

MATLAB[®]

The Language of Technical Computing

Computation

Visualization

Programming

Using MATLAB Graphics

Version 6



How to Contact The MathWorks:



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



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



508-647-7000	Phone
--------------	-------



508-647-7001	Fax
--------------	-----



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

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

Using MATLAB Graphics

© COPYRIGHT 1984 - 2002 by The MathWorks, Inc.

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

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by or for the federal government of the United States. By accepting delivery of the Program, the government hereby agrees that this software qualifies as "commercial" computer software within the meaning of FAR Part 12.212, DFARS Part 227.7202-1, DFARS Part 227.7202-3, DFARS Part 252.227-7013, and DFARS Part 252.227-7014. The terms and conditions of The MathWorks, Inc. Software License Agreement shall pertain to the government's use and disclosure of the Program and Documentation, and shall supersede any conflicting contractual terms or conditions. If this license fails to meet the government's minimum needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to MathWorks.

MATLAB, Simulink, Stateflow, Handle Graphics, and Real-Time Workshop are registered trademarks, and TargetBox is a trademark of The MathWorks, Inc.

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

Printing History:	January 1997	First printing	New for MATLAB 5.1
	January 1998	Second printing	Revised for MATLAB 5.2
	January 1999	Third printing	Revised for MATLAB 5.3 (Release 11)
	September 2000	Fourth printing	Revised for MATLAB 6.0 (Release 12)
	June 2001	Online only	Revised for MATLAB 6.1 (Release 12.1)
	July 2002	Online only	Revised for MATLAB 6.5 (Release 13)

Graphics

Overview of MATLAB Graphics

1

Plotting Your Data	1-2
Anatomy of a Plot	1-3
Editing Plots	1-4
Interactive Plot Editing	1-4
Using Functions to Edit Graphs	1-4
Using Plot Edit Mode	1-5
Starting Plot Edit Mode	1-6
Exiting Plot Edit Mode	1-6
Selecting Objects in a Graph	1-6
Cutting, Copying, and Pasting Objects	1-7
Moving and Resizing Objects	1-7
Editing Objects in a Graph	1-8
Saving Your Work	1-9
Saving a Graph in MAT-File Format	1-9
Saving to a Different Format (Exporting Figures)	1-10
Printing Figures	1-10
Getting Help	1-12
Changing Your View of an Axes	1-13
Zooming In and Out on an Axes	1-13
Rotating 3-D Axes	1-14
Using the Property Editor	1-15
Starting the Property Editor	1-16

Closing the Property Editor	1-17
Editing Object Properties	1-17
Navigating Among Objects in a Graph	1-18
Identifying Objects in a Graph	1-20
Applying Your Changes	1-21
Using the Data Statistics Tool	1-22
Adding Plots of Statistics to a Graph	1-22
Saving Statistics to the Workspace	1-23

Basic Plotting

2

Basic Plotting Commands	2-2
Creating Line Plots	2-2
Specifying Line Style	2-4
Specifying the Color and Size of Lines	2-6
Adding Plots to an Existing Graph	2-7
Plotting Only the Data Points	2-8
Plotting Markers and Lines	2-9
Line Styles for Black and White Output	2-10
Setting Default Line Styles	2-11
Line Plots of Matrix Data	2-13
Plotting Imaginary and Complex Data	2-15
Plotting with Two Y-Axes	2-16
Combining Linear and Logarithmic Axes	2-16
Setting Axis Parameters	2-19
Axis Limits and Ticks	2-19
Example – Specifying Ticks and Tick Labels	2-22
Setting Aspect Ratio	2-23
Figure Windows	2-26
Displaying Multiple Plots per Figure	2-26

Specifying the Target Axes	2-28
Default Color Scheme	2-28

Formatting Graphs

3

Overview	3-2
Adding Titles to Graphs	3-3
Using the Title Option on the Insert Menu	3-3
Using the Property Editor to Add a Title	3-4
Using the title Function	3-5
Adding Legends to Graphs	3-6
Using the Legend Option on the Insert Menu	3-7
Using the Legend Function	3-7
Positioning the Legend	3-8
Editing the Legend	3-9
Removing the Legend	3-10
Adding Axis Labels to Graphs	3-11
Using the Label Options on the Insert Menu	3-12
Using the Property Editor to Add Axis labels	3-12
Using Axis-Label Commands	3-15
Adding Text Annotations to Graphs	3-17
Creating Text Annotations in Plot Editing Mode	3-18
Creating Text Annotations with the text or gtext Command .	3-18
Text Alignment	3-22
Example – Aligning Text	3-23
Editing Text Annotations	3-24
Mathematical Symbols, Greek Letters, and TeX Characters .	3-25
Using Character and Numeric Variables in Text	3-27
Example - Multiline Text	3-28
Drawing Text in a Box	3-29
Adding Arrows and Lines to Graphs	3-31

Creating Arrows and Lines in Plot Editing Mode	3-31
Editing Arrows and Line Annotations	3-32
Adding Plots of Basic Statistics to Graphs	3-33
Example - Plotting the Mean of a Data Set	3-34
Formatting Plots of Data Statistics	3-36
Statistics Plotted by the Data Statistics Tool	3-36
Viewing Statistics for Multiple Plots	3-37
Saving Statistics to the MATLAB Workspace	3-38

Creating Specialized Plots

4

Bar and Area Graphs	4-2
Types of Bar Graphs	4-2
Stacked Bar Graphs to Show Contributing Amounts	4-5
Specifying X-Axis Data	4-7
Overlaying Plots on Bar Graphs	4-9
Area Graphs	4-11
Comparing Datasets with Area Graphs	4-12
Pie Charts	4-14
Removing a Piece from a Pie Charts	4-16
Histograms	4-17
Histograms in Cartesian Coordinate Systems	4-17
Histograms in Polar Coordinates	4-19
Specifying Number of Bins	4-20
Discrete Data Graphs	4-22
Two-Dimensional Stem Plots	4-22
Combining Stem Plots with Line Plots	4-25
Three-Dimensional Stem Plots	4-26
Stairstep Plots	4-29
Direction and Velocity Vector Graphs	4-31
Compass Plots	4-31

Feather Plots	4-32
Two-Dimensional Quiver Plots	4-34
Three-Dimensional Quiver Plots	4-35
Contour Plots	4-37
Creating Simple Contour Plots	4-37
Labeling Contours	4-39
Filled Contours	4-40
Drawing a Single Contour Line at a Desired Level	4-41
The Contouring Algorithm	4-42
Changing the Offset of a Contour	4-43
Displaying Contours in Polar Coordinates	4-44
Interactive Plotting	4-48
Animation	4-50
Movies	4-50
Example – Visualizing an FFT as a Movie	4-51
Erase Modes	4-52

Displaying Bit-Mapped Images

5

Overview	5-2
Images in MATLAB	5-4
Bit Depth Support	5-4
Data Types	5-4
Image Types	5-6
Indexed Images	5-6
Intensity Images	5-7
RGB (Truecolor) Images	5-9
Working with 8-Bit and 16-Bit Images	5-11
8-Bit and 16-Bit Indexed Images	5-11
8-Bit and 16-Bit Intensity Images	5-12

8-Bit and 16-Bit RGB Images	5-12
Mathematical Operations Support for uint8 and uint16	5-13
Other 8-Bit and 16-Bit Array Support	5-13
Summary of Image Types and Numeric Classes	5-14
Reading, Writing, and Querying Graphics Image Files ...	5-15
Reading a Graphics Image	5-15
Writing a Graphics Image	5-16
Obtaining Information About Graphics Files	5-16
Displaying Graphics Images	5-18
Summary of Image Types and Display Methods	5-19
Controlling Aspect Ratio and Display Size	5-19
The Image Object and Its Properties	5-23
CData	5-23
CDataMapping	5-23
XData and YData	5-24
EraseMode	5-26
Printing Images	5-28
Converting the Data or Graphic Type of Images	5-29

Printing and Exporting

6

Overview of Printing and Exporting	6-2
Print and Export Operations	6-2
Graphical User Interfaces	6-2
Command Line Interface	6-3
Specifying Parameters and Options	6-5
Default Settings and How to Change Them	6-6
How to Print or Export	6-9
Printing a Figure	6-9
Printing to a File	6-13

Exporting to a File	6-15
Exporting to the Windows Clipboard	6-20
Examples of Basic Operations	6-23
Printing a Figure at Screen Size	6-23
Printing with a Specific Paper Size	6-24
Printing a Centered Figure	6-25
Exporting in a Specific Graphics Format	6-26
Exporting in EPS Format with a TIFF Preview	6-27
Exporting a Figure to the Clipboard	6-28
Changing a Figure's Settings	6-31
Selecting the Figure	6-33
Selecting the Printer	6-33
Setting the Figure Size and Position	6-34
Setting the Paper Size or Type	6-38
Setting the Paper Orientation	6-39
Selecting a Renderer	6-41
Setting the Resolution	6-44
Setting the Axes Ticks and Limits	6-47
Setting the Background Color	6-49
Setting Line and Text Characteristics	6-50
Setting the Line and Text Color	6-52
Setting CMYK Color	6-54
Excluding User Interface Controls	6-55
Producing Uncropped Figures	6-55
Choosing a Graphics Format	6-57
Frequently Used Graphics Formats	6-58
Factors to Consider in Choosing a Format	6-59
Properties Affected by Choice of Format	6-61
Impact of Rendering Method on the Output	6-63
Description of Selected Graphics Formats	6-63
How to Specify a Format for Exporting	6-66
Choosing a Printer Driver	6-68
Types of Printer Drivers	6-68
Factors to Consider in Choosing a Driver	6-69
Driver-Specific Information	6-72
How to Specify the Printer Driver to Use	6-76

Troubleshooting	6-78
Printing Problems	6-79
Exporting Problems	6-82
General Problems	6-86

Handle Graphics Objects

7

Graphics Object Hierarchy	7-2
Types of Graphics Objects	7-3
Handle Graphics Objects	7-4
Object Properties	7-8
Changing Values	7-8
Default Values	7-8
Properties Common to All Objects	7-9
Graphics Object Creation Functions	7-11
Example — Creating Graphics Objects	7-12
Parenting	7-14
High-Level Versus Low-Level	7-14
Simplified Calling Syntax	7-15
Setting and Querying Property Values	7-17
Setting Property Values	7-17
Querying Property Values	7-19
Factory-Defined Property Values	7-21
Setting Default Property Values	7-22
How MATLAB Searches for Default Values	7-22
Defining Default Values	7-24
Examples — Setting Default LineStyles	7-25
Accessing Object Handles	7-29
The Current Figure, Axes, and Object	7-29
Searching for Objects by Property Values — findobj	7-31

