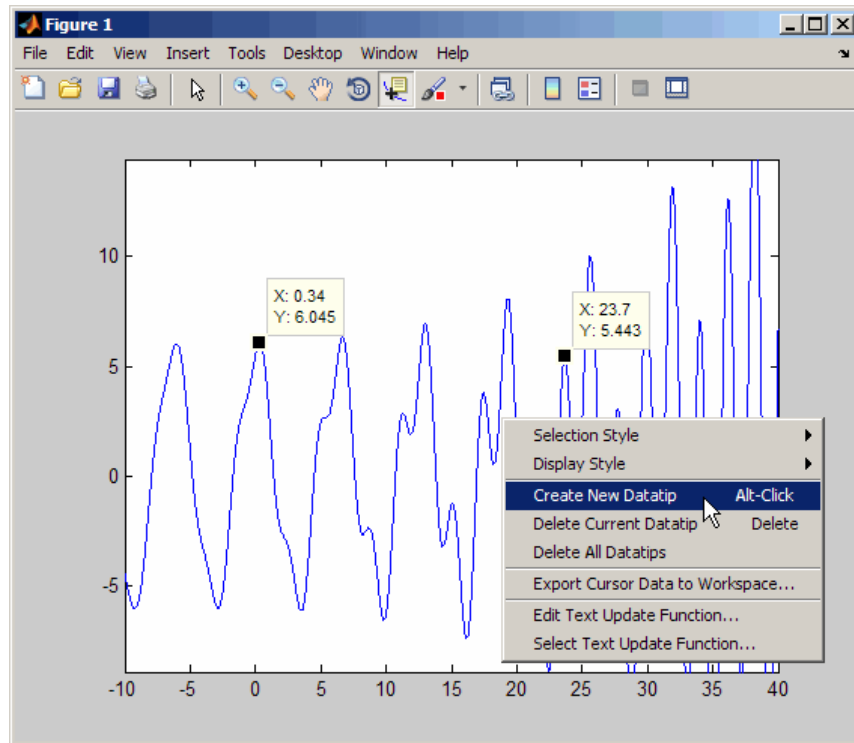
To obtain a better view, zoom in on the graph using horizontal zoom.

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**. You can reverse your zoom direction by doing **Shift+left-click**, or using the context menu.

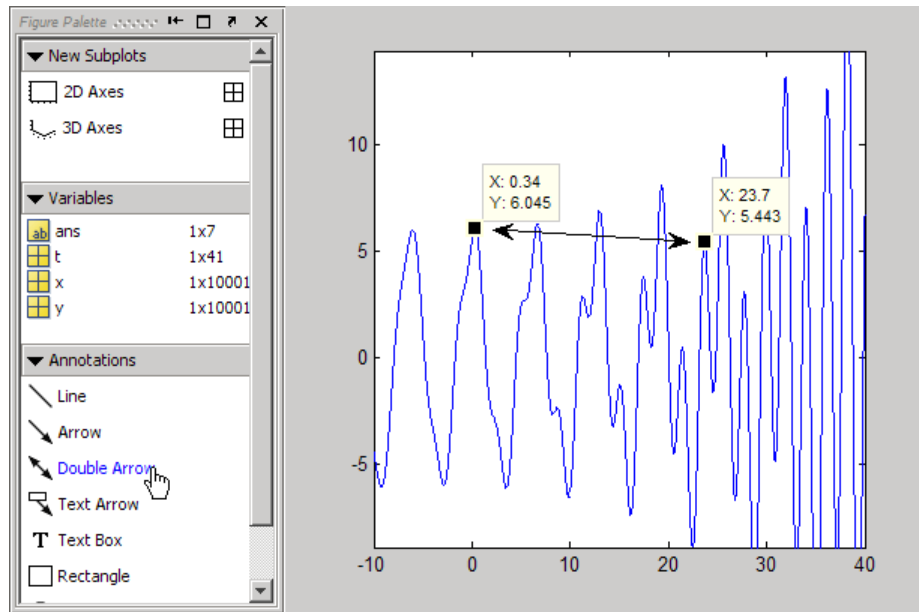


Left-click to zoom in on a region of the graph and use the **Pan** tool  to position the points of interest where you want them on the graph.

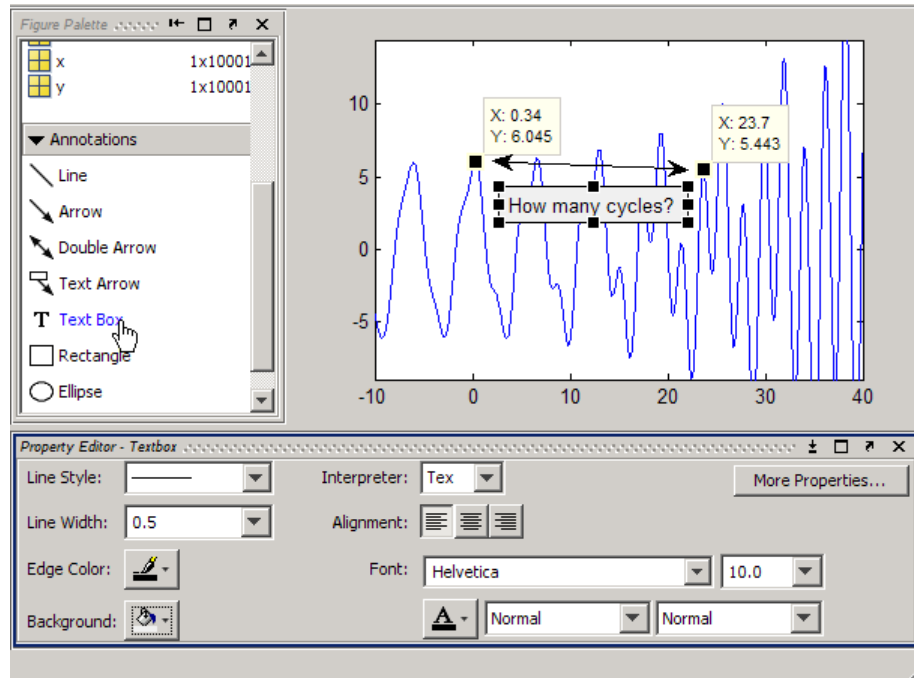
Label some key points with data tips using the **Data Cursor** tool . Left-clicking the line moves the last datatip you created to where you just clicked. To create a new datatip, press **Alt+click** or use the tool's context menu. See “Data Cursor — Displaying Data Values Interactively” in the MATLAB 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 in the following figure.



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, color, and also the text box line and background colors.

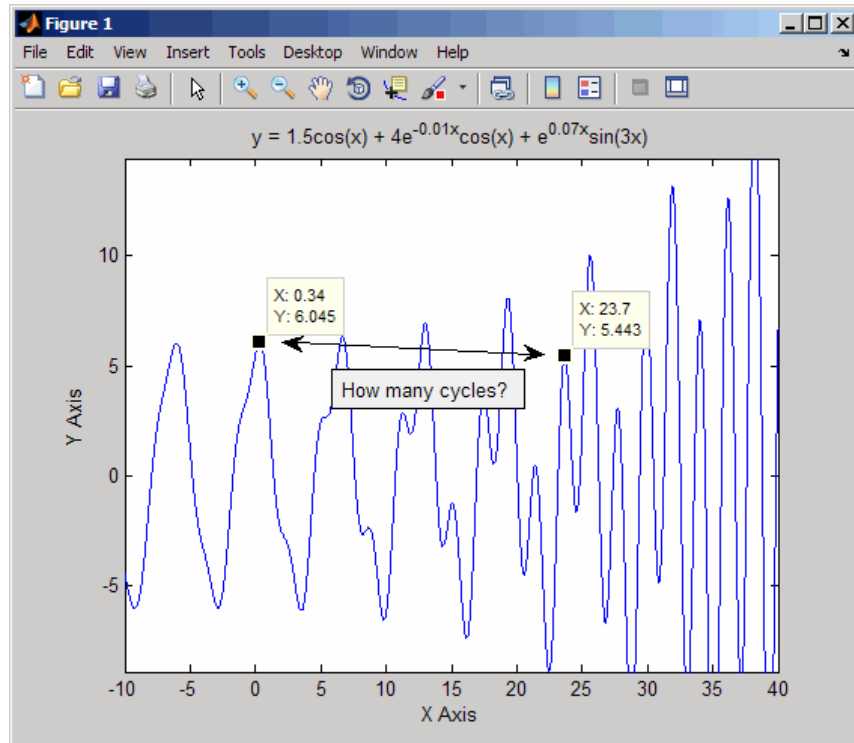


Finally, add text annotations, axis labels, and a title. You can add the title and axis labels using the following commands:

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

Note The text string passed to `title` uses T_EX syntax to produce the exponents. See “Information About Using T_EX” in the Text Properties reference page for more about using T_EX syntax to produce mathematical symbols.

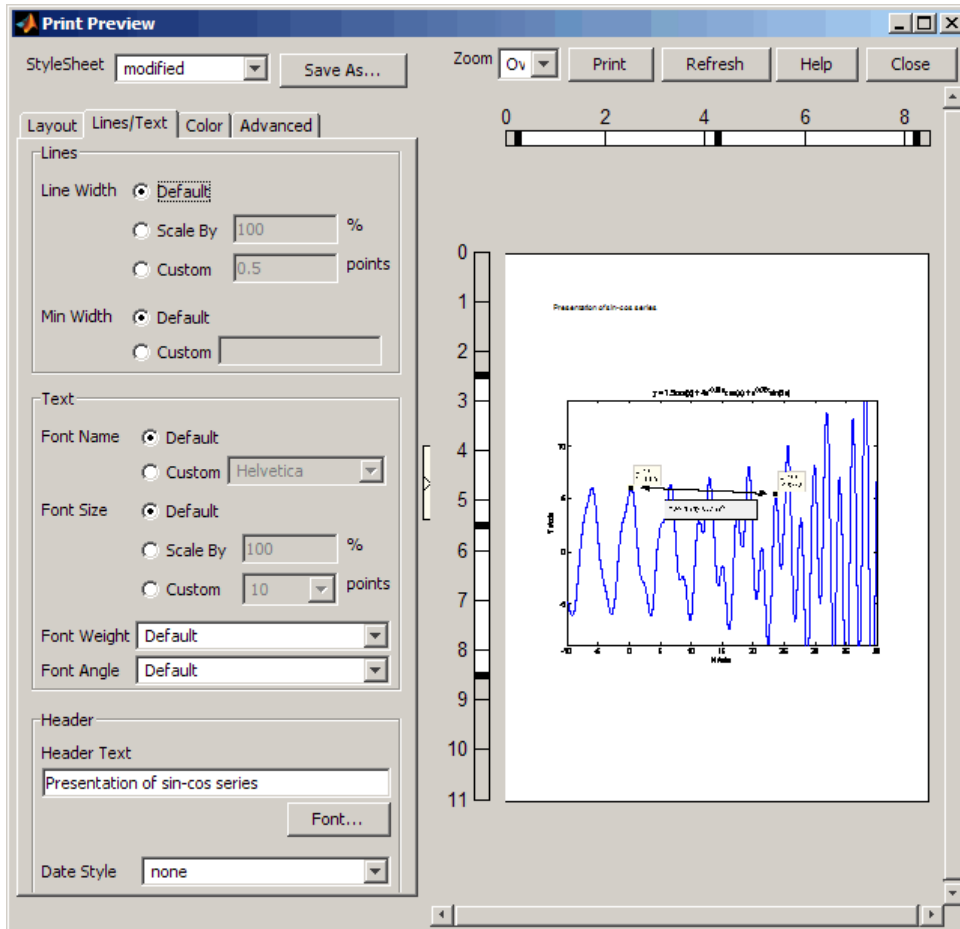
You can also add these annotations by selecting the axes and typing the above strings into their respective fields in the Property Editor. The graph is now ready to print and export.



Printing the Graph

Before printing the graph, select **File > Print Preview** to view 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.



- The rulers along the left and top sides of the preview pane contain three black handlebars. 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.

The image shows a dialog box with four tabs: 'Layout', 'Lines/Text', 'Color', and 'Advanced'. The 'Advanced' tab is active. It contains three main sections: 'Placement', 'Paper', and 'Units/Orientation'.

- Placement:** Two radio buttons are present. The first, labeled 'Auto (Actual Size, Centered)', is selected. The second is 'Use manual size and position'. Below these are four input fields: 'Left: 1.33', 'Top: 3.31', 'Width: 5.83', and 'Height: 4.38'. At the bottom of this section are four buttons: 'Use defaults', 'Fill page', 'Fix aspect ratio', and 'Center'.
- Paper:** A dropdown menu for 'Format' is set to 'USLetter'. Below it are three input fields: 'Width: 8.50', 'Height: 11.00', and an unlabeled field.
- Units:** Three radio buttons are shown: 'Inches' (selected), 'Centimeters', and 'Points'.
- Orientation:** Three radio buttons are shown: 'Portrait' (selected), 'Landscape', and 'Rotated'.

- By default, the locations of the axes tick marks are recalculated 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

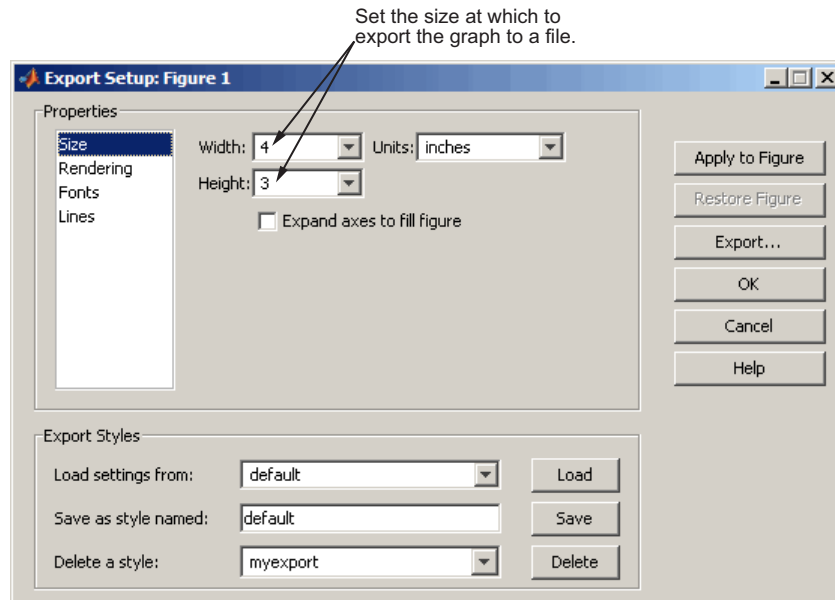
Exporting a graph is the process of creating a standard graphics file format of the graph (such as EPS or TIFF), which you can then import into other applications like word processors, drawing packages, etc.

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.

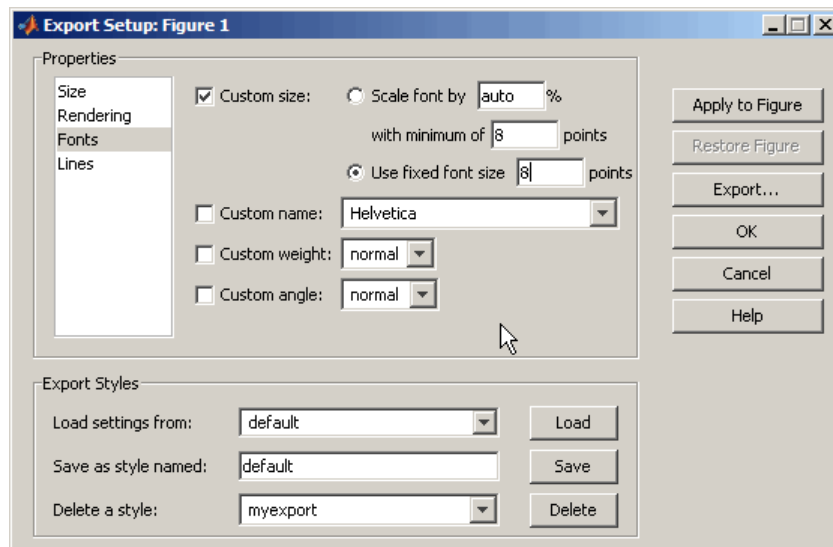
Specifying the Size of the Graph

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. A standard Save As dialog box opens 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” on page 3-51
“Plotting Multiple Data Sets in One Graph” on page 3-52
“Specifying Line Styles and Colors” on page 3-53
“Plotting Lines and Markers” on page 3-54
“Graphing Imaginary and Complex Data” on page 3-56
“Adding Plots to an Existing Graph” on page 3-57
“Figure Windows” on page 3-58
“Displaying Multiple Plots in One Figure” on page 3-59
“Controlling the Axes” on page 3-61
“Adding Axis Labels and Titles” on page 3-62
“Saving Figures” on page 3-63

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`.

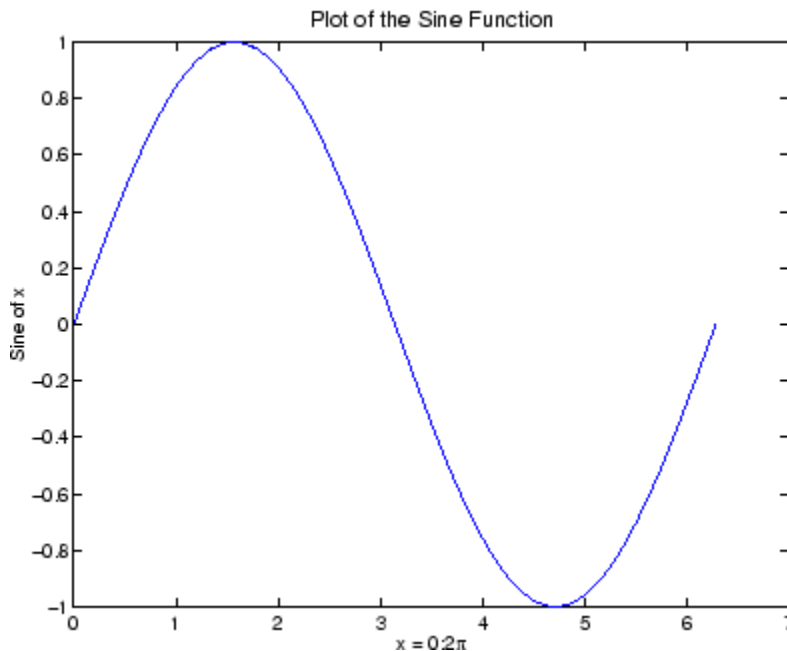
For example, these statements use the colon operator to create a vector of `x` values ranging from 0 to 2π , compute the sine of these values, and plot the result:

```
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

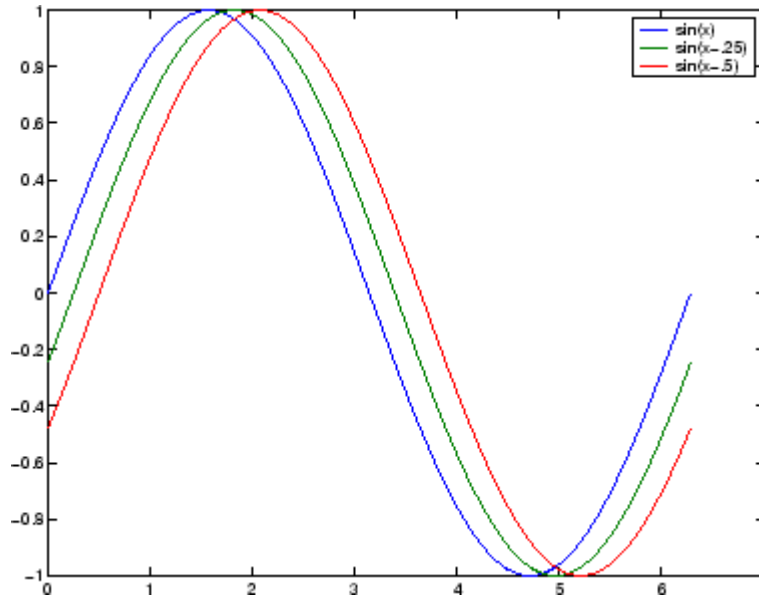
Multiple x-y pair arguments create multiple graphs with a single call to `plot`. `plot` automatically cycle through a predefined (but customizable) list of colors to allow discrimination among sets of data. See the axes `ColorOrder` and `LineStyleOrder` properties.

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. The strings are composed of combinations of the following elements:

Type	Values	Meanings
Color	'c' 'm' 'y' 'r' 'g' 'b' 'w' 'k'	cyan magenta yellow red green blue white black
Line style	'-' '--' ':' '.-' no character	solid dashed dotted dash-dot no line
Marker type	'+' 'o' '*' 'x' 's' 'd' '^' 'v' '>' '<' 'p' 'h' no character or none	plus mark unfilled circle asterisk letter x filled square filled diamond filled upward triangle filled downward triangle filled right-pointing triangle filled left-pointing triangle filled pentagram filled hexagram no marker

You can also edit color, line style, and markers interactively. See “Editing Plots” on page 3-19 for more information.

Plotting Lines and Markers

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

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

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

The statement

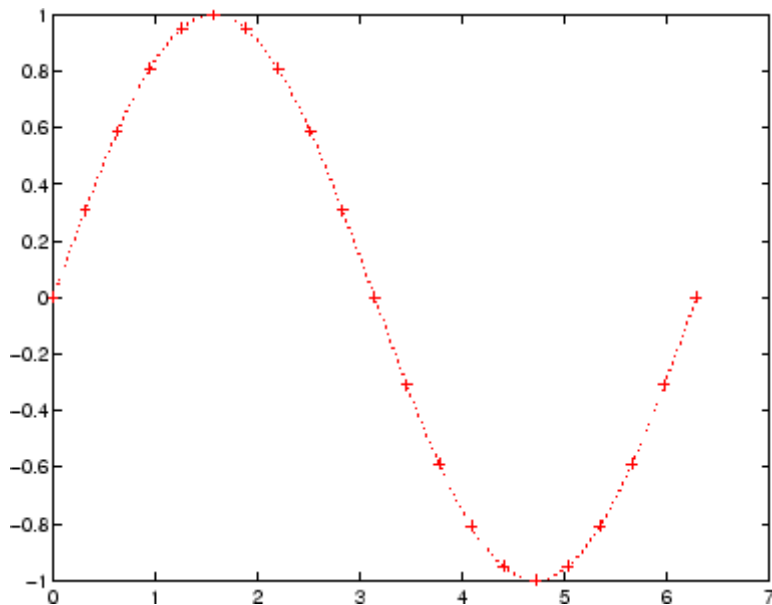
```
plot(x,y,'r:+')
```

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)
```

