

MATLAB[®]

The Language of Technical Computing

Computation

Visualization

Programming

Using MATLAB Graphics

Version 5.2

How to Contact The MathWorks:



508-647-7000

Phone



508-647-7001

Fax



The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760-1500

Mail



<http://www.mathworks.com>
<ftp.mathworks.com>
<comp.soft-sys.matlab>

Web
Anonymous FTP server
Newsgroup



support@mathworks.com
suggest@mathworks.com
bugs@mathworks.com
doc@mathworks.com
subscribe@mathworks.com
service@mathworks.com
info@mathworks.com

Technical support
Product enhancement suggestions
Bug reports
Documentation error reports
Subscribing user registration
Order status, license renewals, passcodes
Sales, pricing, and general information

Using MATLAB Graphics

© COPYRIGHT 1984 - 1997 by The MathWorks, Inc. All Rights Reserved.

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.

U.S. GOVERNMENT: If Licensee is acquiring the Programs on behalf of any unit or agency of the U.S. Government, the following shall apply: (a) For units of the Department of Defense: the Government shall have only the rights specified in the license under which the commercial computer software or commercial software documentation was obtained, as set forth in subparagraph (a) of the Rights in Commercial Computer Software or Commercial Software Documentation Clause at DFARS 227.7202-3, therefore the rights set forth herein shall apply; and (b) For any other unit or agency: NOTICE: Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction, and disclosure are as set forth in Clause 52.227-19 (c)(2) of the FAR.

MATLAB, Simulink, Handle Graphics, and Real-Time Workshop are registered trademarks and Stateflow and Target Language Compiler are trademarks of The MathWorks, Inc.

Other product or brand names are trademarks or registered trademarks of their respective holders.

Printing History:	December 1996	First printing	New for 5.0
	June 1997	Revised for 5.1	(Online version)
	January 1998	Revised for 5.2	(Online version)

Preface

What Is MATLAB?	ii
MATLAB Documentationiii
How to Use the Documentation Set.iii
Typographical and Alphabetic Conventionsiv

What Is MATLAB?

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

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

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

The name MATLAB stands for *matrix laboratory*. MATLAB was originally written to provide easy access to matrix software developed by the LINPACK and EISPACK projects, which together represent 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 application-specific solutions called *toolboxes*. Very important to most users of MATLAB, toolboxes allow you to *learn* and *apply* specialized technology. Toolboxes are comprehensive collections of MATLAB functions (M-files) that extend the MATLAB environment to solve particular classes of problems. Areas in which toolboxes are available include signal processing, control systems, neural networks, fuzzy logic, wavelets, simulation, and many others.

MATLAB Documentation

MATLAB comes with an extensive set of both online and printed documentation. The online MATLAB Function Reference is a compendium of all MATLAB commands functions. You can access this documentation from the MATLAB Help Desk. Users on all platforms can access the Help Desk with the MATLAB `doc` command. MS-Windows and Macintosh users can also access the Help Desk with the **Help** menu or the ? icon on the Command Window toolbar. From the Help Desk main menu, choose “MATLAB Functions” to display the Function Reference.

The online resources are augmented with printed documentation consisting of the following titles:

- *Getting Started with MATLAB* describes MATLAB fundamentals.
- *Using MATLAB* explains how to use MATLAB as both a programming language and a command-line application.
- *Using MATLAB Graphics* describes how to use MATLAB's graphics and visualization tools.
- *Building GUIs with MATLAB* discusses the construction of graphical user interfaces and introduces the Guide GUI building tool.
- The *MATLAB Application Programmer's Interface Guide* explains how to write C or Fortran programs that interact with MATLAB.
- The *MATLAB 5 New Features Guide* provides information useful in making the transition from MATLAB 4.x to MATLAB 5.
- The *MATLAB 5 Late-Breaking News* provides additional information about new features that are not covered in the other guides. They also include lists of problems fixed since the previous release and known documentation errors.

How to Use the Documentation Set

If you need to install MATLAB, you should read the appropriate booklet. Once you install MATLAB, you can decide which document you prefer to use to learn the MATLAB commands.

If you are a new MATLAB user, you should start by reading *Getting Started with MATLAB*. *Using MATLAB* provides an extensive description of the MATLAB language.

Using MATLAB Graphics describes how to use MATLAB for visualizing data with both high-level functions and Handle Graphics. Information about constructing user interfaces is provided in *Building GUIs with MATLAB*.

Typographical and Alphabetic Conventions

This manual uses certain typographical conventions.

Font	Usage
Monospace	Commands, function names, and screen displays; for example, conv.
<i>Monospace Italics</i>	Names of arguments that are meant to be replaced and not typed literally; for instance: cd <i>directory</i> .
<i>Italics</i>	Book titles, mathematical notation, and the introduction of new terms.
Boldface Initial Cap	Names of keys, such as the Return key.

Preface

What Is MATLAB?	ii
MATLAB Documentation	iii
How to Use the Documentation Set	iii
Typographical and Alphabetic Conventions	v

Introduction

1

Overview	1-2
High-Level Graphics	1-3
Handle Graphics	1-3
Building Interactive GUIs	1-3
How It All Fits Together	1-4
Where to Begin	1-4

Building 2-D Graphs

2

Building a 2-D Graph	2-2
Figure Windows	2-3
Multiple Axis Regions (subplot)	2-3
Default Color Scheme	2-5

Elementary Plotting Functions	2-7
Creating a Plot	2-7
Adding Plots to an Existing Graph (hold)	2-10
Matrix Data Plots	2-10
Imaginary and Complex Data	2-12
 Basic Plot Control	 2-14
Colors, Line Styles, and Markers	2-14
Axis Limits	2-15
Axis Tick Marks	2-16
Axes Aspect Ratio	2-17
 Graph Annotation	 2-19
Labeling the Individual Axes	2-19

Building 3-D Graphs

3

Building a 3-D Graph	3-2
 Elementary 3-D Plotting Functions	 3-3
Line Plots in 3-D	3-3
 Representing a Matrix as a Surface	 3-5
Mesh and Surface Plots	3-5
Visualizing Functions of Two Variables	3-6
Parametric Surfaces	3-9
Hidden Line Removal	3-11
 Coloring Mesh and Surface Plots	 3-12
Colormaps and Indexed Colors	3-12
Truecolor	3-17
Texture Mapping	3-19
 Lighting	 3-21
Light Objects	3-21
Properties that Affect Lighting	3-22

Controlling the Effects of Lighting	3-24
Face and Edge Lighting Methods	3-24
Reflectance Characteristics of Graphics Objects	3-25
Lighting Example	3-30
Viewpoint Control	3-32
Setting the Viewpoint	3-32
Camera Properties	3-35
Default Behavior	3-36
Moving In and Out on the Scene	3-37
Revolving Around the Scene	3-39
Translating the Viewpoint	3-41
View Projection Types	3-43
Projection Types and Camera Location	3-44
Aspect Ratio	3-47
Stretch-to-fill	3-47
axis Command Options	3-47
Properties That Affect Aspect Ratio	3-51
Default Behavior	3-52
Overriding Stretch-to-Fill	3-54
Specifying the Aspect Ratio	3-55

Specialized Graphs

4

Bar and Area Graphs	4-2
Bar Graph	4-2
Overlaying Plots on Bar Graphs	4-8
Area Graphs	4-10
Pie Charts	4-13
Pie Charts Missing a Piece	4-15

Histograms	4-16
Histograms in Cartesian Coordinate Systems	4-16
Histograms in Polar Coordinate Systems	4-17
Discrete Data Graphs	4-20
Two- and Three-dimensional Stem Plots	4-20
Stairstep Plots	4-26
Direction and Velocity Vector Graphs	4-28
Compass Plots	4-28
Feather Plots	4-29
Quiver Plots	4-31
Contour Plots	4-34
Creating Simple Contour Plots	4-34
Labeling Contours	4-36
Filled Contours	4-37
Drawing a Single Contour Line at a Desired Level	4-38
The Contouring Algorithm	4-38
Changing the Offset of a Contour	4-40
Displaying Contours in Polar Coordinates	4-40
Interactive Plotting	4-43
Animation	4-45
Movies	4-45
Erase Modes	4-47

Images

5

Overview	5-2
Image Types	5-3
Indexed Images	5-3
Intensity Images	5-3
Truecolor Images	5-4

Summary of Image Types and Display Methods	5-5
Working with 8-Bit Images	5-6
8-Bit Indexed Images	5-6
8-Bit Intensity Images	5-7
8-Bit Truecolor Images	5-7
Summary of Image Types and Numeric Class	5-8
Other 8-Bit Array Support	5-9
Controlling Aspect Ratio and Display Size	5-10
Printing Images	5-13
The Image Object and its Properties	5-14
CData	5-14
CDataMapping	5-14
XData and YData	5-15
EraseMode	5-17
Reading and Writing Image Files	5-19

3-D Modeling

6

Introduction to Patches	6-2
Defining Patches	6-2
Behavior of the patch Function	6-4
Patches with Multiple Faces	6-6
Example – Multifaceted Patch	6-6
Patch Coloring	6-11
Face and Edge Coloring	6-12
Interpreting Color Data	6-14
Interpolating in Indexed vs. Truecolor	6-18

Introduction	7-2
Printing from the Menu	7-3
PC	7-3
Macintosh	7-4
UNIX	7-4
Adjusting the Size and Location of the Graphic	7-5
Printing from the Command Line	7-6
The print Command	7-6
Options	7-11
Selecting a Device Driver	7-17
PostScript	7-17
HPGL Compatible Plotters (-dhpgl)	7-18
Adobe Illustrator 88 (-dill)	7-20
Saving and Reloading Figures (-dmfile)	7-20
PC-Specific Options	7-21
Macintosh-Specific Options	7-24
Printing Tips and Troubleshooting	7-25
Controlling Output Size and Aspect Ratio	7-25
Specifying Fonts and Character Sets	7-27
Specifying Line Styles	7-29
Selecting the Rendering Method	7-32
Changing Background Colors	7-35
Setting Printing Preferences (Macintosh)	7-36
Troubleshooting MS-Windows Printing	7-36
Using MATLAB Graphics in Other Applications	7-37
Creating Graphics Files	7-37
Importing MATLAB Graphics into Other Applications	7-40

Handle Graphics Organization	8-2
Graphics Objects	8-2
Object Properties	8-7
 Graphics Object Creation Functions	8-10
Example – Creating Graphics Objects	8-11
Parenting	8-12
High-Level Versus Low-Level	8-13
Simplified Calling Syntax	8-13
 Using set and get	8-15
Setting Property Values	8-15
Getting Property Values	8-17
Factory-Defined Property Values	8-19
 Default Property Values	8-20
Specifying Default Values	8-22
Examples – Setting Defaults	8-23
 Accessing Object Handles	8-27
The Current Figure, Axes, and Object	8-27
Searching for Objects by Property Values — findobj	8-29
Copying and Deleting Objects	8-30
 Controlling Graphics Output	8-33
Specifying the Target for Graphics Output	8-33
Preparing Figures and Axes for Graphics	8-33
Testing for Hold State	8-38
Protecting Figures and Axes	8-39
 Efficient Programming	8-44
Save Information First	8-44

Properties Changed by Built-In Functions	8-45
---	-------------

9

Figures

Introduction	9-2
Figure Properties	9-3
Positioning Figures	9-5
The Position Vector	9-5
Example — Specifying Figure Position	9-7
Controlling Color	9-8
Indexed Color Displays	9-8
Colormap Colors and Fixed Colors	9-9
Using a Large Number of Colors	9-10
Nonactive Figures and Shared Colors	9-12
Dithering Truecolor on Indexed Color Systems	9-13
Rendering Options	9-15
Backing Store	9-15
Z-Buffer	9-15
Figure Pointers	9-17
Custom Pointers	9-18
Printing Figures	9-21
Positioning the Figure on the Printed Page	9-21
Examples — Readjusting PaperPosition	9-23
Reversing Figure Colors	9-24

Axes Properties	10-2
Labeling and Appearance Properties	10-4
TeX Characters	10-6
Adding Text to Axes	10-8
Text Alignment	10-9
Using Variables in Text Strings	10-10
Example – Text Annotation	10-10
Example – Multiline Text	10-12
Positioning Axes	10-13
The Position Vector	10-13
Units	10-14
Multiple Axes	10-15
Individual Axis Control	10-18
Changing Axis Limits	10-18
Setting Tick Mark Locations	10-20
Changing Axis Direction	10-21
Automatic-Mode Properties	10-23
Multiaxis Axes	10-26
Example – Double Axis Graphs	10-26
Colors Controlled By Axes	10-29
Axes Colors	10-29
Axes Color Limits – The CLim Property	10-31
Color of Lines Used for Plotting	10-37

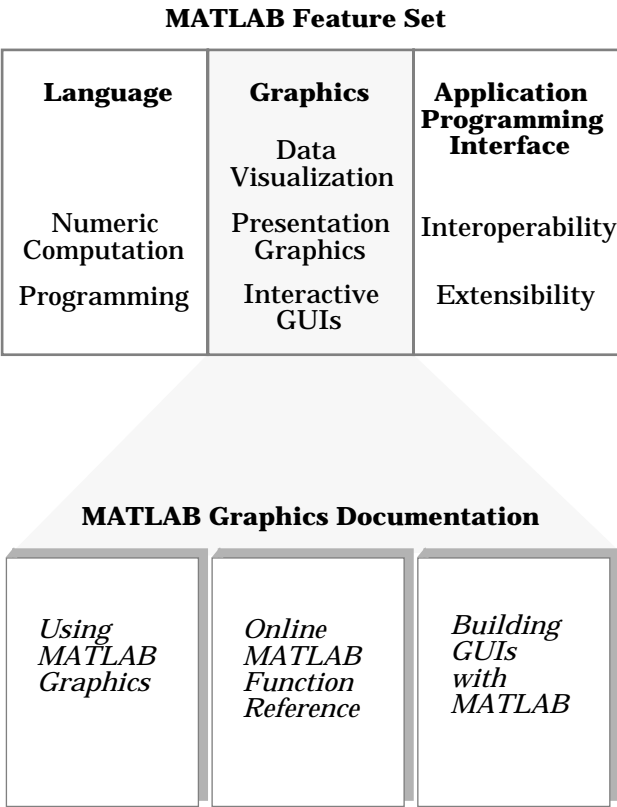
Introduction

Overview	1-1
High-Level Graphics	1-3
Handle Graphics	1-3
Building Interactive GUIs	1-3
How It All Fits Together	1-4
Where to Begin	1-4

Overview

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

This manual describes MATLAB graphics features for visualizing data and preparing presentation graphics. The organization of the manual is based on the organization of the commands and functions: end-user oriented high-level graphics functions and the programmable interface provided by Handle Graphics®.



High-Level Graphics

MATLAB provides a set of high-level graphing routines. These routines implement commonly used techniques for displaying data, such as line plots in rectangular and polar coordinates, bar and histogram graphs, contour plots, mesh and surface plots, and animation. In addition, you can control color and shading, axis labeling, and the general appearance of graphs. High-level commands automatically control plot characteristics such as axis scaling and line color to produce acceptable graphs without requiring you to manipulate low-level properties.

Handle Graphics

You can exert more precise control over the way MATLAB displays data or you can develop your own graphics commands using Handle Graphics, MATLAB's object-oriented graphics system. Handle Graphics defines a set of graphics objects, such as Lines, Surfaces, and Text, and provides mechanisms to manipulate the characteristics of these objects to achieve the desired results. You can use Handle Graphics in a number of ways:

- On the command line, you can “fine tune” the appearance of your plots by altering the properties of the graphics objects used to display your data.
- In M-files, you can define your own graphics commands that provide precise control over the graphics display.
- Within existing M-files, which include many high-level graphics commands, you can customize the behavior to meet your specific requirements.

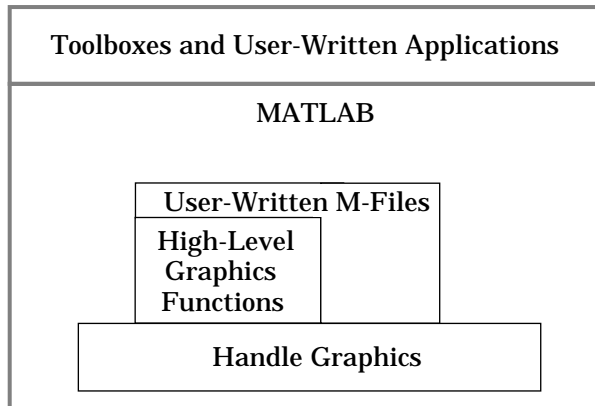
Building Interactive GUIs

Using Handle Graphics, you can create menus, push buttons, text boxes, and other user interface devices that allow your MATLAB program to obtain user input and process this input within MATLAB.

With Handle Graphics, you can add a GUI to any M-file or define your own environment that starts whenever you begin a MATLAB session. You can build sophisticated user interfaces for any MATLAB-based application. The *Building GUIs with MATLAB* manual discusses this material.

How It All Fits Together

Handle Graphics provides the basis for the high-level graphics functions supplied with MATLAB. User-written M-files that perform graphics operations can use both high-level functions and Handle Graphics directly.



Where to Begin

If you are new to MATLAB, you will probably find high-level graphics functions suitable for most of your plotting needs. If you want to customize the way high-level routines work or if you want to create your own routines, you should delve into Handle Graphics.

Building 2-D Graphs

Building a 2-D Graph	2-2
Figure Windows	2-3
Multiple Axis Regions (subplot)	2-3
Default Color Scheme	2-5
Elementary Plotting Functions	2-7
Creating a Plot	2-7
Adding Plots to an Existing Graph (hold)	2-10
Matrix Data Plots	2-10
Imaginary and Complex Data	2-12
Basic Plot Control	2-14
Colors, Line Styles, and Markers	2-14
Axis Limits	2-15
Axis Tick Marks	2-16
Axes Aspect Ratio	2-17
Graph Annotation	2-19
Labeling the Individual Axes	2-19

Building a 2-D Graph

The process of constructing a 2-D graph to meet your presentation graphics needs can take as few as one step or as many as seven steps. The table below shows seven typical steps and some example code for each.

If you are only doing analysis, you may want to view various graphs just to explore your data. In this case, steps 1 and 3 may be all you need. When creating presentation graphics, you may want to fine-tune your graph by positioning it on the page, setting line styles and colors, adding annotations, and making other such improvements.

Step	Typical Code
1 Prepare your data	<pre>x = 0: . 2: 12; y1 = bessel (1, x); y2 = bessel (2, x); y3 = bessel (3, x);</pre>
2 Select window and position plot region within window	<pre>fi gure(1) subpl ot(2, 2, 1)</pre>
3 Call elementary plotting function	<pre>h = pl ot(x, y1, x, y2, x, y3);</pre>
4 Select line and marker characteristics	<pre>set(h, 'Li neWi dth', 2, {'Li neStyl e'}, {'--'; ':' ; '-. '}) set(h, {'Col or'}, {'r'; 'g'; 'b'})</pre>
5 Set axis limits, tick marks, and grid lines	<pre>axi s([0 12 -0. 5 1]) grid on</pre>
6 Annotate the graph with axis labels, legend, and text	<pre>xl abel ('Ti me') yl abel ('Ampl i tude') legend(h, 'Fi rst', 'Seco nd', 'Thi rd') title('Bessel Functions') [y, ix] = mi n(y1); text(x(ix), y, 'Fi rst Mi n \righ tarrow', ... 'Hori zo ntal Al i gnment', 'ri ght')</pre>
7 Print graph	<pre>pr int -dps2</pre>

This chapter describes each step in sequence and provides examples of the options available. Note that printing is described in the *Printing* chapter.

Figure Windows

MATLAB directs graphics output to a window separate from the command window called a *Figure window*. The characteristics of this window are controlled by your computer's windowing system and MATLAB Figure properties.

Graphics functions automatically create new Figure windows if none currently exist. If a Figure window already exists, MATLAB uses that window. If multiple Figure windows exist, one is designated as the current Figure and is used by MATLAB (this is generally the last Figure window used).

The `figure` function creates Figure windows. For example,

```
figure
```

creates a new window and makes it the current target for graphics output. You can make an existing Figure current by clicking on it with the mouse or by passing its number, which is indicated in the window title bar, as an argument to `figure`:

```
figure(h)
```

See the `figure` function description in the online MATLAB Function Reference for more information on Figure properties. See the *Figure* chapter for more information on target window selection.

Multiple Axis Regions (subplot)

You can display multiple plots in the same Figure window and print them on the same piece of paper with the `subplot` function.

`subplot(m, n, i)` breaks the Figure window into an m -by- n matrix of small subplots and selects the i th subplot for the current plot. The plots are numbered along the top row of the Figure window, then the second row, and so forth.

For example, the following statements plot data in four different subregions of the Figure window.

```

t = 0: pi /20: 2*pi ;
[x, y] = meshgrid(t);

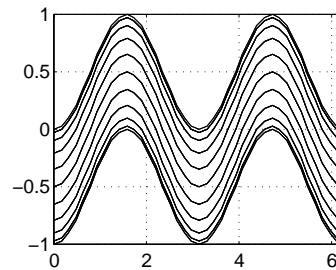
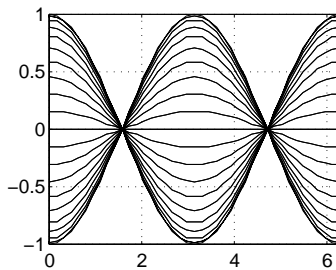
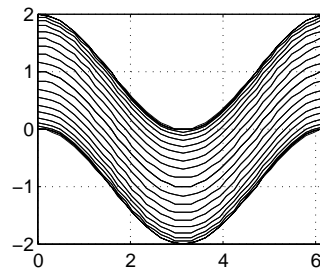
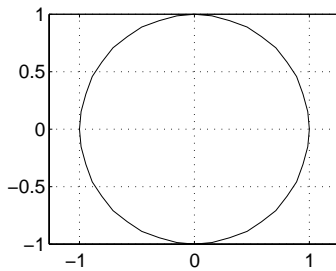
subplot(2, 2, 1)
plot(sin(t), cos(t))
axis equal

subplot(2, 2, 2)
z = sin(x)+cos(y);
plot(t, z)
axis([0 2*pi -2 2])

subplot(2, 2, 3)
z = sin(x) .* cos(y);
plot(t, z)
axis([0 2*pi -1 1])

subplot(2, 2, 4)
z = (sin(x) . ^2) - (cos(y) . ^2);
plot(t, z)
axis([0 2*pi -1 1])

```



Each subregion contains its own axes with characteristics you can control independently of the other subregions. This example uses the `axis` command to set limits and change the shape of the subplots.

See the `axes`, `axis`, and `subplot` functions in the online MATLAB Function Reference for more information.

Specifying the Target Axes

The current axes is the last one defined by `subplot`. If you want to access a previously defined subplot, for example to add a title, you must first make that axes current.

You can make an axes current in three ways:

- Click on the subplot with the mouse
- Call `subplot` the `m, n, i` specifiers
- Call `subplot` with the handle (identifier) of the axes

For example,

```
subplot(2, 2, 2)
title('Top Right Plot')
```

adds a title to the plot in the upper-right side of the Figure.

You can obtain the handles of all the subplot axes with the statement:

```
h = get(gcf, 'Children');
```

MATLAB returns the handles of all the axes, with the most recently created one first. That is, `h(1)` is subplot 224, `h(2)` is subplot 223, `h(3)` is subplot 222, and `h(4)` is subplot 221. For example, to replace subplot 222 with a new plot, first make it the current axes with:

```
subplot(h(3))
```

Default Color Scheme

The default Figure color scheme produces good contrast and visibility for the various graphics functions. This scheme defines colors for the window background, the axis background, the axis lines and labels, the colors of the lines used for plotting and surface edges, and other properties that affect appearance.

The `col ordef` function enables you to select from predefined color schemes and to modify colors individually. `col ordef` predefines three color schemes:

- `col ordef whi te` – sets the axis background color to white, the window background color to gray, the colormap to `jet`, surface edge colors to black, and defines appropriate values for the plotting color order and other properties.
- `col ordef bl ack` – sets the axis background color to black, the window background color to dark gray, the colormap to `jet`, surface edge colors to black, and defines appropriate values for the plotting color order and other properties.
- `col ordef none` – set the colors to match that of MATLAB 4. This is basically a black background with white axis lines and no grid. MATLAB programs that are based on the MATLAB 4 color scheme may need to call `col ordef` with the `none` option to produce the expected results.

You can examine the `col ordef.m` M-file to determine what properties it sets (enter `type col ordef` at the MATLAB prompt). See the *Handle Graphics* chapter for information on setting properties individually.

Elementary Plotting Functions

MATLAB provides a variety of functions for displaying vector data as graphs, as well as functions for annotating and printing these graphs. This section describes these functions and provides examples of some typical applications.

The following table summarizes the functions that produce basic line plots of data. These functions differ only in the way they scale the plot's axes. Each accepts input in the form of vectors or matrices and automatically scales the axes to accommodate the data.

Function	Used to Create
<code>plot</code>	Graph with linear scales for both axes
<code>loglog</code>	Graph with logarithmic scales for both axes
<code>semilogx</code>	Graph with a logarithmic scale for the x -axis and a linear scale for the y -axis
<code>semilogy</code>	Graph with a logarithmic scale for the y -axis and a linear scale for the x -axis
<code>plotyy</code>	Graph with y -tick labels on the left and right side

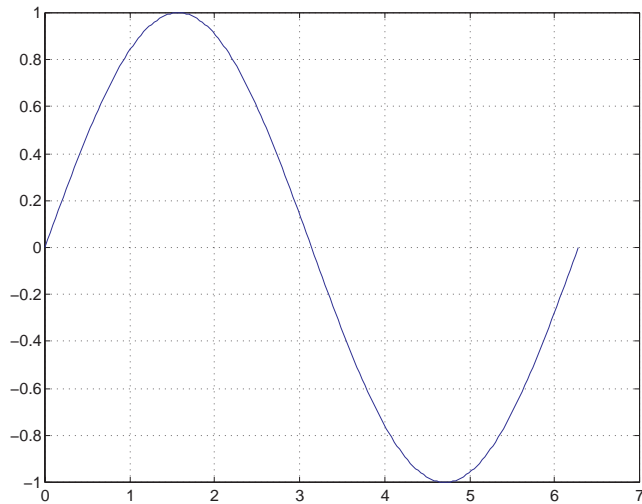
Creating a Plot

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

For example, these statements create a vector of values in the range $[0, 2\pi]$ in increments of $\pi/100$ and then use this vector to evaluate the sine function over that range. MATLAB plots the vector on the x -axis and the value of the sine function on the y -axis.

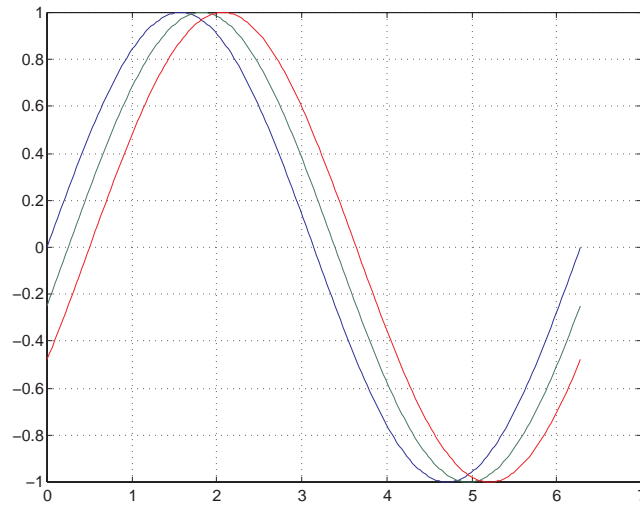
```
t = 0: pi / 100: 2*pi ;
y = sin(t);
plot(t, y)
```

MATLAB automatically selects appropriate axis ranges and tick mark locations:



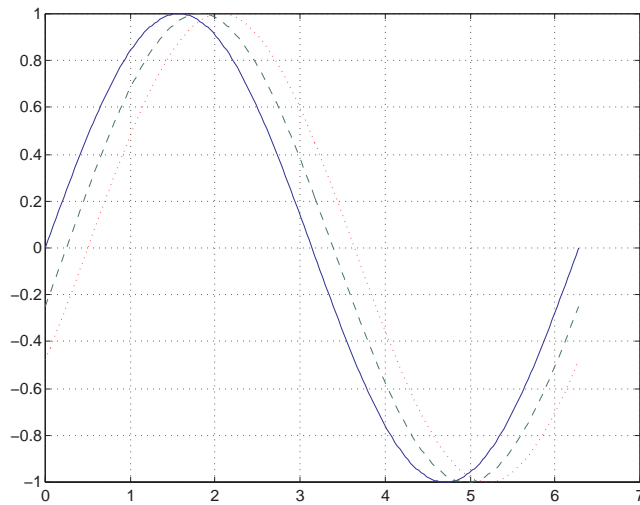
You can plot multiple graphs in one call to `plot` using x - y pairs. MATLAB automatically cycles through a predefined list of colors to allow discrimination between each set of data. Plotting three curves as a function of t produces:

```
y2 = sin(t-.25);  
y3 = sin(t-.5);  
plot(t, y, t, y2, t, y3)
```



You can assign different line styles to each data set by passing line style identifier strings to `plot`. Line styles are useful if you are printing the graph on a black and white printer. For example,

```
plot(t, y, '-', t, y2, '--', t, y3, ':')
```



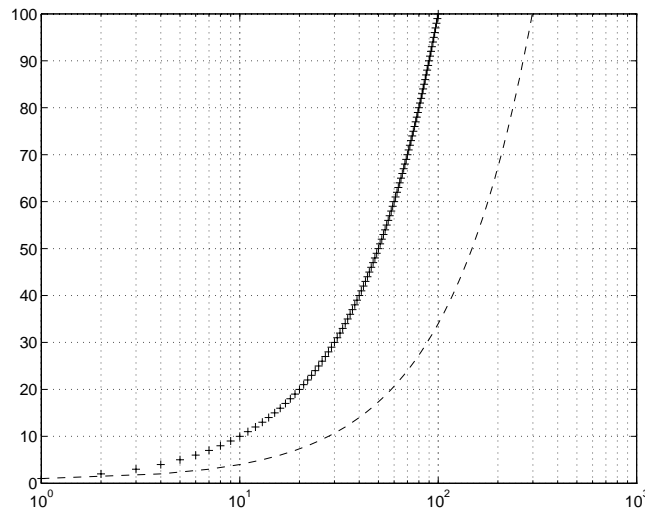
Adding Plots to an Existing Graph (hold)

You can add plots to an existing graph using the `hold` command. When you set `hold` to on, MATLAB does not remove the existing graph; it adds the new data to the current graph, rescaling if the new data falls outside the range of the previous axis limits.

For example, these statements first create a semilogarithmic plot, then add a linear plot:

```
semilogx(1:100, '+' )
hold on
plot(1:3:300, 1:100, '--')
hold off
```

While MATLAB resets the x -axis limits to accommodate the new data, it does not change the scaling from logarithmic to linear.



Matrix Data Plots

When you call the `plot` function with a single matrix argument,

```
plot(Y)
```

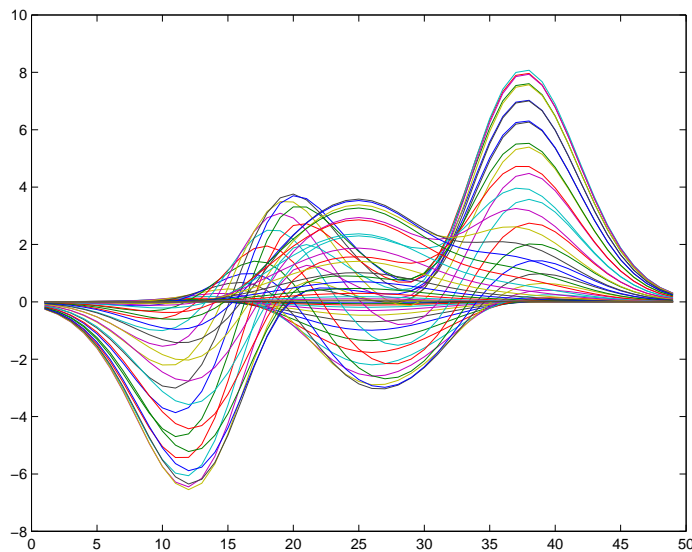
MATLAB draws one line for each column of the matrix. The x -axis is labeled with the row index vector, $1:m$, where m is the number of rows in Y . For example,

```
Z = peaks;
```

returns a 49-by-49 matrix obtained by evaluating a function of two variables. Plotting this matrix,

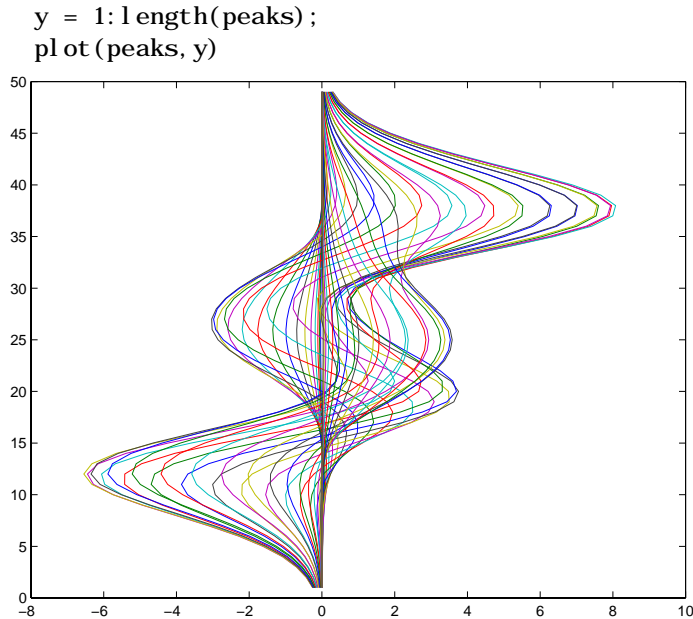
```
plot(Z)
```

produces a graph with 49 lines:



In general, if `plot` is used with two arguments and if either X or Y has more than one row or column, then

- If Y is a matrix, and x is a vector, `plot(x, Y)` successively plots the rows or columns of Y versus vector x , using different colors or line types for each. The row or column orientation is dependent on whether the number of elements in x matches the number of rows in Y or the number of columns. If Y is square, its columns are used.
- If X is a matrix and y is a vector, `plot(X, y)` plots each row or column of X versus vector y . For example, plotting the peaks matrix versus the vector `1:length(peaks)` rotates the previous plot.



- If X and Y are both matrices of the same size, `plot(X, Y)` plots the columns of X versus the columns of Y .

You can also use the `plot` function with multiple pairs of matrix arguments:

```
plot(X1, Y1, X2, Y2, ...)
```

This statement graphs each X - Y pair, generating multiple lines. The different pairs can be of different dimensions.

Imaginary and Complex Data

When the arguments to `plot` are complex (i.e., the imaginary part is nonzero), MATLAB ignores the imaginary part *except* when `plot` is given a single complex argument. For this special case, the command is a shortcut for a plot of the real part versus the imaginary part. Therefore,

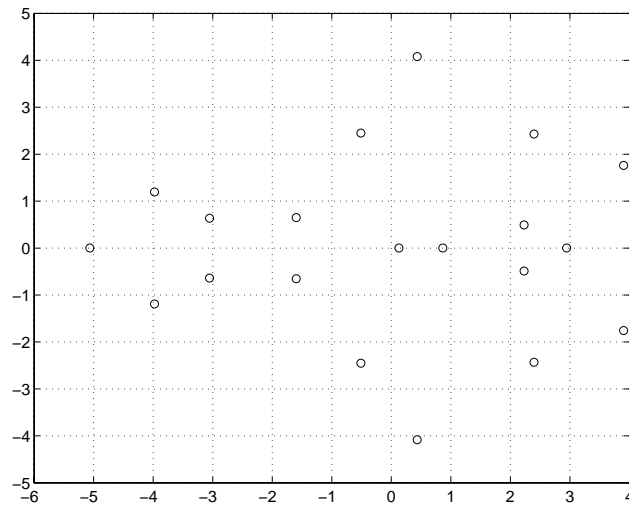
```
plot(Z)
```

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

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

For example, this statement plots the distribution of the eigenvalues of a random matrix using circular markers to indicate the data points:

```
plot(eig(randn(20,20)), 'o', 'MarkerSize', 6)
```



To plot more than one complex matrix, there is no shortcut; the real and imaginary parts must be taken explicitly.

Basic Plot Control

MATLAB enables you to customize graphs by setting line characteristics, axis limits, and axis tick marks. This section provides information on the available options. The next section discusses how to annotate your graph.

Colors, Line Styles, and Markers

The `plot` function accepts character-string arguments that specify various line styles, marker symbols, and colors for each vector plotted. In the general form,

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

color_linestyle_marker is a character string (delineated by single quotation marks) constructed from a color, a line style, and a marker type. For example:

```
plot(x, y, 'y: square')
```

plots a yellow dotted line and places square markers at each data point. If you specify a marker type, but not a line style, MATLAB draws only the marker.

You can also specify the size of the marker and, for markers that are closed shapes, you can specify separately the color of the edges and the face. See the `line` and `LineSpec` entries in the online MATLAB Function Reference for more information.

Available Line Styles and Markers

The following tables show the colors, line styles, and marker types available. You can specify the color as either the single letter abbreviation or the actual color name. For example, 'y' and 'yellow' both specify yellow.

Symbol	Color (RGB)	Symbol	Line Style
c	cyan (0 1 1)	—	solid line (default)
m	magenta (1 0 1)	- -	dashed line
y	yellow (1 1 0)	:	dotted line
r	red (1 0 0)	—.	dash-dot line
g	green (0 1 0)	none	no line
b	blue (0 0 1)		

Symbol	Color (RGB)	Symbol	Line Style
w	white (1 1 1)	—	—
k	black (0 0 0)	—	—

Marker Specifier	Description
+	plus sign
o	circle
*	asterisk
.	point
x	cross
square	square
di amond	diamond
^	upward pointing triangle
v	downward pointing triangle
>	right pointing triangle
<	left pointing triangle
pentagram	five-pointed star
hexagram	six-pointed star
none	no marker (default)

Axis Limits

MATLAB selects axis limits based on the range of the plotted data. However, you can specify the limits using the `axis` command. Call `axis` with the new limits defined as a four-element vector:

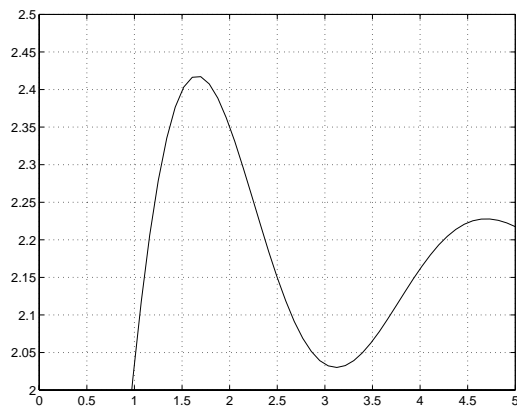
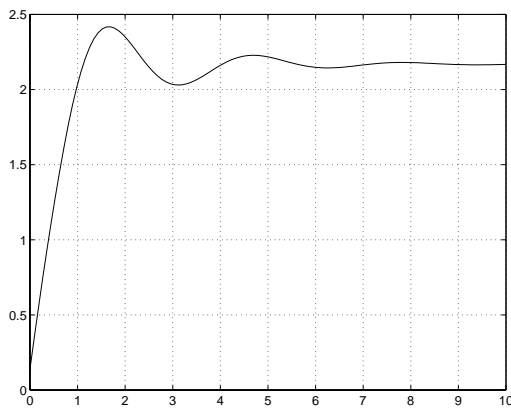
```
axis([xmi n, xmax, ymi n, ymax])
```

Note that the minimum values must be less than the maximum values.

Semiautomatic Limits

If you want MATLAB to autoscale one of the limits, but you want to specify the other, use the MATLAB variable `Inf` or `-Inf` for the autoscaled limit. For example, the graph on the left uses default scaling. The graph on the right sets the limits with the command:

```
axis([-Inf 5 2 2.5])
```



The `-Inf` causes MATLAB to autoscale the lower x -axis limit.

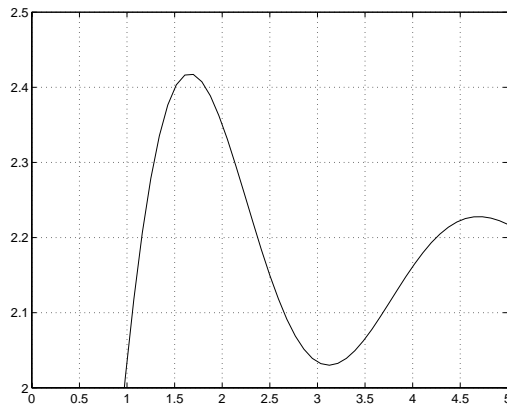
Axis Tick Marks

MATLAB selects the tick mark locations based on the data range to produce equally spaced ticks (for linear graphs). You can specify different tick marks by setting the Axes `XTick` and `YTick` properties. Define tick marks as a vector of increasing values. The values do not need to be equally spaced.

For example, setting the y -axis tick marks for the graph from the preceding example,

```
set(gca, 'ytick', [2 2.1 2.2 2.3 2.4 2.5])
```

produces a graph with only the specified ticks on the y -axis.



Note that if you specify tick mark values that are outside the axis limits, MATLAB does not display them (that is, specifying tick marks cannot cause axis limits to change).

Axes Aspect Ratio

By default, MATLAB displays graphs in a rectangular axes that has the same aspect ratio as the Figure window. This makes optimum use of space available for plotting. MATLAB provides control over the aspect ratio with the `axis` command.

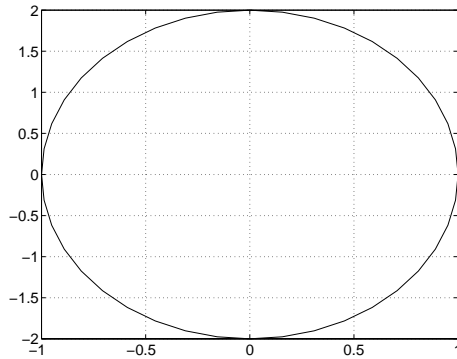
For example,

```
t = 0: pi/20: 2*pi;
plot(sin(t), 2*cos(t))
```

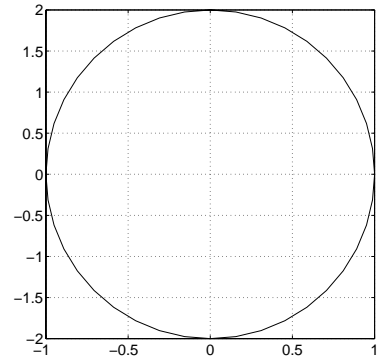
produces a graph with the default aspect ratio. The command

```
axis square
```

makes the x - and y -axes equal in length.



`axis normal`

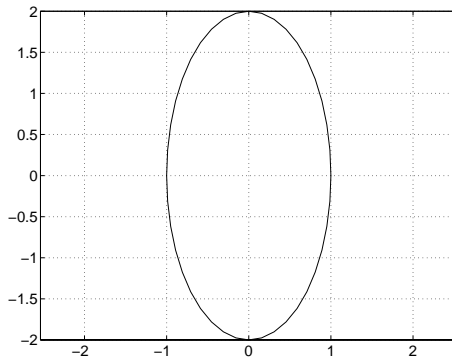


`axis square`

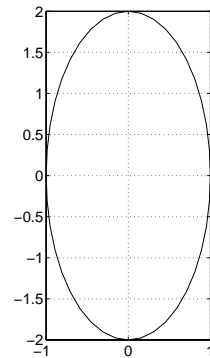
The square axes requires one data unit in x to equal two data units in y . If you want the x - and y -data units to be equal, use the command:

`axis equal`

This produces an axes that is rectangular in shape, but has equal scaling along each axis.



`axis equal`



`axis equal`
`axis tight`

If you want the axes shape to conform to the plotted data, use the `tight` option in conjunction with `equal` :

`axis equal tight`

Graph Annotation

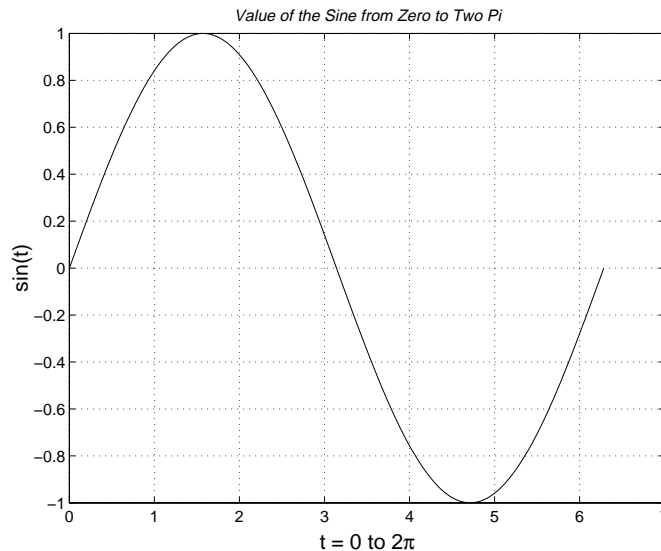
MATLAB provides commands to label each axis and place text at arbitrary locations on the graph. These commands include:

- `title` – adds a title to the graph
- `xlabel` – adds a label to the x -axis
- `ylabel` – adds a label to the y -axis
- `zlabel` – adds a label to the z -axis
- `legend` – adds a legend to an existing graph
- `text` – displays a text string at a specified location
- `gtext` – places text on the graph using the mouse

Labeling the Individual Axes

You can add x -, y -, and z -axis labels using the `xlabel`, `ylabel`, and `zlabel` commands. For example, these statements label the axes and add a title:

```
xlabel('t = 0 to 2\pi', 'FontSize', 16)
ylabel('sin(t)', 'FontSize', 16)
title('\it{Value of the Sine from Zero to Two Pi}')
```



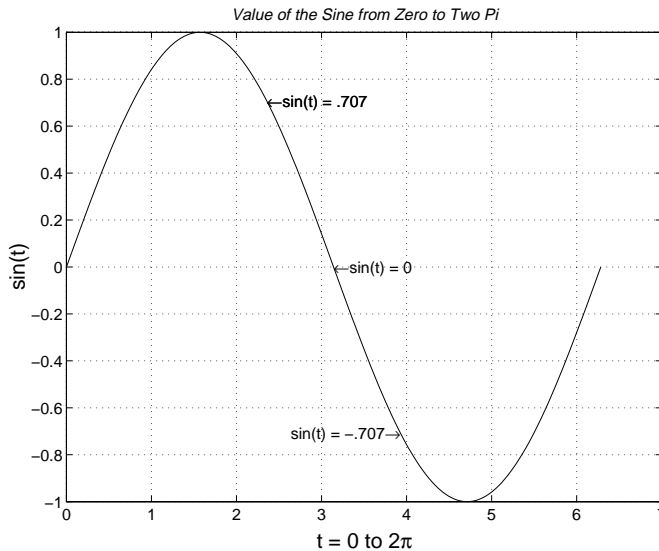
The labeling commands automatically position the text string appropriately. MATLAB interprets the characters immediately following the backslash “\” as TeX commands. These commands draw symbols such as Greek letters and arrows. See the `text` function in the online MATLAB Function Reference for a list of TeX character sequences.

Text Labels in Data Coordinates

You can place a text string at any location on the plot using the `text` function. This function positions the text string in the data space of the plot. For example, to label three data points on the previous graph, create three text strings:

```
text(3*pi/4, sin(3*pi/4), '\leftarrow sin(t) = .707')
text(pi, sin(pi), '\leftarrow sin(t) = 0')
text(5*pi/4, sin(5*pi/4), 'sin(t) = -.707\rightarrow', ...
     'Horizontal Alignment', 'right')
```

The `Horizontal Alignment` of the text string '`sin(t) = -.707 \rightarrow`' is set to `right` to place it on the left side of the point $[5\pi/4, \sin(5\pi/4)]$ on the graph:



Placing Text Interactively

You can place character strings on graphs interactively using the `gtext` function. This function accepts a string as an argument and waits while you select a location on the graph with the mouse. MATLAB then displays the text string at the indicated location.

`gtext` is a convenient way to annotate your graph if you do not want precise positioning of the text. It works only on 2-D graphs.

Building 3-D Graphs

Building a 3-D Graph.	3-2
Elementary 3-D Plotting Functions	3-3
Representing a Matrix as a Surface	3-5
Coloring Mesh and Surface Plots	3-12
Lighting	3-21
Controlling the Effects of Lighting.	3-24
Lighting Example	3-30
Viewpoint Control	3-32
Camera Properties.	3-35
View Projection Types	3-43
Aspect Ratio.	3-47
Properties That Affect Aspect Ratio	3-51

Building a 3-D Graph

The table below illustrates typical steps involved in producing 3-D scenes containing either data graphs or models of 3-D objects. Example applications include pseudocolor surfaces illustrating the values of functions over specific regions and objects drawn with polygons and colored with light sources to produce realism. Usually, you follow either step 4a or step 4b. Steps in gray indicate material covered in the *Building 2-D Graphs* chapter.

Step	Typical Code
1 Prepare your data	<code>Z = peaks(20);</code>
2 Select window and position plot region within window	<code>figure(1) subplot(2, 1, 2)</code>
3 Call 3-D graphing function	<code>h = surf(Z);</code>
4aSet colormap and shading algorithm	<code>colormap hot shading interp set(h, 'EdgeColor', 'k')</code>
4bAdd lighting	<code>light('Position', [-2, 2, 20]) lighting phong material([0.4, 0.6, 0.5, 30]) set(h, 'FaceColor', [0.7 0.7 0], ... 'BackFaceLighting', 'lit')</code>
5 Set viewpoint	<code>view([30, 25]) set(gca, 'CameraViewAngleMode', 'Manual')</code>
6 Set axis limits and tick marks	<code>axis([5 15 5 15 -8 8]) set(gca, 'ZTickLabel', 'Negative Positive')</code>
7 Set aspect ratio	<code>set(gca, 'PlotBoxAspectRatio', [2.5 2.5 1])</code>
8 Annotate the graph with axis labels, legend, and text	<code>xlabel('X Axis') ylabel('Y Axis') zlabel('Function Value') title('Peaks')</code>
9 Print graph	<code>set(gcf, 'PaperPositionMode', 'auto') print -dps2</code>

Elementary 3-D Plotting Functions

MATLAB provides a variety of functions for displaying 3-D data (i.e., data containing x -, y -, and z -coordinates). You can display the data as line plots (`plot3`) or rectangular grids (`mesh`, `surf`). See the *3-D Modeling* chapter information on how to display polygons (`patch`).

This chapter discusses 3-D line and surface plots as well as coloring and lighting.

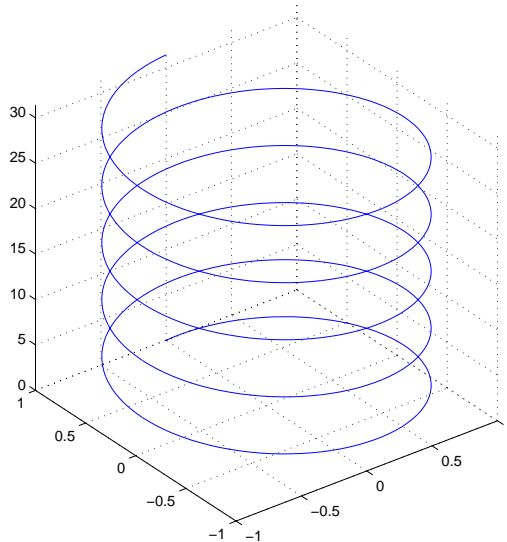
Line Plots in 3-D

The 3-D analog of the `plot` function is `plot3`. If x , y , and z are three vectors of the same length,

```
plot3(x, y, z)
```

generates a line in 3-D through the points whose coordinates are the elements of x , y , and z and then produces a 2-D projection of that line on the screen. For example these statements produce a helix:

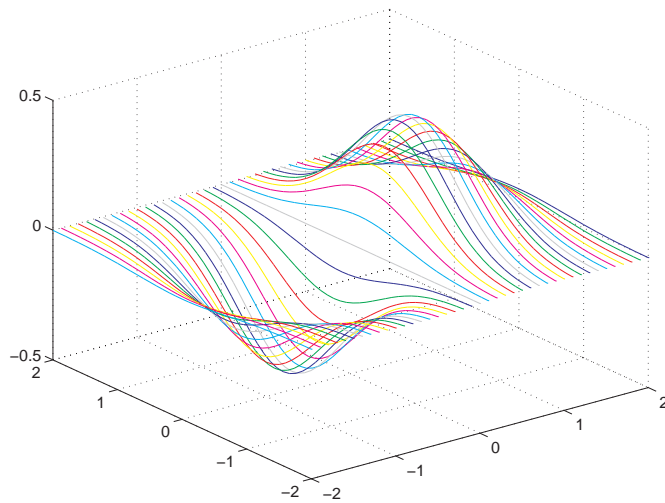
```
t = 0: pi/50: 10*pi;
plot3(sin(t), cos(t), t)
axis square; grid on
```



If the arguments to `plot3` are matrices of the same size, MATLAB plots lines obtained from the columns of `X`, `Y`, and `Z`. For example,

```
[X, Y] = meshgrid([-2: .1: 2]);  
Z = X.*exp(-X.^2-Y.^2);  
plot3(X, Y, Z)
```

Notice how MATLAB cycles through line colors:



Representing a Matrix as a Surface

MATLAB defines a surface by the z -coordinates of points above a rectangular grid in the x - y plane. The plot is formed by joining adjacent points with straight lines. Surface plots are useful for visualizing matrices that are too large to display in numerical form and for graphing functions of two variables.

MATLAB can create different forms of surface plots. Mesh plots are wire-frame surfaces that color only the lines connecting the defining points. Surface plots display both the connecting lines and the faces of the surface in color. This table lists the various forms:

Function	Used to Create
<code>mesh</code> , <code>surf</code>	Surface plot
<code>meshc</code> , <code>surfc</code>	Surface plot with contour plot beneath it
<code>meshz</code>	Surface plot with curtain plot (reference plane)
<code>pcolor</code>	Flat surface plot (value is proportional only to color)
<code>surf1</code>	Surface plot illuminated from specified direction
<code>surface</code>	Low-level function (on which high-level functions are based) for creating Surface graphics objects

Mesh and Surface Plots

The `mesh` and `surf` functions create 3-D surface plots of matrix data. If Z is a matrix for which the elements $Z(i,j)$ define the height of a surface over an underlying (i,j) grid, then

`mesh(Z)`

generates a colored, wire-frame view of the surface and displays it in a 3-D view. Similarly,

`surf(Z)`

generates a colored, faceted view of the surface and displays it in a 3-D view. Ordinarily, the facets are quadrilaterals, each of which is a constant color, out-

lined with black mesh lines, but the `shading` command allows you to eliminate the mesh lines (`shading flat`) or to select interpolated shading across the facet (`shading interp`).

Surface object properties provide additional control over the visual appearance of the surface. You can specify edge line styles, vertex markers, face coloring, lighting characteristics, and so on.

See the description of the `surface` function in the online MATLAB Function Reference for a complete list of properties. Also, see the sections on coloring and lighting later in this chapter for information on how MATLAB applies color to surfaces.

Visualizing Functions of Two Variables

The first step in displaying a function of two variables, $z = f(x, y)$, is to generate X and Y matrices consisting of repeated rows and columns, respectively, over the domain of the function. Then use these matrices to evaluate and graph the function.

The `meshgrid` function transforms the domain specified by two vectors, x and y , into matrices, X and Y . You then use these matrices to evaluate functions of two variables. The rows of X are copies of the vector x and the columns of Y are copies of the vector y .

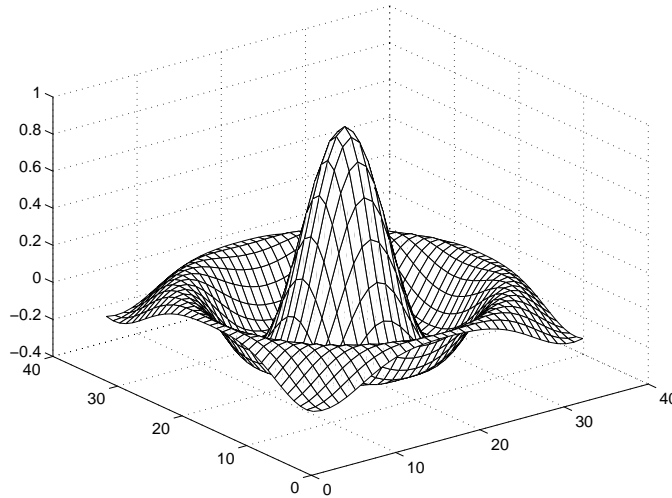
To illustrate the use of `meshgrid`, consider the $\sin(r)/r$ or *sinc* function. To evaluate this function between -8 and 8 in both x and y , you need pass only one vector argument to `meshgrid`, which is then used in both directions:

```
[X, Y] = meshgrid(-8:.5:8);
R = sqrt(X.^2 + Y.^2) + eps;
```

The matrix R contains the distance from the center of the matrix, which is the origin. Adding `eps` prevents the divide by zero (in the next step) that produces NaNs in the data.

Forming the *sinc* function and plotting *Z* with `mesh` results in the 3D surface:

```
Z = sin(R) ./ R;  
mesh(Z)
```



See the `surf` function in the online MATLAB Function Reference for more information on surface plots.

Surface Plots of Nonuniformly Sampled Data

The previous example uses `meshgrid` to create a grid of uniformly sampled data points at which to evaluate and graph the *sinc* function. MATLAB then constructs the surface plot by connecting neighboring matrix elements to form a mesh of quadrilaterals.

To produce a surface plot from nonuniformly sampled data, first use `griddata` to interpolate the values at uniformly spaced points, and then use `mesh` and `surf` in the usual way.

Example. This example evaluates the *sinc* function at random points within a specific range and then generates uniformly sampled data for display as a surface plot. The process involves these steps:

- Use `linspace` to generate evenly spaced values over the range of your unevenly sampled data.
- Use `meshgrid` to generate the plotting grid with the output of `linspace`.
- Use `griddata` to interpolate the irregularly sampled data to the regularly spaced grid returned by `meshgrid`.
- Use a plotting function to display the data.

First, generate unevenly sampled data within the range $[-8, 8]$ and use it to evaluate the function.

```
x = rand(100, 1) * 16 - 8;
y = rand(100, 1) * 16 - 8;
r = sqrt(x.^2 + y.^2) + eps;
z = sin(r) ./ r;
```

The `linspace` function provides a convenient way to create uniformly spaced data with the desired number of elements. The following statements produce vectors over the range of the random data with the same resolution as that generated by the `-8: .5: 8` statement in the previous `sinc` example:

```
xlin = linspace(min(x), max(x), 33);
ylin = linspace(min(y), max(y), 33);
```

Now use these points to generate a uniformly spaced grid:

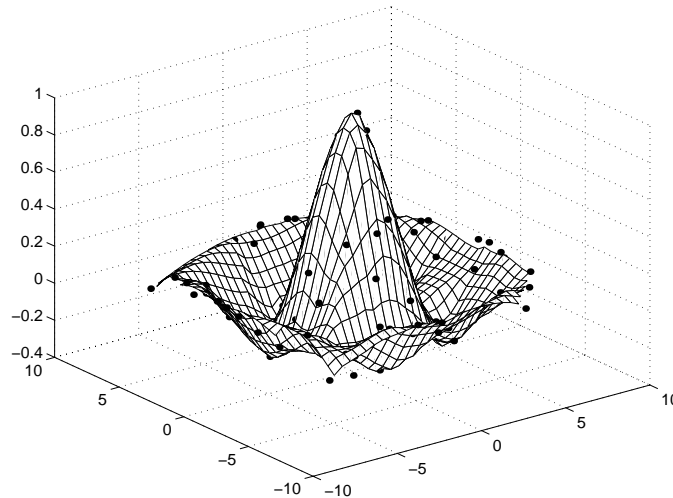
```
[X, Y] = meshgrid(xlin, ylin);
```

The key to this process is to use `griddata` to interpolate the values of the function at the uniformly spaced points, based on the values of the function at the original (random in this example) data points. This statement uses a triangle-based cubic interpolation to generate the new data:

```
Z = griddata(x, y, z, X, Y, 'cubic');
```


Plotting the interpolated and the nonuniform data produces:

```
mesh(X, Y, Z) %interpolated
hold on
plot3(x, y, z, 'o', 'MarkerSize', 15) %nonuniform
```



Parametric Surfaces

The functions that draw surfaces can take two additional vector or matrix arguments to describe surfaces with specific x and y data (see the previous mesh example). If Z is an m -by- n matrix, x is an n -vector, and y is an m -vector, then

```
mesh(x, y, Z, C)
```

describes a mesh surface with vertices having color $C(i, j)$ which are located at the points:

```
(x(j), y(i), Z(i, j))
```

where x corresponds to the columns of Z and y to its rows.

More generally, if X , Y , Z , and C are matrices of the same dimensions, then

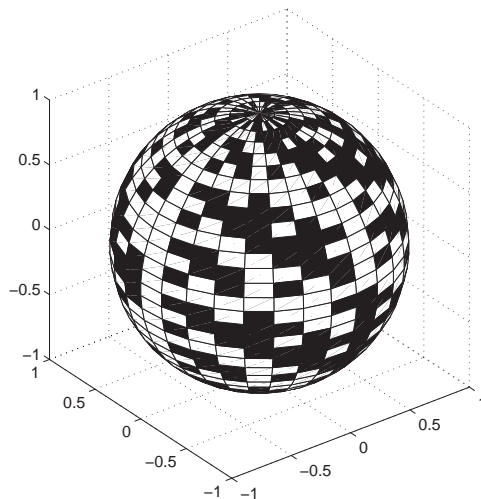
```
mesh(X, Y, Z, C)
```

describes a mesh surface with vertices having color $C(i, j)$ which are located at the points:

$$(X(i, j), Y(i, j), Z(i, j))$$

This example uses spherical coordinates to draw a sphere and color it with the pattern of pluses and minuses in a Hadamard matrix, an orthogonal matrix used in signal processing coding theory. The vectors θ and ϕ are in the range $-\pi \leq \theta \leq \pi$ and $-\pi/2 \leq \phi \leq \pi/2$. Since θ is a row vector and ϕ is a column vector, the multiplications that produce the matrices X , Y , and Z are vector outer products.

```
k = 5;
n = 2^k-1;
theta = pi * (-n:2:n) / n;
phi = (pi / 2) * (-n:2:n)' / n;
X = cos(phi) * cos(theta);
Y = cos(phi) * sin(theta);
Z = sin(phi) * ones(size(theta));
colormap([0 0 0; 1 1 1])
C = hadamard(2^k);
surf(X, Y, Z, C)
axis square
```

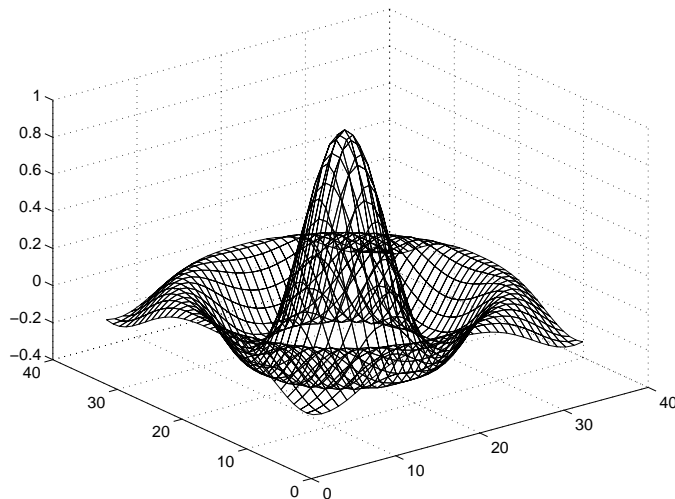


Hidden Line Removal

By default, MATLAB removes lines that are hidden from view in mesh plots, even though the faces of the plot are not colored. You can disable hidden line removal and allow the faces of a mesh plot to be transparent with the command:

```
hidden off
```

This is the surface plot with `hidden` set to `off`:



Coloring Mesh and Surface Plots

You can enhance the information content of surface plots by controlling the way MATLAB applies color to these plots. MATLAB can map particular data values to colors specified explicitly or can map the entire range of data to a pre-defined range of colors called a *colormap*.

There are basically two coloring techniques:

- **Indexed Color** – MATLAB colors the surface plot by assigning each data point an index into the Figure's colormap. The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated).
- **Truecolor** – MATLAB colors the surface plot using the explicitly specified colors (i.e., the RGB triplets). The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated). To be accurately rendered, truecolor requires computers with 24-bit displays; however, MATLAB simulates truecolor on indexed systems. See the shading function in the online MATLAB Function Reference for information on the types of shading.

The type of color data you specify (i.e., single values or RGB triplets) determines how MATLAB interprets it. When you create a surface plot, you can:

- Provide no explicit color data, in which case MATLAB generates colormap indices from the *z*-data.
- Specify an array of color data that is equal in size to the *z*-data and is used for indexed colors.
- Specify an *m*-by-*n*-by-3 array of color data that defines an RGB triplet for each element in the *m*-by-*n* *z*-data array and is used for truecolor.

Colormaps and Indexed Colors

Each MATLAB Figure window has a colormap associated with it. A colormap is simply a three-column matrix whose length is equal to the number of colors it defines. Each row of the matrix defines a particular color by specifying three values in the range 0 to 1. These values define the RGB components (i.e., the intensities of the red, green, and blue video components).

The `colormap` function, with no arguments, returns the current Figure's colormap.

For example, MATLAB's default colormap contains 64 colors and the 57th color is red:

```
cm = colormap;
cm(57, :)
ans =
    1    0    0
```

This table lists some representative RGB color definitions:

Red	Green	Blue	Color
0	0	0	black
1	1	1	white
1	0	0	red
0	1	0	green
0	0	1	blue
1	1	0	yellow
1	0	1	magenta
0	1	1	cyan
.5	.5	.5	gray
.5	0	0	dark red
1	.62	.40	copper
.49	1	.83	aquamarine

You can create colormaps with MATLAB's array operations or you can use any of several functions that generate useful maps, including `hsv`, `hot`, `cool`, `summer`, and `gray`. Each function has an optional parameter that specifies the number of rows in the resulting map.

For example,

```
hot(m)
```

creates an m -by-3 matrix whose rows specify the RGB intensities of a map that varies from black, through shades of red, orange, and yellow, to white.

If you do not specify the colormap length, MATLAB creates a colormap the same length as the current colormap. The default colormap is `jet(64)`.

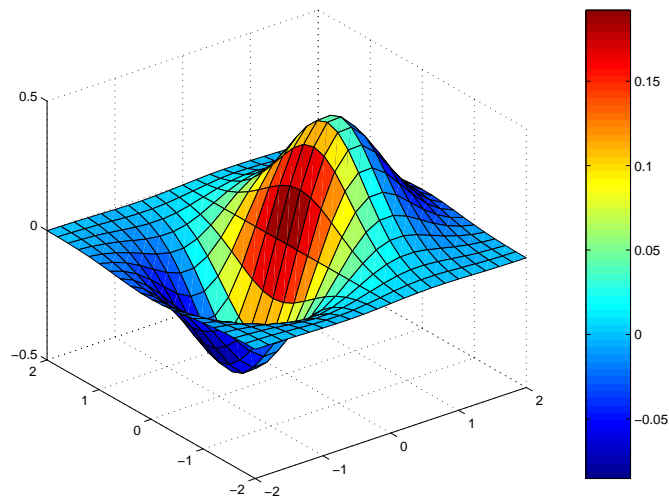
If you use long (> 64 colors) colormaps in each of several Figures windows, it may become necessary for the operating system to swap in different color lookup tables as the active focus is moved among the windows. See the *Figures* chapter for more information on how MATLAB manages color.

Displaying Colormaps

The `colorbar` function displays the current colormap, either vertically or horizontally, in the Figure window along with your graph. For example, the statements:

```
[x, y] = meshgrid([-2: .2: 2]);
Z = x.*exp(-x.^2-y.^2);
surf(x, y, Z, gradient(Z))
colorbar
```

produce a surface plot and a vertical strip of color corresponding to the colormap:



Direct and Scaled Indexed Colors

MATLAB can use two different methods to map indexed color data to the colormap – direct and scaled.