Copying Objects	7-32
Deleting Objects	7-34
Controlling Graphics Output	7-36
Specifying the Target for Graphics Output	7-36
Preparing Figures and Axes for Graphics	7-36
Targeting Graphics Output with newplot	7-38
Example — Using newplot	7-39
Testing for Hold State	7-41
Protecting Figures and Axes	7-42
The Close Request Function	7-44
Handle Validity Versus Handle Visibility	7-45
Saving Handles in M-Files	7-47
Properties Changed by Built-In Functions	7-48
Callback Properties for Graphics Objects	7-51
Graphics Object Callbacks	7-51
Uicontrol, Uimenu, and Uicontextmenu Callbacks	7-51
Figures Callbacks	7-51
Function Handle Callbacks	7-53
Function Handle Syntax	7-53
Why Use Function Handle Callbacks	7-54
Example — Using Function Handles in a GUI	7-56

Figure Properties

8

Figure Objects	9-2
Positioning Figures	9-3
The Position Vector	9-3
Example – Specifying Figure Position	9-5
Controlling How MATLAB Uses Color	9-7

Indexed Color Displays	9-7
Colormap Colors and Fixed Colors	9-8
Using a Large Number of Colors	9-9
Nonactive Figures and Shared Colors	9-11
Dithering Truecolor on Indexed Color Systems	9-12
Selecting Drawing Methods	9-15
Backing Store	9-15
Double Buffering	9-15
Selecting a Renderer	9-16
Specifying the Figure Pointer	9-18
Defining Custom Pointers	9-19
Interactive Graphics	9-23

Axes Properties

9

Axes Objects	8-2
Labeling and Appearance Properties	8-3
Creating Axes with Specific Characteristics	8-3
Positioning Axes	8-5
The Position Vector	8-5
Position Units	8-6
Multiple Axes per Figure	8-7
Placing Text Outside the Axes	8-7
Multiple Axes for Different Scaling	8-8
Individual Axis Control	8-10
Setting Axis Limits	8-11
Setting Tick Mark Locations	8-12
Changing Axis Direction	8-13

Using Multiple X and Y Axes	8-16
Example – Double Axis Graphs	8-16
Automatic-Mode Properties	8-19
Colors Controlled by Axes	8-22
Specifying Axes Colors	8-22
Axes Color Limits – The CLim Property	8-25
Example – Simulating Multiple Colormaps in a Figure	8-25
Calculating Color Limits	8-26
Defining the Color of Lines for Plotting	8-29
Line Styles Used for Plotting – LineStyleOrder	8-31

3-D Visualization

Creating 3-D Graphs

10

A Typical 3-D Graph	10-2
Line Plots of 3-D Data	10-3
Representing a Matrix as a Surface	10-5
Mesh and Surface Plots	10-5
Visualizing Functions of Two Variables	10-6
Surface Plots of Nonuniformly Sampled Data	10-8
Parametric Surfaces	10-10
Hidden Line Removal	10-12
Coloring Mesh and Surface Plots	10-13
Coloring Techniques	10-13
Types of Color Data	10-13
Colormaps	10-14
Indexed Color Surfaces – Direct and Scaled Colormapping ..	10-16
Example – Mapping Surface Curvature to Color	10-17

Altering Colormaps	10-19
Truecolor Surfaces	10-20
Texture Mapping	10-23

Defining the View

11

Viewing Overview	11-2
Positioning the Viewpoint	11-2
Setting the Aspect Ratio	11-2
Default Views	11-3
Setting the Viewpoint with Azimuth and Elevation	11-4
Azimuth and Elevation	11-4
Defining Scenes with Camera Graphics	11-8
View Control with the Camera Toolbar	11-9
Camera Toolbar	11-9
Camera Motion Controls	11-12
Orbit Camera	11-12
Orbit Scene Light	11-13
Pan/Tilt Camera	11-14
Move Camera Horizontally/Vertically	11-15
Move Camera Forward and Backwards	11-16
Zoom Camera	11-17
Camera Roll	11-18
Walk Camera	11-19
Camera Graphics Functions	11-21
Example — Dollying the Camera	11-22
Example — Moving the Camera Through a Scene	11-24
Summary of Techniques	11-24
Graphing the Volume Data	11-24
Setting Up the View	11-25

Specifying the Light Source	11-25
Selecting a Renderer	11-26
Defining the Camera Path as a Stream Line	11-26
Implementing the Fly-Through	11-26
Low-Level Camera Properties	11-30
Default Viewpoint Selection	11-31
Moving In and Out on the Scene	11-31
Making the Scene Larger or Smaller	11-33
Revolving Around the Scene	11-33
Rotation Without Resizing of Graphics Objects	11-33
Rotation About the Viewing Axis	11-34
View Projection Types	11-36
Projection Types and Camera Location	11-37
Understanding Axes Aspect Ratio	11-41
Stretch-to-Fill	11-41
Specifying Axis Scaling	11-41
Specifying Aspect Ratio	11-42
Example — axis Command Options	11-43
Additional Commands for Setting Aspect Ratio	11-44
Axes Aspect Ratio Properties	11-46
Default Aspect Ratio Selection	11-47
Overriding Stretch-to-Fill	11-49
Effects of Setting Aspect Ratio Properties	11-50
Example — Displaying Real Objects	11-54

Lighting as a Visualization Tool

12

Lighting Overview	12-2
Lighting Examples	12-2
Lighting Commands	12-3

Light Objects	12-4
Adding Lights to a Scene	12-5
Properties That Affect Lighting	12-8
Selecting a Lighting Method	12-10
Face and Edge Lighting Methods	12-10
Reflectance Characteristics of Graphics Objects	12-11
Specular and Diffuse Reflection	12-11
Ambient Light	12-12
Specular Exponent	12-13
Specular Color Reflectance	12-14
Back Face Lighting	12-14
Positioning Lights in Data Space	12-17

Transparency

13

Making Objects Transparent	13-2
Specifying Transparency	13-3
Specifying a Single Transparency Value	13-5
Example – Transparent Isosurface	13-5
Mapping Data to Transparency	13-8
Size of the Alpha Data Array	13-9
Mapping Alpha Data to the Alphamap	13-9
Example: Mapping Data to Color or Transparency	13-9
Selecting an Alphamap	13-11
Example: Modifying the Alphamap	13-13

Creating 3-D Models with Patches

14

Introduction to Patch Objects	14-2
Defining Patches	14-2
Behavior of the patch Function	14-2
Creating a Single Polygon	14-4
Multi-Faceted Patches	14-6
Example – Defining a Cube	14-6
Specifying Patch Coloring	14-11
Patch Color Properties	14-11
Patch Edge Coloring	14-13
Example – Specifying Flat Edge and Face Coloring	14-13
Coloring Edges with Shared Vertices	14-14
Interpreting Indexed and Truecolor Data	14-16
Indexed Color Data	14-16
Truecolor Patches	14-19
Interpolating in Indexed Color Versus Truecolor	14-19

Volume Visualization Techniques

15

Overview of Volume Visualization	15-3
Examples of Volume Data	15-3
Selecting Visualization Techniques	15-3
Steps to Create a Volume Visualization	15-4
Volume Visualization Functions	15-5
Functions for Scalar Data	15-6
Functions for Vector Data	15-6
Visualizing Scalar Volume Data	15-8
Techniques for Visualizing Scalar Data	15-8

Visualizing MRI Data	15-9
Example - Ways to Display MRI DATA	15-9
Exploring Volumes with Slice Planes	15-15
Example – Slicing Fluid Flow Data	15-15
Modifying the Color Mapping	15-18
Connecting Equal Values with Isosurfaces	15-20
Example – Isosurfaces in Fluid Flow Data	15-20
Isocaps Add Context to Visualizations	15-22
Defining Isocaps	15-23
Example – Adding Isocaps to an Isosurface	15-24
Visualizing Vector Volume Data	15-27
Using Scalar Techniques with Vector Data	15-27
Specifying Starting Points for Stream Plots	15-28
Accessing Subregions of Volume Data	15-30
Stream Line Plots of Vector Data	15-32
Displaying Curl with Stream Ribbons	15-34
Displaying Divergence with Stream Tubes	15-36
Creating Stream Particle Animations	15-39
Vector Field Displayed with Cone Plots	15-42

Graphics

This section discusses techniques for plotting data and provides examples showing how to plot, annotate, and print graphs.

Overview of MATLAB Graphics	Editing plots and graphic object properties
Basic Plotting	Plotting vector and matrix data in 2-D representations
Formatting Graphs	Adding annotations, axis labels, titles, and legends to graphs
Creating Specialized Plots	Creating bar graphs, histograms, contour plots and other specialized plots
Displaying Bit-Mapped Images	Displaying and modifying bit-mapped images with MATLAB
Printing and Exporting	Printing graphs on paper and exporting graphs to standard graphic file formats
Handle Graphics Objects	MATLAB graphics objects and properties
Figure Properties	How to use figure properties
Axes Properties	How to use axes properties

Related Information

These other collections provide additional information about plotting.

3-D Visualization	Using viewing and lighting techniques to achieve complex graphic effects
Creating Graphical User Interfaces	How to include menus, push buttons, text boxes, and other user interface objects in MATLAB applications



Overview of MATLAB Graphics

Plotting Your Data (p. 1-2)	Illustrates the steps and commands used for a typical plotting task.
Editing Plots (p. 1-4)	Techniques for editing plots.
Using Plot Edit Mode (p. 1-5)	Using the Plot Editor tool to modify plots.
Saving Your Work (p. 1-9)	Ways to save your graph to reload into MATLAB or export to external applications.
Changing Your View of an Axes (p. 1-13)	Change your plot by zooming or rotating the axes.
Using the Property Editor (p. 1-15)	Easily get and set Handle Graphics properties on any objects in the current hierarchy.
Using the Data Statistics Tool (p. 1-22)	Perform basic statistical analysis on your data and plot the results.