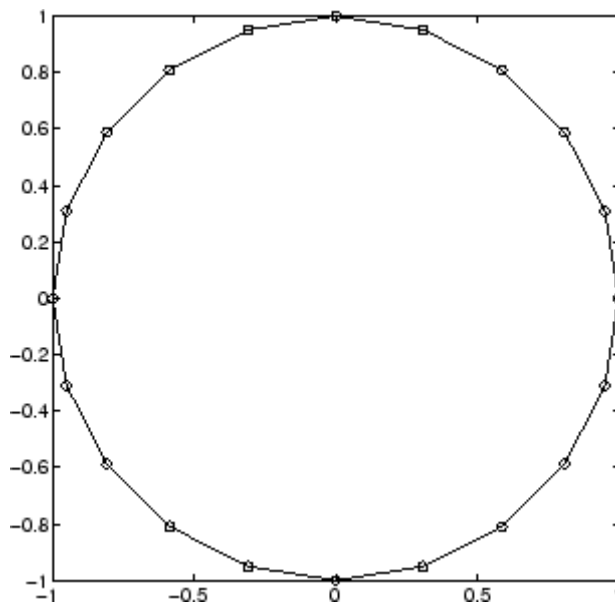
where Z is a complex vector or matrix, is equivalent to

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

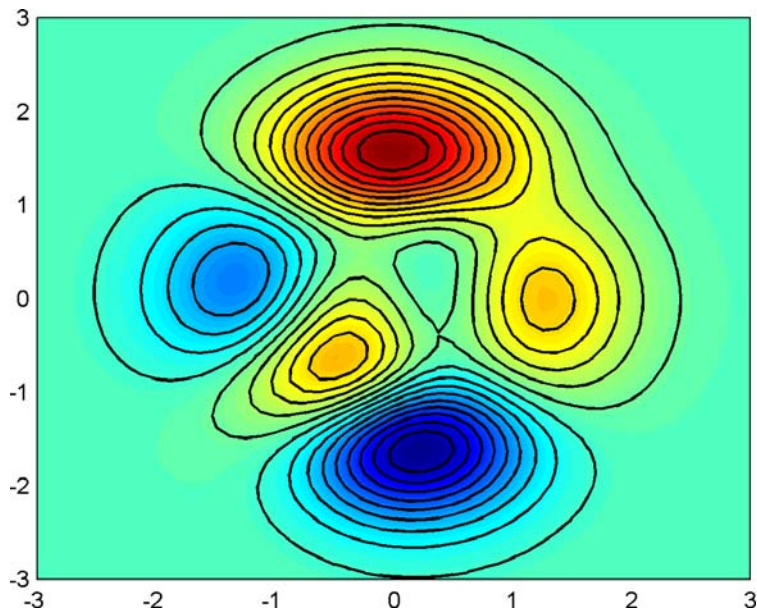
```
hold on
```

Now 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, it is used for graphics output. If there are multiple figure windows open, the one that is designated the “current figure” (the last figure used or clicked in) is used. See documentation for the `gcf` function for more information.

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.

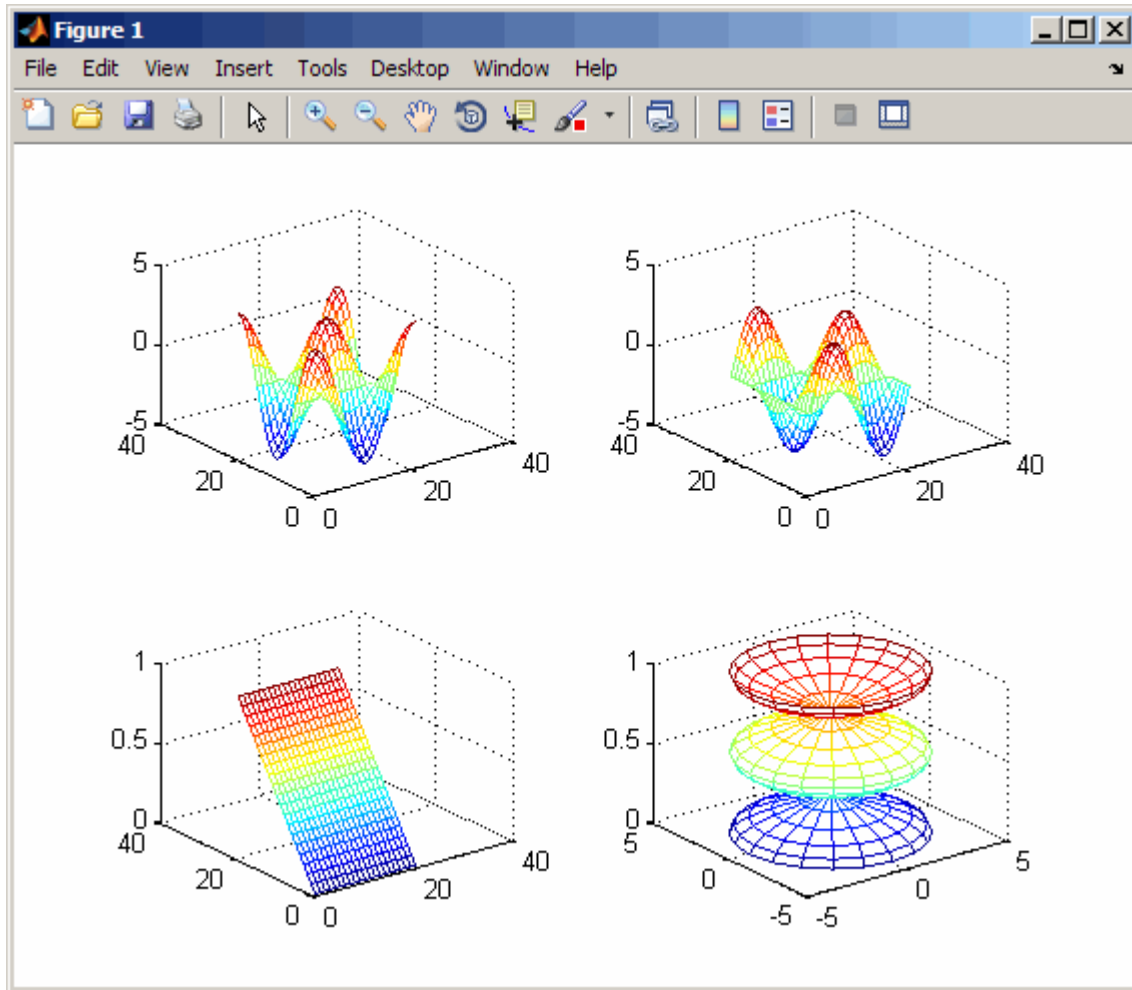
Displaying Multiple Plots in One Figure

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 3-19 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 enable automatic limit selection again.

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.

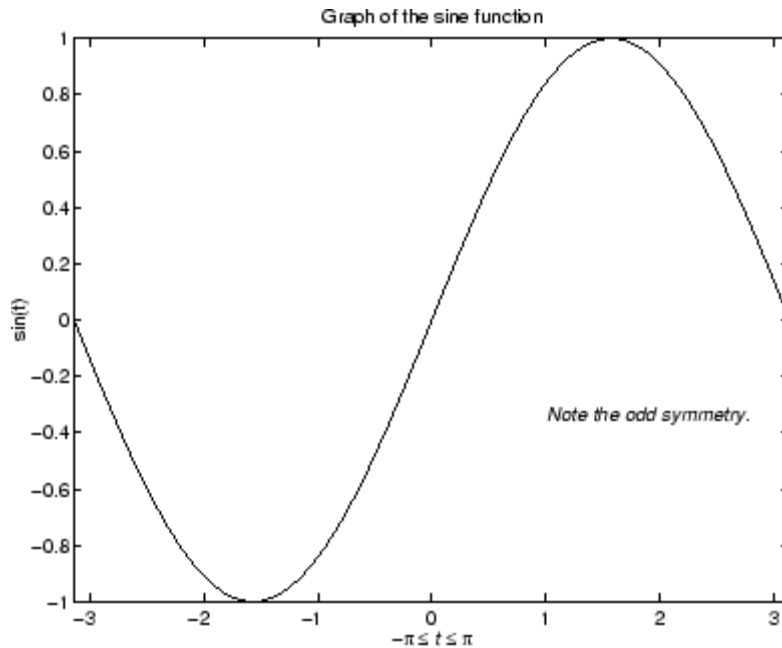
Adding Axis Labels and Titles

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 t \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 3-19 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. This saves the figure, the data within it, and all annotations (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 3-48 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 MATLAB Software Version

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” on page 3-65

“Visualizing Functions of Two Variables” on page 3-65

About Mesh and Surface Plots

You define 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 the surface is colored.

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.

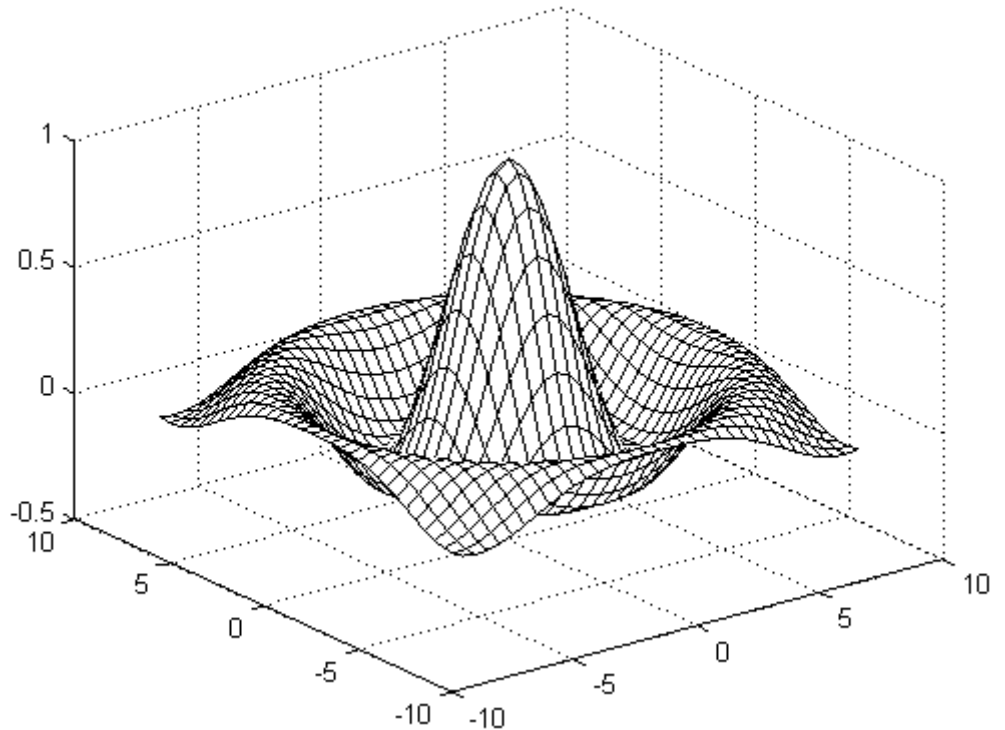
The `meshgrid` function transforms the domain specified by a single vector or two vectors x and y into matrices X and Y for use in evaluating functions of two variables. The rows of X are copies of the vector x and the columns of Y are copies of the vector y .

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, the current colormap is used to color the mesh. 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:

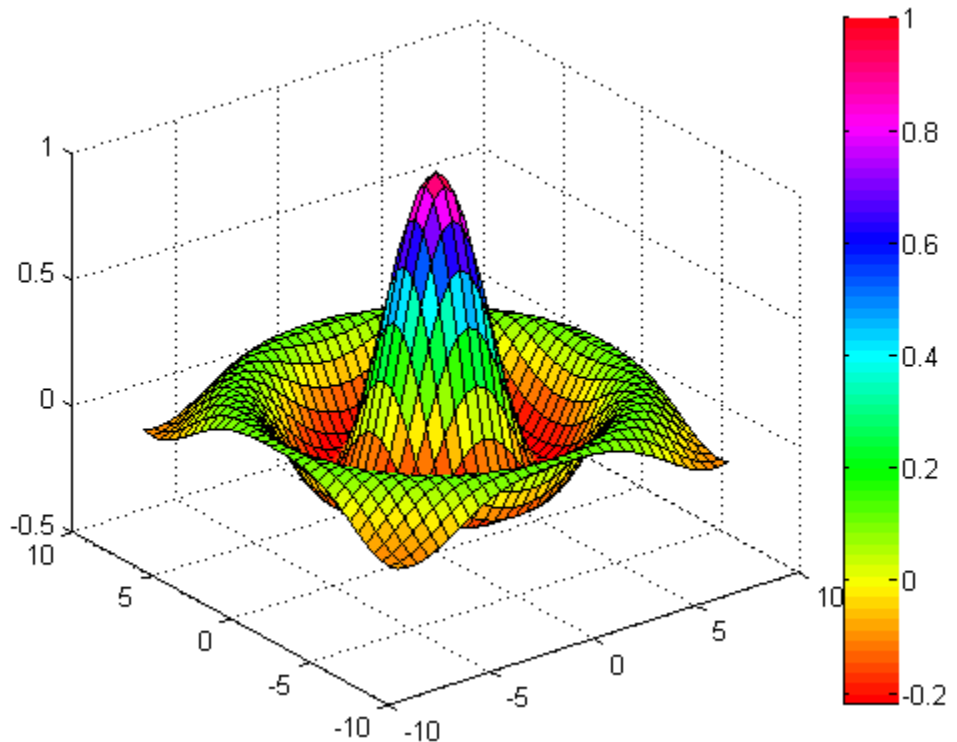
```
hidden off
```

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 the rectangular faces of the surface are colored. 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.

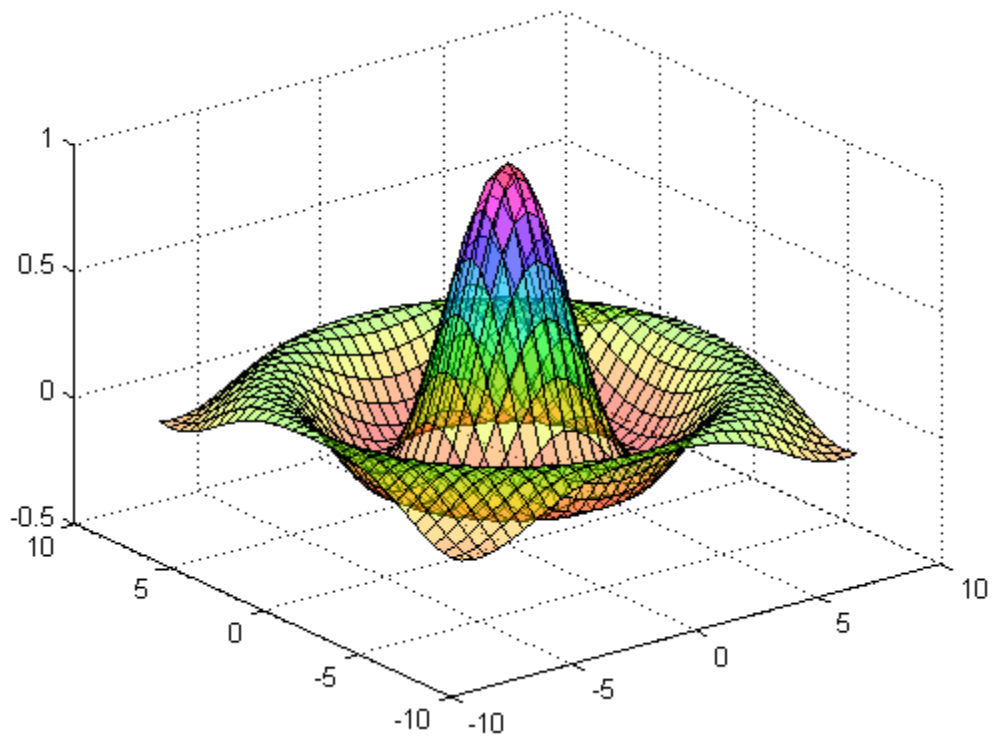
For More Information See “Creating 3-D Graphs” in the MATLAB 3-D Visualization documentation for more information on surface plots.

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.

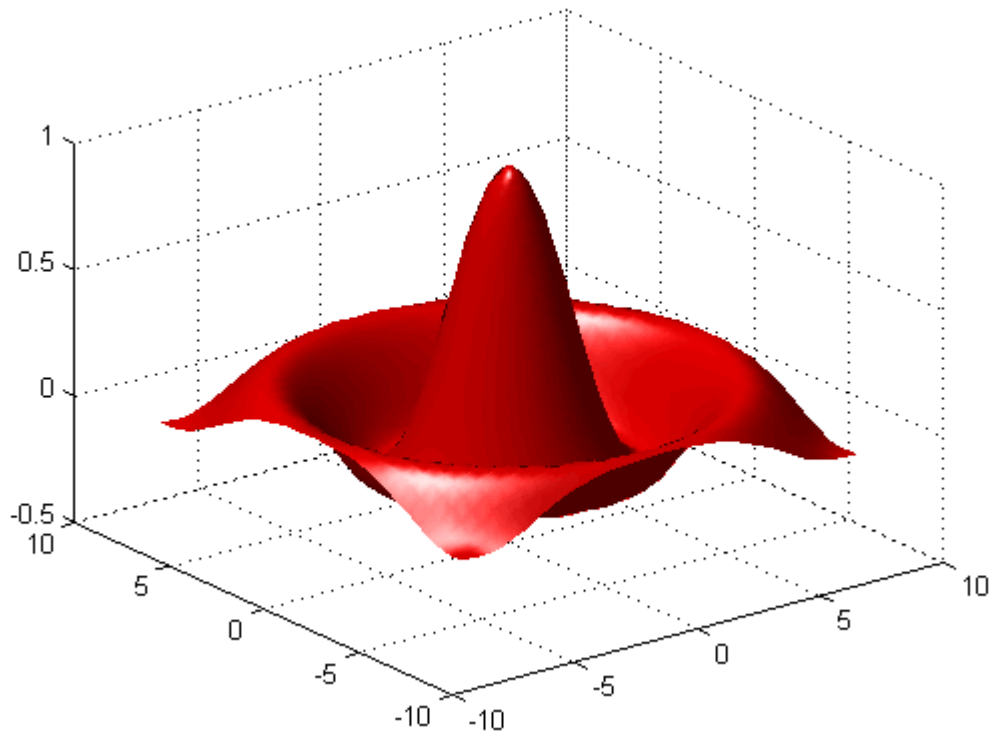
Illuminating Surface Plots with Lights

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):

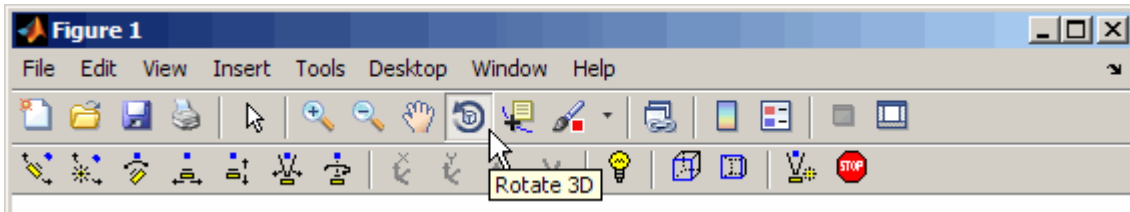
```
surf(X,Y,Z,'FaceColor','red','EdgeColor','none')  
camlight left; lighting phong
```



Manipulating the Surface

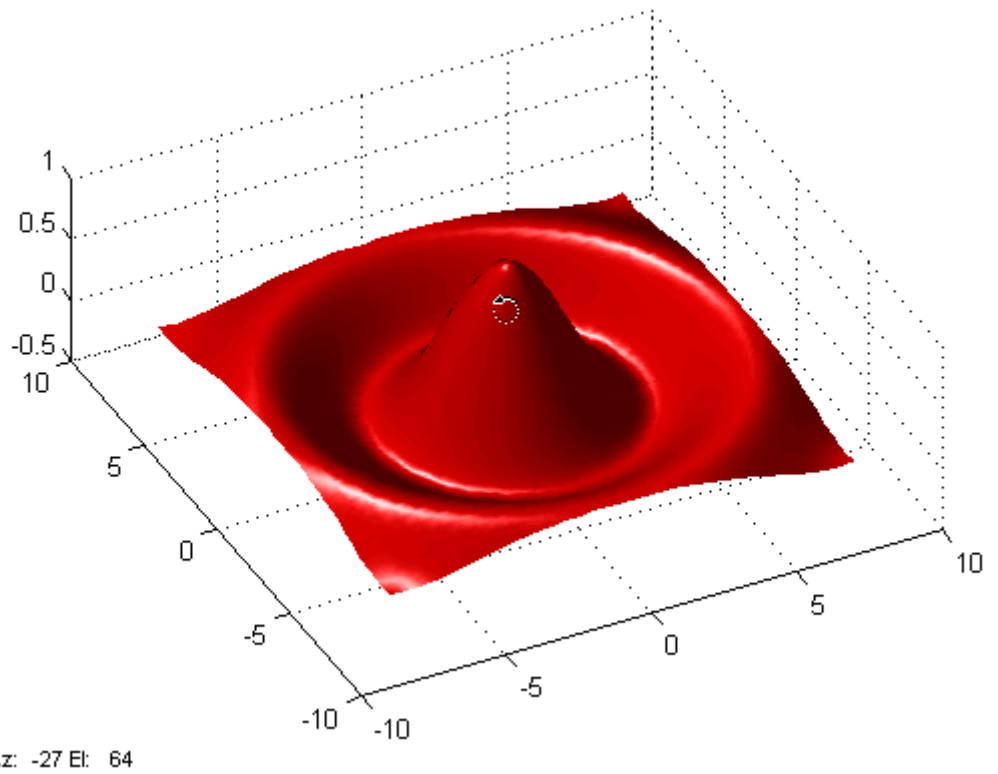
The figure toolbar and the camera toolbar provide ways to explore 3-D graphics interactively. Display the camera toolbar by selecting **Camera Toolbar** from the figure **View** menu.