Direct Mapping. Direct mapping uses the color data directly as indices into the colormap. For example, a value of 1 points to the first color in the colormap, a value of 2 points to the second color, and so on. If the color data is not integer, MATLAB rounds it towards zero. Values greater than the number of colors in the colormap are set equal to the last color in the colormap (i.e., the number `length(colormap)`). Values less than 1 are set to 1.

Scaled Mapping. Scaled mapping uses a two-element vector `[cmin cmax]` (specified with the `caxis` command) to control the mapping of color data to the Figure colormap. `cmin` specifies the data value to map to the first color in the colormap and `cmax` specifies the data value to map to the last color in the colormap. Data values in between are linearly transformed from the second to the next to last color, using the expression:

$$\text{colormap_index} = \text{fix}(\text{color_data} - \text{cmin}) / (\text{cmax} - \text{cmin}) * \text{cm_length} + 1$$

`cm_length` is the length of the colormap.

By default, MATLAB sets `cmin` and `cmax` to span the range of the color data of all graphics objects within the axes. However, you can set these limits to any range of values. This enables you to display multiple axes within a single Figure window and use different portions of the Figure's colormap for each one. See the *Axes* chapter in the section “Calculating Color Limits,” for an example that uses color limits. Also see the `caxis` command in the online MATLAB Function Reference.

By default, MATLAB uses scaled mapping. To use direct mapping, you must turn off scaling when you create the plot. For example,

```
surf(Z, C, 'CDataMapping', 'direct')
```

See the surface function in the online MATLAB Function Reference for more information on specifying color data.

Specifying Indexed Colors

When creating a surface plot with a single matrix argument, `surf(Z)` for example, the argument `Z` specifies both the height and the color of the surface. MATLAB transforms `Z` to obtain indices into the current colormap.

With two matrix arguments, the statement

```
surf(Z, C)
```

independently specifies the color using the second argument. The next example illustrates how to use a color array to enhance the information displayed in a graph.

Example – Mapping Curvature to Color

The Laplacian of a surface plot is related to its curvature; it is positive for functions shaped like $i^2 + j^2$ and negative for functions shaped like $-(i^2 + j^2)$. The function `del2` computes the discrete Laplacian of any matrix. For example, use `del2` to determine the color for the data returned by `peaks`:

The `peaks` command returns a matrix of data based on a function of two variables.

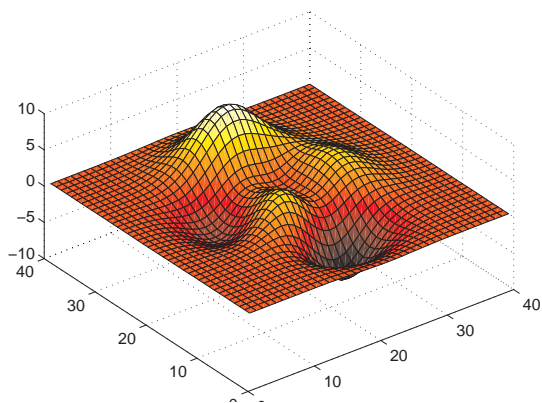
```
P = peaks(40);
C = del2(P);
surf(P, C)
colormap hot
```

Creating a color array by applying the Laplacian to the data is useful because it causes regions with similar curvature to be drawn in the same color. Compare this surface coloring with that produced by the statements:

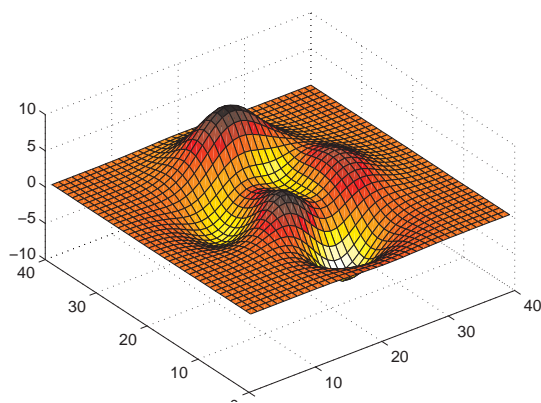
```
surf(P)
colormap hot
```

which use the same colormap, but maps regions with similar height to the same color:

`surf(P)`

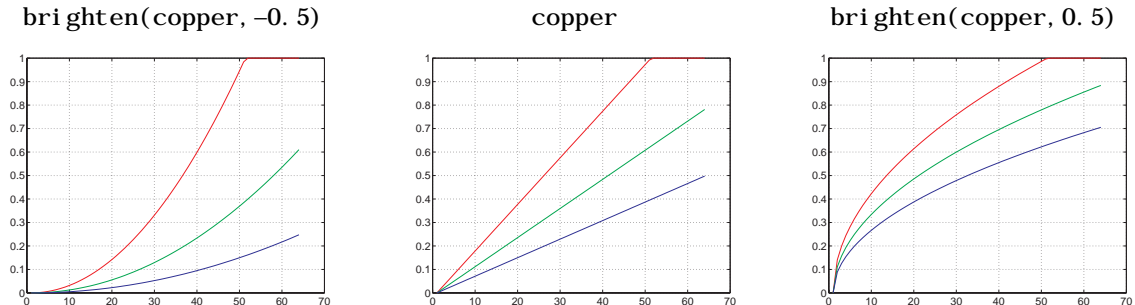


`surf(P, del2(P))`



Altering Colormaps

Since colormaps are matrices, you can manipulate them like other arrays. The `brighten` function takes advantage of this fact to increase or decrease the intensity of the colors. Plotting the values of the R, G, and B components of a colormap using `rgbplot` illustrates the effects of `brighten`:



NTSC Color Encoding

The brightness component of television signals uses the NTSC color encoding scheme:

$$\begin{aligned} b &= .30 * \text{red} + .59 * \text{green} + .11 * \text{blue} \\ &= \text{sum}(\text{diag}([.30 \ .59 \ .11]) * \text{map}')'; \end{aligned}$$

Using the nonlinear grayscale map,

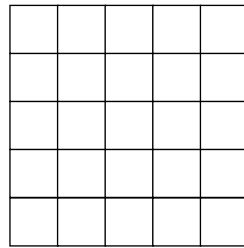
```
colormap([b b b])
```

effectively converts a color image to its NTSC black-and-white equivalent.

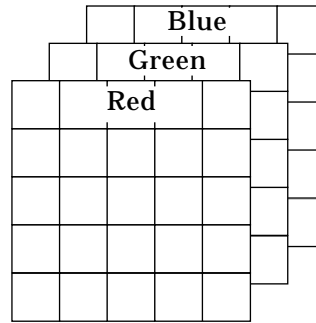
Truecolor

Computer systems with 24-bit displays are capable of displaying over 16 million (2^{24}) colors, as opposed to the 256 colors available on 8-bit displays. You can take advantage of this capability by defining color data directly as RGB values and eliminating the step of mapping numerical values to locations in a colormap.

Specify truecolor using an m -by- n -by-3 array, where the size of Z is m -by- n :



m -by- n matrix defining surface plot

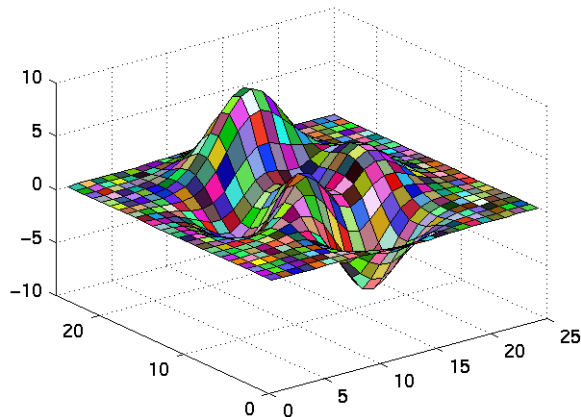


Corresponding m -by- n -by-3 matrix specifying truecolor for the surface plot

For example, the statements:

```
Z = peaks(25);
C(:, :, 1) = rand(25);
C(:, :, 2) = rand(25);
C(:, :, 3) = rand(25);
surf(Z, C)
```

create a plot of the peaks matrix with random coloring:



Rendering Method for Truecolor

MATLAB always uses the z-buffer render method when displaying truecolor. If the Figure `RendererMode` property is set to `auto`, MATLAB automatically switches the value of the `Renderer` property to `zbuffer` whenever you specify truecolor data.

If you explicitly set `Renderer` to `painters` (this sets `RendererMode` to `manual`) and attempt to define an `Image`, `Patch`, or `Surface` object using truecolor, MATLAB returns a warning and does not render the object.

See the `figure` function in the online MATLAB Function Reference for more information on the `Renderer` property and see the `image`, `patch`, and `surface` functions for information on defining truecolor for these objects.

Simulating Truecolor – Dithering

You can use truecolor on computers that do not have 24-bit displays. In this case, MATLAB uses a special colormap designed to produce results that are as close as possible, given the limited number of colors available. See the “Indexed Color Displays” section in the *Figure* chapter for more information.

Texture Mapping

Texture mapping is a technique for mapping a 2-D image onto a 3-D surface by transforming color data so that it conforms to the surface plot. It allows you to apply a “texture,” such as bumps or wood grain, to a surface without performing the geometric modeling necessary to create a surface with these features. The color data can also be any image, such as a scanned photograph.

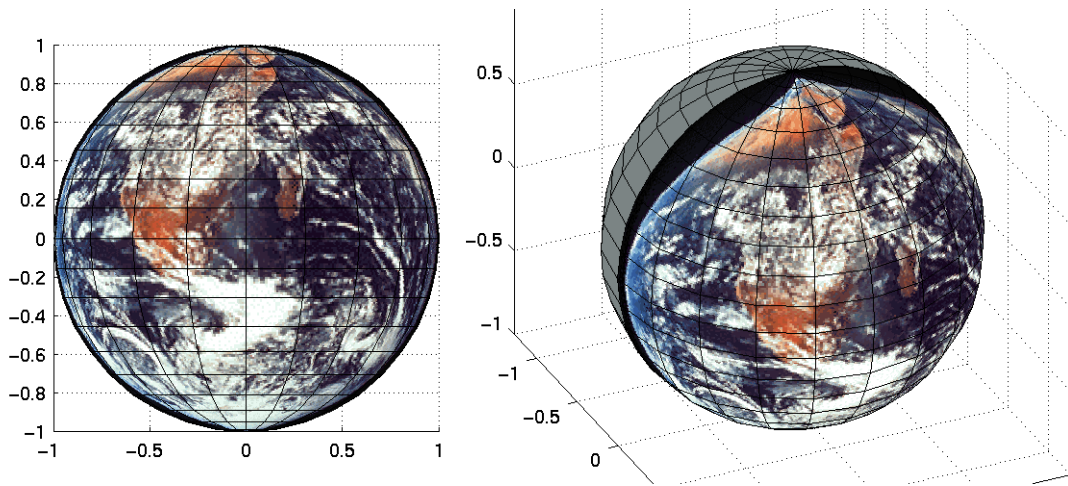
Texture mapping allows the dimensions of the color data array to be different from the data defining the surface plot. You can apply an image of arbitrary size to any surface. MATLAB interpolates texture color data so that it is mapped to the entire surface.

Example

This example creates a spherical surface using the `sphere` function and texture maps it with an image of the earth taken from space. Since the earth image is a view of earth from one side, this example maps the image to only one side of the sphere, padding the image data with 1s. In this case, the image data is a 257-by-250 matrix so it is padded equally on each side with two 257-by-125 matrices of 1s by concatenating the three matrices together.

To use texture mapping, set the `FaceColor` to `texturemap` and assign the image to the surface's `CData`:

```
load earth % load image data, X, and colormap, map
sphere; h = findobj('Type','surface');
hemi sphere = [ones(257,125),...
               X,...
               ones(257,125)];
set(h,'CData',flipud(hemi sphere),'FaceColor','texturemap')
colormap(map)
axis equal
view([90 0])
set(gca,'CameraViewAngleMode','manual')
view([65 30])
```



Lighting

Lighting is a technique for adding realism to a graphical scene. It does this by simulating the highlights and dark areas that occur on objects under natural lighting (e.g., the directional light that comes from the sun). To create lighting effects, MATLAB defines a graphics object called a `Light`. See the *Handle Graphics* chapter for more information on graphics objects.

Light Objects

You create a `Light` object using the `light` function. Three important `Light` object properties are:

- `Color` – the color of the light cast by the `Light` object
- `Style` – either infinitely far away (the default) or local
- `Position` – the direction (for infinite light sources) or the location (for local light sources)

The `Light` object's `Color` property determines the color of the directional light. The `Style` property determines whether the light source is a point source (`Style` set to `local`), which radiates from the specified position in all directions, or a light source placed at infinity (`Style` set to `infinite`), which shines from the direction of the specified position with parallel rays.

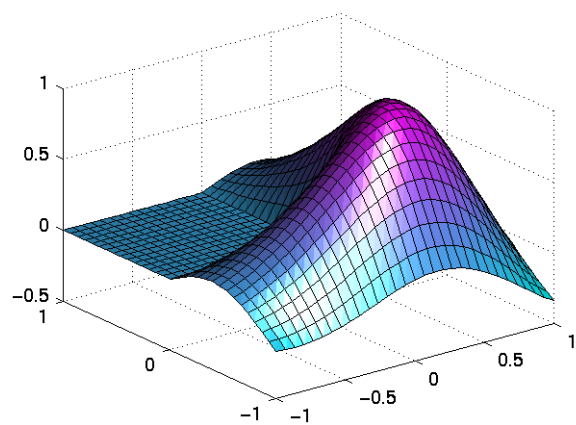
See the `light` function in the online MATLAB Function Reference for a complete list of properties.

Example – Simple Lighting

This example displays the membrane surface and illuminates it with a light source emanating from the direction defined by the position vector `[0 -2 1]`. This vector defines a direction from the axes origin passing through the point with the coordinates 0, -2, 1. The light shines from this direction towards the axes origin.

```
membrane
light('Position',[0 -2 1])
```

Creating a light activates a number of lighting-related properties controlling characteristics, such as the ambient light and reflectance properties of objects. It also switches to Z-buffer renderer if not already in that mode.



Properties that Affect Lighting

You cannot see light objects themselves, but you can see their effect on any patch and surface objects present in the axes containing the light. A number of functions create these objects, including surf, mesh, pcolor, fill, and fill3 as well as the surface and patch functions. You control lighting effects by setting various Axes, Light, Patch, and Surface object properties:

Property	Effect
AmbientLightColor	An Axes property that specifies the color of the background light in the scene, which has no direction and affects all objects uniformly. Ambient light effects occur only when there is a visible Light object in the Axes.
AmbientStrength	A Patch and Surface property that determines the intensity of the ambient component of the light reflected from the object.
DiffuseStrength	A Patch and Surface property that determines the intensity of the diffuse component of the light reflected from the object.
SpecularStrength	A Patch and Surface property that determines the intensity of the specular component of the light reflected from the object.
SpecularExponent	A Patch and Surface property that determines the size of the specular highlight.

Property	Effect
SpecularColorReflectance	A Patch and Surface property that determines the degree to which the specularly reflected light is colored by the object color or the light source color.
FaceLighting	A Patch and Surface property that determines the method used to calculate the effect of the light on the faces of the object. Choices are either no lighting, or flat, Gouraud, or Phong lighting algorithms.
EdgeLighting	A Patch and Surface property that determines the method used to calculate the effect of the light on the edges of the object. Choices are either no lighting, or flat, Gouraud, or Phong lighting algorithms.
BackFaceLighting	A Patch and Surface property that determines how faces are lit when their vertex normals point away from the camera. This property is useful for discriminating between the internal and external surfaces of an object.
FaceColor	A Patch and Surface property that specifies the color of the object faces.
EdgeColor	A Patch and Surface property that specifies the color of the object edges.
VertexNormals	A Patch and Surface property that contains normal vectors for each vertex of the object. MATLAB used vertex normal vectors to perform lighting calculations. While MATLAB automatically generates this data, you can also specify your own vertex normals.
Normal Mode	A Patch and Surface property that determines whether MATLAB recalculates vertex normals if you change object data (auto) or uses the current values of the VertexNormals property (manual). If you specify values for VertexNormals, MATLAB sets this property to manual.

For a description of all axes, surface, and patch object properties, see the axes, surface, and patch functions in the online MATLAB Function Reference.

Controlling the Effects of Lighting

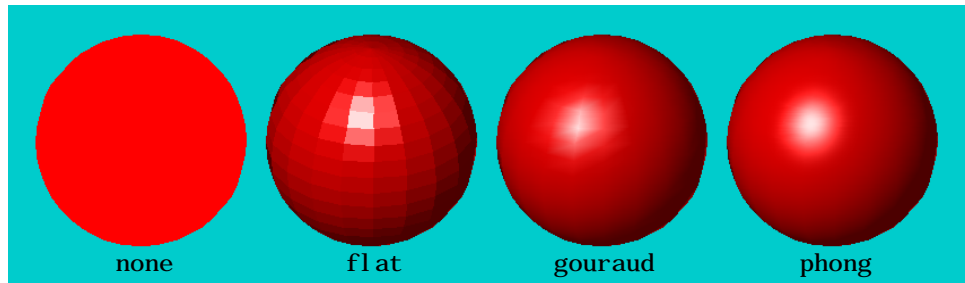
This section illustrates the visual effects of the various properties that affect lighting. All properties have default values that generally produce desirable results. However, you can achieve the specific effect you want by adjusting the values of these properties.

Face and Edge Lighting Methods

MATLAB supports three different algorithms for lighting calculations, selected by setting the `FaceLighting` and `EdgeLighting` properties of each `Surface` and `Patch` object in the scene. Each algorithm produces somewhat different results:

- Flat lighting produces uniform color across each of the faces of the object. Select this method to view faceted objects.
- Gouraud lighting calculates the colors at the vertices and then interpolates colors across the faces. Select this method to view curved surfaces.
- Phong lighting interpolates the vertex normals across each face and calculates the reflectance at each pixel. Select this choice to view curved surfaces. Phong lighting generally produces better results than Gouraud lighting, but takes longer to render.

This illustration shows how a red sphere looks using each of the lighting methods with one white light source.



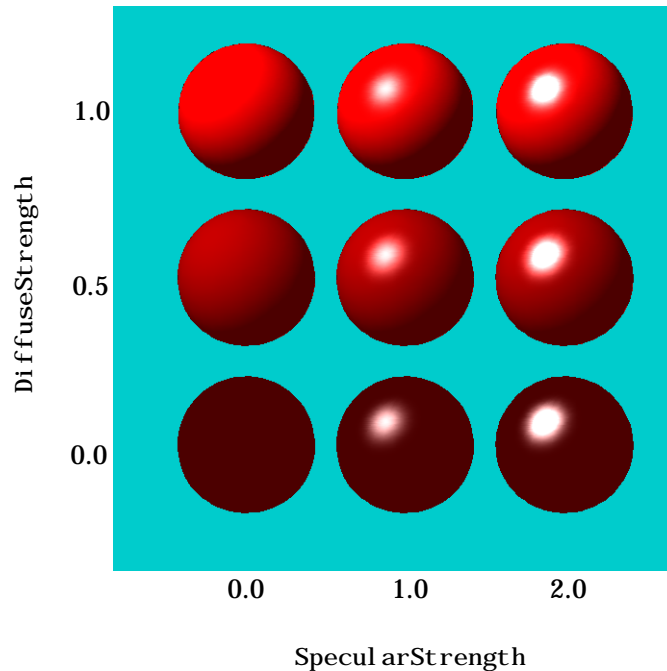
The `lighting` command (as opposed to the `light` function) provides a convenient way to set the lighting method. See the online MATLAB Function Reference for more information on this command.

Reflectance Characteristics of Graphics Objects

This section illustrates how Surface and Patch properties affect reflection of light. It is likely you will use these properties in combination to produce particular results. See the `material` command for a convenient way to produce certain effects.

Specular and Diffuse Reflection

You can control the amount of specular and diffuse reflection from the surface of an object by setting the `SpecularStrength` and `DiffuseStrength` properties. This picture illustrates various settings:

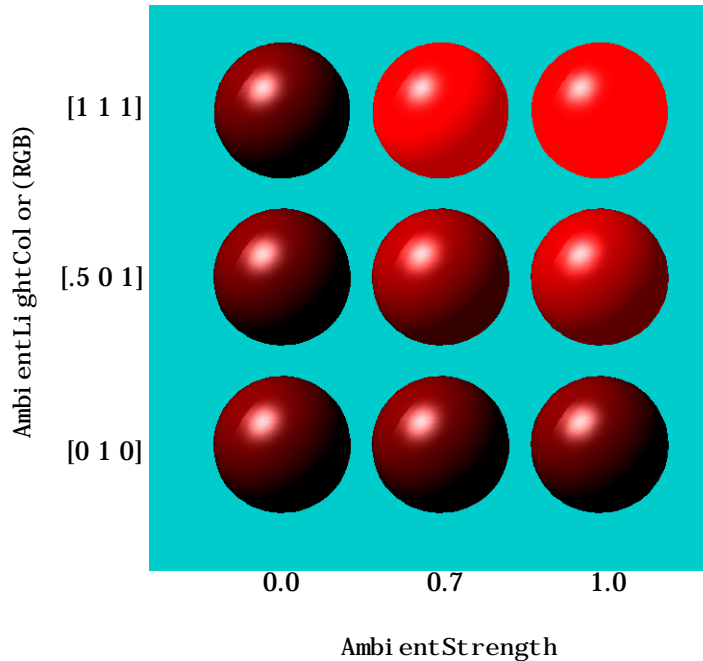


Ambient Light

Ambient light is a directionless light that shines uniformly on all objects in the scene. Ambient light is visible only when there are Light objects in the Axes. There are two properties that control ambient light – `AmbientLightColor` is an Axes property that sets the color, and `AmbientStrength` is a property of Surface

and Patch objects that determines the intensity of the ambient light on the particular object.

This illustration shows three different ambient light colors at various intensities. The sphere is red and there is a white Light object present.

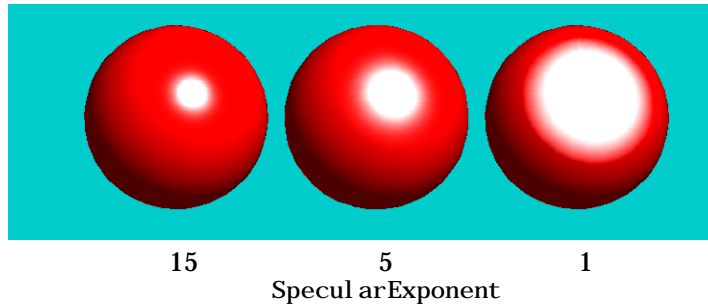


Note how the green [0 1 0] ambient light does not affect the scene. This is because there is no red component in green light. However, the color defined by the RGB values [.5 0 1] does have a red component so it contributes to the light on the sphere (but less than the white [1 1 1] ambient light).

Specular Exponent

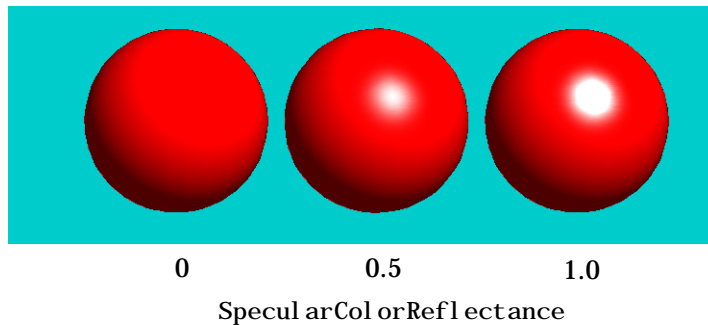
The size of the specular highlight spot depends on the value of the Surface or Patch object's SpecularExponent property. Typical values for this property range from 1 to 500, with normal objects having values in the range 5 to 20.

This illustration shows a red sphere illuminated by a white light with three different values for the `SpecularExponent` property:



Specular Color Reflectance

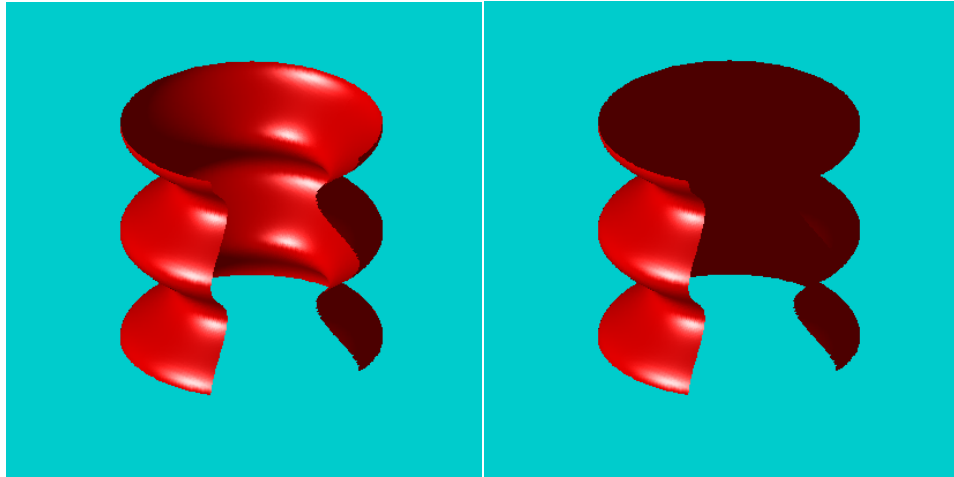
The color of the specularly reflected light can range from a combination of the color of the object and the color of the light source to the color of the light source only. The `Surface` or `Patch` `SpecularColorReflectance` property controls this color. This illustration shows a red sphere illuminated by a white light. The values of the `SpecularColorReflectance` property range from 0 (object and light color) to 1 (light color).



BackFaceLighting

Back face lighting is useful for showing the difference between internal and external faces. These pictures of cut-away cylindrical surfaces illustrate the

effects of back face lighting:



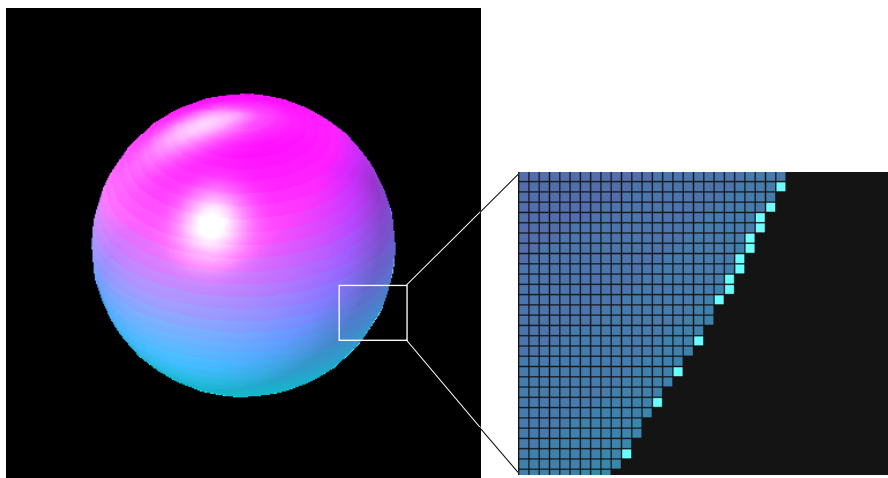
BackFaceLighting = reverselit

BackFaceLighting = unlit

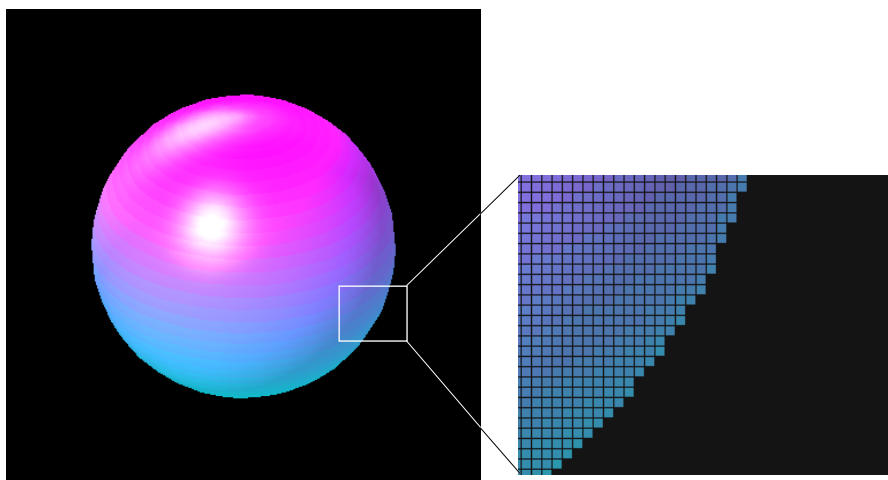
The default value for BackFaceLighting is reverselit. This setting reverses the direction of the vertex normals that face away from the camera, causing the interior surface to reflect light towards the camera. Setting BackFaceLighting to unlit disables lighting on faces with normals that point away from the camera.

You can also use BackFaceLighting to remove edge effects for closed objects. These effects occur when BackFaceLighting is set to reverselit and pixels along the edge of a closed object are lit as if their vertex normals faced the camera. This produces an improperly lit pixel because the pixel is visible, but is really facing away from the camera.

To illustrate this effect, the following picture shows a blowup of the edge of a lit sphere. Setting BackFaceLighting to lit prevents the improper lighting of pixels.



BackFaceLi ght i ng = reversel i t



BackFaceLi ght i ng = l i t

Lighting Example

This example creates a sphere and a cube to illustrate the effects of various properties on lighting. The variables `vert` and `fac` define the cube using the `patch` function:

```

vert =                                     fac =

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

sphere(36);
h = findobj('Type','surface');
set(h,'FaceLighting','phong',...
    'FaceColor','interp',...
    'EdgeColor',[.4 .4 .4],...
    'BackFaceLighting','lit')
hold on
patch('faces',fac,'vertices',vert,'FaceColor','y');
light('Position',[1 3 2]);
light('Position',[-3 -1 3]);
material shiny
axis vis3d off
hold off

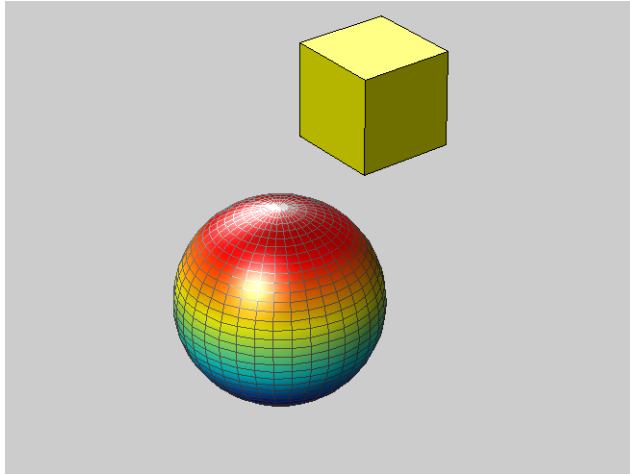
```

All faces of the cube have `FaceColor` set to yellow. The `sphere` function creates a spherical surface and the handle of this surface is obtained using `findobj` to search for the object whose `Type` property is `surface`. The `light` functions define two, white (the default color) Light objects located at infinity in the direction specified by the `Position` vectors. These vectors are defined in Axes coordinates $[x, y, z]$.

The `Patch` uses `flat FaceLighting` (the default) to enhance the visibility of each side. The `Surface` uses `phong FaceLighting` because it produces the smoothest interpolation of lighting effects. The `material shiny` command

affects the reflectance properties of both the cube and sphere (although its effects are noticeable only on the sphere because of the cube's flat shading).

Since the sphere is closed, the `BackFaceLighting` property is changed from its default setting, which reverses the direction of vertex normals that face away from the camera, to normal lighting, which removes undesirable edge effects:



Examining the code in the `lighting` and `material M`-files can help you understand how various properties affect lighting.

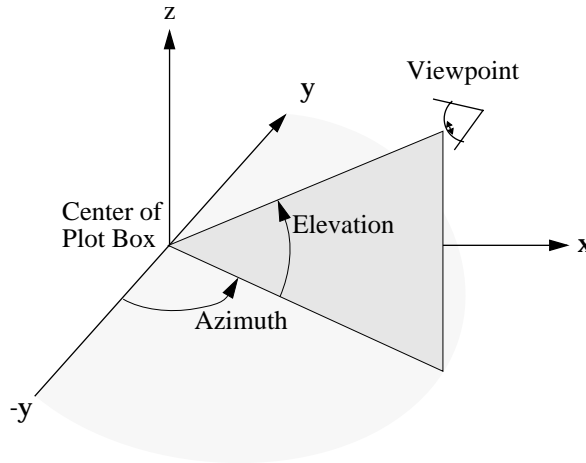
Viewpoint Control

MATLAB enables you to control the orientation of the graphics displayed in an axes. You can specify the viewpoint, view target, orientation, and extent of the view displayed in a Figure window. These viewing characteristics are controlled by a set of graphics properties. You can specify values for these properties directly or use the `view` command to select a view direction and rely on MATLAB's automatic property selection to define a reasonable view.

Setting the Viewpoint

The `view` command specifies the viewpoint by defining azimuth and elevation with respect to the axis origin. Azimuth is a polar angle in the x - y plane, with positive angles indicating counter-clockwise rotation of the viewpoint. Elevation is the angle above (positive angle) or below (negative angle) the x - y plane.

This diagram illustrates the coordinate system. The arrows indicate positive directions:

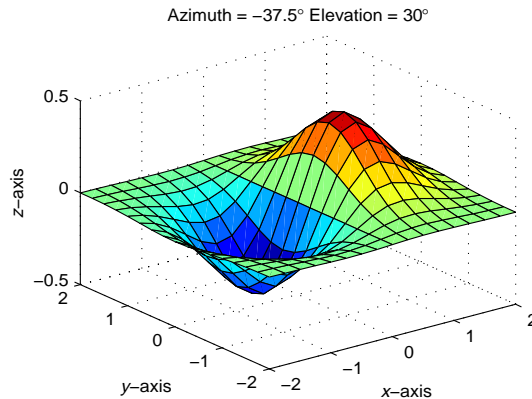


MATLAB automatically selects a viewpoint determined by whether the plot is 2-D or 3-D:

- For 2-D plots, the default is azimuth = 0° and elevation = 90° .
- For 3-D plots, the default is azimuth = -37.5° and elevation = 30° .

For example, these statements create a 3-D surface plot and display it in the default 3-D view:

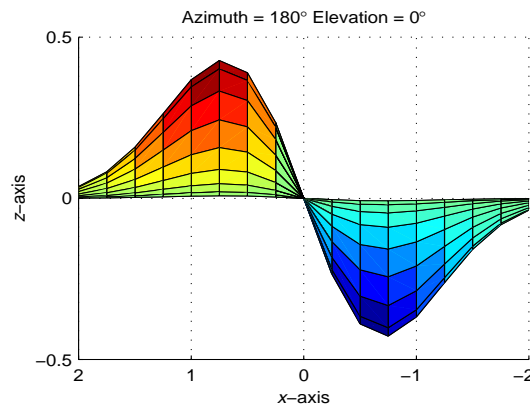
```
[X, Y] = meshgrid([-2: .25: 2]);
Z = X.*exp(-X.^2 -Y.^2);
surf(X, Y, Z)
```



The statement,

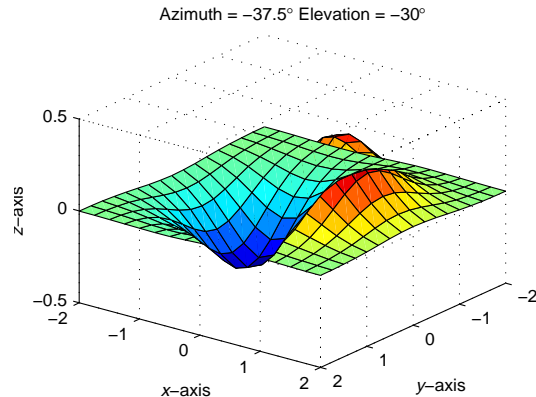
```
view([180 0])
```

sets the viewpoint so you are looking in the negative y -direction with your eye at the $z = 0$ elevation:



You can move the viewpoint to a location below the axis origin using a negative elevation:

```
view([-37.5 -30])
```



Limitations of Azimuth and Elevation

Specifying the viewpoint in terms of azimuth and elevation is conceptually simple, but it has limitations. It does not allow you to specify the actual position of the viewpoint, just its direction, and the z -axis is always pointing up. It does not allow you to zoom in and out on the scene or perform arbitrary rotations and translations. The Axes camera properties provide greater control than the simple adjustments allowed with azimuth and elevation.

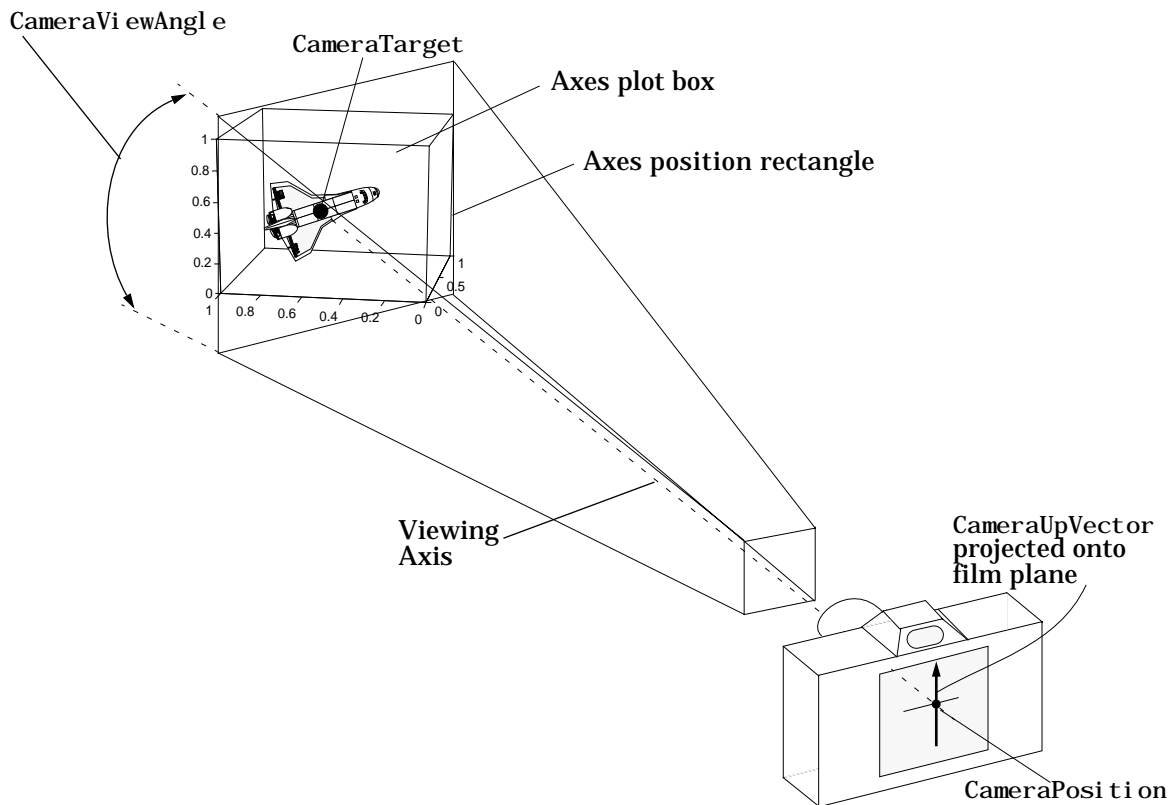
The next section discusses how to use camera properties to control the view.

Camera Properties

When you look at the graphics objects displayed in an axes, you are viewing a scene from a particular location in space having a particular orientation with regard to the viewpoint. MATLAB provides functionality, analogous to that of a camera with a zoom lens, that enables you to control many aspects of the view. This functionality is realized with the Axes camera properties:

Property	What It Is
CameraPosi ti on	Specifies the location of the viewpoint in axes units.
CameraPosi ti onMode	In automati c mode, MATLAB determines the position based on the scene. In manual mode, you specify the viewpoint location.
CameraTarget	Specifies the location in the axes that the camera points to. Together with the CameraPosi ti on, it defines the viewing axis.
CameraTargetMode	In automati c mode, MATLAB specifies the CameraTarget as the center of the axes plot box. In manual mode, you specify the location.
CameraUpVector	The rotation of the camera around the viewing axis is defined by a vector indicating the direction taken as up.
CameraUpVectorMode	In automati c mode, MATLAB orients the up vector along the positive y-axis for 2-D views and along the positive z-axis for 3-D views. In manual mode, you specify the direction.
CameraVi ewAngl e	Specifies the field of view of the “lens.” If you specify a value for CameraVi ewAngl e, MATLAB overrides stretch-to-fill behavior (see the “Aspect Ratio” section).
CameraVi ewAngl eMode	<p>In automati c mode, MATLAB adjusts the view angle to the smallest angle that captures the entire scene. In manual mode, you specify the angle.</p> <p>Setting CameraVi ewAngl eMode to manual overrides stretch-to-fill behavior (see the “Aspect Ratio” section).</p>
Proj ecti on	Selects either an orthographic or perspective projection.

This picture illustrates how the camera properties are defined using the camera metaphor:



See the axes function in the online MATLAB Function Reference for a more detailed description of each property.

Default Behavior

When all the camera mode properties are set to auto (the default), MATLAB automatically controls the view, selecting appropriate values based on the assumption that you want the scene to fill the position rectangle (which is defined by the width and height components of the Axes Position property).

By default, MATLAB:

- Sets the `CameraPosition` so the orientation of the scene is the standard MATLAB 2-D or 3-D view (see the `view` command)
- Sets the `CameraTarget` to the center of the plot box
- Sets the `CameraUpVector` so the *y*-direction is up for 2-D views and the *z*-direction is up for 3-D views
- Sets the `CameraViewAngle` to the minimum angle that makes the scene fill the position rectangle (the rectangle defined by the `AxesPosition` property)
- Uses orthographic Projection

This default behavior generally produces desirable results. However, you can change these properties to produce useful effects.

Moving In and Out on the Scene

You can move the camera anywhere in the 3-D space defined by the axes. The camera continues to point towards the target regardless of its position. When the camera moves, MATLAB varies the camera view angle to ensure the scene fills the position rectangle.

Moving Through a Scene

You can create a fly-by effect by moving the camera through the scene. To do this, continually change `CameraPosition` property, moving it towards the target. Since the camera is moving through space, it turns as it moves past the camera target. Override MATLAB's automatic resizing of the scene each time you move the camera by setting the `CameraViewAngleMode` to `manual`.

If you update the `CameraPosition` and the `CameraTarget`, the effect is to pass through the scene while continually facing the direction of movement.

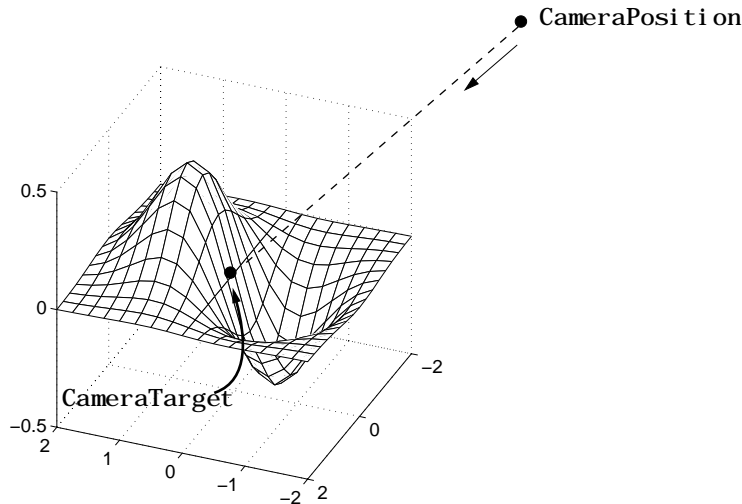
If the `Projection` is set to `perspective`, the amount of perspective distortion increases as the camera gets closer to the target and decreases as it gets farther away.

Example – Moving Towards or Away from the Target

To move the camera along the viewing axis, you need to calculate new coordinates for the `CameraPosition` property. This is accomplished by subtracting (to move closer to the target) or adding (to move away from the target) some fraction of the total distance between the camera position and the camera target.

The function `movecamera` calculates a new `CameraPosition` that moves in on the scene if the argument `dist` is positive and moves out if `dist` is negative:

```
function movecamera(dist) %dist in the range [-1 1]
set(gca, 'CameraViewAngleMode', 'manual')
newcp = cpos - dist * (cpos - ctarg);
set(gca, 'CameraPosition', newcp)
function out = cpos
out = get(gca, 'CameraPosition');
function out = ctarg
out = get(gca, 'CameraTarget');
```



Note that setting the `CameraViewAngleMode` to `manual` overrides MATLAB's stretch-to-fill behavior and may cause an abrupt change in the aspect ratio. See the “Aspect Ratio” section for more information on stretch-to-fill.

Making the Scene Larger or Smaller

Adjusting the `CameraViewAngle` property makes the view of the scene larger or smaller. Larger angles cause the view to encompass a larger area, thereby making the objects in the scene appear smaller. Similarly, smaller angles make the objects appear larger. Changing `CameraViewAngle` makes the scene larger or smaller without affecting the position of the camera. This is desirable if you want to zoom in without moving the viewpoint past objects that will then no

longer be in the scene (as could happen if you changed the camera position). Also, changing the `CameraViewAngle` does not affect the amount of perspective applied to the scene, as changing `CameraPosition` does when the `FigureProjection` property is set to `perspective`.

Revolving Around the Scene

You can use the `view` command to revolve the viewpoint about the z -axis by varying the azimuth, and about the azimuth by varying the elevation. This has the effect of moving the camera around the scene along the surface of a sphere whose radius is the length of the viewing axis. You could create the same effect by changing the `CameraPosition`, but doing so requires you to perform calculations that MATLAB performs for you when you call `view`.

For example, the function `orbit` moves the camera around the scene:

```
function orbit(deg)
[az el] = view;
rotvec = 0:deg/10:deg;
for i = 1:length(rotvec)
    view([az+rotvec(i) el])
    drawnow
end
```

Rotation without Resizing of Graphics Objects

When `CameraViewAngleMode` is `auto`, MATLAB calculates the `CameraViewAngle` so that the scene is as large as can fit in the axes position rectangle. This causes an apparent size change during rotation of the scene. To prevent resizing during rotation, you need to set the `CameraViewAngleMode` to `manual` (which happens automatically when you specify a value for the `CameraViewAngle` property). To do this in the `orbit` function, add the statement:

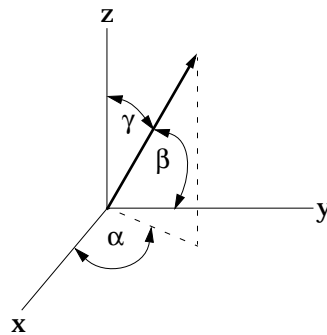
```
set(gca, 'CameraViewAngleMode', 'manual')
```

Rotation About the Viewing Axis

You can change the orientation of the scene by specifying the direction defined as *up*. By default, MATLAB defines *up* as the y -axis in 2-D views (the `CameraUpVector` is `[0 1 0]`) and the z -axis for 3-D views (the `CameraUpVector` is `[0 0 1]`). However, you can specify *up* as any arbitrary direction.

The vector defined by the `CameraUpVector` property forms one axis of the camera's coordinate system. Internally, MATLAB determines the actual orientation of the camera up vector by projecting the specified vector onto the plane that is normal to the camera direction (i.e., the viewing axis). This simplifies the specification of the `CameraUpVector` property since it need not lie in this plane.

In many cases, you may find it convenient to visualize the desired up vector in terms of angles with respect to the Axes x -, y -, and z -axes. You can then use *direction cosines* to convert from angles to vector components. For a unit vector, the expression simplifies to:



where the angles α , β , and γ are specified in degrees:

$$\begin{aligned} XComponent &= \cos(\alpha \times (\pi / 180)); \\ YComponent &= \cos(\beta \times (\pi / 180)); \\ ZComponent &= \cos(\gamma \times (\pi / 180)); \end{aligned}$$

(Consult a mathematics book on vector analysis for a more detailed explanation of direction cosines.)

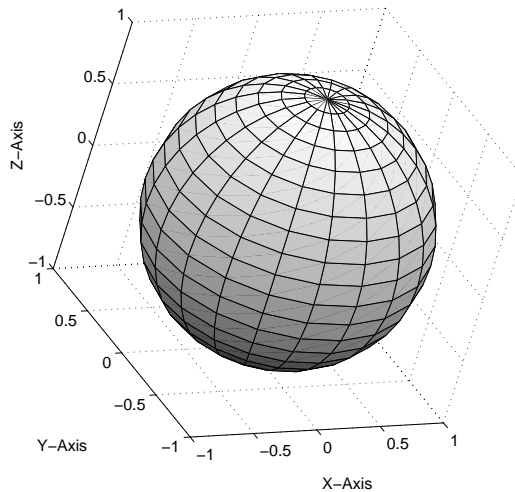
Example – Calculating a Camera Up Vector. To specify an up vector that makes an angle of 30° with the z -axis and lies in the y - z plane, use the expression:

$$\text{upvector} = [\cos(90 * (\pi / 180)) \cos(60 * (\pi / 180)) \cos(30 * (\pi / 180))];$$

and then set the `CameraUpVector` property:

$$\text{set}(\text{gca}, 'CameraUpVector', \text{upvector})$$

Drawing a sphere with this orientation produces:



Translating the Viewpoint

To translate the viewpoint, you need to move both the camera position and the camera target in the same direction. For a 2-D view, this is a fairly simple operation since the viewing axis lies along the z -axis. In this case, values of the `CameraPosition` and the `CameraTarget` properties differ only in z -coordinates.

Example – 2-D Translation

Suppose you want to zoom in on a 2-D scene and move around to examine particular details. You can use the `ginput` function to obtain new locations for the position and target. This example creates an M-file, `pan2D`, that calls `ginput` to obtain the coordinates of a point in the x - y plane and then updates the `CameraPosition` and the `CameraTarget` properties to the selected location. The location along the z -axis is held constant.

```

function pan2D
    cp = get(gca, 'CameraPosition');
    ct = get(gca, 'CameraTarget');
    cva = get(gca, 'CameraViewAngle');
    set(gca, 'CameraViewAngle', cva/2.5)
    while SelectionType is normal, get input
        [a, b] = ginput;
        set(gca, 'CameraPosition', [a, b, cp(3)], ...
            'CameraTarget', [a, b, ct(3)])
    end

Subfunction to monitor selection type
function out = seltype
    st = get(gcf, 'SelectionType');
    if strcmp(st, 'normal')
        out = 1;
    else
        out = 0;
    end
end

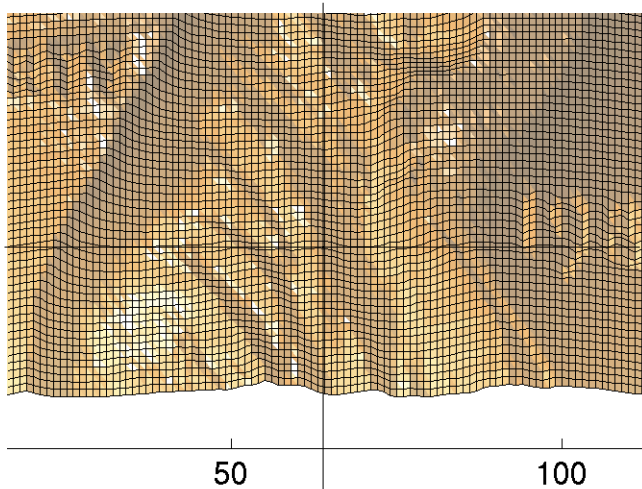
```

Create a graph to scan, for example:

```

load penny
surface(P)
axis ij
pan2D

```

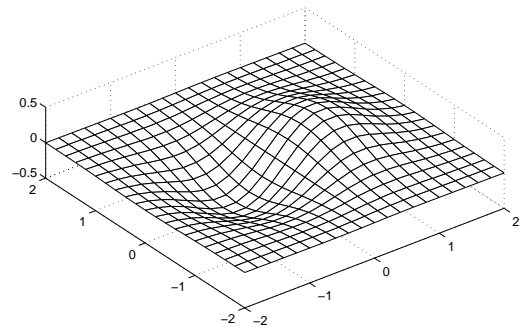
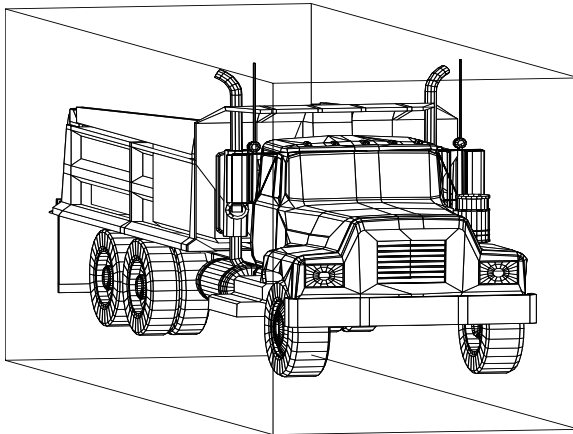


View Projection Types

MATLAB supports both orthographic and perspective projection types for displaying 3-D graphics. The one you select depends on the type of graphics you are displaying:

- **orthographic** projects the viewing volume as a rectangular parallelepiped (i.e., a box whose opposite sides are parallel). Relative distance from the camera does not affect the size of objects. This projection type is useful when it is important to maintain the actual size of objects and the angles between objects.
- **perspective** projects the viewing volume as the frustum of a pyramid (a pyramid whose apex has been cut off parallel to the base). Distance causes foreshortening; objects further from the camera appear smaller. This projection type is useful when you want to display realistic views of real objects.

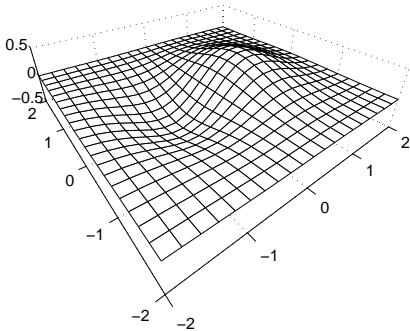
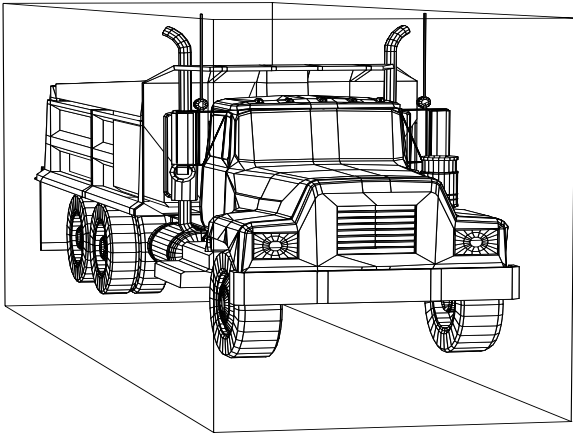
By default, MATLAB displays objects using orthographic projection. These pictures show a drawing of a dump truck (created with `patch`) and a surface plot of a mathematical function, both using orthographic projection:



If you measure the width of the front and rear faces of the box enclosing the dump truck, you'll see they are the same size. This picture looks unnatural because it lacks the apparent perspective you see when looking at real objects with depth. On the other hand, the surface plot accurately indicates the values

of the function within rectangular space.

Now look at the same graphics objects with perspective added. The dump truck looks more natural because portions of the truck that are farther from the viewer appear smaller. This projection mimics the way human vision works. The surface plot, on the other hand, looks distorted:

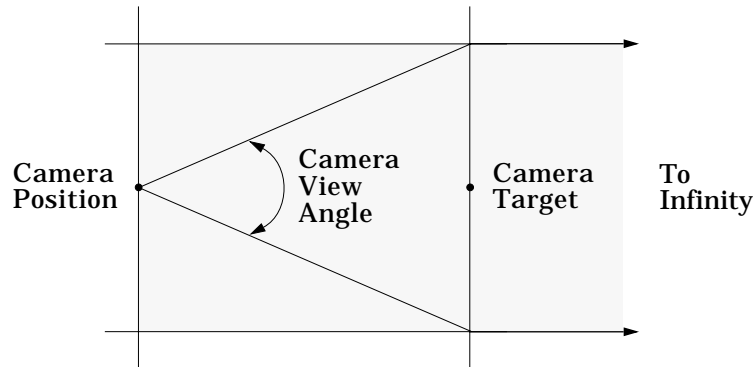


Projection Types and Camera Location

By default, MATLAB adjusts the CameraPosition, CameraTarget, and CameraViewAngle properties to point the camera at the center of the scene and to include all graphics objects in the axes. If you position the camera so that there are graphics objects behind the camera, the scene displayed can be affected by both the Axes Projection property and the Figure Renderer property. The following table summarizes the interactions between projection type and rendering method:

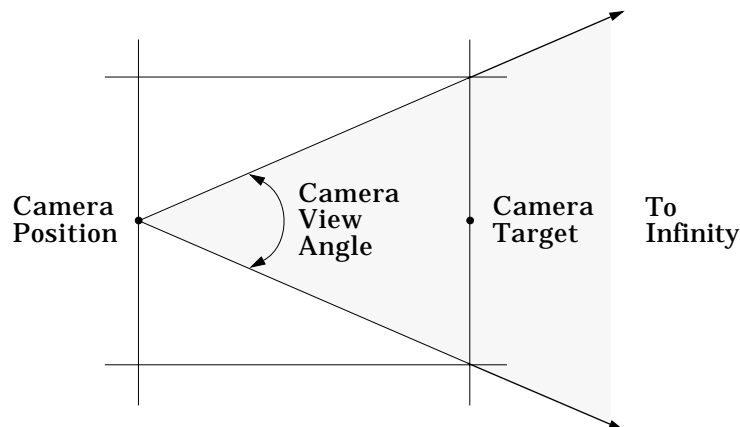
	Orthographic	Perspective
Z-buffer	CameraViewAngle determines extent of scene at CameraTarget	CameraViewAngle determines extent of scene from CameraPosition to infinity
Painters	All objects display regardless of CameraPosition	Not recommended if graphics objects are behind the CameraPosition

This diagram illustrates what you see (gray area) when using orthographic projection and Z-buffer. Anything in front of the camera is visible:



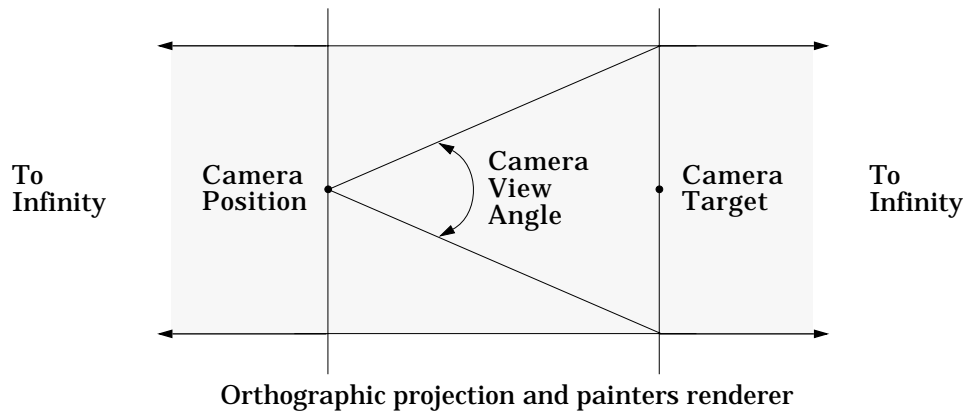
Orthographic projection and Z-buffer renderer

In perspective projection, you see only what is visible in the cone of the camera view angle:



Perspective projection and Z-buffer renderer

Painters rendering method is less suited to moving the camera in 3-D space because MATLAB does not clip along the viewing axis. Orthographic projection in painters method results in all objects contained in the scene being visible regardless of the camera position:



Printing 3-D Scenes

The same effects described in the previous section occur in hardcopy output. However, because of the differences in the process of rendering to the screen and to a printing format, MATLAB may render in Z-buffer and generate printed output in painters. You may need to explicitly specify Z-buffer printing to obtain the results displayed on the screen (use the `-zbuffer` option with the `print` command).

See the *Printing* chapter for information on printing, the *Figures* chapter for information on rendering methods, and the `axes`, `figure`, and `print` function descriptions in the online MATLAB Function Reference for information on graphics properties.

Aspect Ratio

Axes shape graphics objects by setting the scaling and limits of each axis. When you create a graph, MATLAB automatically determines axis scaling based on the values of the plotted data and then draws the axes to fit the space available for display. The definition of axes characteristics is controlled by Axes graphics object properties. You can specify values for these properties to optimize each graph.

This section discusses MATLAB's default behavior as well as techniques for customizing graphs.

Stretch-to-fill

By default, the size of the axes MATLAB creates for plotting is normalized to the size of the Figure window (but is slightly smaller to allow for borders). If you resize the Figure, the size (and aspect ratio) of the axis changes proportionally. This enables the axes to always fill the available space in the window. MATLAB also sets the x -, y -, and z -axis limits to provide the greatest resolution in each direction, again optimizing the use of available space.

This stretch-to-fill behavior is generally desirable; however, you may want to control this process to produce specific results. For example, images need to be displayed in correct proportions regardless of the aspect ratio (the ratio of width to height) of the Figure window, or you may want graphs always to be a particular size on a printed page.

axis Command Options

The `axis` command enables you to adjust the scaling and aspect ratio of graphs. See the `axis` command in the online MATLAB Function Reference for a complete description of all `axis` options.

Axis Scaling

By default, MATLAB finds the maxima and minima of the plotted data and chooses appropriate axes ranges. You can override the defaults by setting axis limits:

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

You can control how MATLAB scales the axes with predefined `axis` options:

- `axis auto` returns the axis scaling to its default, automatic mode. `v = axis` saves the scaling of the axes of the current plot in vector `v`. For subsequent graphics commands to have these same axis limits, follow them with `axis(v)`.
- `axis manual` freezes the scaling at the current limits. If you then set `hold on`, subsequent plots use the current limits. Specifying values for axis limits also sets axis scaling to manual.
- `axis tight` sets the axis limits to the range of the data.
- `axis ij` places MATLAB into its “matrix” axes mode. The coordinate system origin is at the upper-left corner. The *i*-axis is vertical and is numbered from top to bottom. The *j*-axis is horizontal and is numbered from left to right.
- `axis xy` places MATLAB into its default Cartesian axes mode. The coordinate system origin is at the lower-left corner. The *x*-axis is horizontal and is numbered from left to right. The *y*-axis is vertical and is numbered from bottom to top.

Aspect Ratio

Normally MATLAB stretches the axes to fill the window. In many cases, it is more useful to specify the aspect ratio of the axes based on a particular characteristic such as the relative length or scaling of each axis. The `axis` command provides a number of useful options for adjusting the aspect ratio.

- `axis equal` changes the current axes scaling so that equal tick mark increments on the *x*-, *y*-, and *z*-axis are equal in length. This makes the surface displayed by `sphere` look like a sphere instead of an ellipsoid. `axis equal` overrides stretch-to-fill behavior.
- `axis square` makes each axis the same length and overrides stretch-to-fill behavior.
- `axis vis3d` freezes aspect ratio properties to enable rotation of 3-D objects and overrides stretch-to-fill. Use this option after other `axis` options to keep settings from changing while you rotate the scene.
- `axis image` makes the aspect ratio of the axes the same as the image.
- `axis auto` returns the *x*-, *y*-, and *z*-axis limits to automatic selection mode.
- `axis normal` restores the current axis box to full size and removes any restrictions on the scaling of the units. It undoes the effects of `axis square`. Used in conjunction with `axis auto`, it undoes the effects of `axis equal`.

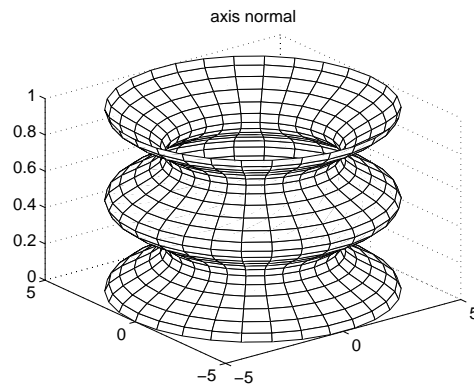
The `axis` command works by manipulating Axes graphics object properties. See the `axis` function in the online MATLAB Function Reference for a description of these properties. See the `axes` function for a description of all axes properties.

Example – axis Options

The following three pictures illustrate the effects of three `axis` options on a cylindrical surface created with the statements:

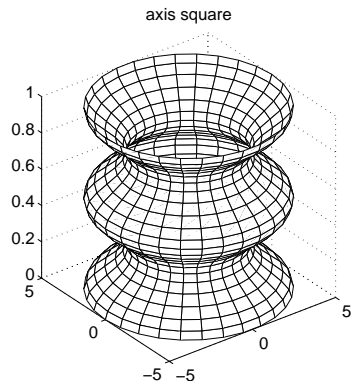
```
t = 0:pi/6:4*pi;
[x, y, z] = cylinder(4*cos(t), 30);
mesh(x, y, z)
```

`axis normal` is the default behavior. MATLAB automatically sets the axis limits to span the data range along each axis and stretches the plot to fit the Figure window.

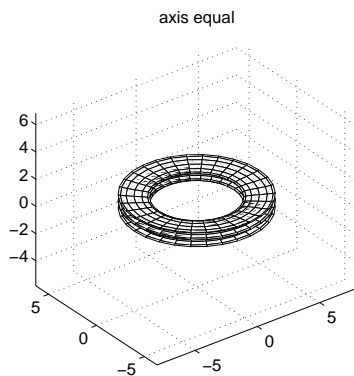


`axis square` creates an axis that is square regardless of the shape of the Figure window. The cylindrical surface is no longer distorted because it is not warped to fit the window. However, the size of one data unit is not

equal along all axes (the z -axis spans only one unit while the x - and y -axes span 10 units each).



axis equal makes the length of one data unit equal along each axis while maintaining a nearly square plot box. It also prevents warping of the axis to fill the window's shape.



Properties That Affect Aspect Ratio

The `axis` command works by setting various Axes object properties. You can set these properties directly to achieve precisely the effect you want. These properties include:

Property	What It Does
<code>DataAspectRatio</code>	Sets the relative scaling of the individual axis data values. Set <code>DataAspectRatio</code> to <code>[1 1 1]</code> to display real-world objects in correct proportions. Specifying a value for <code>DataAspectRatio</code> overrides stretch-to-fill behavior.
<code>DataAspectRatioMode</code>	In <code>auto</code> , MATLAB selects axis scales that provide the highest resolution in the space available.
<code>PlotBoxAspectRatio</code>	Sets the proportions of the axes plot box (Set <code>box</code> to <code>on</code> to see the box). Specifying a value for <code>PlotBoxAspectRatio</code> overrides stretch-to-fill behavior.
<code>PlotBoxAspectRatioMode</code>	In <code>auto</code> , MATLAB sets the <code>PlotBoxAspectRatio</code> to <code>[1 1 1]</code> unless you explicitly set the <code>DataAspectRatio</code> and/or the axis limits.
<code>Position</code>	Defines the location and size of the axes with a four-element vector: <code>[left offset, bottom offset, width, height]</code> .
<code>XLim</code> , <code>YLim</code> , <code>ZLim</code>	The minimum and maximum limits of the respective axes.
<code>XLimMode</code> , <code>YLimMode</code> , <code>ZLimMode</code>	In <code>auto</code> , MATLAB selects the axis limits.

By default, MATLAB automatically determines values for all of these properties (i.e., all the modes are `auto`) and then applies stretch-to-fill. You can override any property's automatic operation by specifying a value for the property or setting its mode to `manual`. The value you select for a particular property depends primarily on what type of data you want to display.