Plotting Your Data

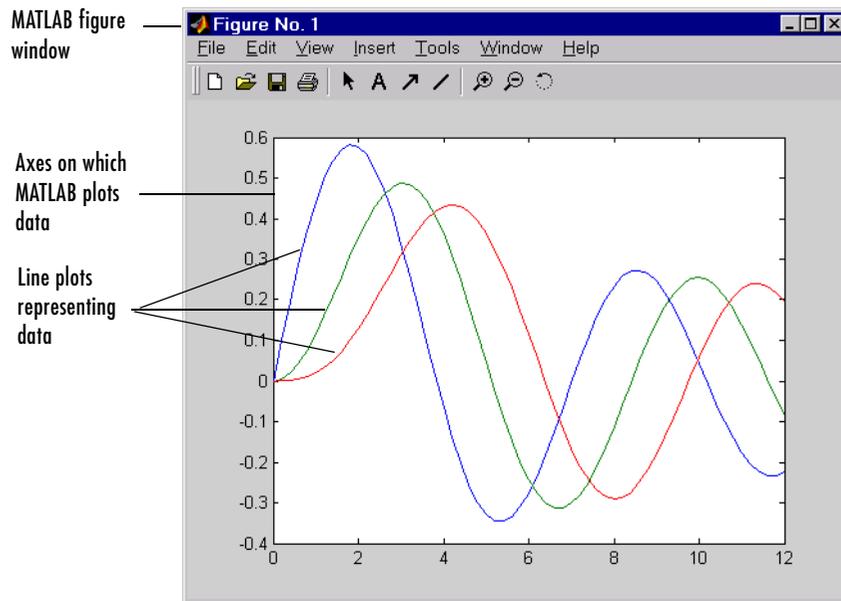
The process of constructing a basic graph to meet your presentation graphics requirements is outlined in the following table. The table shows seven typical steps and some example code for each.

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

Step	Typical Code
1 Prepare your data	<pre>x = 0:0.2:12; y1 = bessell(1,x); y2 = bessell(2,x); y3 = bessell(3,x);</pre>
2 Select a window and position a plot region within the window	<pre>figure(1) subplot(2,2,1)</pre>
3 Call elementary plotting function	<pre>h = plot(x,y1,x,y2,x,y3);</pre>
4 Select line and marker characteristics	<pre>set(h,'LineWidth',2,{'LineStyle'},{'--';':';'-.'}) set(h,{'Color'},{'r';'g';'b'})</pre>
5 Set axis limits, tick marks, and grid lines	<pre>axis([0 12 -0.5 1]) grid on</pre>
6 Annotate the graph with axis labels, legend, and text	<pre>xlabel('Time') ylabel('Amplitude') legend(h,'First','Second','Third') title('Bessel Functions') [y,ix] = min(y1); text(x(ix),y,'First Min \rightarrow',... 'HorizontalAlignment','right')</pre>
7 Export graph	<pre>print -depsc -tiff -r200 myplot</pre>

Anatomy of a Plot

MATLAB[®] plotting functions direct their output to a window that is separate from the command window. In MATLAB this window is referred to as a *figure*. For example, the following illustrates the plot of the Bessel functions, described in “Plotting Your Data” on page 1-2, highlighting the basic components of the graph.



By default, MATLAB uses line style and color to distinguish the data sets plotted in the graph. However, you can change the appearance of these graphic components or add annotations to the graph to help explain your data for presentation graphics. For information, see “Editing Plots” on page 1-4.

Editing Plots

MATLAB formats a graph to provide readability, setting the scale of axes, including tick marks on the axes, and using color and line style to distinguish the plots in the graph. However, if you are creating presentation graphics, you may want to change this default formatting or add descriptive labels, titles, legends and other annotations to help explain your data.

MATLAB supports two ways to edit the plots you create:

- Using the mouse to select and edit objects interactively
- Using MATLAB functions at the command-line or in an M-file

Interactive Plot Editing

If you enable plot editing mode in the MATLAB figure window, you can perform point-and-click editing of your graph. In this mode, you can change the format of objects in your graph by double-clicking on the object and changing the values of its properties. In plot editing mode, you access the properties through the graphical user interface, called the Property Editor.

For more information about interactive editing, see “Using Plot Edit Mode” on page 1-5. For information about editing object properties in plot editing mode, see “Using the Property Editor” on page 1-15.

Using Functions to Edit Graphs

If you prefer to work from the MATLAB command line or if you are creating an M-file, you can use MATLAB commands to edit the graphs you create. Taking advantage of the MATLAB Handle Graphics system, you can use the `set` and `get` commands to change the properties of the objects in a graph.

Note Plot editing mode provides an alternative way to access the properties of MATLAB graphic objects. However, you can only access a subset of object properties through this mechanism. You may need to use a combination of interactive editing and command line editing to achieve the effect you desire.

Using Plot Edit Mode

To start plot edit mode, click this button.

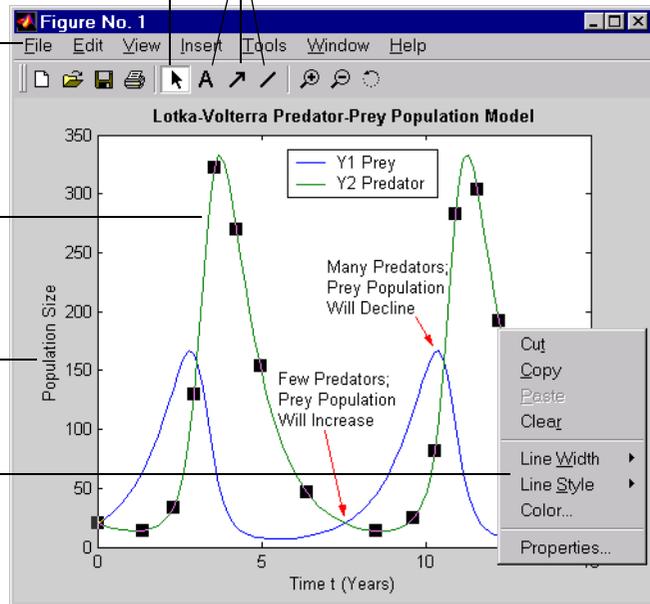
Use the **Edit**, **Insert**, and **Tools** menus to add objects or edit existing objects in a graph.

Double-click on an object to select it.

Position labels, legends, and other objects by clicking and dragging.

Access object-specific plot edit functions through context-sensitive pop-up menus.

Use these toolbar buttons to add text, arrows, and lines.



The MATLAB figure window supports a point-and-click style editing mode that you can use to customize the appearance of your graph. This section describes how to start plot edit mode and perform basic editing tasks, including:

- “Selecting Objects in a Graph” on page 1-6
- “Cutting, Copying, and Pasting Objects” on page 1-7
- “Moving and Resizing Objects” on page 1-7
- “Editing Objects in a Graph” on page 1-8
- “Saving Your Work” on page 1-9
- “Changing Your View of an Axes” on page 1-13

Starting Plot Edit Mode

Before you can select objects in a figure by clicking on them, you must activate plot editing mode. There are several ways to activate plot edit mode:

- Choose the **Edit Plot** option on the figure window **Tools** menu.
- Click on the selection button in the figure window toolbar.

Click this button to start plot edit mode.



- Choose an option from the **Edit** or **Insert** menu. For example, if you choose the **Axes Properties** option on the **Edit** menu, MATLAB activates plot edit mode and the axes appear selected.
- Run the `plottedit` command in the MATLAB command window.

When a figure window is in plot edit mode, the **Edit Plot** option on the **Tools** menu is checked and the selection button in the toolbar is highlighted.

Exiting Plot Edit Mode

To exit plot edit mode, click the selection button or click the **Edit Plot** option on the **Tools** menu. When plot edit mode is turned off, the selection button is no longer highlighted.

Selecting Objects in a Graph

To select an object in a graph:

- 1 Start plot edit mode.
- 2 Move the cursor over the object and click on it.

Selection handles appear on the selected object.

Selecting Multiple Objects

To select multiple objects at the same time:

- 1 Start plot edit mode.

- 2 Move the cursor over an object and shift-click to select it. Repeat for each object you want to select.

You can perform actions on all of the selected objects. For example, to remove a text annotation and an arrow annotation from a graph, select the objects and then select **Cut** from the **Edit** menu.

Deselecting Objects

To deselect an object, move the cursor off the object onto the figure window background and click the left mouse button. You can also shift-click on a selected object to deselect it.

Cutting, Copying, and Pasting Objects

To cut an object from a graph, or copy and paste an object in a graph, perform these steps:

- 1 Start plot edit mode.
- 2 Select the object.
- 3 Select the **Cut**, **Copy**, or **Paste** option from the **Edit** menu or use standard shortcut keys for your platform.

Alternatively, with plot edit mode enabled, you can right-click on an object and then select an editing command from the context menu associated with the object.

Note If you cut an axes label or title and then paste it back into a figure, the label or title is no longer anchored to the axis. If you move the axes, the label or title will not move with the axes.

Moving and Resizing Objects

To move or resize an object in a graph, perform these steps:

- 1 Start plot edit mode.

- 2 For axes objects only: unlock the axes by right-clicking on it and choosing **Unlock Axes Position** from the context menu.
- 3 Select the object. Selection handles appear on the object.

To move the object, drag it to the new location.

To resize the object, drag a selection handle.

Note You can move, but cannot resize text objects.

Editing Objects in a Graph

In MATLAB, every object in a graph supports a set of properties that control the graph's appearance and behavior. For example, line objects support properties that control thickness, color, and line style.

In plot edit mode, MATLAB provides a graphical user interface to object properties called the Property Editor. The Property Editor is a dialog box that supports a specific set of tabbed panels for each object you can select. For more information, see "Using the Property Editor" on page 1-15.

Saving Your Work

After editing a graph, you can:

- Save your work in a format that can be opened during another MATLAB session.
- Save your work in a format that can be used by other applications.

Saving a Graph in MAT-File Format

MATLAB supports a binary format in which you can save figures so that they can be opened in subsequent MATLAB sessions. MATLAB assigns these files the `.fig` file name extension.

To save a graph in a figure file:

- 1 Select **Save** from the figure window **File** menu or click the **Save** button on the toolbar. If this is the first time you are saving the file, the **Save As** dialog box appears.
- 2 Make sure that the **Save as type** is **Fig-file**.
- 3 Specify the name you want assigned to the figure file.
- 4 Click **OK**.

The graph is saved as a figure file (`.fig`), which is a binary file format used to store figures.

You can also use the `saveas` command.

If you want to save the figure in a format that can be used by another application, see “Saving to a Different Format (Exporting Figures)” on page 1-10.