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**. You can see the tool's cursor icon on the surface where as you drag to rotate the view. As you drag, the viewing altitude and elevation read out in the lower left corner of the axes, as the picture shows.



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” on page 3-73

“Reading and Writing Images” on page 3-74

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 the MATLAB software 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) 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 MATLAB image processing capabilities.

Printing Graphics

In this section...

“Overview of Printing” on page 3-75

“Printing from the File Menu” on page 3-75

“Exporting the Figure to a Graphics File” on page 3-76

“Using the Print Command” on page 3-76

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 that MATLAB supports. 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 3-39 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 3-44 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 3-48 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 magicssquare.tiff
```

If you type `print` on the command line

```
print
```

the current figure prints 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.

Understanding Handle Graphics Objects

In this section...

“Using the Handle” on page 3-78

“Graphics Objects” on page 3-79

“Setting Object Properties” on page 3-81

“Specifying the Axes or Figure” on page 3-84

“Finding the Handles of Existing Objects” on page 3-85

Handle Graphics objects are MATLAB objects that 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 MATLAB plotting function, it creates the graph using various graphics objects, such as a figure window, axes, lines, text, and so on. You can query the value of each property and set the values of most of them.

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')
```

You obtain the answer:

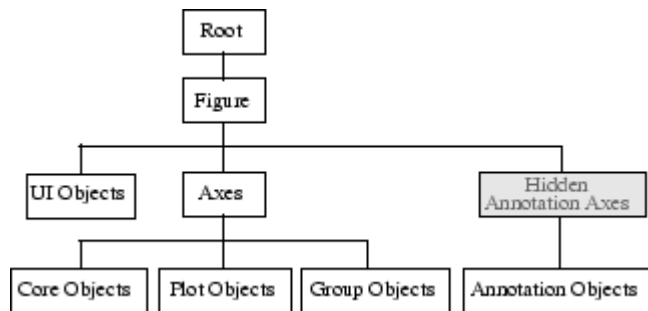
```
ans =
    0.5000
```

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 MATLAB graph, a hierarchy of graphics objects results. 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
<code>allchild</code>	Find all children of specified objects.
<code>ancestor</code>	Find ancestor of graphics object.
<code>copyobj</code>	Copy graphics object.

Function	Purpose
<code>delete</code>	Delete an object.
<code>findall</code>	Find all graphics objects (including hidden handles).
<code>findobj</code>	Find the handles of objects having specified property values.
<code>gca</code>	Return the handle of the current axes.
<code>gcf</code>	Return the handle of the current figure.
<code>gco</code>	Return the handle of the current object.
<code>get</code>	Query the values of an object's properties.
<code>ishandle</code>	True if the value is a valid object handle.
<code>set</code>	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 3-20 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'};
```

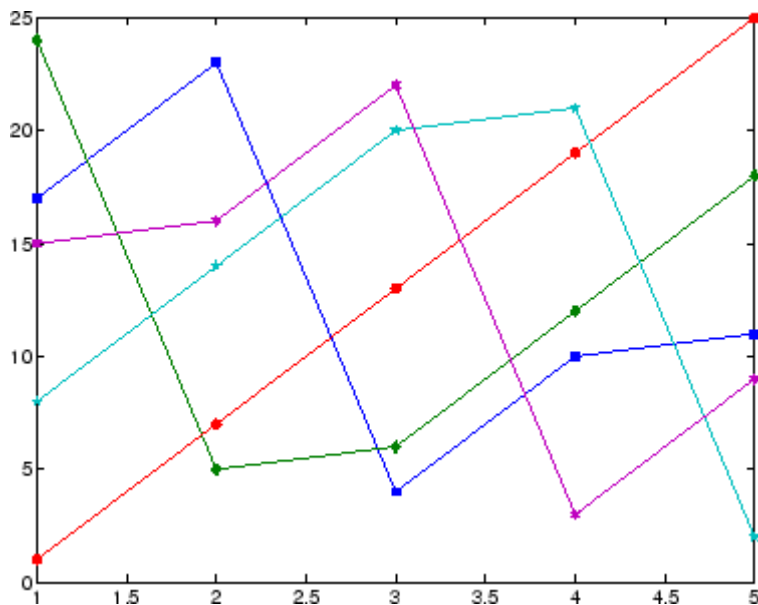
The `prop_values` cell array contains 10 values: five values for the `Marker` property and five values for the `MarkerFaceColor` property. Notice that `prop_values` is a two-dimensional cell array. The first dimension indicates which handle in `h` the values apply to and the second dimension indicates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

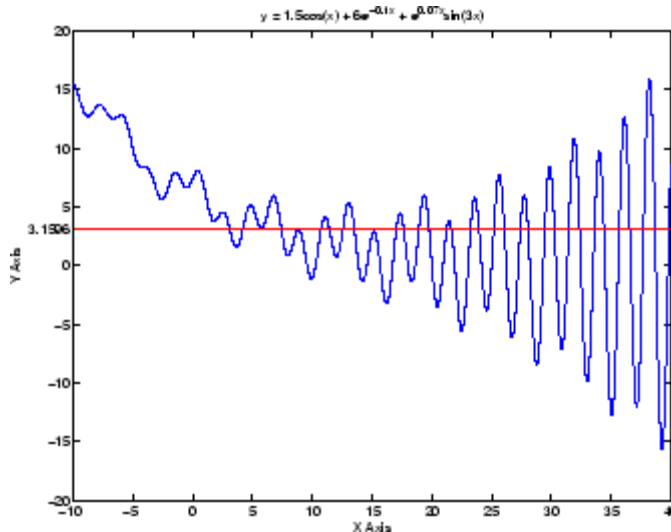
The following example shows a simple M-file that plots a function and the mean of the function over the specified range:

```
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 $\pi/2$ 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.

Programming

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

- “Flow Control” on page 4-2
- “Other Data Structures” on page 4-9
- “Scripts and Functions” on page 4-20
- “Object-Oriented Programming” on page 4-33

Flow Control

In this section...

“Conditional Control – if, else, switch” on page 4-2

“Loop Control – for, while, continue, break” on page 4-5

“Error Control – try, catch” on page 4-7

“Program Termination – return” on page 4-8

Conditional Control – if, else, switch

This section covers those MATLAB product 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, the 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
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) | strcmp(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” on page 4-9
“Cell Arrays” on page 4-11
“Characters and Text” on page 4-13
“Structures” on page 4-16

Multidimensional Arrays

Multidimensional arrays in the MATLAB environment 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 k th matrix, or the t_k th 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

```

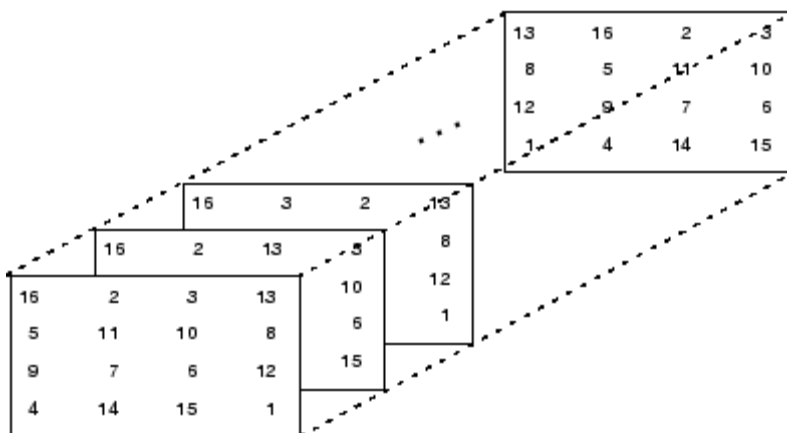
stores the sequence of 24 magic squares in a three-dimensional array, M. The size of M is

```

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.

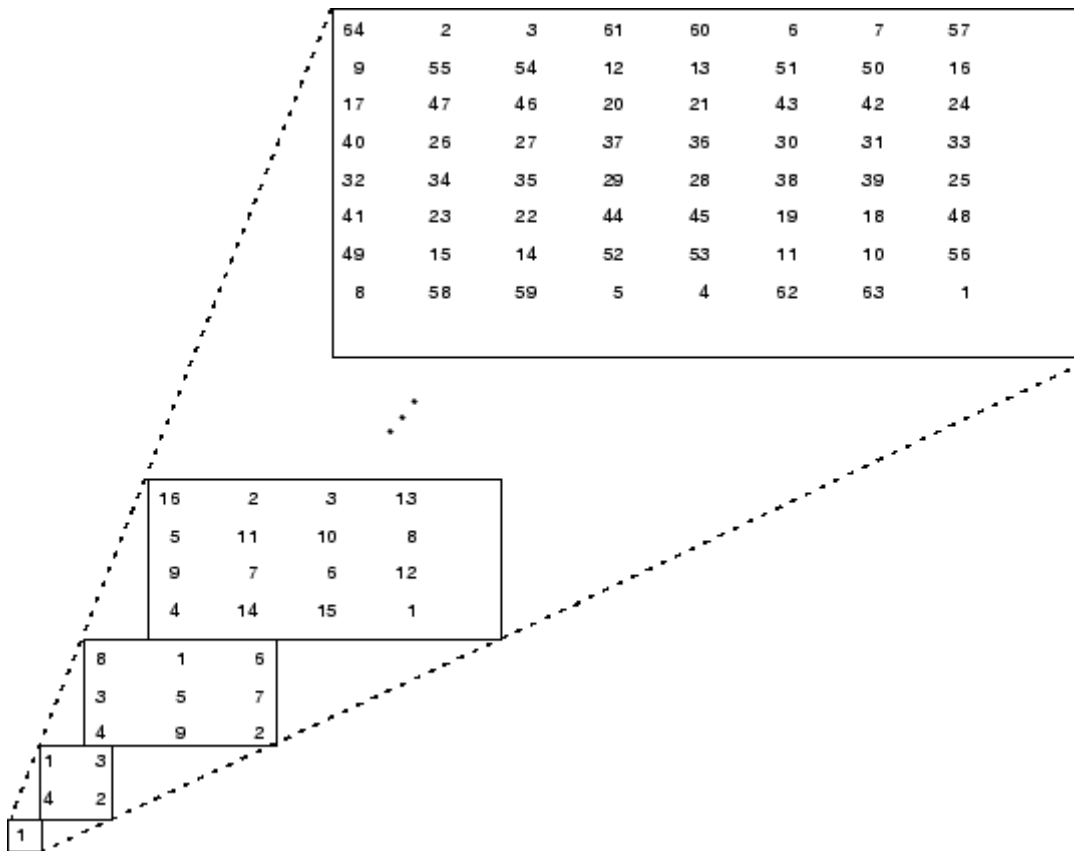
Here are two important points to remember. First, to retrieve the contents of one of the cells, use subscripts in curly braces. For example, `C{1}` retrieves the magic square and `C{3}` is 16!. Second, cell arrays contain *copies* of other arrays, not *pointers* to those arrays. If you subsequently change 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 nonprintable 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 and then click **Fonts** 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 the MATLAB environment, 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” on page 4-20

“Scripts” on page 4-21

“Functions” on page 4-22

“Types of Functions” on page 4-24

“Global Variables” on page 4-26

“Passing String Arguments to Functions” on page 4-27

“The eval Function” on page 4-28

“Function Handles” on page 4-28

“Function Functions” on page 4-29

“Vectorization” on page 4-31

“Preallocation” on page 4-32

Overview

The MATLAB product provides 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`.

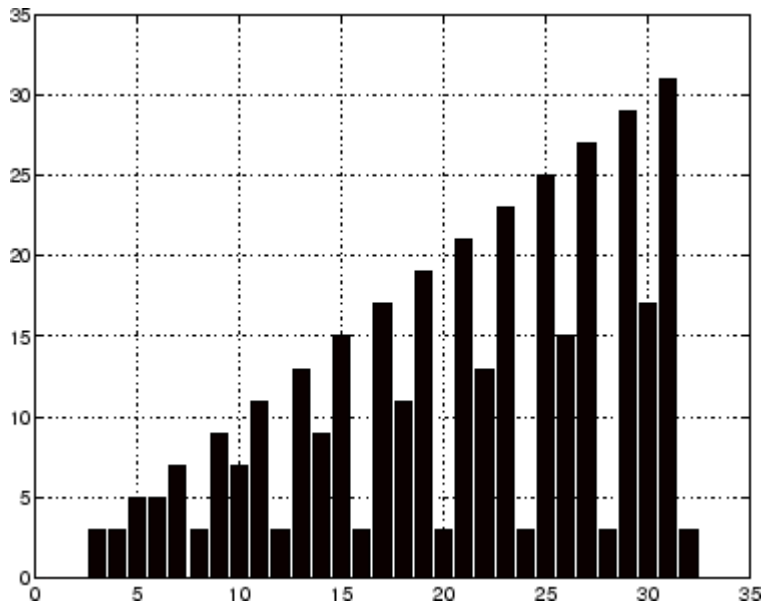
For example, create a file called `magicrank.m` that contains these MATLAB commands:

```
% 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 the 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 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:

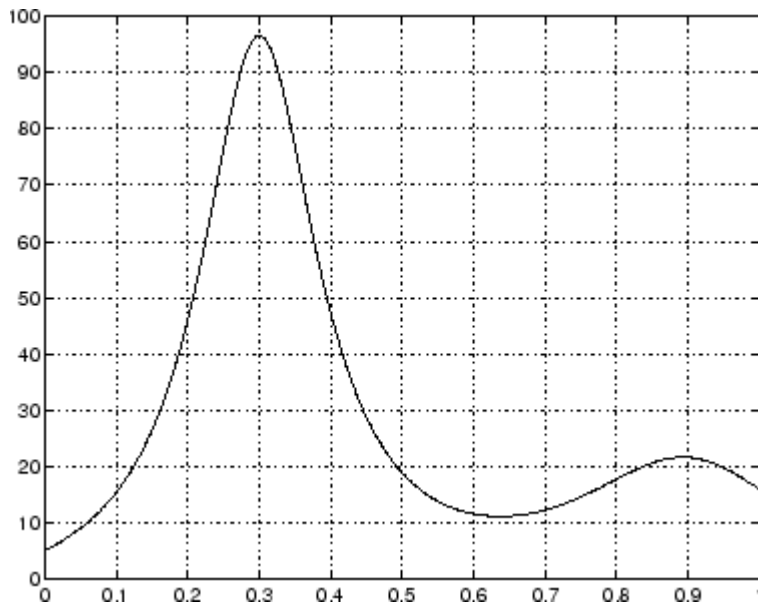
```
function y = humps(x)
    y = 1./((x-.3).^2 + .01) + 1./((x-.9).^2 + .04) - 6;
```

Evaluate this function at a set of points in the interval $0 \leq x \leq 1$ 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 = quad1(@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 “Performance” in the MATLAB Programming Fundamentals 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.

Object-Oriented Programming

In this section...
“MATLAB Classes and Objects” on page 4-33
“Learn About Defining MATLAB Classes” on page 4-33

MATLAB Classes and Objects

MATLAB object-oriented programming capabilities enable you to develop and maintain large application and define complex data structures that integrate seamlessly into the MATLAB environment. Some of the features include:

- Support for value and handle (reference) classes
- Definition of events and listeners
- Class introspection
- JIT/Accelerator support for classes to improve performance

Learn About Defining MATLAB Classes

For complete information on using and creating MATLAB classes, see *Object-Oriented Programming*.

Data Analysis

- “Introduction” on page 5-2
- “Preprocessing Data” on page 5-3
- “Summarizing Data” on page 5-10
- “Visualizing Data” on page 5-14
- “Modeling Data” on page 5-27

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 carry out a basic data analysis in the MATLAB environment.

Preprocessing Data

In this section...

“Overview” on page 5-3

“Loading the Data” on page 5-3

“Missing Data” on page 5-3

“Outliers” on page 5-4

“Smoothing and Filtering” on page 5-6

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 5-27.

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

Missing Data

The MATLAB NaN (Not a Number) value is normally used to 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 `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 “Missing Data” in the MATLAB Data Analysis documentation for more information on handling missing data.

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 $n = 1, 2$:

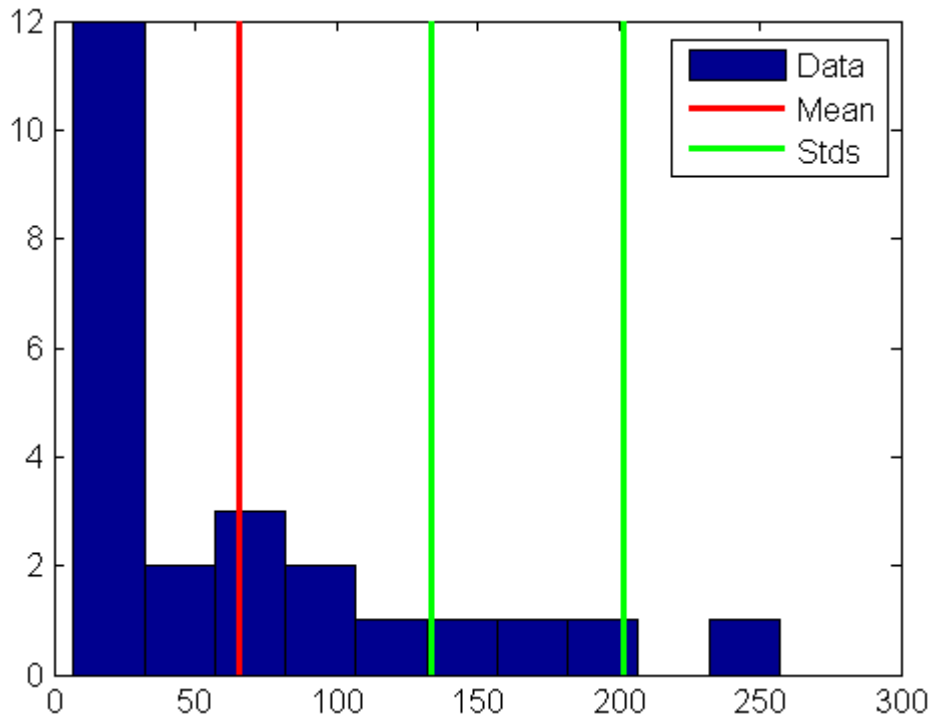
```
bin_counts = hist(c3); % Histogram bin counts
N = max(bin_counts); % Maximum bin count
mu3 = mean(c3); % Data mean
sigma3 = std(c3); % Data standard deviation

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

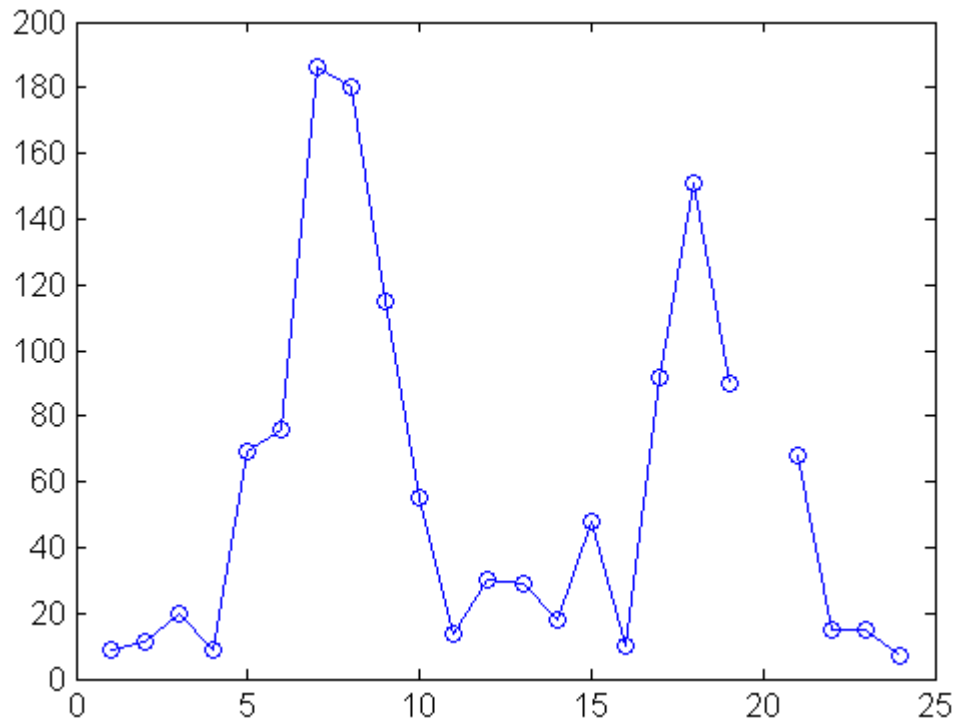
```

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

Smoothing and Filtering

A time-series plot of the data at the third intersection (with the outlier removed in “Outliers” on page 5-4) results in the following plot:

```
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:

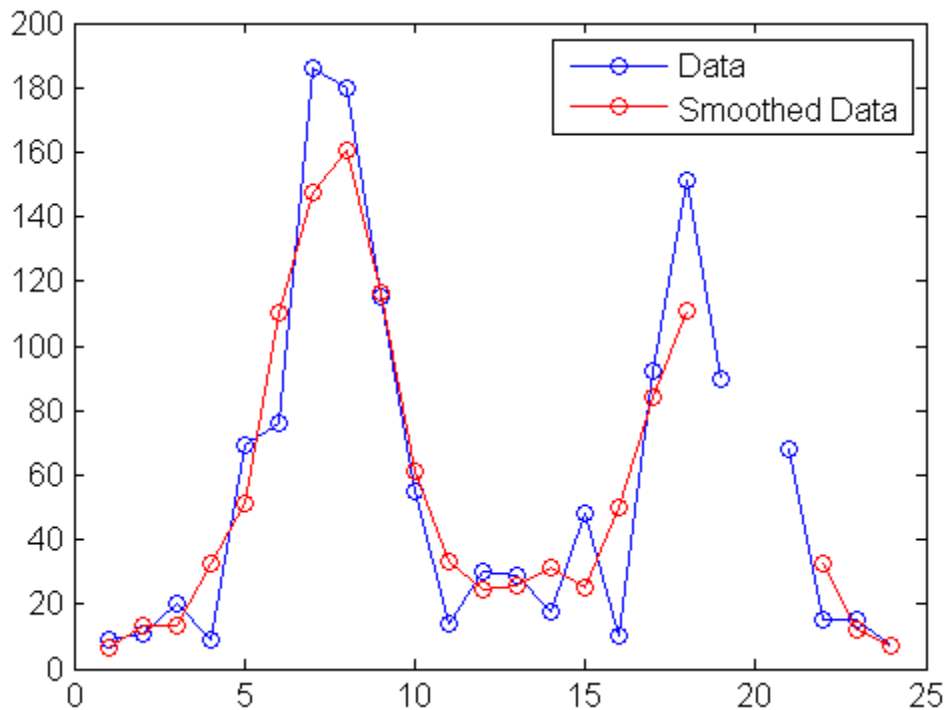
- The relationship between the predictor (time) and the response (traffic volume) is smooth.