Much of the data visualized with MATLAB is either:

- Numerical data displayed as line or mesh plots
- Representations of real-world objects (e.g., a dump truck or a section of the earth's topography)

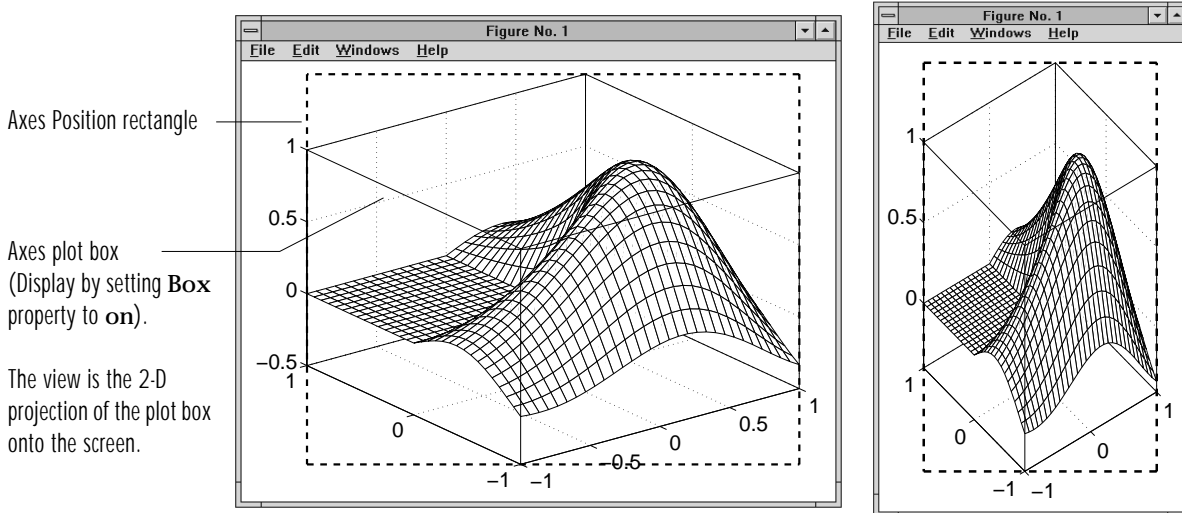
In the first case, it is generally desirable to select axis limits that provide good resolution in each axis direction and to fill the available space. Real-world objects, on the other hand, need to be represented accurately in proportion, regardless of the angle of view.

Default Behavior

There are two key elements to MATLAB's default behavior – normalizing the axes size to the window size and stretch-to-fill.

See the *Axes* chapter for a discussion of the *Axes Position* property.

The *Axes Position* property specifies the location and dimensions of the axes. The third and fourth elements of the *Position* vector (*width* and *height*) define a rectangle in which MATLAB draws the axes (indicated by the dotted line in the following pictures). MATLAB stretches the axes to fill this rectangle. The default value for the *Units* property is *normalized* to the parent Figure dimensions. This means the shape of the Figure window determines the shape of the position rectangle. As you change the size of the window, MATLAB reshapes the position rectangle to fit it:

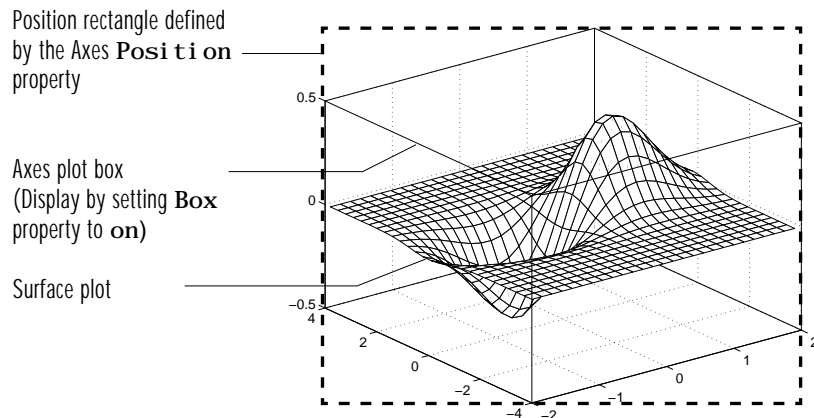


As you can see, reshaping the axes to fit into the Figure window can change the aspect ratio of the graph. MATLAB applies stretch-to-fill so the axes fill the position rectangle, and in the process may distort the shape. This is generally desirable for graphs of numeric data, but not for displaying realistic objects.

Example – MATLAB Defaults

MATLAB surface plots are well suited for visualizing mathematical functions of two variables. For example, to display a mesh plot of the function, $z = xe^{(-x^2 - y^2)}$ evaluated over the range $-2 \leq x \leq 2$, $-4 \leq y \leq 4$, use the statements:

```
[X, Y] = meshgrid([-2: .15: 2], [-4: .3: 4]);
Z = X.*exp(-X.^2 - Y.^2);
mesh(X, Y, Z)
```



MATLAB's default property values are designed to:

- Select axis limits to span the range of the data (XLimMode, YLimMode, and ZLimMode are set to auto).
- Provide the highest resolution in the available space by setting the scale of each axis independently (DataAspectRatioMode and the PlotBoxAspectRatioMode are set to auto).
- Draw axes that fit the position rectangle by adjusting the CameraViewAngle and then stretch-to-fill the axes if necessary.

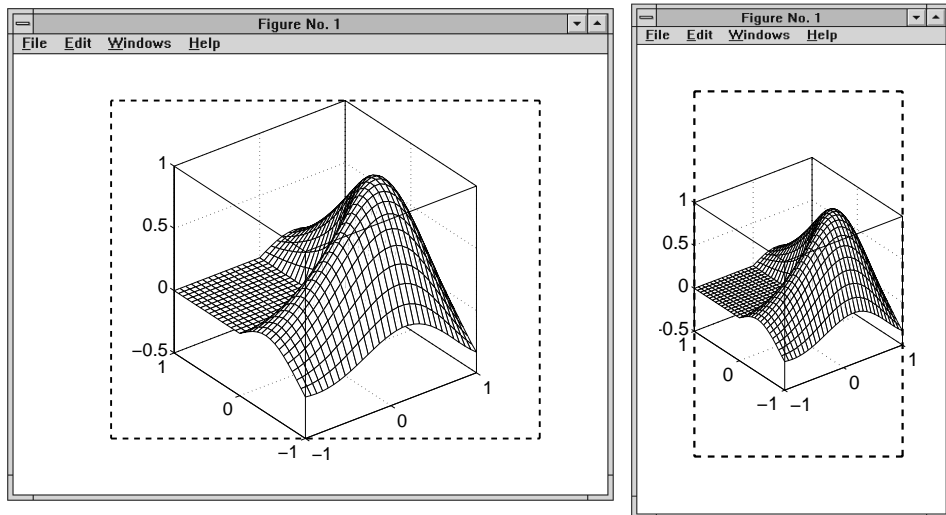
Overriding Stretch-to-Fill

To maintain a particular shape, you can specify the size of the axes in absolute units such as inches, which are independent of the Figure window size. However, this is not a good approach if you are writing an M-file that you want to work with a Figure window of any size. A better approach is to specify the aspect ratio of the axes and override automatic stretch-to-fill.

In cases where you want a specific aspect ratio, you can override stretching by specifying a value for these Axes properties:

- `DataAspectRatio` or `DataAspectRatioMode`
- `PlotBoxAspectRatio` or `PlotBoxAspectRatioMode`
- `CameraViewAngle` or `CameraViewAngleMode`

The first two sets of properties affect the aspect ratio directly. Setting either of the mode properties to `manual` simply disables stretch-to-fill while maintaining all current property values. In this case, MATLAB enlarges the axes until one dimension of the position rectangle constrains it:



Setting the `CameraViewAngle` property disables stretch-to-fill, and also prevents MATLAB from readjusting the size of the axes if you change the view.

Specifying the Aspect Ratio

It is important to understand how properties interact with each other to obtain the results you want. The `DataAspectRatio`, `PlotBoxAspectRatio`, and the `x`-, `y`-, and `z`-axis limits (`XLim`, `YLim`, and `ZLim` properties) all place constraints on the shape of the axes.

DataAspectRatio

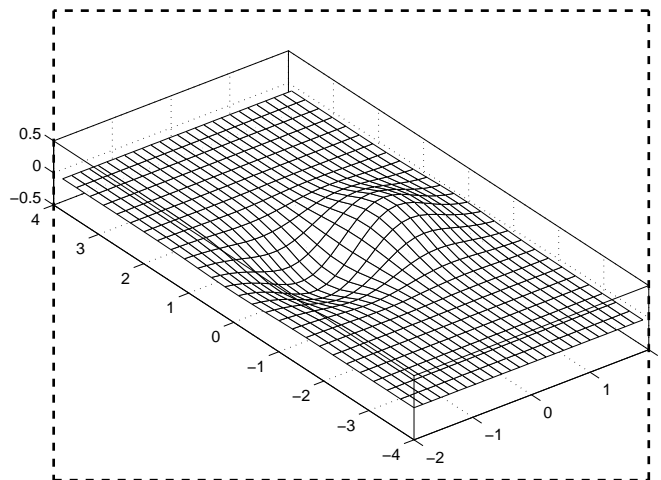
The `DataAspectRatio` property controls the ratio of the axis scales. For the mesh displayed in the “Example – MATLAB Defaults” section, the values are:

```
get(gca, 'DataAspectRatio')
ans =
    4    8    1
```

This means that four units in length along the *x*-axis cover the same data values as eight units in length along the *y*-axis and one unit in length along the *z*-axis. The axes fill the plot box, which has an aspect ratio of `[1 1 1]` by default.

If you want to view the mesh plot so that the relative magnitudes along each axis are equal with respect to each other, you can set the `DataAspectRatio` to `[1 1 1]`:

```
set(gca, 'DataAspectRatio', [1 1 1])
```



Setting the value of the `DataAspectRatio` property also sets the `DataAspectRatioMode` to `manual` and overrides stretch-to-fill so the specified aspect ratio is achieved.

PlotBoxAspectRatio

Looking at the value of the `PlotBoxAspectRatio` for the graph in the previous section shows that it has now taken on the former value of the `DataAspectRatio`:

```
get(gca, 'PlotBoxAspectRatio')
ans =
    4 8 1
```

MATLAB has rescaled the plot box to accommodate the graph using the specified `DataAspectRatio`.

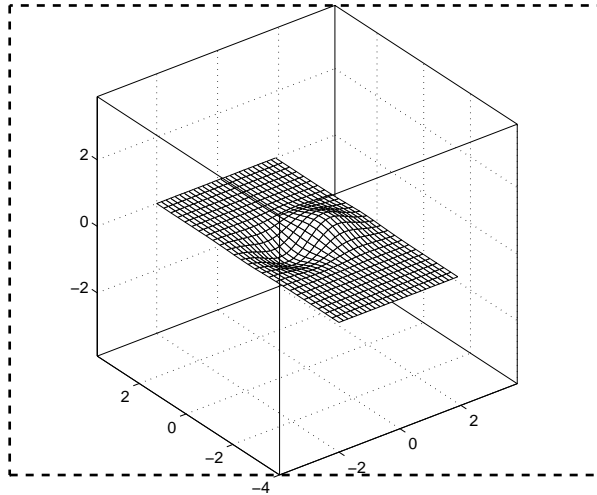
The `PlotBoxAspectRatio` property controls the shape of the Axes plot box. MATLAB sets this property to `[1 1 1]` by default and adjusts the `DataAspectRatio` property so that graphs fill the plot box if stretching is on, or until reaching a constraint if stretch-to-fill has been overridden.

When you set the value of the `DataAspectRatio` and thereby prevent it from changing, MATLAB varies the `PlotBoxAspectRatio` instead. If you specify both the `DataAspectRatio` and the `PlotBoxAspectRatio`, MATLAB is forced to change the axis limits to obey the two constraints you have already defined.

Continuing with the mesh example, if you set both properties,

```
set(gca, 'DataAspectRatio', [1 1 1], ...
    'PlotBoxAspectRatio', [1 1 1])
```

MATLAB changes the axis limits to satisfy the two constraints placed on the axes:



Adjusting Axis Limits

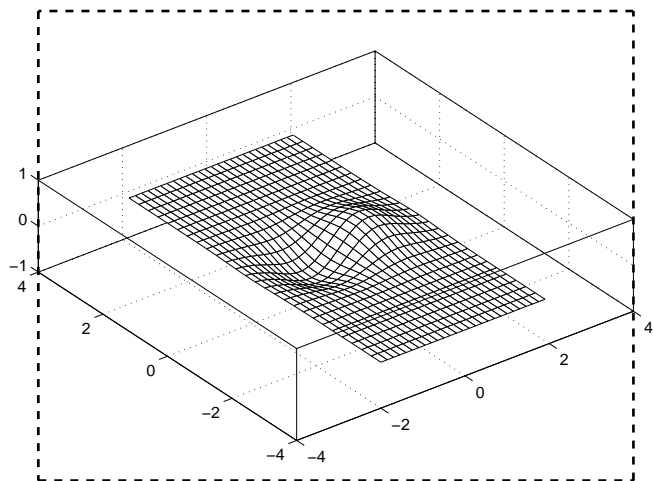
MATLAB enables you to set the axis limits to whichever values you want. However, specifying a value for `DataAspectRatio`, `PlotBoxAspectRatio`, and the axis limits, overconstrains the axes definition. For example, it is not possible for MATLAB to draw the axes if you set these values:

```
set(gca, 'DataAspectRatio', [1 1 1], ...
        'PlotBoxAspectRatio', [1 1 1], ...
        'XLim', [-4 4], ...
        'YLim', [-4 4], ...
        'ZLim', [-1 1])
```

In this case, MATLAB ignores the setting of the `PlotBoxAspectRatio` and automatically determines its value. These particular values cause the `PlotBoxAspectRatio` to return to its calculated value:

```
get(gca, 'PlotBoxAspectRatio')
ans =
    4    8    1
```

MATLAB can now draw the axes using the specified `DataAspectRatio` and axis limits:

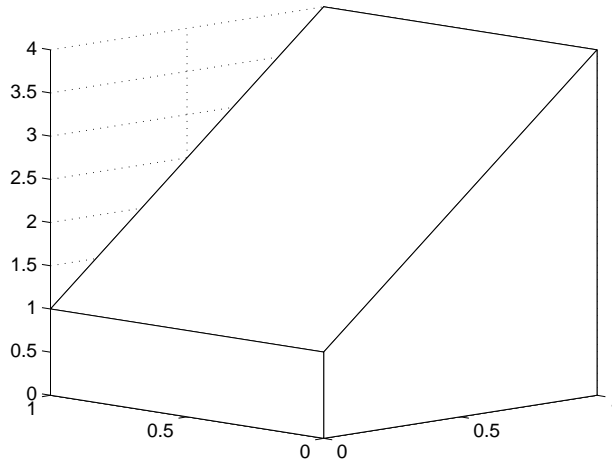


Example – Displaying Real Objects

If you want to display an object so that it looks realistic, you need to change MATLAB's defaults. For example, this data defines a wedge-shaped Patch object:

vertex_list =			vertex_connection =			
0	0	0	1	2	3	4
0	1	0	2	6	7	3
1	1	0	4	3	7	8
1	0	0	1	5	8	4
0	0	1	1	2	6	5
0	1	1	5	6	7	8
1	1	4				
1	0	4				

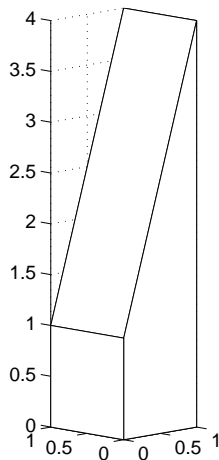
```
patch('Vertices', vertex_list, 'Faces', vertex_connection)
```



However, this axes distorts the actual shape of the solid object defined by the data. To display it in correct proportions, set the `DataAspectRatio`:

```
set(gca, 'DataAspectRatio', [1 1 1])
```

The units are now equal in the x -, y -, and z -directions and the axes is not being stretched to fill the position rectangle, revealing the true shape of the object:



Specialized Graphs

Bar and Area Graphs	4-2
Pie Charts.	4-13
Histograms	4-16
Discrete Data Graphs.	4-20
Direction and Velocity Vector Graphs	4-28
Contour Plots	4-34
Interactive Plotting	4-43
Animation.	4-45

Bar and Area Graphs

Bar and area graphs display vector or matrix data. These types of graphs are useful for viewing results over a period of time, comparing results from different datasets, and showing how individual elements contribute to an aggregate amount. Bar graphs are suitable for displaying discrete data, whereas area graphs are more suitable for displaying continuous data.

Bar Graph

MATLAB has four specialized functions that display bar graphs. These functions display 2- and 3-D bar graphs, and vertical and horizontal bar graphs.

	Two-Dimensional	Three-Dimensional
Vertical	bar	bar3
Horizontal	barh	bar3h

Grouped Bar Graph

By default, a bar graph represents each element in a matrix as one bar. Bars in a 2-D bar graph, created by the bar function, are distributed along the *x*-axis with each element in a column drawn at a different location. All elements in a row are clustered around the same location on the *x*-axis.

For example, define Y as a simple matrix:

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

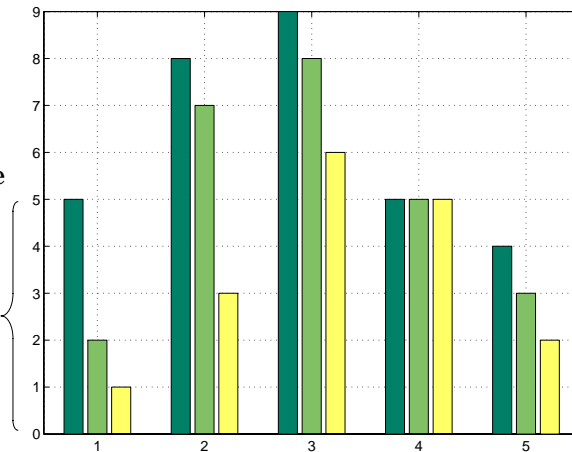
and issue the bar statement in its simplest form:

```
bar(Y)
```

The bars are clustered together by rows and evenly distributed along the x -axis.

The first cluster of bars represents the first row in Y .

$Y(1, :) = [5 \ 2 \ 1]$

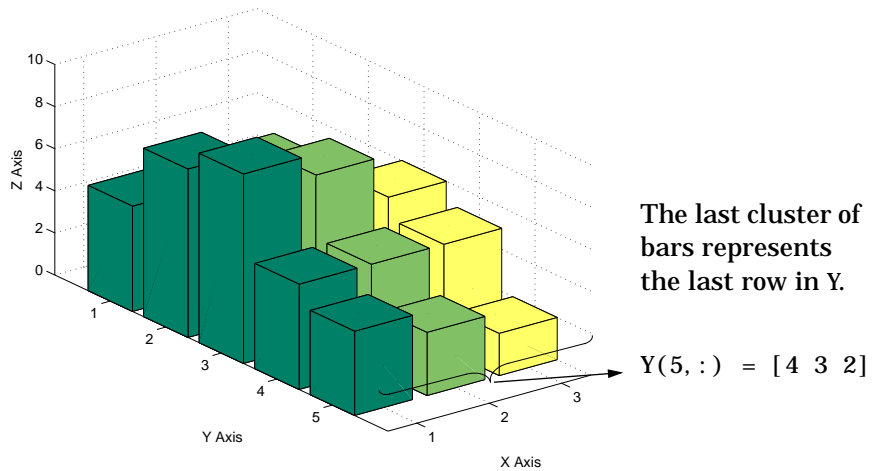


Detached 3-D Bars

The `bar3` function, in its simplest form, draws each element as a separate 3-D block, with the elements of each column distributed along the y -axis. Bars that represent elements in the first column of the matrix are centered at 1 along the x -axis. Bars that represent elements in the last column of the matrix are centered at `size(Y, 2)` along the x -axis. For example,

```
bar3(Y)
```

displays five groups of three bars along the y -axis. Notice that larger bars obscure $Y(1, 2)$ and $Y(1, 3)$.



By default, `bar3` draws detached bars. The statement `bar3(Y, 'detach')` has the same effect.

Labeling the Graph. To add axes labels and x tick marks to this bar graph, use the statements:

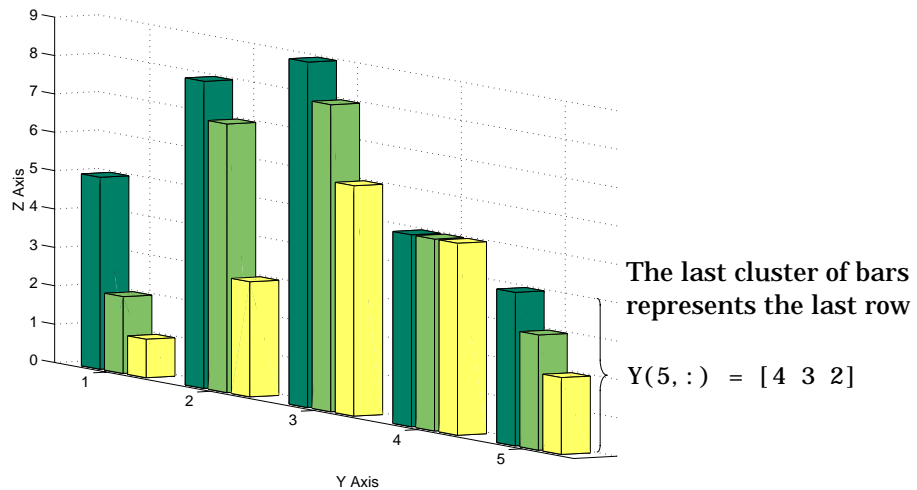
```
xl label (' X Axis')
yl label (' Y Axis')
zl label (' Z Axis')
set(gca, 'XTi ck', [1 2 3])
```

Grouped 3-D Bars

Cluster the bars from each row beside each other by specifying the argument `'group'`. For example:

```
bar3(Y, 'group')
```

groups the bars according to row and distributes the clusters evenly along the y -axis.



Stacked Bar graphs to Show Contributing Amounts

Bar graphs can show how elements in the same row of a matrix contribute to the sum of all elements in the row. These types of bar graphs are referred to as stacked bar graphs.

Stacked bar graphs display one bar per row of a matrix. The bars are divided into n segments, where n is the number of columns in the matrix. For vertical bar graphs, the height of each bar equals the sum of the elements in the row. Each segment is equal to the value of its respective element. Redefining Y:

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

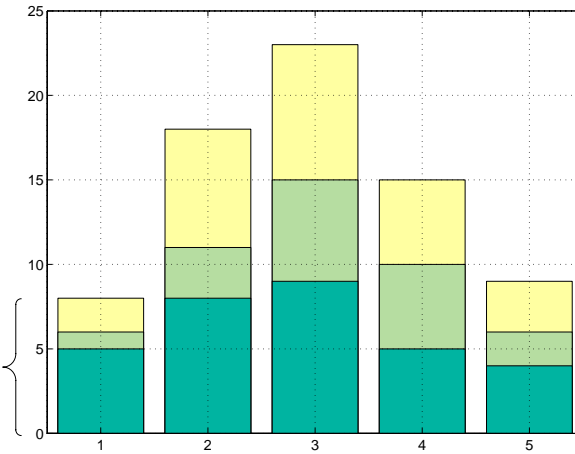
Create stacked bar graphs using the optional 'stack' argument. For example,

```
bar(Y, 'stack')
grid on
set(gca, 'Layer', 'top') % display gridlines on top of graph
```

creates a 2-D stacked bar graph, where all elements in a row correspond to the same x location.

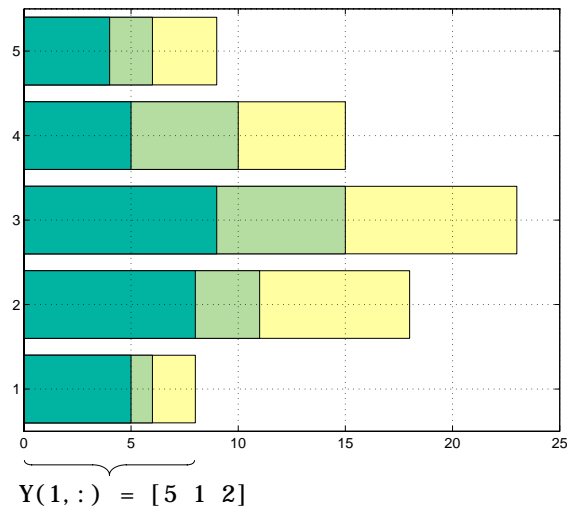
The first stack of bars represents the first row in Y.

$Y(1, :) = [5 \ 1 \ 2]$



For horizontal bar graphs, the length of each bar equals the sum of the elements in the row. The length of each segment is equal to the value of its respective element.

```
barh(Y, 'stack')
grid on
set(gca, 'Layer', 'top') % display gridlines on top of graph
```



The lower stack of bars represents the first row in Y.

$Y(1, :) = [5 \ 1 \ 2]$

Providing Your Own X Data

Bar graphs automatically generate x -axis values and label the x -axis tick lines. You can specify a vector of x values (or y values in the case of horizontal bar graphs) to label the axes.

For example, given temperature data,

```
temp = [29 23 27 25 20 23 23 27];
```

obtained from samples taken every five days during a thirty-five day period,

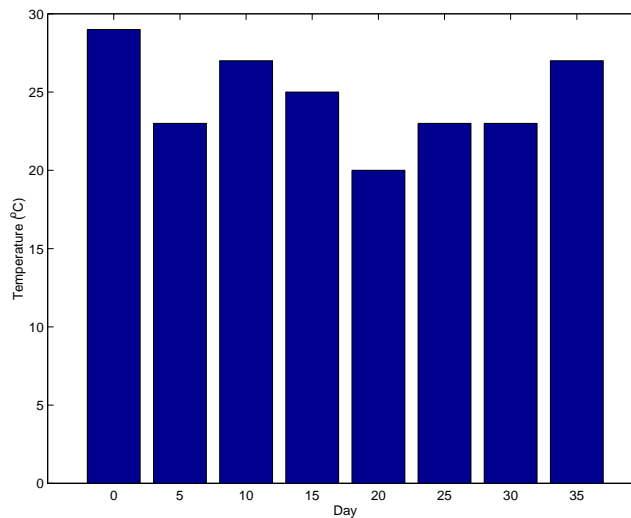
```
days = 0:5:35;
```

you can display a bar graph showing temperature measured along the y -axis and days along the x -axis using:

```
bar(days, temp)
```

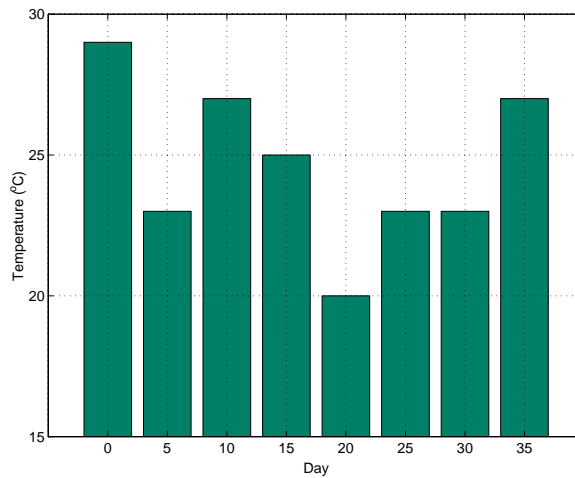
These statements add labels to the x - and y -axis:

```
xlabel('Day')  
ylabel('Temperature (°C)')
```



By default, the y -axis range is from 0 to 30. To focus on the temperature range from 15 to 30, change the y -axis limits:

```
set(gca, 'YLim', [15 30], 'Layer', 'top')
```



Overlaying Plots on Bar Graphs

You can overlay data on a bar graph by creating another axes in the same position. This enables you to have an independent y -axis for the overlaid dataset (in contrast to the `hold on` statement, which uses the same axes).

For example, consider a bioremediation experiment that breaks down hazardous waste components into nontoxic materials. The trichloroethylene (TCE) concentration and temperature data from this experiment are:

```
TCE = [515 420 370 250 135 120 60 20];
temp = [29 23 27 25 20 23 23 27];
```

This data was obtained from samples taken every five days during a thirty-five day period:

```
days = 0:5:35;
```

Display a bar graph and label the x - and y -axis using the statements

```
bar(days, temp)
xlabel('Day')
ylabel('Temperature (^{o}C)')
```

To overlay the concentration data on the bar graph, position a second axes at the same location as the first axes, but first save the handle of the first axes:

```
h1 = gca;
```

Create the second axes at the same location before plotting the second dataset:

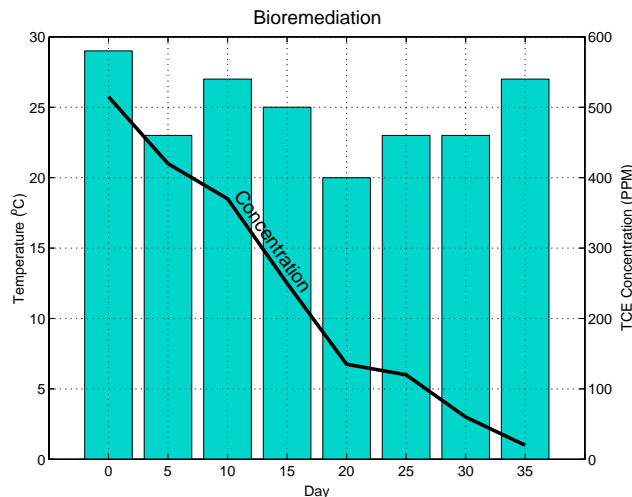
```
h2 = axes('Position', get(h1, 'Position'));
plot(days, TCE, 'LineWidth', 3)
```

To ensure that the second axes does not interfere with the first, locate the y -axis on the right side of the axes, make the background transparent, and set the second axes' x -tick marks to the empty matrix:

```
set(h2, 'YAxisLocation', 'right', 'Color', 'none', 'XTickLabel', [])
```

Align the x -axis of both axes and display the grid lines on top of the bars.

```
set(h2, 'XLim', get(h1, 'XLim'), 'Layer', 'top')
```



Annotating the Graph. These statements annotate the graph:

```
text(11, 380, 'Concentration', 'Rotation', -55, 'FontSize', 16)
ylabel('TCE Concentration (PPM)')
title('Bioremediation', 'FontSize', 16)
```

To print the graph, set the current Figure's `PaperPositionMode` to `auto`, which ensures the printed output matches the display.

```
set(gcf, 'PaperPositionMode', 'auto')
```

Area Graphs

The `area` function displays curves generated from a vector or from separate columns in a matrix. `area` plots the values in each column of a matrix as a separate curve and fills the area between the curve and the x -axis.

Area Graphs Showing Contributing Amounts

Area graphs are useful for showing how elements in a vector or matrix contribute to the sum of all elements at a particular x location. By default, `area` accumulates all values from each row in a matrix and creates a curve from those values.

Using this matrix,

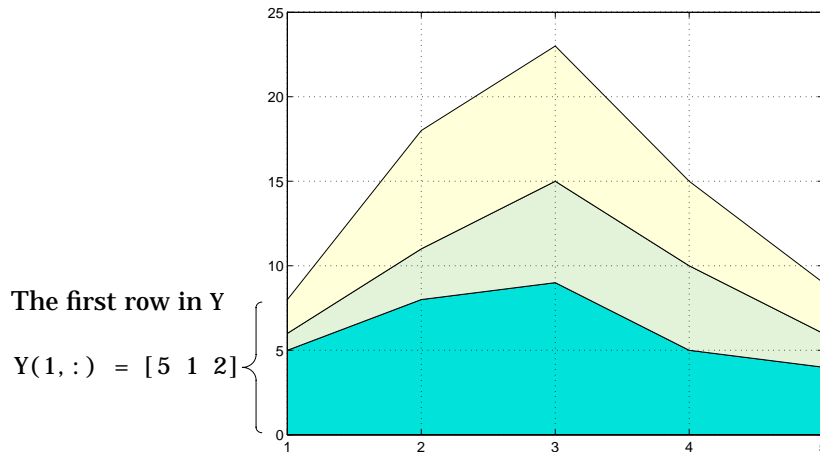
```
Y = [ 5 1 2
      8 3 7
      9 6 8
      5 5 5
      4 2 3];
```

the statement,

```
area(Y)
```

displays a graph containing three area graphs, one per column.

The height of the area graph is the sum of the elements in each row. Each successive curve uses the preceding curve as its base.



Displaying the Grid on Top. To display the grid lines in the foreground of the area graph and display only five grid lines along the x -axis, use the statements:

```
set(gca, 'Layer', 'top')
set(gca, 'XTick', 1:5)
```

Comparing Datasets with Area Graphs

Area graphs are useful for comparing different datasets. For example, given a vector containing sales figures,

```
sales = [51.6 82.4 90.8 59.1 47.0];
```

for the five-year period

```
x = 90:94;
```

and a vector containing profits figures for the same five-year period

```
profits = [19.3 34.2 61.4 50.5 29.4];
```

display both as two separate area graphs within the same axes. Set the color of the area interior (FaceColor), its edges (EdgeColor), and the width of the edge

lines (LineWidth). See the patch function in the online MATLAB Function Reference for a complete list of settable properties:

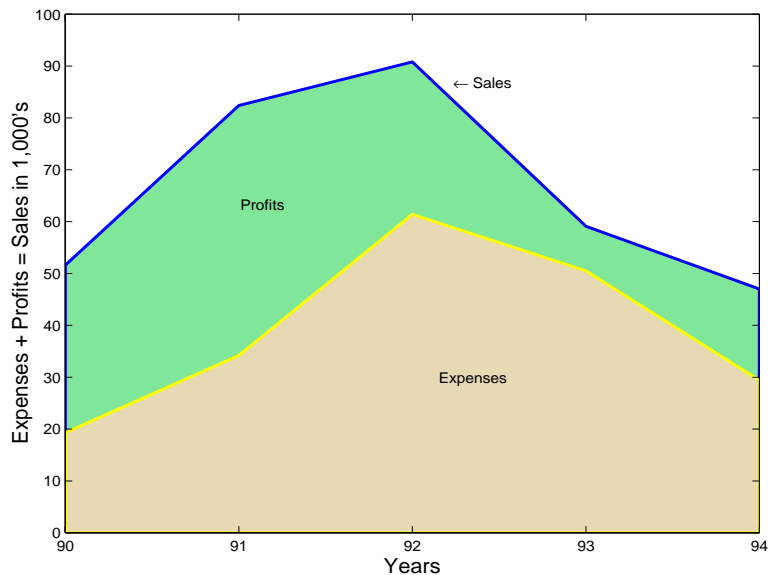
```
area(x, sales, 'FaceColor', [.5 .9 .6], ...
      'EdgeColor', 'b', ...
      'LineWidth', 2)

hold on
area(x, profits, 'FaceColor', [.9 .85 .7], ...
      'EdgeColor', 'y', ...
      'LineWidth', 2)

hold off
```

To annotate the graph, use the statements

```
set(gca, 'XTick', [90:94])
set(gca, 'Layer', 'top')
gtext('\leftarrow Sales')
gtext('Profits')
gtext('Expenses')
xlabel('Years', 'FontSize', 14)
ylabel('Expenses + Profits = Sales in 1,000's', 'FontSize', 14)
```



Pie Charts

Pie charts display the percentage that each element in a vector or matrix contributes to the sum of all elements. `pie` and `pie3` create 2-D and 3-D pie charts.

Here is an example using the `pie` function to visualize the contribution three products make to total sales. Given a matrix `X` where each column of `X` contains yearly sales figures for a specific product over a five-year period,

```
X = [ 19.3  22.1  51.6;
      34.2  70.3  82.4;
      61.4  82.9  90.8;
      50.5  54.9  59.1;
      29.4  36.3  47.0];
```

sum each row in `X` to calculate total sales for each product over the five-year period:

```
x = sum(X);
```

You can offset the slice of the pie that makes the greatest contribution using the `explode` input argument. This argument is a vector of zero and nonzero values. Nonzero values offset the respective slice from the chart.

First, create a vector containing zeros:

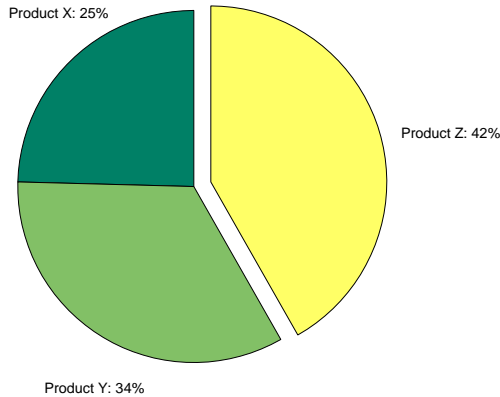
```
explode = zeros(size(x));
```

Then find the slice that contributes the most and set the corresponding `explode` element to 1:

```
[c, offset] = max(x);
explode(offset) = 1;
```

The `explode` vector contains the elements `[0 0 1]`. To create the exploded pie chart, use the statement:

```
h = pie(x, explode); colormap summer
```



Labeling the Graph. The pie chart's labels are Text graphics objects. To modify the text strings and their positions, first get the objects' strings and extents. Braces around a property name ensure that `get` outputs a cell array, which is important when working with multiple objects.

```
textObjs = findobj(h, 'Type', 'text');
oldStr = get(textObjs, {'String'});
val = get(textObjs, {'Extent'});
oldExt = cat(1, val{:});
```

Create the new strings, then set the Text objects' `String` properties to the new strings:

```
Names = {'Product X: ', 'Product Y: ', 'Product Z: '};
newStr = strcat(Names, oldStr);
set(textObjs, {'String'}, newStr)
```

Find the difference between the widths of the new and old text strings and change the values of the Position properties:

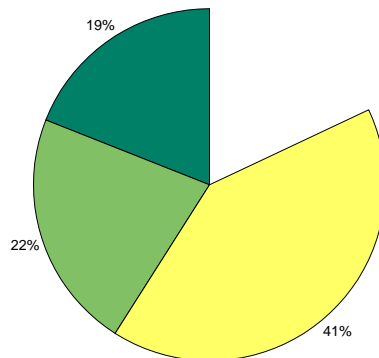
```
val1 = get(textObj s, {'Extent'});
newExt = cat(1, val1{:});
offset = sign(oldExt(:, 1)) .* (newExt(:, 3) - oldExt(:, 3)) / 2;
pos = get(textObj s, {'Position'});
textPos = cat(1, pos{:});
textPos(:, 1) = textPos(:, 1) + offset;
set(textObj s, {'Position'}, num2cell(textPos, [3, 2]))
```

Pie Charts Missing a Piece

When the sum of the elements in the first input argument is equal to or greater than 1, `pie` and `pie3` normalize the values. So, given a vector of elements x , each slice has an area of $x_i / \text{sum}(x_i)$, where x_i is an element of x . The normalized value specifies the fractional part of each pie slice.

When the sum of the elements in the first input argument is less than 1, `pie` and `pie3` do not normalize the elements of vector x . They draw a partial pie. For example,

```
x = [.19 .22 .41];
pie(x)
```



Histograms

MATLAB's histogram functions show the distribution of data values. The functions that create histograms are `hist` and `rose`. `hist` displays data in a Cartesian coordinate system and `rose` displays data in a polar coordinate system.

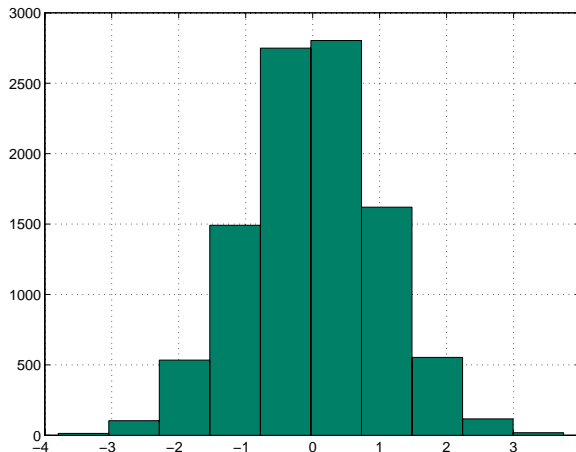
The histogram functions count the number of elements within a range and display each range as a rectangular bin. The height (or length when using `rose`) of the bins represents the number of values that fall within each range.

Histograms in Cartesian Coordinate Systems

The `hist` function shows the distribution of the elements in `Y` as a histogram with equally spaced bins between the minimum and maximum values in `Y`. If `Y` is a vector and is the only argument, `hist` creates up to 10 bins. For example,

```
yn = randn(10000, 1);  
hist(yn)
```

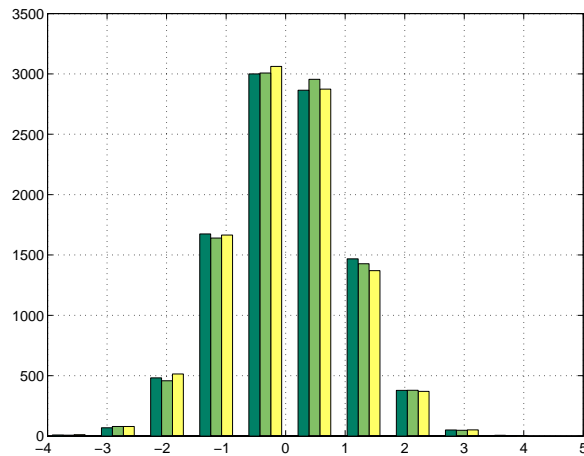
generates 10,000 random numbers and creates a histogram with 10 bins distributed along the *x*-axis between the minimum and maximum values of `yn`.



When `Y` is a matrix, `hist` creates a set of bins for each column, displaying each set in a separate color. The statements

```
Y = randn(10000, 3);
hist(Y)
```

create a histogram showing 10 bins for each column in `Y`.



Histograms in Polar Coordinate Systems

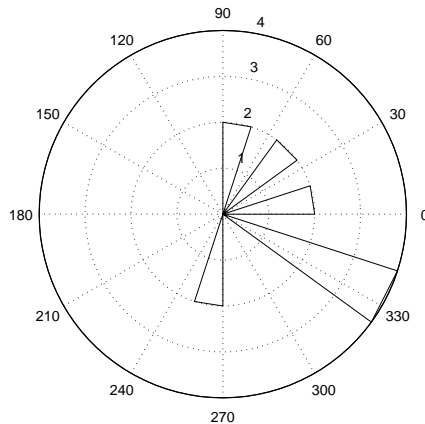
A rose plot is a histogram created in a polar coordinate system. For example, consider samples of the wind direction taken over a 12-hour period:

```
wdir = [45 90 90 45 360 335 360 270 335 270 335 335];
```

To display this data using the `rose` function, convert the data to radians; then use the data as an argument to the `rose` function:

```
wdir = wdir * pi / 180;
rose(wdir)
```

The plot shows that the wind direction was primarily 335° during the 12-hour period.



Number of Bins Created

`hist` and `rose` interpret their second argument in one of two ways—as the locations on the axis or the number of bins. When the second argument is a vector `x`, it specifies the locations on the axis and distributes the elements in `length(x)` bins. When the second argument is a scalar `x`, `hist` and `rose` distribute the elements in `x` bins.

For example, compare the distribution of data created by two MATLAB functions that generate random numbers. The `randn` function generates normally distributed random numbers, whereas, the `rand` function generates uniformly distributed random numbers.

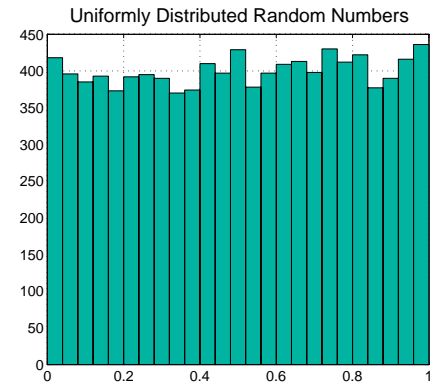
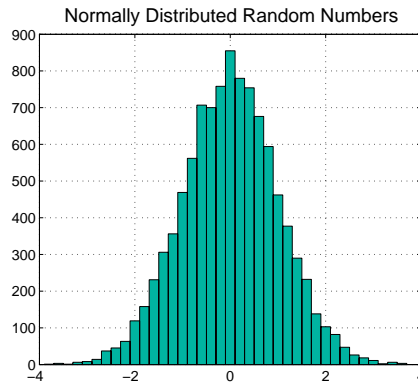
```
yn = randn(10000, 1);
yu = rand(10000, 1);
```

The first histogram displays the data distribution resulting from the `randn` function. The locations on the *x*-axis and number of bins depend on the vector `x`:

```
x = min(yn) : .2 : max(yn);
subplot(1, 2, 1)
hist(yn, x)
title('Normally Distributed Random Numbers', 'FontSize', 16)
```

The second histogram displays the data distribution resulting from the `rand` function and explicitly creates 25 bins along the x -axis:

```
subplot(1, 2, 2)
hist(yu, 25)
title('Uniformly Distributed Random Numbers', 'FontSize', 16)
```



Note: You can change the aspect ratio of the histogram plots using the mouse to resize the Figure window. However, before creating hardcopy output, set the Figure's `PaperPositionMode` to `auto` to produce printed output that matches the display:

```
set(gcf, 'PaperPositionMode', 'auto')
```

Discrete Data Graphs

MATLAB has a number of specialized functions that are appropriate for displaying discrete data. This section describes how to use stem plots and stairstep plots to display this type of data. (Bar charts, discussed earlier in this chapter, are also suitable for displaying discrete data.)

Two- and Three-dimensional Stem Plots

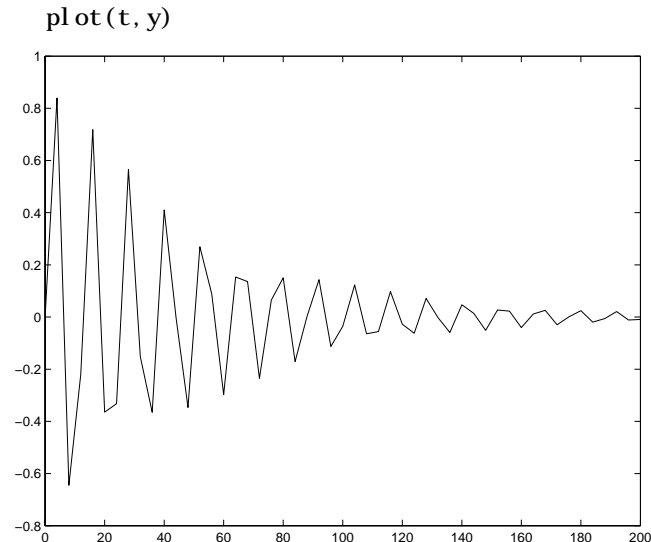
A stem plot displays data as lines (stems) terminated with a marker symbol at each data value. In a 2-D graph, stems extend from the x -axis. In a 3-D graph, stems extend from the xy -plane.

Two-dimensional Stem Plots

The `stem` function displays two-dimensional discrete sequence data. For example, evaluating the function $y = e^{-\alpha t} \cos \beta t$ with the values,

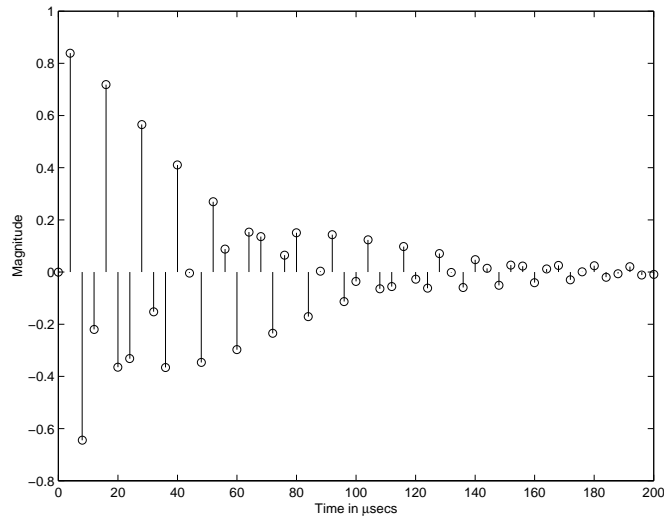
```
alpha = .02; beta = .5; t = 0:4:200;
y = exp(-alpha*t) .* sin(beta*t);
```

yields a vector of discrete values for y at given values of t . A line plot shows the data points connected with a straight line:



A stem plot of the same function plots only discrete points on the curve:

```
stem(t, y)
```



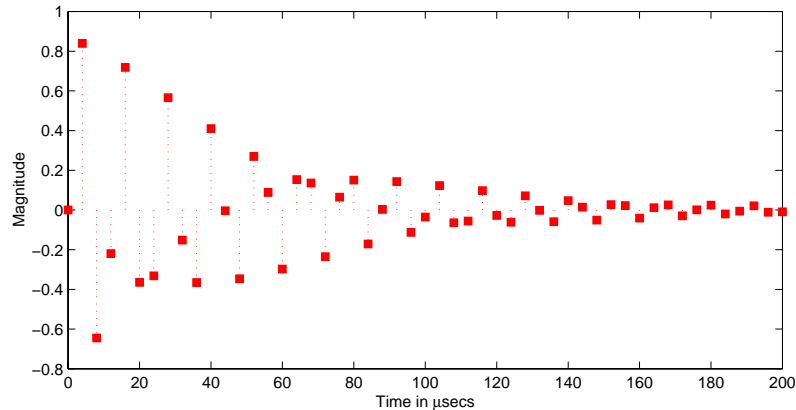
Add axes labels to the x- and y-axis:

```
xlabel('Time in \musecs')
ylabel('Magni tude')
```

If you specify only one argument, the number of samples is equal to the length of that argument. In this example, the number of samples is a function of `t`, which contains 51 elements and determines the length of `y`.

Customizing the Graph. You can specify the line style, the type of marker, and the color used in the stem plot. For example, adding the string `'sr'` specifies a dotted line (`:`), a square marker (`s`), and a red color (`r`). The `'fill'` argument colors the face of the marker.

```
stem(t, y, '—sr', 'fill')
```



Setting the aspect ratio of the x- and y-axis to 2:1 improves the utility of the graph.

```
set(gca, 'PlotBoxAspectRatio', [2 1 1])
```

See `axes` and `LineStyle` in the online MATLAB Function Reference for information on the `PlotBoxAspectRatio` property and a list of line styles and marker types.

Combining plots. Sometimes it is useful to display more than one plot simultaneously with a stem plot to show how you arrived at a result. For example, create a linearly spaced vector with 60 elements and define two functions, `a` and `b`:

```
x = linspace(0, 2*pi, 60);
a = sin(x);
b = cos(x);
```

Create a stem plot showing the linear combination of the two functions:

```
stem_handles = stem(x, a+b)
```

Overlaying a and b as line plots helps visualize the functions. Before plotting the two curves, set `hold` to on so MATLAB does not clear the stem plot:

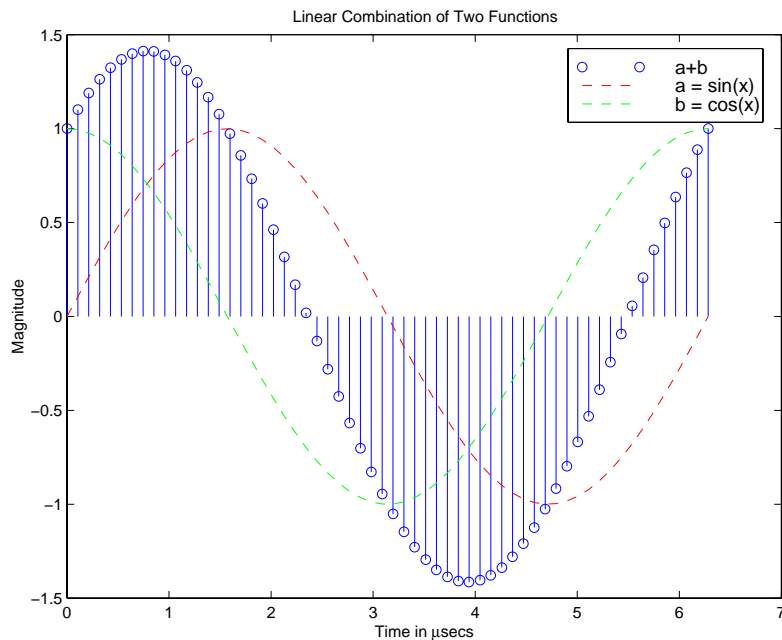
```
hold on
plot_handles = plot(x, a, '--r', x, b, '--g')
hold off
```

Use `legend` to annotate the graph. The stem and plot handles passed to `legend` identify which lines to label. Stem plots are composed of two lines, one draws the markers and the other draws the vertical stems. To create the legend, use the first handle returned by `stem`, which identifies the marker line:

```
legend_handles = [stem_handles(1); plot_handles];
legend(legend_handles, 'a + b', 'a = sin(x)', 'b = cos(x)')
```

Labeling the axes and creating a title finishes the graph:

```
xlabel('Time in \musecs')
ylabel('Magnitude')
title('Linear Combination of Two Functions')
```



Three-dimensional Stem Plots

`stem3` displays 3-D stem plots extending from the xy -plane. With only one vector argument, MATLAB plots the stems in one row at $x = 1$ or $y = 1$, depending on whether the argument is a column or row vector. `stem3` is intended to display data that you cannot visualize in a 2-D view.

For example, fast Fourier transforms are calculated at points around the unit circle on the complex plane. So, it is interesting to visualize the plot around the unit circle. Calculating the unit circle,

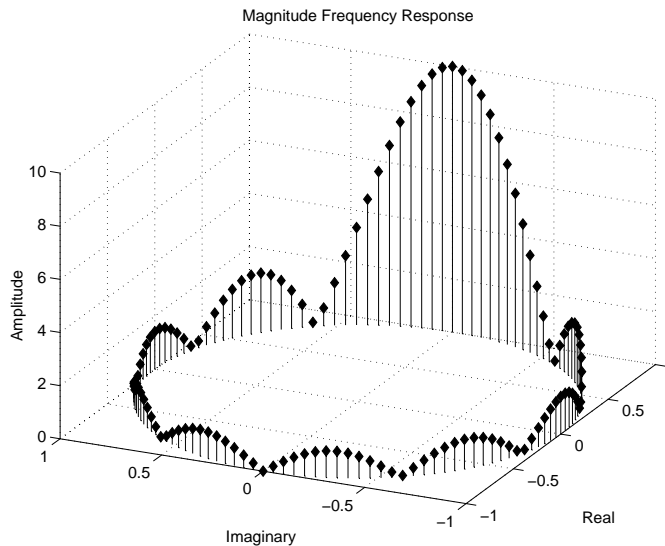
```
th = (0:127)/128*2*pi;  
x = cos(th);  
y = sin(th);
```

and the magnitude frequency response of a step function,

```
f = abs(fft(ones(10,1),128));
```

display the data using a 3-D stem plot, terminating the stems with filled diamond markers:

```
stem3(x,y,f,'d','fill')  
view([-65 30])
```



Label the graph with the statements:

```
xlabel('Real')
ylabel('Imaginary')
zlabel('Amplitude')
title('Magnitude Frequency Response')
```

To change the orientation of the view, turn on mouse-based 3-D rotation:

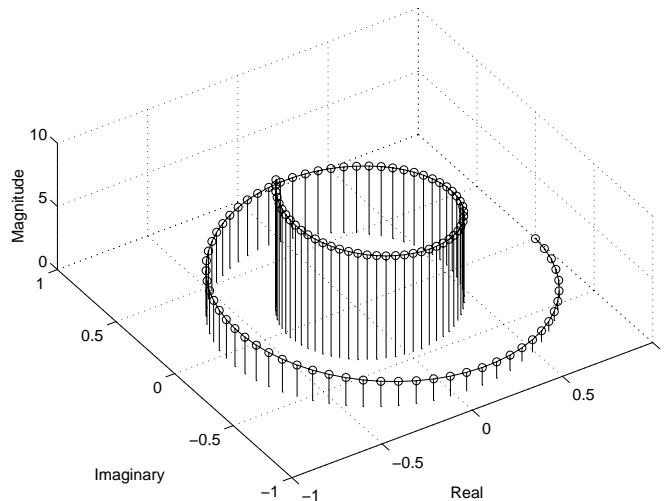
```
rotate3d on
```

Three-dimensional stem plots work well when visualizing discrete functions that do not output a large number of data points. For example, you can use `stem3` to visualize the Laplace transform basis function, $y = e^{-st}$ for a particular constant value of s :

```
t = 0: .1: 10;          % Time limits
s = 0.1+i;              % Spiral rate
y = exp(-s*t);          % Compute decaying exponential
```

Using t as magnitudes that increase with time, create a spiral with increasing height and draw a curve through the tops of the stems to improve definition:

```
stem3(real(y), imag(y), t)
hold on
plot3(real(y), imag(y), t, 'k')
hold off
```



Add axes labels, with the statements:

```
xl label (' Real ' )
yl label (' Imagi nary' )
zl label (' Magni tude' )
```

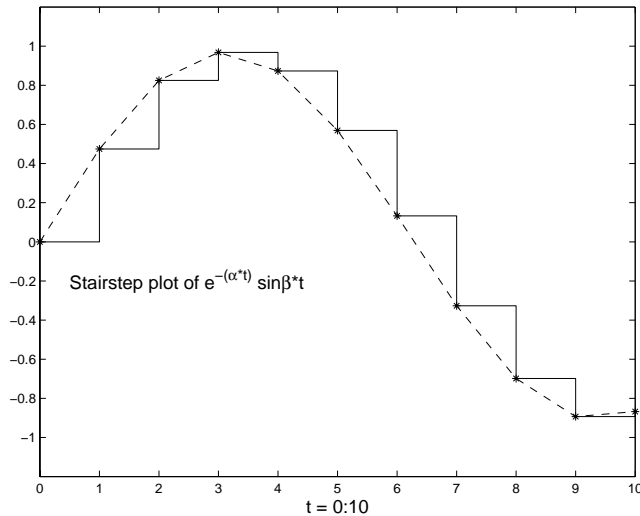
Stairstep Plots

Stairstep plots display data as the leading edges of a constant interval (i.e., zero-order hold state). This type of plot holds the data at a constant y -value for all values between $x(i)$ and $x(i+1)$, where i is the index into the x data. This type of plot is useful for drawing time-history plots of digitally sampled data systems. For example, define a function f that varies over time:

```
alpha = .01;
beta = .5;
t = 0:10;
f = exp(-alpha*t) .* sin(beta*t);
```

Display the function as a stairstep plot and a linearly interpolated function:

```
stairs(t, f)
hold on
plot(t, f, '—*')
hold off
```



Annotate the graph and set the axes limits:

```
label = 'Stairstep plot of  $e^{-(\alpha*t)} \sin\beta t$ ';  
text(.5, -0.2, label, 'FontSize', 14)  
xlabel('t = 0: 10', 'FontSize', 14)  
axis([0 10 -1.2 1.2])
```

Direction and Velocity Vector Graphs

Several MATLAB functions display data consisting of direction vectors and velocity vectors. This section describes these functions.

Function	Description
<code>compass</code>	Displays vectors emanating from the origin of a polar plot.
<code>feather</code>	Displays vectors extending from equally spaced points along a horizontal line.
<code>quiver</code>	Displays 2-D vectors specified by (u,v) components.
<code>quiver3</code>	Displays 3-D vectors specified by (u,v,w) components.

You can define the vectors using one or two arguments. The arguments specify the x and y components of the vectors relative to the origin.

If you specify two arguments, the first specifies the x components of the vectors and the second the y components of the vectors. If you specify one argument, the functions treat the elements as complex numbers. The real parts are the x components and the imaginary parts are the y components.

Compass Plots

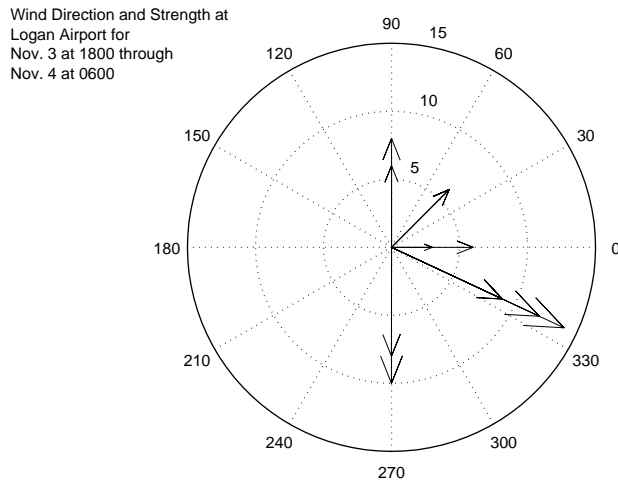
The `compass` function shows vectors emanating from the origin of a graph. The function takes Cartesian coordinates and plots them on a circular grid.

This example shows a compass plot indicating the wind direction and strength during a 12-hour period. Two vectors define the wind direction and strength:

```
wdi r = [45 90 90 45 360 335 360 270 335 270 335 335];  
knots = [6 6 8 6 3 9 6 8 9 10 14 12];
```


Convert the wind direction, given as angles, into radians before converting the wind direction into Cartesian coordinates:

```
rdi r = wdi r * pi /180;
[x, y] = pol2cart(rdi r, knots);
compass(x, y)
```



Create text to annotate the graph:

```
desc = {'Wind Direction and Strength at',  
        'Logan Airport for ',  
        'Nov. 3 at 1800 through',  
        'Nov. 4 at 0600'};  
text(-28, 15, desc)
```

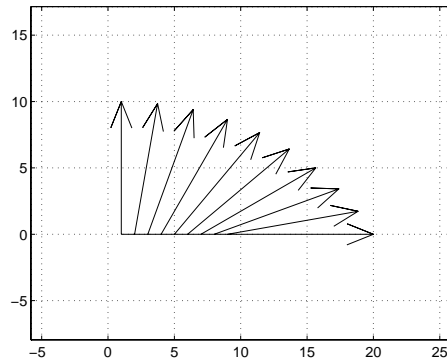
Feather Plots

The feather function shows vectors emanating from a straight line parallel to the x -axis. For example, create a vector of angles from 90° to 0° and a vector the same size, with each element equal to 1:

```
theta = 90:-10:0;  
r = ones(size(theta));
```

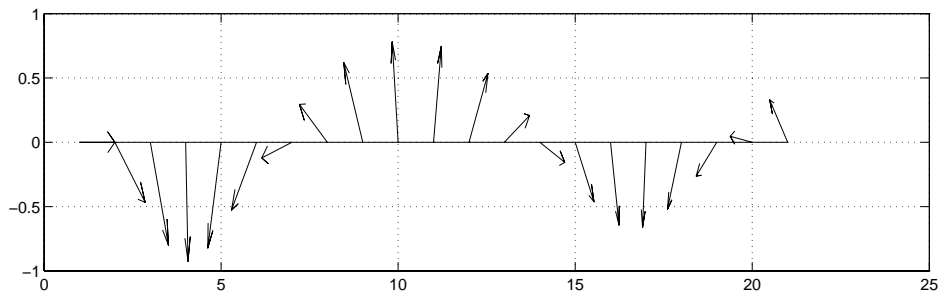
Before creating a feather plot, transform the data into Cartesian coordinates and increase the magnitude of r to make the arrows more distinctive:

```
[u, v] = pol2cart(theta*pi/180, r*10);
feather(u, v)
axis equal
```



If the input argument, Z , is a matrix of complex numbers, `feather` interprets the real parts of Z as the x components of the vectors and the imaginary parts as the y components of the vectors:

```
t = 0: .5: 10; % Time limits
s = .05+i; % Spiral rate
Z = exp(-s*t); % Compute decaying exponential
feather(Z)
```



This particular graph looks better if you change the Figure's aspect ratio by stretching the Figure lengthwise using the mouse. However, to maintain this shape in the printed output, set the Figure's `PaperPositionMode` to `auto`:

```
set(gcf, 'PaperPositionMode', 'auto')
```

In this mode, MATLAB prints the Figure exactly as it appears on screen.

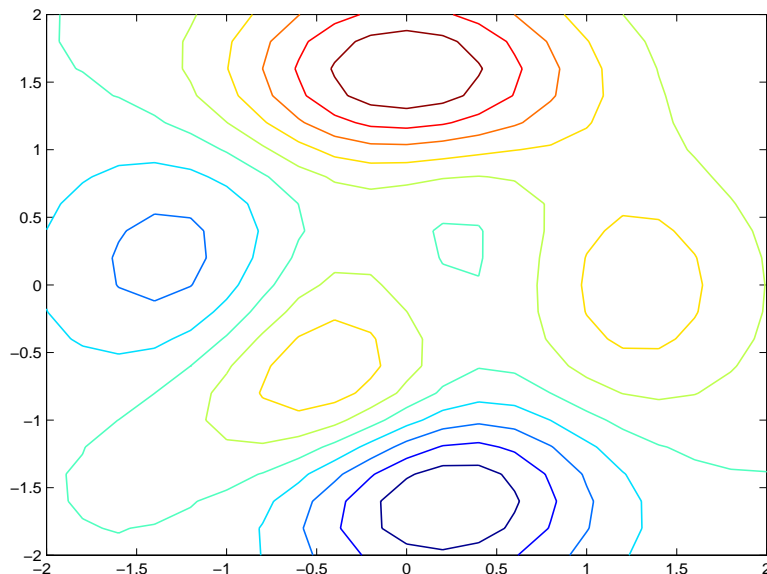
Quiver Plots

The `quiver` and `quiver3` functions show vectors at given points in two- and three-dimensional space. The vectors are defined by x and y components.

Two-dimensional Quiver Plots

A quiver plot is useful when displayed with another plot. For example, create 10 contours of the peaks function (the next section describes contour plots):

```
n = -2.0:2.0;
[X, Y, Z] = peaks(n);
contour(X, Y, Z, 10)
```

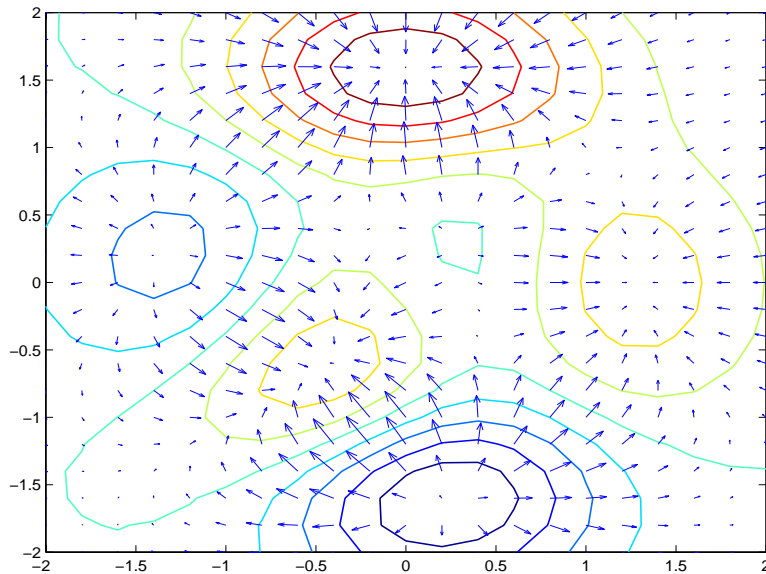


Now use the gradient function to create the vector components to use as inputs to `quiver`.

```
[U, V] = gradient(Z, .2);
```

Set `hold` to on and add the contour plot:

```
hold on
quiver(X, Y, U, V)
hold off
```



Three-dimensional Quiver Plots

Three-dimensional quiver plots display vectors consisting of (u,v,w) components at (x,y,z) locations. For example, you can show the path of a projectile as a function of time:

$$z(t) = v_z t + \frac{at^2}{2}$$

First, assign values to the constants v_z and a :

```
vz = 10;           % Velocity
a = -32;           % Acceleration
```

Then, calculate the height z , as time varies from 0 to 1 in increments of 0.1:

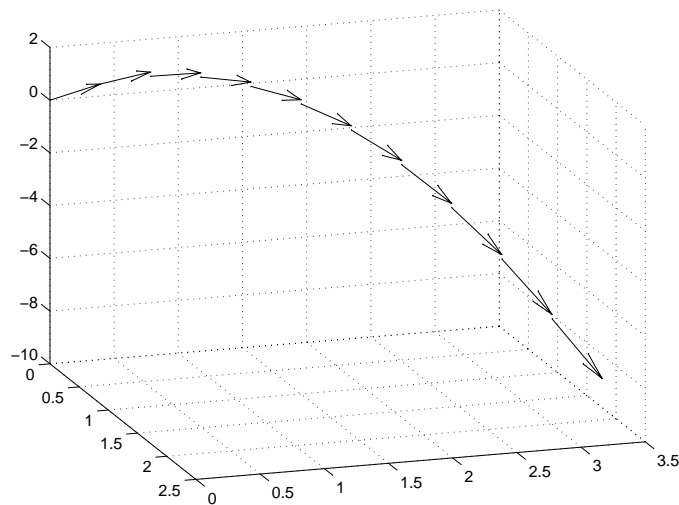
```
t = 0: .1: 1;
z = vz*t + 1/2*a*t.^2;
```

Calculate the position in the x and y directions:

```
vx = 2;
x = vx*t;
vy = 3;
y = vy*t;
```

Compute the components of the velocity vectors and display the vectors using the 3-D quiver plot:

```
u = gradient(x);
v = gradient(y);
w = gradient(z);
scale = 0;
quiver3(x, y, z, u, v, w, scale)
axis square
```



Contour Plots

The contour functions create, display, and label isolines determined by one or more matrices.