Opening a Figure File

To open a figure file, perform these steps:

- 1 Select **Open** from the **File** menu or click the **Open** button on the toolbar.

- 2 Select the figure file you want to open and click **OK**.

The figure file appears in a new figure window.

You can also use the open command.

Saving to a Different Format (Exporting Figures)

To save a figure in a format that can be used by another application, such as the standard graphics file formats TIFF or EPS, perform these steps:

- 1 Select **Export** from the figure window **File** menu. The **Export** dialog box appears.
- 2 Select the format from the list of formats in the **Save as type:** menu. This selects the standard filename extension given to files of that type.
- 3 Enter the name you want to give the file.
- 4 Click **Save**.

Copying a Figure to the Clipboard

On Windows systems, you can also copy a figure to the clipboard and then paste it into another application:

- 1 Select **Copy Options** from the Figure window **Edit** menu. The **Copying Options** page of the **Preferences** dialog box appears.
- 2 Complete the fields on the **Copying Options** page and click **OK**.
- 3 Select **Copy Figure** from the **Edit** menu.

The figure is copied to the Windows clipboard. You can then paste the figure from the Windows clipboard into a file in another application.

Printing Figures

Before printing a figure:

- 1 Select **Page Setup** from the figure window **File** menu to set printing options.

The **Page Setup** dialog box opens.

- 2** Make changes in the dialog box. If you want the printed output to exactly match the annotated plot you see on the screen:
 - a** On the **Axes and Figure** tab, click **Keep screen limits and ticks**.
 - b** On the **Size and Position** tab, click **Use screen size, centered on page**.

For more information about other options for page setup, click the **Help** button in the dialog box.

To print a figure, select **Print** from the figure window **File** menu and complete the **Print** dialog box that appears.

You can also use the print command.

Getting Help

To access help for the plot editing mode, select **Plot Editing** from the figure window **Help** menu. You can also access help at the command line by typing `help plottedit` or `doc plottedit`. For information about other graphics features, select **Graphics** from the figure window **Help** menu.

Changing Your View of an Axes

MATLAB lets you change your view of axes by either zooming in for a closer look at a portion of an axes or, for 3-D axes, rotating the axes:

- “Zooming In and Out on an Axes” on page 1-13
- “Rotating 3-D Axes” on page 1-14

The zoom and rotate 3-D options provides basic view changing capabilities. For more advanced viewing, select the Camera option from the Tools menu. See for more information.

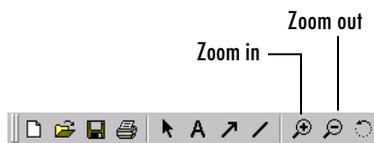
Note Activating any of the zoom, rotate, or camera functions automatically turns off plot editing mode.

Zooming In and Out on an Axes

The zoom in function lets you get a closer view of a portion of an axes. The zoom out function lets you view a larger portion of an axes.

To zoom in on a portion of axes:

- 1 Activate the zoom function by choosing the **Zoom In** option on the **Tools** menu or by clicking the Zoom In button in the toolbar.



- 2 Click on the area of the axes where you want to zoom in, or drag the cursor to draw a box around the area where you want to zoom in.

MATLAB redraws the axes, zooming in on the area you specified. To further magnify the area, click repeatedly in the axes or draw additional boxes in the axes. To end zoom in mode, click the Zoom In button.

Note Properties of the axes, such as the tick marks, automatically adjust to the new magnification or orientation. Annotations do not.

To zoom out from an axes, click the Zoom Out button on the toolbar and then click on an area of the axes where you want to zoom out. To end zoom out mode, click the Zoom Out button.

The zoom buttons use the `camzoom` command.

Rotating 3-D Axes

To rotate a 3-D axes:

- 1 Activate the rotate 3-D function by choosing the **Rotate 3-D** option on the **Tools** menu or by clicking the Rotate 3-D button in the toolbar.



Note Activating the rotate 3-D function automatically turns off plot editing mode.

- 2 Click on the axes and an outline of the figure appears in the axes to help you visualize the rotation. Drag the cursor in the direction you want to rotate. When you release the mouse button, MATLAB redraws the axes in the new orientation.
- 3 Click the rotate button again to end rotate 3-D mode.

The Rotate 3-D button uses the `rotate3d` command.

Using the Property Editor

The Property Editor provides access to many properties of graphics objects, including figures, axes, lines, lights, patches, images, surfaces, rectangles, text, and the root object.

Use these buttons to move back and forth among the graphics objects you have edited.

Use the navigation bar to select the object you want to

Click on a tab to view a group of properties.

Click here to view a list of values for this field.

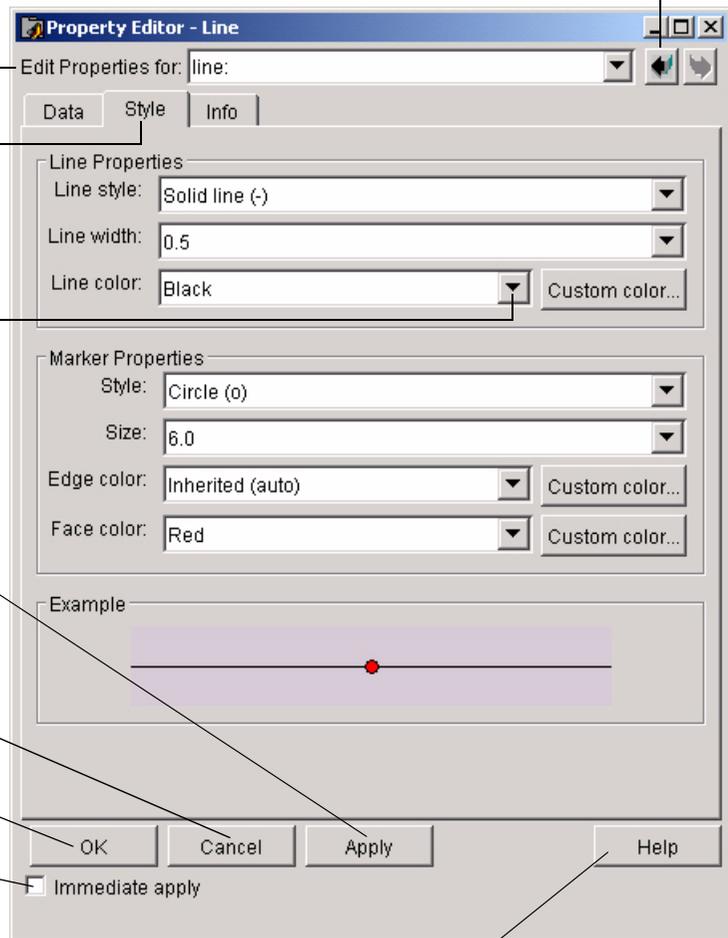
Click Apply to apply your changes without dismissing the Property Editor.

Click Cancel to dismiss the Property Editor without applying your changes.

Click OK to apply your changes and dismiss the Property Editor.

Check this box to see the effect of your changes as you make them.

Click Help to get information about particular properties.



This section describes:

- Starting the Property Editor
- “Editing Object Properties” on page 1-17
- “Navigating Among Objects in a Graph” on page 1-18
- “Applying Your Changes” on page 1-21

Starting the Property Editor

There are several ways to start the Property Editor.

If plot editing mode is enabled, you can:

- Double-click on an object in the graph.

Note Double-clicking on a text object does not start the Property Editor. It opens a edit box around the text. To modify the properties of a text object, use one of the other mechanisms.

- Right-click on an object, view the context menu, and select the **Properties** option. The context menus associated with objects also provide direct access to certain commonly used properties.
- Select **Figure Properties**, **Axes Properties**, or **Current Object Properties** from the figure window **Edit** menu. These options automatically enable plot editing mode, if it is not already enabled.

You can also start the Property Editor from the command line using the `propedit` function.

Note Once you start the Property Editor, keep it open throughout an editing session. If you click on another object in the graph, the Property Editor displays the set of panels associated with that object type. You can also use the navigation bar in the Property Editor to select other objects to edit in the graph.

Closing the Property Editor

Once activated, the Property Editor remains on your screen until you explicitly dismiss it by clicking either the **Cancel** button or the **OK** button. The **Cancel** button dismisses the Property Editor without applying any changes that may have been made to property values that haven't been applied. The **OK** button dismisses the Property Editor dialog box and applies any changes that have been made to property values.

Editing Object Properties

To edit the properties of an object:

- 1 Start plot editing mode.
- 2 Start the Property Editor by double-clicking on the object in the graph or using one of the other mechanisms.

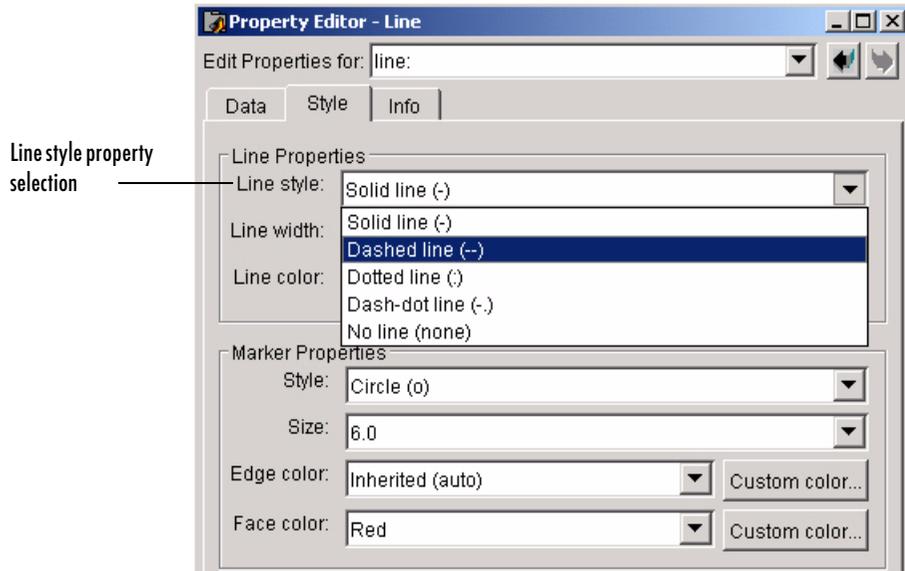
The Property Editor displays the set of panels associated with the object you have selected.

- 3 Click on the tab of the panel that contains the property you want to modify.
- 4 Change the value of the property. For some properties, you must select a value from a menu of values. For other fields, you can either select a value or type a value directly into the field.