- 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')
```

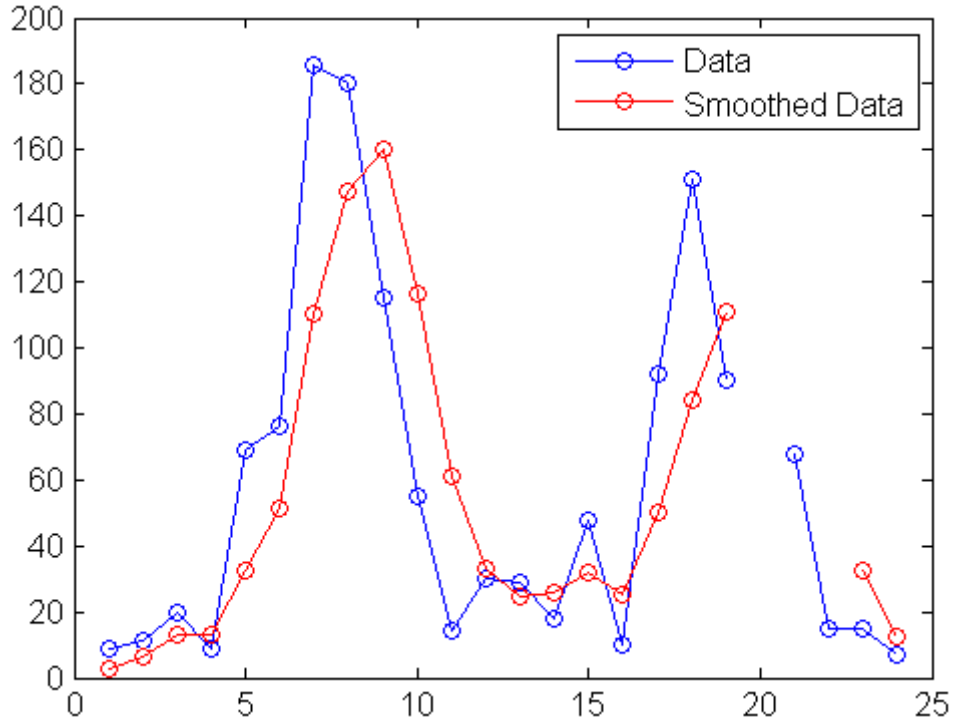


The extent of the smoothing is controlled with the variable `span`. The averaging calculation returns NaN values whenever the smoothing window includes the NaN value in the data, thus increasing the size of the gap in the smoothed data.

The 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 independent*. Accordingly, you can use smoothed data to *identify* a model, but avoid using smoothed data to *fit* a model.

See “Filtering Data” in the MATLAB Data Analysis documentation for more information on smoothing and filtering.

Summarizing Data

In this section...

“Overview” on page 5-10

“Measures of Location” on page 5-10

“Measures of Scale” on page 5-11

“Shape of a Distribution” on page 5-11

Overview

Many MATLAB functions enable you to summarize 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 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 its statistical functions, the MATLAB 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

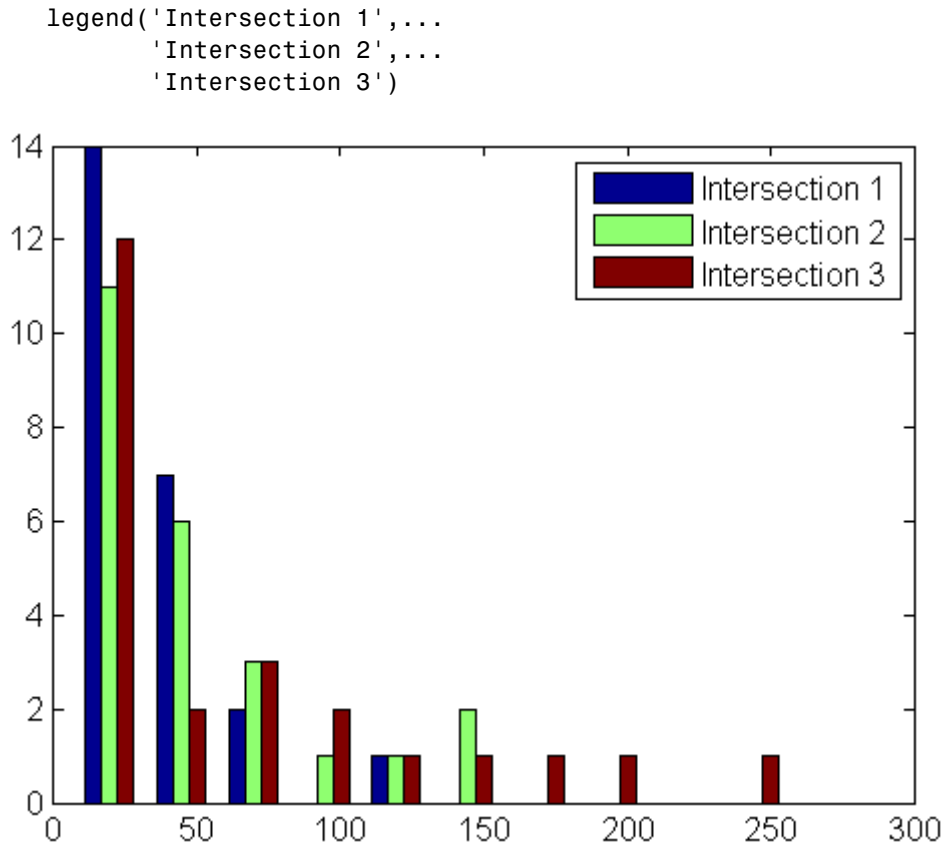
dx3 = var(count)
dx3 =
    1.0e+003 *
    0.6437    1.7144    4.6278
```

Like all of its statistical functions, the MATLAB 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)
```



Parametric models give analytic summaries of distribution shapes. Exponential distributions, with parameter μ given by the data mean, are a good choice for the traffic data:

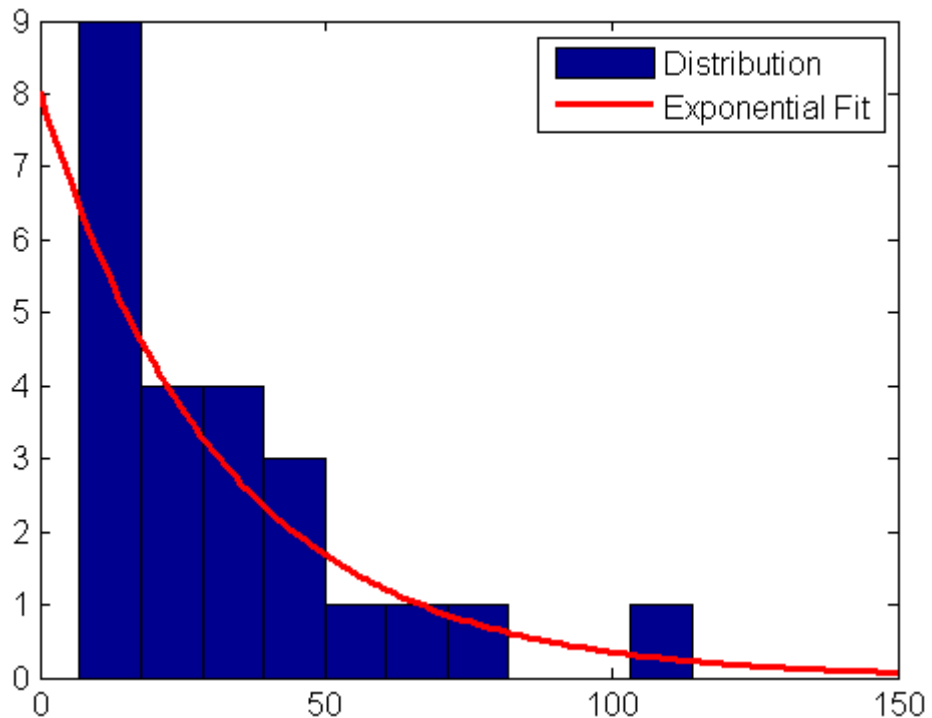
```

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. Statistics Toolbox software provides functions for computing maximum likelihood estimates of distribution parameters.

See “Descriptive Statistics” in the MATLAB Data Analysis documentation for more information on summarizing data samples.

Visualizing Data

In this section...
“Overview” on page 5-14
“2-D Scatter Plots” on page 5-14
“3-D Scatter Plots” on page 5-16
“Scatter Plot Arrays” on page 5-18
“Exploring Data in Graphs” on page 5-19

Overview

You can use many MATLAB graph types for visualizing data patterns and trends. Scatter plots, described in this section, help to visualize relationships among the traffic data at different intersections. Data exploration tools let you query and interact with individual data points on graphs.

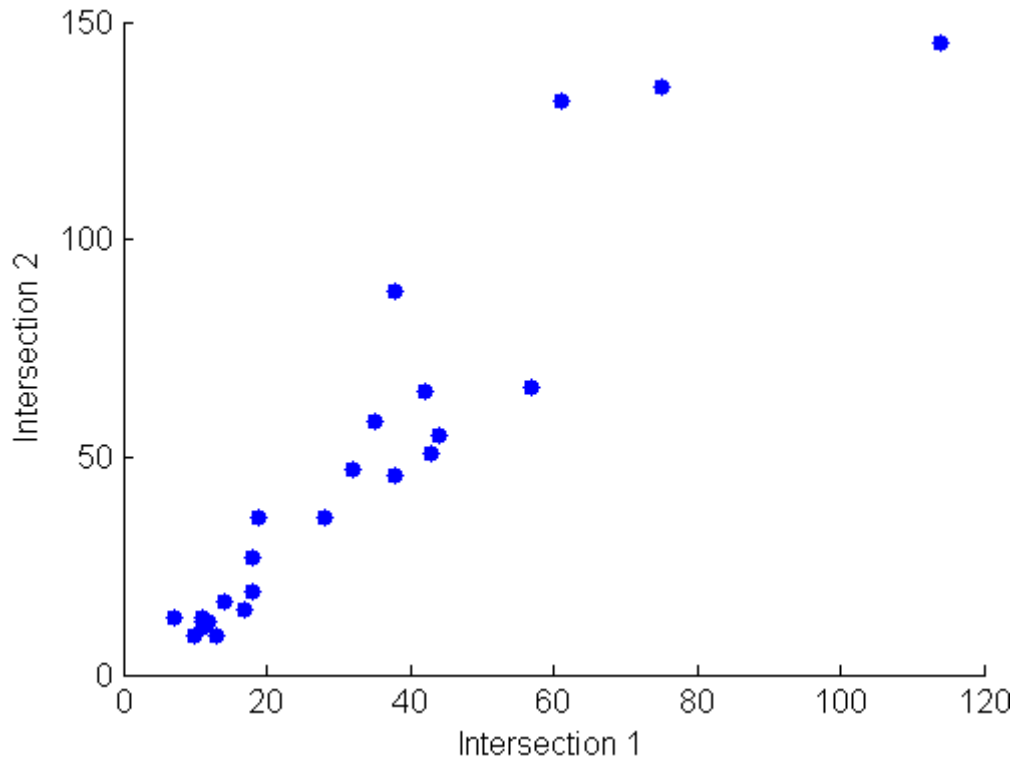
Note This section continues the data analysis from “Summarizing Data” on page 5-10.

2-D Scatter Plots

A 2-D scatter plot, created with the `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 *covariance*, computed by the `cov` function measures the strength of the linear relationship between the two variables (how tightly the data lies along a least-squares line through the scatter):

```
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. You can divide a covariance by the standard

deviations of the variables to normalize values between +1 and -1. The `corrcoef` function computes *correlation coefficients*:

```
R12 = corrcoef([c1 c2])
R12 =
    1.0000    0.9331
    0.9331    1.0000

r12 = R12(1,2) % Correlation coefficient
r12 =
    0.9331

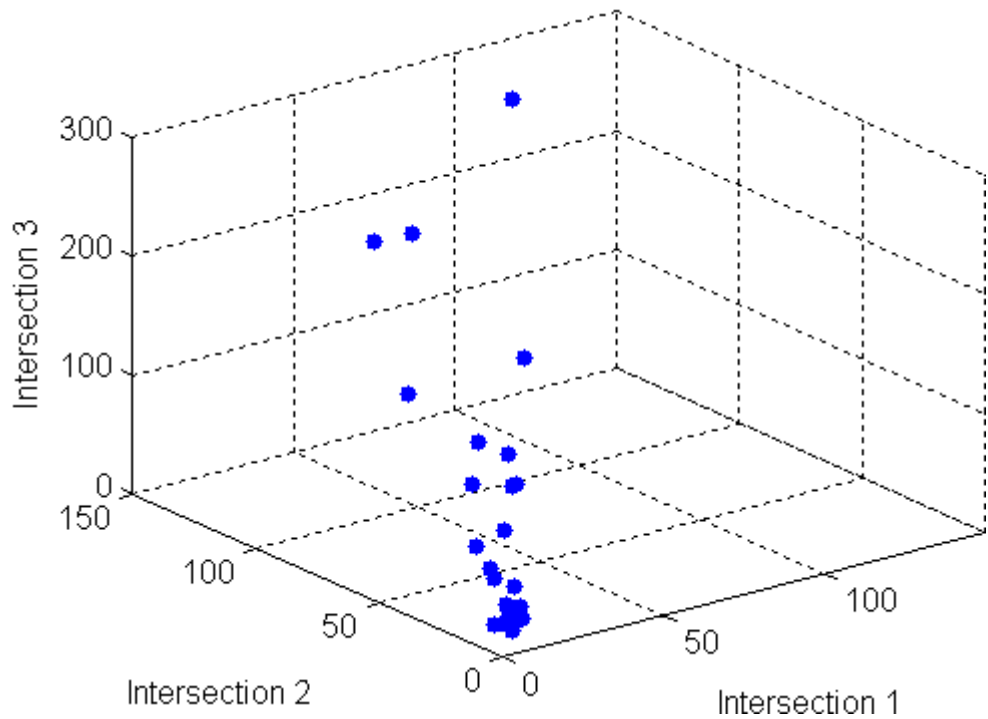
r12sq = r12^2 % Coefficient of determination
r12sq =
    0.8707
```

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. Thus, it is the proportion of variation in the response (in this case, the traffic volume at intersection 2) that is eliminated or statistically explained by a least-squares line through the scatter.

3-D Scatter Plots

A 3-D scatter plot, created with the `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')
```

Measure the strength of the linear relationship among the variables in the 3-D scatter by computing eigenvalues of the covariance matrix with the 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
```

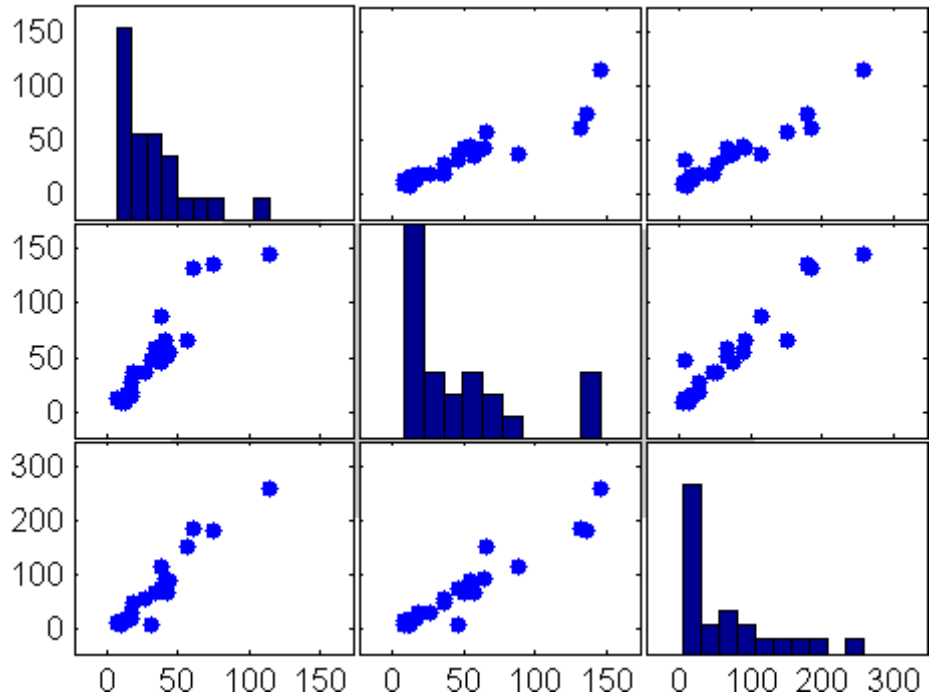
The eigenvalues are the variances along the *principal components* of the data. The variable explained measures the proportion of variation explained by

the first principal component, along the axis of the data. Unlike the coefficient of determination for 2-D scatters, this measure distinguishes predictor and response variables.

Scatter Plot Arrays

Use the `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.

For more information on statistical visualization, see “Plotting Data” and “Interactive Data Exploration” in the MATLAB Data Analysis documentation.

Exploring Data in Graphs


Using your mouse, you can pick observations on almost any MATLAB graph with two tools from the figure toolbar:

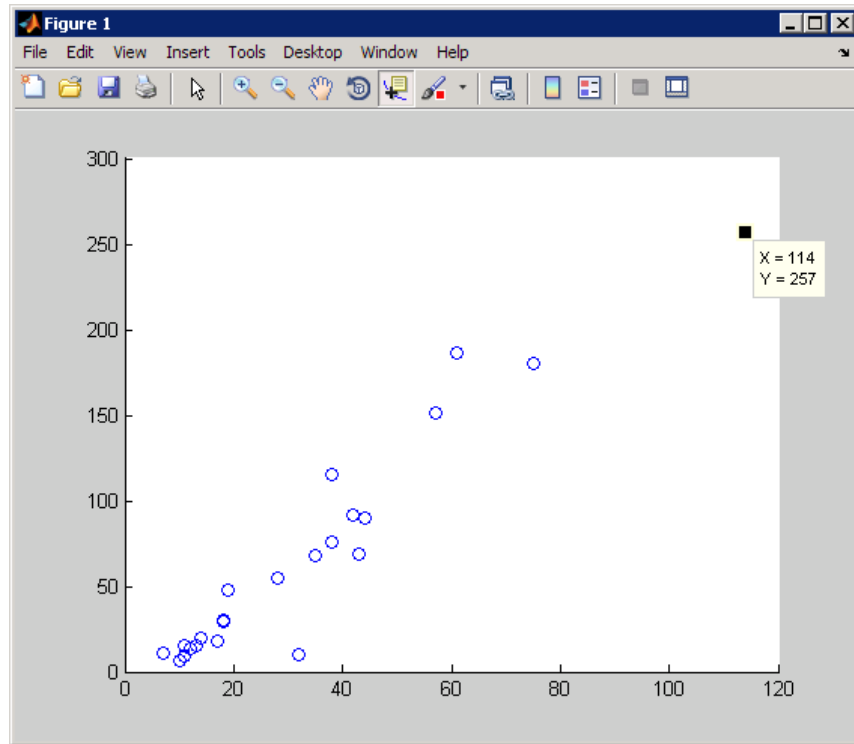
- Data Cursor 
- Data Brushing 

These tools each place you in exploratory modes in which you can select data points on graphs to identify their values and create workspace variables to contain specific observations. When you use data brushing, you can also copy, remove or replace the selected observations.

For example, make a scatter plot of the first and third columns of `count`:


```
scatter(count(:,1),count(:,3))
```

Select the Data Cursor Tool  and click the right-most data point. A `datatip` displaying the point's x and y value is placed there.



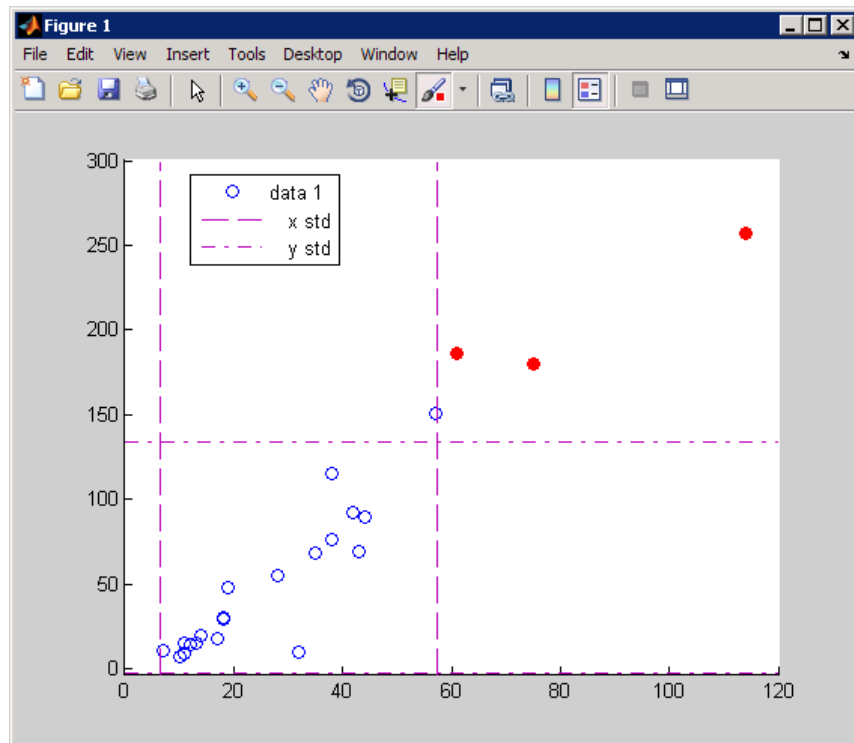
Datatips display x -, y -, and z - (for 3-D plots) coordinates by default. You can drag a datatip from one data point to another to see new values or add additional datatips by right-clicking a datatip and using the context menu. You can also customize the text that datatips display using M-code. For more information, see the `datacursormode` function and “Interacting with Graphed Data” in the MATLAB Data Analysis documentation.

Data brushing is a related feature that lets you highlight one or more observations on a graph by clicking or dragging. To enter data brushing mode,

click the left side of the Data Brushing tool  on the figure toolbar. Clicking the arrow on the right side of the tool icon drops down a color palette for selecting the color with which to brush observations. This figure shows the same scatter plot as the previous figure, but with all observations beyond

one standard deviation of the mean (as identified using the **Tools > Data Statistics** GUI) brushed in red.

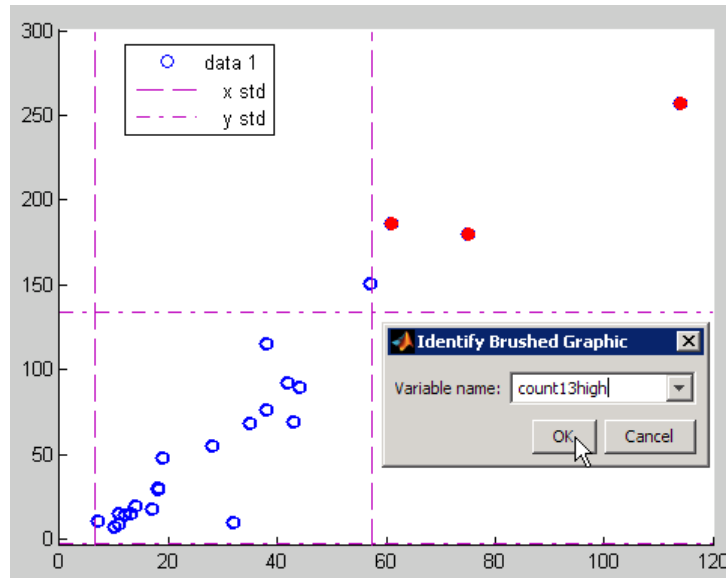
```
scatter(count(:,1),count(:,3))
```



After you brush data observations, you can perform the following operations on them:

- Delete them.
- Replace them with constant values.
- Replace them with NaN values.
- Drag or copy, and paste them to the Command Window.
- Save them as workspace variables.

For example, use the Data Brush context menu or the **Tools > Brushing > Create new variable** option to create a new variable called `count13high`.



A new variable in the workspace results:

```
count13high

count13high =
    61    186
    75    180
   114    257
```

For more information, see the MATLAB `brush` function and “Marking Up Graphs with Data Brushing” in the MATLAB Data Analysis documentation.

Linked plots is a feature closely related to data brushing. A plot is said to be linked when it has a live connection to the workspace data it depicts. The copies of variables stored in a plot object’s `XData`, `YData`, (and, where appropriate, `ZData`), automatically updated whenever the workspace variables to which they are linked change or are deleted. This causes the graphs on which they appear to update automatically.

Linking plots to variables lets you track specific observations through different presentations of them. When you brush data points in linked plots, brushing one graph highlights the same observations in every graph that is linked to the same variables.

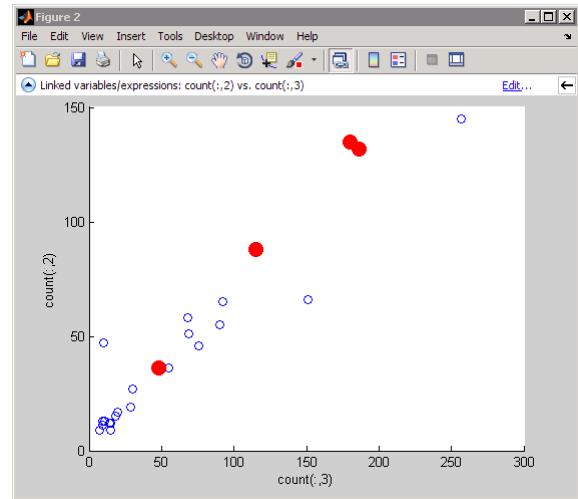
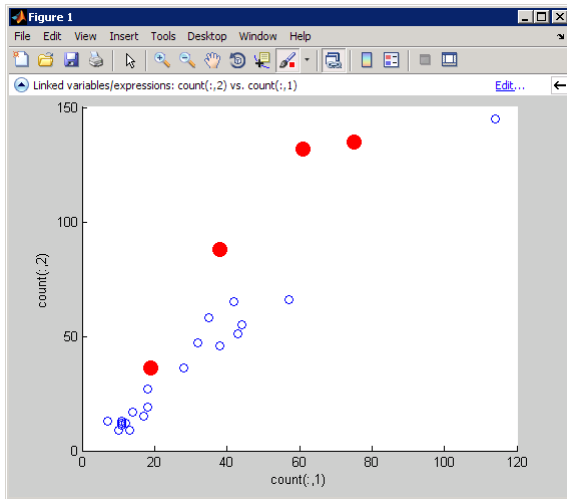
Data linking establishes immediate, two-way communication between figures and workspace variables, in the same way that the Variable Editor communicates with workspace variables. You create links by activating the



Data Linking tool on a figure's toolbar. Activating this tool causes the Linked Plot information bar, displayed in the next figure, to appear at the top of the plot (possibly obscuring its title). You can dismiss the bar (shown in the following figure) without unlinking the plot; it does not print and is not saved with the figure.

The following two graphs depict scatter plot displays of linked data after brushing some observations on the left graph. The common variable, count carries the brush marks to the right figure. Even though the right graph is not in data brushing mode, it displays brush marks because it is linked to its variables.

```
figure
scatter(count(:,1),count(:,2))
xlabel ('count(:,1)')
ylabel ('count(:,2)')
figure
scatter(count(:,3),count(:,2))
xlabel ('count(:,3)')
ylabel ('count(:,2)')
```



The right plot shows that the brushed observations are more linearly related than in the left plot.

Brushed data observations appear highlighted in the brushing color when you display those variables in the Variable Editor, as you can see here:

```
openvar count
```


	1	2	3	4
1	11	11	9	
2	7	13	11	
3	14	17	20	
4	11	13	9	
5	43	51	69	
6	38	46	76	
7	61	132	186	
8	75	135	180	
9	38	88	115	
10	28	36	55	
11	12	12	14	
12	18	27	30	
13	18	19	29	
14	17	15	18	
15	19	36	48	
16	32	47	10	
17	42	65	92	
18	57	66	151	
19	44	55	90	
20	114	145	257	
21	35	58	68	
22	11	12	15	
23	13	9	15	
24	10	9	7	

In the Variable Editor, you can alter any values of linked plot data, and the graphs will reflect your edits. To see existing brush marks or to brush data observation from the Variable Editor, click its Brushing Tool button. When you brush a variable that is a column in a matrix, the other variables (columns) in that row are also brushed. Separate variables (other vectors or matrices) can also be brushed in this way if they are linked to graphs.

For more information, see the `linkdata` function, “Making Graphs Responsive with Data Linking” in the MATLAB Data Analysis documentation, and “Viewing and Editing Values in the Current Workspace” in the MATLAB Desktop Environment documentation.

Modeling Data

In this section...
“Overview” on page 5-27
“Polynomial Regression” on page 5-27
“General Linear Regression” on page 5-28

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.

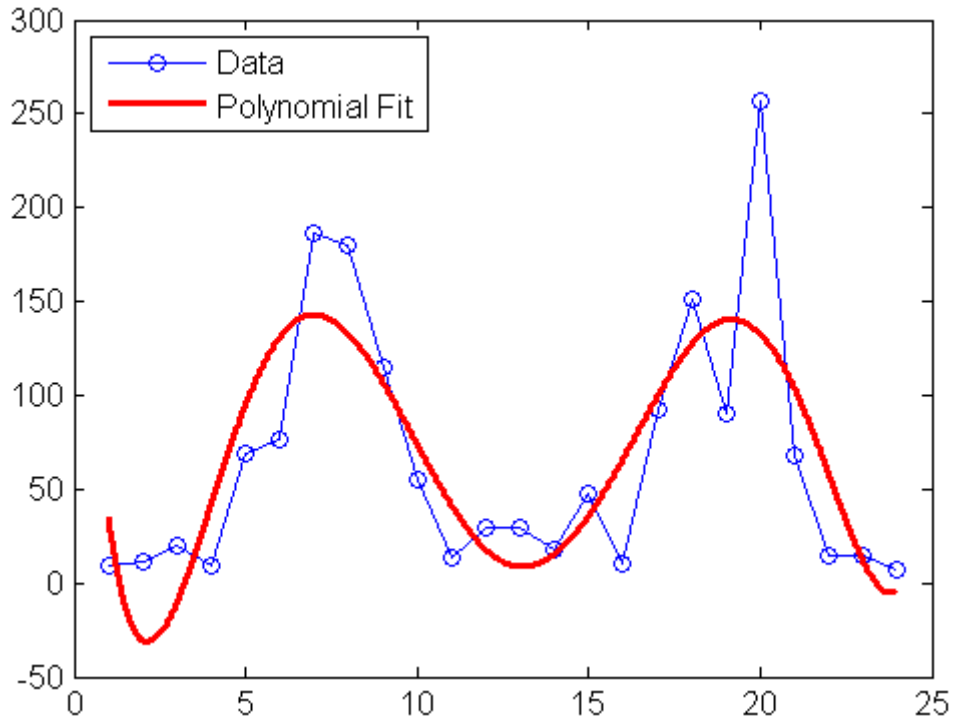
Polynomial Regression

Use the `polyfit` function to estimate coefficients of polynomial models, then use the `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:

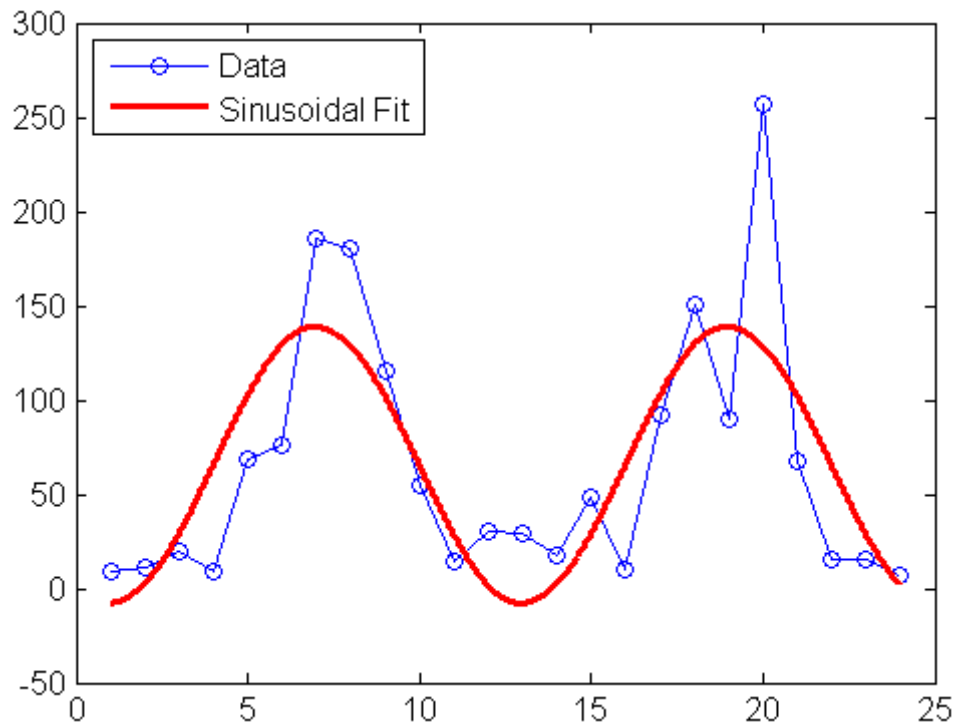
```
c3 = count(:,3); % Data at intersection 3
tdata = (1:24)';
```