Function	Description
clabel	Generates labels using the contour matrix and displays the labels in the current Figure.
contour	Displays 2-D isolines generated from values given by a matrix Z.
contour3	Displays 3-D isolines generated from values given by a matrix Z.
contourf	Displays a 2-D contour plot and fills the area between the isolines with a solid color.
contourc	Low-level function to calculate the contour matrix used by the other contour functions.

Two other functions also create contours. `meshc` displays a contour in addition to a mesh, and `surf` displays a contour in addition to a surface. The section “Changing the Offset” briefly discusses these functions.

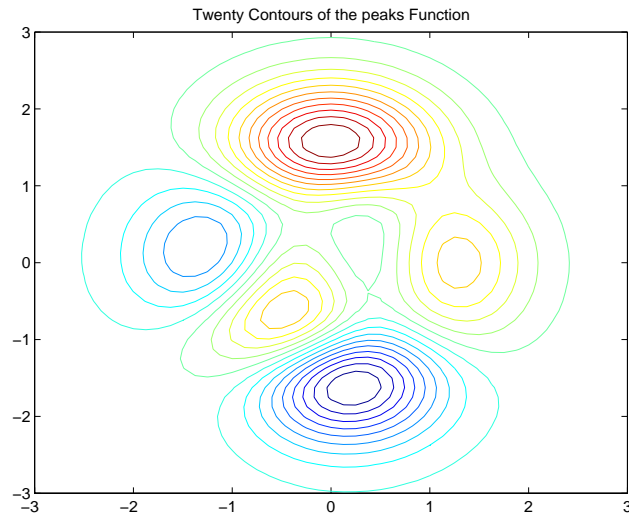
Creating Simple Contour Plots

`contour` and `contour3` display 2- and 3-D contours, respectively. They require only one input argument—a matrix interpreted as heights with respect to a plane. In this case, the contour functions determine the number of contours to display based on the minimum and maximum data values.

To explicitly set the number of contour levels displayed by the functions, you specify a second optional argument. For example,

```
[X, Y, Z] = peaks;  
contour(X, Y, Z, 20)
```

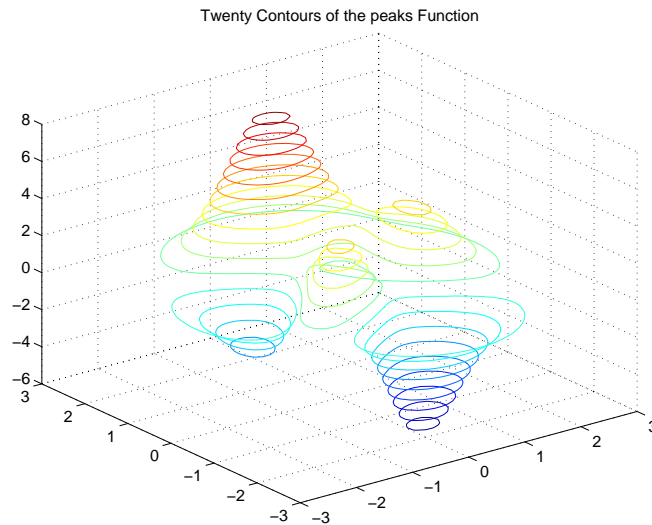
displays 20 contours of the peaks function in a 2-D view:



The statements

```
[X, Y, Z] = peaks;  
contour3(X, Y, Z, 20)
```

display 20 contours of the peaks function in a 3-D view:



Labeling Contours

Each contour level has a value associated with it. `clabel` uses these values to display labels for 2-D contour lines. The contour matrix contains the values `clabel` uses for the labels. This matrix is returned by `contour`, `contour3`, and `contourf`. (See the “Contouring Algorithm” section.)

`clabel` optionally returns the handles of the Text objects used as labels. You can then use these handles to set the properties of the label string.

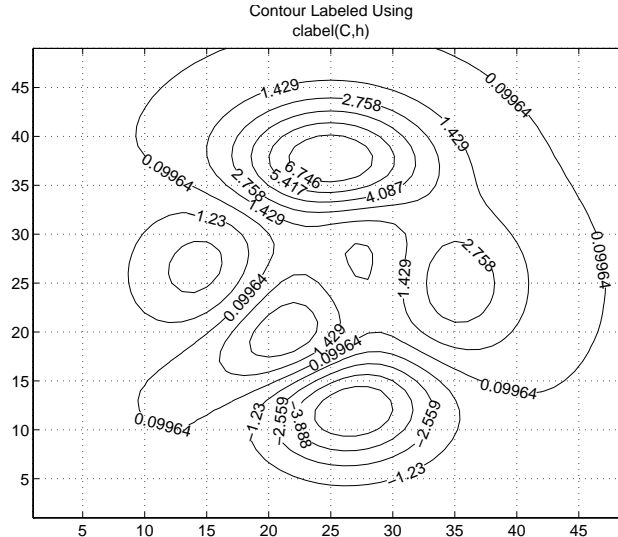
For example, display 10 contour levels of the peaks function,

```
Z = peaks;  
[C, h] = contour(Z, 10);
```

then label the contours and display a title.

```
clabel(C, h)  
title({'Contour Labeled Using', 'clabel(C,h)'} )
```

Note that `clabel` labels only those contour lines that are large enough to have an inline label inserted.



The 'manual' option enables you to add labels by selecting the contour you want to label with the mouse.

You can also use this option to label only those contours you select interactively.

For example,

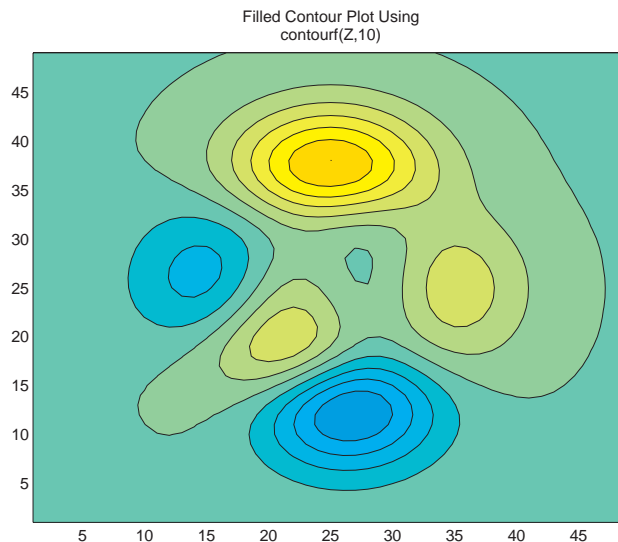
```
clabel(C,h,'manual')
```

displays a crosshair cursor when your cursor is inside the Figure. Pressing any mouse button labels the contour line closest to the center of the crosshair.

Filled Contours

`contourf` displays a two-dimensional contour plot and fills the areas between contour lines. Use the `caxis` function to control the mapping of contour to color. For example, this filled contour plot of the peaks data uses `caxis` to map the fill colors into the center of the colormap.

```
Z = peaks;  
[C,h] = contourf(Z,10);  
caxis([-20 20])  
title({'Filled Contour Plot Using', 'contourf(Z,10)'})
```



See the `caxis` description in the online MATLAB Function Reference.

Drawing a Single Contour Line at a Desired Level

The contouring functions permit you to specify the number of contour levels or the particular contour levels to draw. In the case of `contour`, the two forms of the function are `contour(Z, n)` and `contour(Z, v)`. `Z` is the data matrix, `n` is the number of contour lines, and `v` is a vector of specific contour levels.

MATLAB does not differentiate between a scalar and a one-element vector. So, if `v` is a one-element vector specifying a single contour at that level, `contour` interprets it as the number of contour lines, not the contour level.

Consequently, `contour(Z, v)` behaves in the same manner as `contour(Z, n)`.

To display a single contour line, define `v` as a two-element vector with both elements equal to the desired contour level. For example, create a 3-D contour of the peaks function:

```
xrange = -3: .125: 3;  
yrange = xrange;  
[X, Y] = meshgrid(xrange, yrange);  
Z = peaks(X, Y);  
contour3(X, Y, Z)
```

To display only one contour level at `Z = 1`, define `v` as `[1 1]`:

```
v = [1 1]  
contour3(X, Y, Z, v)
```

The Contouring Algorithm

The `contourc` function calculates the contour matrix for the other contour functions. It is a low-level function that is not called from the command line.

The contouring algorithm first determines which contour levels to draw. If you specified the input vector `v`, the elements of `v` are the contour level values and `length(v)` determines the number of contour levels generated. If you do not specify `v`, the algorithm chooses no more than 20 contour levels that are divisible by 2 or 5.

The contouring algorithm treats the input matrix `Z` as a regularly spaced grid, with each element connected to its nearest neighbors. The algorithm scans this matrix comparing the values of each block of four neighboring elements (i.e., a cell) in the matrix to the contour level values. If a contour level falls within a cell, the algorithm performs a linear interpolation to locate the point at which

the contour crosses the edges of the cell. The algorithm connects these points to produce a segment of a contour line.

`contour`, `contour3`, and `contourf` return a two-row matrix specifying all the contour lines. The format of the matrix is:

```
C = [   value1   xdata(1)   xdata(2)...
        numv     ydata(1)   ydata(2)... ]
```

The first row of the column that begins each definition of a contour line contains the value of the contour, as specified by `v` and used by `clabel`. Beneath that value is the number of (x,y) vertices in the contour line. Remaining columns contain the data for the (x,y) pairs. For example, the contour matrix calculated by `C = contour(peaks(3))` is

Three vertices at $v = -.2$	Columns 1 through 7						
	-0.2000	1.8165	2.0000	2.1835	0	1.0003	2.0000
	3.0000	1.0000	1.0367	1.0000	3.0000	1.0000	1.1998
Three vertices at $v = 0$	Columns 8 through 14						
	3.0000	0	1.0000	1.0359	1.0000	0.2000	1.6669
	1.0002	3.0000	2.9991	2.0000	1.0018	5.0000	3.0000
	Columns 15 through 21						
	1.2324	2.0000	2.8240	2.3331	0.4000	2.0000	2.6130
	2.0000	1.3629	2.0000	3.0000	5.0000	2.8530	2.0000
	Columns 22 through 28						
	2.0000	1.4290	2.0000	0.6000	2.0000	2.4020	2.0000
	1.5261	2.0000	2.8530	5.0000	2.5594	2.0000	1.6892
Five vertices at $v = .8$	Columns 29 through 35						
	1.6255	2.0000	0.8000	2.0000	2.1910	2.0000	1.8221
	2.0000	2.5594	5.0000	2.2657	2.0000	1.8524	2.0000
	Column 36						
	2.0000						
	2.2657						

The circled values begin each definition of a contour line.

Changing the Offset of a Contour

The `surf` and `meshc` functions display contours beneath a surface or a mesh plot. These functions draw the contour plot at the axes' minimum z -axis limit. To specify your own offset, you must change the `ZData` values of the contour lines. First, save the handles of the graphics objects created by `meshc` or `surf`:

```
h = meshc(peaks(20));
```

The first handle belongs to the mesh or surface. The remaining handles belong to the contours you want to change. To raise the contour plane, add 2 to the z coordinate of each contour line:

```
for i = 2:length(h);
    newz = get(h(i), 'Zdata') + 2;
    set(h(i), 'Zdata', newz)
end
```

Displaying Contours in Polar Coordinates

You can contour data defined in the polar coordinate system. As an example, set up a grid in polar coordinates and convert the coordinates to Cartesian coordinates:

```
[th, r] = meshgrid((0:5:360)*pi/180, 0:.05:1);
[X, Y] = pol2cart(th, r);
```

Then, generate the complex matrix Z on the interior of the unit circle:

```
Z = X+i*Y;
```

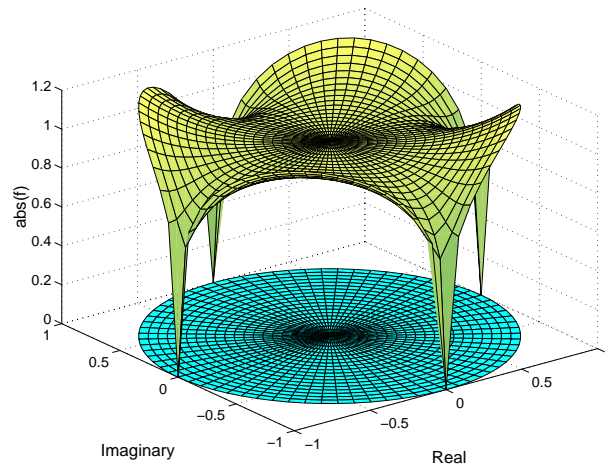
X , Y , and Z are points inside the circle.

Create and display a surface of the function $\sqrt[4]{Z^4 - 1}$:

```
f = (Z.^4-1).^(1/4);
surf(X, Y, abs(f))
```

Display the unit circle beneath the surface using the statements:

```
hold on
surf(X, Y, zeros(size(X)))
hold off
```

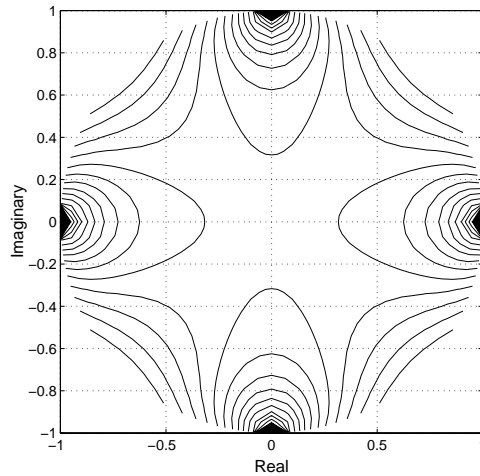


Labeling the Graph. These statements add labels to the plot,

```
xlabel('Real', 'FontSize', 14);
ylabel('Imaginary', 'FontSize', 14);
zlabel('abs(f)', 'FontSize', 14);
```

and these display a contour of this surface in Cartesian coordinates and label the x - and y -axis:

```
contour(X, Y, abs(f), 30)
axis equal
xlabel('Real', 'FontSize', 14);
ylabel('Imaginary', 'FontSize', 14);
```

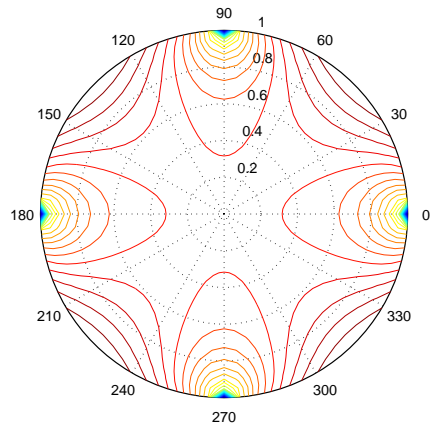


You can also display the contour within a polar axes. Create a polar axes using the `pol ar` function, and then delete the line specified with the `pol ar` function:

```
h = pol ar([0 2*pi ], [0 1]);  
delete(h)
```

With `hold on`, display the contour on the polar grid:

```
hold on  
contour(X, Y, abs(f) , 30)
```



Interactive Plotting

The `ginput` function enables you to use the mouse or the arrow keys to select points to plot. `ginput` returns the coordinates of the pointer's position; either the current position or the position when a mouse button or key is pressed. See the `ginput` function in the online MATLAB Function Reference for more information on this function.

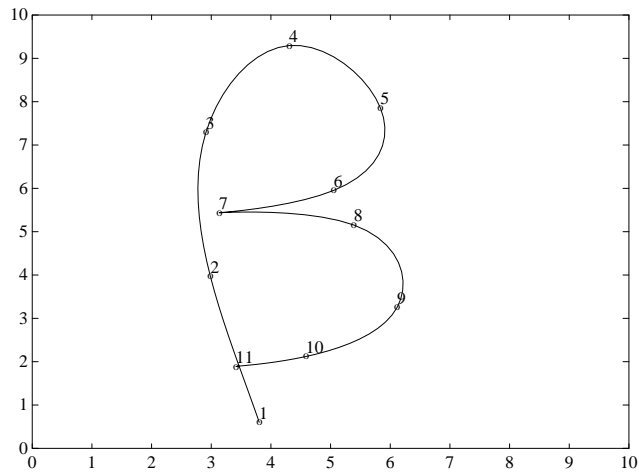
This example illustrates the use of `ginput` with the `spline` function to create a curve by interpolating in two dimensions.

First, select a sequence of points, $[x, y]$, in the plane with `ginput`. Then pass two, one-dimensional splines through the points, evaluating them with a spacing $1/10^{\text{th}}$ of the original spacing.

```
% start with a clean slate
clf
axis([0 10 0 10])
hold on
% Initially, the list of points is empty.
x = [];
y = [];
n = 0;
% Loop, picking up the points.
disp(' Left mouse button picks points. ')
disp(' Right mouse button picks last point. ')
but = 1;
while but == 1
    [xi, yi, but] = ginput(1);
    plot(xi, yi, 'go')
    n = n+1;
    x(n, 1) = xi;
    y(n, 1) = yi;
end
% Interpolate with two splines and finer spacing.
t = 1:n;
ts = 1: 0.1: n;
```

```
xs = spline(t, x, ts);  
ys = spline(t, y, ts);  
% Plot the interpolated curve.  
plot(xs, ys, 'c-');  
hold off
```

This plot shows some typical output.



Animation

You can create animated sequences with MATLAB in two different ways:

- Save a number of different pictures and then play them back as a movie.
- Continually erase and then redraw the objects on the screen, making incremental changes with each redraw.

Movies are better suited to situations where each frame is fairly complex and cannot be redrawn rapidly. You create each movie frame in advance so the original drawing time is not important during playback, which is just a matter of blitting the frame to the screen. A movie is not rendered in real-time; it is simply a playback of previously rendered frames.

The second technique, drawing, erasing, and then redrawing, makes use of different drawing modes supported by MATLAB. These modes allow faster redrawing at the expense of some rendering accuracy, so you must consider which mode to select.

This section provides an example of each technique. To see more sophisticated demonstrations of these features, type `demo` at the MATLAB prompt and explore the animation demonstrations.

Movies

You can save any sequence of plots and then play the sequence back in a short movie. There are three steps to this process:

- Use `movie` to initialize memory for a matrix large enough to hold the specified number of frames based on the size of the current axis.
- Use `getframe` to generate each movie frame, which it returns as a column vector you can then build into a movie matrix.
- Use `movie` to run the movie the specified number of times at the specified rate.

Visualizing an FFT

This example illustrates the use of movies to visualize the quantity `fft(eye(n))`, which is a complex n -by- n matrix whose elements are various powers of the n^{th} root of unity, $\exp(i * 2 * \pi / n)$.

Creating the Movie

The first step in creating a movie is to initialize the movie matrix. However, before calling `moviein`, you need to create an axes the same size as the one that will display the movie. Since this example displays data equally spaced around the unit circle, use the `axis equal` command to set the aspect ratio of the axes.

Call `moviein` to create a matrix large enough to hold the 16 frames that compose this movie. This step is not required, but if you do not initialize `M`, the code (but not the movie) runs more slowly because the storage for `M` is reallocated each time a new column is appended. Note that the `axis equal` statement must precede the `M = moviein(16);` statement to ensure MATLAB initializes `M` to the correct dimensions.

```
axis equal
M = moviein(16);
set(gca, 'NextPlot', 'replacechildren')
for j = 1:16
    plot(fft(eye(j+16)))
    M(:,j) = getframe;
end
```

The statement,

```
set(gca, 'NextPlot', 'replacechildren')
```

prevents the `plot` function from resetting the axis shaping to `axis normal` each time it is called. See the `axes` function in the online MATLAB Function Reference for more information about the `NextPlot` property.

The `getframe` function with no arguments returns a pixel snapshot of the current Axes in the current Figure. Each frame consists of byte-oriented data packed into a MATLAB column vector. The complexity of the plot does not affect the length of the column required, but the size of the current window does. Larger windows require more storage.

Note that `getframe` returns the contents of the current Axes, exclusive of the axis labels, title, or tick labels. `getframe(gcf)` captures the entire interior of the current Figure window.

If you plan to convert the MATLAB movie to another format (such as QuickTime) and you want to include the axis labels in this new format, you should capture the Figure, not just the Axes. If you are using `moviein` to pre-allocate

the movie matrix, be sure to specify the same Figure handle that you use with `getframe`.

Running the Movie

After generating the movie, you can play it back any number of times. To play it back 30 times, type

```
movie(M, 30)
```

You can readily generate and smoothly play back movies with a few dozen frames on most computers. Longer movies require large amounts of primary memory or a very effective virtual memory system.

See the `movie` function in the online MATLAB Function Reference for information on other options.

Full-Figured Movies

If you want to capture the contents of the entire Figure window (for example, to include Uicontrols in the movie), specify the Figure's handle with *both* the `moviein` and `getframe` commands. For example, suppose you want to add a slider to indicate the value of `j` in the previous example.

```
axis equal
M = moviein(16, gcf);
set(gca, 'NextPlot', 'replacechildren')
h = uicontrol('style','slider','position',...
    [100 10 500 20], 'Min', 1, 'Max', 16)
for j = 1:16
    plot(fft(eye(j+16)))
    set(h, 'Value', j)
    M(:,j) = getframe(gcf);
end
clf; axes('Position',[0 0 1 1]); movie(M, 30)
```

Erase Modes

You can select the method MATLAB uses to redraw graphics objects. One event that causes MATLAB to redraw an object is changing the properties of that object. You can take advantage of this behavior to create animated sequences. A typical scenario is to draw a graphics object, then change its position by

respecifying the x -, y -, and z -coordinate data by a small amount with each pass through a loop.

You can create different effects by selecting different erase modes. This section illustrates how to use the three modes that are useful for dynamic redrawing:

- none – MATLAB does not erase the objects when it is moved.
- background – MATLAB erases the object by redrawing it in the background color. This mode erases the object and anything below it (such as grid lines).
- xor – This mode erases only the object and is usually used for animation.

All three modes are faster (albeit less accurate) than the normal mode used by MATLAB.

Example

It is often interesting and informative to see 3-D trajectories develop in time. This example involves chaotic motion described by a nonlinear differential equation known as the Lorenz strange attractor. It can be written

in the form $\frac{dy}{dt} = Ay$

with a vector valued function $y(t)$ and a matrix A , which depends upon y :

$$A(y) = \begin{bmatrix} \frac{8}{3} & 0 & y(2) \\ 0 & -10 & 10 \\ -y(2) & 28 & -1 \end{bmatrix}$$

The solution orbits about two different attractive points without settling into a steady orbit about either. This example approximates the solution with the simplest possible numerical method – Euler's method with fixed step size. The

result is not very accurate, but it has the same qualitative behavior as other methods.

```

A = [ -8/3 0 0; 0 -10 10; 0 28 -1 ];
y = [ 35 -10 -7 ]';
h = .01;
p = plot3(y(1), y(2), y(3), ' . ', ...
          'EraseMode', 'none', 'MarkerSize', 5);
axis([0 50 -25 25 -25 25])
hold on
for i=1:4000
    A(1,3) = y(2);
    A(3,1) = -y(2);
    ydot = A*y;
    y = y + h*ydot;
    set(p, 'XData', y(1), 'YData', y(2), 'ZData', y(3))
    drawnow
    i=i+1;
end

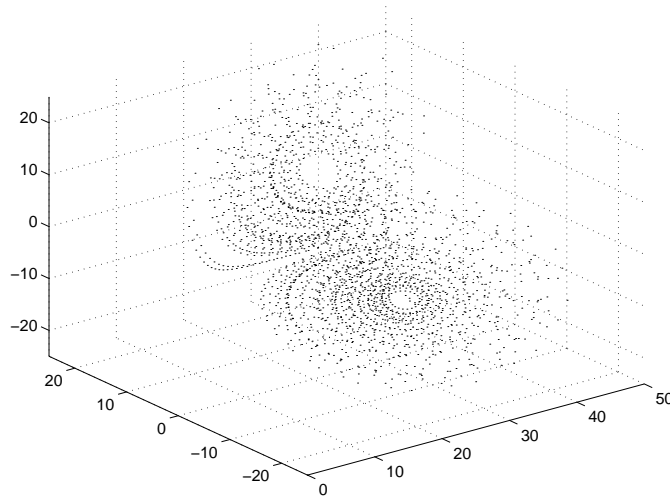
```

This **plot** statement sets the **EraseMode** to **none**.

This **set** statement moves the object by changing the coordinate data.

The **plot3** statement sets **EraseMode** to **none**, indicating that the points already plotted should not be erased when the plot is redrawn. In addition, the handle of the plot object is saved. Within the **for** loop, a **set** statement references the plot object and changes its internally stored coordinates for the new location.

While this manual cannot show the dynamically evolving output, the following picture shows a static snapshot.



Note that, as far as MATLAB is concerned, the graph created by this example contains only one dot. What you see on the screen are remnants of previous plots that MATLAB has been instructed not to erase. The only way to print this graph from MATLAB is with a screen capture. You can use the capture command to generate a MATLAB Image of the Figure window contents.

Background Erase Mode. To see the effect of `EraseMode` background, add these statements to the previous program:

This `plot` statement sets the `EraseMode` to background.

```

p = plot3(y(1), y(2), y(3), 'square', ...
    'EraseMode', 'background', 'MarkerSize', 10, ...
    'MarkerEdgeColor', [1 .7 .7], 'MarkerFaceColor', [1 .7 .7]);
for i=1: 4000
    A(1, 3) = y(2);
    A(3, 1) = -y(2);
    ydot = A*y;
    y = y + h*ydot;
    set(p, 'XData', y(1), 'YData', y(2), 'ZData', y(3))
    drawnow
    i=i+1;
end
hold off

```

Since `hold` is still on, this code erases the previously created graph by setting the `EraseMode` property to `background` and changing the marker to a “pink eraser” (a square marker colored pink).

Xor Erase Mode. If you change the `EraseMode` of the first `plot3` statement from `none` to `xor`, you will see a moving dot (Marker `'.'`) only. Xor mode is used to create animations where you do not want to leave remnants of previous graphics on the screen.

Additional Examples

The MATLAB demo, `lorenzshow`, provides a more accurate numerical approximation, and a more elaborate display of Lorenz strange attractor example. Other MATLAB demos illustrate animation techniques.

See the *Handle Graphics* chapter of this manual for more information on manipulating object properties.

See the `line` function in the online MATLAB Function Reference for a description of its `EraseMode` property.

Images

Overview	5-2
Image Types.	5-3
Indexed Images.	5-3
Intensity Images	5-3
Truecolor Images	5-4
Summary of Image Types and Display Methods	5-5
Working with 8-Bit Images	5-6
8-Bit Indexed Images	5-6
8-Bit Intensity Images	5-7
8-Bit Truecolor Images.	5-7
Summary of Image Types and Numeric Class	5-8
Other 8-Bit Array Support	5-9
Controlling Aspect Ratio and Display Size	5-10
Printing Images	5-13
The Image Object and its Properties	5-14
CData.	5-14
CDataMapping	5-14
XData and YData	5-15
EraseMode.	5-17
Reading and Writing Image Files	5-19

Overview

MATLAB provides commands for displaying several types of images, including indexed images, intensity images, and truecolor images. These commands all create a Handle Graphics Image object, whose properties can be adjusted to fine-tune the appearance of the image.

MATLAB supports two different numeric classes for image display: double-precision floating-point (`double`) and 8-bit unsigned integer (`uint8`). The image display commands interpret data values differently depending on the numeric class.

MATLAB reads and writes image data in several different graphics file formats, including TIFF, JPEG, BMP, PCX, XWD, and HDF.

Functions discussed in this chapter include:

Function	Purpose
<code>image</code>	Display image (create image object)
<code>imagesc</code>	Scale data and display as image
<code>imread</code>	Read image from graphics file
<code>imwrite</code>	Write image to graphics file
<code>iminfo</code>	Get image information from graphics file
<code>axis</code>	Plot axis scaling and appearance

See the online MATLAB Function Reference to view the entries for these functions. See also the Image Processing Toolbox, a separate product option that includes a comprehensive collection of additional image processing tools.

Image Types

In MATLAB an image consists of a data matrix and possibly a colormap matrix. There are three basic image types that differ in the way that data matrix elements are interpreted as pixel colors:

- In an indexed image, data matrix elements are interpreted as indices into the colormap matrix.
- In an intensity image, data matrix elements contain values that span a given range of intensities, typically [0, 1] or [0, 255]. Values within the range are linearly scaled to form colormap indices.
- In a truecolor image, a three-dimensional data array has dimensions m -by- n -by-3. The third dimension is used to store red, green, and blue color information for each individual pixel. No colormap is used.

Indexed Images

An indexed image consists of a data matrix, X , and a colormap matrix, map . The colormap is an m -by-3 array containing floating-point values in the range [0, 1]. Each row of map specifies the red, green, and blue components of a single color. The color of each image pixel is determined by using the corresponding value of X as an index into map . The value 1 points to the first row in map , the value 2 points to the second row, and so on. You can display an indexed image with the statements:

```
image(X); colormap(map)
```

Intensity Images

An intensity image is a data matrix, I , whose values represent intensities within some range. For double-precision data the intensity range is typically [0, 1], where 0 represents black, 1 represents white, and values in between represent intermediate shades of gray. Use the two-input form of `imagesc` to display an intensity image:

```
imagesc(I, [0 1]); colormap(gray);
```

The second input argument to `imagesc` specifies the desired intensity range. The function `imagesc` displays I by mapping the first value in the range (usually 0) to the first colormap entry, and the second value (usually 1) to the last

colormap entry. Values in between are linearly distributed throughout the remaining colormap colors.

Although it is conventional to display intensity images using a grayscale colormap, it is possible to use other colormaps. For example, the following statements display the intensity image *I* in shades of blue and green:

```
imagesc(I, [0 1]); colormap(winter)
```

To display a matrix *A* with an arbitrary range of values as an intensity image, use the single-argument form of `imagesc`. With one input argument, `imagesc` maps the minimum value of the data matrix to the first colormap entry, and maps the maximum value to the last colormap entry. For example, these two lines are equivalent:

```
imagesc(A); colormap(gray)
imagesc(A, [min(A(:)) max(A(:))]); colormap(gray)
```

Truecolor Images

A truecolor image, sometimes called an RGB image, is an m -by- n -by-3 data array that defines red, green, and blue color components for each individual pixel. Each color component is a value between 0 and 1. A pixel whose color components are (0,0,0) displays as black, and a pixel whose color components are (1,1,1) displays as white. The three color components for each pixel are stored along the third dimension of the data array. For example, if *RGB* is a truecolor image, then the red, green, and blue color components of the pixel (10,5) are stored in `RGB(10, 5, 1)`, `RGB(10, 5, 2)`, and `RGB(10, 5, 3)`, respectively.

To display the truecolor image *RGB*, use the `image` function:

```
image(RGB)
```

If MATLAB is running on a computer that does not have hardware support for truecolor image display, then MATLAB uses color approximation and dithering to display an approximation of the image. See the "Dithering Truecolor on Indexed Color Systems" section in the *Figures* chapter for more information.

Summary of Image Types and Display Methods

This table summarizes display methods for the three types of images:

Image Type	Display Commands	Uses Colormap Colors
Indexed	<code>image(X); colormap(map)</code>	Yes
Intensity	<code>imagesc(I, [0 1]); colormap(gray)</code>	Yes
Truecolor	<code>image(RGB)</code>	No

Working with 8-Bit Images

MATLAB usually works with double-precision (64-bit) floating-point numbers. However, to reduce memory requirements for working with images, MATLAB provides limited support for storing images as 8-bit unsigned integers with the numeric class `uint8`. An image whose data matrix has class `uint8` is called an 8-bit image.

The `image` function can display 8-bit images directly without converting them to double precision. However, `image` interprets matrix values slightly differently when the image matrix is `uint8`. The specific interpretation depends on the image type.

8-Bit Indexed Images

If the class of `X` is `uint8`, its values are offset by one before being used as colormap indices. The value 0 points to the first row of the colormap, the value 1 points to the second row, and so on. The `image` command automatically supplies the proper offset, so the display method is the same whether `X` is `double` or `uint8`:

```
image(X); colormap(map)
```

The colormap index offset for `uint8` data is intended to support standard graphics file formats, which typically store image data in indexed form with a 256-length colormap. The offset allows you to manipulate and display images of this form in MATLAB using the more memory-efficient `uint8` arrays.

Because of the offset, you must add 1 to convert a `uint8` indexed image to `double`. For example:

```
X64 = double(X8) + 1;
```

Conversely, subtract 1 to convert a `double` indexed image to `uint8`:

```
X8 = uint8(X64 - 1);
```

The order of operations must be as shown, because you cannot perform mathematical operations on `uint8` arrays.

8-Bit Intensity Images

The range of 8-bit intensity ranges is usually [0, 255] rather than [0, 1]. Use this command to display an 8-bit intensity image with a grayscale colormap:

```
image(I, [0 255]); colormap(map)
```

To convert an intensity image from double to uint8, first multiply by 255:

```
I8 = uint8(round(I64*255));
```

Conversely, divide by 255 after converting a uint8 intensity image to double:

```
I64 = double(I8)/255;
```

8-Bit Truecolor Images

The color components of an 8-bit truecolor image are integers in the range [0, 255] rather than floating-point values in the range [0, 1]. A pixel whose color components are (255,255,255) displays as white. The image command displays a truecolor image correctly whether its class is double or uint8:

```
image(RGB)
```

To convert a truecolor image from double to uint8, first multiply by 255:

```
RGB8 = uint8(round(RGB64*255));
```

Conversely, divide by 255 after converting a uint8 truecolor image to double:

```
RGB64 = double(RGB8)/255
```

Summary of Image Types and Numeric Class

This table summarizes the way MATLAB interprets data matrix elements as pixel colors, depending on the image type and data class:

Image Type	double Data	uint8 Data
Indexed	<p>Image is an m-by-n array of integers in the range $[1, p]$.</p> <p>Colormap is a p-by-3 array of floating-point values in the range $[0, 1]$.</p>	<p>Image is an m-by-n array of integers in the range $[0, p - 1]$.</p> <p>Colormap is a p-by-3 array of floating-point values in the range $[0, 1]$.</p>
Intensity	<p>Image is an m-by-n array of floating-point values that are linearly scaled by MATLAB to produce colormap indices. The typical range of values is $[0, 1]$.</p> <p>Colormap is a p-by-3 array of floating-point values in the range $[0, 1]$ and is typically grayscale.</p>	<p>Image is an m-by-n array of integers that are linearly scaled by MATLAB to produce colormap indices. The typical range of values is $[0, 255]$.</p> <p>Colormap is a p-by-3 array of floating-point values in the range $[0, 1]$ and is typically grayscale.</p>
Truecolor	<p>Image is an m-by-n-by-3 array of floating-point values in the range $[0, 1]$.</p>	<p>Image is an m-by-n-by-3 array of integers in the range $[0, 255]$.</p>

Other 8-Bit Array Support

In addition to image display, MATLAB supports several other operations on `uint8` arrays, including:

- Reading graphics file data into MATLAB as `uint8` arrays using `imread`
- Indexing into `uint8` arrays using standard MATLAB subscripting
- Reshaping, reordering, and concatenating arrays using the functions `reshape`, `cat`, `permute`, and the `[]` and `'` operators
- Saving and loading `uint8` arrays in MAT-files using `save` and `load`
- Locating the indices of nonzero elements in `uint8` arrays using `find`

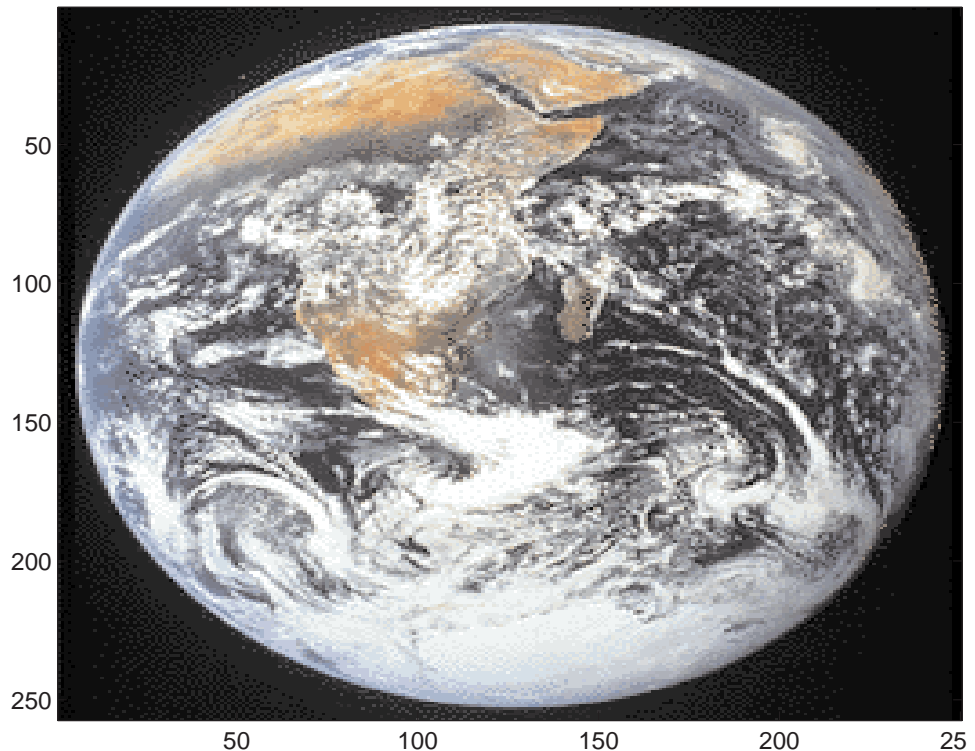
Mathematical operators and functions are not supported on `uint8` arrays. To perform any mathematical computations on a `uint8` array, first convert it to double precision using the `double` function.

Controlling Aspect Ratio and Display Size

The `image` function displays the image in a default-sized Figure and Axes. MATLAB stretches or shrinks the image to fit the display area. Sometimes you want the aspect ratio of the display to match the aspect ratio of the image data matrix. The easiest way to do this is with the command `axis image`.

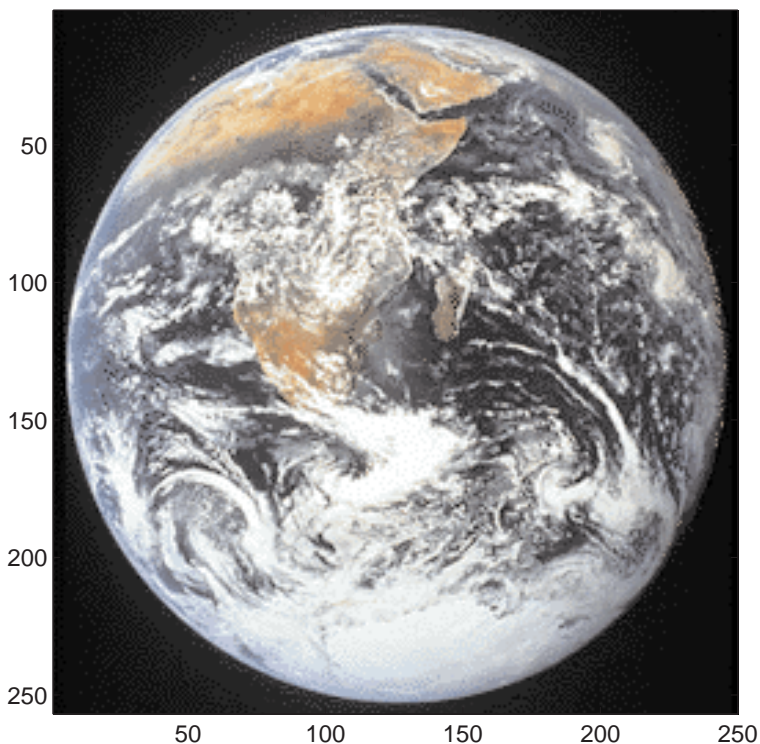
For example, these commands display the `earth` image in the `demos` directory using the default Figure and Axes positions:

```
load earth  
image(X); colormap(map)
```



The somewhat elongated globe results from stretching the image display to fit the axes position. Use the `axis image` command to force the aspect ratio to be one-to-one:

```
axis image
```



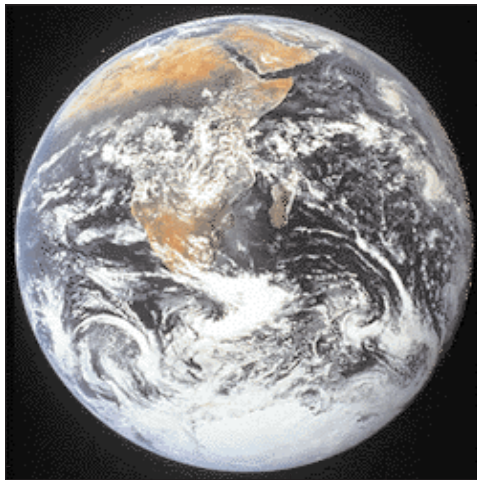
The command `axis image` works by setting the `DataAspectRatio` property of the Axes object to `[1 1 1]`. See the online MATLAB Function Reference entries for the `axis` and `axes` commands for more information on how to control the appearance of Axes objects.

Sometimes you may want to display an image so that each element in the data matrix corresponds to a single screen pixel. To display an image with this one-to-one matrix-element-to-screen-pixel mapping, you need to resize the

Figure and Axes. For example, these commands display the earth image so that one data element corresponds to one screen pixel:

```
[m, n] = size(X);  
figure('Units', 'pixels', 'Position', [100 100 n m])  
image(X); colormap(map)  
set(gca, 'Position', [0 0 1 1])
```

The Figure's `Position` property is a four-element vector that specifies the Figure's location on the screen as well as its size. The second statement above positions the Figure so that its lower-left corner is at position (100,100) on the screen and so that its width and height match the image width and height. Setting the Axes position to [0 0 1 1] in normalized units creates an Axes that fills the Figure. The resulting picture is:



Printing Images

When you set the Axes Position to `[0 0 1 1]` so that it fills the entire figure, the aspect ratio will not be preserved when you print because MATLAB adjusts the Figure size when printing according to the Figure's PaperPosition property. To preserve the image aspect ratio when printing, set the Figure's PaperPositionMode to 'auto' from the command line:

```
set(gcf, 'PaperPositionMode', 'auto')  
print
```

When PaperPositionMode is set to 'auto', the width and height of the printed Figure are determined by the Figure's dimensions on the screen, and the Figure position is adjusted to center the Figure on the page. If you want the default value of PaperPositionMode to be 'auto', enter this line in your startup.m file:

```
set(0, 'DefaultFigurePaperPositionMode', 'auto')
```

The Image Object and its Properties

The commands `image` and `imagesc` create Image objects. Image objects are children of Axes objects, as are Line, Surface, Patch, and Text objects. Like all Handle Graphics objects, the Image object has a number of properties you can set to fine-tune its appearance on the screen. The most important properties of the Image object with respect to appearance are `CData`, `CDataMapping`, `XData`, `YData`, and `EraseMode`. You can find detailed information about all the properties of the Image object in the online MATLAB Function Reference in the entry for the `image` command.

CData

The `CData` property of an Image object contains the data array. In the commands below, `h` is the handle of the Image object created by `image`, and the matrices `X` and `Y` are the same.

```
h = image(X); colormap(map)
Y = get(h, 'CData');
```

The dimensionality of the `CData` array controls whether MATLAB displays the image using colormap colors or as a truecolor image. If the `CData` array is two-dimensional, then the image is either an indexed image or an intensity image, and in either case the image is displayed using colormap colors. If, on the other hand, the `CData` array is m -by- n -by-3, then MATLAB displays it as a truecolor image, ignoring the colormap colors.

CDataMapping

The `CDataMapping` property controls whether an image is indexed or intensity. An indexed image is displayed by setting the `CDataMapping` property to `'direct'`, in which case the values of the `CData` array are used directly as indices into the Figure's colormap. When the `image` command is used with a single input argument, it sets the value of `CDataMapping` to `'direct'`:

```
h = image(X); colormap(map)
get(h, 'CDataMapping')
ans =

direct
```

Intensity images are displayed by setting the `CDataMapping` property to `'scaled'`. In this case the `CData` values are linearly scaled to form colormap indices. The scale factors are controlled by the `CLim` property. The `imagesc` function creates an Image object whose `CDataMapping` property is set to `'scaled'`, and it also adjusts the `CLim` property of the parent Axes. For example:

```
h = imagesc(I, [0 1]); colormap(map)
get(h, 'CDataMapping')
ans =

scaled

get(gca, 'CLim')
ans =

[0 1]
```

See the "Color Axis Scaling" section in the *Three-Dimensional Graphs* chapter or the "Axes Color Limits – The `CLim` Property" section in the *Axes* chapter for more information.

XData and YData

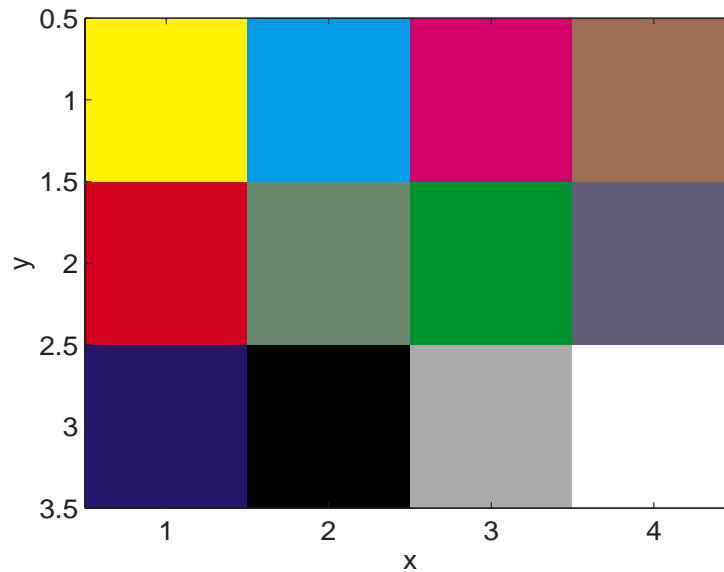
The `XData` and `YData` properties control the coordinate system of the image. For an m -by- n image, the default `XData` is `[1 n]` and the default `YData` is `[1 m]`. These settings imply the following:

- The left column of the image has an x -coordinate of 1
- The right column of the image has an x -coordinate of n
- The top row of the image has a y -coordinate of 1
- The bottom row of the image has a y -coordinate of m

For example, the statements:

```
X = [1 2 3 4; 5 6 7 8; 9 10 11 12];
h = image(X); colormap(colocube(12))
xlabel x; ylabel y
```

produce the picture:



The XData and YData properties of the resulting Image object have the default values:

```
get(h, 'XData')
ans =
```

```
1 4
```

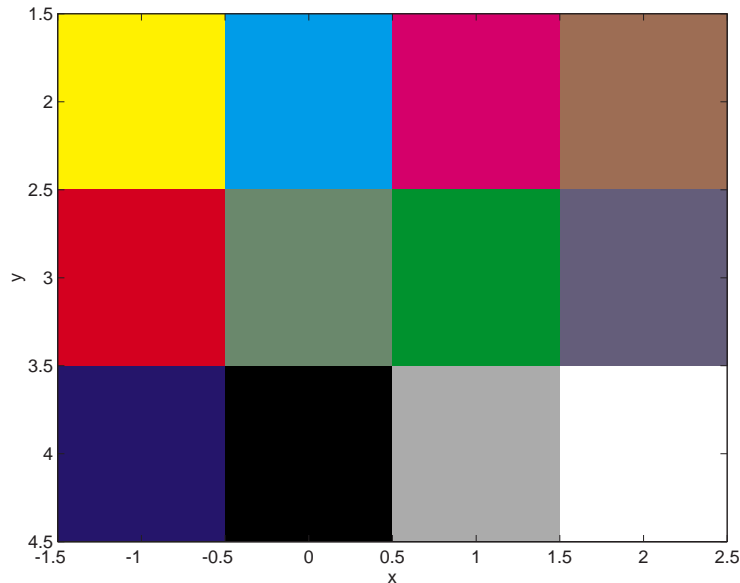
```
get(h, 'YData')
ans =
```

```
1 3
```


However, you can override the default settings to specify your own coordinate system. For example, the statements:

```
X = [1 2 3 4; 5 6 7 8; 9 10 11 12];
image(X, 'XData', [-1 2], 'YData', [2 4]); colormap(colcube(12))
xlabel x; ylabel y
```

produce the picture:



EraseMode

The `EraseMode` property controls how MATLAB updates the image on the screen if the Image object's `CData` property changes. The default setting of `EraseMode` is `'normal'`. With this setting, if you change the `CData` of the Image object using the `set` command, MATLAB erases the image on the screen before redrawing the image using the new `CData` array. The erase step is a problem if you want to display a series of images quickly and smoothly.

You can achieve fast and visually smooth updates of displayed images as you change the image `CData` by setting the Image object `EraseMode` property to `'none'`. With this setting, MATLAB does not take the time to erase the dis-

played image; rather, it immediately draws the updated image when the CData changes.

Suppose, for example, that you have an m -by- n -by-3-by- x array A , containing x different truecolor images of the same size. You can display them dynamically with:

```
h = image(A(:,:,:,1), 'EraseMode', 'none');  
for i = 2:x  
    set(h, 'CData', A(:,:,:,i))  
    drawnow  
end
```

Rather than creating a new Image object each time through the loop, this code simply changes the CData of the Image object created on the first line. The drawnow command causes MATLAB to update the display with each pass through the loop. Because the image EraseMode is set to 'none', changes to the CData do not cause the image on the screen to erase each time through the loop, resulting in faster and smoother rendering.

Reading and Writing Image Files

MATLAB provides functions for reading and writing image data from several graphics file formats, including:

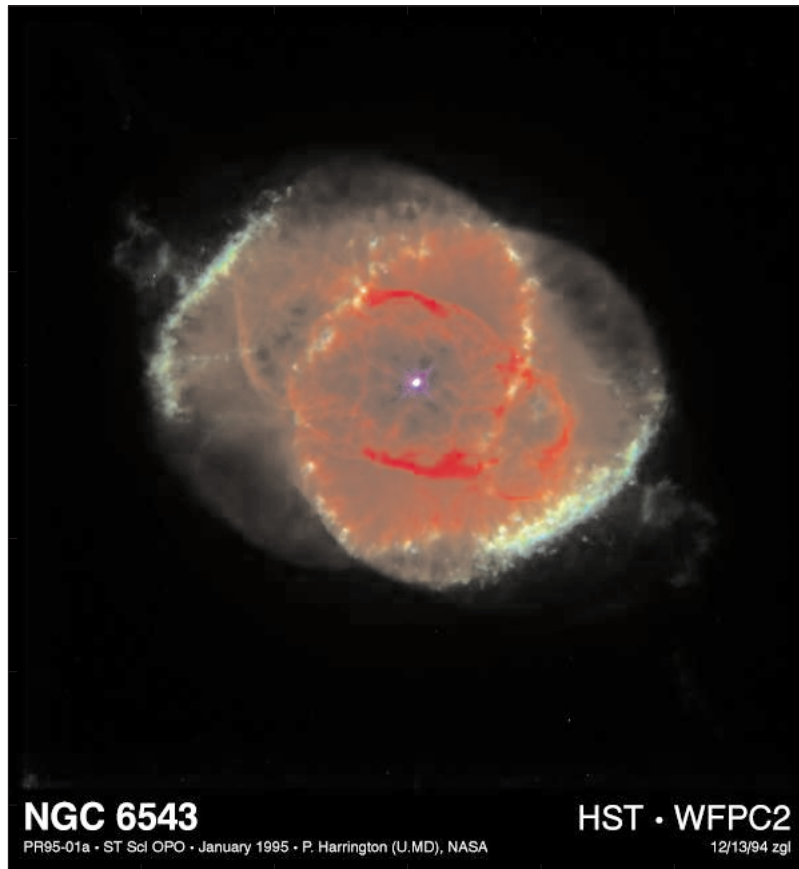
- Microsoft Windows Bitmap (BMP)
- Hierarchical Data Format (HDF)
- Joint Photographic Experts Group (JPEG)
- Paintbrush (PCX)
- Tagged Image File Format (TIFF)
- X Window Dump (XWD)

The function `imread` reads an image from a file in any of these formats. Depending on the particular format, `imread` can read indexed, intensity, or truecolor images.

- If the file contains an intensity image, `imread` reads the data into an m -by- n matrix of class `uint8`.
- If the file contains an indexed image, `imread` reads the data into an m -by- n matrix of class `uint8` and converts the associated colormap into a double-precision matrix with values in the range $[0, 1]$.
- If the file contains a truecolor image, `imread` reads the data into an m -by- n -by-3 array of class `uint8`.

This example shows how to read and display a 24-bit image from a JPEG file*:

```
RGB = imread('ngc6543a.jpg');  
figure('Position', [100 100 size(RGB, 2) size(RGB, 1)]);  
image(RGB); set(gca, 'Position', [0 0 1 1])
```



* This image was created with support to the Space Telescope Science Institute, operated by the Association of Universities for Research in Astronomy, Inc., from NASA contract NAS5-26555, and is reproduced with permission from AURA/STScI. Digital renditions of images produced by AURA/STScI are obtainable royalty-free. Credits: J.P. Harrington and K.J. Borkowski (University of Maryland), and NASA.

You can save image data using the `imwrite` function. The statements

```
load clown
imwrite(X, map, 'clown.bmp')
```

create a BMP file containing the clown image.

The function `imwrite` automatically converts double-precision data to 8-bit data in the appropriate way, depending on whether the image is indexed, intensity, or truecolor.

See the `imread` and `imwrite` entries in the online MATLAB Function Reference for more information.

Obtaining Information About Graphics Files

The `imfinfo` function enables you to obtain information about graphics files that are in any of the standard formats listed above. The information you obtain depends on the type of file, but it always includes at least the following:

- Name of the file, including the directory path if the file is not in the current directory
- File format
- Version number of the file format
- File modification date
- File size in bytes
- Image width in pixels
- Image height in pixels
- Number of bits per pixel
- Image type: truecolor, grayscale (intensity), or indexed

See the `imfinfo` entry in the online MATLAB Function Reference for more information.

3-D Modeling

Introduction to Patches.	6-2
Defining Patches	6-2
Behavior of the patch Function	6-4
 Patches with Multiple Faces.	 6-6
Example – Multifaceted Patch	6-6
 Patch Coloring.	 6-11
Face and Edge Coloring	6-12
Interpreting Color Data	6-14
Interpolating in Indexed vs. Truecolor	6-18

Introduction to Patches

A Patch graphics object is composed of one or more polygons that may or may not be connected. Patches are useful for modeling real-world objects such as airplanes or automobiles, and for drawing 2- or 3-D polygons of arbitrary shape. In contrast, Surfaces objects are rectangular grids of quadrilaterals and are better suited for displaying planar topographies such as the values of some mathematical functions, the contours of data in a rectangular plane, or parameterized surfaces such as a sphere.

There are three MATLAB functions that create Patch objects – `fill`, `fill3`, and `patch`. See the online MATLAB Function Reference for a description of these functions and examples of how to use them. This section concentrates on use of `patch` function since it provides capabilities not supported by the `fill` and `fill3` functions.

Defining Patches

You define a Patch by specifying the coordinates of its vertices and some form of color data. Patches support a variety of coloring options that are useful for visualizing data superimposed on geometric shapes.

There are two ways to specify a Patch:

- By specifying the coordinates of the vertices of each polygon, which MATLAB connects to form the Patch
- By specifying the coordinates of each *unique* vertex and a matrix that specifies how to connect these vertices to form the faces

The second technique is preferred for multifaceted Patches since it generally requires less data to define the Patch. This is because vertices shared by more than one face need be defined only once. This chapter provides examples of both techniques.

Single Polygons

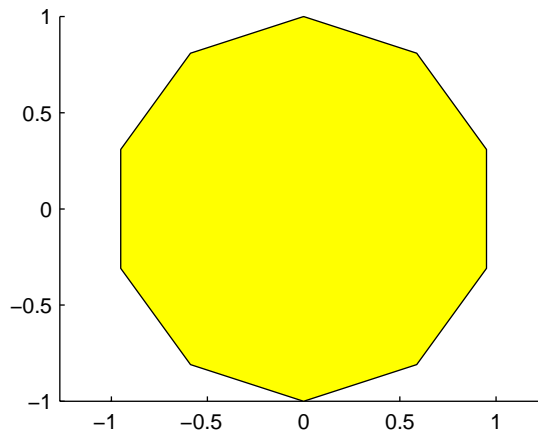
A polygon is simply a Patch with one face. To create a polygon, specify the coordinates of the vertices and color data with a statement of the form:

```
patch(x-coordinates, y-coordinates, [z-coordinates], colordata)
```


For example, these statements display a 10-sided polygon with a yellow face enclosed by a black edge:

```
t = 0:pi/5:2*pi;
patch(sin(t), cos(t), 'y')
axis equal
```

The `axis equal` command produces a correctly proportioned polygon.

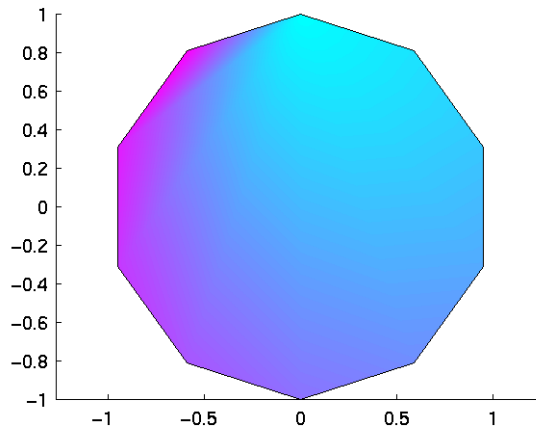


The first and last vertices need not coincide; MATLAB automatically closes each polygonal face of the Patch. In fact, it is generally better to define each vertex only once, particularly if you are using interpolated face coloring.

You can control many aspects of the Patch coloring. For example, instead of specifying a single color, you can provide a range of numerical values that map the color at each vertex to a color in the Figure colormap:

This statement removes the redundant vertex definition.

```
a = t(1:length(t)-1)
patch(sin(a), cos(a), 1:length(a), 'FaceColor', 'interp')
colormap cool;
axis equal
```



MATLAB now interpolates the colors across the face of the Patch. You can color the edges of the Patch the same way, by setting the edge colors to be interpolated. The command is:

```
patch(sin(t), cos(t), 1:length(t), 'EdgeColor', 'interp')
```

The “Patch Coloring” section provides more information on options for coloring Patches.

Behavior of the patch Function

There are two forms of the `patch` function – informal syntax and formal syntax. The behavior of the `patch` function differs somewhat depending on which syntax you use. When you use the informal syntax (as in the previous examples), MATLAB automatically determines how to color each face based on the color data you specify. The informal syntax enables you to omit the property names for the x -, y -, and z -coordinates and the color data, as long as you specify these arguments in the correct order:

Informal syntax ————— `patch(x-coordinates, y-coordinates, z-coordinates, color data)`

However, you must specify color data so MATLAB can determine what type of coloring to use. If you do not specify color data, MATLAB returns an error:

```
patch(sin(t), cos(t))
??? Error using ==> patch
Not enough input arguments.
```

The formal syntax accepts only property name/property value pairs as arguments and does not automatically color the faces unless you also change the value of the `FaceColor` property. For example, the statement

Formal syntax ————— `patch('XData', sin(t), 'YData', cos(t))`

draws a Patch with white face color because the factory default value for the `FaceColor` property is the color white:

```
get(0, 'FactoryPatchFaceColor')
ans =
     1     1     1
```

See the online MATLAB Function Reference for a description of the `patch` function and a list of Patch object properties. Also see the description of the `get` function for information on how to obtain the factory and user default values for properties.

Interpreting the Color Argument

When you use the informal syntax, MATLAB interprets the third (or fourth if there are *z*-coordinates) argument as color data. If you intend to define a Patch with *x*-, *y*-, and *z*-coordinates, but leave off the color, MATLAB interprets the *z*-coordinates as color data, and then draws a 2-D Patch. For example,

```
h = patch(sin(t), cos(t), 1:length(t))
```

draws a Patch with all vertices at *z* = 0, colored by interpolating the vertex colors (since there is one color for each vertex), whereas

```
h = patch(sin(t), cos(t), 1:length(t), 'y')
```

draws a Patch with vertices at increasing values of *z*, colored yellow.

The “Patch Coloring” section of this chapter provides more information on Patch coloring options.

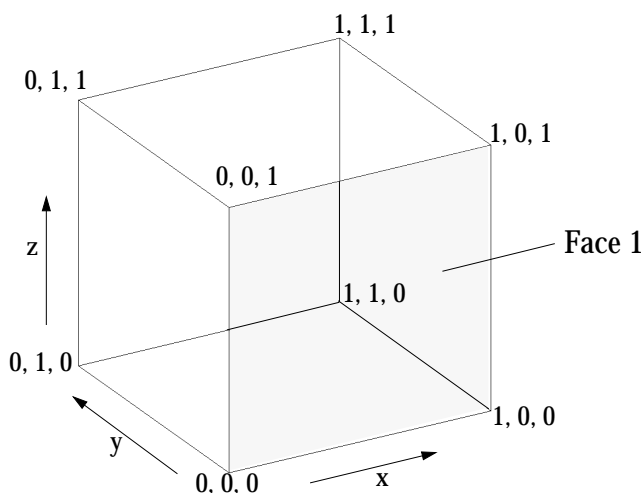
Patches with Multiple Faces

If you specify the x -, y -, and z -coordinate arguments as vectors, MATLAB draws a single polygon by connecting the points. If the arguments are matrices, MATLAB draws one polygon per column, producing a single Patch with multiple faces. These faces need not be connected and can be self intersecting.

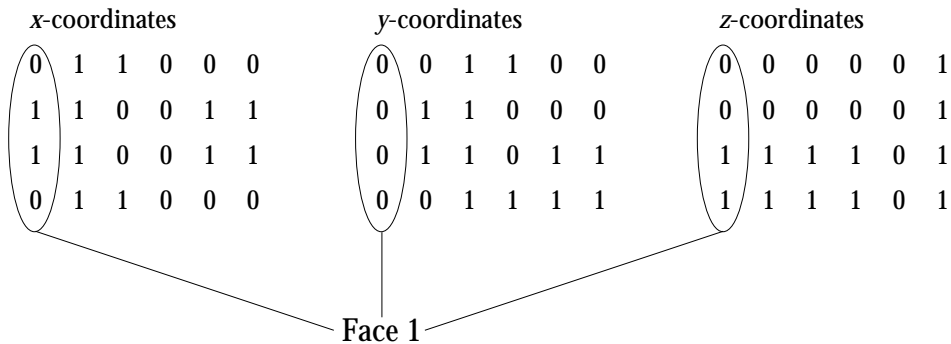
Alternatively, you can specify the coordinates of each unique vertex and the order in which to connect them to form the faces. The following example illustrates both techniques.

Example – Multifaceted Patch

A cube is defined by eight vertices that form six sides. This illustration shows the coordinates of the vertices defining a cube in which the sides are one unit in length:



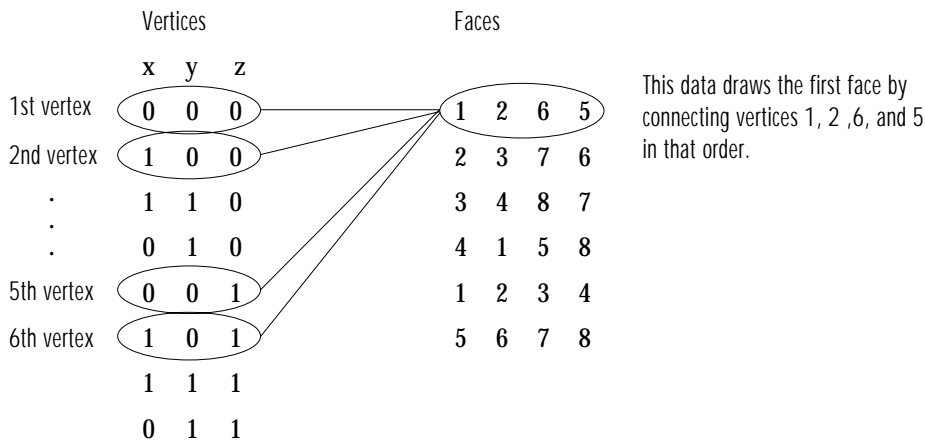
Each of the six faces has four vertices. Since you do not need to close each polygon (i.e., the first and last vertices do not need to be the same), you can define this cube using a 4-by-6 matrix for each of the x -, y -, and z -coordinates.



Each column of the matrices specifies a different face. Note that while there are only eight vertices, you must specify 24 vertices to define all six faces. Since each face shares vertices with four other faces, you can define the Patch more efficiently by defining each vertex only once and then specifying the order in which to connect these vertices to form each face. The Patch Vertices and Faces properties define Patches in just this way.

Specifying Faces and Vertices

These matrices specify the cube using Vertices and Faces:



Using the vertices/faces technique can save a considerable amount of computer memory when Patches contain a large number of faces. This technique requires

the formal patch function syntax, which entails assigning values to the Vertices and Faces properties explicitly. For example,

```
patch('Vertices', vertex_matrix, 'Faces', faces_matrix)
```

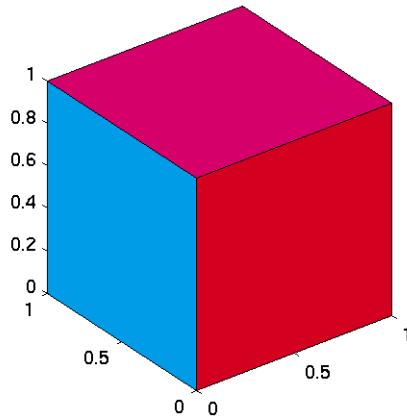
Since the formal syntax does not automatically assign face or edge colors, you must set the appropriate properties to produce Patches with colors other than the default white face color and black edge color.

Flat Face Color

Flat face color is the result of specifying one color per face. For example, using the vertices/faces technique and the FaceVertexCData property to define color, this statement specifies one color per face and sets the FaceColor property to flat:

```
patch('Vertices', vertex_matrix, 'Faces', faces_matrix, ...  
      'FaceVertexCData', hsv(6), 'FaceColor', 'flat')
```

Since true color specified with the FaceVertexCData property has the same format as a MATLAB colormap (i.e., an n -by-3 array of RGB values), this example uses the hsv colormap to generate the six colors required for flat shading.