Note If you place the cursor over a field, a data tip appears that displays the name of the property being edited and its current value.

- 5 Click the **Apply** button.

For example, if you double-click on a line object in a graph, the Property Editor displays the set of tabbed panels specific to line objects: **Data**, **Style**, and **Info**. To change the style of a line from solid to dashed, click on the Style tab to view the style panel and click on the **Line style** menu. Select **Dashed line** from the list of styles.



Editing Multiple Objects

If you select multiple objects of the same type, the Property Editor displays the set of panels specific to that object type. For example, if you select several lines in a graph, the Property Editor displays the panels associated with line objects. If you change the value of a line property and apply your change, it affects all the objects you have selected.

If you select multiple objects of different types, for example, a line and an axes, the Property Editor displays only the **Info** panel, which provides access to properties that are common to all object types.

For information about navigating among multiple selections, see “Multiple Selections and the Navigation Bar” on page 1-20.

Navigating Among Objects in a Graph

The navigation bar at the top of the Property Editor, labeled **Edit Properties for**, identifies the object being edited by its type (class) and tag, if the object’s tag property has a value. (A tag is a user-defined text string associated with an object.)

You can also use the navigation bar to select other objects, or groups of objects, in the graph that you want to edit. The following sections describe how to use the navigation bar:

- “Selecting Objects from the Navigation Bar” on page 1-19
- “Using the Navigation Bar To Search for Objects” on page 1-20
- “Multiple Selections and the Navigation Bar” on page 1-20

Selecting Objects from the Navigation Bar

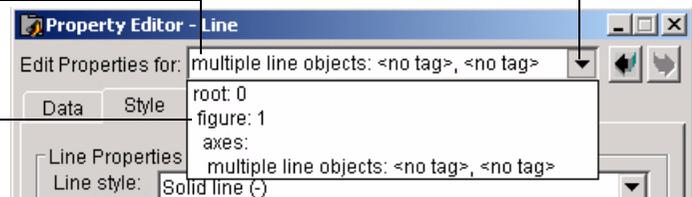
To edit one of the other objects in the graph:

- 1 Click on the navigation bar menu. The Property Editor displays a hierarchical list of all the objects in the current figure. (The Property Editor includes other figures that may be open in the list but does not include the child objects of these figures.)

Click here to display the navigation bar

Navigation bar identifies object being

Navigation bar menu lists all the objects in the current figure.



- 2 Select the object you want to edit from this list. The Property Editor displays the set of panels associated with the type of object you have selected. See “Identifying Objects in a Graph” on page 1-20 for more information.

Note Only objects that have their `HandleVisibility` property set to `On` appear in the navigation bar hierarchical list. However, objects appear in the navigation bar even if their `Visibility` property is set to `Off`.

Using the Navigation Bar To Search for Objects

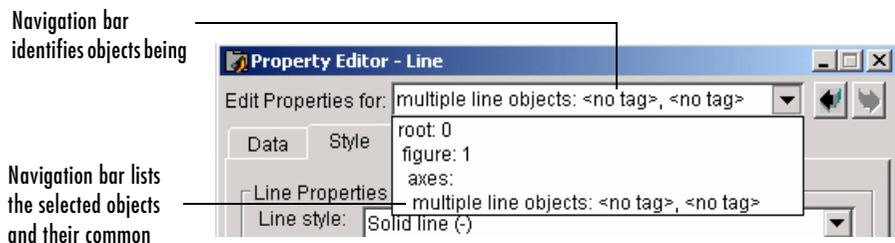
You can use the navigation bar to search for a particular object, or group of objects, in a figure by a tag, type (class), or handle.

For example, to edit every line in a figure, enter the text string `line` in the navigation bar. The Property Editor displays the set of property panels associated with line objects and lists all the lines in the navigation bar. Individual line objects are identified by their tags, if present.

Multiple Selections and the Navigation Bar

When you select multiple objects, the Property Editor's navigation bar displays the objects' ancestors and children if the objects share a common parent. The Property Editor displays all the children in a single, non-hierarchical list. If you select objects that do not have a common parent, the Property Editor navigation bar only displays the selected objects.

For example, if you create a graph containing multiple line plots and you select several of the lines, the Property Editor's navigation bar display looks like this.



Identifying Objects in a Graph

In the hierarchical display of the navigation bar, the Property Editor lists all the objects in a graph by their type and tag, if the object has a tag.

If a graph contains numerous line objects, tags can help identify which line object listed in the navigation bar list represents which line in the graph.

To create a tag for a particular object in a graph:

- 1 Double-click on the object in the graph. Plot editing mode must be enabled.
- 2 Click on the **Info** tab in the Property Editor.

3 Enter a text string in the **Tag** field that identifies the object in the graph.

4 Click **Apply**.

The Property Editor immediately updates the navigation bar display with the tag value.

Applying Your Changes

To apply your changes, click the **Apply** button.

If you have checked the **Immediate Apply** box, your changes will appear automatically; you do not need to click on the **Apply** button.

If you make changes to fields on a panel and then attempt to switch panels without applying your changes, the Property Editor displays a warning message, asking you if you want to apply your changes before moving.

If you click **OK**, you apply your changes and dismiss the Property Editor.

Note During an editing session, keep the Property Editor open. Throughout the session, you can edit the properties of any object in your graph without restarting the Property Editor.

Canceling Changes

If you have changed the values of properties and decide not to apply the changes, click on the **Revert** button. The **Revert** button resets all the properties to their values at the last **Apply**.

To reset a property value change and close the Property Editor, click the **Cancel** button.

Using the Data Statistics Tool

The Data Statistics tool:

- Calculates basic statistics about the central tendency and variability of data plotted in a graph
- Plots any of the statistics in a graph.

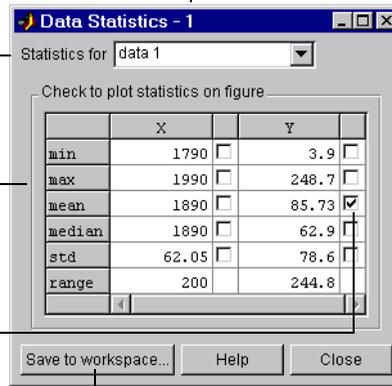
When you select **Data Statistics** from the MATLAB figure window **Tools** menu, MATLAB calculates the statistics for each data set plotted in the graph and displays the results in the **Data Statistics** dialog box.

Identifies the figure in which the data is

Identifies the data set for which statistics have been calculated.

Lists the statistics calculated for both the x- and y-data that define the

To add a plot of a statistic to a graph, click in the check box next to the



Click here to create workspace variables of the statistics.

Adding Plots of Statistics to a Graph

To plot a statistic in a graph, click in the check box next to its value. “Adding Plots of Data Statistics to a Graph” in the “Formatting Graphs” chapter provides an example of using the Data Statistics tool.

Saving Statistics to the Workspace

To save a set of statistics as a workspace variable, click on the **Save to workspace...** button. The Data Statistics tool saves the statistics as a structure.

Basic Plotting

Basic Plotting Commands (p. 2-2)	Basic commands for creating line plots, specifying line styles, colors, and markers, and setting defaults.
Line Plots of Matrix Data (p. 2-13)	Line plots of the rows or column of matrices.
Plotting Imaginary and Complex Data (p. 2-15)	How the plot command handles complex data as a special case.
Plotting with Two Y-Axes (p. 2-16)	Creating line plots that have left and right y -axes.
Setting Axis Parameters (p. 2-19)	Specifying axis ticks location, tick labels, and axes aspect ratio.
Figure Windows (p. 2-26)	Displaying multiple plots per figure, targeting a specific axes, figure color schemes.

Basic Plotting Commands

MATLAB provides a variety of functions for displaying vector data as line plots, as well as functions for annotating and printing these graphs. The following table summarizes the functions that produce basic line plots. These functions differ in the way they scale the plot's axes. Each accepts input in the form of vectors or matrices and automatically scales the axes to accommodate the data.

Function	Description
plot	Graph 2-D data with linear scales for both axes
plot3	Graph 3-D data with linear scales for both axes
loglog	Graph with logarithmic scales for both axes
semilogx	Graph with a logarithmic scale for the x -axis and a linear scale for the y -axis
semilogy	Graph with a logarithmic scale for the y -axis and a linear scale for the x -axis
plotyy	Graph with y -tick labels on the left and right side

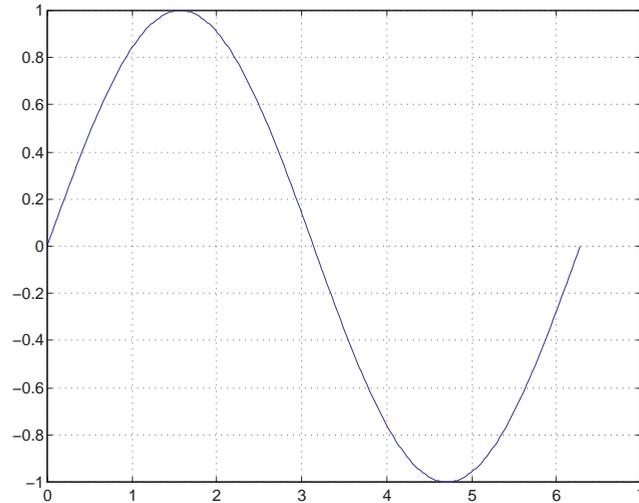
Creating Line Plots

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

For example, 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.