```

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 `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

```

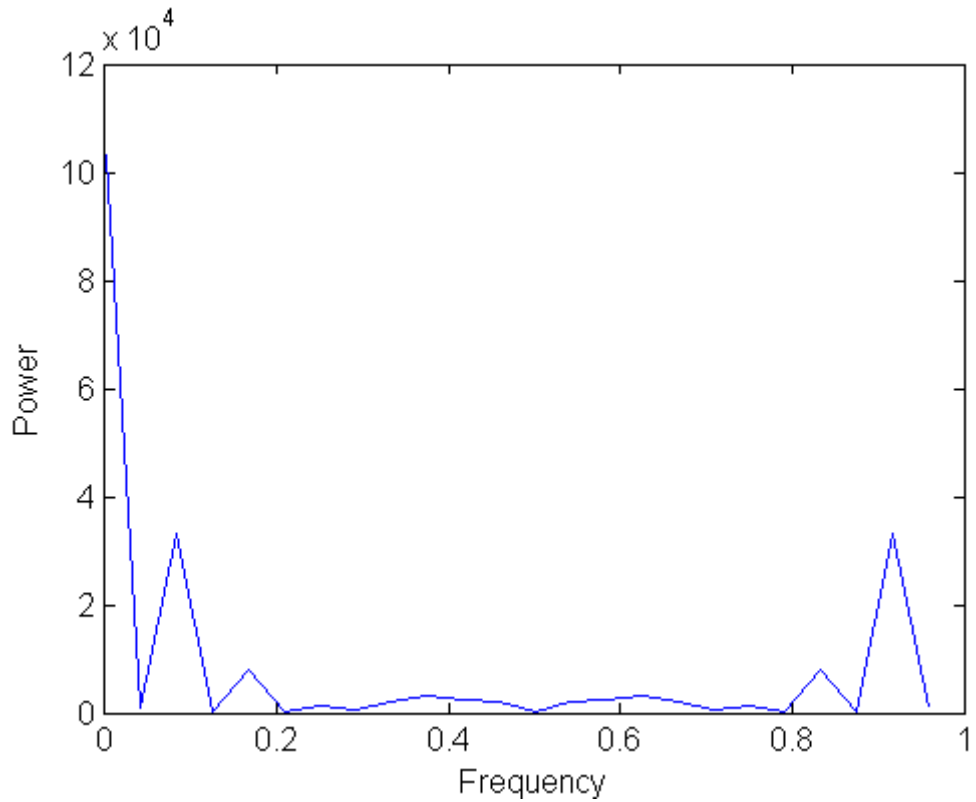
```
73.2819
stdx =
    8.9185
    12.6127
mse =
    1.9090e+003
```

Check the assumption of a 12-hour period in the data with a *periodogram*, computed using the `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 “Regression Analysis” and “Fourier Transforms” in the MATLAB Data Analysis and Mathematics documentation for more information on data modeling.

Creating Graphical User Interfaces

- “What Is GUIDE?” on page 6-2
- “Laying Out a GUI” on page 6-3
- “Programming a GUI” on page 6-7

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 perform the following tasks:

- 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

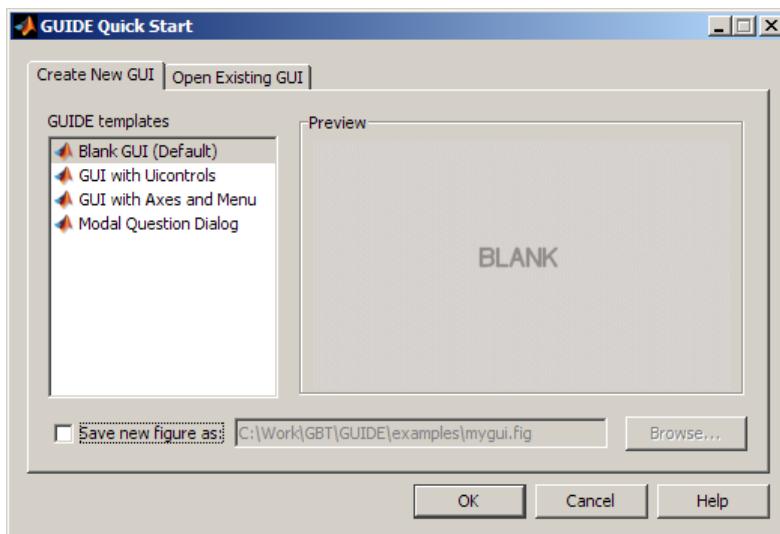
In this section...

“Starting GUIDE” on page 6-3

“The Layout Editor” on page 6-4

Starting GUIDE

Start GUIDE by typing `guide` at the MATLAB command prompt. This command displays the GUIDE Quick Start dialog box, as shown in the following figure.

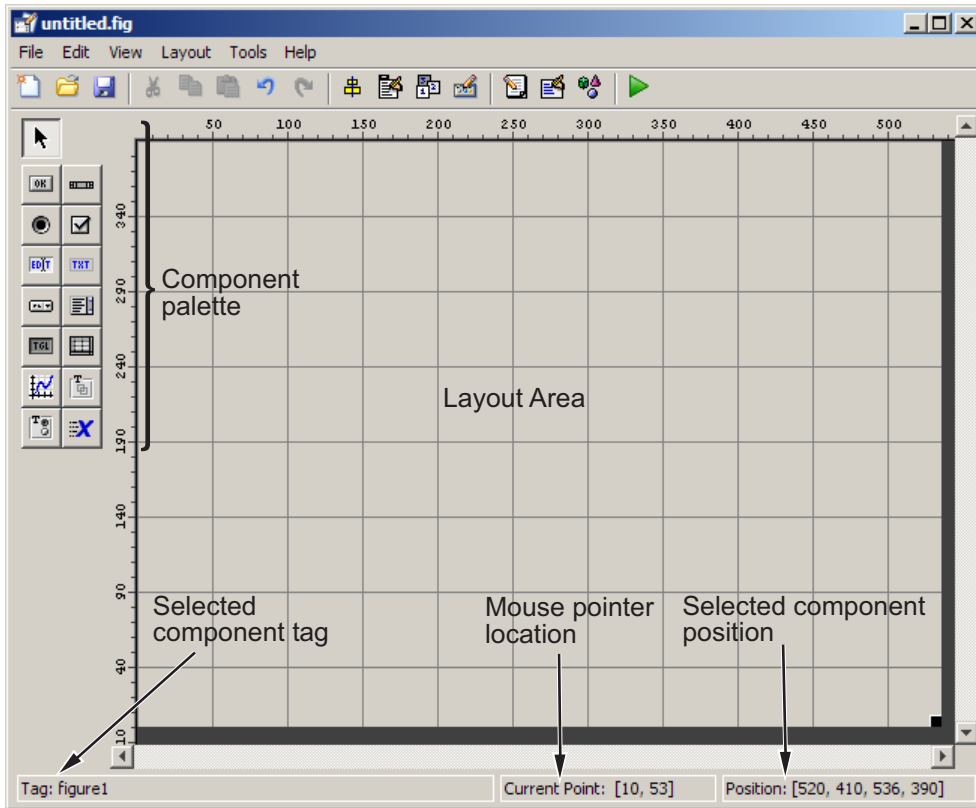


From the GUIDE Quick Start dialog box, you can perform the following tasks:

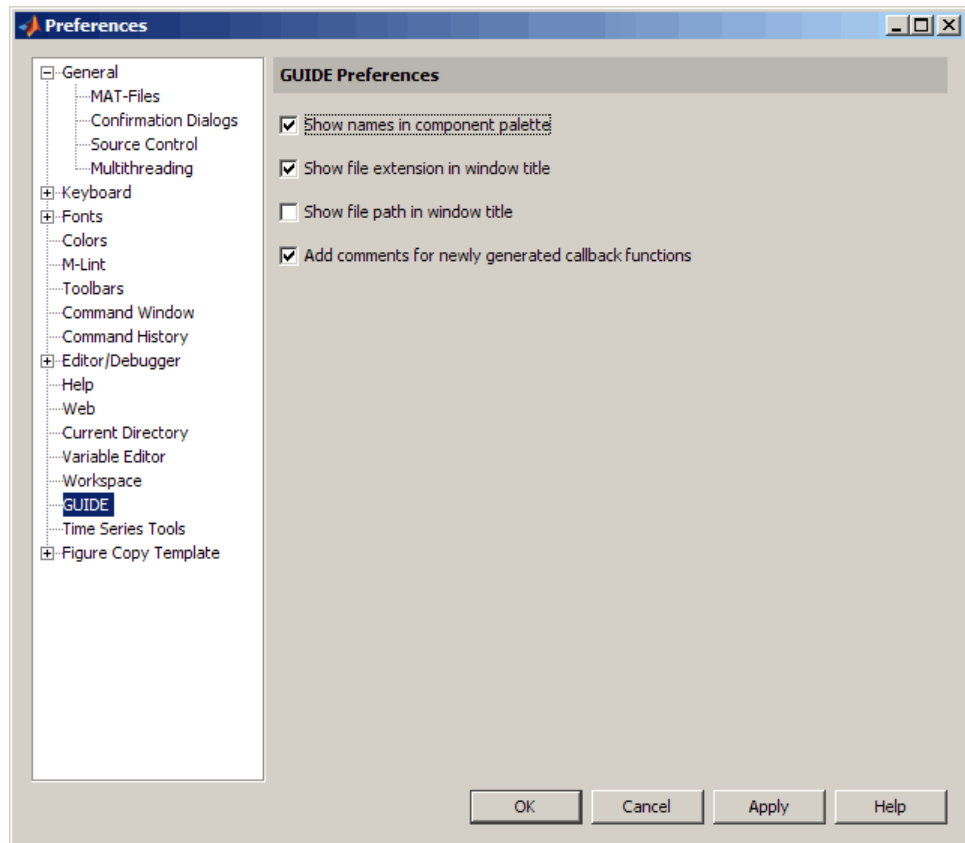
- Create a new GUI from one of the GUIDE templates—prebuilt GUIs that you can modify for your own purposes.
- Open an existing GUI.

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.

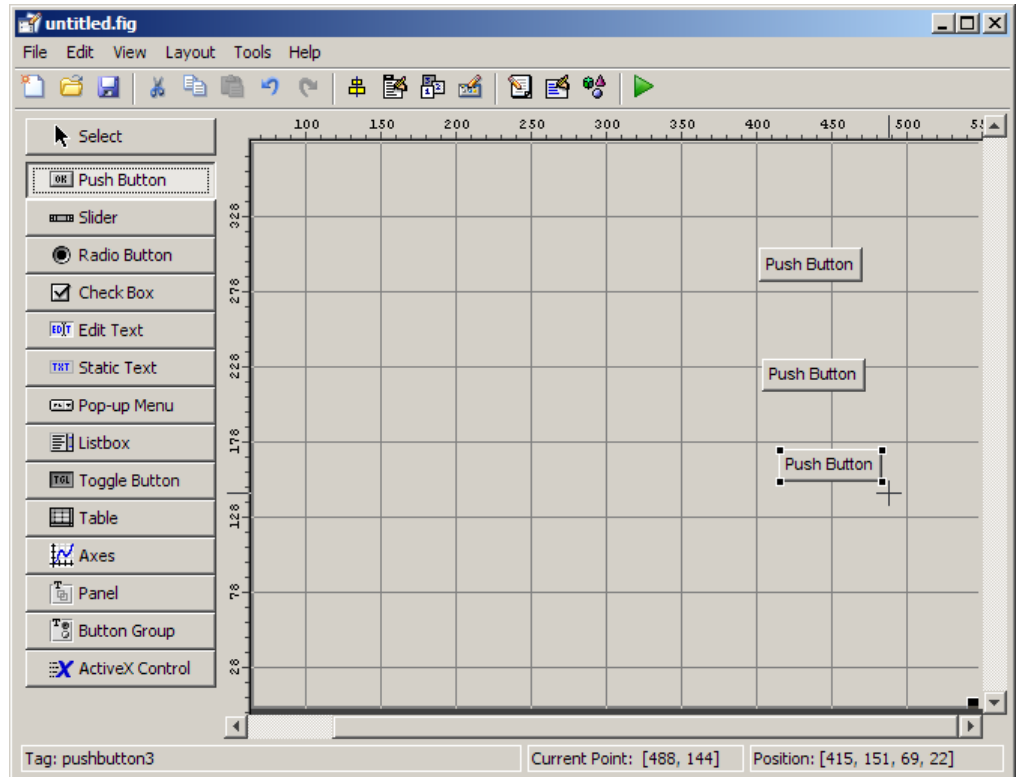


You can expand the tool icons in the Layout Editor to include component names. Open **File > Preferences** from GUIDE or the MATLAB Desktop and select **Show Names in Component Palette**, as shown in the following illustration.



See “GUIDE Preferences” in the MATLAB Creating Graphical Interfaces documentation for details.

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 three push buttons into the layout area, it might look like this.



The illustration also shows how the GUIDE tool palette looks when you set a preference to show component names, as described above.

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.

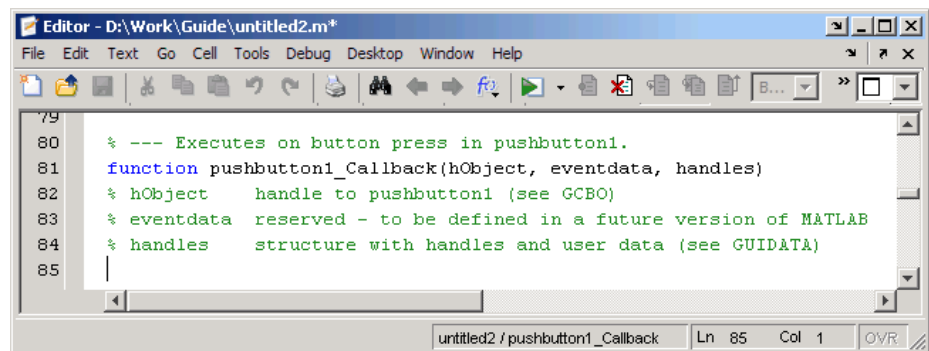
To get started using the Layout Editor and setting property values, see “Creating a Simple GUI with GUIDE” in the MATLAB Creating Graphical User Interfaces documentation. “Examples of GUIDE GUIs” in the same documentation illustrates the variety of GUIs that you can create with GUIDE.

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.