Interpolated Face Color

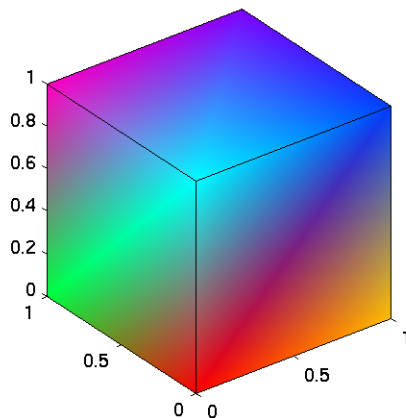
Interpolated face color means the vertex colors of each face define a transition of color from one vertex to the next. To interpolate the colors between vertices, you must specify a color for each vertex and set the `FaceColor` property to `interp`:

```
patch('Vertices', vertex_matrix, 'Faces', faces_matrix, ...
      'FaceVertexCData', hsv(8), 'FaceColor', 'interp')
```

Changing to the standard 3-D view and making the axis square,

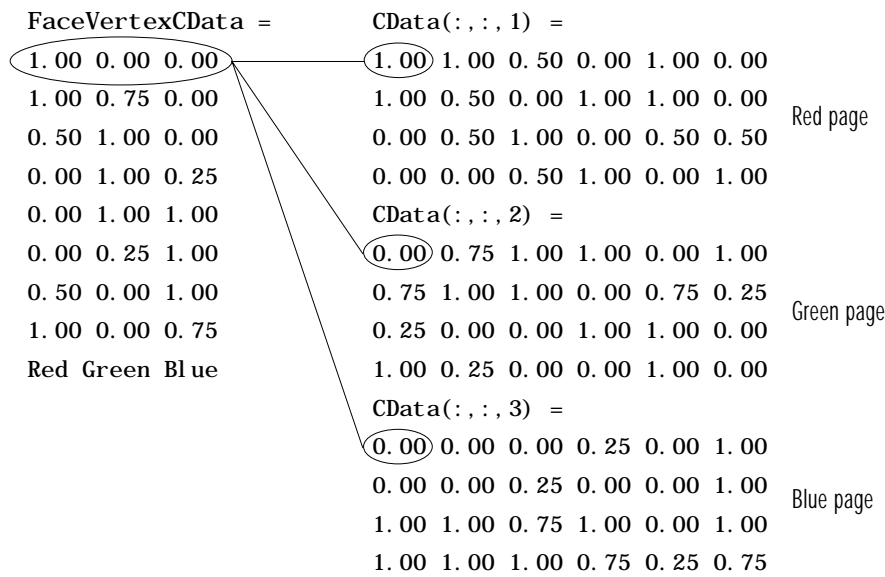
```
view(3); axis square
```

produces a cube with each face colored by interpolating the vertex colors.



To specify the same coloring using the `x, y, z, c` technique, `c` must be an m -by- n -by-3 array, where the dimensions of `x`, `y`, and `z` are m -by- n .

This diagram shows the correspondence between the FaceVertexCData and CData properties.



The next section discusses Patch coloring in more detail.

Patch Coloring

Patch objects employ a coloring scheme that is basically different from that used by Surface objects in that Patches do not automatically generate color data based on the value of the *z*-coordinate at each vertex. You must explicitly specify Patch coloring, or MATLAB uses the default white face color and black edge color.

Patch coloring methods provide a means to display pictures of real-world objects with information superimposed on them through the use of color. An example of this is an airplane wing colored so as to indicate the air pressure across its surface.

The following table summarizes the Patch properties that control color (exclusive of those used when light sources are present). See `patch` in the online MATLAB Function Reference for a more detailed discussion of Patch properties.

Property	Purpose
<code>CData</code>	Specify single, per face, or per vertex colors in conjunction with <i>x</i> , <i>y</i> , and <i>z</i> data.
<code>CDataMapping</code>	Specifies whether color data is scaled or used directly as indices into the Figure colormap.
<code>FaceVertexCData</code>	Specify single, per face, or per vertex colors in conjunction with faces and vertices data.
<code>EdgeColor</code>	Edges can be invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors.
<code>FaceColor</code>	Faces can be invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors.
<code>MarkerEdgeColor</code>	The color of the marker, or the edge color for filled markers.
<code>MarkerFaceColor</code>	The fill color for markers that are closed shapes.

See “Coloring Mesh and Surface Plots” in the Building *3-D Graphs* chapter for information on surface coloring.

Face and Edge Coloring

You can specify Patch face coloring by defining:

- A single color for all faces
- One color for each face, which is used for flat coloring
- One color for each vertex, which is used for interpolated coloring

Specify the face color using either the `CData` property, if you are using x -, y -, and z -coordinates or the `FaceVertexCData` property, if you are specifying vertices and faces.

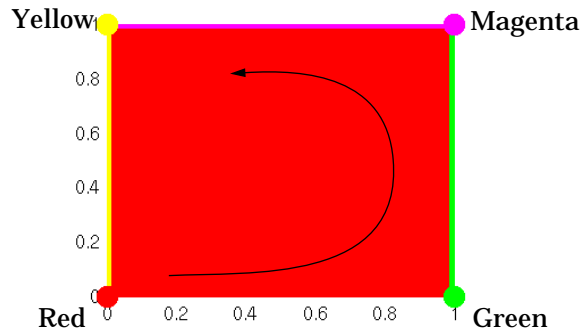
Each Patch face has a bounding edge, which you can color as:

- A single color for all edges
- A flat color defined by the color of the vertex that precedes the edge
- Interpolated colors determined by the two vertices that bound the edge

Note that Patch edge colors can be flat or interpolated only when you specify a color for each vertex. For flat edge coloring, MATLAB uses the color of the vertex preceding the edge to determine the color of the edge. The order in which you specify the vertices establishes which vertex colors a particular edge.

For example, these statements create a square Patch:

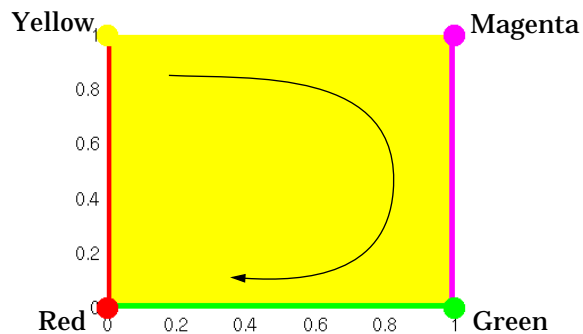
```
v = [0 0 0; 1 0 0; 1 1 0; 0 1 0];
f = [1 2 3 4];
fvc = [1 0 0; 0 1 0; 1 0 1; 1 1 0];
patch('Vertices', v, 'Faces', f, 'FaceVertexCData', fvc, ...
      'FaceColor', 'flat', 'EdgeColor', 'flat', ...
      'Marker', 'o', 'MarkerFaceColor', 'flat')
```



The Faces property value, `[1 2 3 4]`, determines the order in which MATLAB connects the vertices. In this case, the order is red, green, magenta, and yellow. If you change this order, the results can be quite different. For example, specifying the Faces property as:

```
f = [4 3 2 1];
```

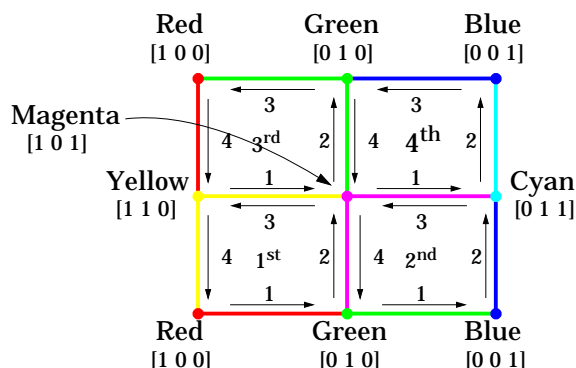
changes the order to yellow, magenta, green, and red. Note that changing the order not only changes the color of the edges, but also the color of the face, which is the color of the first vertex specified:



Shared Edges

Each Patch face is bound by edges, which are line segments that connect the vertices. When Patches have multiple faces that share vertices, some of the edges may overlap. In such cases, the edges of the most recently drawn face overlay previously drawn edges.

For example, this illustration shows a Patch with four faces and flat colored edges (FaceCol or set to none, EdgeCol or set to flat):



The arrows indicate the order each edge is drawn in the first, second, third, and fourth face. The color at each vertex determines the color of the edge that follows it. Notice how the second edge in the first face would be green except that the second face drew its fourth edge from the magenta vertex. You can see similar effects in all shared edges.

For EdgeCol or set to interp, MATLAB interpolates colors between adjacent vertices. In this case the order you specify the vertices does not affect the edge color.

Interpreting Color Data

MATLAB interprets the color data in one of two ways:

- Indexed color data – numerical values that are mapped to colors defined in the Figure colormap
- Truecolor data – RGB triples that define colors explicitly and do not make use of the Figure colormap

The dimensions of the color data (CData or FaceVertexCData) determine how MATLAB interprets it. If you specify only one numeric value per Patch, per face, or per vertex, then MATLAB interprets the data as indexed. If there are three numeric values per Patch, face, or vertex, then MATLAB interprets the data as RGB values. See the description of the CData and FaceVertexCData properties under the patch function in the online MATLAB Function Reference for more information.

Indexed Color Data

MATLAB interprets indexed color data as either values to scale before mapping to the colormap, or directly as indices into the colormap. You control the interpretation by setting the `CDataMapping` Patch property. The default is to scale the data.

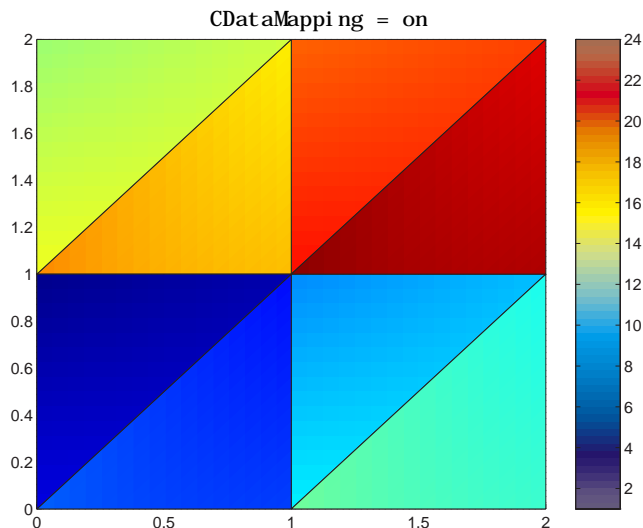
Scaled Color. By default, MATLAB scales the color data so that the minimum value maps to the first color in the colormap, the maximum value maps to the last color in the colormap, and values in between are linearly transformed to span the colormap. This enables you to use colormaps of different sizes without changing your data and to use data in any range of values without changing the colormap.

For example, the following Patch has eight triangular faces with a total of 24 (nonunique) vertices. The color data are integers that range from one to 24, but could be any values.

The variable `c` contains the color data. It is a 3-by-8 matrix, with each column specifying the colors for the three vertices of each face.

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

The color bar on the right side of the Patch illustrates the colormap used and indicates with the vertical axis which color is mapped to the respective data value:



The `Colorbar` function displays the colormap to the right of the Patch.

See the `caxis` command in the online MATLAB Function Reference for more information.

You can alter the mapping of color data to colormap entry using the `caxis` command. This command uses a two-element vector `[cmin cmax]` to specify what data values map to the beginning and end of the colormap, thereby shifting the color mapping.

By default, MATLAB sets `cmin` to the minimum value and `cmax` to the maximum value of the color data of all graphics objects within the Axes. However, you can set these limits to span any range of values and thereby shift the color mapping. See the “Calculating Color Limits” section in the *Axes* chapter for more information.

The color data does not need to be a sequential list of integers; it can be any matrix with dimensions matching the coordinate data. For example,

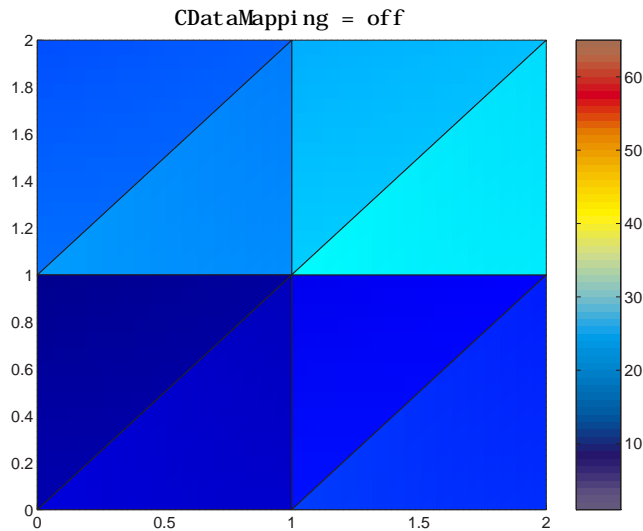
```
patch(x, y, z, rand(size(z)))
```

Direct Color. If you set the Patch `CDataMapping` property to `off`,

```
set(patch_handle, 'CDataMapping', 'off')
```

MATLAB interprets each color data value as a direct index into the colormap. That is, a value of 1 maps to the first color, a value of 2 maps to the second color, and so on.

The Patch from the previous example would then use only the first 24 colors in the colormap.



This example uses integer color data. However, if the values are not integers, MATLAB converts them according to these rules:

- If value is < 1 , it maps to the first color in the colormap.
- If value is not an integer, it is rounded to the nearest integer towards zero.
- If value $> \text{length}(\text{colormap})$, it maps to the last color in the colormap.

Unscaled color data is more commonly used for images where there is typically a colormap associated with a particular image.

Truecolor Patches

Truecolor is a means to specify a color explicitly with RGB values rather than pointing to an entry in the Figure colormap. Truecolor generally provides a greater range of colors than can be defined in a colormap.

Using truecolor eliminates the mapping of data to colormap entries. On the other hand, you cannot change the coloring of the Patch without redefining the color data (as opposed to just changing the colormap).

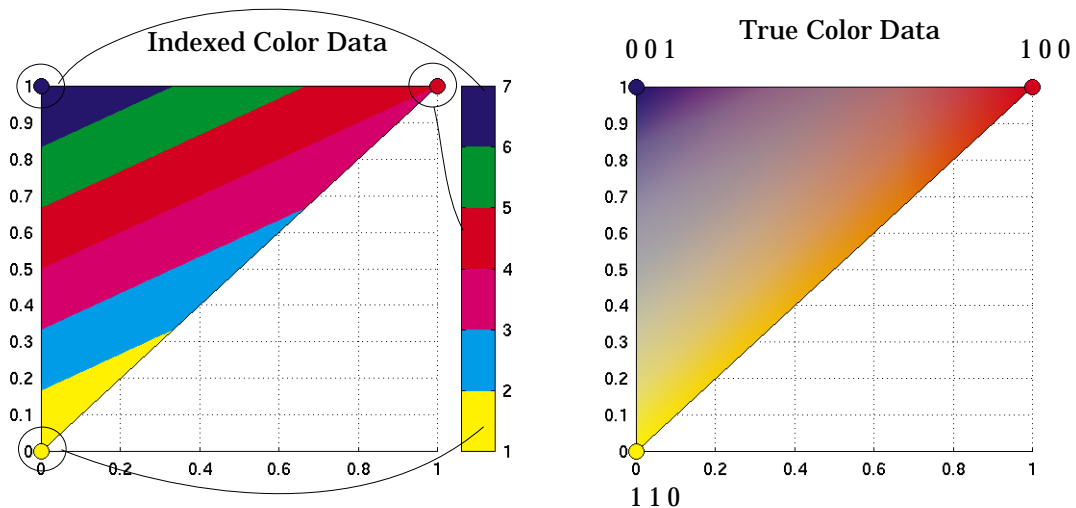
You can use truecolor on computers that do not have true color (24-bit) displays. In this case, MATLAB uses a special colormap designed to produce results that are as close as possible with the limited number of colors available. See the “Properties That Control Colors on Pseudocolor Displays” section in the *Figure* chapter for more information.

Interpolating in Indexed vs. Truecolor

When you specify interpolated face coloring, MATLAB determines the color of each face by interpolating the vertex colors. The method of interpolation depends on whether you specified truecolor data or indexed color data.

With truecolor data, MATLAB interpolates the numeric RGB values defined for the vertices. This generally produces a smooth variation of color across the face. In contrast, indexed color interpolation uses only colors that are defined in the colormap. With certain colormaps, the results can be quite different.

To illustrate this difference, these two Patches are defined with the same vertex colors. Circular markers indicate the yellow, red, and blue vertex colors.



The Patch on the left uses indexed colors obtained from the six-element colormap shown next to it. The color data maps the vertex colors to the colormap elements indicated in the picture. With this colormap, interpolating

from the cyan vertex to the blue vertex can include only the colors green, red, yellow, and magenta, hence the banding.

Interpolation in RGB space makes no use of the colormap. It is simply the gradual transition from one numeric value to another. For example, interpolating from the cyan vertex to the blue vertex follows a progression similar to these values:

0 1 1, 0 0.9 1, 0 0.8 1, ... 0 0.2 1, 0 0.1 1, 0 0 1

In reality each pixel would be a different color so the incremental change would be much smaller.

Printing MATLAB Graphics

Introduction.	7-2
Printing from the Menu.	7-3
Printing from the Command Line	7-6
Selecting a Device Driver	7-17
Printing Tips and Troubleshooting	7-25
Using MATLAB Graphics in Other Applications	7-37

Introduction

MATLAB provides a number of different methods for producing graphical output from Figures. These methods include ways to:

- Print from the menu or print from the command line
- Use MATLAB's built-in print engine or use system-specific printing services
- Print directly to hardcopy or create graphics-format files to incorporate in documents for other applications
- Create M-files to reproduce Figures in MATLAB

The method you use depends on what you want to accomplish. For example, the simplest way to produce output is to choose the **Print** option from the **File** menu, but if you want to print from an M-file, you need to use the `print` command. If you want to produce graphics to use in other applications, there are many options, depending on your platform and the file format you want to use. This chapter discusses all of these methods and provides guidelines for choosing among them.

Printing from the Menu

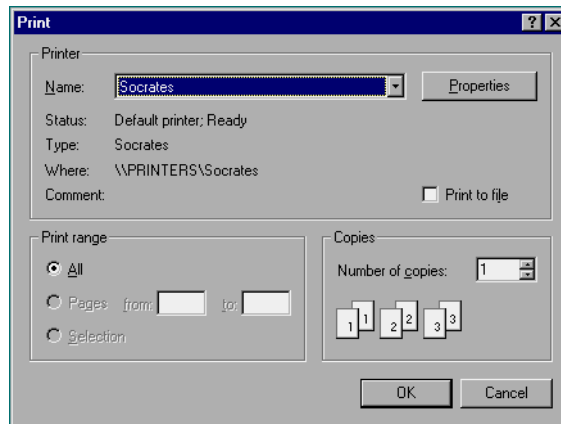
This section discusses how to print a Figure by selecting options from the **File** menu of the Figure window. This is the simplest way to print in MATLAB and is the preferred method in many cases. More detailed information about various aspects of printing is available in other sections of this chapter.

Printing differs depending on the platform you are running MATLAB on. Read the section corresponding to the platform you are using. Also see page 7-5 for information about adjusting the size and location of the graphic on the page.

PC



To print a Figure, choose the **Print** option from the Figure window's **File** menu. MATLAB brings up the Windows print dialog box:

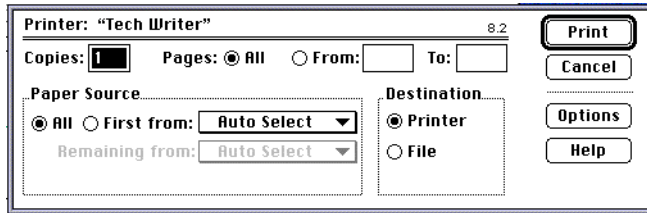


Fill in the dialog box, and then click the **OK** button to print the Figure.

Macintosh



To print a Figure, make it the current window, and then choose the **Print** option from the Macintosh **File** menu. MATLAB brings up the Macintosh print dialog box:

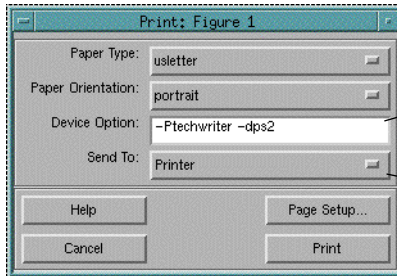


Fill in the dialog box, and then click the **Print** button to print the Figure.

UNIX



To print a Figure, choose the **Print** option from the Figure window's **File** menu. MATLAB brings up the print dialog box:



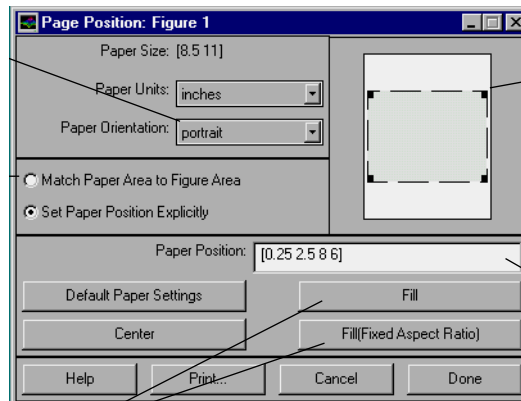
This field specifies the name of the printer and the device type. (See the discussion of printing from the command line for more information.)

Use this popup to choose whether to direct the output to a printer or a file.

Fill in the dialog box, and then click the button in the lower right corner of the box. This button is labeled **Print** (if you are sending the output to a printer) or **Save** (if you are sending the output to a file). If you are saving to a file, MATLAB displays another dialog box where you specify the filename.

Adjusting the Size and Location of the Graphic

To adjust the size and location of the printed graphic, choose the **Page Position** option from the Figure window's **File** menu. MATLAB displays this dialog box:



This dashed box represents the position of the Figure on the printed page. The white box represents the page. Position the Figure on the page by dragging the dashed box around with the mouse. To resize the Figure, click on one of the corners and drag.

Click the **Fill** button to resize the box to fill the page, or click the **Fill (Fixed Aspect Ratio)** button to fill as much of the page as is possible without changing the aspect ratio.

As you resize or position the dashed box, the Paper Position field updates automatically. You can also edit this field directly, and the box will move automatically.

When you are satisfied with the position of the Figure, click the **Print...** button. MATLAB brings up the print dialog box, as shown above.

Printing from the Command Line

This section discusses how to print a graphic using MATLAB's `print` command. This section discusses:

- The `print` command
- Options for modifying the behavior of `print`

When you print from the command line, the output is controlled by options to the `print` command and the values of Handle Graphics properties. For information about ways to control the printed output, see page 7-25.

The `print` Command

To print from the MATLAB command line, you use the `print` command and specify the appropriate device type. The syntax of the `print` command is:

```
print -device type -options
```

If you do not specify a device type, MATLAB uses the default device for your system. For example, these commands plot a sine function and print the resulting Figure on your default printer:

```
x = -pi : 0.1 : pi ;
plot(x, sin(x))
print
```

To send the output to a file rather than to a printer, the syntax is:

```
print -device type -options filename
```

For example, the following command creates an Encapsulated PostScript file from the current Figure:

```
print -deps fig1.eps
```

Changing Default Settings

The `print` command obtains default settings by calling the `printopt` function. You or your system manager can change these default values by editing the file `printopt.m`, which is found in the `toolbox/local` directory. If you are working on a multiuser system, you can make a copy of `printopt.m` and place it on your search path ahead of the MATLAB version.

The syntax for `printopt` is:

```
[pcmd, dev] = printopt
```

`pcmd` and `dev` are strings representing the operating-system command for printing and the default device type for your platform. The printing command is the actual operating-system command that MATLAB invokes after it creates the temporary file. The device type is the MATLAB command-line switch used to specify the type of device to format the output for. (If you specify a device in the `print` command, `dev` is ignored.)

This table shows the default values for `pcmd` and `dev` on each platform:

Platform	pcmd	dev
MS-Windows	<code>COPY /B %s LPT1:</code>	<code>-dwi n</code>
Macintosh	(not applicable)	<code>-dps2</code>
UNIX (except Silicon Graphics)	<code>lpr -r -s</code>	<code>-dps2</code>
Silicon Graphics	<code>lp</code>	<code>-dps2</code>

As the table shows, the default value for `dev` on most platforms is `-dps2`, which means MATLAB produces black and white Level 2 PostScript. On Windows systems, the default value for `dev` is `-dwi n`, which specifies printing through the Windows Print Manager.

Editing `printopt.m`. If you want to edit `printopt.m` to change the value of `pcmd` or `dev`, enter the command:

```
edit printopt
```

This command opens your text editor with the `printopt.m` file. Scroll down about 40 lines until you come to this comment line:

```
%--> Put your own changes to the defaults here (if needed)
```

On the line below this, enter the values you want to use. For example, this line sets the default device type to Level 2 color PostScript:

```
dev = '-dpvc2';
```

Built-in Device Drivers

When you enter a `print` command, MATLAB uses the device type returned by `printopt`. You can override the default by specifying a different device with a command-line switch.

The set of devices you can specify varies depending on your system. All systems support a core set of built-in device drivers. For information about additional devices for PC or Macintosh systems, see page 7-21 or page 7-24.

MATLAB has built-in drivers for these device types:

- PostScript
- Hewlett-Packard Graphics Language (HPGL)
- Adobe Illustrator 88

In addition, MATLAB has a built-in driver for saving a Figure to an M-file so it can be reloaded later.

This table summarizes the command-line switches for MATLAB's built-in device drivers:

Device	Description
-dps	Level 1 black and white PostScript
-dpssc	Level 1 color PostScript
-dps2	Level 2 black and white PostScript
-dpssc2	Level 2 color PostScript
-deps	Level 1 black and white Encapsulated PostScript (EPS)

Device	Description
-depsc	Level 1 color Encapsulated PostScript (EPS)
-deps2	Level 2 black and white Encapsulated PostScript (EPS)
-depsc2	Level 2 color Encapsulated PostScript (EPS)
-dhpgl	HPGL compatible with HP 7475A plotter
-di11	Adobe Illustrator 88 compatible illustration file
-dmfile	M-file, and MAT-file when appropriate, containing Handle Graphics commands to recreate the Figure

Ghostscript Device Drivers

On Windows and UNIX systems, The MathWorks distributes with MATLAB a program called Ghostscript. Ghostscript is optionally used by MATLAB's `print` command to provide support for a variety of output devices that are not supported by MATLAB's built-in drivers. When you use a Ghostscript device, MATLAB generates a Level 1 PostScript file (either color or black and white, depending on the Ghostscript device), and then calls the appropriate Ghostscript driver, which converts the output to the specified format. This output is then saved to the filename you specify in the `print` command, or sent to the printer (if you do not specify a filename).

Ghostscript is copyrighted by Aladdin Enterprises and is provided under the terms of the Free Software Foundation's GNU General Public License. This license allows you to make and distribute copies of the Ghostscript files provided with MATLAB, namely the executable file `gs` and all other files found in the `ghostscript` directory, provided that you comply with the terms of the GNU General Public License. You will find a copy of this license in the file `gsrights`, which is part of the MATLAB distribution. This file, and the rights described therein, do *not* apply to the whole of, or any other part of, the MATLAB, Simulink, or toolbox programs.

The MathWorks will provide you with source code for Ghostscript if you so request. Ghostscript (including source code) is also available directly from the Free Software Foundation and from many sources on the Internet.

This table summarizes the Ghostscript device drivers provided with MATLAB:

Device	Description
-dl aserj et	HP LaserJet
-dl j etpl us	HP LaserJet+
-dl j et2p	HP LaserJet IIP
-dl j et3	HP LaserJet III
-dl j et4	HP LaserJet 4 (defaults to 600 dpi)
-ddeskj et	HP DeskJet and DeskJet Plus
-ddj et500	HP Deskjet 500
-dcdeskj et	HP DeskJet 500C with 1 bit/pixel color
-dcdj mono	HP DeskJet 500C printing black only
-dcdj col or	HP DeskJet 500C with 24 bit/pixel color and high-quality color (Floyd-Steinberg) dithering
-dcdj 500	HP DeskJet 500C
-dcdj 550	HP Deskjet 550C
-dpai ntj et	HP PaintJet color printer
-dpj xl	HP PaintJet XL color printer
-dpj etxl	HP PaintJet XL color printer
-dpj xl 300	HP PaintJet XL300 color printer
-ddnj 650c	HP DesignJet 650C
-dbj 10e	Canon BubbleJet BJ10e
-dbj 200	Canon BubbleJet BJ200
-dbj c600	Canon Color BubbleJet BJC-600 and BJC-4000
-dl n03	DEC LN03 printer

Device	Description
-depson	Epson-compatible dot matrix printers (9- or 24-pin)
-depsonc	Epson LQ-2550 and Fujitsu 3400/2400/1200
-deps9hi gh	Epson-compatible 9-pin, interleaved lines (triple resolution)
-di bmpro	IBM 9-pin Proprinter
-dbmp256	8-bit (256-color) BMP file format
-dbmp16m	24-bit BMP file format
-dpcxmono	Monochrome PCX file format
-dpcx16	Older color PCX file format (EGA/VGA, 16-color)
-dpcx256	Newer color PCX file format (256-color)
-dpcx24b	24-bit color PCX file format, three 8-bit planes
-dpbm	Portable Bitmap (plain format)
-dpbmraw	Portable Bitmap (raw format)
-dpgm	Portable Graymap (plain format)
-dpgmraw	Portable Graymap (raw format)
-dppm	Portable Pixmap (plain format)
-dppmraw	Portable Pixmap (raw format)

Options

The `print` command accepts a number of different options that control various aspects of the output. Some of these options are valid only with certain drivers or on certain platforms.

This table summarizes the available printing options. They are discussed in detail below.

Option	Description
<code>-epsi</code>	Add 1-bit deep EPSI preview to Encapsulated PostScript
<code>-l oose</code>	Use loose bounding box for Encapsulated PostScript
<code>-cmyk</code>	Use CMYK colors in PostScript instead of RGB
<code>-append</code>	Append to existing PostScript file without overwriting
<code>-r number</code>	Specify resolution in dots per inch
<code>-adobecset</code>	Use PostScript default character set encoding
<code>-Pprinter</code>	Specify printer to use
<code>-f handle</code>	Specify handle of Figure to print
<code>-s window title</code>	Specify name of Simulink system window to print
<code>-painters</code>	Render using painter's algorithm
<code>-zbuffer</code>	Render using Z-buffer
<code>-noui</code>	Suppress printing of user interface controls

Specifying the Figure to Print (`-f`, `-s`)

By default, MATLAB takes the current Figure (i.e., the value returned by `gcf`) as the window to print. To print a Figure other than the current Figure, use the `-f` option. Note that you must use this option if the Figure's handle is hidden (i.e., the `HandleVisibility` property is set to `off`).

The syntax is:

```
print -f handle
```

For example, this command prints the Figure whose handle is 2, regardless of which Figure is the current Figure:

```
print -f2
```

The handle of a Figure corresponds to the title of the window, so in the example above, MATLAB prints the Figure in the window titled “Figure No. 2.”

You can also pass the handle as a variable to the function form of `print`. For example:

```
h = figure; plot(1:4, 5:8)
print(h)
```

To print the block diagram displayed in a Simulink window, use the `-s` option. The syntax is:

```
print -s windowtitle
```

For example, this command prints the Simulink window titled “f14”:

```
print -sf14
```

If the window title includes any spaces, you can call the function form rather than the command form of `print`. For example, this command prints a Simulink window title “Thruster Control” to a file named `thrstcon.ps`, using the MATLAB Level 1 black and white PostScript driver:

```
print(' -sThruster Control ', '-dps', 'thrstcon.ps')
```

You can omit the window title if you want to print the current system. Just use:

```
print -s
```

For information about issues specific to printing Simulink windows, see the Simulink documentation.

Specifying the Printer to Use (-P)

X

In general, MATLAB sends the output from a `print` command to the default printer on your system. If you want to send the output to a different printer, you can use the `-P` option. The syntax is:

```
print -Pprintername
```

For example, this command sends the output to a printer named “homer”:

```
print -Phomer
```

This option does not work on MS-Windows or Macintosh systems. On these systems, MATLAB prints to whatever printer you have set as your default.

Print Preview Images for EPS (-epsi, -loose)

When you create an Encapsulated PostScript (EPS) file on a Macintosh system, MATLAB automatically creates a print preview image for the file. This preview image is Macintosh-specific and does not display on other platforms.

On platforms other than the Macintosh, MATLAB does not automatically include a preview image with EPS files. When you import an EPS file without a preview image into a file from another application, the image prints properly but appears on screen as a gray box.

If you want to include a preview image with the EPS file, use the `-epsi` option. When you use this option, MATLAB creates a preview image in Encapsulated PostScript Interchange (EPSI) format.

For example, this command creates an EPS file named `figure1.eps` that includes a preview image:

```
print -deps -epsi figure1
```

When you enter this command, MATLAB redraws the Figure on screen in order to capture the preview image.

Note that the EPSI preview image is a black and white bitmap, regardless of whether the actual PostScript image is color, grayscale, or black and white. Also note that this image will be visible only within an application that recognizes EPSI previews.

Placement of Preview Image. The size and placement of the bitmap on the page of the printed document may not exactly match its appearance on screen, because the bitmap includes some white space around the Figure but the EPS itself does not. If you need the screen placement to match the printed document, use the `-loose` option. This option instructs MATLAB to create the EPS with a loose bounding box (that is, including white space around the Figure) to match the preview.

CMYK Color Separations (-cmyk)

By default, MATLAB produces color output based upon red, green, blue (RGB) color values. If you plan to publish MATLAB Figures using four-color separations, you may want to use cyan, magenta, yellow, black (CMYK) color values rather than RGB.

The `-cmk` option automatically converts RGB values to CMYK values. This option applies only to the PostScript and Encapsulated PostScript drivers. When you print the Figure, the PostScript interpreter that renders the file must include the CMYK Extension Set. This set is available on all color Level 1 PostScript printers, most newer black and white Level 1 PostScript printers, and all Level 2 PostScript printers.

Appending to an Existing File (`-append`)

To include more than one Figure in a single output file, print the first Figure to a file, and then for subsequent files use the `-append` option and specify the same file. For example, these commands create a file named `fi gs. ps`, which contains two different Figures:

```
print -dps -f1 fi gs
print -dps -f2 -append fi gs
```

When you print the resulting file, each Figure will appear on a separate page.

The `-append` option is not valid for Encapsulated PostScript files.

Specifying Resolution (`-r`)

When you print a Figure rendered using Z-buffer, you can specify the resolution of the output. The default resolution is 72 dpi on the Macintosh, and 150 dpi on other platforms. To specify a different resolution, use the `-r` option. The syntax for this option is:

```
print -r $number$ 
```

For example, this command prints the current Figure at 300 dpi:

```
print -r300
```

If *number* is 0, MATLAB prints the Figure at screen resolution. (On most systems, screen resolution is between 72 and 100 dpi.)

For more information about resolution and its relationship to the rendering method used, see page 7-32.

Default Character-Set Encoding (`-adobecset`)

Some early PostScript Level 1 printers do not support the PostScript operator `ISOlatin1Encoding` that is used in MATLAB PostScript files generated on UNIX and Windows. If your printer does not support this operator, you may

notice problems in the text of MATLAB printouts. If this happens, use the `-adobecset` option to specify default character-set encoding. This encoding is supported by all PostScript printers.

On the Macintosh, MATLAB uses the Macintosh Standard Roman character set for both screen display and printing. It does not use the `ISOLatin1Encoding` operator, and therefore does not have these problems.

Selecting a Device Driver

This section provides information to help you select the device driver to use. This section discusses using MATLAB's built in drivers for PostScript, HPGL, and Adobe Illustrator, as well as system-specific drivers on the PC and the Macintosh.

PostScript

MATLAB has several built-in drivers for generating PostScript output. When you select a PostScript driver, you can choose among these options:

- PostScript Level 1 or Level 2
- Black and white or color
- PostScript or Encapsulated PostScript

For example, if you want to create a Level 2 color Encapsulated Postscript file, use the `-depsc2` switch.

Level 1 or Level 2

Level 2 PostScript files generally are smaller and render more quickly than Level 1 files, so if your printer supports Level 2 PostScript, you should use one of the Level 2 drivers. If your printer does not support Level 2, or if you're not sure, use a Level 1 driver. Level 1 PostScript will produce good results on a Level 2 printer, but Level 2 PostScript will not print properly on a Level 1 printer.

Black and White or Color

If you are using a color printer, you should select a color driver. If you are using a black and white printer, you can use either a color driver or a black and white driver; however, a black and white driver will produce smaller output files and will render lines and text better. (Note that black and white drivers produce grayscale output. You do not need to use a color driver to produce different shades of gray.)

See page 7-35 for more information about color and grayscale printing.

PostScript or Encapsulated PostScript

The type of PostScript device you select depends on whether you want to print the file directly or import it into another application (such as a word processing program). If you want to send the output directly to a printer, or save it to a file and then send that file to the printer, use a regular PostScript driver. If you want to import the output into another application, use an Encapsulated PostScript (EPS) driver.

If you select a regular PostScript driver, you can provide a filename (in which case MATLAB creates an output file but does not send it to the printer) or you can omit the filename (in which case MATLAB sends the output to the printer and deletes the temporary file it creates).

If you select an EPS driver, MATLAB always creates a file; MATLAB does not print EPS directly. If you do not specify a filename, MATLAB creates a file named after the Figure window used to create the file. For example, if the current Figure window is titled “Figure 2,” and you enter this command:

```
print -deps
```

MATLAB displays this message:

```
Encapsulated PostScript files cannot be sent to the printer.  
File saved to disk under name 'figure2.eps'
```

HPGL Compatible Plotters (-dhpgl)

MATLAB provides HPGL support for the HP 7475A plotter and other plotters that are fully compatible with the HP 7475A. To specify the HPGL format, use the `-dhpgl` option.

If you specify this option and do not provide a filename, MATLAB sends the output directly to the plotter. If you provide a filename, the `print` command creates a file called `filename.hgl` for later output to a plotter. HPGL files can also be imported into documents of other applications, such as Microsoft Word.

When plotting a Figure, it is especially important that the background color be white, because this driver does not do background fills. If the background color is black, make sure the value of the `InvertHardCopy` property is on. When this property is on, MATLAB inverts the colors of the Figure for printing, so that black backgrounds print as white.

Color Selection

The HP 7475A plotter supports six pens, none of which can be white. If MATLAB tries to draw in white while rendering in HPGL mode, the driver ignores all drawing commands until a different color is chosen.

Pen 1 is assumed to be black, and is used for drawing axes. The remaining colors are the first five colors in the `ColorOrder` property of the current Axes object. If `ColorOrder` specifies fewer than five colors, the unspecified pens are not used.

For Simlunk systems, which ordinarily use a maximum of eight colors, the six pens available on the plotter are assumed to be:

- Pen 1: black
- Pen 2: red
- Pen 3: green
- Pen 4: blue
- Pen 5: cyan
- Pen 6: magenta

If you attempt to draw a MATLAB object containing a color that is not a known pen color, the driver chooses the nearest approximation to the unlisted color.

Limitations

The HPGL driver has these limitations:

- Display colors and plotted colors sometimes differ.
- Areas (faces on mesh and surface plots, patches, blocks, and arrowheads) are not filled.
- There is no hidden line or surface removal.
- Text is printed in the plotter's default font.
- Line width is determined by pen width.
- Images and Uicontrols cannot be plotted.
- Interpolated edge lines between two vertices are drawn with the pen whose color best matches the average color of the two vertices.
- Figures cannot be rendered using Z-buffer; this driver always uses painter's algorithm. (See page 7-32 for more information.)

Adobe Illustrator 88 (-dill)

MATLAB provides the capability to generate illustrations that can be viewed and modified by Adobe Illustrator 88 or any other application that supports a compatible file format. Regardless of where an illustration was initially created, the MATLAB output file can be further processed with Illustrator running on any platform.

By default these illustrations are always in color and appear in portrait orientation. The Illustrator group command is used to give the illustrations a hierarchy similar to that of the Handle Graphics or Simulink graphic represented.

Creating Adobe Illustrator 88 files

The syntax of the command is:

```
print -dill filename
```

If you do not provide a filename, MATLAB gives the file a default name based on the Figure window used to create the file.

To view the output, open the saved file within Illustrator. It will have no template.

Limitations

The Illustrator driver has these limitations:

- Interpolated patches and surfaces cannot be created. The color of each polygon will be determined by the average of the CData values for all of the polygon's vertices.
- Images cannot be rendered.
- No fonts are downloaded to the Illustrator file. Any fonts used must be available to Illustrator when the file is viewed.
- The file must be opened in Illustrator before it can be printed.

Saving and Reloading Figures (-dmfile)

You can use the `-dmfile` option to save a Figure for future display. MATLAB creates an M-file that contains the necessary object creation and set commands to reproduce the Figure. If necessary, MATLAB also creates a MAT-file that contains data needed to create the Figure.

For example, this command creates a file named `mygraph.m`, and, if needed, a file named `mygraph.mat`.

```
print -dmfile mygraph
```

Do not include an extension in the filename. MATLAB will create the files with the appropriate extensions.

To display a Figure that you have saved, execute the M-file. MATLAB loads the corresponding MAT-file and displays the Figure.

PC-Specific Options



On the PC, MATLAB uses two different printing mechanisms, depending on whether you print through Windows print drivers or with MATLAB's own built-in print drivers. By default, MATLAB uses Windows print drivers. To print using one of MATLAB's built-in drivers, you must either edit `printopt` (as described on page 7-6) to change the default device or else use the `print` command with the appropriate command-line switch.

The command-line switches for MATLAB's built-in drivers are listed on page 7-8. The command-line switches for printing through Windows drivers are listed on page 7-22.

Choosing Between Windows Drivers and MATLAB Drivers

When you use Windows drivers, printing is managed through the Windows Print Manager, which enables you to monitor printer queues and control various aspects of the printing process. When you print through MATLAB's drivers, MATLAB generates the output and copies it to a port, bypassing the Print Manager.

By default, MATLAB prints using Windows drivers. However, you may find the MATLAB drivers preferable in certain situations:

- If you are creating a file to import into a document, MATLAB has several Encapsulated PostScript drivers that create high-quality graphics for importing into word processing and page layout files.
- If you need to print to a printer for which you do not have the right Windows driver, you may be able to use one of the MATLAB drivers as a substitute.
- If you are having problems with a Windows driver, you can use a MATLAB driver instead.

This table summarizes the command-line switches that call Windows device drivers:

Device	Description
-dwi n	Use Windows printing services (black and white)
-dwi nc	Use Windows printing services (color)
-dmet a	Windows Enhanced Metafile format
-dbi tmap	Windows Bitmap (BMP) format
-dsetup	Display the Print Setup dialog box, but do not print
-v	Verbose mode to display the Print dialog box (suppressed by default)

If you use the `print` command and do not specify any device-type option, MATLAB uses a default value. This default is usually `-dwi n`, unless you have modified the `printopt` function. This means that if you do not specify a device type, MATLAB will use a Windows driver and create black and white output. Note that if you are using a color printer, you must use the `-dwi nc` switch (or modify `printopt` to make `-dwi nc` the default). If you use `-dwi n`, your output will be black and white, even if you use a color printer.

Note that when you print using the `-dwi n` or `-dwi nc` device type, the output is always directed to a printer. Therefore, you should not specify a filename. If you do specify a filename, MATLAB will create the file using one of its built-in PostScript drivers rather than a Windows driver.

When you print with one of MATLAB's built-in drivers, MATLAB generates output in the appropriate format and then either saves the output to a file (if you provided a filename) or else sends the output to the printer (if you did not give a filename).

If the output is directed to a printer, MATLAB creates a temporary file and then executes the MS-DOS command stored in the `pcmd` string returned by `printopt`. The default value for `pcmd` is:

```
COPY /B %s LPT1:
```

This command copies the output file to the LPT1 port. The file is then deleted.

Ghostscript Drivers. Using Ghostscript drivers is similar to using the built-in MATLAB drivers. When you specify a Ghostscript driver, MATLAB generates PostScript, which Ghostscript then converts to the selected format. MATLAB then executes the command stored in `pcmd` to print the file, or else saves the output to the specified filename.

Troubleshooting. Occasionally, you may run into problems when printing with a Windows driver, because of a bug in the driver or an incompatibility between the driver and MATLAB. If you do have a problem printing with a Windows driver, try one of these options:

- Use a different Windows driver. There may be a newer version of the driver available from the manufacturer, or there may be a driver available from a different vendor. You may also be able to use a driver for a different printer, such as an earlier model from the same manufacturer.
- Use a built-in MATLAB driver or a Ghostscript driver. For example, if you are having trouble printing to an HP LaserJet printer using a Windows driver, you can use one of the Ghostscript LaserJet drivers instead. If your printer supports PostScript, use one of MATLAB's built-in PostScript drivers.

Network Printing. If your PC is on a Microsoft or Novell network, you can print to a network printer using a Windows driver. If you want to use one of MATLAB's built-in drivers (or a Ghostscript driver), you must first map the LPT1 port to the printer you want to use.

To map LPT1 on Microsoft networks, issue this command at the system's command prompt:

```
net use LPT1: \\server\printer /persistent:yes
```

where *server* is the name of the server sharing the printer and *printer* is the name of the printer.

On Novell NetWare networks, use this command:

```
capture l=1 q=printer
```

where *printer* is the name of the print queue.

If you are using a Microsoft network, you can map LPT1, or you can instead edit `prntopt` to change the definition of `pcmd` to:

```
COPY /B %s \\server\printer:
```

where *server* is the name of the server sharing the printer and *printer* is the name of the printer.

Macintosh-Specific Options



MATLAB uses a feature of the PICT format that ensures that Figures can be printed by either QuickDraw or PostScript printers. When you print your Figure, MATLAB instructs the operating system to generate a PICT image containing QuickDraw commands. In addition, MATLAB generates PostScript commands (via its internal printer driver) and embeds the PostScript into the PICT as picture comments. If your printer is a PostScript printer, it uses the PostScript commands rather than the QuickDraw.

If your printer does not support QuickDraw or PostScript (for example, certain Hewlett-Packard printers don't), then you must have an appropriate driver for that printer installed on your Macintosh, and select that driver in the Chooser. This driver will interpret the QuickDraw commands, translating them into the appropriate language for the printer.

If you want to generate PICT output only, you can use the `-dpi ct` command-line switch. The resulting output can be directed to a QuickDraw printer or to a file. If you create a file, it can be read into any application that supports MacDraw-compatible PICT graphics.

Printing Tips and Troubleshooting

MATLAB's printed output does not always match the image you see on your screen. Certain display elements are changed for printing, to better match the characteristics of the output device.

This section discusses how to control the appearance of Figures to ensure that MATLAB's printed output is what you expect. This section addresses several common questions about printing:

- How do I control the size and aspect ratio of the graphic?
- How do I specify fonts and character sets?
- How do I make different lines print in different line styles?
- How do I specify the rendering method?
- How do I change the background colors?
- How do I set printing preferences on my Macintosh?

Controlling Output Size and Aspect Ratio

The Handle Graphics Figure object has several properties that control the size and aspect ratio of the printed graphic. These properties all begin with `Paper`. See the entry for `figure` in the online MATLAB Function Reference for information about these properties.

The most important of these properties is the `PaperPosition` property. This property is a four-element row vector that specifies the dimensions and position of the printed output. The form of the vector is:

`[left right width height]`

where `left` specifies the distance from the left edge of the paper to the left edge of the graphic, `right` specifies the distance from the bottom of the paper to the bottom of the graphic, and `width` and `height` specify the graphic's width and height.

By default, the value of `PaperPosition` does not change when you resize or reshape a Figure. This means that the size of the printed output may not match

the screen display. If you want the printed Figure to match its size and shape on the screen, you can do one of the following:

- Choose **Page Position** from the **File** menu. In the dialog box, click the button labeled **Match Paper Area to Figure Area**.
- From the command line, enter:

```
set(gcf, 'PaperPositionMode', 'auto')
```

When `PaperPositionMode` is set to `auto`, the width and height of the printed Figure are determined by the Figure's dimensions on the screen, and the Figure position is adjusted to center the Figure on the page. Note that when `PaperPositionMode` is `auto`, MATLAB actually sets the value of the `PaperPosition` property when you resize the Figure. Therefore, if you change `PaperPositionMode` back to `manual` and then print the Figure, the output will still be the same size as the Figure is on screen, unless you also set `PaperPosition` to default.

If you want the default value of `PaperPositionMode` to be `auto`, enter this line in your startup. `m` file:

```
set(0, 'DefaultFigurePaperPositionMode', 'auto')
```

Setting `PaperPositionMode` to `auto` is especially important if you want to print out Figures that include `Uicontrols` or `Images`. If `PaperPositionMode` is `manual`, these objects are likely to be distorted when you print them.

Paper Size

You can use the Handle Graphics `PaperType` property to set the size of the paper you are printing on. The values for `PaperType` are standard paper sizes such as `usletter` and `a4`. When you set `PaperType`, MATLAB also sets `PaperSize`, which is a read-only property that indicates the actual dimensions for the current `PaperType`.

If you are printing on a paper size that MATLAB does not recognize, set the `PaperPosition` property to position the output appropriately. MATLAB will use the values you set, even if the size specified by `PaperPosition` is larger than the current value of `PaperSize`.

For example, suppose you want to print on paper that is 30 inches by 40 inches. You could set `PaperPosition` to `[.5 .5 29 39]`, either at the command line or in the **Page Position** dialog box.

Orientation

By default, Figures print in portrait mode. If you want to print in landscape mode, set the `PaperOrientation` property to `landscape`. You can do this in the **Page Position** dialog box, or by entering this command at the command line:

```
set(gcf, 'PaperOrientation', 'landscape')
```

You may also need to set the `PaperPosition` property so that the Figure fits on the page.

MATLAB provides a command, `orient`, that can simplify this process. `orient` sets the orientation of the printed output, by setting the `PaperOrientation` and `PaperPosition` properties of the Figure. For more information about this command, see the online MATLAB Function Reference.

Specifying Fonts and Character Sets

MATLAB Figures support several kinds of text objects, such as titles, axis labels, and tick labels. This section discusses how to control the font and character set for text objects in Figures so that the printed output uses the fonts you want.

Font characteristics are properties of Axes, Uicontrols, and Text objects. For each of these objects, you can set these properties:

- `FontName`
- `FontSize`
- `FontUnits`
- `FontWeight`
- `FontAngle`

For example, to specify 10-point Helvetica-BoldOblique for the current Axes:

```
set(gca, 'FontName', 'Helvetica', 'FontSize', 10, 'FontUnits', ...
    'points', 'FontWeight', 'bold', 'FontAngle', 'oblique')
```

Note that when MATLAB generates hardcopy output, it does not attempt to determine what fonts are available on the hardcopy device before it sends output to the device. If you specify a font that is not available on your printer, the printer will substitute another font. A PostScript printer will substitute Courier for any unavailable font.

MATLAB can generate PostScript output using the fonts listed below. These are the actual names you should use when you specify the `FontName` property for a text object:

- AvantGarde
- Bookman
- Courier
- Helvetica
- Helvetica-Narrow
- NewCenturySchlbk
- Palatino
- Symbol
- Times-Roman
- ZapfChancery
- ZapfDingbats

If you use a font not on this list, MATLAB's PostScript driver substitutes Courier. This substitution affects the Ghostscript drivers as well, because they work by converting MATLAB's PostScript output.

Font properties for the Axes object itself affect the x -, y -, and z -tick labels. Axis labels (`XLabel`, `YLabel`, and `ZLabel`) and `Title`s also use the Axes font characteristics; however, you can set the font characteristics for these Text objects explicitly to override the Axes font values. For example, to change the font size of the Title, you could enter:

```
h = get(gca, 'Title');
set(h, 'FontSize', 18);
```

The character set used for a Text object is determined by its font. On most platforms, most fonts use the primary character set encoding for the platform. For PostScript output, you can also specify default PostScript character-set encoding by using the `-adobecset` option, as described on page 7-15.

PC



On the PC, the valid fonts and character sets depend on whether you print using Windows drivers or MATLAB's built-in drivers.

The built-in MATLAB drivers support only fonts that are compatible with the Windows Latin-1 character set. If you use a built-in MATLAB driver, you

should choose fonts that match the standard set of supported PostScript fonts (such as Times or Helvetica). TrueType fonts are acceptable as long as they meet this requirement. For example, the TrueType Symbol font works. MATLAB also accepts the TrueType fonts Arial, New Times Roman, and New Courier, and maps them to their PostScript equivalents (Helvetica, Times Roman, and Courier, respectively).

The native Windows drivers support Windows Latin-1 as well as a wide variety of other Windows character sets. If you print using the Windows drivers, Windows requires that you use TrueType fonts for text to be printed correctly. (You can tell if a font is TrueType by looking at the Fonts Control Panel. The icon for a TrueType font has “TT” on it, and the filename extension is “TTF”.)

UNIX



On UNIX systems, MATLAB supports the ISO Latin-1 primary character set. For example, suppose a text object is created with these commands:

```
h = text(0.1, 0.1, 'some text');
set(h, 'FontName', 'Times');
```

There might be several X Window System fonts with the name “Times.” MATLAB tries to find a font that supports the primary ISO Latin-1 character set. For backward compatibility, this preference is ranked below any other specifications you provide, such as the font size, style, and so forth. ISO Latin-1 fonts have an X font specification that ends in `iso8859-1`, which is the formal name of the ISO Latin-1 character set. Here is an example of such a font specification:

```
-misc-fixed-medium-r-semi-condensed--13-120-75-75-c-60-iso8859-1
```

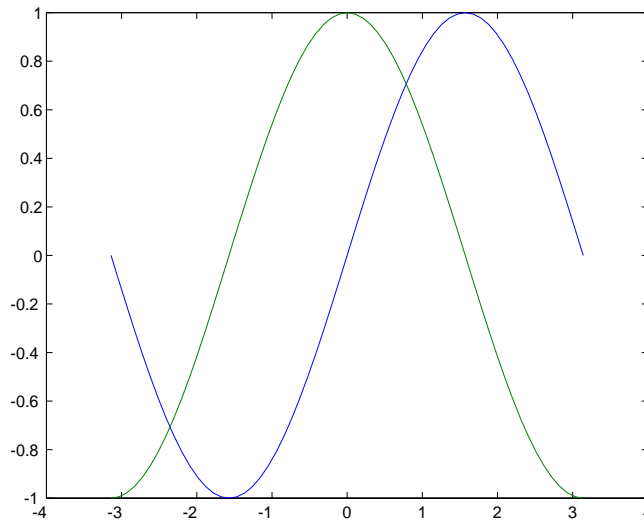
You can use `xlsfonts` at the UNIX prompt to list the set of fonts available on your system.

Specifying Line Styles

When displaying on a color screen or printing to a color printer, MATLAB usually distinguishes different lines in a Figure by their colors. For example, these

commands plot the sine and cosine functions; MATLAB sets the colors of the lines according to the value of the `ColorOrder` property:

```
x = -pi : pi / 30 : pi ;  
plot(x, sin(x), x, cos(x))
```

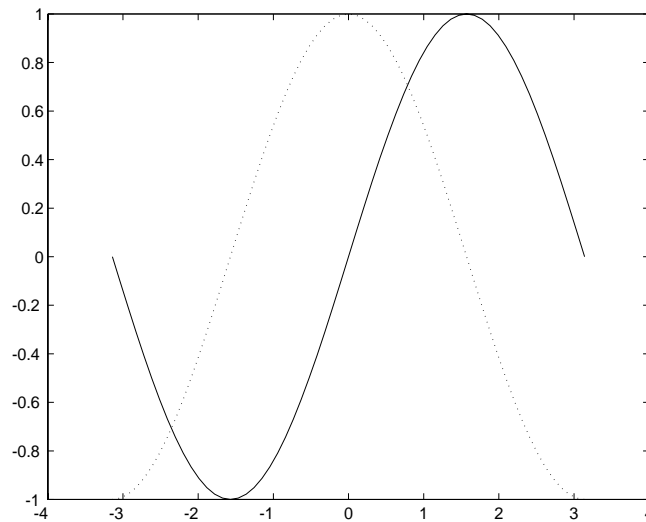


In addition to color, you can distinguish lines by line style or marker symbol. If you want to print the Figure on a black and white printer, keep in mind that all lines will print as black (or white, if the background is black and `InvertHardCopy` is off). If you want MATLAB to dither the lines to attempt to render them as different shades of gray, you can use a color driver; however, lines are generally too thin to be dithered effectively. A better approach is to vary the line style or marker symbol. Many of the plotting functions provide a mechanism for setting the line style and marker symbol for each line being plotted.

You can also control the line styles by setting the `Axes LineStyleOrder` and `ColorOrder` properties. To distinguish lines, MATLAB cycles first through the `ColorOrder` values and then the `LineStyleOrder` values. The factory default for `ColorOrder` is a set of six colors, while the factory default for `LineStyleOrder` is a single style (a solid line). This means that MATLAB will use different colors but the same line style for all lines, unless you specify otherwise.

If you print to a black and white printer, you may want to change `Col orOrder` to a single color, and `Li neStyl eOrder` to multiple styles. This will cause MATLAB to use the same color for each line, but different styles. These values must be set before the Axes object is created. For example, this code creates a new Figure, sets the appropriate Axes properties, and then creates the plot:

```
x = -pi : pi / 30 : pi ;
figure('DefaultAxesCol orOrder', [0 0 0], ...
'DefaultAxesLi neStyl eOrder', '- | : | - - | - . ')
plot(x, si n(x), x, cos(x))
```



Windows 95 Limitation



Microsoft Windows 95 does not support broken line styles for lines whose width is greater than 1 pixel. Unfortunately, most printers produce lines more than 1 pixel thick, so in most cases, Windows 95 drivers produce solid lines, regardless of the setting of `Li neStyl eOrder`.

There are various ways you can work around this problem:

- Set up MATLAB to use lines 1 pixel wide, by adding this line to the [MATLAB Settings] section of your MATLAB.INI file:

```
ThinLineStyles=1
```

This will result in very thin lines, but the lines will print with the specified styles.

- Set the Figure's `Renderer` property to `zbuffer`:

```
set(gcf, 'Renderer', 'zbuffer')
```

This will result in the printed output matching the screen display. See "Selecting the Rendering Method" for more information about Z-buffer.

- Use a Ghostscript driver. These drivers bypass the Windows Print Manager. See page 7-9 for a list of the Ghostscript drivers.

Selecting the Rendering Method

MATLAB uses two different methods to render Figures, painter's algorithm and Z-buffer. Painter's algorithm draws Figures using vector graphics, while Z-buffer uses raster (bitmap) graphics.

In general, painter's algorithm produces higher-resolution results than Z-buffer. However, Z-buffer works in situations where painter's algorithm either produces inaccurate results or does not work at all. By default, MATLAB automatically selects the best method, based on the complexity of the Figure and the settings of various Handle Graphics properties.

You can specify the rendering method by setting the Figure `Renderer` and `RendererMode` properties. When the `RendererMode` property is set to `auto` (the factory default), MATLAB selects the rendering method for displaying and for printing. The rendering method used for printing the Figure is not always the same method used to display the Figure.

When `RendererMode` is set to `manual`, MATLAB uses the method specified by the `Renderer` property for both displaying and printing.

In some cases, you may want to override MATLAB's renderer selection when you print, without changing the `Renderer` property. You can specify the rendering method to use for printing by using the `-zbuffer` or `-painters` option with the `print` command.

For example, these commands create a Figure, display it using painter's algorithm, and print it using Z-buffer:

```
surf(peaks(32)); set(gcf, 'Renderer', 'painters')  
print -zbuffer
```

Limitations of Each Method

For many Figures, it is possible to use either rendering method. There are certain situations, however, where painter's algorithm does not work or produces unacceptable results. For example:

- If the Figure uses truecolor for Patch or Surface objects, it cannot be rendered with painter's algorithm. If you set `Renderer` to `painters`, MATLAB issues a warning and the graphics objects do not display or print.
- If the Figure includes any lights, the lighting cannot be rendered with painter's algorithm. If you set `Renderer` to `painters`, the lighting disappears.

Note that in each case the Figure retains all the appropriate data, so if you set `Renderer` back to `zbuffer` or set `RenderMode` to `auto`, the missing objects reappear.

In general, if you find that your printed output does not match what you see on screen, you should set `Renderer` to `zbuffer`, or use the `-zbuffer` switch when you print.

However, you cannot use Z-buffer rendering if your device type is HPGL or Adobe Illustrator. If you attempt to print to one of these formats and `Renderer` is set to `zbuffer` (or if you use the `-zbuffer` option), MATLAB uses painter's algorithm instead.

Size of Output Files

When you print a Figure rendered with painter's algorithm, the resolution has little effect on the size of the output file or the amount of memory needed for printing. Therefore, the default resolution is quite high (864 dpi for MATLAB's built-in PostScript drivers).

When you print a Figure rendered using Z-buffer, certain factors directly influence the size of the output file or amount of memory needed for printing:

- Resolution of the output
- Size of the printed graphic
- Use of a color or black and white (grayscale) driver

These relationships exist because Z-buffer Figures are rendered as bitmaps, and the number of pixels in a bitmap is a function of the resolution and of the size of the graphic. For example, if a Figure is 2 inches by 3 inches, it will consist of 60,000 pixels at 100 dpi. Increasing either the resolution or the size increases the number of pixels proportionately. For example:

- If you keep the Figure the same size but increase resolution to 200 dpi, the number of pixels is 240,000.
- If you keep the resolution at 100 dpi but enlarge the Figure to 4 inches by 6 inches, the number of pixels is also 240,000.

Note that the size of the actual output is what matters, not the size on the screen (although, if `PaperPositionMode` is set to `auto`, the size on screen and on paper are the same).

For Figures rendered using Z-buffer, the default resolution is 72 dpi on the Macintosh, and 150 dpi on other platforms. To set the resolution to a different value, use the `-r` option. The syntax is:

```
print -r $number$ 
```

$number$ is the number of dots per inch. For example, to specify a resolution of 100 dpi:

```
print -r100
```

To specify printing at screen resolution, set $number$ to 0 (zero):

```
print -r0
```

In addition to the Figure size and resolution, the choice of color or black and white also affects the size of the file, because the amount of information stored for each pixel is larger for color than for black and white. Color files are three times as large as black and white files, so be sure to use a black and white driver unless you want to print to a color device.

Because of these issues, you must make trade-offs between resolution, size, color, and printing resources, when printing a Figure rendered using Z-buffer. For example, you can specify any resolution, but you may find at higher resolutions the resulting files are too big and require too much memory to print. However, you are likely to find that much lower resolutions produce acceptable results.

Changing Background Colors

By default, MATLAB Figures display on screen as colored lines and surfaces on a white background. When you print a Figure on a color device, the colors remain unchanged. If you print to a black and white device, surface colors are dithered to render them as shades of gray, except for lines and text, which are changed to black because these objects are too thin to be dithered effectively.

If you want MATLAB to dither lines, use a color driver rather than a black and white driver. For example, if you are printing on a black and white PostScript printer, you could use the `-dpsc` option. Note, however, that you may not be able to distinguish between different colored lines on the basis of the dithering.

If you prefer, you can display Figures on screen as colored lines and surfaces on a black background, by typing:

```
col ordef black
```

When you print a Figure with a black background, MATLAB inverts the colors for printing: anything black (including the background) is changed to white, and anything white (such as lines, surfaces, or text) is changed to black. These changes are made so the printer will use less toner and produce better looking output.

If you do not want MATLAB to invert the colors when you print the Figure, set the Figure's `InvertHardCopy` property to `off`. For example:

```
set(gcf, 'InvertHardCopy', 'off')
```

Note that MATLAB does not invert Image or Uicontrol objects when you print them, regardless of the value of `InvertHardCopy`.



Setting Printing Preferences (Macintosh)

On all platforms, you can control the printed output by setting Handle Graphics properties. On Macintosh systems, however, you can set preferences that override some of these properties.

You can use the **Preferences** item on the **File** menu to customize various aspects of printing Figures, saving them to a file, and copying them to the clipboard. For example, you can select the type of PostScript to produce, and the type of preview image for EPS files. Preferences you set through this option persist from one MATLAB session to the next, and change only when you explicitly change them.

The preferences you set apply only to printing, saving, and copying done by selecting items from the Macintosh menu bar. Preferences do not affect the MATLAB `print` command.

Troubleshooting MS-Windows Printing

If you encounter problems such as segmentation violations, general protection faults, application errors, or the output does not appear as you expect when using MS-Windows printer drivers, try the following:

- If your printer is PostScript compatible, print with one of MATLAB's built-in PostScript drivers. There are four PostScript device options that you can use with the `print` command: `-dps`, `-dpSC`, `-dps2`, and `-dpSC2`. See the `print` documentation in the online MATLAB function reference for more information (type `doc print` on the MATLAB command line).
- The behavior you are experiencing may occur only with certain versions of the print driver. Contact the print driver vendor for information on how to obtain and install a different driver. If you are using Windows 95, try installing the drivers that ship with the Windows 95 CD-ROM.
- Try printing with one of MATLAB's built-in GhostScript devices. These devices use GhostScript to convert PostScript files into other formats, such as HP LaserJet, PCX, Canon BubbleJet, and so on.
- Copy the Figure as a Windows Metafile using the **Edit-->CopyFigure** menu item on the Figure window menu or the `print -dmeta` option at the command line. You can then import the file into another application for printing.

Using MATLAB Graphics in Other Applications

In addition to options for printing directly to hardcopy devices, MATLAB provides the ability to produce files in various graphics formats for importing into other applications.

Creating Graphics Files

There are several ways to create graphics files in MATLAB. MATLAB supports these methods on all platforms:

- Use the `print` command with an appropriate driver; for example, one of the Encapsulated PostScript drivers.
- Use the `capture` command to create an image of the Figure, and then use `imwrite` to write the file.

On the PC and Macintosh, there are additional methods, such as copying the Figure to the clipboard. This section describes how to use `print` and `capture`, as well as system-specific methods.

Using the `print` Command

When you use the `print` command and specify a filename, MATLAB creates the file but does not send it to the printer. Depending on the device driver you use, the file may be in a format that you can import into other applications.

On all platforms, you can use MATLAB's built-in drivers to produce graphics files in Encapsulated PostScript, Adobe Illustrator 88, and HPGL formats. To produce a file in one of these formats, specify the appropriate driver, and provide a name for the file. For example, this command produces an HPGL file named `surfplot.hgl`:

```
print -dhpgl surfplot
```

Additional formats are available only on certain platforms. On PC and UNIX systems, you can use Ghostscript drivers to produce standard graphics file formats such as PCX. On the PC, you can create files in Windows Bitmap and Windows Enhanced Metafile format by using the `-dbitmap` option or the `-dmeta` option and specifying a filename. (If you omit the filename, the Metafile or Bitmap is placed in the clipboard.)

For example, this command creates a Windows Bitmap file named `surfplot.bmp`:

```
print -dbitmap surfplot
```

On Macintosh systems, you can use the `-dpic` switch to produce PICT files.

Using the capture Command

Another way to produce a graphics file is by using the `capture` command to create a bitmapped image of the Figure, and then writing the image to a file by using the `imwrite` function. For example, to create a TIFF file from the Figure whose handle is 2:

```
[X, map] = capture(2);  
imwrite(X, map, 'fig2.tif')
```

`capture` works by creating a screen capture of the Figure. The image matrix is a pixel-for-pixel map of the Figure as it is displayed on screen, so the captured image is identical in size, shape, and appearance to the displayed Figure.

After you use `capture`, use `imwrite` to write the image to a file. `imwrite` supports several common formats:

- BMP
- HDF
- JPEG
- PCX
- TIFF
- XWD

For more information about `imwrite`, see the online MATLAB Function Reference.

PC-Specific Options



On the PC, you can import a MATLAB graphic into another application by copying the Figure to the clipboard in Windows Bitmap or Windows Enhanced Metafile format, and then pasting the graphic into the other application.

There are two ways to copy a Figure to the clipboard:

- Select the **Copy Figure** command from the **Edit** menu of the Figure window. The format of the output is determined by preferences you can set.
- At the command line, use the `print` command with the `-dbi tmap` or `-dmeta` option. Do not provide a filename. The Figure will be copied to the clipboard as a Windows Bitmap or Metafile, depending on which switch you use.

You then import the graphic into another application by using the **Paste** command.

Choosing the Format. The Windows Bitmap and Enhanced Metafile formats are fundamentally different in the way they represent the Figure. The Bitmap format creates a bitmapped copy of the Figure window, while the Metafile format uses a vectorized approach. In general, the bitmap format is of lower resolution than the Metafile format.

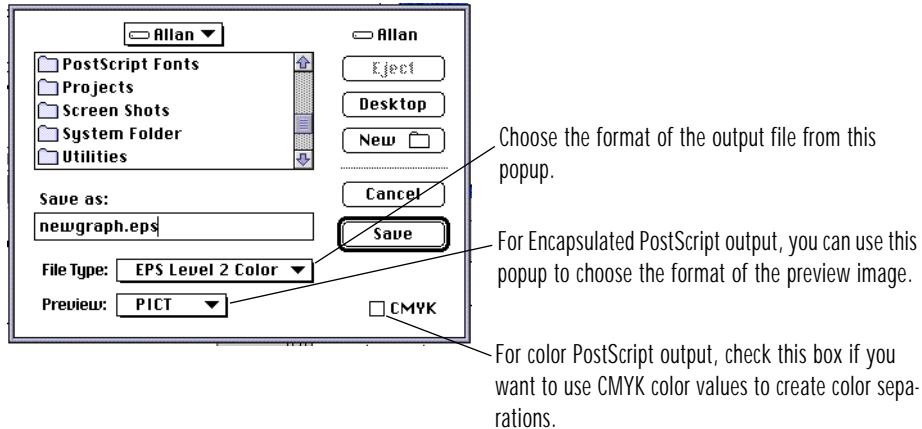
The Windows Enhanced Metafile format is a device-independent format for sharing graphics between Windows applications. This format is capable of producing high-quality graphics, and is the preferred graphics format to use on Windows systems. See page 7-43 for more information about using this format.