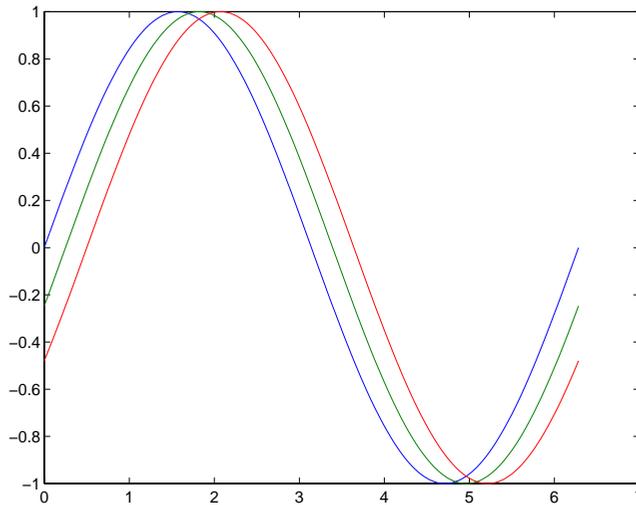
```
t = 0:pi/100:2*pi;  
y = sin(t);  
plot(t,y)  
grid on
```

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

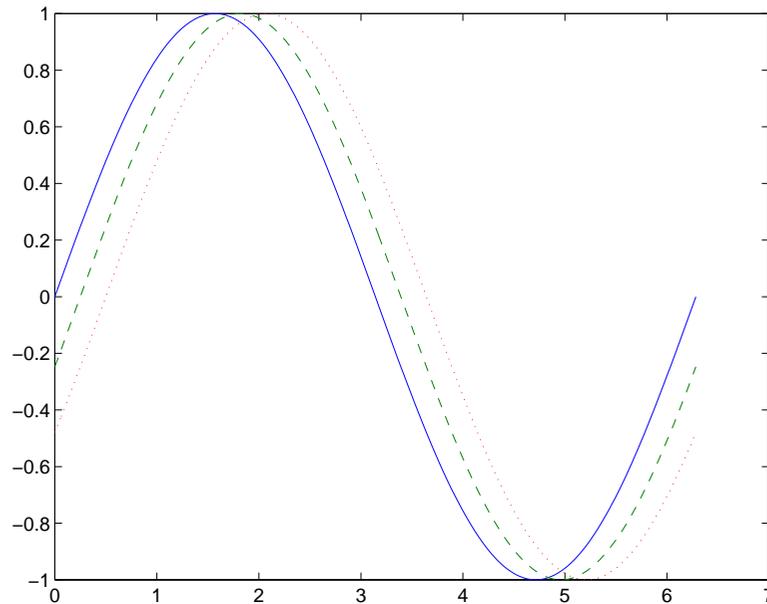
```
y2 = sin(t-0.25);  
y3 = sin(t-0.5);  
plot(t,y,t,y2,t,y3)
```



Specifying Line Style

You can assign different line styles to each data set by passing line style identifier strings to `plot`. For example,

```
t = 0:pi/100:2*pi;  
y = sin(t);  
y2 = sin(t-0.25);  
y3 = sin(t-0.5);  
plot(t,y,'-',t,y2,'--',t,y3,':')
```



Colors, Line Styles, and Markers

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

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

linestyle_marker_color is a character string (delineated by single quotation marks) constructed from:

- A line style (e.g., dashed, dotted, etc.)
- A marker type (e.g., x, *, o, etc.)
- A predefined color specifier (c, m, y, k, r, g, b, w)

For example,

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

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

The specification can consist of one or none of each specifier in any order. For example, the string,

```
'go--'
```

defines a dashed line with circular markers, both colored green.

You can also specify the size of the marker and, for markers that are closed shapes, you can specify separately the color of the edges and the face.

See the `LineStyleSpec` discussion for more information.

Specifying the Color and Size of Lines

You can control a number of line style characteristics by specifying values for line properties:

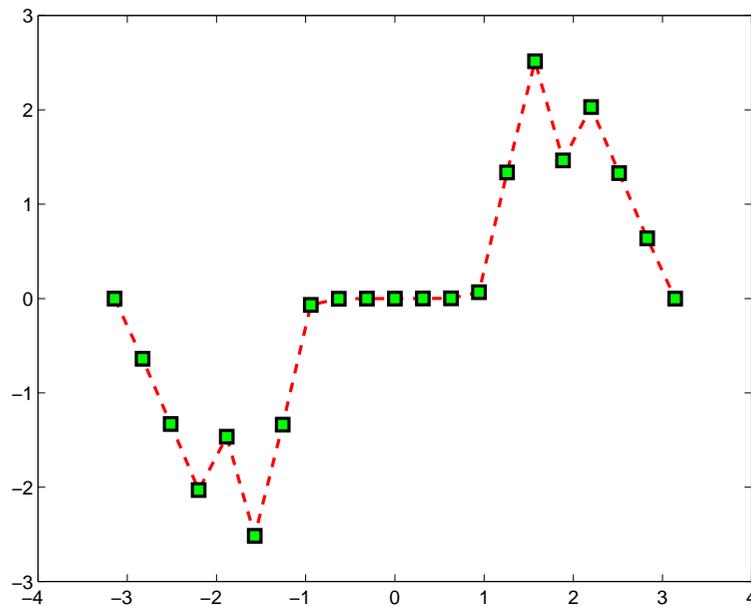
- `LineWidth` – specifies the width of the line in units of points.
- `MarkerEdgeColor` – specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- `MarkerFaceColor` – specifies the color of the face of filled markers.
- `MarkerSize` – specifies the size of the marker in units of points.

For example, these statements,

```
x = -pi:pi/10:pi;  
y = tan(sin(x)) - sin(tan(x));  
plot(x,y,'--rs','LineWidth',2,...  
      'MarkerEdgeColor','k',...  
      'MarkerFaceColor','g',...  
      'MarkerSize',10)
```

produce a graph with:

- A red dashed line with square markers
- A line width of two points
- The edge of the marker colored black
- The face of the marker colored green
- The size of the marker set to 10 points



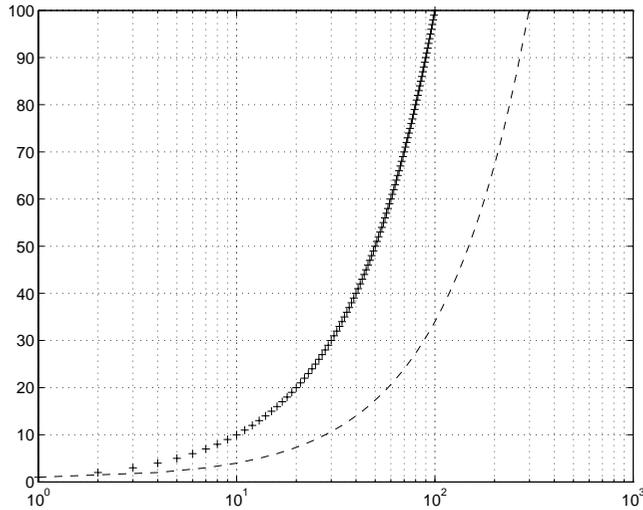
Adding Plots to an Existing Graph

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

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

```
semilogx(1:100, '+')
hold 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.



Plotting Only the Data Points

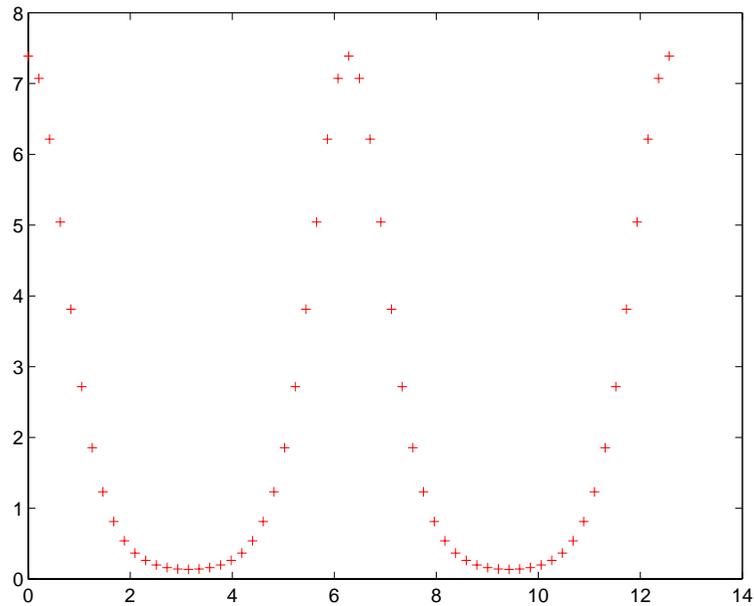
To plot a marker at each data point without connecting the markers with lines, use a specification that does not contain a line style. For example, given two vectors,

```
x = 0:pi/15:4*pi;  
y = exp(2*cos(x));
```

calling plot with only a color and marker specifier

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

plots a red plus sign at each data point.

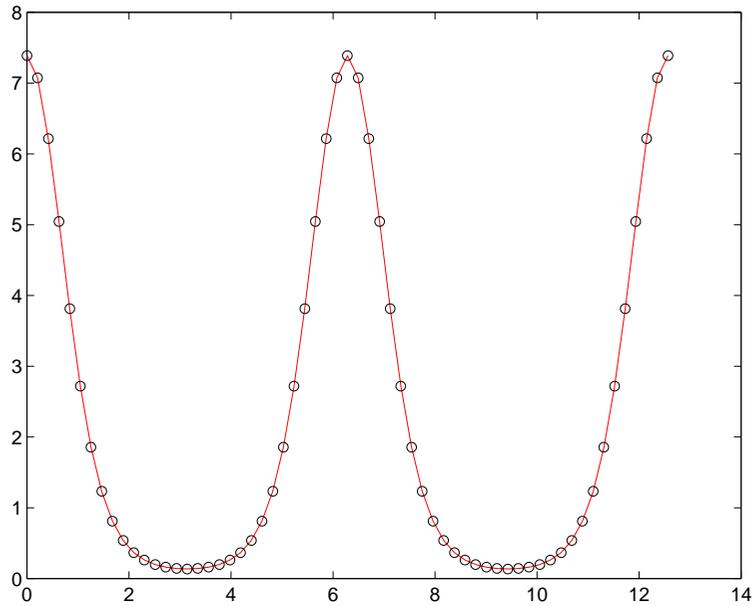


See LineSpec for a list of available line styles, markers, and colors.

Plotting Markers and Lines

To plot both markers and the lines that connect them, specify a line style and a marker type. For example, the following command plots the data as a red, solid line and then adds circular markers with black edges at each data point.

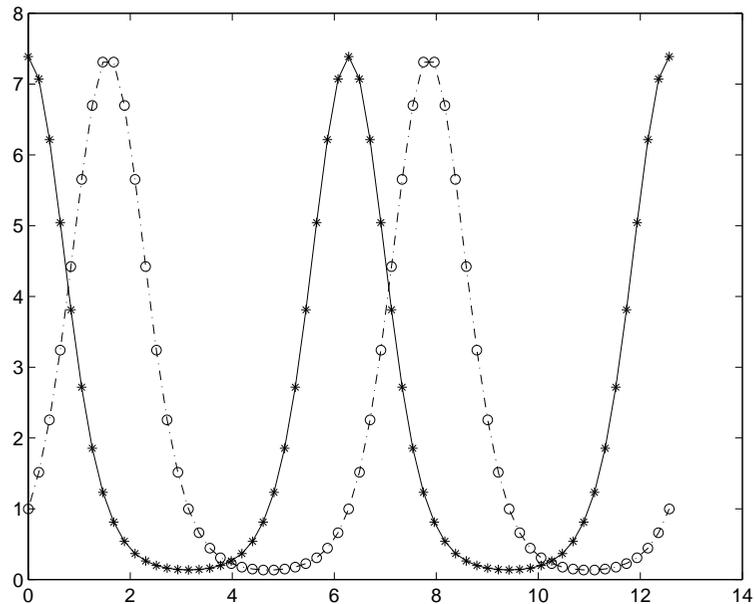
```
x = 0:pi/15:4*pi;  
y = exp(2*cos(x));  
plot(x,y,'-r',x,y,'ok')
```



Line Styles for Black and White Output

Line styles and markers enable you to discriminate different plots on the same graph when color is not available. For example, the following statements create a graph using a solid ('-k') line with asterisk markers colored black and a dash-dot ('-.ok') line with circular markers colored black.