```
79
80  % --- Executes on button press in pushbutton1.
81  function pushbutton1_Callback(hObject, eventdata, handles)
82  % hObject    handle to pushbutton1 (see GCBO)
83  % eventdata  reserved - to be defined in a future version of MATLAB
84  % handles    structure with handles and user data (see GUIDATA)
85  |
```

To learn more about programming a GUI, see “Creating a Simple GUI with GUIDE” in the MATLAB Creating Graphical User Interfaces documentation.

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” on page 7-2
- “Command Window and Command History” on page 7-6
- “Help” on page 7-8
- “Current Directory, Search Path, and File Management” on page 7-13
- “Workspace Browser and Variable Editor” on page 7-17
- “Editor” on page 7-19
- “Improving and Tuning M-Files” on page 7-25

Desktop Overview

In this section...
“Introduction to the Desktop” on page 7-2
“Arranging the Desktop” on page 7-4
“Start Button” on page 7-4

Introduction to the Desktop

Use desktop tools to manage your work and become more productive using MATLAB software. You can also use MATLAB functions to perform the equivalent of most of the features found in the desktop tools.

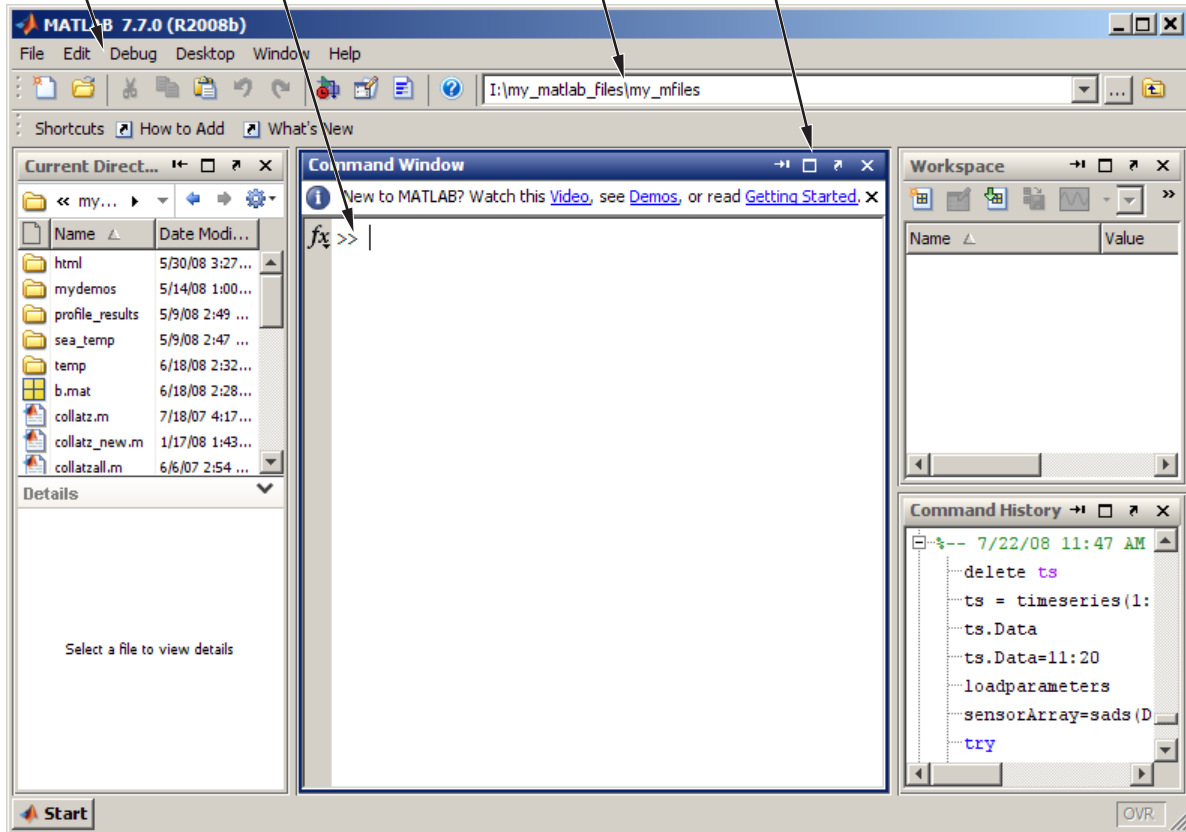
The following illustration shows the default configuration of the MATLAB desktop. You can modify the setup to meet your needs.

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.



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.

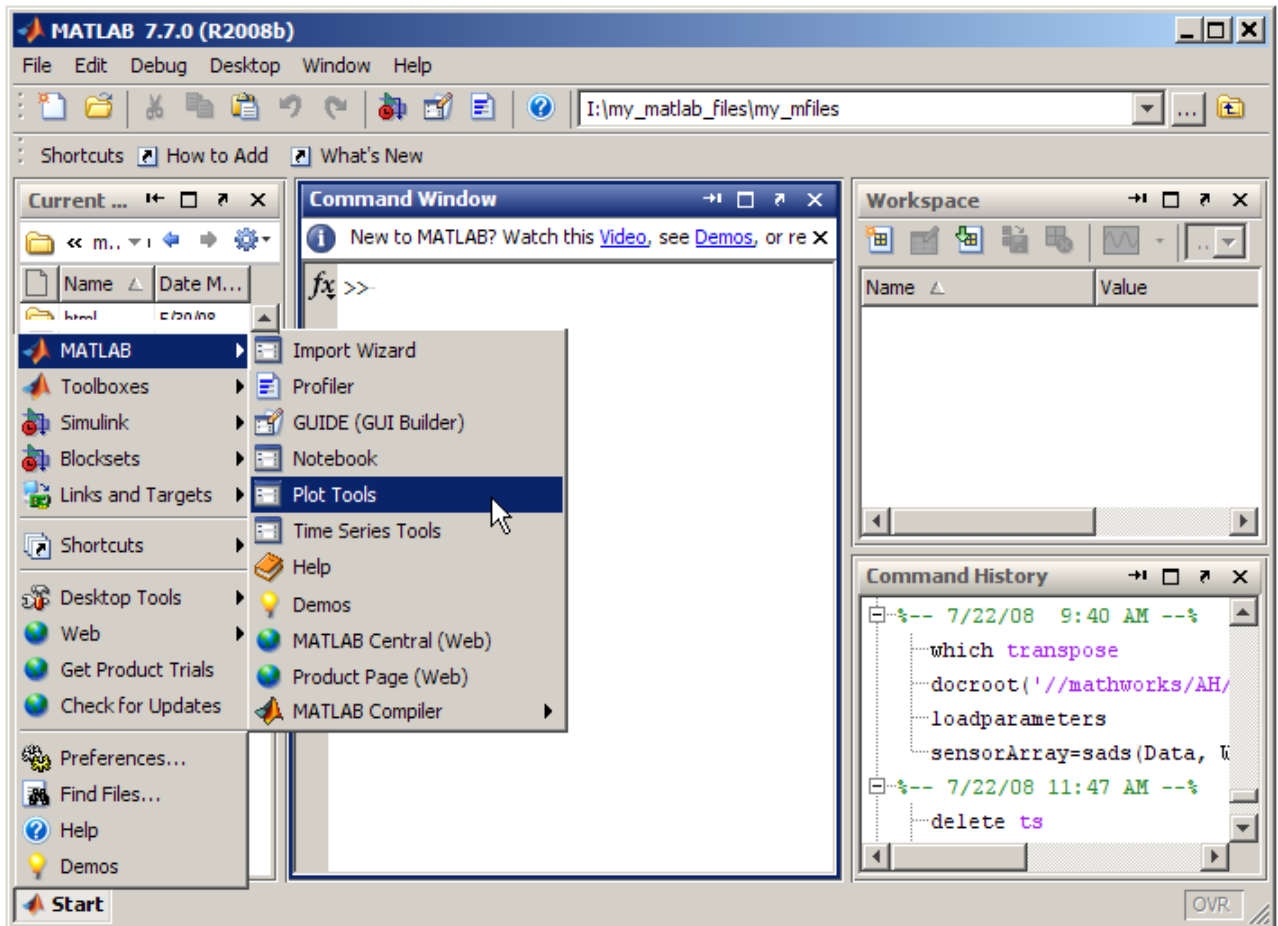
Arranging the Desktop

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. When they do, you access a tool by clicking its title bar.
- Maximize or minimize (temporarily hide) a tool within the desktop via the **Desktop** menu.
- Change fonts, customize the toolbar, and access 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...

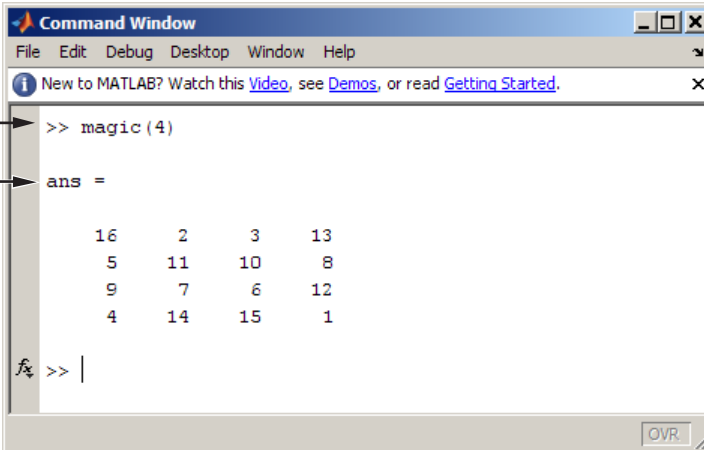
“Command Window” on page 7-6

“Command History” on page 7-7

Command Window

Use the Command Window to enter variables and to run functions and M-file scripts.

Run functions and enter variables at the MATLAB prompt.




The screenshot shows the MATLAB Command Window interface. The title bar reads "Command Window". The menu bar includes "File", "Edit", "Debug", "Desktop", "Window", and "Help". A status bar at the top contains an information icon and the text "New to MATLAB? Watch this [Video](#), see [Demos](#), or read [Getting Started](#)". The main area displays the command prompt ">> magic(4)" followed by the output "ans =". The output is a 4x4 magic square matrix:

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

Below the matrix, the prompt "fx >>|" is visible. A small "OVR" button is located in the bottom right corner of the window.

MATLAB displays the results.

Press the up arrow key \uparrow 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.

There are other tools available to help you remember functions and their syntax, and to enter statements correctly. For example, to look for functions, use the Function Browser to look for functions—click the  button at the left of the prompt to open the tool. For more information on ways to get help while you work in the Command Window, see “Assistance While Entering Statements”.

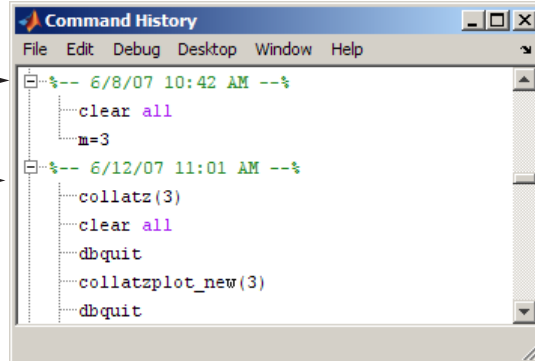
For More Information See “Running Functions — Command Window and History” in the MATLAB Desktop Tools and Development Environment documentation for complete details.

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.

Timestamp marks the start of each session. →

Select one or more entries and right-click to copy, evaluate, or create an M-file from the selection. →



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” on page 7-8
“Other Forms of Help” on page 7-11
“Typographical Conventions” on page 7-11

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 environment.

To open the Help browser, click the Help button  in the desktop toolbar.

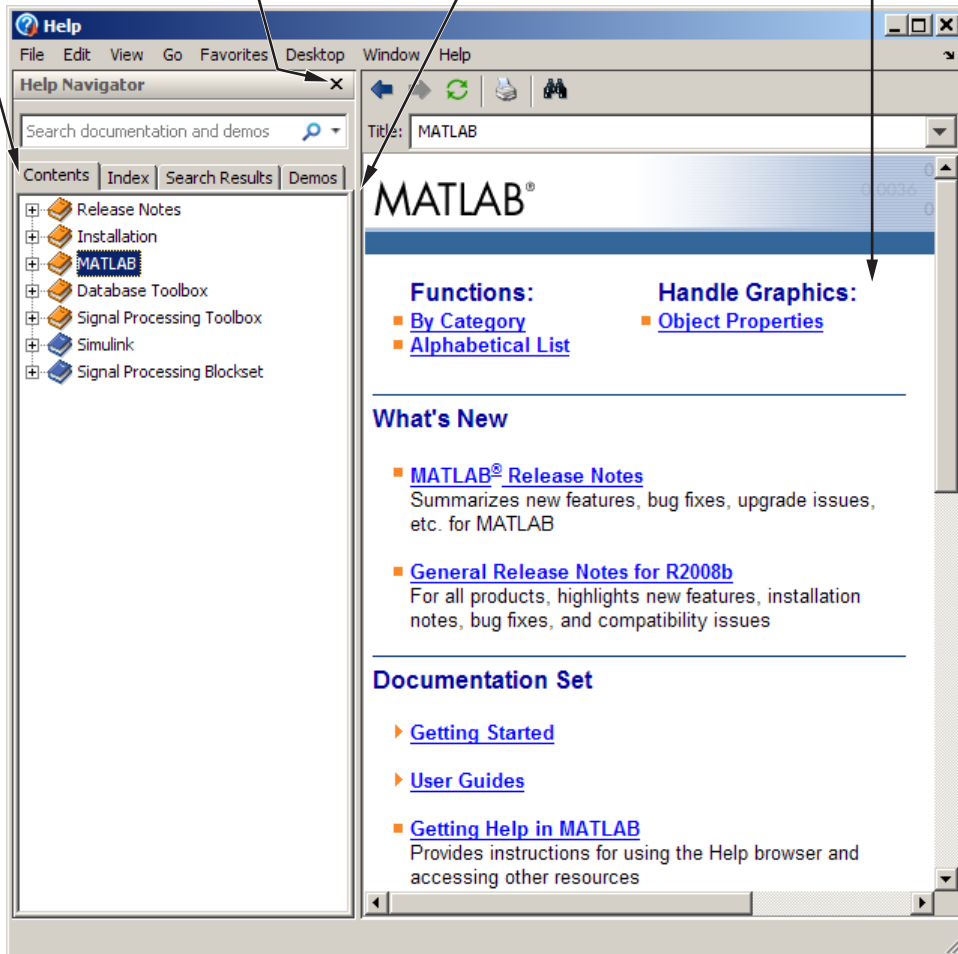
The Help browser consists of two panes: the **Help Navigator**, which you use to find information, and the display pane, where you view the information.

Tabs in the **Help Navigator** pane provide different ways to find information.

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.



These are the key features of the Help browser:

- **Search field** — Look for specific words in the documentation and demos. Type words in the field and press **Enter**. You can also:
 - 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** — 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 for the page you are viewing within the overall documentation.
- **Index** — Find specific index entries (selected keywords) in the documentation.
- **Search Results** — Displays results from the search, with the results in Documentation separated from the results in Demos.
- **Demos** — 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 perform the following tasks:

- **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 text button  in the toolbar.
- **Copy or evaluate a selection** — Select text in the help page, such as code from an example, then right-click and use context menu items 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 the reference page for the `format` function in the Help browser.

You also can access the reference page from the desktop. For example, you can use the Function Browser to quickly find functions and view documentation for them. To access the Function Browser, press **Shift+F1**.

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 consist of both uppercase and lowercase letters, and the help text clearly indicates that. For those functions, use the exact case shown in the Command Window help.

Other means for getting help include using Technical Support resources (www.mathworks.com/support) and participating in the Usenet newsgroup for users of MATLAB, `comp.soft-sys.matlab`, which you can access at MATLAB Central.

For More Information See “Help and Related Resources” and “Finding Functions Using the Function Browser” in the MATLAB Desktop Tools and Development Environment documentation. Also see the reference pages for the `doc` and `help` functions.

Typographical Conventions

These conventions are used in the Help browser and PDF documentation.

Item	Convention	Example
Buttons and keys	Boldface	Press the Enter key.

Item	Convention	Example
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	<i>Italics</i> for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.
Output from MATLAB	Monospace font	MATLAB responds with A = 5
Menu titles and items	Boldface	Select File > Save .
New terms and for emphasis	<i>Italics</i>	In MATLAB, a <i>matrix</i> is a rectangular array of numbers.
Omitted input arguments	An ellipsis (...) denotes all of the input/output arguments from preceding syntaxes	[c, ia, ib] = union(...)
String variables (from a finite list)	<i>Monospace italics</i>	format('type')

Current Directory, Search Path, and File Management

In this section...

“About the Current Directory and Search Path” on page 7-13

“Current Directory” on page 7-13

“Search Path” on page 7-15

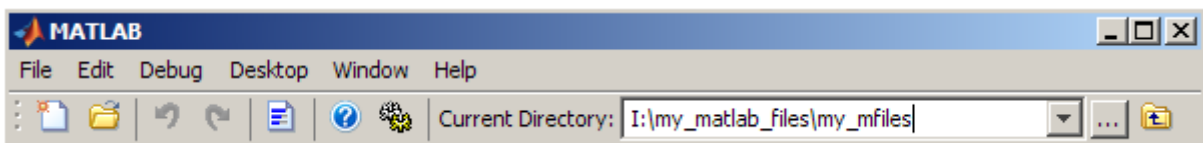
About the Current Directory and Search Path

The current directory and the search path are important reference points for the MATLAB software. Any file you want to work with in MATLAB must either be in the current directory or in a directory on the search path. The Current Directory browser is a primary tool you use to work with the current directory and manage files in MATLAB.

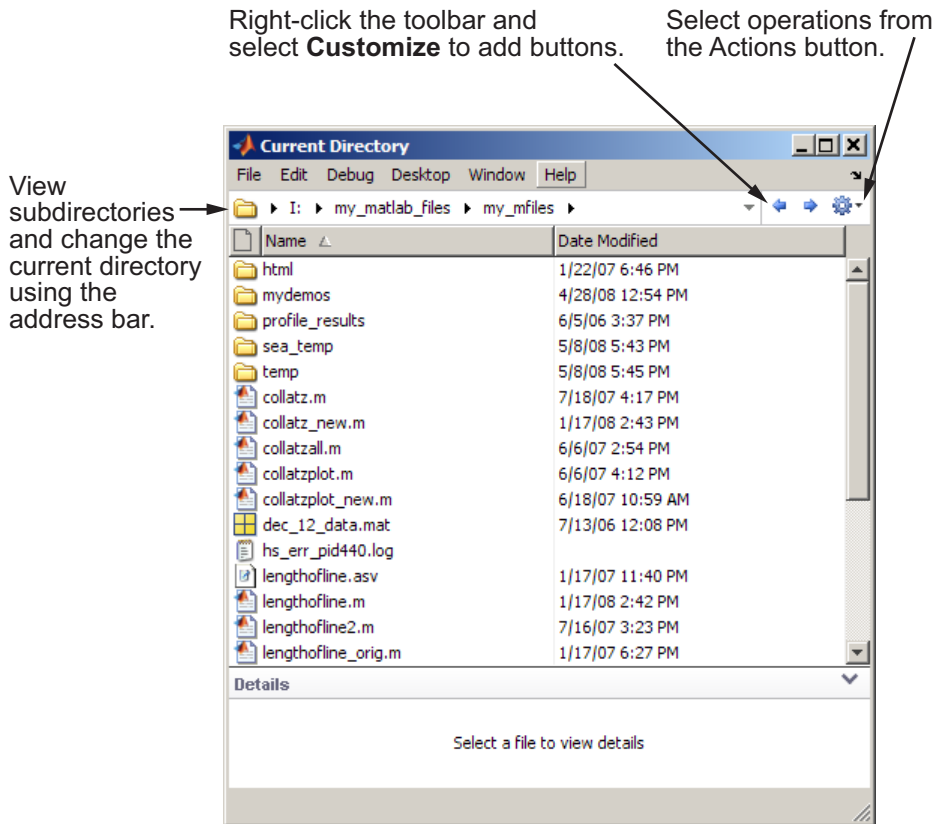
Current Directory

The current directory is also known as the present working directory or the current folder. You can specify the current directory for the MATLAB application when it first starts. This directory is called the *startup directory*.

A quick way to see the full path to the current directory or change it to a different directory is by using the Current Directory field in the desktop toolbar, as shown in the following illustration.



You can use the Current Directory browser, a desktop tool that provides a convenient way to organize and manage your files in MATLAB. You can view, open, and make changes to directories and files. Using the Current Directory browser, you can find files and use MATLAB application features on your files such as running them or opening them. Alternatively, you can use equivalent functions to organize and manage your files, including `dir`, `cd`, and `delete`.



There are many other features you can use in the Current Directory browser, such as generating reports to help you tune and manage your M-files. You can also interface with your source control system.

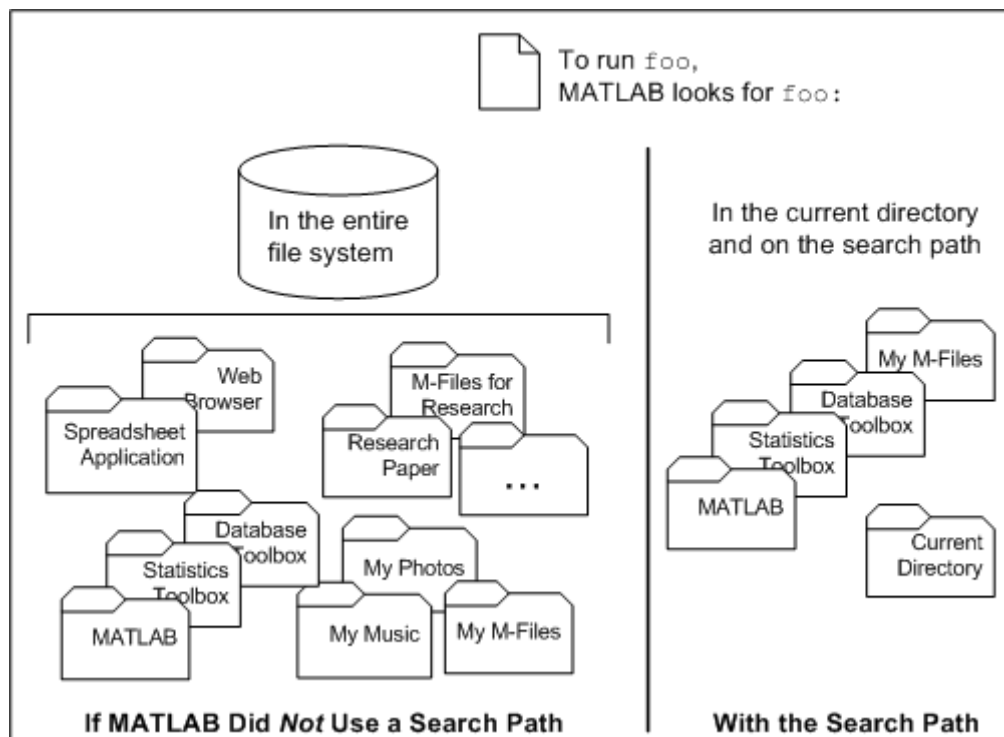
For More Information See “Managing Files and Working with the Current Directory” and “Source Control Interface” in the MATLAB Desktop Tools and Development Environment documentation. Also see the reference pages for the `dir`, `cd`, and `delete` functions.

Search Path

MATLAB uses a *search path* to find M-files and other files related to MATLAB, 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. By default, the search path includes files supplied with MATLAB and other MathWorks products. At startup, MATLAB also adds the `userpath` directory, a location where you can store files you create for use with MATLAB. If you have M-files and related files for MATLAB in other locations, add the directories in which they are located to the MATLAB search path. However, do not store the files in directories supplied by The MathWorks, or you might experience problems.

To see which directories are on the search path or to add directories to or remove them from 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.

The following illustration shows an example of how the MATLAB search path works.



The order of directories on the path can be important. When there are two files with the same name, one in the current directory and the other in a directory on the search path, or both in directories on the search path, it is considered a name clash, and MATLAB follows rules to determine which file to run. If you get unexpected results, MATLAB might not be running the file you think it is, because another file of the same name is located in a directory that is higher up on the search 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 Variable Editor

In this section...

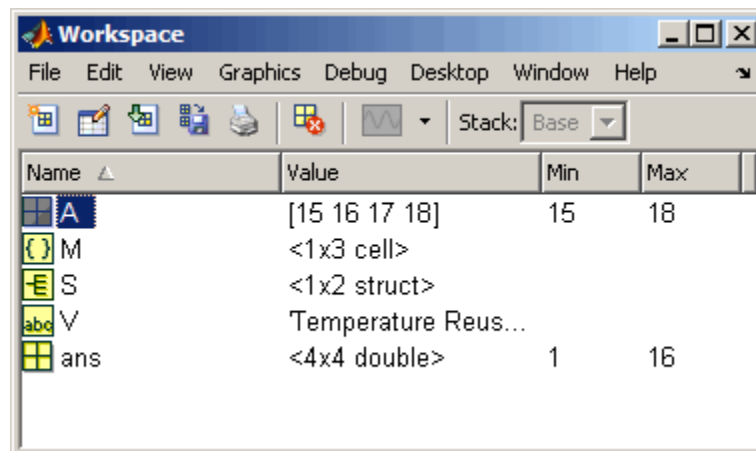
“Workspace Browser” on page 7-17

“Variable Editor” on page 7-18

Workspace Browser

The MATLAB workspace consists of the set of variables 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 then select **Edit > Delete**. Alternatively, use the `clearvars` or `clear` functions.

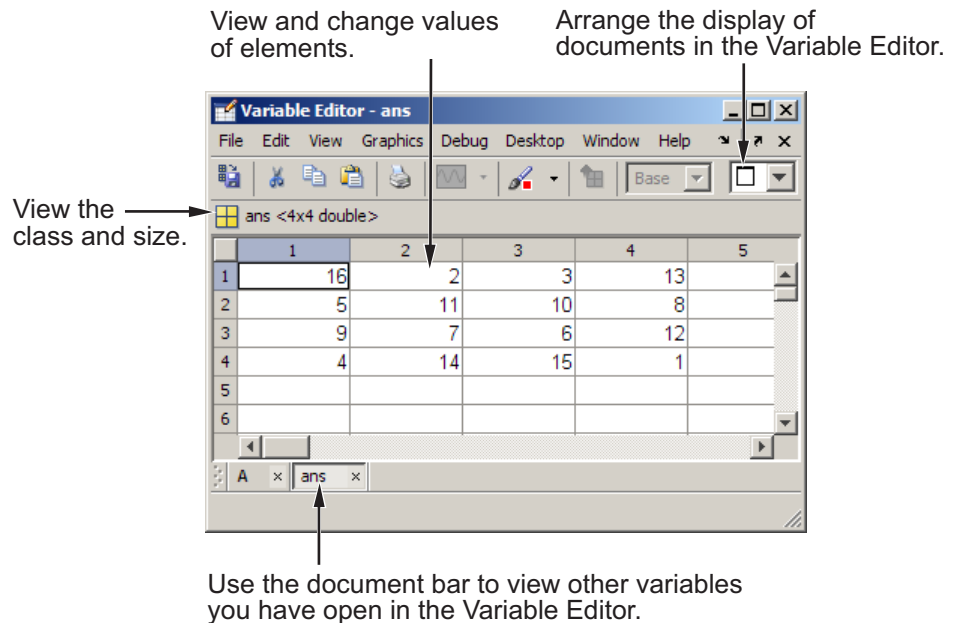
The workspace does not persist after you end the MATLAB session. To save the workspace to a file that can be read during a later MATLAB session, select **File > Save**, or use the `save` function. Saving preserves the workspace in a binary file called a MAT-file, which has a `.mat` extension. You can 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.

Variable Editor

Double-click a variable in the Workspace browser, or use `openvar variablename`, to see it in the Variable Editor. Use the Variable Editor to view and edit a visual representation of variables in the workspace.



For More Information See “Viewing and Editing Workspace Variables with the Variable Editor” in the MATLAB Desktop Tools and Development Environment documentation and the reference page for the `openvar` function.

Editor

In this section...
“Editing M-Files” on page 7-19
“Publishing M-Files” on page 7-21

Editing M-Files

Use the Editor to create and debug M-files, which are programs you write to run MATLAB functions. The Editor 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.

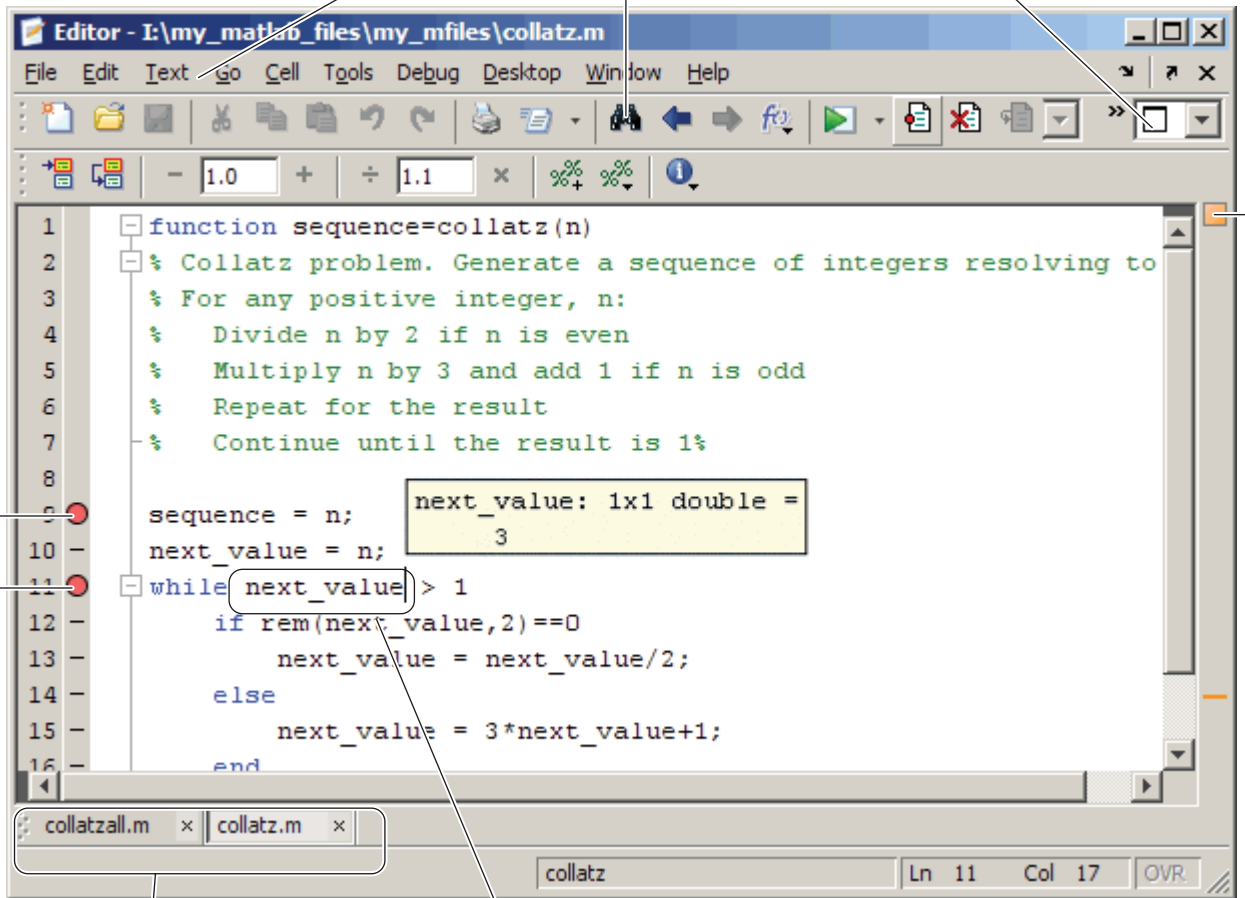
Set breakpoints where you want execution to pause so you can examine the variables.

Comment selected lines and specify the indenting style using the Text menu.

Arrange the display of documents in the Editor/Debugger.

M-lint automatic code analyzer.

Find and replace text.



Use the document bar to access other documents open in the Editor/Debugger.