Macintosh-Specific Options



On the Macintosh, you can use the **Save As** option to create a graphics file. You can also copy and paste.

Saving to a File. When a Figure window is active, select **Save As** from the **File** menu. MATLAB displays this dialog box:



Copy and Paste. To copy a Figure to the clipboard, make the Figure the current window and select the **Copy** command from the **Edit** menu. You then import the graphic into another application by using the **Paste** command.

You can copy a Figure to the clipboard as either PICT drawing or a bitmap. The format used is determined by preferences you can set (see page 7-36). If you copy the image as a PICT drawing, the drawing may be editable in certain applications, such as Canvas.

Importing MATLAB Graphics into Other Applications

The graphics files that MATLAB creates can be imported into a wide variety of applications for word processing, desktop publishing, presentations, and graphics. To import MATLAB graphics into a specific application, you need to keep in mind certain considerations. This section discusses:

- Choosing the graphics format
- Copying graphics files to another platform
- Application-specific issues

Choosing the Format

As described above, MATLAB provides many different options and formats for graphical output. The best format to use depends on your platform and which applications you want to import graphics into.

This section offers some guidelines for selecting a format. Note that these are only guidelines, and are not meant to be definitive.

When deciding which format to use, you should consider these questions:

- What formats does the target application support?
- Do you need to be able to edit the graphic in the target application?
- What level of quality do you need?
- Do you need to be able to use the graphic on two or more platforms?
- What is the most convenient format to use?

Obviously, the most important criterion is what format your application can import. However, most applications can import several formats, so there is often a choice available.

Another issue is whether you need to be able to edit the graphic once it is imported into the new application. If you do not need to edit the graphic, you can use any format that the application will import. If you need to be able to edit the graphic, there are a few options:

- If the target application is Adobe Illustrator, create the file using the `print -dill` command. The resulting file will be editable in Illustrator.
- On the PC, the Enhanced Metafile format can be edited in many applications, such as Powerpoint.
- On the Macintosh, PICT drawings are editable in certain drawing programs, such as Canvas.
- Some painting programs can edit bitmaps in certain formats. For example, the Paintbrush application that comes with Windows can edit PCX files.
- For image processing applications, use one of the formats produced by `imwrite`, such as TIFF.

In terms of quality, the main issue is whether to use a vector format or a raster format. Vector formats store graphics as geometric objects, while raster formats store graphics as matrices of pixels (bitmaps). Vector formats generally produce higher quality line and surface plots than raster formats, while raster

formats are better for images. You can resize a vector graphic without losing quality, while a raster graphic will have lines with jagged edges. The vector formats that MATLAB supports are Encapsulated PostScript, Adobe Illustrator 88, HPGL, Windows Metafile, and PICT. The other graphics formats that MATLAB supports are all raster formats.

For many applications, the best format to use is Encapsulated PostScript. This format provides very high quality output, because it is a vector format. EPS is also very portable, as it is supported on every platform that MATLAB runs on. The main drawback of EPS is that when you print the document that the graphic is embedded in, you must use a PostScript printer, or the graphic will not print. Also, a MATLAB EPS graphic may not contain a preview, or the preview may be in a format that the target application does not support. If you import an EPS graphic that does not have a valid preview, the graphic will appear as a gray box on screen, but will appear appropriately on paper when you print the document.

Portability is an issue if you use more than one computer platform. If you use the target application on more than one platform, you need to use a format that is not platform-specific. For example, if you use FrameMaker on both Macintosh and UNIX systems, you should not use PICT format graphics, because these graphics may not display or print properly on UNIX systems. TIFF and EPS formats are better choices, because they are supported on all platforms that MATLAB runs on.

Finally, in terms of convenience, copying to the clipboard and pasting into the target application is generally the simplest method. The disadvantages of this approach are that you are limited to working on a single platform, and no file is created.

Note that if you need output in a graphics format that MATLAB does not produce, you may be able to use a format conversion application to convert a MATLAB-produced graphic to another format. For example, MATLAB does not produce GIF files (due to patent restrictions), but there are many applications that can convert TIFF files to GIF.

Copying Output Files to Another Platform

When you create a graphics file from a MATLAB Figure, you can import the file into another application on the same platform that you are running MATLAB on, or you can import the file into an application running on another platform. For example, if you are running MATLAB on a UNIX system you

may want to import the file into a Windows or Macintosh word-processing application.

Keep in mind that not all applications import the same graphics formats, and the formats commonly supported vary from platform to platform. If you want to transfer graphics files between platforms, the best formats to use are generally Encapsulated PostScript and TIFF. These formats are supported by most applications on all of the platforms that MATLAB runs on.

Application-Specific Issues

This section discusses issues related to importing MATLAB graphics into several commonly used applications. These applications are:

- Microsoft Word
- Corel Draw
- Scientific Word
- LaTeX

Microsoft Word. When you import a graphic into a Microsoft Word document, first create a frame in the document and import the graphic into it. Importing into a frame will enable you to reposition the graphic by dragging it.

Corel Draw. You can import Windows Enhanced Metafiles and PCX files into Corel Draw. Note that the graphic appears to be black and white until you make the picture full screen.

Scientific Word. You can import a MATLAB Figure into Scientific Word by creating an Encapsulated PostScript file. Note that you cannot control the size of the graphic in Scientific Word, so be sure to make the image the size you want when you create it in MATLAB. You can do this by setting the `PaperPosition` parameter, as described on page 7-25.

LaTeX. You can import a MATLAB Figure into LaTeX by creating an Encapsulated PostScript file. The general syntax for including a PostScript figure in LaTeX is:

```
\begin{figure}[h]
\centerline{\psfig{figure=figure.ps,height=height,angle=angle}}
\caption{caption}
\end{figure}
```

(The items in *italics* are placeholders for the actual values you specify.)

You can specify the height in any LaTeX compatible dimension. To set the height to 3.5 inches, use the command:

```
height=3.5in
```

You can use the `angle` command to rotate the graph. For instance, to rotate the graph 90 degrees, use the command:

```
angle=90
```

Handle Graphics

Handle Graphics Organization	8-2
Graphics Objects	8-2
Object Properties	8-7
Graphics Object Creation Functions	8-10
Example – Creating Graphics Objects	8-11
Parenting	8-12
High-Level Versus Low-Level	8-13
Simplified Calling Syntax	8-13
Using set and get	8-15
Setting Property Values	8-15
Getting Property Values	8-17
Factory-Defined Property Values	8-19
Default Property Values	8-20
Specifying Default Values	8-22
Examples – Setting Defaults	8-23
Accessing Object Handles	8-27
The Current Figure, Axes, and Object	8-27
Searching for Objects by Property Values — findobj	8-29
Copying and Deleting Objects	8-30
Controlling Graphics Output	8-33
Specifying the Target for Graphics Output.	8-33
Preparing Figures and Axes for Graphics	8-33
Testing for Hold State	8-38
Protecting Figures and Axes	8-39
Efficient Programming	8-44
Save Information First.	8-44
Properties Changed by Built-in Functions	8-45

Handle Graphics Organization

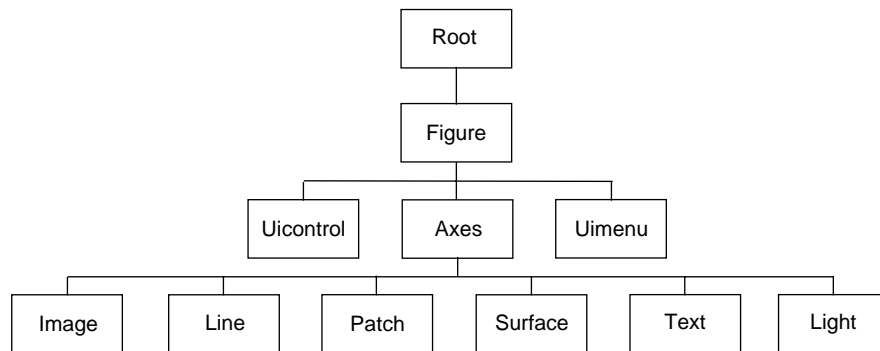
Handle Graphics is an object-oriented graphics system that provides the components necessary to create computer graphics. It supports drawing commands to create lines, text, meshes and polygons as well as interactive devices such as menus, pushbuttons, and dialog boxes.

With Handle Graphics, you can directly manipulate the lines, surfaces, and other graphics elements that MATLAB's high-level routines use to produce various types of graphs. You can use Handle Graphics from the MATLAB command line to modify the display or in M-files to create customized graphics functions.

Graphics Objects

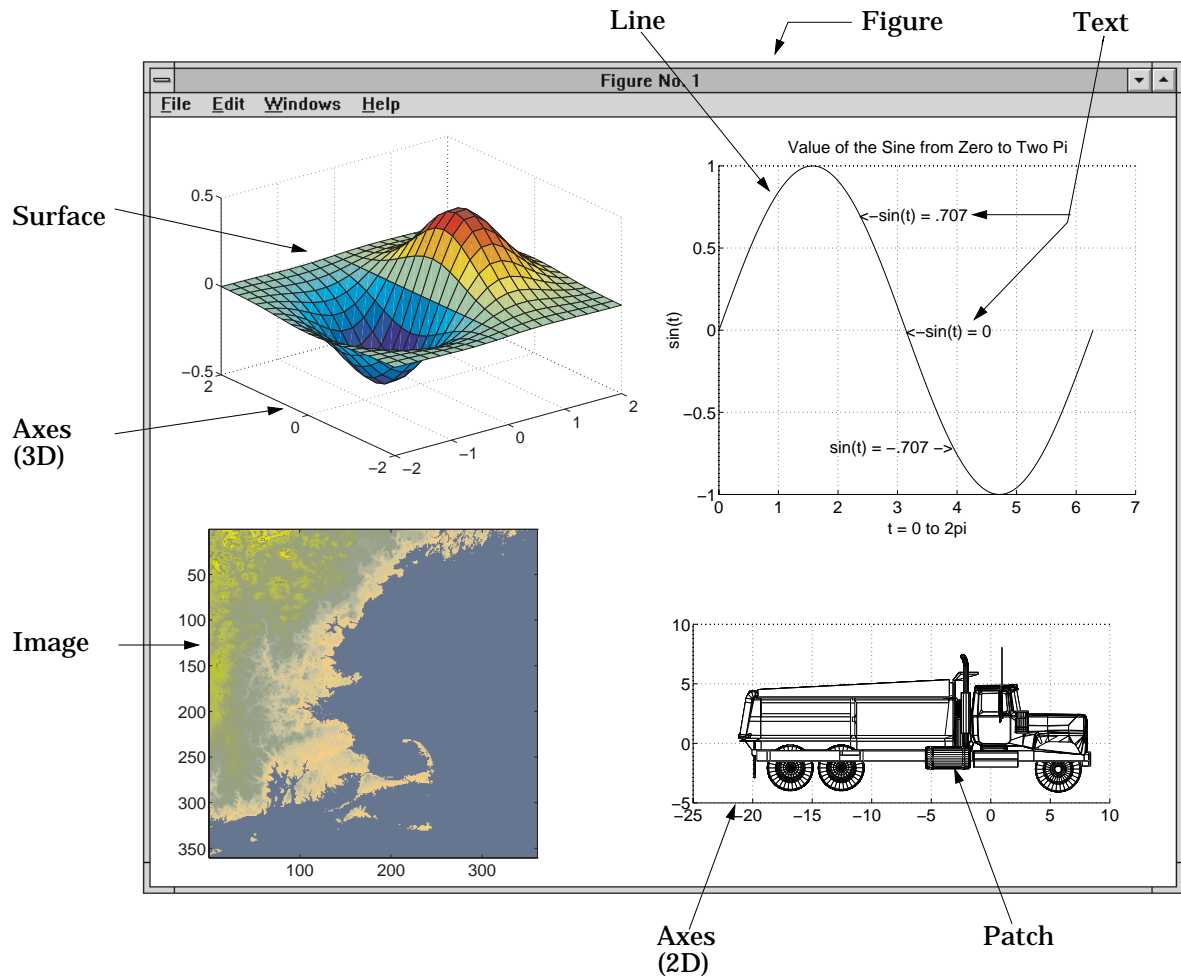
Handle Graphics objects are the basic drawing elements used by MATLAB to display data and to create graphical user interfaces (GUIs). Each instance of an object is associated with a unique identifier called a *handle*. Using this handle, you can manipulate the characteristics (called object *properties*) of an existing graphics object. You can also specify values for properties when you create a graphics object.

These objects are organized into a tree-structured hierarchy:



The hierarchical nature of Handle Graphics is based on the interdependencies of the various graphics objects. For example, to draw a Line object, MATLAB needs an Axes object to orient and provide a frame of reference to the Line. The Axes, in turn, needs a Figure window to display the Line.

Because graphics objects are interdependent, the graphics display typically contains a variety of objects that, in conjunction, produce a meaningful graph or picture. The following picture of a Figure window contains a number of graphics objects.



Each type of graphics object has a corresponding creation function that you use to create an instance of that class of object. Object creation functions have the same names as the objects they create (e.g., the `text` function creates `Text` objects, the `figure` function creates `Figure` objects, and so on).

The Root

At the top of the hierarchy is the Root object. It corresponds to the computer screen. There is only one Root object and all other objects are its descendants. You do not create the Root object; it exists when you start MATLAB. You can, however, set the values of Root properties and thereby affect the graphics display.

Figure

Figure objects are the individual windows on the Root screen where MATLAB displays graphics. MATLAB places no limits on the number of Figure windows you can create (your computer may, however). All Figures are children of the Root and all other graphics objects are descendants of Figures.

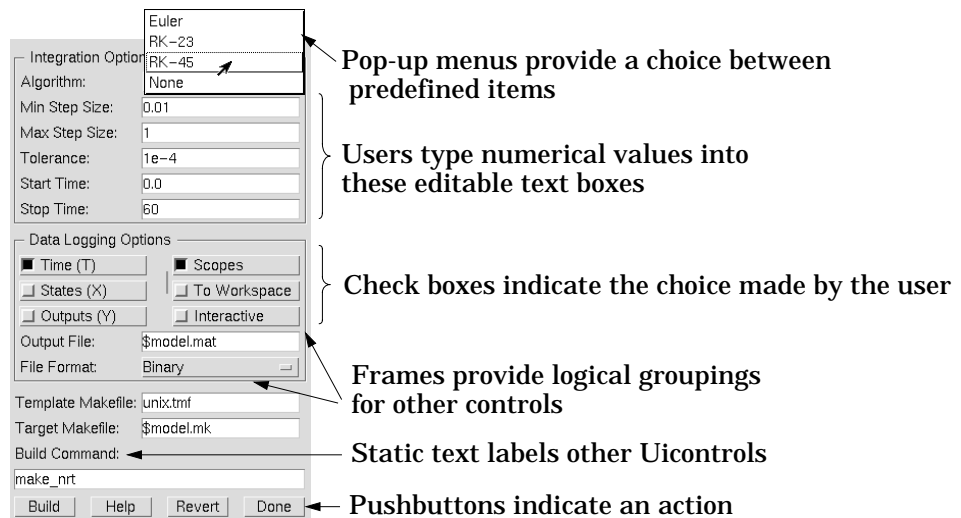
All functions that draw graphics (e.g., `plot` and `surf`) automatically create a Figure if one does not exist. If there are multiple Figures within the Root, one Figure is always designated as the “current” Figure, and is the target for graphics output. See chapter entitled *Figures* for information on using Figures.

Uicontrol

Uicontrol objects are user interface controls that execute callback routines when users activate the object. There are a number of styles of controls such as pushbuttons, listboxes, and sliders. Each device is designed to accept a certain type of information from users. For example, listboxes are typically used to provide a list of filenames from which you select one or more items for action carried out by the control’s callback routine.

The `ui control` entry in the online MATLAB Function Reference describes the available types of controls.

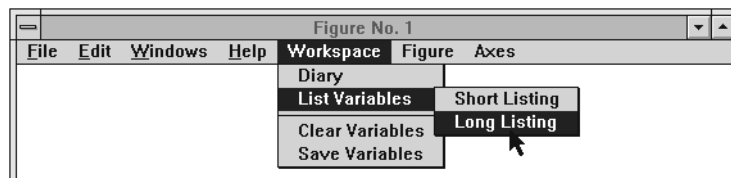
You can use Uicontrols in combinations to construct control panels and dialog boxes. Pop-up menus, editable text boxes, check boxes, pushbuttons, static text, and frames compose this particular example:



Uicontrol objects are children of Figures and are therefore independent of Axes.

Uimenu

Uimenu objects are pull-down menus that execute callback routines when users select an individual menu item. MATLAB places Uimenu on the Figure window menu bar, to the right of existing menus defined by the system. This picture shows the top of an MS-Windows Figure that has three top-level Uimenu defined (titled **Workspace**, **Figure**, and **Axes**). Two levels of submenus are visible under **Workspace** top-level Uimenu.



Uimenu are children of Figures and are therefore independent of Axes.

Axes

Axes objects define a region in a Figure window and orient their children within this region. Axes are children of Figures and are parents of Image, Light, Line, Patch, Surface, and Text objects.

All functions that draw graphics (e.g., `plot`, `surf`, `mesh`, and `bar`) create an Axes object if one does not exist. If there are multiple Axes within the Figure, one Axes is always designated as the “current” Axes, and is the target for display of the above mentioned graphics objects (Uicontrols and Uimenu are not children of Axes). The chapter entitled *Axes* provides information on using Axes.

Image

A MATLAB Image consists of a data matrix and possibly a colormap. There are three basic Image types that differ in the way that data matrix elements are interpreted as pixel colors – indexed, intensity, and truecolor. Since Images are strictly 2-D, you can view them only at the default 2-D view.

Light

Light objects define light sources that affect all objects within the Axes. You cannot see Lights, but you can set properties that control the style of light source, color, location, and other properties common to all graphics objects.

Line

Line objects are the basic graphics primitives used to create most 2-D and some 3-D plots. High-level functions `plot`, `plot3`, and `contour` (and others) create Line objects. The coordinate system of the parent Axes positions and orients the Line.

Patch

Patch objects are filled polygons with edges. A single Patch can contain multiple faces, each colored independently with solid or interpolated colors. `fill`, `fill3`, and `contour3` create patch objects. The coordinate system of the parent Axes positions and orients the Patch.

Surface

Surface objects are 3-D representations of matrix data created by plotting the value of each matrix element as a height above the x - y plane. Surface plots are

composed of quadrilaterals whose vertices are specified by the matrix data. MATLAB can draw Surfaces with solid or interpolated colors or with only a mesh of lines connecting the points. The coordinate system of the parent Axes positions and orients the Surface.

The high-level function `pcolor` and the `surf` and `mesh` group of functions create Surface objects.

Text

Text objects are character strings. The coordinate system of the parent Axes positions the Text. The high-level functions `title`, `xlabel`, `ylabel`, `zlabel`, and `gtext` create Text objects.

Object Properties

A graphics object's properties control many aspects of its appearance and behavior. Properties include general information such as the object's type, its parent and children, whether it is visible, as well as information unique to the particular class of object.

For example, from any given Figure object you can obtain the identity of the last key pressed in the window, the location of the pointer, or the handle of the most recently selected menu.

MATLAB organizes graphics information into a hierarchy and stores this information in properties. For example, Root properties contain the handle of the current Figure and the current location of the pointer (cursor), Figure properties maintain lists of their descendants and keep track of certain events that occur within the window, and Axes properties contain information about how each of its child objects uses the Figure colormap and the color order used by the `plot` function.

You can query the current value of any property and specify most property values (although some are set by MATLAB and are read only). Property values apply uniquely to a particular instance of an object; setting a value for one object does not change this value for other objects of the same type.

You can set default values that affect all subsequently created objects. Whenever you do not define a value for a property, either as a default or when you create the object, MATLAB uses "factory-defined" values.

The reference entry for each object creation function provides a complete list of the properties associated with that class of graphics object.

Properties Common to All Objects

Some properties are common to all graphics objects. These include:

Property	Information Contained
BusyActi on	Controls the way MATLAB handles callback routine interruption defined for the particular object
But tonDownFcn	Callback routine that executes when button press occurs
ChangeFcn	Callback routine that executes when a property belonging to this object changes
Chi l dren	Handles of all this object's children objects.
Cl i ppi ng	Mode that enables or disables clipping (meaningful only for Axes children)
CreateFcn	Callback routine that executes when this type of object is created
Del eteFcn	Callback routine that executes when you issue a command that destroys the object
Handl eVi si bi l i ty	Allows you to control the availability of the object's handle from the command line and from within callback routines
Interrupti ble	Determines whether a callback routine can be interrupted by a subsequently invoked callback routine
Parent	The object's parent

Property	Information Contained
Selected	Indicates whether object is selected
Select ionHl igh t	Specifies whether object visually indicates the selection state
Tag	User-specified object label
Type	The type of object (Figure, Line, Text, etc.)
UserData	Any data you want to associate with the object
Vi si bl e	Determines whether or not the object is visible

Graphics Object Creation Functions

Each graphics object (except the Root object) has a corresponding creation function, named for the object it creates. This table lists the creation functions:

Function	Object Description
axes	Rectangular coordinate system that scales and orients Axes children Image, Light, Line, Patch, Surface, and Text objects.
figure	Window for displaying graphics.
image	2-D picture defined by either colormap indices or RGB values. The data can be 8-bit or double precision data.
light	Directional light source located within the Axes and affecting Surfaces and Patches.
line	Line formed by connecting the coordinate data with straight line segments, in the sequence specified.
patch	Polygonal shell created by interpreting each column in the coordinate matrices as a separate polygon.
surface	Surface created with rectangular faces defined by interpreting matrix elements as heights above a plane.
text	Character string located in the Axes coordinate system.
ui control	Programmable user-interface device, such as push-button, slider, or listbox.
ui menus	Programmable menu appearing at the top of a Figure window.

All object creation functions have a similar format:

```
handle = function('propertyname', propertyvalue, ...)
```

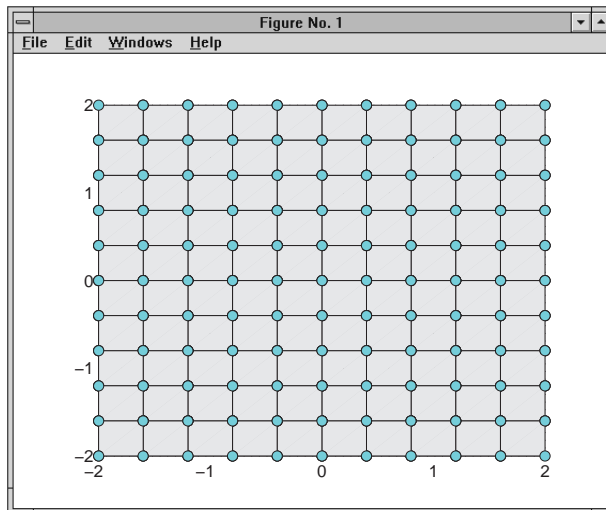
You can specify a value for any object property (except those that are read only) by passing property name/property value pairs as arguments. The function returns the handle of the object it creates, which you can use to query and modify properties after creating the object.

Example – Creating Graphics Objects

The statements,

```
[x, y] = meshgrid([-2: .4: 2]);
Z = x.*exp(-x.^2-y.^2);
fh = figure('Position', [350 275 400 300], 'Color', 'w');
ah = axes('Color', [.8 .8 .8], 'XTick', [-2 -1 0 1 2], ...
         'YTick', [-2 -1 0 1 2]);
sh = surface('XData', x, 'YData', y, 'ZData', Z, ...
           'FaceColor', get(ah, 'Color')+.1, ...
           'EdgeColor', 'k', 'Marker', 'o', ...
           'MarkerFaceColor', [.5 1 .85]);
```

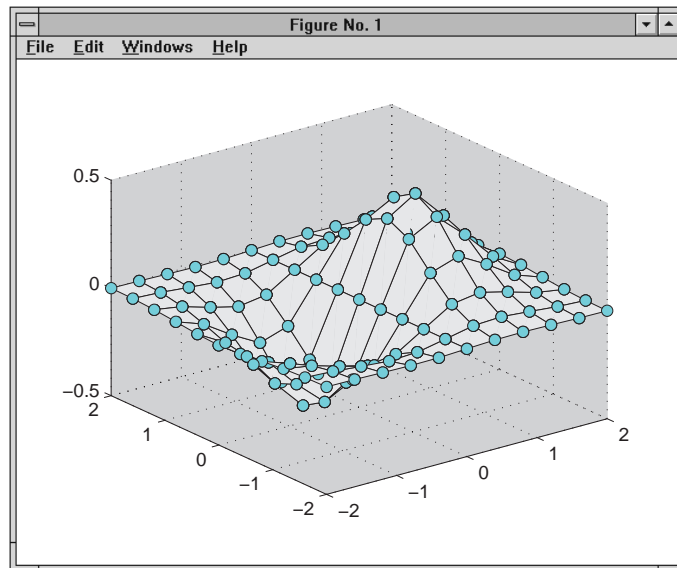
evaluate a function and create three graphics objects using the property values specified as arguments and default values of all other properties:



Note that the surface function does not use a 3-D view like the high-level surf functions. Object creation functions simply add new objects to the current Axes without changing Axes properties, except the `Children` property, which now includes the new object and the axis limits (`XLim`, `YLim`, and `ZLim`), if necessary.

You can change the view using the Axes camera properties (see the *Three-Dimensional Graphs* chapter) or use the `view` command:

```
view(3)
```



Parenting

By default, all statements that create graphics objects do so in the current Figure and the current Axes (if the object is an Axes child). However, you can specify the parent of an object when you create it. For example, the statement:

```
axes('Parent', figure_handle, ...)
```

creates an Axes in the Figure identified by `figure_handle`. You can also move an object from one parent to another by redefining its `Parent` property:

```
set(gca, 'Parent', figure_handle)
```

High-Level Versus Low-Level

MATLAB's high-level graphics routines (e.g., `plot` or `surf`) call the appropriate object creation function to draw graphics objects. However, high-level routines also clear the Axes or create a new Figure, depending on the settings of the Axes and Figure `NextPlot` properties.

In contrast, object creation functions simply create their respective graphics objects and place them in the current parent object. They do not respect the setting of the Figure and Axes `NextPlot` properties.

For example, if you call the `line` function,

```
line('XData', x, 'YData', y, 'ZData', z, 'Color', 'r')
```

MATLAB draws a red line in the current Axes using the specified data values. If there is no Axes, MATLAB creates one. If there is no Figure window in which to create the Axes, MATLAB creates it as well.

If you call the `line` function a second time, MATLAB draws the second line in the current Axes without erasing the first line. This behavior is different from high-level functions like `plot` that delete graphics objects and reset all Axes properties (except `Position` and `units`). You can change the behavior of high-level functions using the `hold` command or changing the setting of the Axes `NextPlot` property.

See the “Controlling Graphics Output” section in this chapter for more information on this behavior and on using the `NextPlot` property.

Simplified Calling Syntax

Object creation functions have convenience forms that allow you to use a simpler syntax. For example,

```
text(.5, .5, .5, 'Hello')
```

is equivalent to,

```
text('Position', [.5 .5 .5], 'String', 'Hello')
```

Note that using the convenience form of an object creation function can cause subtle differences in behavior when compared to formal property name/property value syntax. See the reference manual for specific information on the calling syntax of object creation routines.

A Note About Property Names

By convention, MATLAB documentation capitalizes the first letter of each word that makes up a property name, such as `LineStyle` or `XMinorTickMode`. While this makes property names easier to read, MATLAB does not check for uppercase letters. In addition, you need use only enough letters to identify the name uniquely, so you can abbreviate most property names.

In M-files, however, using the full property name can prevent problems with futures releases of MATLAB if a shortened name is no longer unique because of the addition of new properties.

Using set and get

The `set` and `get` functions specify and retrieve the value of existing graphics object properties. They also allow you to list possible values for properties that have a fixed set of values.

Setting Property Values

See the “Accessing Object Handles” section for information on finding the handle of an existing object.

You can change the properties of an existing object using the `set` function and the handle returned by the creating function. For example, this statement moves the *y*-axis to the right side of the plot on the current Axes:

```
set(gca, 'YAxisLocation', 'right')
```

If the handle argument is a vector, MATLAB sets the specified value on all identified objects.

You can specify property names and property values using structure arrays or cell arrays. This can be useful if you want to set the same properties on a number of objects. For example, you can define a structure to set Axes properties appropriately to display a particular graph:

```
view1.CameraViewAngleMode = 'manual';
view1.DataAspectRatio = [1 1 1];
view1.ProjectionType = 'Perspective';
```

To set these values on the current Axes, type:

```
set(gca, view1)
```

See the `set` function in the online MATLAB Function Reference.

Listing Possible Values

You can use `set` to display the possible values for many properties without actually assigning a new value. For example, this statement obtains the values you can specify for Line object markers:

```
set(obj_handle, 'Marker')
```

MATLAB returns a list of values for the `Marker` property for the type of object specified by `obj_handle`. Braces indicate the default value:

```
[ + | o | * | . | x | square | diamond | v | ^ | > | < | pentagram
  | hexagram | {none} ]
```

To see a list of all settable properties along with possible values of properties that accept string values, use `set` with just an object handle:

```
set(object_handle)
```

For example, for a Surface object, MATLAB returns:

```
CData
CDataScaling: [ {on} | off]
EdgeColor: [ none | {flat} | interp ] ColorSpec.
EraseMode: [ {normal} | background | xor | none ]
FaceColor: [ none | {flat} | interp | texturemap ] ColorSpec.
LineStyle: [ {-} | — | : | -. | none ]
.
.
.
Visible: [ {on} | off ]
```

If you assign the output of the `set` function to a variable, MATLAB returns the output as a structure array. For example,

```
a = set(gca);
```

The field names in `a` are the object's property names and the field values are the possible values for the associated property. For example,

```
a.GridLineStyle
ans =

    '-'
    '--'
    ':'
    '-.'
    'none'
```

returns the possible value for the Axes grid line styles. Note that while property names are not case sensitive, MATLAB structure field names are. For example,

```
a.gridlinestyle
??? Reference to non-existent field 'gridlinestyle'.
```

returns an error.

Getting Property Values

Use `get` to query the current value of a property or of all the object's properties. For example, check the value of the current `Axes PlotBoxAspectRatio` property:

```
get(gca, 'PlotBoxAspectRatio')

ans =
     1     1     1
```

MATLAB lists the values of all properties, where practical. However, for properties containing data, MATLAB lists the dimensions only (for example, `CurrentPoint` and `ColorOrder`):

```
AmbientLightColor = [1 1 1]
Box = off
CameraPosition = [0.5 0.5 2.23205]
CameraPositionMode = auto
CameraTarget = [0.5 0.5 0.5]
CameraTargetMode = auto
CameraUpVector = [0 1 0]
CameraUpVectorMode = auto
CameraViewAngle = [32.2042]
CameraViewAngleMode = auto
CLim: [0 1]
CLimMode: auto
Color: [0 0 0]
CurrentPoint: [ 2x3 double]
ColorOrder: [ 7x3 double]
.
.
.
Visible = on
```

You can obtain the data from the property by getting that property individually:

```
get(gca, 'ColorOrder')
ans =
    0         0    1.0000
    0    0.5000         0
    1.0000         0         0
    0    0.7500    0.7500
    0.7500         0    0.7500
    0.7500    0.7500         0
    0.2500    0.2500    0.2500
```

If you assign the output of `get` to a variable, MATLAB creates a structure array whose field names are the object property names, and field values are the current values of the named property.

For example, if you plot some data, `x` and `y`:

```
h = plot(x, y);
```

and get the properties of the Line object created by `plot`:

```
a = get(h);
```

You can now access the values of the Line properties using the field name. This call to the `text` function places the string '`x and y data`' at the first data point and colors the text to match the line color:

```
text(x(1), y(1), 'x and y data', 'Color', a.Color)
```

If `x` and `y` are matrices, `plot` draws one line per column. To label the plot of the second column of data, reference that Line:

```
text(x(1, 2), y(1, 2), 'Second set of data', 'Color', a(2).Color)
```


Querying Groups of Properties

You can define a cell array of property names and conveniently use it to obtain the values for those properties. For example, suppose you want to query the values of the Axes “camera mode” properties. First define the cell array:

```
camera_props(1) = {' CameraPosi ti onMode' };
camera_props(2) = {' CameraTargetMode' };
camera_props(3) = {' CameraUpVectorMode' };
camera_props(4) = {' CameraVi ewAngl eMode' };
```

Use this cell array as an argument to obtain the current values of these properties:

```
get(gca, camera_props)
ans =
    'auto' 'auto' 'auto' 'auto'
```

Factory-Defined Property Values

MATLAB defines values for all properties, which are used if you do not specify values as arguments or as defaults. You can obtain a list of all factory-defined values with the statement:

```
a = get(0, ' Factory' );
```

get returns a structure array whose field names are the object type and property name concatenated together, and field values are the factory value for the indicated object and property. For example, this field:

```
UimenuSel ecti onHi ghli ght: 'on'
```

indicates that the factory value for the Sel ecti onHi ghli ght property on Uimenu objects is on.

You can get the factory value of an individual property with:

```
get(0, ' FactoryOb jectTypePropertyName' )
```

For example

```
get(0, ' FactoryTextFontName' )
```

See the set and get functions in the online MATLAB Function Reference for more information.

Default Property Values

All object properties have “default” values built into MATLAB (i.e., factory-defined values). You can also define your own default values at any point in the object hierarchy.

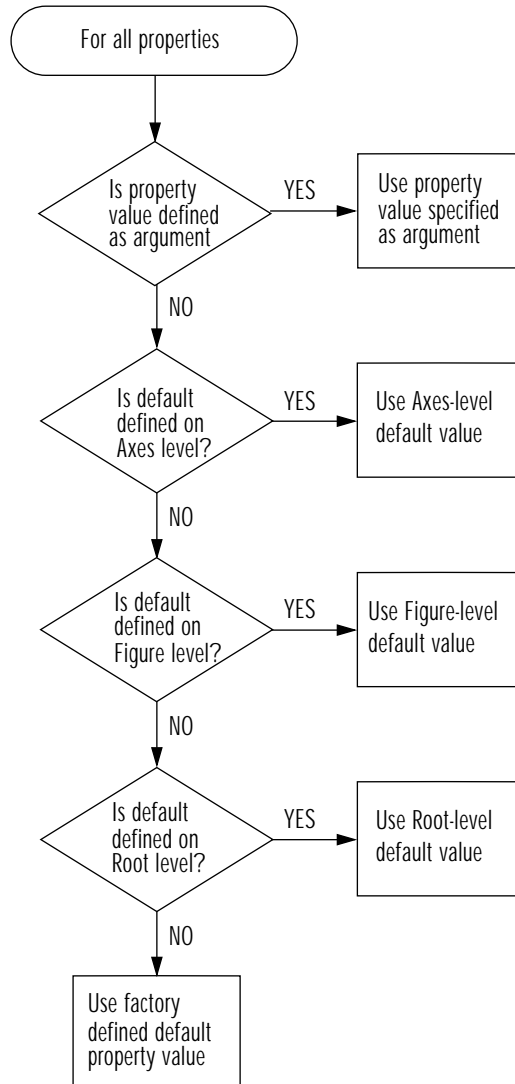
MATLAB’s search for a default value begins with the current object and continues through the object’s ancestors until it finds a user-defined default value or until it reaches the factory-defined value. Therefore, a search for property values is always satisfied.

The closer to the Root of the hierarchy you define the default, the broader is its scope. If you specify a default value for Line objects on the Root level, MATLAB uses that value for all Lines (since the Root is at the top of the hierarchy). If you specify a default value for Line objects on the Axes level, then MATLAB uses that value for Line objects drawn only in that Axes.

If you define default values on more than one level, the value defined on the closest ancestor takes precedence since MATLAB terminates the search as soon as it finds a value.

Note that setting default values affects only those objects created after you set the default. Existing graphics objects are not affected.

This diagram shows the steps MATLAB follows in determining the value of a graphics object property:



Specifying Default Values

To specify default values, create a string beginning with the word `Default` followed by the object type and finally the object property. For example, to specify a default value of 1.5 points for the `LineLineWidth` property at the level of the current Figure, use the statement:

```
set(gcf, 'DefaultLineLineWidth', 1.5)
```

The string, `DefaultLineLineWidth` identifies the property as a Line property. To specify the Figure color, use `DefaultFigureColor`. Note that it is meaningful to specify a default Figure color only on the Root level:

```
set(0, 'DefaultFigureColor', 'b')
```

Use `get` to determine what default values are currently set on any given object level, for example:

```
get(gcf, 'default')
```

returns all default values set on the current Figure.

Setting Properties to the Default

Specifying a property value of `'default'` sets the property to the first encountered default value defined for that property. For example, these statements result in a green `SurfaceEdgeColor`:

```
set(0, 'DefaultSurfaceEdgeColor', 'k')
h = surface(peaks);
set(gcf, 'DefaultSurfaceEdgeColor', 'g')
set(h, 'EdgeColor', 'default')
```

Since a default value for `SurfaceEdgeColor` exists on the Figure level, MATLAB encounters this value first and uses it instead of the default `EdgeColor` defined on the Root.

Removing Default Values

Specifying a property value of `'remove'` gets rid of user-defined default values. The statement,

```
set(0, 'DefaultSurfaceEdgeColor', 'remove')
```

removes the definition of the default `SurfaceEdgeColor` from the Root.

Setting Properties to Factory-Defined Values

Specifying a property value of 'factory' sets the property to its factory-defined value. (The descriptions of the object creation functions in the online MATLAB Function Reference indicate the factory settings for properties having predefined sets of values.)

For example, these statements set the `EdgeColor` of `Surface` `h` to black (its factory setting) regardless of what default values you have defined.

```
set(gcf, 'DefaultSurfaceEdgeColor', 'g')
h = surface(peaks);
set(h, 'EdgeColor', 'factory')
```

Reserved Words

Setting a property value to `default`, `remove`, or `factory` produces the effect described in the previous sections. In order to set a property to one of these words (e.g., a `Text` or `UicontrolString` property set to the word "Default") you must precede the word with the backslash character. For example,


```
h = uicontrol('Style','edit','String','\Default');
```

Examples – Setting Defaults

The `plot` function cycles through the colors defined by the `AxesColorOrder` property when displaying multiline plots. If you define more than one value for the `AxesLineStyleOrder` property, MATLAB increments the linestyle after each cycle through the colors.

You can set default property values that cause the `plot` function to produce graphs using varying linestyles, but not varying colors. This is useful when working on a monochrome display or printing on a black and white printer.

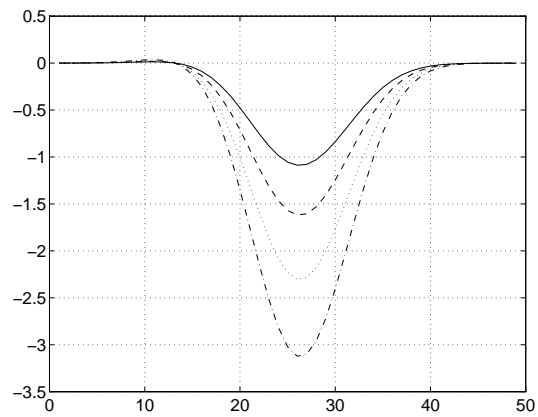
First Example. This example creates a Figure with a white plot (Axes) background color, then sets default values for Axes objects on the Root level:

Create a Figure and use a  white background color scheme

```
whitebg('w')
set(0, 'DefaultAxesColorOrder', [0 0 0], ...
      'DefaultAxesLineStyleOrder', '-|—|:|-.')
```

Whenever you call `plot`,

```
Z = peaks; plot(1:49, Z(4:7,:))
```



it uses one color for all data plotted because the `Axes ColorOrder` contains only one color, but cycles through the line styles defined for `LineStyleOrder`.

Second Example. This example sets default values on more than one level in the hierarchy. These statements create two Axes in one Figure window, setting default values on the Figure level and the Axes level.

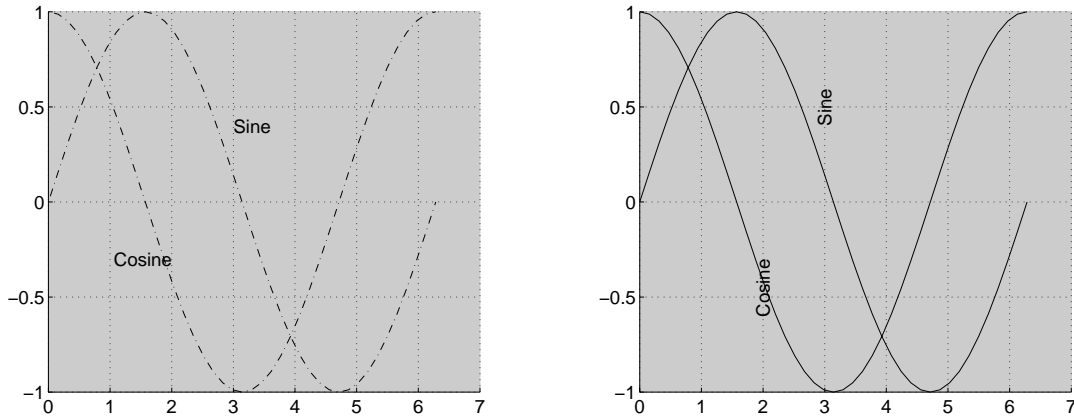
```
t = 0: pi /20: 2*pi ;
s = sin(t);
c = cos(t);
```

Set default value for Axes **Color** property — `figh = figure('Position', [30 100 800 350], ...
'DefaultAxesColor', [.8 .8 .8])`

Set default value for Line **LineStyle** property used in the first Axes. — `axh1 = subplot(1,2,1); grid on
set(axh1, 'DefaultLineStyle', '-. ')
line('XData', t, 'YData', s)
line('XData', t, 'YData', c)
text('Position', [3 .4], 'String', 'Sine')
text('Position', [2 -.3], 'String', 'Cosine', ...
'HorizontalAlignment', 'right')`

Set default value for Text **Rotation** property used in the second Axes. — `axh2 = subplot(1,2,2); grid on
set(axh2, 'DefaultTextRotation', 90)
line('XData', t, 'YData', s)
line('XData', t, 'YData', c)
text('Position', [3 .4], 'String', 'Sine')
text('Position', [2 -.3], 'String', 'Cosine', ...
'HorizontalAlignment', 'right')`

Issuing the same `line` and `text` statements to each subplot region results in a different display, reflecting different default settings:



Since the default `Axes Color` property is set on the Figure level of the hierarchy, MATLAB creates both Axes with the specified gray background color.

The Axes on the left (subplot region 121) defines a dash-dot line style (`-.`) as the default, so each call to the `line` function uses dash-dot lines. The Axes on the right does not define a default linestyle so MATLAB uses solid lines (the factory setting for Lines).

The Axes on the right defines a default `Text Rotation` of 90 degrees, which rotates all Text by this amount. MATLAB obtains all other property values from their factory settings, which results in nonrotated text on the left.

To install default values whenever you run MATLAB, specify them in your startup. `m` file. Note that MATLAB may install default values for some appearance properties when started by calling the `colordef` command. See the online MATLAB Function Reference for more information.

Accessing Object Handles

MATLAB assigns a handle to every graphics object it creates. All object creation functions optionally return the handle of the created object. If you want to access the object's properties (e.g., from an M-file) you should assign its handle to a variable at creation time to avoid searching for it later. However, you can always obtain the handle of an existing object with the `findobj` function or by listing its parent's `Children` property. See the "Protecting Figures and Axes" section for more information.

The Root object's handle is always zero. The handle of a Figure is either:

- An integer that, by default, displays in the window title bar
- A floating point number requiring full MATLAB internal precision

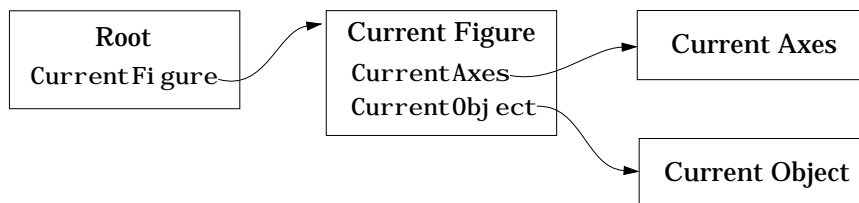
The Figure `IntegerHandle` property controls which type of handle the Figure receives.

All other graphics object handles are floating-point numbers. You must maintain the full precision of these numbers when you reference handles. Rather than attempting to read handles off the screen and retype them, it is necessary to store the value in a variable and pass that variable whenever a handle is required.

The Current Figure, Axes, and Object

An important concept in Handle Graphics is that of being current. The current Figure is the window designated to receive graphics output. Likewise, the current Axes is the target for commands that create Axes children. The current object is the last graphics object created or clicked on by the mouse.

MATLAB stores the three handles corresponding to these objects in the ancestor's property list:



These properties enable you to obtain the handles of these key objects:

```
get(0, 'CurrentFigure');
get(gcf, 'CurrentAxes');
get(gcf, 'CurrentObject');
```

The following commands are shorthand notation for the `get` statements:

- `gcf` – returns the value of the Root `CurrentFigure` property
- `gca` – returns the value of the current Figure's `CurrentAxes` property
- `gco` – returns the value of the current Figure's `CurrentObject` property

You can use these commands as input arguments to functions that require object handles. For example, you can click on a Line object and then use `gco` to specify the handle to the `set` command:

```
set(gco, 'Marker', 'square')
```

or list the values of all current Axes properties with,

```
get(gca)
```

You can get the handles of all the graphic objects in the current Axes (except those with hidden handles),

```
h = get(gca, 'Children');
```

and then determine the types of the objects:

```
get(h, 'type')
ans =
    'text'
    'patch'
    'surface'
    'line'
```

While `gcf` and `gca` provide a simple means of obtaining the current Figure and Axes handles, they are less useful in M-files. This is particularly true if your M-file is part of an application layered on MATLAB where you do not necessarily have knowledge of user actions that can change these values.

See the “Controlling Graphics Output” section for information how to prevent users from accessing the handles of graphics objects that you want to protect.

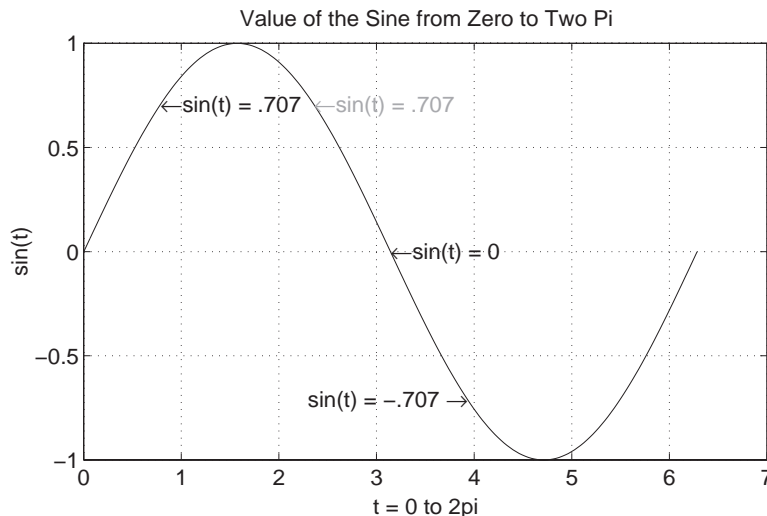
Searching for Objects by Property Values — findobj

The `findobj` function provides a means to traverse the object hierarchy quickly and obtain the handles of objects having specific property values. If you do not specify a starting object, `findobj` searches from the Root object, finding all occurrences of the property name/property value combination you specify.

See also the `findobj` function description in the online MATLAB Function Reference for more information.

Example

This plot of the sine function contains Text objects labeling particular values of function:



Suppose you want to move the text string labeling the value $\sin(t) = .707$ from its current location at $[\pi/4, \sin(\pi/4)]$ to the point $[3\pi/4, \sin(3\pi/4)]$ where the function has the same value (shown grayed out in the picture). To do this, you need to determine the handle of the Text object labeling that point and change its `Position` property.

To use `findobj`, pick a property value that uniquely identifies the object. In this case, the `TextString` property:

```
text_handle = findobj('TextString', '\leftarrow sin(t) = .707');
```

Next move the object to the new position, defining the Text Position in Axes units:

```
set(text_handle, 'Position', [3*pi/4, sin(3*pi/4), 0])
```

`findobj` also lets you restrict the search by specifying a starting point in the hierarchy, instead of beginning with the Root object. This results in faster searches if there are many objects to search. In the previous example, you know the Text object of interest is in the current Axes so you can type:

```
text_handle = findobj(gca, 'String', '\leftarrow sin(t) = .707');
```

Copying and Deleting Objects

You can copy objects from one parent to another using the `copyobj` function. The new object differs from the original object only in the value of its Parent property and its handle; it is otherwise a clone of the original. You can copy a number of objects to a new parent, or one object to a number of new parents as long as the result maintains the correct parent/child relationship.

When you copy an object having children objects, MATLAB copies all children as well.

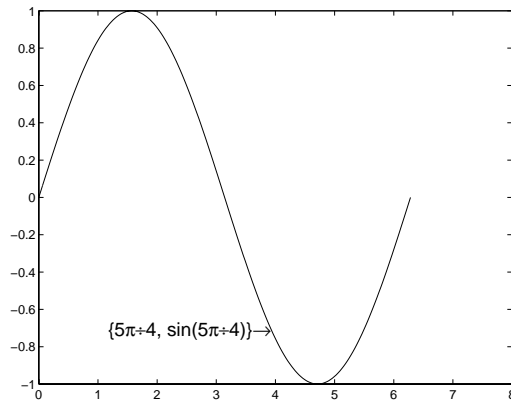
Example – Copying Objects

Suppose you are plotting a variety of data and want to label the point having the x - and y -coordinates determined by $5\pi \div 4$, $\sin(5\pi \div 4)$ in each plot. The `text` function allows you to specify the location of the label in the coordinates defined by the x - and y -axis limits, simplifying the process of locating the Text:

```
text('String', '\{5\pi \div 4, \sin(5\pi \div 4)\} \rightarrow', ...
     'Position', [5*pi/4, sin(5*pi/4), 0], ...
     'HorizontalAlignment', 'right')
```

In this statement, the `text` function:

- Labels the data point with the string $\{5\pi \div 4, \sin(5\pi \div 4)\}$, using TeX commands to draw a right-facing arrow and mathematical symbols.
- Specifies the Position in terms of the data being plotted.
- Places the data point to the right of the Text string by changing the Horizontal Alignment to `right` (the default is `left`).

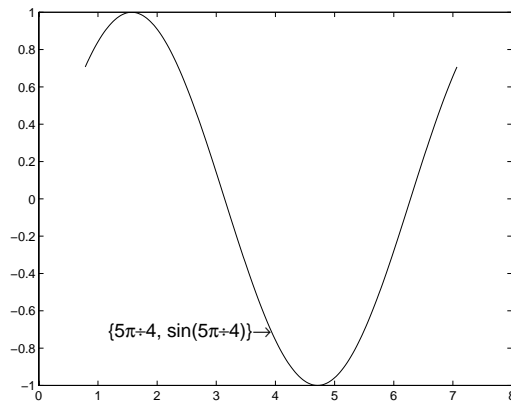


To label the same point with the same string in another plot, copy the Text using `copyobj`. Since the last statement did not save the handle to the Text object, you can find it using `findobj` and the 'String' property:

```
text_handle = findobj('String', ...
    '\{5\pi\di v4, \sin(5\pi\di v4)\}\rightarrow');
```

After creating the next plot, add the label by copying it from the first plot.

```
copyobj(text_handle, gca).
```



This particular example takes advantage of the fact that Text objects define their location in the Axes' data space. Therefore the Text Position property did not need to change from one plot to another.

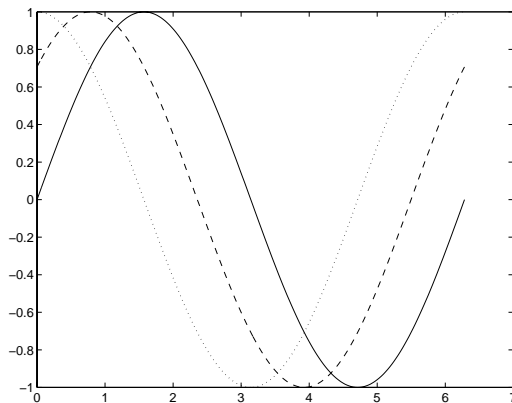
See the `copyobj` reference page for a complete discussion of the various ways you can use `copyobj`.

Deleting Objects

You can remove a graphics object with the `delete` command, using the object's handle as an argument. For example, you can delete the current Axes (and all of its descendants) with the statement:

```
delete(gca)
```

You can use `findobj` to get the handle of a particular object you want to delete. For example, to find the handle of the dotted Line in this multiline plot,



use `findobj` to locate the object whose `LineStyle` property is `'.'`

```
line_handle = findobj('LineStyle',':');
```

then use this handle with the `delete` command:

```
delete(line_handle)
```

You can combine these two statements, substituting the `findobj` statement for the handle:

```
delete(findobj('LineStyle',':'))
```

Controlling Graphics Output

MATLAB allows many Figure windows to be open simultaneously during a session. A MATLAB application may create Figures to display graphical user interfaces as well as plotted data. It is necessary then to protect some Figures from becoming the target for graphics display and to prepare (e.g., reset properties and clear existing objects from) others before receiving new graphics.

This section discusses how to control where and how MATLAB displays graphics output. Topics include:

- Specifying the target for graphics output
- Preparing the Figure and Axes to accept new objects
- Protecting Figures and Axes from becoming targets
- Accessing the handles of protected Figure and Axes

Specifying the Target for Graphics Output

By default, MATLAB functions that create graphics objects display them in the current Figure and current Axes (if an Axes child). You can direct the output to another parent by explicitly specifying the `Parent` property with the creating function. For example,

```
plot(1:10, 'Parent', axes_handle)
```

where `axes_handle` is the handle of the target Axes. The `ui control` and `ui menu` functions have a convenient syntax that enables you to specify the parent as the first argument,

```
ui control (Figure_handle, ... )
ui menu (parent_menu_handle, ... )
```

or you can set the `Parent` property. See the online MATLAB Function Reference for more information.

Preparing Figures and Axes for Graphics

By default, commands that generate graphics output display the graphics objects in the current Figure without clearing or resetting Figure properties. However, if the graphics objects are Axes children, MATLAB clears the Axes and resets most Axes properties to their default values before displaying the objects.

You can change this behavior by setting the Figure and Axes NextPlot property.

The NextPlot Property

MATLAB high-level graphics functions check the value of the NextPlot properties to determine whether to add, clear, or clear and reset the Figure and Axes before drawing. Low-level object creation functions do not check the NextPlot properties. They simply add the new graphics objects to the current Figure and Axes.

Low-level functions are designed primarily for use in M-files where you can implement whatever drawing behavior you want. However, when developing a MATLAB-based application, controlling MATLAB's drawing behavior is essential to creating a program that behaves predictably.

This table summarizes the possible values for the NextPlot property:

NextPlot	Figure	Axes
add	Add new graphics objects without clearing or resetting the current Figure. (Default setting)	Add new graphics objects without clearing or resetting the current Axes.
replacechildren	Remove all child objects, but do not reset Figure properties. Equivalent to clf.	Remove all child objects, but do not reset Axes properties. Equivalent to cla.
replace	Remove all child objects and reset Figure properties to their defaults. Equivalent to clf reset.	Remove all child objects and reset Axes properties to their defaults. Equivalent to cla reset. (Default setting)

Note that a reset returns all properties, except Position and Units, to their default values.

The hold command provides convenient access to the NextPlot properties. The statement

```
hold on
```

sets both Figure and Axes NextPlot to add.

The statement

```
hold off
```

sets the Axes `NextPlot` property to `replace`.

Controlling Graphics Output with the `newplot` Function

MATLAB provides the `newplot` function to simplify the process of writing graphics M-files that conform to the settings of the `NextPlot` properties.

`newplot` checks the values of the `NextPlot` properties and takes the appropriate action based on these values. You should place `newplot` at the beginning of any M-file that calls object creation functions.

When your M-file calls `newplot`, these possible actions occur:

1 `newplot` checks the current Figure's `NextPlot` property:

- If there are no Figures in existence, `newplot` creates one and makes it the current Figure.
- If the value of `NextPlot` is `add`, `newplot` makes the Figure the current Figure.
- If the value of `NextPlot` is `replacechildren`, `newplot` deletes the Figure's children (Axes objects and their descendents) and makes this Figure the current Figure.
- If the value of `NextPlot` is `replace`, `newplot` deletes the Figure's children, resets the Figure's properties to the defaults, and makes this Figure the current Figure.

2 `newplot` checks the current Axes' `NextPlot` property:

- If there are no Axes in existence, `newplot` creates one and makes it the current Axes.
- If the value of `NextPlot` is `add`, `newplot` makes the Axes the current Axes.
- If the value of `NextPlot` is `replacechildren`, `newplot` deletes the Axes' children and makes this Axes the current Axes.
- If the value of `NextPlot` is `replace`, `newplot` deletes the Axes' children, resets the Axes' properties to the defaults, and makes this Axes the current Axes.

MATLAB's Default Behavior. Consider the default situation where the Figure `NextPlot` property is `add` and the Axes `NextPlot` property is `replace`. When you call `newplot`, it:

- 1 Checks the value of the current Figure's `NextPlot` property (which is `add`) and determines MATLAB can draw into the current Figure with no further action (if there is no current Figure, `newplot` creates one, but does not recheck its `NextPlot` property).
- 2 Checks the value of the current Axes' `NextPlot` property (which is `replace`), deletes all graphics objects from the Axes, reset all Axes properties (except `Position` and `Units`) to their defaults, and returns the handle of the current Axes.

Example – Using `newplot`

To illustrate the use of `newplot`, this example creates a function that is similar to the built-in `plot` function, except it automatically cycles through different linestyles instead of using different colors for multiline plots:

`newplot` returns the handle of the current Axes.

Use Axes handle to set Axes properties and to identify Figure handle.

```
function my_plot(x, y)
    cax = newplot;
    LS0 = ['-'; '-.-'; ':'; '-.'];
    {
        set(cax, 'FontName', 'Times', 'FontAngle', 'italic')
        set(get(cax, 'Parent'), 'MenuBar', 'none')
    }
    line_handles = line(x, y, 'Color', 'b');
    style = 1;
    for i = 1:length(line_handles)
        if style > length(LS0), style = 1; end
        set(line_handles(i), 'LineStyle', LS0(style, :))
        style = style + 1;
    end
    grid on
```

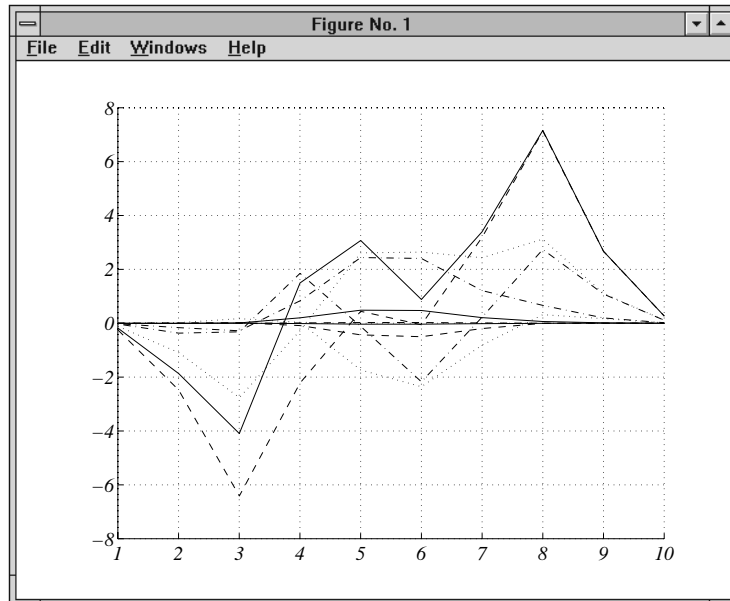
See the `line` function in the online MATLAB Function Reference for a description of its various forms.

The function `my_plot` uses the informal `line` function syntax to plot the data. This provides the same flexibility in input argument dimension that the built-in `plot` function supports. The `line` function does not check the value of the Figure or Axes `NextPlot` property. However, because `my_plot` calls `newplot`, it behaves the same way the high-level `plot` function does – with default values in place, `my_plot` clears and reset the Axes each time you call it.

`my_plot` uses the handle returned by `newplot` to access the target Figure and Axes. This example sets Axes font properties and disables the Figure's menu bar. Note how the Figure handle is obtained via the Axes Parent property.

Typical output for this function is:

```
my_plot(1:10, peaks(10))
```



This example illustrates the basic structure of graphics M-files:

- Call `newplot` early to conform to the NextPlot properties and to obtain the handle of the target Axes.
- Reference the Axes handle returned by `newplot` to set any Axes properties or to obtain the Figure's handle.
- Call object creation functions to draw graphics objects with the desired characteristics.

MATLAB's default settings for the NextPlot properties facilitate writing M-files that adhere to MATLAB's standard behavior: reuse the Figure window, but clear and reset the Axes with each new graph. Other values for these properties allow you to implement different behaviors.

Replacing Only the Children Objects —`replacechildren`

The `replacechildren` value for `NextPlot` causes `newplot` to remove child objects from the Figure or Axes, but does not reset any property values (except the list of handles contained in the `Children` property).

This can be useful after setting properties you want to use for subsequent graphs without having to reset properties. For example, if you type on the command line

```
set(gca, 'ColorOrder', [0 0 1], 'LineStyleOrder', '-|—|:|-.', ...
    'NextPlot', 'replacechildren')
plot(x, y)
```

`plot` produces the same output as the M-file `my_plot` in the previous section, but only within the current Axes. Calling `plot` still erases the existing graph (i.e., deletes the Axes children), but it does not reset Axes properties. The values specified for the `ColorOrder` and `LineStyleOrder` properties remain in effect.

Testing for Hold State

There are situations in which your M-file should change the visual appearance of the Axes to accommodate new graphics objects. For example, if you want the M-file `my_plot` from the previous example to accept 3-D data, it makes sense to set the view to 3-D when the input data has *z*-coordinates.

However, to be consistent with the behavior of MATLAB's high-level routines, it is a good practice to test if `hold` is on before changing parent Axes or Figure properties. When `hold` is on, the Axes and Figure `NextPlot` properties are both set to `add`.

The M-file, `my_plot3`, accepts 3-D data and also checks the hold state, using `ishold`, to determine if it should change the view:

```
function my_plot3(x, y, z)
    cax = newplot;
    hold_state = ishold;
    LS0 = ['-'; '-'; ':'; '-'];
    if nargin == 2
        hlines = line(x, y, 'Color', 'k');
        if ~hold_state
            view(2)
        end
    elseif nargin == 3
        hlines = line(x, y, z, 'Color', 'k');
        if ~hold_state
            view(3)
        end
    end
    ls = 1;
    for hindex = 1:length(hlines)
        if ls > length(LS0), ls = 1; end
        set(hlines(hindex), 'LineStyle', LS0(ls, :))
        ls = ls + 1;
    end
```

`ishold` tests the current hold state.

Change the view only if hold is off.

If `hold` is on when you call `my_plot3`, it does not change the view. If `hold` is off, `my_plot3` sets the view to 2-D or 3-D, depending on whether there are two or three input arguments.

Protecting Figures and Axes

There are situations in which it is important to prevent particular Figures or Axes from becoming the target for graphics output (i.e., preventing them from becoming the `gcf` or `gca`). An example of this is a Figure containing the `Uicontrols` that implement a user interface.

You can prevent MATLAB from drawing into a particular Figure or Axes by removing its handle from the list of handles that are visible to the `newplot` function, as well as any other functions that either return or implicitly reference handles (i.e., `gca`, `gcf`, `gco`, `cla`, `clf`, `close`, and `findobj`). Two properties control handle hiding: `HandleVisibility` and `ShowHiddenHandles`.

HandleVisibility Property

`HandleVisibility` is a property of all objects. It controls the scope of handle visibility within three different ranges. Property values can be:

- `on` – The object's handle is available to any function executed on the MATLAB command line or from an M-file. This is the default setting.
- `callback` – The object's handle is hidden from all functions executing on the command line, even if it is on the top of the screen stacking order. However, during callback routine execution (MATLAB statements or functions that execute in response to user action), the handle is visible to all functions, such as `gca`, `gcf`, `gco`, `findobj`, and `newplot`. This setting enables callback routines to take advantage of MATLAB's handle access functions, while ensuring that users typing at the command line do not inadvertently disturb a protected object.
- `off` – The object's handle is hidden from all functions executing on the command line and in callback routines. This setting is useful when you want to protect objects from possibly damaging user commands.

For example, if a GUI accepts user input in the form of text strings, which are then evaluated (using the `eval` function) from within the callback routine, a string such as `'close all'` could destroy the GUI. To protect against this situation, you can temporarily set `HandleVisibility` to `off` on key objects:

```
user_input = get(editbox_handle, 'String');
set(gui_handles, 'HandleVisibility', 'off')
eval(user_input)
set(gui_handles, 'HandleVisibility', 'commandline')
```

Values Returned by `gca` and `gcf`. When a protected Figure is topmost on the screen, but has nonprotected Figures stacked beneath it, `gcf` returns the topmost unprotected Figure in the stack. The same is true for `gca`. If no unprotected Figures or Axes exist, calling `gcf` or `gca` causes MATLAB to create one in order to return its handle.

Accessing Protected Objects

The `Root ShowHiddenHandles` property enables and disables handle visibility control. By default, `ShowHiddenHandles` is `off`, which means MATLAB obeys the setting of the `HandleVisibility` property. When set to `on`, all handles are visible from the command line and within callback routines. This can be useful

when you want access to all graphics objects that exist at a given time, including the handles of Axes text labels, which are normally hidden.

The `close` function also allows access to nonvisible Figures using the `hidden` option. For example,

```
close('hidden')
```

closes the topmost Figure on the screen, even if it is protected. Combining `all` and `hidden` options,

```
close('all', 'hidden')
```

closes all Figures.

The Close Request Function

MATLAB executes a callback routine defined by the Figure's `CloseRequestFcn` whenever you:

- Issue a `close` command on a Figure.
- Quit MATLAB while there are visible Figures. (If a Figure's `Visible` property is set to `off`, MATLAB does not execute its close request function when you quit MATLAB; the Figure is just deleted).
- Close a Figure from the windowing system using a close box or a close menu item.

The close request function enables you to prevent or delay the closing of a Figure or the termination of a MATLAB session. This is useful to perform such actions as:

- Displaying a dialog box requiring the user to confirm the action
- Saving data before closing
- Preventing unintentional command-line deletion of a graphical user interface built with MATLAB

The default callback routine for the `CloseRequestFcn` is an M-file called `closereq`. It contains the statements:

```
shh=get(0, 'ShowHiddenHandles');
set(0, 'ShowHiddenHandles', 'on');
delete(get(0, 'CurrentFigure'));
set(0, 'ShowHiddenHandles', shh);
```

This callback disables `HandleVisibility` control by setting the `Root.ShowHiddenHandles` property to on, which makes all Figure handles visible.

Quitting MATLAB. When you quit MATLAB, the current Figure's `CloseRequestFcn` is called, and if the Figure is deleted, the next Figure in the Root's list of children (i.e., the Root's `Children` property) becomes the current Figure, and its `CloseRequestFcn` is in turn executed, and so on.

If you change a Figure's `CloseRequestFcn` so that it does not delete the Figure (e.g., defining this property as an empty string), then issuing the close command on that Figure does not cause it to be deleted. Furthermore, if you attempt to quit MATLAB, the quit is aborted because MATLAB does not delete the Figure.