```
x = 0:pi/15:4*pi;  
y1 = exp(2*cos(x));  
y2 = exp(2*sin(x));  
plot(x,y1,'-k',x,y2,'-.ok')
```



Setting Default Line Styles

You can configure MATLAB to use line styles instead of colors for multi-line plots by setting a default value for the axes `LineStyleOrder` property. For example, the command,

```
set(0, 'DefaultAxesLineStyleOrder', {'-o', ':s', '--+'})
```

defines three line styles and makes them the default for all plots.

To set the default line color to dark gray, use the statement

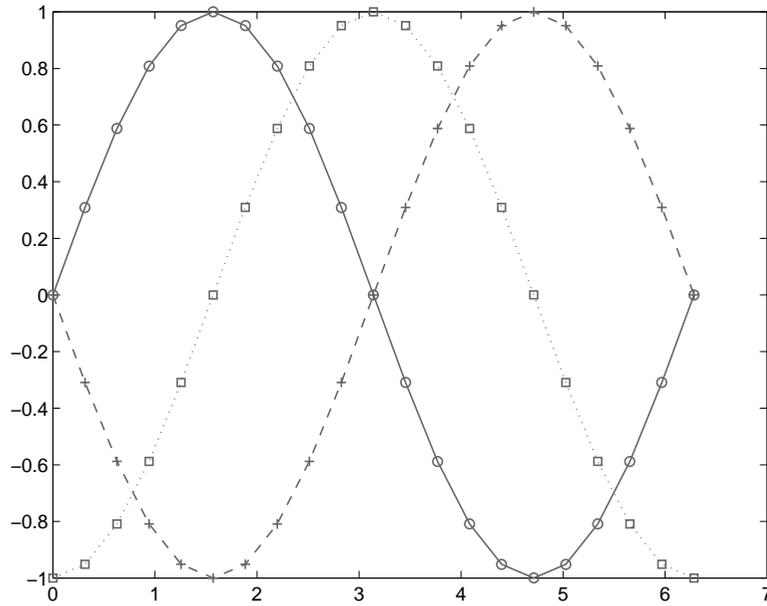
```
set(0, 'DefaultAxesColorOrder', [0.4, 0.4, 0.4])
```

See `ColorSpec` for information on how to specify color as a three-element vector of RGB values.

Now the plot command uses the line styles and colors you have defined as defaults. For example, these statements create a multiline plot.

```
x = 0:pi/10:2*pi;
y1 = sin(x);
```

```
y2 = sin(x-pi/2);  
y3 = sin(x-pi);  
plot(x,y1,x,y2,x,y3)
```



The default values persist until you quit MATLAB. To remove default values during your MATLAB session, use the reserved word `remove`.

```
set(0,'DefaultAxesLineStyleOrder','remove')  
set(0,'DefaultAxesColorOrder','remove')
```

See “Setting Default Property Values” in the “Handle Graphics Objects” chapter for more information.

Line Plots of Matrix Data

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

```
plot(Y)
```

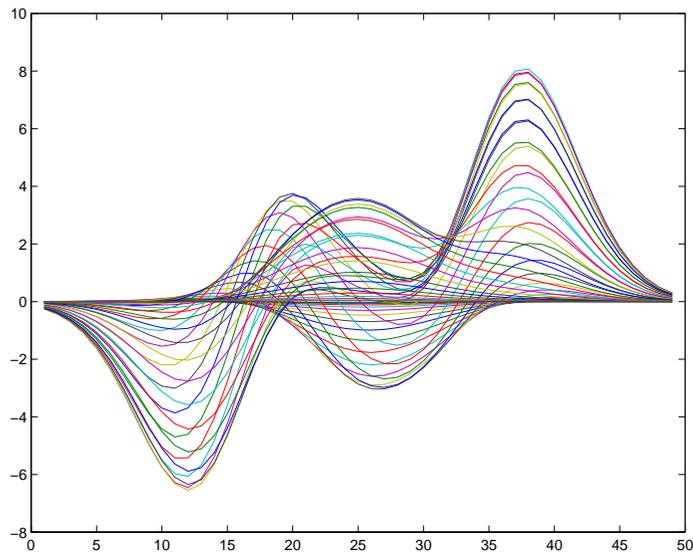
MATLAB draws one line for each column of the matrix. The x -axis is labeled with the row index vector, $1:m$, where m is the number of rows in Y . For example,

```
Z = peaks;
```

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

```
plot(Z)
```

produces a graph with 49 lines.



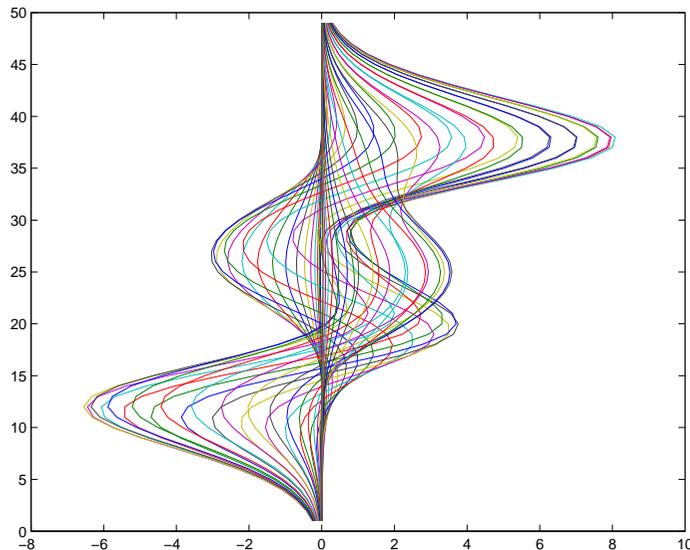
In general, if `plot` is used with two arguments and if either X or Y has more than one row or column, then:

- If Y is a matrix, and x is a vector, `plot(x, Y)` successively plots the rows or columns of Y versus vector x , using different colors or line types for each. The

row or column orientation varies depending on whether the number of elements in x matches the number of rows in Y or the number of columns. If Y is square, its columns are used.

- If X is a matrix and y is a vector, `plot(X,y)` plots each row or column of X versus vector y . For example, plotting the peaks matrix versus the vector `1:length(peaks)` rotates the previous plot.

```
y = 1:length(peaks);  
plot(peaks,y)
```



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

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

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

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

Plotting Imaginary and Complex Data

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

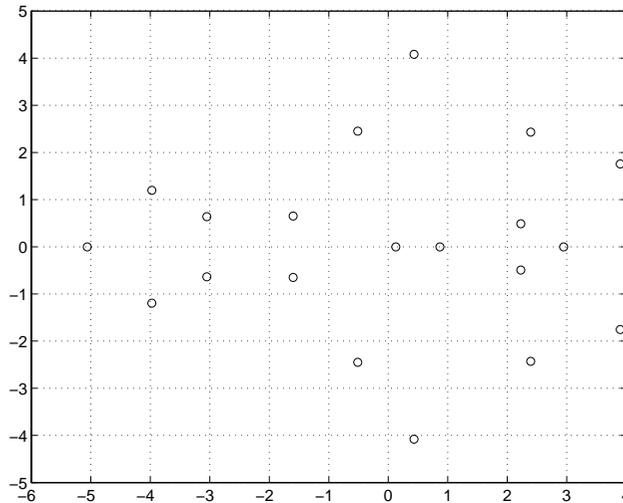
```
plot(Z)
```

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

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

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

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

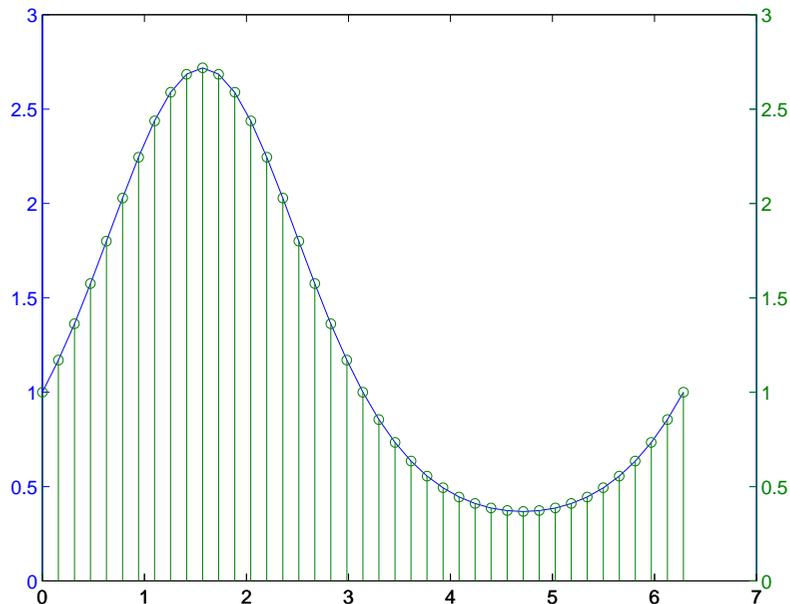


To plot more than one complex matrix, there is no shortcut; the real and imaginary parts must be taken explicitly.

Plotting with Two Y-Axes

The `plotyy` command enables you to create plots of two data sets and use both left and right side y -axes. You can also apply different plotting functions to each data set. For example, you can combine a line plot with a stem plot of the same data.