Hold the cursor over a variable and its current value appears (known as a data tip).

You can use any text editor to create M-files, such as Emacs. Use Editor/Debugger preferences (accessible from the desktop by selecting **File > Preferences > Editor/Debugger**) to specify that editor as the default. If you use another editor, you can still use the MATLAB Editor 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.

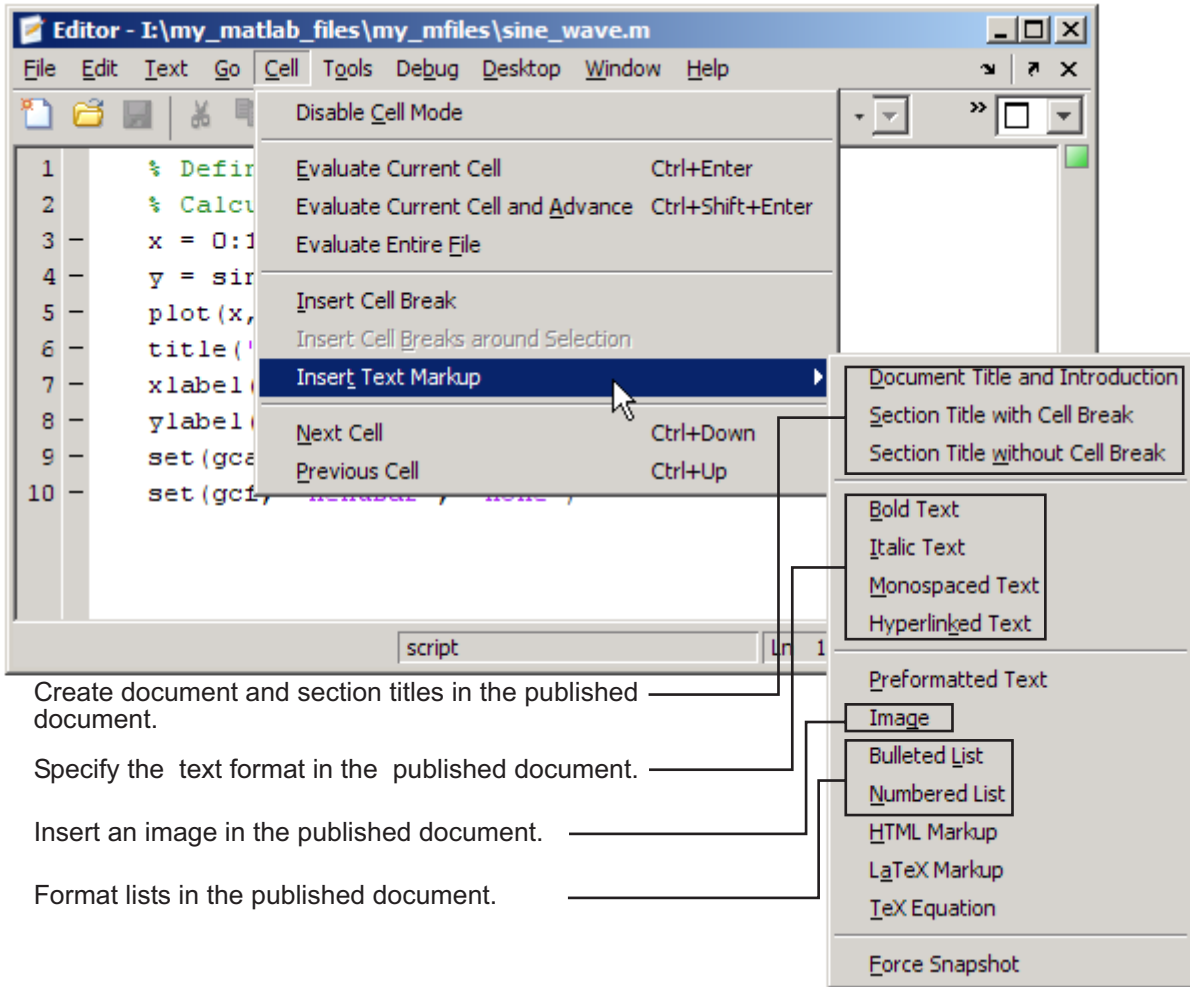
Use the M-Lint automatic code analyzer to help you identify problems and potential improvements in your code. For details, see “Improving and Tuning M-Files” on page 7-25.


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.

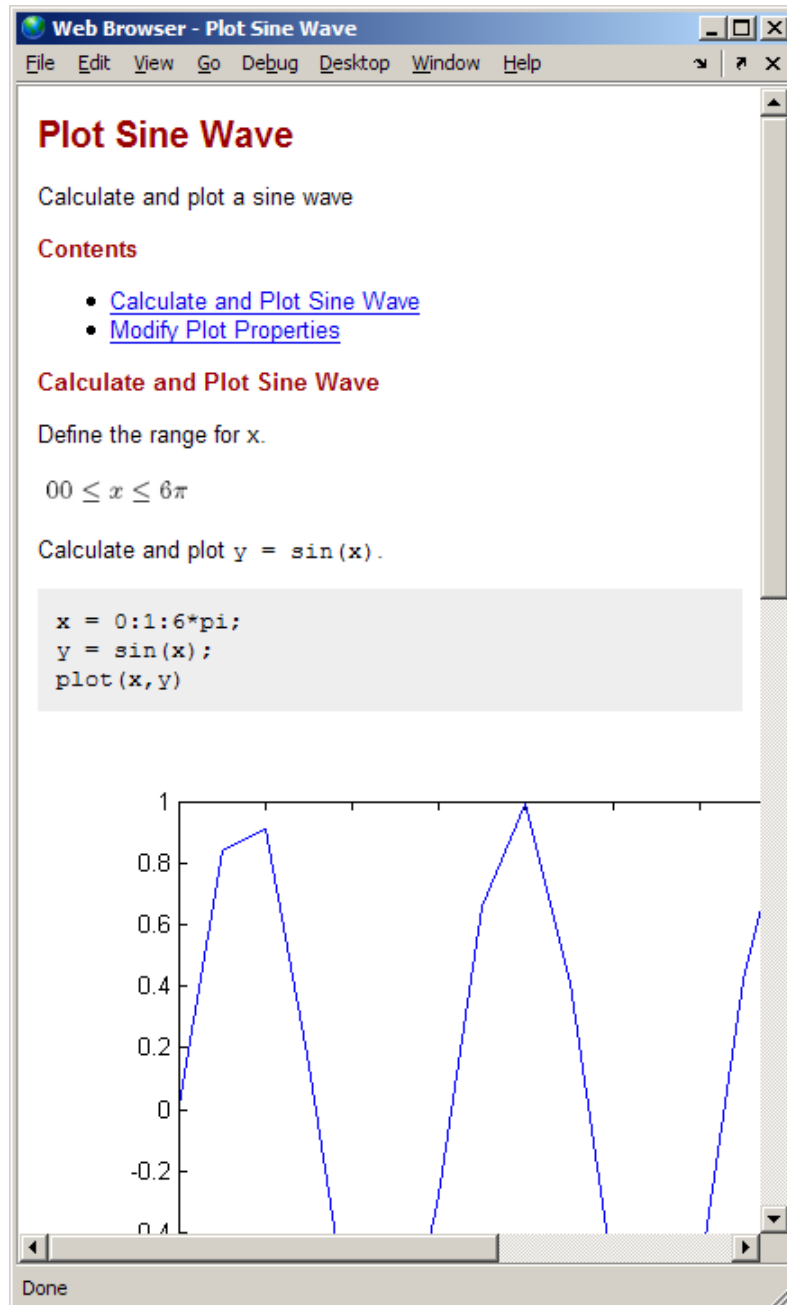
For More Information See “Editing and Debugging M-Files” in the MATLAB Desktop Tools and Development Environment documentation, and the function reference pages for `edit`, `type`, and `debug`.

Publishing M-Files

While or after you create an M-file, you can use the Editor’s **Cell > Insert Text Markup** menu options to specify the formatting you want for publishing the M-file to various output types. For example, you might want to publish an M-file to present it at a meeting, include it in a blog, or share it with colleagues.



After adding text markup, click the Publish button  on the Editor toolbar. By default, MATLAB software generates formatted HTML output and presents the document in the MATLAB Web Browser.



You can publish M-Files to HTML, XML, and LaTeX. If your system is a personal computer (PC), you also can publish M-Files to Microsoft Word and PowerPoint®.

For More Information See “Publishing M-Files” in the MATLAB Desktop Tools and Development Environment documentation, and the function reference pages for `publish`.

Improving and Tuning M-Files

In this section...

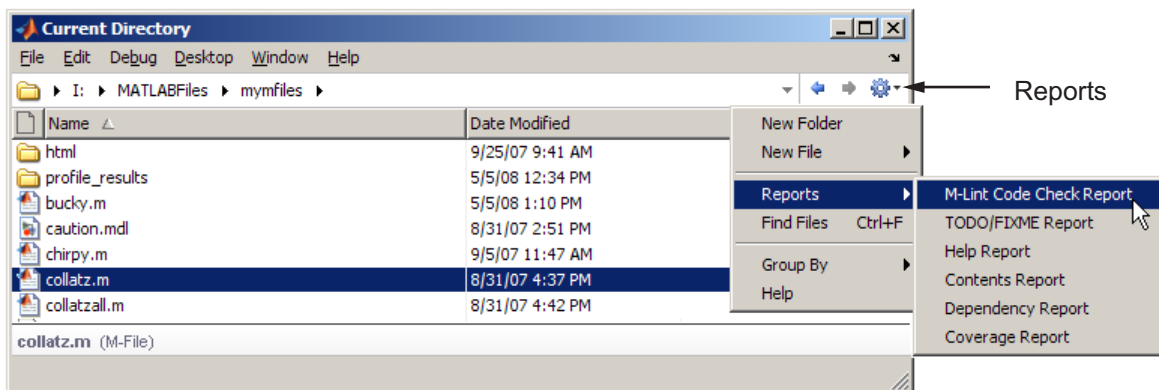
“Finding Errors Using the M-Lint Code Check Report” on page 7-25

“Improving Performance Using the Profiler” on page 7-27

Finding Errors Using the M-Lint Code Check Report

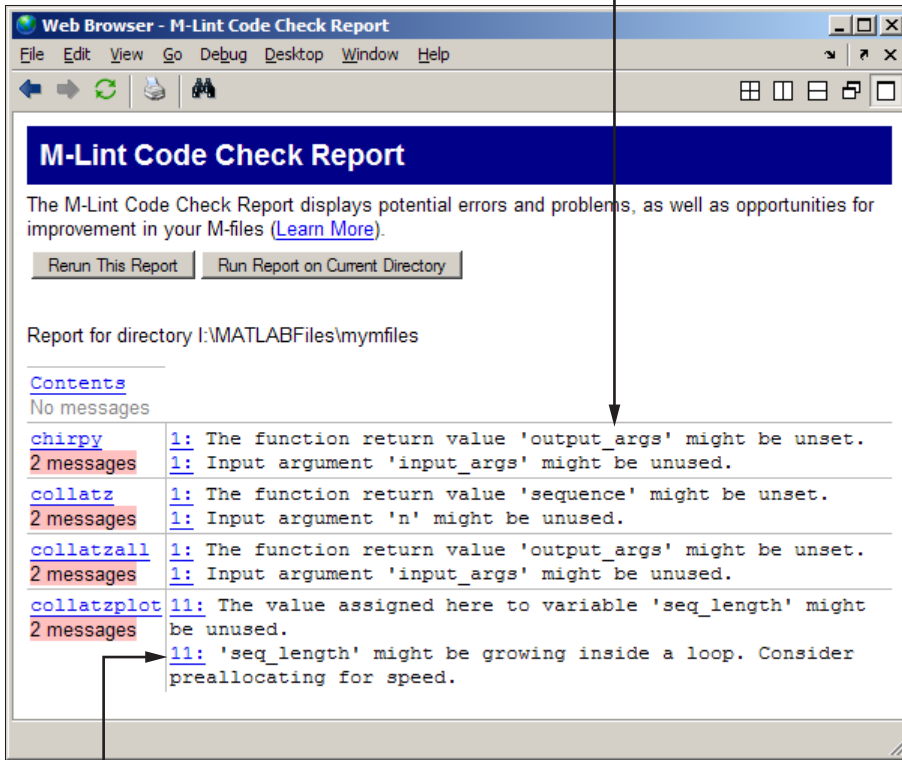
The M-Lint Code Check Report displays potential errors and problems, as well as opportunities for improvement in your M-files. The term *lint* is used by similar tools in other programming languages such as C.

Access the M-Lint Code Check Report and other reports from the Current Directory browser. You run a report for all files in the current directory.



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



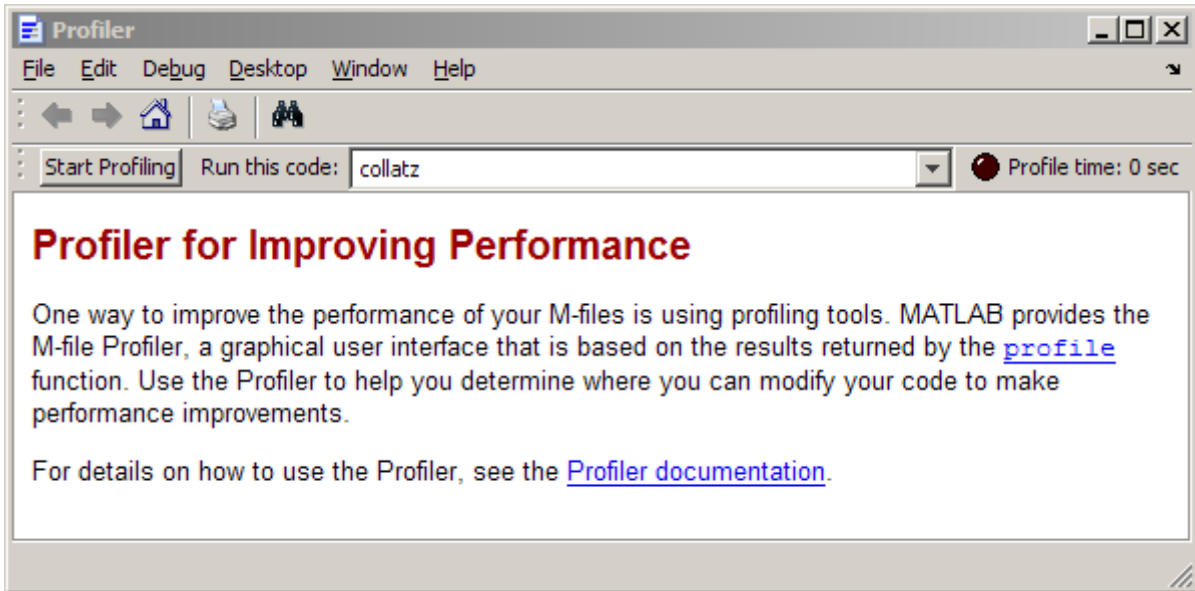
To open the M-file in the Editor at the line indicated, click the line number.

Alternatively, you can use automatic M-Lint code checking to view M-Lint messages while you work on a file in the Editor. 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.

Improving Performance Using the Profiler

The MATLAB Profiler helps 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.

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” on page 8-2
- “Component Object Model Interface” on page 8-4
- “Web Services” on page 8-5
- “Serial Port Interface” on page 8-6

Programming Interfaces

In this section...

“Call MATLAB Software from C and Fortran Programs” on page 8-2

“Call C and Fortran Programs from MATLAB Command Line” on page 8-2

“Call Sun Java Commands from MATLAB Command Line” on page 8-3

“Call Functions in Shared Libraries” on page 8-3

“Import and Export Data” on page 8-3

Call MATLAB Software 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 Command Line

Use MEX-files to call your own C or Fortran subroutines from the MATLAB command line 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 Sun Java Commands from MATLAB Command Line

Every installation of MATLAB software includes Java Virtual Machine (JVM™) software. The JVM software allows you to use the MATLAB interpreter with Java 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 Microsoft Windows platforms, a shared object file (.so) on The Open Group UNIX and Linus Torvald's Linux® platforms, or a dynamic shared library (.dylib) on Apple Macintosh platforms based on Intel® technology.

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 software can include Microsoft® 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 Microsoft® Visual Basic® or C programming languages.

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 the MATLAB software 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 perform the following tasks:

- Configure serial port communications.
- Use serial port control pins.
- Write and read data.
- Use events and callbacks.
- Record information to disk.

Symbols and Numerics

- : operator 2-8
- 2-D scatter plots
 - getting started 5-14
- 3-D scatter plots
 - getting started 5-16

A

- algorithms
 - vectorizing 4-31
- annotating plots 3-19
- ans function 2-5
- application program interface (API) 1-4
- array operators 2-24
- arrays
 - and matrices 2-24
 - cell 4-11
 - character 4-13
 - columnwise organization 2-26
 - creating in M-files 2-17
 - deleting rows and columns 2-19
 - elements 2-12
 - generating with functions and operators 2-16
 - listing contents 2-11
 - loading from external data files 2-17
 - multidimensional 4-9
 - notation for elements 2-12
 - preallocating 4-32
 - structure 4-16
 - variable names 2-11
- arrow keys for editing commands 2-32
- aspect ratio of axes 3-61
- axes
 - managing 3-61
 - visibility 3-62
- axis
 - labels 3-62
 - titles 3-62
- axis function 3-61

B

- bit map 3-74
- break function 4-7
- built-in functions
 - defined 2-14

C

- callbacks 6-7
- case function 4-4
- catch function 4-7
- cell arrays 4-11
- char function 4-15
- character arrays 4-13
- characteristic polynomial 2-23
- coefficient of determination
 - described 5-16
- colon operator 2-8
- colormap 3-67
- colors
 - lines for plotting 3-53
- Command History 7-7
- command line
 - editing 2-32
- Command Window 7-6
- complex numbers 2-12
 - plotting 3-56
- concatenation
 - defined 2-18
 - of strings 4-14
- constants
 - special 2-14
- continue function 4-6
- continuing statements on multiple lines 2-32
- control keys for editing commands 2-32
- correlation coefficient
 - example using corrcoef 5-16
- covariance
 - example using cov 5-15
- current directory 7-13

Current Directory browser 7-13
current folder. *See* current directory

D

data analysis
 getting started 5-1
data source
 for graphs 3-34
debugging M-files 7-19
deleting array elements 2-19
demos
 running from the Start button 7-4
desktop
 for MATLAB 1-7
 tools 7-1
determinant of matrix 2-21
diag function 2-5
distribution modeling
 getting started 5-11
documentation 7-8

E

editing command lines 2-32
Editor 7-19 7-21
eigenvalue 2-22
eigenvector 2-22
elements of arrays 2-12
entering matrices 2-4
eval function 4-28
executing MATLAB 1-7
exiting MATLAB 1-8
exporting graphs 3-48
expressions
 evaluating 4-28
 examples 2-15
 using in MATLAB 2-11

F

figure function 3-58
figure tools 3-7
figure windows 3-58
 with multiple plots 3-59
figures
 adding and removing graphs 3-4
filtering data
 getting started 5-6
find function 2-28
finding object handles 3-85
fliplr function 2-7
floating-point numbers 2-12
flow control 4-2
for loop 4-5
format
 of output display 2-30
format function 2-30
function functions 4-29
function handles
 defined 4-28
 using 4-30
function keyword 4-23
function M-files 4-20
 naming 4-22
function of two variables 3-65
functions
 built-in, defined 2-14
 defined 4-22
 how to find 2-13
 running 7-6
 variable number of arguments 4-23

G

global variables 4-26
graphical user interface
 creating 6-1
 laying out 6-3
 programming 6-7

graphics
 files 3-76
 Handle Graphics 3-78
 objects 3-79
 printing 3-75
 grids 3-62
 GUIDE 6-1

H

Handle Graphics 3-78
 defined 1-4
 finding handles 3-85
 Help browser 7-8
 help functions 7-11
 hold function 3-57

I

if function 4-2
 images 3-73
 imaginary numbers 2-12

K

keys for editing in Command Window 2-32

L

legend
 adding to plot 3-52
 legend function 3-52
 library
 mathematical function 1-3
 lighting 3-69
 limits
 axes 3-61
 line continuation 2-32
 line styles of plots 3-53
 linear regression
 getting started 5-28

load function 2-17
 loading arrays 2-17
 local variables 4-23
 log of functions used 7-7
 logical vectors 2-27

M

M-files
 and toolboxes 1-3
 creating 4-20
 editing 7-19
 for creating arrays 2-17
 function 4-20
 publishing 7-21
 script 4-20

magic function 2-9

magic square 2-5

markers 3-54

MAT-file 3-73

mathematical functions

library 1-3

listing advanced 2-13

listing elementary 2-13

listing matrix 2-13

MATLAB

application program interface 1-4

desktop 1-7

executing 1-7

exiting 1-8

history 1-2

language 1-3

mathematical function library 1-3

overview 1-2

quitting 1-8

running 1-7

shutting down 1-8

starting 1-7

user newsgroup 7-11

- matrices 2-20
 - creating 2-16
 - entering 2-4
- matrix 2-2
 - antidiagonal 2-7
 - determinant 2-21
 - main diagonal 2-6
 - multiplication 2-21
 - singular 2-21
 - swapping columns 2-10
 - symmetric 2-20
 - transpose 2-5
- measures of location
 - getting started 5-10
- measures of scale
 - getting started 5-11
- mesh plot 3-65
- missing data
 - getting started 5-3
- modeling data
 - getting started 5-27
- multidimensional arrays 4-9
- multiple data sets
 - plotting 3-52
- multiple plots per figure 3-59
- multivariate data
 - organizing 2-26

N

- newsgroup for MATLAB users 7-11
- numbers 2-12
 - complex 2-12
 - floating-point 2-12

O

- object oriented programming 4-33
- object properties 3-81

- objects
 - finding handles 3-85
 - graphics 3-79
- online help
 - viewing 7-8
- operators 2-13
 - colon 2-8
- outliers
 - getting started 5-4
- output
 - controlling format 2-30
 - suppressing 2-31
- overlying plots 3-57

P

- path 7-15
- periodogram 5-30
- plot edit mode
 - description 3-19
- plot function 3-51
- plots
 - editing 3-19
- plotting
 - adding legend 3-52
 - adding plots 3-57
 - basic 3-51
 - complex data 3-56
 - complex numbers 3-56
 - contours 3-57
 - editing 3-19
 - functions 3-51
 - line colors 3-53
 - line styles 3-53
 - lines and markers 3-54
 - mesh and surface 3-65
 - multiple data sets 3-52
 - multiple plots 3-59
 - overview 3-2
 - tools 3-10

- polynomial regression
 - getting started 5-27
- PostScript 3-76
- preallocation 4-32
- preprocessing data
 - getting started 5-3
- present working directory. *See* current directory
- presentation graphics 3-39
- principal components 5-17
- print function 3-75
- print preview
 - using 3-44
- printing
 - example 3-44
 - graphics 3-75
- Profiler 7-27
- Property Editor
 - interface 3-24
- Property Inspector 3-21
 - using 3-21
- publishing M-files 7-21

Q

- quitting MATLAB 1-8

R

- return function 4-8
- revision control systems
 - interfacing to MATLAB 7-13
- running
 - functions 7-6
 - MATLAB 1-7

S

- scalar expansion 2-27
- scatter plot arrays
 - getting started 5-18
- scientific notation 2-12

- script M-files 4-20
- scripts 4-21
- search path 7-15
- semicolon to suppress output 2-31
- shutting down MATLAB 1-8
- singular matrix 2-21
- smoothing data
 - getting started 5-6
- source control systems
 - interfacing to MATLAB 7-13
- special constants
 - infinity 2-14
 - not-a-number 2-14
- specialized graphs 3-58
- Start button 7-4
- starting MATLAB 1-7
- statements
 - continuing on multiple lines 2-32
 - executing 4-28
- strings
 - concatenating 4-14
- structures 4-16
- subplot function 3-59
- subscripting
 - with logical vectors 2-27
- subscripts 2-7
- sum function 2-5
- summarizing data
 - getting started 5-10
- suppressing output 2-31
- surface plot 3-65
- switch function 4-4
- symmetric matrix 2-20

T

- text
 - entering in MATLAB 4-13
- TIFF 3-76

- title
 - figure 3-62
- toolboxes 1-3
- tools in the desktop 7-1
- transpose function 2-5
- try function 4-7

V

- Variable Editor 7-18
- variables 2-11
 - global 4-26
 - local 4-23
- vectorization 4-31
- vectors 2-2
 - logical 2-27
 - preallocating 4-32

- version control systems
 - interfacing to MATLAB 7-13
- visibility of axes 3-62
- visualizing data
 - getting started 5-14

W

- while loop 4-5
- windows
 - in MATLAB 1-7
 - windows for plotting 3-58
- wireframe
 - surface 3-65
- working directory. *See* current directory
- workspace 7-17
- Workspace browser 7-17