Errors in the Close Request Function. If the `CloseRequestFcn` generates an error when executed, MATLAB aborts the close operation. However, errors in the `CloseRequestFcn` do not abort attempts to quit MATLAB. If an error occurs in a Figure's `CloseRequestFcn`, MATLAB closes the Figure unconditionally following a `quit` or `exit` command.

Overriding the Close Request Function. The `delete` command always deletes the specified Figure, regardless of the value of its `CloseRequestFcn`. For example, the statement:

```
delete(get(0, 'Children'))
```

deletes all Figures whose handles are not hidden (i.e., the `HandleVisibility` property is set to off). If you want to delete all Figures regardless of whether their handles are hidden, you can set the `Root.ShowHiddenHandles` property to on. The `Root.Children` property then contains the handles of all Figures. For example, statements:

```
set(0, 'ShowHiddenHandles', 'yes')
delete(get(0, 'Children'))
```

unconditionally delete all Figures.

Validity versus Visibility

All handles remain valid regardless of whether they are visible or not. If you know an object's handle, you can set and get its properties. By default, Figure handles are integers which are displayed at the top of the window. You can provide further protection to Figures by setting the `IntegerHandle` property to

off. MATLAB then uses a floating-point number for Figure handles. See the `figure` function in the online MATLAB Function Reference for a list of all Figure properties.

Efficient Programming

Graphics M-files frequently use handles to access property values and to direct graphics output to a particular target. MATLAB provides utility routines that return the handles to key objects (such as the current Figure and Axes). In M-files, however, these utilities may not be the best way to obtain handles because:

- Querying MATLAB for the handle of an object or other information is less efficient than storing the handle in a variable and referencing that variable.
- The current Figure, Axes, or object may change during M-file execution due to user interaction.

Save Information First

It is a good practice to save relevant information about MATLAB's state in the beginning of your M-file. For example, you can begin an M-file with

```
cax = newplot;
cfig = get(cax, 'Parent');
hold_state = ishold;
```

rather than querying this information each time you need it. Remember that utility commands like `ishold` obtain the values they return whenever called. (The `ishold` command issues a number of `get` commands and string compares (`strcmp`) to determine the hold state.)

If you are temporarily going to alter the hold state within the M-file, you should save the current values of the `NextPlot` properties so you can reset them later:

```
ax_nextplot = lower(get(cax, 'NextPlot'));
fig_nextplot = lower(get(cfig, 'NextPlot'));
.
.
.
set(cax, 'NextPlot', ax_nextplot)
set(cfig, 'NextPlot', fig_nextplot)
```

Properties Changed by Built-In Functions

To achieve their intended effect, many built-in functions change Axes properties, which can then affect the workings of your M-file. This table lists MATLAB's built-in graphics functions and the properties they change. Note that these properties change only if `hold` is off.

Function	Axes Property: Set To
<code>fill</code>	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view
<code>fill3</code>	CameraPosition: 3-D view CameraTarget: 3-D view CameraUpVector: 3-D view CameraViewAngle: 3-D view XScale: linear YScale: linear ZScale: linear
<code>image</code> (high-level)	Box: on Layer: top CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XDir: normal XLim: <code>[0 size(CData, 1)]+0.5</code> XLimMode: manual YDir: reverse YLim: <code>[0 size(CData, 2)]+0.5</code> YLimMode: manual

Function	Axes Property: Set To
loglog	Box: on CameraPosi ti on: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraVi ewAngl e: 2-D view XScale: log YScale: log
plot	Box: on CameraPosi ti on: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraVi ewAngl e: 2-D view
plot3	CameraPosi ti on: 3-D view CameraTarget: 3-D view CameraUpVector: 3-D view CameraVi ewAngl e: 3-D view XScale: linear YScale: linear ZScale: linear
semilogx	Box: on CameraPosi ti on: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraVi ewAngl e: 2-D view XScale: log YScale: linear

Function	Axes Property: Set To
semi logy	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: linear YScale: log

The *Figures* and *Axes* chapters discuss some important features that are under the control of Figure and Axes properties.

Figures

Introduction	9-2
Figure Properties	9-3
Positioning Figures	9-5
The Position Vector	9-5
Example — Specifying Figure Position	9-7
Controlling Color	9-8
Indexed Color Displays	9-8
Colormap Colors and Fixed Colors	9-9
Using a Large Number of Colors	9-10
Nonactive Figures and Shared Colors	9-12
Dithering Truecolor on Indexed Color Systems	9-13
Rendering Options	9-15
Backing Store	9-15
Z-Buffer	9-15
Figure Pointers	9-17
Custom Pointers	9-18
Printing Figures	9-21
Positioning the Figure on the Printed Page	9-21
Examples — Readjusting PaperPosition	9-23
Reversing Figure Colors	9-24
Interactive Graphics	9-27

Introduction

Figure graphics objects are the windows in which MATLAB displays graphical output. Figure properties allow you to control many aspects of these windows, such as their size and position on the screen, the coloring of graphics objects displayed within them, and the scaling of printed pictures.

This chapter discusses some of the features that are implemented through Figure properties and provides examples of how to use these features. The following table lists all Figure properties arranged by function. It provides an overview of the characteristics affected by Figure properties.

Figure Properties

This table lists Figure properties arranged in nine functional categories. See the `figure` function in the online MATLAB Function Reference for the most current list and descriptions of each individual property.

Category	Properties		
Style and appearance	MenuBar	Name	NumberTitle
	Resize	Visible	WindowState
	Color		
General information	Children	Parent	Position
	Tag	Type	Units
	UserData		
Colormap	Colormap	DitherMap	DitherMapMode
	FixedColors	MinColorMap	ShareColors
Rendering graphics objects	BackingStore	Renderer	RendererMode
Current selections	CurrentAxes	CurrentCharacter	CurrentMenu
	CurrentObject	CurrentPoint	SelectionType
Callback routine execution	ButtonDownFcn	CloseRequestFcn	
	DeleteFcn	KeyPressFcn	ResizeFcn
	BusyAction	Interruptible	WindowButtonUpFcn
	WindowButtonDownFcn WindowButtonMotionFcn		
Pointer definition	Pointer	PointerShapeCData	PointerShapeHotSpot
Figure handles	IntegerHandle	HandleVisibility	NextPlot
Printing	InvertHardcopy	PaperOrientation	PaperPosition

Category	Properties		
	PaperPosi ti onMode	PaperSi ze	PaperType
	PaperUni ts		

Positioning Figures

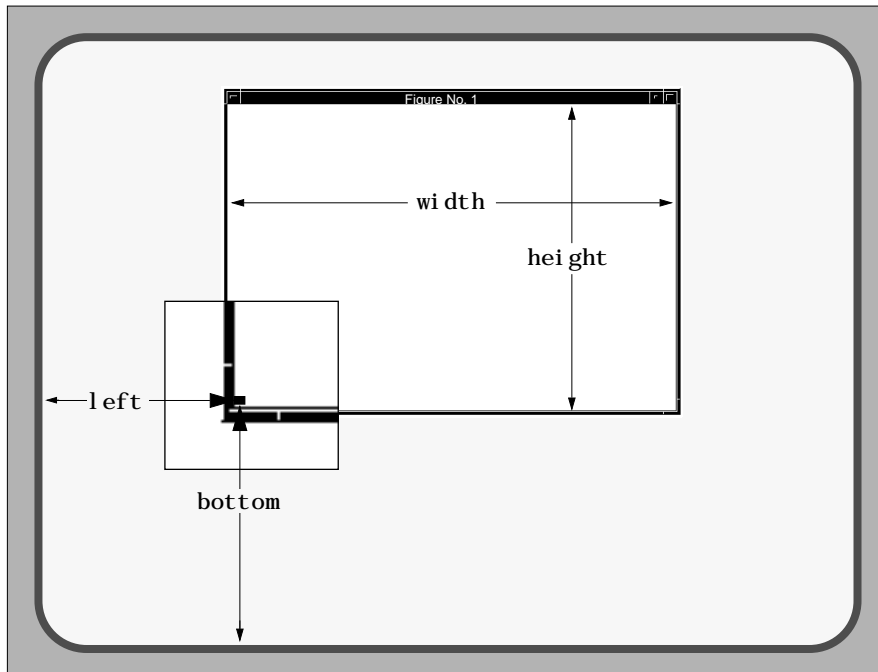
The `Figure Position` property controls the size and location of the Figure window on the Root screen. At startup, MATLAB determines the size of your computer screen and defines a default value for `Position`. This default creates Figures about one-quarter of the screen's size and places them centered left to right and in the top half of the screen.

The Position Vector

MATLAB defines the `Figure Position` property as a vector:

`[left bottom width height]`

`left` and `bottom` define the position of the first addressable pixel in the lower-left corner of the window, specified with respect to the lower-left corner of the screen. `width` and `height` define the size of the interior of the window (i.e., exclusive of the window border):



MATLAB does not measure the window border when placing the Figure; the `Position` property defines only the internal active area of the Figure window.

Since Figures are windows under the control of your computer's windowing system, you can move and resize Figures as you would any other windows. MATLAB automatically updates the `Position` property to the new values.

Units

The Figure's `Units` property determines the units of the values used to specify the position on the screen. Possible values for the `Units` property are:

```
set(gcf, 'Units')
[ inches | centimeters | normalized | points | {pixels} ]
```

with `pixels` being the default. These choices allow you to specify the Figure size and location in absolute units (such as inches) if you want the window to always be a certain size, or in units relative to the screen size (such as pixels).

Determining Screen Extent

Whatever units you use, it is important to know the extent of the screen in those units. You can obtain this information from the `RootScreenSize` property. For example:

```
get(0, 'ScreenSize')
ans =
    1    1 1152  900
```

In this case, the screen is 1152 by 900 pixels. MATLAB returns the `ScreenSize` in the units determined by the `RootUnits` property. For example,

```
set(0, 'Units', 'normalized')
```

normalizes the values returned by `ScreenSize`:

```
get(0, 'ScreenSize')
ans =
    0    0    1    1
```

Defining the Figure `Position` in terms of the `ScreenSize` in normalized units makes the specification independent of variations in screen size. This is useful if you are writing an M-file that is to be used on different computer systems.

Example — Specifying Figure Position

Suppose you want to define two Figure windows that occupy the upper third of the computer screen (e.g., one for Uicontrols and the other to display data). To position the windows precisely, you must consider the window borders when calculating the size and offsets to specify for the `Position` properties.

The Figure `Position` property does not include the window borders, so this example uses a width of 5 pixels on the sides and bottom and 30 pixels on the top.

Ensure Root units are pixels
and get the size of the screen
Define the size and location
of the Figures

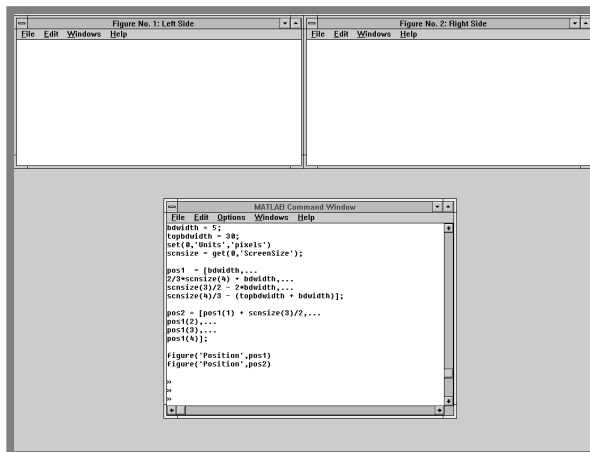
```

bdwidth = 5;
topbdwidth = 30;
set(0, 'Units', 'pixels')
scnsize = get(0, 'ScreenSize');
pos1 = [bdwidth, ...
        2/3*scnsize(4) + bdwidth, ...
        scnsize(3)/2 - 2*bdwidth, ...
        scnsize(4)/3 - (topbdwidth + bdwidth)];
pos2 = [pos1(1) + scnsize(3)/2, ...
        pos1(2), ...
        pos1(3), ...
        pos1(4)];
figure('Position', pos1)
figure('Position', pos2)

```

Create the Figures

The two Figures now occupy the top third of the screen:



Controlling Color

Figure properties control the way MATLAB uses your computer’s color resources. These properties influence both the speed of drawing and the accuracy of the colors used to display graphics. The properties discussed in this section include:

Property	Purpose
Colormap	The Figure colormap. An n -by-3 array of RGB values.
FixedColors	Specific colors used by the Figure that are not in the colormap.
MinColormap	The minimum number of system color table slots MATLAB uses for the Figure colormap.
ShareColors	Determines whether MATLAB shares colors with other Figure colormaps in the system color table.
Dithermap	A predefined colormap for displaying truecolor graphics objects on a pseudocolor system.
DithermapMode	Determines whether MATLAB uses the current dither colormap or creates one based on the colors specified for existing graphics objects.

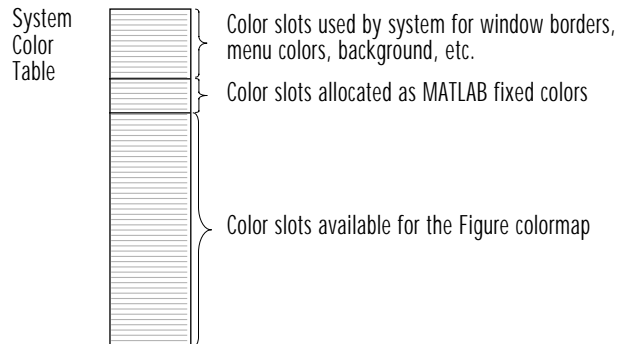
Indexed Color Displays

MATLAB defines a unique colormap as well as fixed colors (which are not part of the colormap) for each Figure object. Your computer system stores these color definitions in a color lookup table along with colors used for window borders, backgrounds, etc.

Indexed color systems associate a color slot (as opposed to a specific color) in the system color table with each screen pixel. When you activate an application program, for example, by moving the focus to a MATLAB Figure window, the system loads the colors associated with that program into the color table.

You can create a number of Figures on the screen at once, but only one has focus at any given time. When you change the focus to a particular Figure, the computer's operating system loads that Figure's colormap and all its fixed colors into the system color table.

For example, the color table might be allocated like this:



Colormap Colors and Fixed Colors

MATLAB maintains two categories of colors for each Figure – colors that are defined in the colormap and colors that are fixed, which do not change when you change the colormap. These two categories are used in different ways.

Only Surface, Patch, and Image objects use the colormap. MATLAB colors these objects based on the order the colors appear in the colormap.

Fixed colors are simply definitions of specific colors that MATLAB uses to color axis lines and labels and values you specify for object colors (i.e., the `Color`, `ColorOrder`, `FaceColor`, `EdgeColor`, `MarkerFaceColor`, and `MarkerEdgeColor` properties).

Defining Fixed Colors

When MATLAB creates a Figure, it defines three fixed colors:

```
figure
get(gcf, 'FixedColors')
ans =
    0.8000    0.8000    0.8000
         0         0         0
    1.0000    1.0000    1.0000
```

Creating an Axes includes the colors defined by the Axes ColorOrder property in the fixed color list, since it is more efficient to predefine these colors:

```
axes
get(gcf, 'FixedColors')
ans =
    0.8000    0.8000    0.8000
         0         0         0
    1.0000    1.0000    1.0000
         0         0    1.0000
         0    0.5000         0
    1.0000         0         0
         0    0.7500    0.7500
    0.7500         0    0.7500
    0.7500    0.7500         0
    0.2500    0.2500    0.2500
```

Any colors you define, for example,

```
set(surf_handle, 'EdgeColor', [.2 .8 .7])
```

also become part of the fixed color list. You can define as many fixed colors as you want without affecting the colors in the Figure colormap. However, fixed colors occupy color table slots that MATLAB cannot use for the colormap.

Using a Large Number of Colors

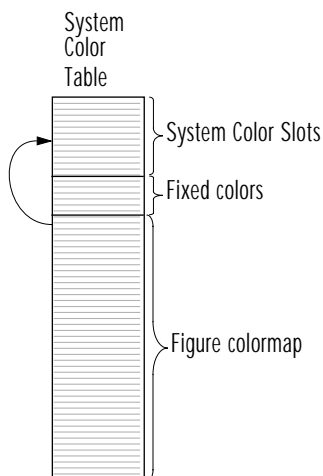
Overview. Set `MinColormap` to a number equal to the size of your colormap when you do not want MATLAB to approximate colors. However, this may cause nonactive windows to display with incorrect colors.

Problems can arise when you define a large colormap and/or a large number of fixed colors. If the number of color slots required exceeds the number available in the system color table, MATLAB specifies all fixed colors first, then linearly subsamples the colormap to fill the remaining slots.

For example, if the original colormap contains 128 colors and there are only 64 slots available, then MATLAB adds every other color to the color table. MATLAB maps each color in the original colormap to the color in the subsampled colormap that most closely matches the original color.

Specifying the Minimum Colormap Size – MinColormap

The `Figure MinColormap` property specifies the minimum number of slots in the system color table that MATLAB uses for the Figure colormap. This enables you to use colormaps of any size up to the value of `MinColormap` and ensure MATLAB does not subsample the colors.



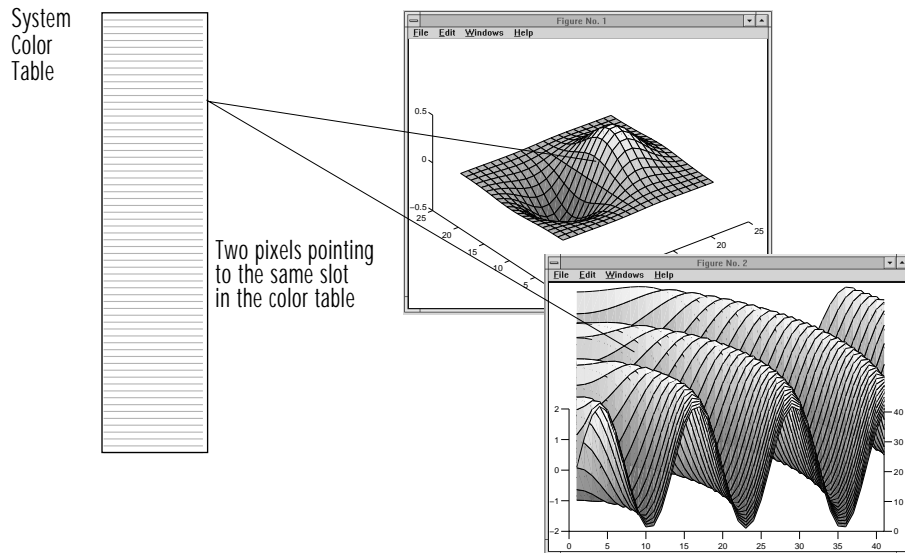
If you specify a value that is greater than the number of available slots, MATLAB takes over slots used to define system colors (on computers that allow overwriting of these colors). When this happens, nonactive windows can display with incorrect colors because MATLAB changed the color of the slot assigned to their pixels.

MATLAB does not take over color slots allocated to fixed colors. Therefore, limiting the number of fixed colors maximizes the number of colors allocated to the colormap. You can limit the number of fixed colors by specifying all noncolormap object colors (e.g., Text, Line, and Figure colors) as the same color, and setting the `Axes ColorOrder` property to just one color (the default is seven colors).

Nonactive Figures and Shared Colors

Overview. Set `ShareColors` to on to conserve resources and to off to allow rapid colormap change.

Since nonactive Figures are still visible, it is generally desirable for them to display correctly colored. However, if a number of Figures with different colormaps exist simultaneously, or have large colormaps, the computer's color resources may not be able to display all Figures correctly colored. When `ShareColors` is on, the Figure does not redefine a color in the system color table if that color already exists.



While sharing colors is a more efficient use of resources, it prevents MATLAB from rapidly changing the colormap (for example, as the `spimmap` function does). This is because MATLAB cannot change the value of a color slot in the system color table if other pixels also point to that slot for their color definition. It must find another slot for the new color. Changing color slot pixel assignments requires rerendering (i.e., recomputing color values and reassigning pixels to these colors) of the Figure whose colormap you are altering.

If you want to change a Figure's colormap rapidly, you should disable color sharing:

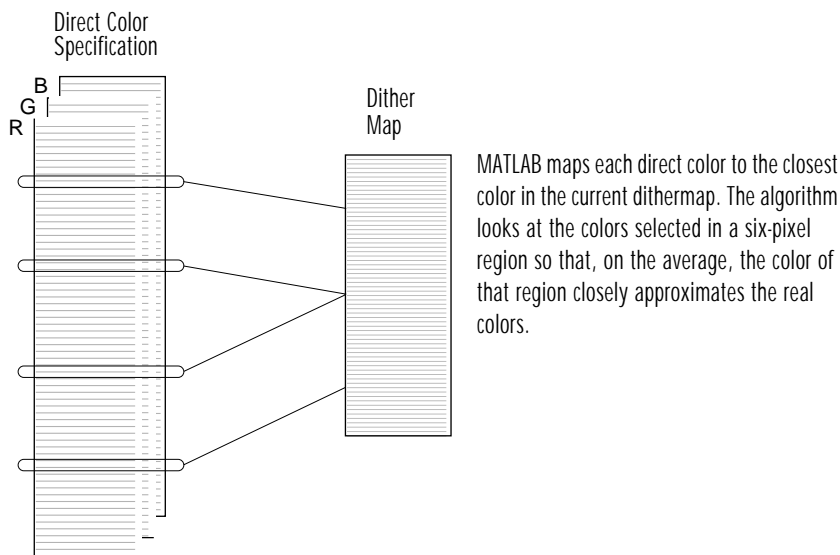
```
set(fig_handle, 'ShareColors', 'off')
```

Note that the new colormap must be the same size as the original one to avoid rerendering the Figure. Look at the `spnmap` M-file for an example of this technique.

Dithering Truecolor on Indexed Color Systems

Overview. Set `DithermapMode` to `manual` to use the current `Dithermap` or `auto` to force MATLAB to create a new `Dithermap` based on the colors displayed in the Figure.

MATLAB enables you to take advantage of truecolor systems (24-bit displays) by specifying `CData` as RGB triples, instead of values that index into the Figure colormap. Index color systems interpret truecolor specifications by mapping each color to the closest color in the dithermap, which is assigned to the `Dithermap` property. MATLAB uses the Floyd-Steinberg algorithm to perform the mapping.



The dithermap is a colormap that replaces the Figure colormap (which is not used in this case). The default dithermap contains a sampling of colors from the entire spectrum. This produces reasonably good quality with any object coloring. However, if the Figure contains objects of primarily one color, a dithermap concentrated in the same color produces better color resolution.

Auto Dither Mode

When you set `DithermapMode` to `auto`, MATLAB automatically creates a dithermap based on the colors in the Figure. MATLAB produces an appropriate dithermap using the minimum variance quantization algorithm; however, the process is time consuming. Also, MATLAB regenerates the dithermap each time it re-renders the Figure.

To avoid excessive rendering time, you should reset `DithermapMode` to `manual` after MATLAB generates the dithermap. MATLAB then uses this dithermap without regenerating it until you once again set `DithermapMode` to `auto`. You do not need to regenerate the dithermap unless you change the colors used in the Figure.

You can save a dithermap by assigning the `Dithermap` property to a variable and saving it as a MAT-file:

```
set(gcf, 'DithermapMode', 'auto')
```

MATLAB creates a dithermap, which you can then save:

```
dmap = get(gcf, 'Dithermap');  
save DitherMaps dmap
```

Dithermap Size

To obtain the highest color resolution, the default dithermap is as large as the system allows. This is usually less than 256 colors because a certain number of slots are reserved for system colors. Also, MATLAB fixed colors are not overwritten by the dithermap.

Effects of Dithering

Dithering reduces the resolution of the displayed graphics because the colors are mapped in groups of six pixels. For example, suppose the color of one pixel is defined as orange, but the dithermap does not have this color. MATLAB selects combinations of colors from the dithermap that, taken together as a six-pixel group, approximate the color orange.

Rendering Options

Two Figure properties affect the rendering speed of graphics. The `BackingStore` property allows faster redrawing when obscured Figure windows are exposed and the `Renderer` property provides faster rendering of graphics objects.

Backing Store

Overview. Enable `BackingStore` to produce fast redraws of previously obscured windows. Disable `BackingStore` to use less system memory.

The term “backing store” refers to an off-screen pixel buffer used to store a copy of the Figure window’s contents. When you move or delete windows on your display, previously obscured windows can become exposed (even partially), requiring the computer system to redraw these windows. With backing store enabled, MATLAB simply copies an exposed Figure window’s contents from the buffer to the screen.

The `BackingStore` property is on by default as this provides the most desirable behavior. However, the off-screen pixel buffers required for each Figure window do consume system memory. If memory is limited on your system, set `BackingStore` to `off` to release the memory used by these buffers.

Z-Buffer

Z-buffering is the process of determining how to render each pixel by drawing only the front-most object, as opposed to drawing all objects back to front, redrawing objects that obscure those behind. The pixel data is buffered and then blitted to the screen all at once.

Z-buffering is generally faster for more complex graphics, but may be slower for very simple graphics. You can set the `Renderer` property to whatever produces the fastest drawing (either `zbuffer` or `painters`), or let MATLAB decide which method to use by setting the `RenderMode` property to `auto` (the default).

Printing and Z-Buffer

You can select the resolution of the PostScript file produced by the `print` command using the `-r` option. By default, MATLAB prints Z-buffered Figures at a











medium resolution of 150 dpi (the default with `Renderer` set to `painters` is 864 dpi).

The size of the file generated from a Z-buffer Figure is not dependent on its contents, just the size of the Figure. To decrease the file size, make the `PaperPosition` property smaller before printing (or set `PaperPositionMode` to `auto` and resize the Figure window). See the *Printing* chapter for more information.

Figure Pointers

MATLAB indicates the position of the pointer within the Figure window using a graphical symbol. You can select a pointer from 15 predefined symbols (see table) or you can define your own symbol. By convention, each of the predefined symbols has a purpose associated with it (although MATLAB enforces no rules for the use of any symbols).

You specify the pointer symbol by setting the value of the `Figure Pointer` property. This table shows the predefined symbols, the associated specifier, and describes the typical use:

Purpose	Specifier	Typical Symbol
Locate a point on a graphics object	<code>crosshair</code>	
Select a point anywhere in the Figure	<code>arrow</code>	
Indicate the system is busy	<code>watch</code>	
Resize an object from the top-left corner	<code>topl</code>	
Resize an object from the top-right corner	<code>topr</code>	
Resize an object from the bottom-left corner	<code>botl</code>	
Resize an object from the bottom-right corner	<code>botr</code>	
View the actual hot spot	<code>circle</code>	
Locate a point	<code>cross</code>	
Popular symbol	<code>fleur</code>	

Purpose	Specifier	Typical Symbol
Resize an object from the left side	l e f t	⏪
Resize an object from the right side	r i g h t	⏩
Resize an object from the top	t o p	⏴
Resize an object from the bottom	b o t t o m	⏵
Align a point with other objects on the display	f u l l c r o s s	⛶
See the next section	c u s t o m	

Custom Pointers

When you set the `Poi n t e r` property to `custom`, MATLAB displays the pointer you define using the `Poi n t e r S h a p e C D a t a` and the `Poi n t e r S h a p e H o t S p o t` properties. Custom pointers are 16-by-16 pixels, where each pixel can be either black, white, or transparent.

Specify the pointer by creating a 16-by-16 matrix containing elements that are:

- 1s where you want the pixel black
- 2s where you want the pixel white
- Nans where you want the pixel transparent

Assign the matrix to the `Figure Poi n t e r S h a p e C D a t a` property. MATLAB displays the defined pointer whenever the pointer is in the Figure window.

The `Poi n t e r S h a p e H o t S p o t` property specifies the pixel that indicates the pointer location. MATLAB then stores this location in the `R o o t Poi n t e r L o c a t i o n` property. Set the `Poi n t e r S h a p e H o t S p o t` property to a two-element vector specifying the row and column indices in the `Poi n t e r S h a p e C D a t a` matrix that corresponds to the pixel specifying the location. The default value for this property is `[1 1]`, which corresponds to the upper-left corner of the pointer.

Examples — Custom Pointers

One way to create a custom pointer is to assign values to a 16-by-16 matrix by hand. For example

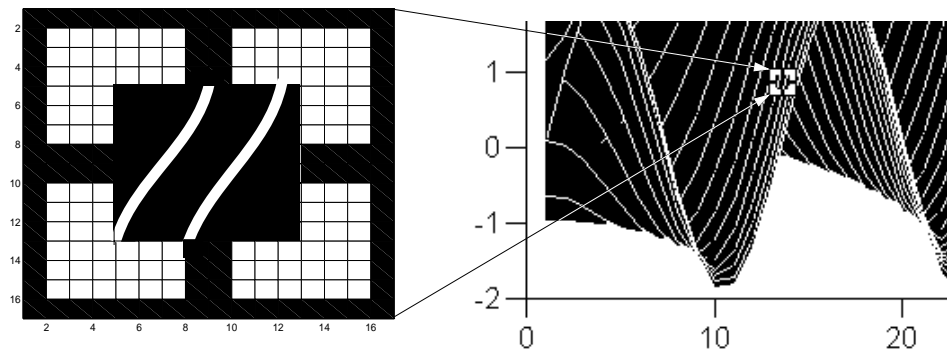
Initialize the matrix, setting all values to 2. Create a black border 1 pixel wide. Add alignment marks.

Create a transparent region in the center.

```
P = ones(16)+1;
P(1, :) = 1; P(16, :) = 1;
P(:, 1) = 1; P(:, 16) = 1;
P(1:4, 8:9) = 1; P(13:16, 8:9) = 1;
P(8:9, 1:4) = 1; P(8:9, 13:16) = 1;
P(5:12, 5:12) = NaN;
set(gcf, 'Pointer', 'custom', 'PointerShapeCData', P, ...
    'PointerShapeHotSpot', [9 9])
```

The last statement sets the `Pointer` property to `custom`, assigns the matrix to the `PointerShapeCData` property, and selects the “hot spot” as element (9,9).

MATLAB now uses the custom pointer within the Figure window:

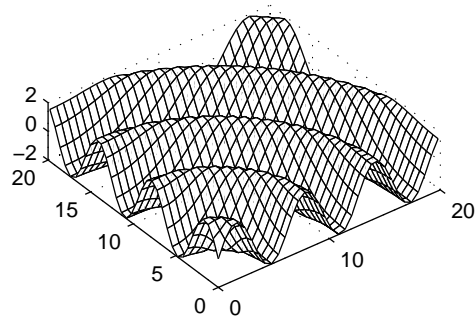


Creating Pointers from Functions. You can use a mathematical function to define the `PointerShapeCData` matrix. For example, evaluating the function,

$$2(\sin(\sqrt{x^2 + y^2})),$$

```
g = 0:2:20;
[X, Y] = meshgrid(g);
Z = 2*sin(sqrt(X.^2 + Y.^2));
mesh(Z);
```

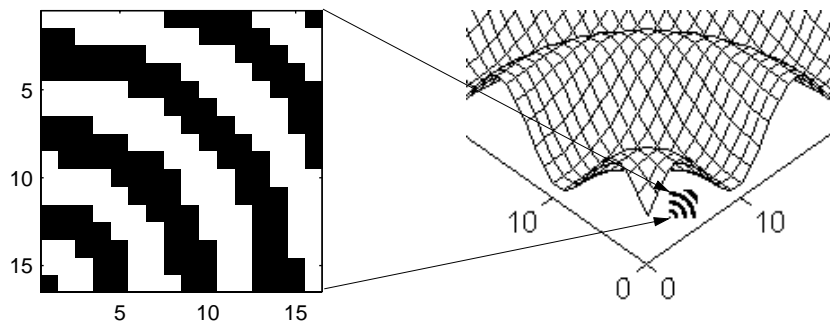
produces an interesting Surface:



Use the values of Z to create a pointer sampling fewer points so that Z is a 16-by-16 matrix:

```
g = linspace(0, 20, 16);
[X, Y] = meshgrid(g);
Z = 2*sin(sqrt(X.^2 + Y.^2));
set(gcf, 'Pointer', 'custom', ...
    'PointerShapeCData', flipud((Z>0) + 1))
```

The statement, `flipud((Z>0) + 1)` sets all values in Z that are greater than zero to two (in MATLAB, `true + 1 = 2`), less than zero to one (`false + 1 = 1`) and then flips the data around so that element (1,1) is the upper-left corner.



Printing Figures

This section discusses Figure properties that control the process of printing Figures. MATLAB produces PostScript and other file formats supported by printing and plotting devices. See the `print` command in the online MATLAB Function Reference for more information.

Positioning the Figure on the Printed Page

You can control the orientation and position of a Figure on a printed page using the Figure properties:

Property	Purpose
<code>PaperOrientation</code>	Horizontal or vertical paper orientation
<code>PaperPosition</code>	Location and size of Figure on printed page
<code>PaperPositionMode</code>	<code>PaperPosition</code> or actual size printed Figure
<code>PaperSize</code>	Size of <code>PaperType</code>
<code>PaperType</code>	Standard paper sizes
<code>PaperUnits</code>	Units used by <code>PaperPosition</code> and <code>PaperSize</code>

MATLAB defines the Figure `PaperPosition` property as a vector:

```
[left bottom width height]
```

`left` and `bottom` define the location on the paper of the lower-left corner of the Figure rectangle. `width` and `height` define the size of this rectangle. Note that the window border is not included in the printed Figure.

Factory Default PaperPosition

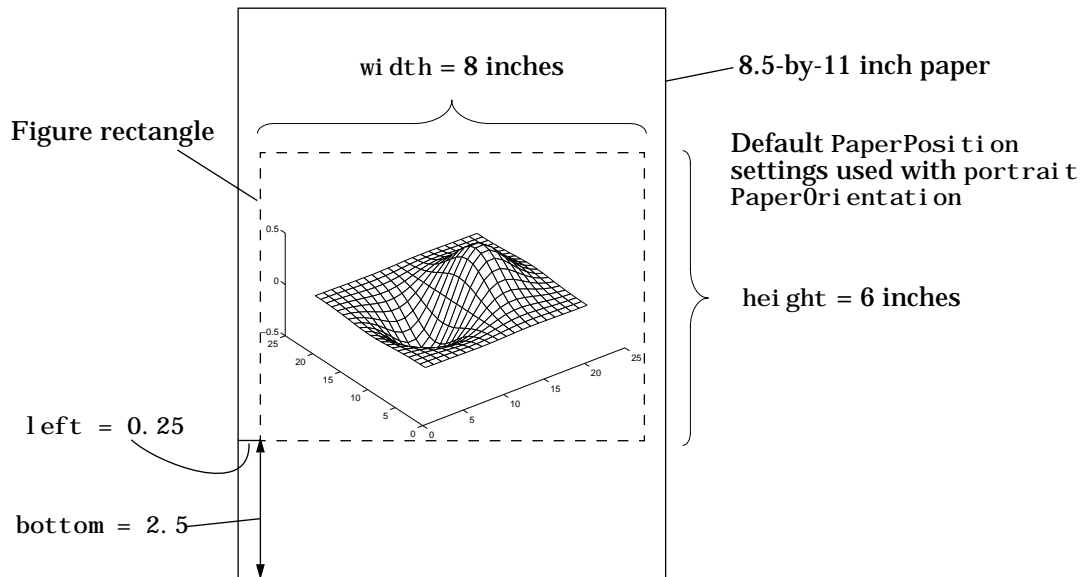
The factory default `PaperPosition`,

```
[0.25 2.5 8.0 6.0]
```

is designed to print the Figure on 8.5-by-11 inch paper in portrait orientation. It results in a printed Figure that is about as large as can fit on the paper (with a one-quarter inch border on the left and right sides) and is centered on the

paper. The width of eight inches and the height of six inches give an aspect ratio (ratio of width to height) that is the same as the default Figure aspect ratio on the screen.

MATLAB does not change the `PaperPosition` property automatically as you change the size of the Figure window. If you print a Figure having an aspect ratio different from that defined by the default `PaperPosition` (i.e., width divided by height equal to something other than 4/3), the printed Figure is distorted. See “Examples - Readjusting PaperPosition” for information on how to compensate for changes in Figure size.



While the default values for `PaperPosition` result in a centered Figure when using portrait orientation, the same values do not center the Figure in landscape orientation. However, the `orient` command repositions the Figure location on the printed page for either orientation. You should use this command rather than setting the `PaperOrientation` property directly to avoid having to recalculate the offsets from the paper edge. See the `orient` command description in the online MATLAB Function Reference.

Automatic PaperPosition – WYSIWYG Printing

Setting the `PaperPositionMode` to `auto` causes MATLAB to calculate the `PaperPosition` values required to print the Figure the same size as it is on the screen, centered on the page. The aspect ratio is maintained.

This mode prevents MATLAB from scaling (and potentially changing the aspect ratio of) the printed Figure, as can happen in `manual` mode using a fixed `PaperPosition`. In `auto` mode, MATLAB adjusts the `PaperPosition` as you change the Figure size and location.

Examples — Readjusting PaperPosition

Problem. You are working with Figure windows of varying sizes and shapes and you want to determine the values to use for the `PaperPosition` property to print each one:

- As large as possible
- With the same aspect ratio as the Figure on the screen
- With a minimum of a one-quarter inch border
- Centered on the page

Solution. To solve this problem in the general case, you need to know the values of the Figure's `Position` and `PaperSize` properties, being careful to use the same units for all dimensions. With this information you can decide how to orient the paper and/or scale the Figure size to fit properly.

Suppose a particular Figure is five inches wide and three and one-half inches high and you want to print it on the default size paper at your site. You need to calculate a new value for `PaperPosition` based on these sizes.

Obtain the size of the Figure and the paper, in inches:

```
set(gcf, 'Units', 'inches', 'PaperUnits', 'inches')
figpos = get(gcf, 'Position');
psize = get(gcf, 'PaperSize');
```

Since the Figure is wider than it is high, the width limits the maximum size that fits on the page. Furthermore, since the Figure is smaller than the paper,

you can set the PaperPosition width to the paper width minus a one-quarter inch border on both sides:

```
newpp(3) = min(psize) - .5;
```

Calculate the PaperPosition height in terms of the width, maintaining the correct aspect ratio:

```
newpp(4) = newpp(3) * figurepos(4) / figurepos(3);
```

To determine the offset from the bottom of the paper, subtract the new PaperPosition height (i.e., the printed height of the Figure) from the paper height, taking one-half this value to center the Figure on the page from top to bottom:

```
newpp(2) = (max(psize) - newpp(4)) / 2;
```

The offset from the left is simply the border width:

```
newpp(1) = .25;
```

You have now fully specified the PaperPosition for this particular Figure,

```
newpp =
```

```
0.2500    2.7000    8.0000    5.6000
```

and can set the property for printing:

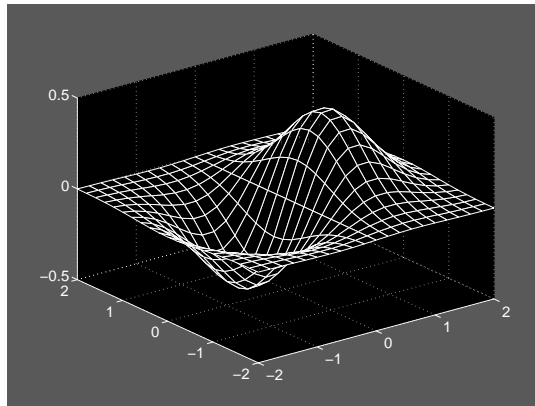
```
set(gcf, 'PaperPosition', newpp)
```

Reversing Figure Colors

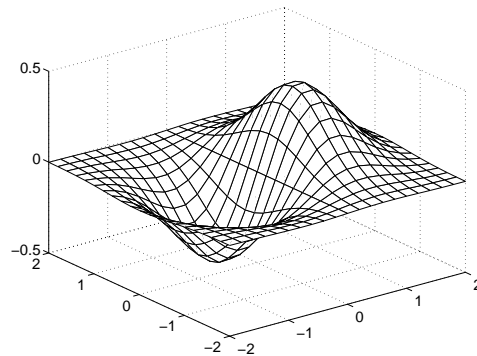
With the `colordf` command, it is possible to configure the default Figure background color to black and the axis lines and labels to white. This color scheme provides good contrast on the computer screen, but is less desirable when printed on white paper by a black and white device such as a laser printer.

The Figure `InvertHardCopy` property provides a simple way to convert the printed output to a white background. When `InvertHardCopy` is on (the default), MATLAB automatically inverts the color scheme to black-on-white output.

This mesh plot of a Surface has a white EdgeCol or. The white-on-black coloring produces a large black area on the printed page that results in poor discrimination between the mesh and background, particularly on low resolution printers.



With `InvertHardCopy` enabled, MATLAB automatically produces output more suited to printing in black and white. Here is the same mesh plot after reversing the color scheme:

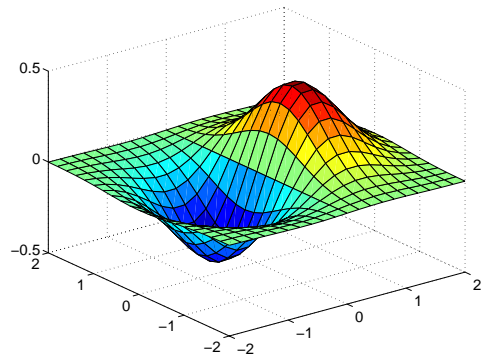


What Happens to Colors

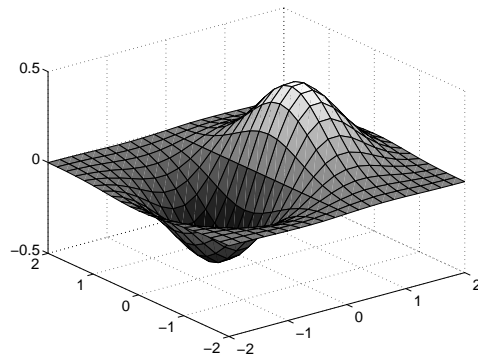
MATLAB changes the Figure background color to white (and also changes the Axes color unless it is set to none). Colors print in shades of gray on devices

capable of printing grays. However, it is not easy to anticipate how colors are mapped to grays.

For example, this Surface uses the `jet` colormap and is printed with `InvertHardCopy` enabled:



In cases where you want to print colored objects in grayscale, you should use a colormap that varies continuously from dark to light, such as `gray`, `copper`, or `bone`. The same Surface using the `gray` colormap prints in predictable shading:



Interactive Graphics

Figure objects contain a number of properties designed to facilitate user interaction with the Figure. These properties fall into two categories.

Properties related to callback routine execution:

- `BusyAction`
- `ButtonDownFcn`
- `DestroyFcn`
- `KeyPressFcn`
- `Interruptible`
- `ResizeFcn`
- `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn`

Properties that contain information about MATLAB's state:

- `CurrentAxes`
- `CurrentCharacter`
- `CurrentMenu`
- `CurrentObject`
- `CurrentPoint`
- `CurrentProperty`
- `SelectionType`

See the `figure` function in the online MATLAB Function Reference for a description of each property. The manual, *Building GUIs with MATLAB*, provides information on creating programs that incorporate interactive graphics.

Axes

Axes Properties	10-2
Labeling and Appearance Properties.	10-4
TeX Characters	10-6
Adding Text to Axes	10-8
Text Alignment	10-9
Using Variables in Text Strings	10-10
Example – Text Annotation	10-10
Example – Multiline Text.	10-12
Positioning Axes	10-13
The Position Vector	10-13
Units	10-14
Multiple Axes	10-15
Individual Axis Control.	10-18
Changing Axis Limits	10-18
Setting Tick Mark Locations	10-20
Changing Axis Direction	10-21
Automatic-Mode Properties	10-23
Multiaxis Axes	10-26
Example – Double Axis Graphs	10-26
Colors Controlled By Axes	10-29
Axes Colors	10-29
Axes Color Limits – The CLim Property.	10-31
Color of Lines Used for Plotting	10-37

Axes Properties

Axes are the parents of Image, Line, Patch, Surface, and Text graphics objects. These objects are the entities used to draw graphs of numerical data and pictures of real-world objects, such as airplanes or automobiles. Axes orient and scale their child objects to produce a particular effect, such as scaling a plot to accentuate certain information or rotating objects through various views.

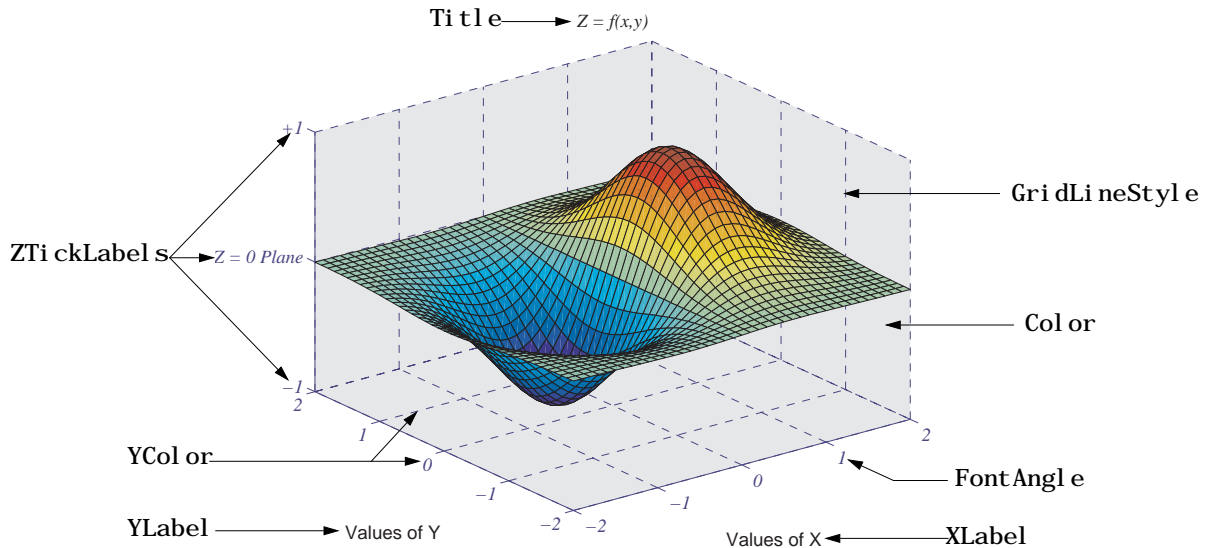
Axes properties control many aspects of how MATLAB displays graphical information. This chapter discusses some of the features that are implemented through Axes properties and provides examples of how to uses these features. The following table lists all Axes properties arranged by function See the axes description in the online MATLAB Function Reference for information on each property.

Category	Properties		
Style and appearance	Box	Cl i ppi ng	Gri dLi neStyl e
	Layer	Li neStyl eOrder	Li neWi dth
	Sel ect i onHi gh l i ght	Ti ckDi r	Ti ckDi rMode
	Ti ckLength	Vi si bl e	
General information	Chi l dren	Current Poi nt	Parent
	Posi ti on	Sele cted	Tag
	Type	Uni ts	UserData
Annotation	FontAngl e	FontName	FontSi ze
	FontUni ts	FontWei ght	Ti tle
	XLabel	YLabel	ZLabel
	XTi ckLabel XTi ckLabel Mode	YTi ckLabel YTi ckLabel Mode	ZTi ckLabel ZTi ckLabel Mode
Axis control	XDi r	XGri d	XLi m, XLi mMode
	YDi r	YGri d	YLi m, YLi mMode
	ZDi r	ZGri d	ZLi m, ZLi mMode

Category	Properties		
	XScale YScale ZScale	XTick, XTickMode YTick, YTickMode ZTick, ZTickMode	XAxisLocation YAxisLocation
Viewpoint	CameraPosition CameraPositionMode	CameraTarget CameraTargetMode	CameraUpVector CameraUpVectorMode
	CameraViewAngle CameraViewAngleMode		
Scaling and aspect ratio	DataAspectRatio DataAspectRatioMode	PlotBoxAspectRatio PlotBoxAspectRatioMode	Projection
Callback execution	BusyAction	ButtonDownFcn	CreateFcn
	DeleteFcn	Interruptible	
Rendering Method	DrawMode		
Targeting Axes	HandleVisibility	NextPlot	
Color	AmbientLightColor	CLim	CLimMode
	Color	ColorOrder	XColor
	YColor	ZColor	

Labeling and Appearance Properties

MATLAB provides a number of properties for labeling and controlling the appearance of Axes. For example, this Surface plot shows some of the labeling possibilities and indicates the controlling property.



To create this Axes, specify values for the indicated properties:

```
h = axes('Color', [.9 .9 .9], ...
        'GridLineStyle', '--', ...
        'ZTickLabels', '-1|Z = 0 Plane|+1', ...
        'FontName', 'times', ...
        'FontAngle', 'italic', ...
        'FontSize', 14, ...
        'XColor', [0 0 .7], ...
        'YColor', [0 0 .7], ...
        'ZColor', [0 0 .7]);
```

The individual axis labels are Text objects whose handles are normally hidden from the command line (their `HandleVisibility` property is set to `callback`). You can use the `xlabel`, `ylabel`, `zlabel`, and `title` functions to create axis labels. However, these functions affect only the current Axes. If you are

labeling axes other than the current axes by referencing the Axes handle, then you must obtain the Text object handle from the corresponding Axes property. For example,

```
get(h, 'XLabel')
```

returns the handle of the Text object used as the x -axis label. Obtaining the Text handle from the Axes is useful in M-files and MATLAB-based applications where you cannot be sure the intended target is the current Axes.

The following statements define the x - and y -axis labels and title for the axes on the previous page:

```
set(get(h, 'XLabel'), 'String', 'Values of X')
set(get(h, 'YLabel'), 'String', 'Values of Y')
set(get(h, 'Title'), 'String', '\fontname{times} \it Z = f(x, y)')
```

Since the labels are Text, you must specify a value for the String property, which is initially set to the empty string (i.e., there are no labels).

MATLAB overrides many of the other Text properties to control positioning and orientation of these labels. However, you can set the Color, FontAngle, FontName, FontSize, FontWeight, and String properties.

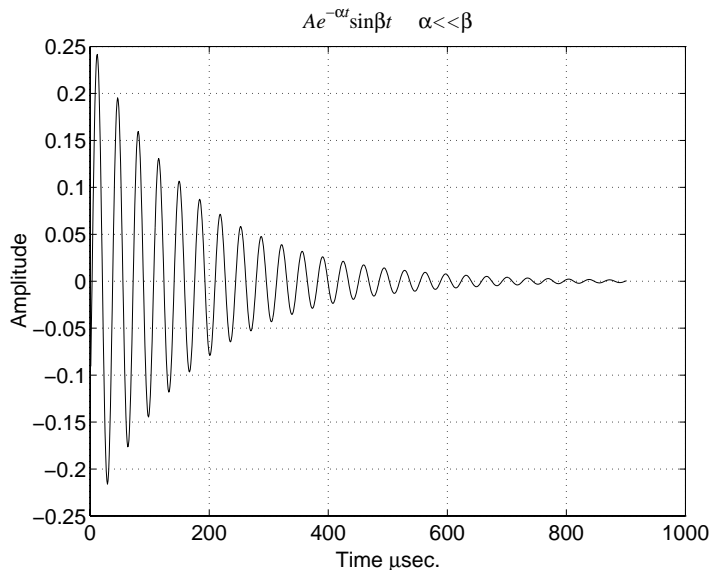
Note that both Axes objects and Text objects have font specification properties. The call to the axes function on the previous page set values for the FontName, FontAngle, and FontSize properties. If you want to use the same font for the labels and title, specify these same property values when defining their String property. For example, the x -axis label statement would be:

```
set(get(h, 'XLabel'), 'String', 'Values of X', ...
    'FontName', 'times', ...
    'FontAngle', 'italic', ...
    'FontSize', 14)
```

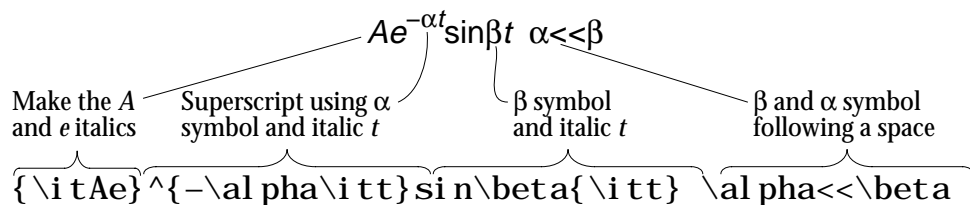
TeX Characters

Text objects support a subset of TeX characters that enable you to use symbols in the title and axis labels. For example:

```
set(get(h, 'Title'), 'String', ...
    '{\it Ae}^{\-\alpha\itt}\sin\beta{\itt}\ \alpha<<\beta')
set(get(h, 'Xlabel'), 'String', 'Time \musec.')
set(get(h, 'Ylabel'), 'String', 'Amplitude'))
```



The backslash character “\” precedes all TeX character sequences. Looking at the string defining the title illustrates how to use these characters:



The `TextInterpreter` property controls the interpretation of TeX characters. If you set this property to `off`, MATLAB interprets the special characters literally.

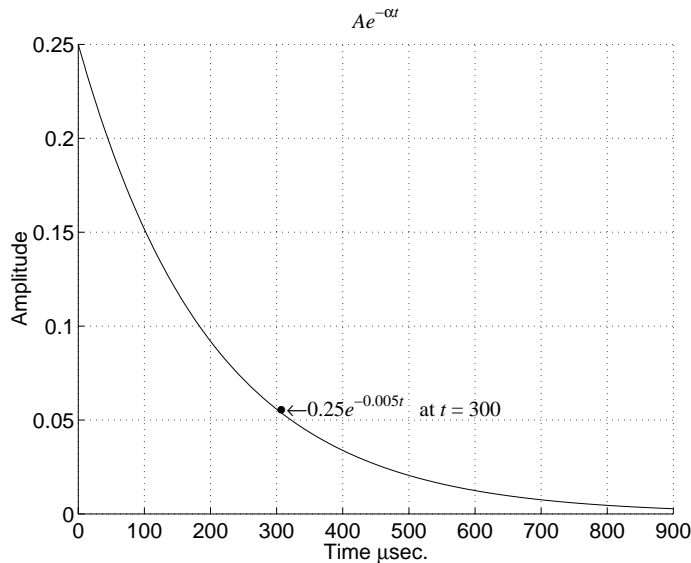
You can also use the `title`, `xlabel`, `ylabel`, and `zlabel` functions to add the labels. In most cases, these functions are easier to use, but only affect the current Axes. Obtaining the Text handle from the Axes is useful in M-files and MATLAB-based applications where you cannot be sure the intended target is the current Axes.

See the `text` function in the online MATLAB Function Reference for a list of available TeX characters.

Adding Text to Axes

You can use Text objects to annotate Axes at arbitrary locations. MATLAB locates Text in the data units of the Axes. For example, suppose you plot the function $y = Ae^{-\alpha t}$ with $A = 0.25$, $\alpha = 0.005$, and $t = 0$ to 900:

```
t = 0:900;
plot(t, 0.25*exp(-0.005*t))
```



To annotate the point where the value of $t = 300$, calculate the text coordinates using the function you are plotting:

```
text(300, .25*exp(-0.005*300), ...
'\bullet\leftarrow\fontname{times}0.25{\it e}^{-0.005{\it t}} at {\it t} = 300');
```

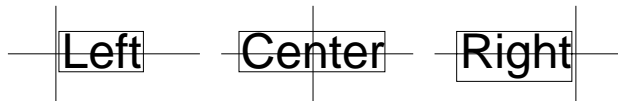
This statement defines the Text Position property as $x = 300$,

$y = 0.25e^{-0.005 \times 300}$. The default Text alignment places this point to the left of the string and centered vertically with the rectangle defined by the Text Extent property. The “Text Alignment” section describes other alignment options.

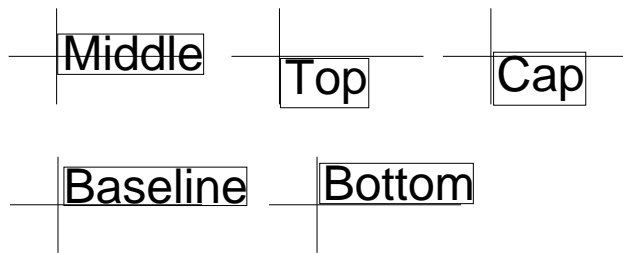
Text Alignment

The `Horizontal Alignment` and the `Vertical Alignment` properties control the placement of the Text characters with respect to the specified x -, y -, and z -coordinates. The following diagram illustrates the options for each property and the corresponding placement of the text:

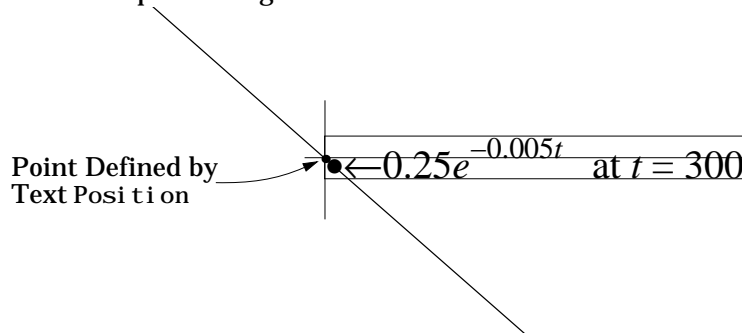
Text `Horizontal Alignment` property viewed with the `Vertical Alignment` property set to `middle` (the default).



Text `Vertical Alignment` property viewed with the `Horizontal Alignment` property set to `left` (the default).



The default alignment is `Horizontal Alignment` = `left`, and `Vertical Alignment` = `middle`. MATLAB does not place the Text String exactly on the specified Position. For example, the previous section showed a plot with a point annotated with Text. Zooming in on the plot allows you to see the actual positioning of the Text:



The small dot is the point specified by the `Text Position` property. The larger dot is the bullet defined as the first character in the `Text String` property.

Using Variables in Text Strings

Any string variable is a valid specification for the `Text String` property. For example, each row of the matrix `Personal Data` contains specific information about a person (note that all but the longest row is padded with a space so that each has the same number of columns).

```
Personal Data = [' Jack Straw   '; ' 489 Main St. '; ' Wichita KN '];
```

To display the data, index into the desired row:

```
text(x1, y1, [' Name: ', Personal Data(1, :)])  
text(x2, y2, [' Address: ', Personal Data(2, :)])  
text(x3, y3, [' City and State: ', Personal Data(3, :)])
```

You can specify numeric variables in Text strings using the `num2str` (number to string) function. For example, if you type on the command line:

```
x = 21;  
[' Today is the ', num2str(x), 'st day. ']
```

MATLAB concatenates the three separate strings into one:

```
Today is the 21st day.
```

Since the result is a valid string, you can specify it for a `Text` object's `String` property:

```
text(xcoord, ycoord, [' Today is the ', num2str(x), 'st day. '])
```

Example – Text Annotation

Suppose you want to label the minimum and maximum values in a mesh plot with text that is anchored to the points and indicates what the values are. You can use the `Surface` object's data to determine the `Text Position` and the data values. The first step is to get the *x*-, *y*-, and *z*-data and compute the minimum and maximum.

```

x = get(mesh_handle, 'XData');
y = get(mesh_handle, 'YData');
z = get(mesh_handle, 'ZData');
minz = min(min(z));
maxz = max(max(z));

```

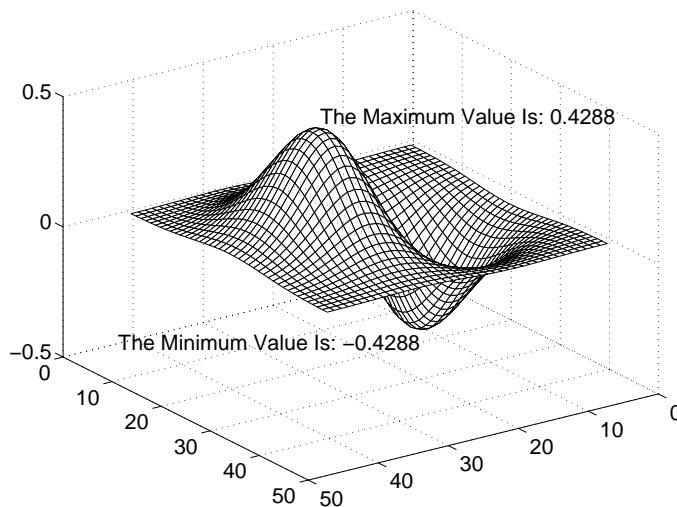
Next, find the indices of the minimum and maximum values to determine the coordinates needed to position the text at the points. Create the string by concatenating the values with a description of what the values mean:

```

[i,j] = find(minz == z);
text(x(j), y(i), z(i,j), ['The Minimum Value Is: ', num2str(minz)], ...
    'VerticalAlignment', 'top', ...
    'HorizontalAlignment', 'right')

[i,j] = find(maxz == z);
text(x(j), y(i), z(i,j), ['The Maximum Value Is: ', num2str(maxz)], ...
    'VerticalAlignment', 'bottom')

```



The Text alignment properties position the string correctly with respect to the mesh plot. The text function places the point specified by the coordinates above and to the right for the minimum value and below and to the left for the maximum value.

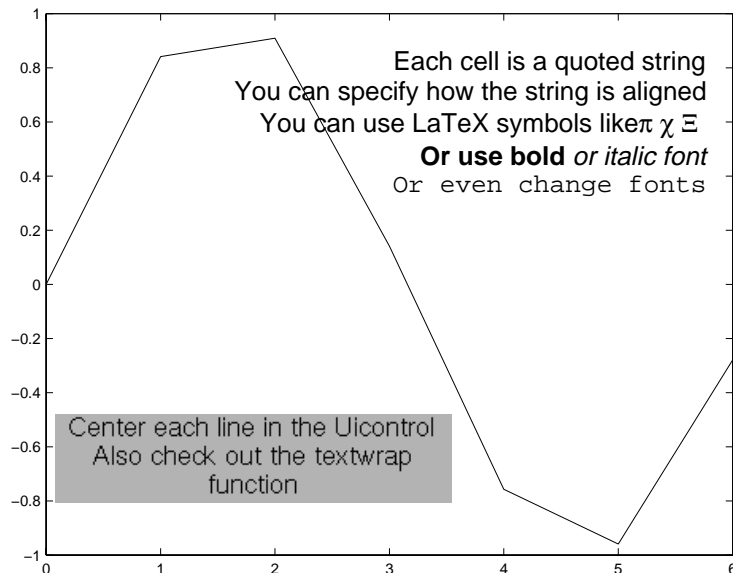
The text always remains in the plane of the computer screen, regardless of the view.

Example – Multiline Text

MATLAB supports multiline text strings using cell arrays. Simply define a string variable as a cell array with one line per cell. This example defines two cell arrays, one used for a Uicontrol and the other as a Text object.

```
str1(1) = {'Center each line in the Uicontrol'};
str1(2) = {'Also check out the textwrap function'};

str2(1) = {'Each cell is a quoted string'};
str2(2) = {'You can specify how the string is aligned'};
str2(3) = {'You can use LaTeX symbols like \pi \chi \Xi'};
str2(4) = {'\bf0r use bold \rm\it0r italic font\rm'};
str2(5) = {'\fontname{courier}0r even change fonts'};
plot(0:6, sin(0:6))
uicontrol('Style','text','Position',[80 80 250 65],...
         'String',str1);
text(5.75, sin(2.5), str2, 'Horizontal Alignment', 'right')
```



Positioning Axes

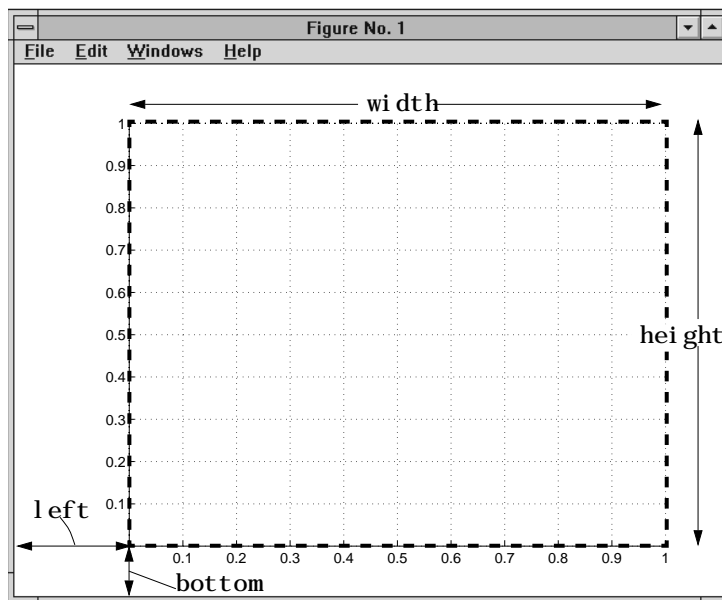
The `Axes Position` property controls the size and location of an Axes within a Figure. The default Axes has the same aspect ratio (ratio of width to height) as the default Figure and fills most of the Figure, leaving a border around the edges. However, you can define the Axes position as any rectangle and place it wherever you want within a Figure.

The Position Vector

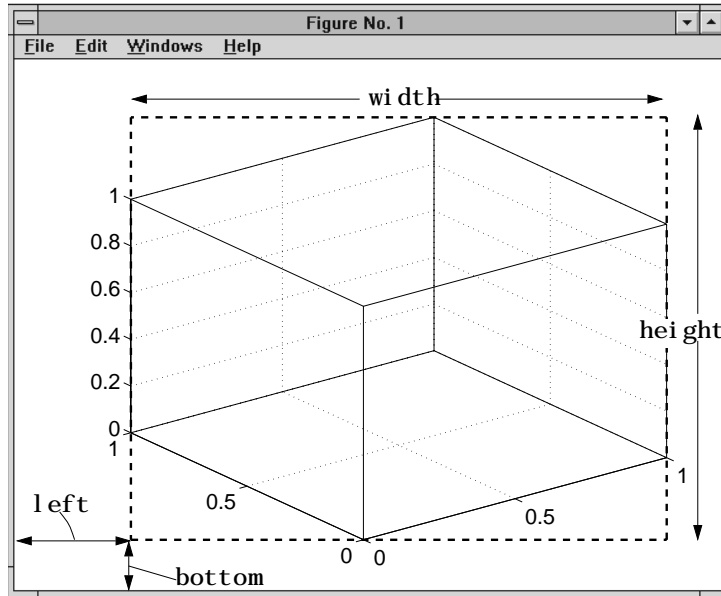
MATLAB defines the `Axes Position` property as a vector:

`[left bottom width height]`

`left` and `bottom` define a point in the Figure that locates the lower-left corner of the Axes rectangle. `width` and `height` specify the dimensions the Axes rectangle. Viewing the Axes in 2-D (azimuth = 0° , elevation = 90°) orients the x -axis horizontally and the y -axis vertically. From this angle, the plot box (the area used for plotting, exclusive of the axis labels) coincides with the Axes rectangle:



The default 3-D view is azimuth = -37.5° , elevation = 30° :



By default, MATLAB draws the plot box to fill the Axes rectangle, regardless of its shape. However, Axes properties enable control over the shape and scaling of the plot box.

Units

The Axes `Units` property determines the units of measurement for the `Position` property. Possible values for this property are:

```
set(gca, 'Units')
[ inches | centimeters | {normalized} | points | pixels ]
```

with `normalized` being the default. Normalized units map the lower-left corner of the Figure to the point (0,0) and the upper-right corner to (1.0,1.0), regardless of the size of the Figure. Normalized units cause Axes to resize automatically whenever you resize the Figure. All other units are absolute measurements that remained fixed as you resize the Figure.

Multiple Axes

The `subplot` function (described in the online MATLAB Function Reference) creates multiple Axes in one Figure by computing values for `Position` that produce the specified number of Axes. See the *3-D Graphs* chapter for more information on using `subplot`.

The `subplot` function is useful for laying out a number of graphs equally spaced in the Figure. However, overlapping Axes can create some other useful effects.

Placing Text Outside the Axes

MATLAB always displays Text objects within an Axes. If you want to create a graph and provide a description of the information alongside the graph, you must create another Axes to position the text. If you create an Axes that is the same size as the Figure and then create a smaller Axes to draw the graph, you can then display text anywhere independently of the graph.

For example, define two Axes:

```
h = axes('Position', [0 0 1 1], 'Visible', 'off');
axes('Position', [.25 .1 .7 .8])
```

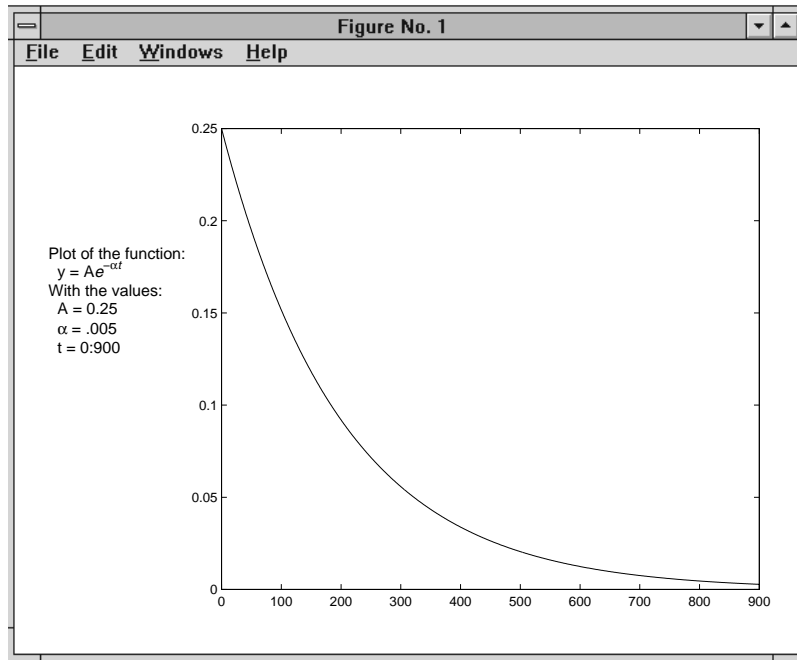
Since the Axes units are normalized to the Figure, specifying the `Position` as `[0 0 1 1]` creates an Axes that encompasses the entire window.

Now plot some data in the current Axes. The last Axes created is the current Axes so MATLAB directs graphics output there.

```
t = 0:900;
plot(t, 0.25*exp(-0.005*t))
```

Define the text and display it in the full-window axes.

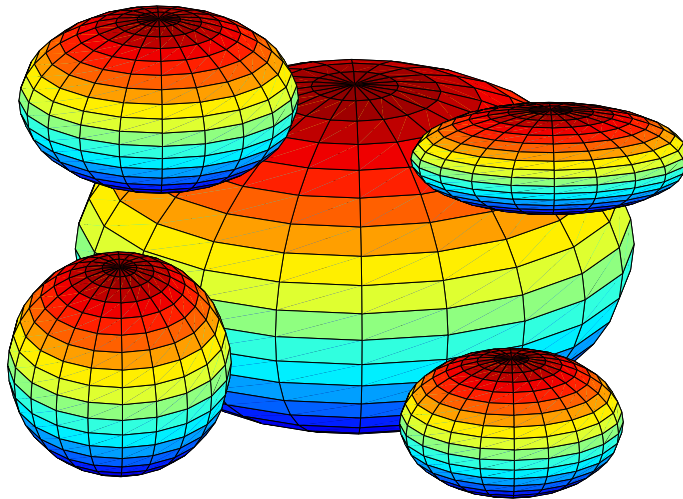
```
str(1) = {'Plot of the function:'};
str(2) = {' y = A{\it e}^{\alpha{\it t}}'};
str(3) = {'With the values:'};
str(3) = {' A = 0.25'};
str(4) = {' \alpha = .005'};
str(5) = {' t = 0:900'};
set(gcf, 'currentaxes', h)
text(.025, .6, str, 'FontSize', 12)
```



Multiple Axes for Different Scaling

You can create multiple Axes to display graphics objects with different scaling without changing the data that defines these objects (which would be required to display them in a single Axes).

```
h(1) = axes('Position', [0 0 1 1]);
sphere
h(2) = axes('Position', [0 0 .4 .6]);
sphere
h(3) = axes('Position', [0 .5 .5 .5]);
sphere
h(4) = axes('Position', [.5 0 .4 .4]);
sphere
h(5) = axes('Position', [.5 .5 .5 .3]);
sphere
set(h, 'Visible', 'off')
set(gcf, 'Renderer', 'painters')
```



Each sphere is defined by the same data. However, since the parent Axes occupy regions of different size and location, the spheres appear to be different sizes and shapes.

Individual Axis Control

MATLAB automatically determines axis limits, tick mark placement, and tick mark labels whenever you create a graph. However, you can specify these values manually by setting the appropriate property.

When you specify a value for a property controlled by a mode (e.g., the `XLim` property has an associated `XLimMode` property), MATLAB sets the mode to manual enabling you to override automatic specification. Since the default values for these mode properties are automatic, calling high-level functions such as `plot` or `surf` resets these modes to `auto`.

The properties discussed in this section include:

Property	Purpose
<code>XLim</code> , <code>YLim</code> , <code>ZLim</code>	Sets the axis range.
<code>XLimMode</code> , <code>YLimMode</code> , <code>ZLimMode</code>	Specifies whether axis limits are determined automatically by MATLAB or specified manually by the user.
<code>XTick</code> , <code>YTick</code> , <code>ZTick</code>	Sets the location of the tick marks along the axis.
<code>XTickMode</code> , <code>YTickMode</code> , <code>ZTickMode</code>	Specifies whether tick mark locations are determined automatically by MATLAB or specified manually by the user.
<code>XTickLabel</code> , <code>YTickLabel</code> , <code>ZTickLabel</code>	Specifies the labels for the axis tick marks.
<code>XTickLabelMode</code> , <code>YTickLabelMode</code> , <code>ZTickLabelMode</code>	Specifies whether tick mark labels are determined automatically by MATLAB or specified manually by the user.
<code>XDir</code> , <code>YDir</code> , <code>ZDir</code>	Sets the direction of increasing axis values.

Changing Axis Limits

MATLAB determines the limits automatically for each axis based on the range of the data. You can override the selected limits by specifying the `XLim`, `YLim`,

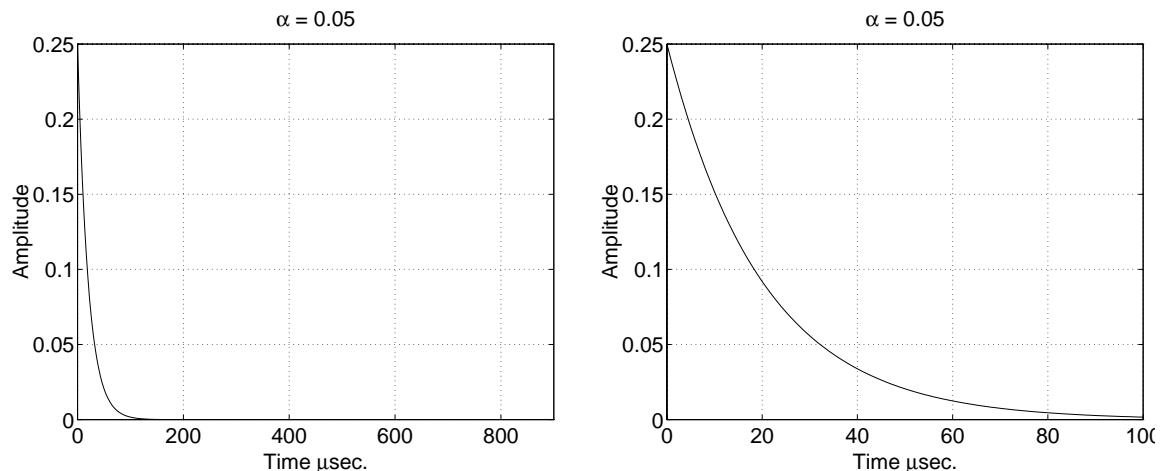
or `ZLim` property. For example, consider a plot of the function $Ae^{-\alpha t}$ evaluated with $A = 0.25$, $\alpha = 0.05$, and $t = 0$ to 900:

```
t = 0:900;
plot(t, 0.25*exp(-0.05*t))
```

The plot on the left shows the results. MATLAB selects axis limits that encompass the range of data in both x and y . However, since the plot contains little information beyond $t = 100$, changing the x -axis limits improves the usefulness of the plot:

```
set(axhandle, 'XLim', [0 100])
```

The plot on the right shows the result:



See the `axis` command for a simplified way to set limits on the current axis only.

Semiautomatic Limits

You can specify either the minimum or maximum value for an axis limit and allow the other limit to autorange. Do this by setting an explicit value for the manual limit and `Inf` for the automatic limit. For example, the statement:

```
set(axhandle, 'XLim', [0 Inf])
```

sets the `XLimMode` property to `auto` and allows MATLAB to determine the maximum value for `XLim`. Similarly, the statement:

```
set(axhandle, 'XLim', [-Inf 800])
```

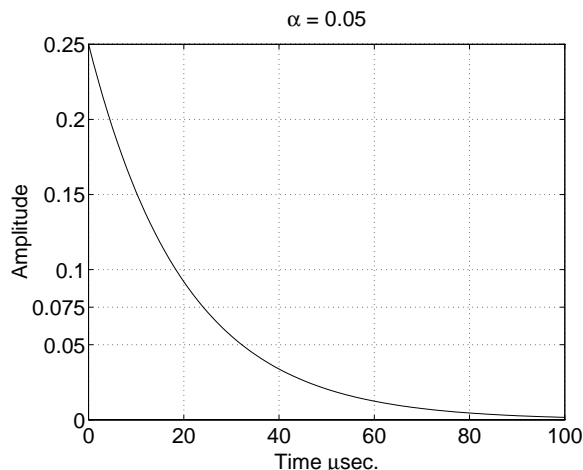
sets the `XLimMode` property to `auto` and allows MATLAB to determine the minimum value for `XLim`.

Setting Tick Mark Locations

MATLAB selects the tick mark location based on the data range to produce equally spaced ticks (for linear graphs). You can specify alternative locations for the tick marks by setting the `XTick`, `YTick`, and `ZTick` properties.

For example, if the value 0.075 is of interest for the amplitude of the function $Ae^{-\alpha t}$, specify tick marks to include that value.

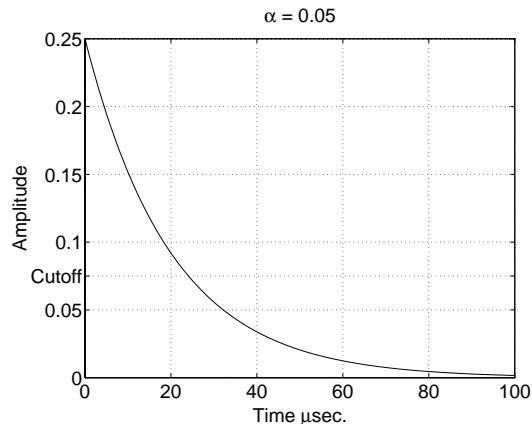
```
set(gca, 'YTick', [0 0.05 0.075 0.1 0.15 0.2 0.25])
```



You can change tick labeling from numbers to strings using the `XTickLabel`, `YTickLabel`, and `ZTickLabel` properties.

For example, to label the y -axis value of 0.075 with the string `Cutoff`, you can specify all y -axis labels as a string, separating each label with the “|” character:

```
set(gca, 'YTickLabel', '0|0.05|Cutoff|0.1|0.15|0.2|0.25')
```



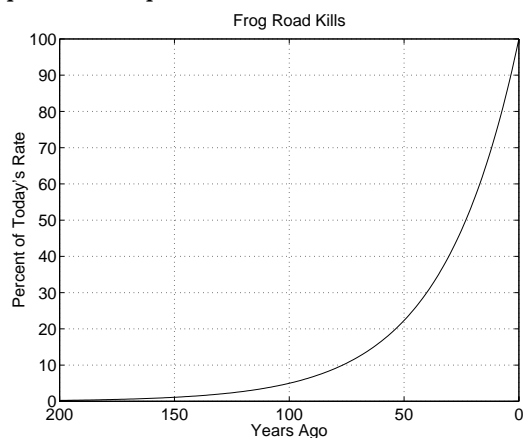
Changing Axis Direction

The `XDir`, `YDir`, and `ZDir` properties control the direction of increasing values on the respective axis. In the default 2-D view, the x -axis values increase from left to right and the y -axis values increase from bottom to top. The z -axis points out of the screen.

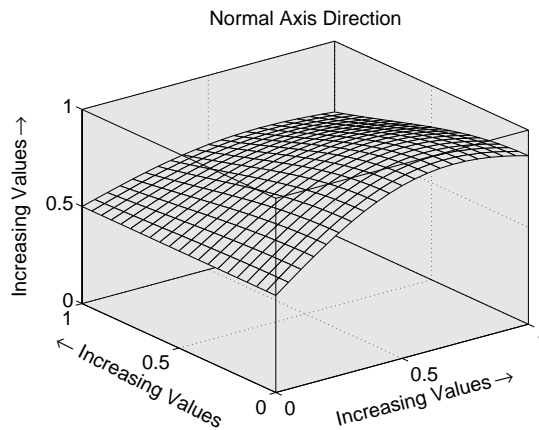
You can change the direction of increasing values by setting the associated property to reverse. For example, setting `XDir` to reverse,

```
set(gca, 'XDir', 'reverse')
```

produces a plot whose x -axis decreases from left to right.



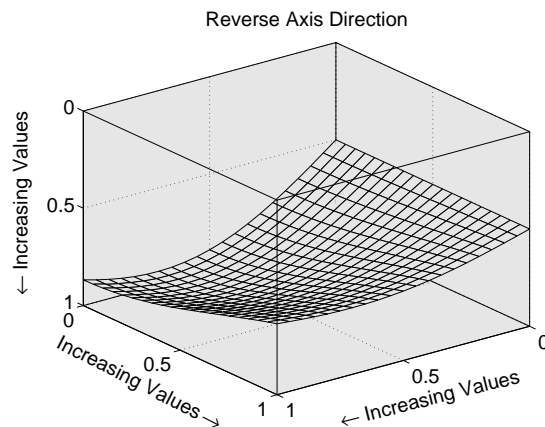
In the 3-D view, the y -axis increases from front to back and the z -axis increases from bottom to top:



Setting the x -, y -, and z -directions to reverse,

```
set(gca, 'XDir','rev','YDir','rev','ZDir','rev')
```

yields:



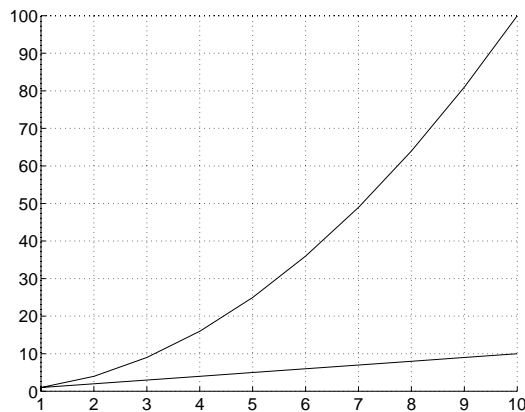
Automatic-Mode Properties

While object creation routines that create Axes children do not explicitly change Axes properties, some Axes properties are under automatic control when their associated mode property is set to `auto` (which is the default).

For example, if all property values are set to their defaults and you enter these statements:

```
line(1: 10, 1: 10)
line(1: 10, [1: 10]. ^2)
```

the second line statement causes the `YLim` property to change from `[0 10]` to `[0 100]`:



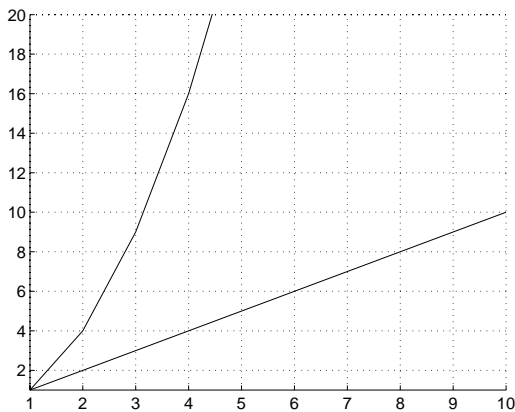
This is because `YLimMode` is `auto`, which always causes MATLAB to recompute the axis limits.

If you set the value controlled by an automatic-mode property, MATLAB sets the mode to `manual` and does not automatically recompute the value.

For example, in the statements:

```
line(1: 10, 1: 10)
set(gca, 'XLim', [1 10], 'YLim', [1 20])
line(1: 10, [1: 10]. ^2)
```

the `set` statement sets the x - and y -axis limits *and* changes the `XLimMode` and `YLimMode` properties to `manual`. The second `line` statement now draws a `Line` that is clipped to the axis limits [1 12] instead of causing the Axes to recompute its limits.



The automatic-mode properties include:

Mode Properties	What It Controls
CameraPositionMode	Positioning of the viewpoint
CameraTargetMode	Positioning of the camera target in the Axes
CameraUpVectorMode	The direction of “up” in 2-D and 3-D views
CameraViewAngleMode	The size of the projected scene and stretch-to-fit behavior
CLimMode	Mapping of data values to colors
DataAspectRatioMode	Relative scaling of data units along x , y , and z axes and stretch-to-fit behavior
PlotBoxAspectRatioMode	Relative scaling of plot box along x , y , and z axes and stretch-to-fit behavior

Mode Properties	What It Controls
Ti ckDi rMode	Direction of axis tick marks (in for 2-D, out for 3-D)
XLi mMode YLi mMode ZLi mMode	Limits of the respective <i>x</i> , <i>y</i> , and <i>z</i> axes
XTi ckMode YTi ckMode ZTi ckMode	Tick mark spacing along the respective <i>x</i> , <i>y</i> , and <i>z</i> axes
XTi ckLabel Mode YTi ckLabel Mode ZTi ckLabel Mode	Tick mark labels along the respective <i>x</i> , <i>y</i> , and <i>z</i> axes

The `axes` function description in the online MATLAB Function Reference provides more detail on Axes properties.

Multiaxis Axes

The `XAxisLocation` and `YAxisLocation` properties specify on which side of the graph to place the x - and y -axes. You can create graphs with two different x -axes and y -axes by superimposing two Axes objects and using `XAxisLocation` and `YAxisLocation` to position each axis on a different side of the graph. This technique is useful to plot different sets of data with different scaling in the same graph.

Example – Double Axis Graphs

This example creates a graph to display two separate sets of data using the bottom and left sides as the x - and y -axis for one, and the top and right sides as the x - and y -axis for the other.

Using low-level `line` and axes routines allows you to superimpose objects easily. Plot the first data, making the color of the Line and the corresponding x - and y -axis the same to more easily associate them:

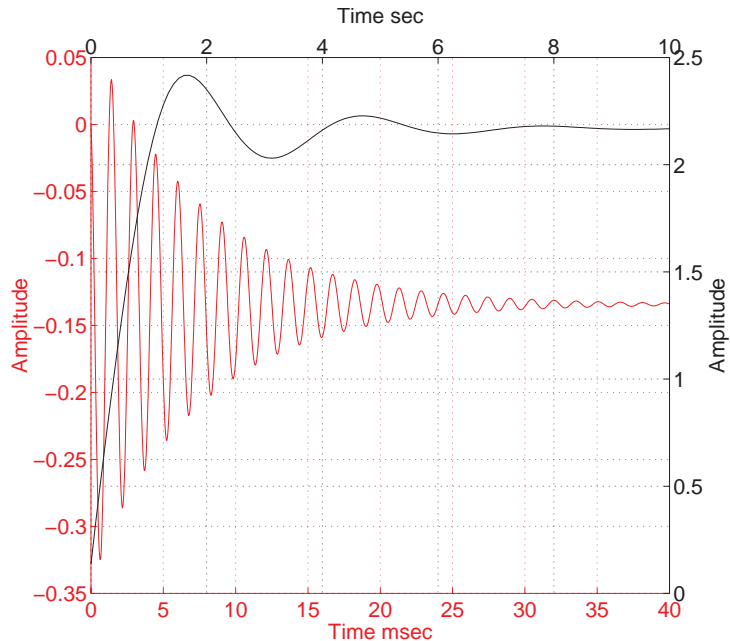
```
hl1 = line(x1, y1, 'Color', 'r');
ax1 = gca;
set(ax1, 'XColor', 'r', 'YColor', 'r')
```

Next, create another Axes at the same location as the first, placing the x -axis on top and the y -axis on the right. Set the `AxesColor` to none to allow the first Axes to be visible and color code the x - and y -axis to match the data:

```
ax2 = axes('Position', get(ax1, 'Position'), ...
           'XAxisLocation', 'top', ...
           'YAxisLocation', 'right', ...
           'Color', 'none', ...
           'XColor', 'k', 'YColor', 'k');
```

Draw the second set of data in the same color as the x - and y -axis:

```
hl2 = line(x2, y2, 'Color', 'k', 'Parent', ax2);
```



Coincident Grids

Since the two Axes are completely independent, MATLAB determines tick mark locations according to the data plotted in each. It is unlikely the gridlines will coincide. This produces a somewhat confusing looking graph, even though the two grids are drawn in different colors. However, if you manually specify tick mark locations, you can make the grids coincide.

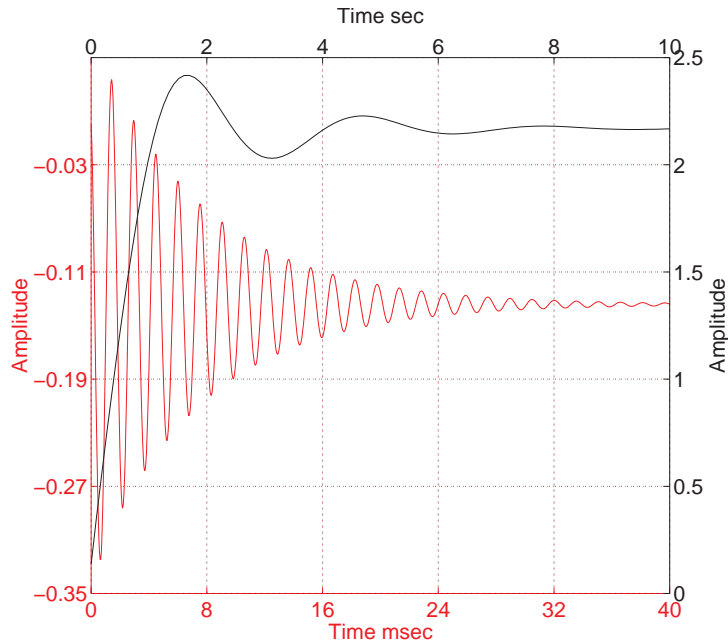
The key is to specify the same number of tick marks along corresponding axis lines (it is also necessary for both Axes to be the same size). The following graph of the same data uses six tick marks per axis, equally spaced within the original limits. To calculate the tick mark location, obtain the limits of each axis and calculate an increment:

```
xlimits = get(ax1, 'XLim');
ylimits = get(ax1, 'YLim');
xinc = (xlimits(2)-xlimits(1))/5;
yinc = (ylimits(2)-ylimits(1))/5;
```

Now set the tick mark locations:

```
set(ax1, 'XTi ck', [x l i m i t s(1) : x i n c: x l i m i t s(2)], ...  
    'YTi ck', [y l i m i t s(1) : y i n c: y l i m i t s(2)])
```

The resulting graph is visually simpler, even though the y -axis on the left has rather odd tick mark values:



Colors Controlled By Axes

Axes properties specify the color of the axis lines, tick marks, labels, and the background. Properties also control the color the Lines drawn by plotting routines and how Image, Patch, and Surface objects obtain colors from the Figure colormap.

Axes properties discussed in this section include:

Property	Characteristic it Controls
Col or	Axes background color
XCol or, YCol or, ZCol or	Color of the axis lines, tick marks, gridlines and labels
Ti t l e, XLabel , YLabel , Zl abel	Title and axis label Text object handles.
CLi m	Controls mapping of graphic object CData to the Figure colormap
CLi mMode	Automatic or manual control of CLi m property
Col orOrder	Line color autocycle order
Li neStyl eOrder	Line styles autocycle order (not a color, but related to Col orOrder)

Axes Colors

The default Axes background color is set up by the `col ordef` command, which is called in your startup file. However, you can easily define your own color scheme.

See the *Printing* chapter for information on how MATLAB automatically changes the color scheme for printing hardcopy.

Changing the Color Scheme

Suppose you want an Axes to use a “black-on-white” color scheme. First, change the background to white and the axis lines, grid, tick marks, and tick mark labels to black:

```
set(gca, 'Color', 'w', ...
        'XColor', 'k', ...
        'YColor', 'k', ...
        'ZColor', 'k')
```

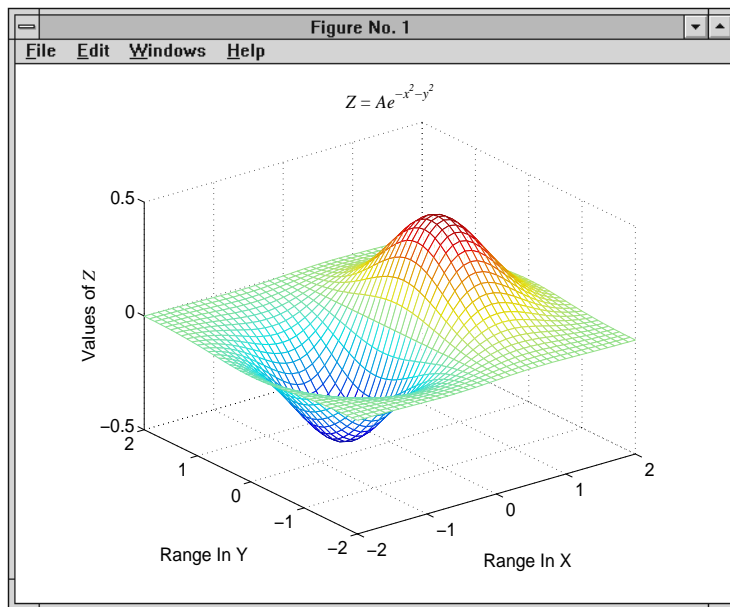
Next, change the color of the Text objects used for the title and axis labels:

```
set(get(gca, 'Title'), 'Color', 'k')
set(get(gca, 'XLabel'), 'Color', 'k')
set(get(gca, 'YLabel'), 'Color', 'k')
set(get(gca, 'ZLabel'), 'Color', 'k')
```

Changing the Figure background color to white completes the new color scheme:

```
set(gcf, 'Color', 'w')
```

When you are done, a Figure containing a mesh plot looks like this:



You can define default values for the appropriate properties and put these definitions in your `startup.m` file. Titles and axis labels are Text objects, so you must set a default color for all Text objects, which is a good idea anyway since the default Text color of white is not visible on the white background. Lines created with the low-level `line` function (but not the plotting routines) also have a default color of white, so you should change the default Line color as well.

To set default values on the Root level, use:

```
set(0, 'DefaultFigureColor', 'w'
    'DefaultAxesColor', 'w', ...
    'DefaultAxesXColor', 'k', ...
    'DefaultAxesYColor', 'k', ...
    'DefaultAxesZColor', 'k', ...
    'DefaultTextColor', 'k', ...
    'DefaultLineColor', 'k')
```

MATLAB colors other Axes children (i.e., Image, Patch, and Surface objects) according to the values of their `CData` properties and the Figure colormap. The next section discusses how Axes properties affect this coloring.

Axes Color Limits – The CLim Property

Many of the 3-D graphics functions produce graphs that use color as another data dimension. For example, surface plots map surface height to color. The color limits control the limits of the color dimension in a way analogous to setting axis limits.

The Axes `CLim` property controls the mapping of Image, Patch, and Surface `CData` to the Figure colormap. `CLim` is a two-element vector [`cmi n` `cmax`] specifying the `CData` value to map to the first color in the colormap (`cmi n`) and the `CData` value to map the last color in the colormap (`cmax`). Data values in between are linearly transformed from the second to the next to last color, using the expression:

$$\text{colormap_index} = \text{fix}((\text{CData} - \text{cmi n}) / (\text{cmax} - \text{cmi n}) * \text{cm_length}) + 1$$

See the `caxis` reference page for more information on color limits.

`cm_length` is the length of the colormap. When `CLimMode` is `auto`, MATLAB sets `CLim` to the range of the `CData` of all graphics objects within the Axes. However, you can set `CLim` to span any range of values. This allows individual Axes

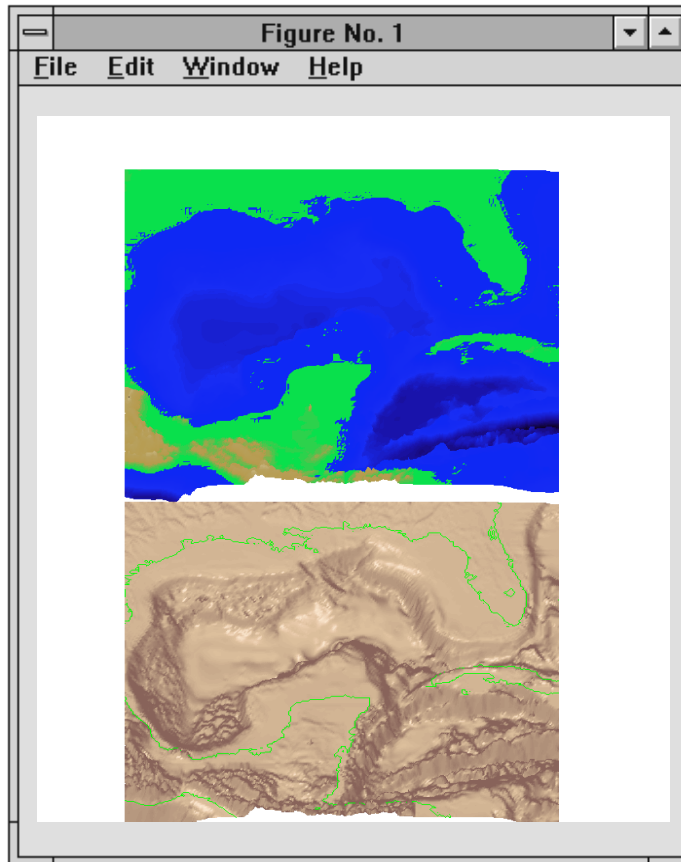
within a single Figure to use different portions of the Figure's colormap. You can create colormaps with different regions, each used by a different Axes.

Example – Simulating Multiple Colormaps In a Figure

Suppose you want to display two different Surfaces in the same Figure and color each Surface with a different colormap. You can produce the effect of two different colormaps by concatenating two colormaps together and then setting the `CLim` property of each Axes to map into a different portion of the colormap.

This example creates two Surfaces from the same topographic data. One uses the color scheme of a typical atlas – shades of blue for the ocean and greens for the land. The other Surface is illuminated with a light source to create the

illusion of a three-dimensional picture. Such illumination requires a colormap that changes monotonically from dark to light.



Calculating Color Limits

The key to this example is calculating values for CL_{im} that cause each Surface to use the section of the colormap containing the appropriate colors.

To calculate the new values for `CLim`, you need know:

- The total length of the colormap (`CmLength`).
- The beginning colormap slot to use for each Axes (`BeginSlot`).
- The ending colormap slot to use for each Axes (`EndSlot`).
- The minimum and maximum `CData` values of the graphic objects contained in the Axes. That is, the values of the Axes `CLim` property determined by MATLAB when `CLimMode` is `auto` (`CDmin` and `CDmax`).

First, define subplots regions, and plot the Surfaces:

```
ax1 = subplot(2, 1, 1);
view([0 80])
surf(topodata)
shading interp
ax2 = subplot(2, 1, 2), ;
view([0 80]);
surf1(topodata, [60 0])
shading interp
```

Concatenate two colormaps together and install the new colormap:

```
colormap([Lightingmap; Atlasmap]);
```

Obtain the data you need to calculate new values for `CLim`:

Colormap length	<code>CmLength = size(get(gcf, 'Colormap'), 1);</code>
Beginning and ending slots	<code>BeginSlot1 = 1;</code> <code>EndSlot1 = size(Lightingmap, 1);</code> <code>BeginSlot2 = EndSlot1+1;</code> <code>EndSlot2 = CmLength;</code>
<code>CLim</code> values for each axis	<code>CLim1 = get(ax1, 'CLim');</code> <code>CLim2 = get(ax2, 'CLim');</code>

Computing new values for `CLim` involves determining the portion of the colormap you want each Axes to use relative to the total colormap size and

scaling its `CLim` range accordingly. You can define a MATLAB function to do this:

	<code>function CLim = newclim(BeginSlot, EndSlot, CDmin, CDmax, CmLength)</code>
	<code>PBeginSlot = (BeginSlot - 1) / (CmLength - 1);</code>
Convert slot numbers and range percent of colormap.	<code>PEndSlot = (EndSlot - 1) / (CmLength - 1);</code>
	<code>PCmRange = PEndSlot - PBeginSlot;</code>
	<code>DataRange = CDmax - CDmin;</code>
Determine range and min and max of new <code>CLim</code> values.	<code>CLimRange = DataRange / PCmRange;</code>
	<code>NewCmin = CDmin - (PBeginSlot * CLimRange);</code>
	<code>NewCmax = CDmax + (1 - PEndSlot) * CLimRange;</code>
	<code>CLim = [NewCmin, NewCmax];</code>

The input arguments are identified in the bulleted list on the preceding page. The M-file first computes the percentage of the total colormap you want to use for a particular Axes (`PCmRange`) and then computes the `CLim` range required to use that portion of the colormap given the `CData` range in the Axes. Finally, it determine the minimum and maximum values required for the calculated `CLim` range and return these values. These values are the color limits for the given Axes.

Using the M-File

Use the `newclim` M-file to set the `CLim` values of each Axes. The statement,

```
set(ax1, 'CLim', newclim(65, 120, clim(1), clim(2)))
```

sets the `CLim` values for the first Axes so the Surface uses color slots 65 to 120. The lit Surface uses the lower 64 slots. You need to reset its `CLim` values as well:

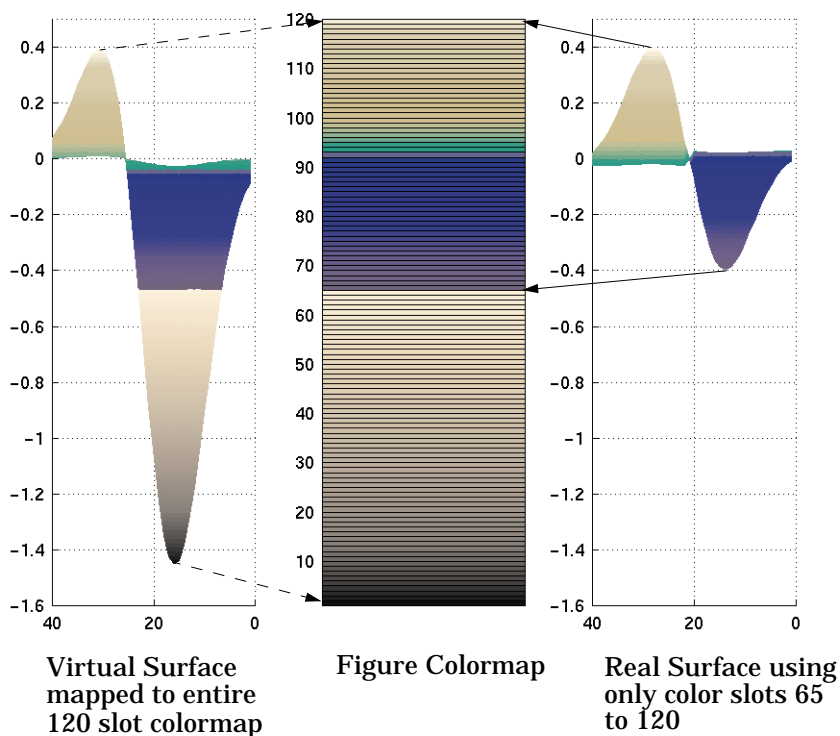
```
set(ax2, 'CLim', newclim(1, 64, clim(1), clim(2)))
```

How the M-File Works

MATLAB enables you to specify any values for the Axes `CLim` property, even if these values do not correspond to the `CData` of the graphics objects displayed in the Axes. MATLAB always maps the minimum `CLim` value to the first color in the colormap and the maximum `CLim` value to the last color in the colormap, whether or not there are really any `CData` values corresponding to these colors. Therefore, if you specify values for `CLim` that extend beyond the object's actual `CData` minimum and maximum, MATLAB colors the object with only a subset of the colormap.

The `newclim` M-file computes values for `CLim` that map the graphics object's actual `CData` values to the beginning and ending colormap slots you specify. It does this by defining a “virtual” graphics object having the computed `CLim` values. The following picture illustrates this concept. It shows a side view of two Surfaces to make it easier to visualize the mapping of color to Surface topography. The virtual Surface is on the left and the actual Surface on the right. In the center is the Figure's colormap.

The real Surface has `CLim` values of $[0.4 \ -0.4]$. To color this Surface with slots 65 to 120, `newclim` computed new `CLim` values of $[0.4 \ -1.4269]$. The virtual Surface on the left represents these values.



Color of Lines Used for Plotting

The `Axes ColorOrder` property determines the color of the individual Lines drawn by the `plot` and `plot3` functions. For multiline graphs, these functions cycle through the colors defined by `ColorOrder`, repeating the cycle when reaching the end of the list.

The `colorddef` function defines various color order schemes for different background colors. `colorddef` is typically called in the `matlabrc` file, which is executed during MATLAB's startup.

Defining Your Own ColorOrder

You can redefine `ColorOrder` to be any m -by-3 matrix of RGB values, where m is the number of colors. However, high-level functions like `plot` and `plot3` reset most Axes properties (including `ColorOrder`) to their defaults each time you call them. To use your own `ColorOrder` definition you must,

- Define a default `ColorOrder` on the Figure or Root level, or
- Change the `Axes NextPlot` property to add or replace children, or
- Use the informal form of the `line` function, which obeys the `ColorOrder` but does not clear the Axes or reset properties

Changing the Default ColorOrder. You can define a new `ColorOrder` that MATLAB uses within a particular Figure, for all Axes within any Figures created during the MATLAB session, or as a user-defined default that MATLAB always uses.

To change the `ColorOrder` for all plots in the current Figure, set a default in that Figure. For example, to set `ColorOrder` to the colors red, green, and blue, use the statement:

```
set(gcf, 'DefaultAxesColorOrder', [1 0 0; 0 1 0; 0 0 1])
```

To define a new `ColorOrder` that MATLAB uses for all plotting during your entire MATLAB session, set a default on the Root level so Axes created in any Figure use your defaults:

```
set(0, 'DefaultAxesColorOrder', [1 0 0; 0 1 0; 0 0 1])
```

To define a new `ColorOrder` that MATLAB always uses, place the previous statement in your startup. `m` file.

Setting the NextPlot Property. The Axes `NextPlot` property determines how high-level graphics functions draw into an existing Axes. You can use this property to prevent `plot` and `plot3` from resetting the `ColorOrder` property each time you call them, but still clear the Axes of any exiting plots.

By default, `NextPlot` is set to `replace`, which is equivalent to a `cla reset` command (i.e., delete all Axes children and reset all properties, except `Position`, to their defaults). If you set `NextPlot` to `replacechildren`,

```
set(gca, 'NextPlot', 'replacechildren')
```

MATLAB deletes the Axes children, but does not reset Axes properties. This is equivalent to a `cla` command without the `reset`.

After setting `NextPlot` to `replacechildren`, you can redefine the `ColorOrder` property and call `plot` and `plot3` without affecting the `ColorOrder`.

Setting `NextPlot` to `add` is the equivalent of issuing the `hold on` command. This setting prevents MATLAB from resetting the `ColorOrder` property, but it does not clear the Axes children with each call to a plotting function (See “Using the `line` Function”).

Using the line Function. The behavior of the `line` function depends on its calling syntax. When you use the informal form (which does not include any explicit property definitions):

```
line(x, y, z)
```

`line` obeys the `ColorOrder` property, but does not clear the Axes with each invocation or change the view to 3-D (as `plot3` does). However, `line` can be useful for creating your own plotting functions where you do not want the automatic behavior of `plot` or `plot3`, but you do want multiline graphs to use a particular `ColorOrder`.

Line styles Used for Plotting – `LineStyleOrder`

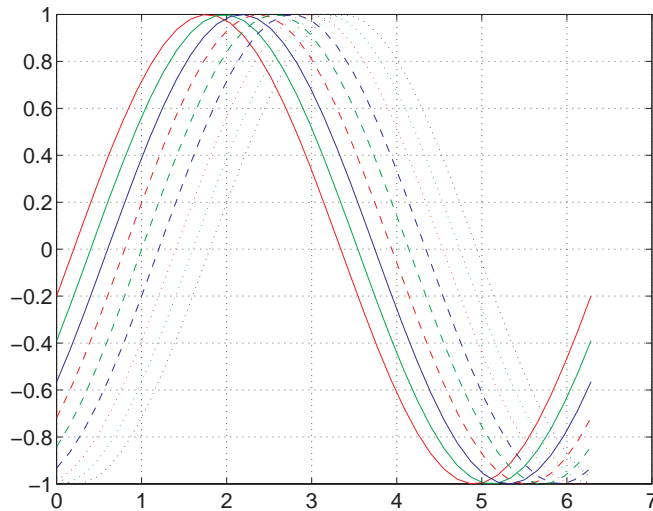
The Axes `LineStyleOrder` property is analogous to the `ColorOrder` property. It specifies the line styles to use for multiline plots created with the `plot` and `plot3` functions. MATLAB increments the line style only after using all of the colors in the `ColorOrder` property. It then uses all the colors again with the second line style, and so on.

For example, define a default ColorOrder of red, green, and blue and a default LineStyleOrder of solid, dashed, and dotted lines:

```
set(0, 'DefaultAxesColorOrder', [1 0 0; 0 1 0; 0 0 1], ...
     'DefaultAxesLineStyleOrder', '-|-|:')
```

Then plot some multiline data:

```
t = 0:pi/20:2*pi;
a = ones(length(t), 9);
for i = 1:9
    a(:, i) = sin(t-i/5)';
end
plot(t, a)
```



MATLAB cycles through all colors for each line style.

A

- Adobe Illustrator 88 7-20
- alignment of text 10-9
- ambient light 3-25
- AmbientLightColor property 3-22
 - illustration 3-25
- AmbientStrength property 3-22
 - illustration 3-25
- animation 4-45
 - erase modes for 4-47
 - movies 4-45
- annotating graphs 2-19, 10-8
 - example 10-10
- area 4-10
- area graphs 4-2, 4-10
- aspect ratio 3-47-3-59
 - for realistic objects 3-58
 - properties that affect 3-51
 - specifying 3-55
- Axes
 - adding text 10-8
 - annotation properties 10-2
 - aspect ratio 3-47, 3-51
 - 2-D 2-17
 - properties that affect 3-51
 - specifying 3-55
 - automatic modes 10-23
 - axis control 10-18
 - properties 10-2
 - axis direction 10-21
 - callback execution properties 10-3
 - camera properties 3-35
 - CLim property 10-31
 - color limits 10-31
 - color properties 10-3
 - ColorOrder property 10-37
 - colors 10-29
 - controlling the shape of 3-55
 - default aspect ratio 3-52
 - general information properties 10-2
 - graphics objects 10-2
 - individual axis control 10-18
 - labeling 2-19
 - labels
 - font properties 10-5
 - using TeX characters 10-6
 - limits 3-47
 - example 3-57
 - making grids coincident 10-27
 - multi-axis 10-26
 - multiple 2-3, 10-15
 - NextPlot property 8-34
 - obtaining handles 8-36
 - overlapping 10-15
 - printing 10-17
 - plot box 3-36
 - position rectangle 3-36
 - positioning 10-13-10-17
 - preparing to accept graphics 8-33
 - properties
 - for labeling 10-4
 - list 10-2
 - protecting from output 8-39
 - rendering method properties 10-3
 - scaling 3-47
 - and aspect ratio properties 10-3
 - independent 10-16
 - setting
 - limits 10-18
 - line styles used for plotting 10-38
 - setting limits 2-15
 - standard plotting behavior 8-37
 - stretch-to-fill 3-47

- style and appearance properties 10-2
- target
 - for graphics 2-5
 - properties controlling 10-3
- tick marks 2-16
 - locating 10-20
- units 10-14
- viewpoint properties 10-3

axes 10-2

axis 3-47

- auto 3-48
- equal 2-18, 3-48
- ij 3-48
- illustrated examples, 2-D 2-18
- illustrated examples, 3-D 3-49
- image 3-48, 5-11
- manual 3-48
- normal 3-48
- square 2-17, 3-48
- tight 2-18, 3-48
- vis3d 3-48
- xy 3-48

azimuth of viewpoint 3-32

- default 2-D 3-32
- default 3-D 3-32
- limitations 3-34

B

- BackFaceLighting property 3-23
 - illustration 3-27
- BackIngStore property 9-15
- bar 4-2
- bar graphs 4-2-4-10
 - 3-D 4-3

- grouped
 - 2-D 4-2
 - 3-D 4-4
- horizontal 4-6
- labeling 4-4, 4-7
- overlaid with plots 4-8
- stacked 4-5

bar3 4-3

bins, specifying for histogram 4-18

BMP 5-2

brighten 3-17

C

- camera position, moving 3-37
- camera properties 3-35
 - illustration showing 3-36
- CameraPosition property 3-35
 - and perspective 3-37
 - fly-by 3-37
- CameraPositionMode property 3-35
- CameraTarget property 3-35
- CameraTargetMode property 3-35
- CameraUpVector property 3-35, 3-39
 - example 3-40
- CameraUpVectorMode property 3-35
- CameraViewAngle property 3-35
 - and perspective 3-39
 - zooming with 3-38
- CameraViewAngleMode property 3-35, 3-39
- CData property
 - images 5-14
 - patches 6-11
- CDataMapping property 3-15
 - images 5-14
 - patches 6-11
- character sets

- encoding 7-15
 - printing 7-15, 7-27
- `cla` 8-34
- `clabel` 4-34, 4-36
- `clf` 8-34
- `close` 8-41
- close request function
 - default 8-41
- `closereq.m` 8-41
- `CloseRequestFcn` property 8-41
 - default value 8-41
 - errors in 8-42
 - overriding 8-42
- closing Figures 8-41
- closing MATLAB, errors occurring when 8-42
- color limits, calculating 10-33
- color property of lights 3-21
- color separations 7-14
- `colorbar` 3-14, 6-16
- `colormap` 2-6
- `colormap` 3-13
- colormaps
 - altering 3-17
 - brightening 3-17
 - brightness component of TV signal 3-17
 - continuous tone for printing 9-26
 - displaying 3-14
 - for surfaces 3-12
 - functions that create 3-13
 - large 9-10
 - minimum size 9-11
 - range of RGB values in 3-12
 - simulating multiple 10-32
 - size of dithermap 9-14
- `ColorOrder` 10-37
- colors
 - changing color scheme 10-30
- colormaps 3-12, 9-9
- controlled by Axes 10-29
- controlled by Figure properties 9-8
- dithering 3-19, 9-13
- effects of dithering 9-14
- fixed 9-9
- indexed 3-12
 - direct 3-15
 - scaled 3-15
- indexed and dithering 9-13
- interpreted by surfaces 3-12
- mapping to data 10-31
- NTSC encoding of 3-17
- of patches 6-11
- of surface plots 3-12
- reversing for printing 9-24
- scaling algorithm 3-15
- shared 9-12
- size of dithermap 9-14
- specifying Figure colors 2-5
- specifying for surface plot, example 3-15
- truecolor 3-12
 - on indexed color systems 3-19
 - specifying 3-17
- typical RGB values 3-13
- used for plotting 2-14, 10-37
- using a large number 9-10
- command-line switches for printing 7-11
- compass 4-28
- compass plots 4-28
- complex numbers, plotting 2-12
 - with feather 4-30
- contour 4-34
- contour plots 4-34
 - algorithm 4-38
 - filled 4-37
 - in polar coordinates 4-40

- labeling 4-36
- specifying contour levels 4-38, 4-40
- contour3 4-34
- contourc 4-34, 4-38
- contourf 4-34, 4-37
- coordinate system and viewpoint 3-32
- copying graphics objects 8-30
- current
 - Axes 8-27
 - Figure 8-27
 - object 8-27
- cursors, see pointers
- CYMK color separations 7-14

D

- DataAspectRatio property 3-51
 - example 3-55
 - images 5-11
- DataAspectRatioMode property 3-51
- default
 - aspect ratio 3-52
 - azimuth
 - 2-D 3-32
 - 3-D 3-32
 - CameraPosition 3-37
 - CameraTarget 3-37
 - CameraUpVector 3-37
 - CameraViewAngle 3-37
 - CloseRequestFcn 8-41
 - elevation
 - 2-D 3-32
 - 3-D 3-32
 - factory 8-19
 - Figure color scheme 2-5
 - Projection 3-37

- property values 8-20-8-26
 - removing 8-22
 - search path, diagram 8-21
 - setting to factory defaults 8-23
 - view 3-36
- del 2 3-16
- deleting graphics objects 8-30
- device drivers 7-8, 7-17
- diffuse reflection 3-25
- DiffuseStrength property 3-22
 - illustration 3-25
- direct color mapping 3-15
- direction cosines 3-40
- discrete data graphs 4-20-4-27
 - stairstep plots 4-26
 - stem plots 4-20
- dithering 9-13
 - algorithm 9-13
 - effects of 9-14
- Dithermap property 9-13
- DithermapMode property 9-13, 9-14

E

- edge effects and lighting 3-28
- EdgeColor property 3-23
- EdgeLighting property 3-23
- edges of patches 6-13
- efficient programming 8-44, 8-45
- elevation of viewpoint 3-32
 - default 2-D 3-32
 - default 3-D 3-32
 - limitations 3-34
- Encapsulated PostScript 7-17
 - preview images 7-14
- Enhanced Metafiles 7-39
- erase modes 4-47

- and printing 4-50
 - background 4-51
 - images 5-17
 - none 4-48
 - xor 4-51
 - errors closing MATLAB 8-42
 - examples
 - 2-D graphs 2-2
 - 3-D graph 3-2
 - animation 4-47
 - area graphs 4-10
 - axis 3-49
 - bar graphs 4-2
 - changing CameraPosition 3-37
 - contour plots 4-34
 - copying graphics objects 8-30
 - custom pointers 9-19
 - DataAspectRatio property 3-55
 - del 2 3-16
 - direction and velocity graphs 4-28
 - direction cosines 3-40
 - discrete data graphs 4-20
 - displaying real objects 3-58
 - double axis graphs 10-26
 - finding objects handles 8-29
 - histograms 4-16
 - hold 8-39
 - lighting 3-30
 - line 8-36
 - linospace 3-8
 - meshgrid 3-4, 3-8
 - movies 4-46
 - multiline text 10-12
 - newplot 8-36
 - object creation functions 8-11
 - overlapping axes 10-15
 - PaperPosition property 9-23
 - parametric surfaces 3-10
 - pie charts 4-13
 - plot 2-7
 - complex data 2-12
 - plot3 3-3
 - PlotBoxAspectRatio property 3-56
 - plotting linestyles 10-38
 - ScreenSize property 9-7
 - setting default property values 8-23
 - simulating multiple colormaps 10-32
 - specifying Figure position 9-7
 - specifying truecolor
 - surfaces 3-17
 - stretch-to-fill 3-55
 - subplot 2-3
 - text 2-20
 - text annotation 10-10
 - texture mapping 3-19
 - translation 2-D 3-41
 - unevenly sampled data 3-8
 - view 3-39
 - extent of computer screen 9-6
- F**
- FaceColor property 3-23
 - FaceLighting property 3-23
 - Faces property 6-7
 - FaceVertexCData property 6-10, 6-11
 - factory defaults 8-19
 - feather 4-28, 4-29
 - feather plots 4-29
 - Figures
 - callback routine execution properties 9-3
 - CloseRequestFcn 8-41
 - closing 8-41
 - colormap properties 9-3

- current selections properties 9-3
- defining custom pointers 9-18
- defining pointers 9-17
- defining the color of 2-5
- fixed colors 9-9
- for plotting 2-3
- general information properties 9-3
- handle properties 9-3
- index color properties 9-8
- introduction to 9-2
- NextPlot property 8-34
- nonactive 9-12
- pointer definition properties 9-3
- positioning 9-5
- positioning example 9-7
- positioning on the printed page 9-21
- preparing to accept graphics 8-33
- printing 7-1, 9-21
- printing properties 9-3
- properties, list of 9-3
- protecting from output 8-39
- rendering graphics properties 9-3
- rendering properties 9-15
- reversing for printing 9-24
- saving as M-files 7-20
- specifying
 - for printing 7-12
 - pointers 9-17
- standard plotting behavior 8-37
- style and appearance properties 9-3
- units 9-6
- visible property 8-41
- with multiple axes 2-3
- fill, properties changed by 8-45
- fill3, properties changed by 8-45
- findobj 8-29
- fixed colors 9-9

- FixedColors property 9-9
- Floyd-Steinberg dithering algorithm 9-13
- fly-by effect 3-37
- fonts
 - axis labels 10-5
 - printing 7-27
 - UNIX systems 7-29
 - Windows systems 7-28
- functions
 - convenience forms 8-13
 - high-level vs. low-level 8-13
 - to create graphics objects 8-10

G

- gca 8-28
 - handle visibility 8-40
- gcf 8-28
 - handle visibility 8-40
- gco 8-28
- get 8-15
- getframe 4-45
- Ghostscript print drivers 7-9
- gi nput 3-41, 4-43
- Gouraud lighting algorithm 3-24
- gradi ent 4-32
- graphical input 4-43
- graphics
 - elementary plotting functions 2-7
 - M-files, structure of 8-37
- graphics objects 8-2
 - accessing handles 8-27
 - accessing hidden handles 8-40
 - Axes 8-6, 10-2
 - See also Axes chapter
 - controlling where they draw 8-33
 - copying 8-30

deleting 8-30
 Figures 8-4
 See also Figure chapter
 functions that create 8-10
 convenience forms 8-13
 handle validity versus visibility 8-42
 Handle visibility property 8-40
 hierarchy 8-2
 images 8-6
 See also Image chapter
 invisible handles 8-40
 Lights 8-6
 See also Building 3-D Graphs chapter
 Line 8-6
 Patches 8-6
 See also 3-D Modeling chapter
 properties 8-7
 changed by functions 8-45
 changed when created 8-12
 common to all objects 8-8
 factory defined 8-19
 getting current values 8-17
 listing possible values 8-15
 querying in groups 8-19
 search path for default values 8-20
 searching for 8-29
 setting values 8-15
 property names 8-14
 Root 8-4
 setting parent of 8-12
 Surface 8-6
 See also Building 3-D Graphs chapter
 Text 8-7
 Uicontrol 8-4
 Uimenu 8-5
 graphs
 2-D 2-2

annotating 2-19
 area 4-10-4-12
 bar 4-2-4-10
 horizontal 4-6
 compass plots 4-28
 contour plots 4-34-4-42
 direction and velocity 4-28-4-33
 discrete data 4-20-4-27
 feather plots 4-29
 histograms 4-16-4-19
 pie charts 4-13-4-15
 quiver plots 4-31
 stairstep plots 4-26
 steps to create 3-D 3-2
 with double axes 10-26
 Greek characters
 see text function
 using to annotate 2-20
 griddata 3-8
 grids, coincident 10-27
 gtext 2-19

H

Hadamard matrix 3-10
 Handle Graphics
 graphics objects 8-2
 hierarchy of graphics objects 8-2
 organization of 8-2
 handles to graphics objects 8-27
 finding 8-29
 Handle visibility property 8-40
 HDF 5-2
 hidden 3-11
 hidden line removal 3-11
 high-level functions 8-13
 hist 4-16

- histograms 4-16
 - in polar coordinates 4-17
 - labeling the bins 4-18
 - rose plot 4-17
 - specifying number of bins 4-18
- hold 2-10
 - and NextPlot 8-35
 - testing state of 8-38
- hold state, testing for 8-39
- Horizontal Alignment property 10-9
- HPGL 7-18

I

- image 5-10
 - properties changed by 8-45
- images
 - 8-bit 5-6
 - indexed 5-6
 - intensity 5-7
 - truecolor 5-7
 - erase modes 5-17
 - file formats supported 5-19
 - indexed 5-3
 - information about files 5-21
 - intensity 5-3
 - numeric classes 5-2
 - printing 5-13
 - properties 5-14
 - CData 5-14
 - CDataMapping 5-14
 - XData and YData 5-15
 - reading from file 5-19
 - size and aspect ratio 5-10
 - truecolor 5-4
 - type read by MATLAB 5-2
 - types 5-3

- writing to file 5-19
- imagesc 5-3
- imfinfo 5-21
- imread 5-19
- imwrite 5-21
- indexed color
 - displays 9-8
 - dithering truecolor 9-13
 - surfaces 3-12
- interpolated colors
 - patches 6-9
 - indexed vs. truecolor 6-18
 - See also shading
- interpreter property 10-7
- InvertHardCopy property 9-24
- ishold 8-39

J

- JPEG 5-2

L

- labeling axes 2-19
- Laplacian of a matrix 3-16
- LaTeX, see TeX 10-6
- legend 2-19, 4-23
- light 3-21
- lighting 3-21-3-31
 - algorithms
 - flat 3-24
 - Gouraud 3-24
 - Phong 3-24
 - ambient light 3-25
 - backface 3-27
 - diffuse reflection 3-25
 - example 3-30

- important properties 3-21
- properties that affect 3-22
- reflectance characteristics 3-25-3-27
- specular
 - color 3-27
 - exponent 3-26
 - reflection 3-25
- lighting command 3-24
- limits
 - axes 2-15, 10-18
- line styles
 - printing 7-29
 - used for plotting 2-14
 - redefining 10-38
- line, example 8-36
- lines
 - adding to existing graph 2-10
 - marker types 2-14
 - removing hidden 3-11
 - styles 2-14
- LineStyleOrder property 10-38
- linspace 3-8
- loglog, properties changed by 8-46
- low-level functions 8-13

M

- Macintosh
 - printing 7-4, 7-24
- mapping data to color 10-31
- markers used for plotting 2-14
- material command 3-25
- mathematical functions
 - visualizing with surface plot 3-6
- MATLAB
 - history ii
- MATLAB 4 color scheme 2-6

- MATLAB, quitting 8-42
- matrix
 - displaying contours 4-35
 - Hadamard 3-10
 - initializing for movie 4-46
 - plotting 2-10
 - representing as
 - area graph 4-10
 - bar graph 4-3
 - histogram 4-17
 - surface 3-5
- mesh 3-5
- meshc 4-40
- meshgrid 3-6
- Metafiles 7-39
 - enhanced 7-39
- M-files
 - basic structure of graphics 8-37
 - closereq 8-41
 - saving Figures as 7-20
 - to set color mapping 10-35
 - using newplot 8-35
 - writing efficient 8-44
- Microsoft Windows
 - printing 7-3
- MinColormap property 9-10
- movie 4-45, 4-47
- moviein 4-45
- movies 4-45
 - example 4-45
- multiaxis axes 10-26
- multiline text 10-12

N

- NaNs, avoiding in data 3-6
- newplot 8-35

- example using 8-36
- NextPlot property 8-34
 - add 8-34
 - replace 8-34
 - replacechildren 8-34, 8-38
 - setting plotting color order 10-38
- nonuniform data, plotting 3-7
- Normal Mode property 3-23
- NTSC color encoding 3-17

O

- organization of Handle Graphics 8-2
- orientation of printed Figures 7-27
- orthographic projection 3-43
 - and Z-buffer 3-45

P

- painters algorithm
 - printing 7-32
- paper size 7-26
- PaperOrientation property 9-21
- PaperPosition property 9-21
 - example 9-23
- PaperPositionMode property 9-21, 9-23
- PaperSize property 9-21
- PaperType property 9-21
- PaperUnits property 9-21
- parametric surfaces 3-9
- parent, of graphics object 8-12
- patch
 - behavior of function 6-4
 - interpreting color 6-5
- patches
 - coloring 6-11
 - faces and edges 6-12

- face coloring
 - flat 6-8
 - interpolated 6-9
- indexed color 6-15
 - direct 6-16
 - scaled 6-15
- interpreting color data 6-14
- multifaceted 6-6
- single polygons 6-2
- specifying faces and vertices 6-7
- truecolor 6-17
- ways to specify 6-2
- PCX 5-2
- perspective projection 3-43
 - and Z-buffer 3-45
- Phong lighting algorithm 3-24
- pie charts 4-13
 - labeling 4-14
 - offsetting a slice 4-13
 - removing a piece 4-15
- plot 2-7
 - properties changed by 8-46
- plot box 3-36
- plot3 3-3
 - properties changed by 8-46
- PlotBoxAspectRatio property 3-51
 - example 3-56
- PlotBoxAspectRatioMode property 3-51
- plotting
 - 3-D
 - matrices 3-4
 - vectors 3-3
 - adding to existing graph 2-10
 - annotating graphs 2-19
 - area graphs 4-10
 - bar graphs 4-2
 - compass plots 4-28

- complex data 2-12
- contour plots 4-34
- contours, labeling 4-36
- creating a plot 2-7
- data-point markers 2-14
- elementary functions for 2-7
- feather plots 4-29
- interactive 4-43
- line colors 10-37
- line styles 2-14
- matrices 2-10
- multiple graphs 2-8
- nonuniform data 3-7
- overlying bar graphs 4-8
- quiver plots 4-31
- specifying line styles 2-9, 10-38
- stairstep plots 4-26
- stem plots 4-20
- surfaces 3-5
- to subaxis 2-3
- vector data 2-7
- windows for 2-3
- Pointer property 9-18
- pointers
 - custom 9-18
 - example defining 9-19
 - specifying 9-17
- PointerShapeCData property 9-18
- PointerShapeHotSpot property 9-18
- polar 4-42
- polar coordinates
 - contour plots 4-40
 - rose plot 4-17
- polygons, creating with patch 6-2
- position of Figure 9-5
- Position property
 - Axes 10-13
 - Figure 9-5
 - position rectangle 3-36
 - positioning of Axes 10-13
- PostScript 7-17
 - character-set encoding 7-15
 - CMYK color separations 7-14
- preview images for EPS 7-14
- print 7-6
- printer, specifying 7-13
- printing
 - 3-D scenes 3-46
 - Adobe Illustrator 88 7-20
 - appending to an existing file 7-15
 - aspect ratio 7-25
 - changing colors 7-35
 - character sets 7-27
 - command line 7-6
 - controlling output 7-25
 - device drivers 7-8, 7-17
 - Figure size 7-5, 7-25
 - Figures 7-1, 9-21
 - fonts 7-27
 - Ghostscript drivers 7-9
 - HPGL 7-18
 - images 5-13
 - introduction 7-2
 - inverting background color 7-35
 - line styles 7-29
 - Macintosh 7-4, 7-24
 - menu 7-3
 - Microsoft Windows 7-3, 7-21
 - network 7-23
 - options 7-11
 - orientation of Figure 7-27
 - painters algorithm 7-32
 - paper size 7-26
 - positioning the Figure on the page 9-21

- PostScript 7-17
 - resolution 7-15
 - with painters renderer 9-16
 - with Z-buffer renderer 9-16
 - reversing colors 9-24
 - saving Figures as M-files 7-20
 - specifying Figures 7-12
 - specifying printer 7-13
 - UNIX 7-4
 - Windows metafiles 7-39
 - WYSIWYG 9-23
 - Z-buffer 7-32, 9-15
 - printopt 7-6
 - programming, efficiently 8-44
 - Projection property 3-35
 - projection types 3-43-3-46
 - camera position 3-44
 - orthographic 3-43
 - perspective 3-43
 - rendering method 3-44
 - properties
 - automatic Axes 10-23
 - Axes 10-2
 - changed by built-in functions 8-45
 - changed by object creation functions 8-12
 - defining in startup. m 8-26
 - for labeling Axes 10-4
 - naming convention 8-14
 - of Axes 10-2
 - See also graphics objects
 - specifying default values 8-22
 - property values
 - defaults 8-20
 - defined by MATLAB 8-19
 - getting 8-15
 - resetting to default 8-22
 - setting 8-15
 - specifying defaults 8-22
 - user defined 8-20
 - pseudocolor displays, see indexed color
- ## Q
- qui ver 4-28, 4-31
 - quiver plots 4-31
 - 2-D 4-31
 - 3-D 4-32
 - combined with contour plot 4-32
 - displaying velocity vectors 4-33
 - qui ver3 4-28
- ## R
- realism, adding with lighting 3-21
 - realistic display 3-58
 - reflection, specular and diffuse 3-25
 - Renderer property 3-19
 - and printing 9-16
 - RenderMode property 3-19
 - rendering
 - options 9-15
 - Z-buffer 9-15
 - reset 8-34
 - resolution for printing 7-15
 - reversing colors for printing 9-24
 - RGB color values 3-13
 - rgbplot 3-17
 - rose 4-17
 - rotation
 - about viewing axis 3-39
 - without resizing 3-39

S

- scaled color mapping 3-15
- screen extent, determining 9-6
- ScreenSize property 9-6
 - example 9-7
- Select ionType property, example 3-42
- semilogx, properties changed by 8-46
- semilogy, properties changed by 8-47
- set 8-15
- setting property values 8-15
- ShareColors property 9-12
- ShowHiddenHandles property 8-40
- specular
 - color 3-27
 - exponent 3-26
 - highlight 3-26
 - reflection 3-25
- SpecularColorReflectance property 3-23
 - illustration 3-27
- SpecularExponent property 3-22
 - illustration 3-27
- SpecularStrength property 3-22
 - illustration 3-25
- sphere 3-19
- spline 4-43
- stairs 4-26
- stairstep plot 4-26
- stem 4-20
- stem plots 4-20
 - 3-D 4-24
 - overlaid with line plot 4-22
- stem3 4-24
- stretch-to-fill 3-47
 - overriding 3-54
- string variable, in text 10-10
- style property of lights 3-21
- subplot 2-3

surf 3-5

Surfaces

- CData 3-20
- coloring 3-12
- curvature mapped to color 3-16
- FaceColor, texturemap 3-20
- parametric 3-9
- plotting 3-5
 - nonuniformly sampled data 3-7
- surf 4-40
- symbols, TeX characters 10-6

T

TeX

- available characters 10-7
- creating mathematical symbols 10-6
- symbols in text 2-20, 10-6

text

- adding to Axes 10-8
- alignment 10-9
- for labeling plots 2-20
- horizontal and vertical alignment 10-9
- multiline 10-12
- placing interactively 2-21
- placing outside of axes 10-15
- positioning 10-8
- TeX characters 10-6
- using variables in 10-10

text 2-19

texture mapping 3-19

thin line styles 7-31

three-dimensional objects, creating with patch
6-2

tick marks, on axes 2-16, 10-20

TIFF 5-2

title 2-19

translating the viewpoint 3-41

truecolor

- dithering on indexed systems 9-13

- patches 6-17

- rendering method used for 3-19

- simulating 3-19

- surface plots 3-12, 3-17

TrueType fonts 7-29

U

Uicontrol graphics objects 8-4

Uimenu graphics objects 8-5

uint8 arrays 5-6

- operations supported on 5-9

units

- Axes 10-14

- used by Figures 9-6

UNIX

- printing 7-4

V

vectors

- determined by direction cosines 3-40

velocity vectors displayed with `quiver` 4-33

vertex normals and back face lighting 3-28

`VertexNormals` property 3-23

`VerticalAlignment` property 10-9

`Vertices` property 6-7

view

- azimuth of viewpoint 3-32

- camera properties 3-35

- coordinate system defining 3-32

- elevation of viewpoint 3-32

- limitation of azimuth and elevation 3-34

- MATLAB's default behavior 3-36

- projection types 3-43

- specifying 3-35

- specifying with azimuth and elevation 3-32

- translation of 3-41

view 3-32

- example of rotation 3-39

- limitations using 3-34

viewing axis 3-36

- moving camera along 3-37

viewpoint, controlling 3-32-3-34

visibility of graphics objects 8-42

visualizing mathematical functions 3-6

W

Windows

- metafiles 7-39

- printing 7-3, 7-21

X

`xlabel` 2-19

XWD 5-2

Y

`ylabel` 2-19

Z

Z-buffer 9-15

- orthographic projection 3-45

- perspective projection 3-45

- printing 7-32, 9-15

- rendering truecolor 3-19

`zlabel` 2-19

zooming by setting camera angle 3-38