```
t = 0:pi/20:2*pi;  
y = exp(sin(t));  
plotyy(t,y,t,y,'plot','stem')
```



Combining Linear and Logarithmic Axes

You can use `plotyy` to apply linear and logarithmic scaling to compare two data sets having a different range of values.

```
t = 0:900; A = 1000; a = 0.005; b = 0.005;  
z1 = A*exp(-a*t);  
z2 = sin(b*t);  
[haxes,hline1,hline2] = plotyy(t,z1,t,z2,'semilogy','plot');
```

This example saves the handles of the lines and axes created to adjust and label the graph. First, label the axes whose y value ranges from 10 to 1000. This is the first handle in `haxes` because we specified this plot first in the call to `plotyy`. Use the `axes` command to make `haxes(1)` the current axes, which is then the target for the `ylabel` command.

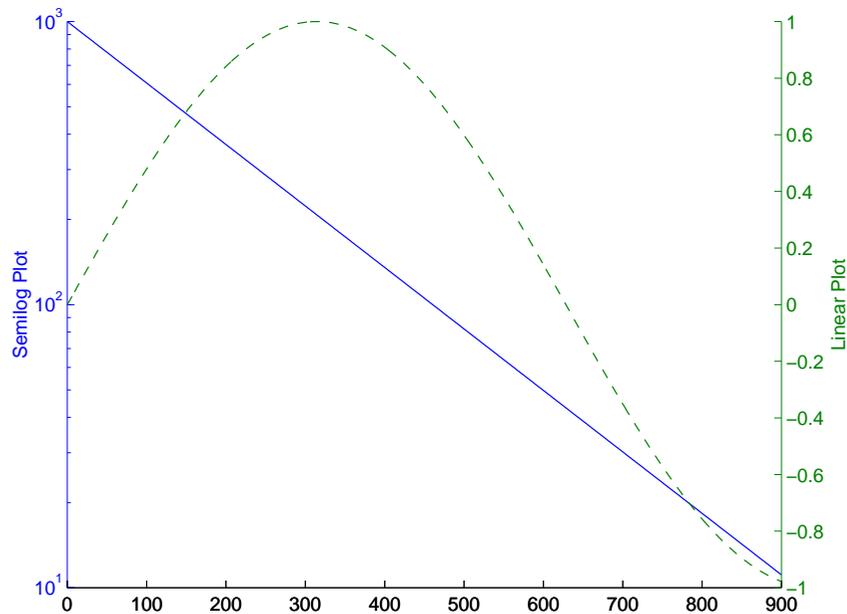
```
axes(haxes(1))  
ylabel('Semilog Plot')
```

Now make the second axes current and call `ylabel` again.

```
axes(haxes(2))  
ylabel('Linear Plot')
```

You can modify the characteristics of the plotted lines in a similar way. For example, to change the line style of the second line plotted to a dashed line, use the statement

```
set(hline2, 'LineStyle', '- -')
```



See “Using Multiple X and Y Axes” in the “Axes Properties” chapter for an example that employs double x - and y -axes.

See `LineStyle` for additional line properties.

Setting Axis Parameters

When you create a graph, MATLAB automatically selects the axis limits and tick-mark spacing based on the data plotted. However, you can also specify your own values for axis limits and tick marks by. You can do this with the following commands:

- `axis` – sets values that affect the current axes object (the most recently created or the last clicked on).
- `axes` – (not `axis`) creates a new axes object with the specified characteristics.
- `get` and `set` – enable you to query and set a wide variety of properties of existing axes.
- `gca` – returns the handle (identifier) of the current axes. If there are multiple axes in the figure window, the current axes is the last graph created or the last graph you clicked on with the mouse. The following two sections provide more information and examples:

“Axis Limits and Ticks” on page 2-19

“Example – Specifying Ticks and Tick Labels” on page 2-22

Related Information

See the chapter “Defining the View” for more extensive information on manipulating 3-D views.

Axis Limits and Ticks

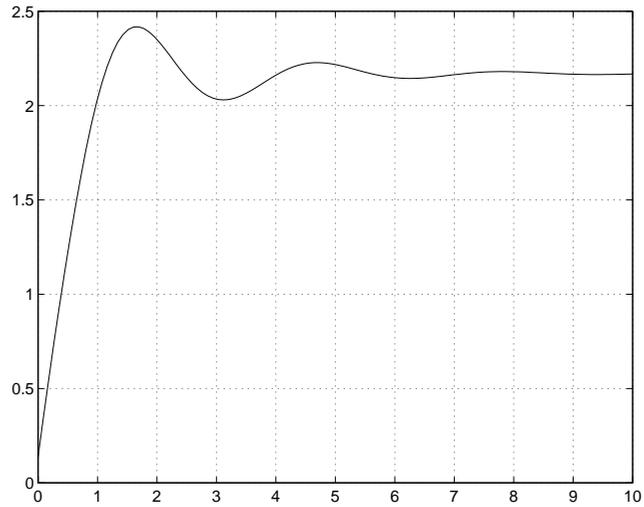
MATLAB selects axis limits based on the range of the plotted data. You can specify the limits manually using the `axis` command. Call `axis` with the new limits defined as a four-element vector.

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

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

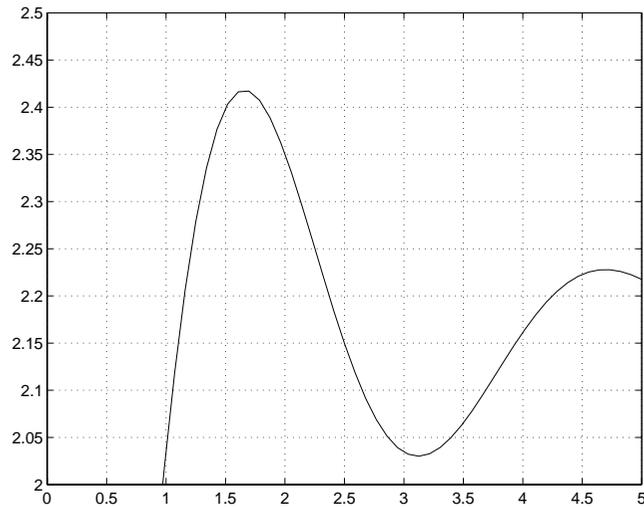
Semiautomatic Limits

If you want MATLAB to autoscale only one of a min/max set of axis limits, but you want to specify the other, use the MATLAB variable `Inf` or `-Inf` for the autoscaled limit. For example, this graph uses default scaling.



Compare the default limits to the following graph, which sets the maximum limit of the x -axis, but autoscales the minimum limit.

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



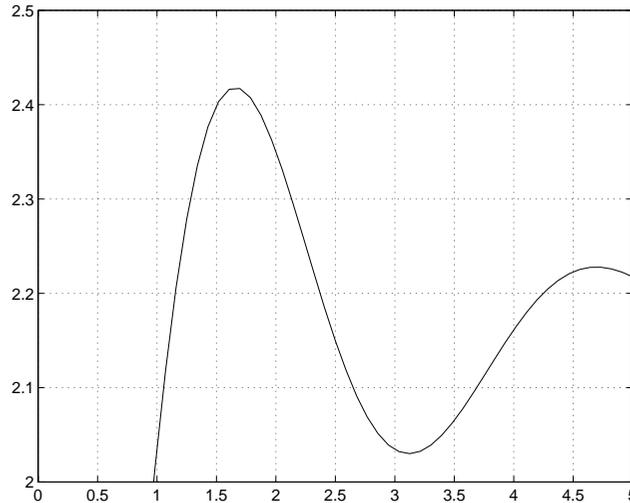
Axis Tick Marks

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

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

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

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



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

Example – Specifying Ticks and Tick Labels

You can adjust the axis tick-mark locations and the labels appearing at each tick mark. For example, this plot of the sine function relabels the x-axis with more meaningful values.

```
x = -pi:.1:pi;  
y = sin(x);  
plot(x,y)  
set(gca,'XTick',-pi:pi/2:pi)  
set(gca,'XTickLabel',{'-pi','-pi/2','0','pi/2','pi'})
```

These commands (`xlabel`, `ylabel`, `title`, `text`) add axis labels and draw an arrow that points to the location on the graph where $y = \sin(-\pi/4)$.

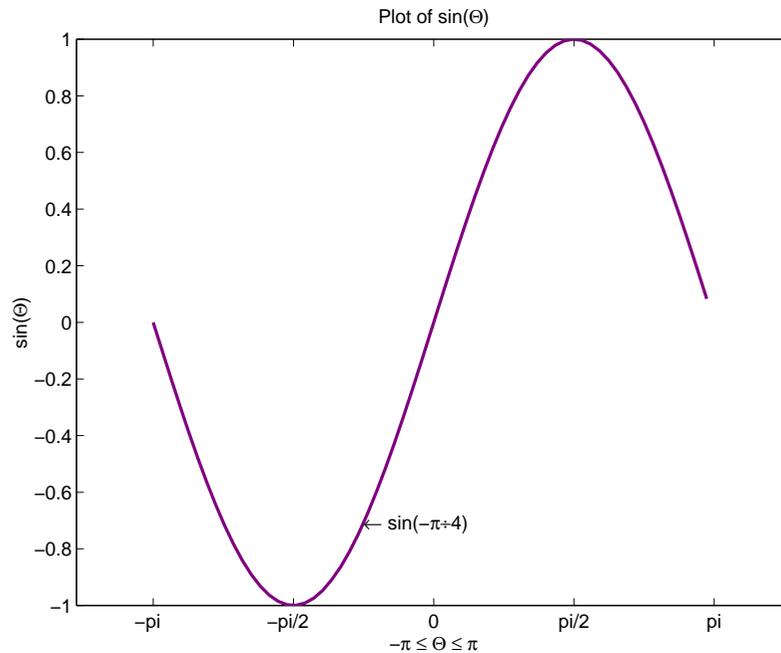
```
xlabel('-\pi \leq \Theta \leq \pi')  
ylabel('sin(\Theta)')  
title('Plot of sin(\Theta)')
```

```
text(-pi/4,sin(-pi/4),'\leftarrow sin(-\pi\div4)',...
     'HorizontalAlignment','left')
```

Setting Line Properties on an Existing Plot

Change the line color to purple by first finding the handle of the line object created by `plot` and then setting its `Color` property. Use `findobj` and the fact that MATLAB creates a blue line (RGB value `[0 0 1]`) by default. In the same statement, set the `LineWidth` property to 2 points.

```
set(findobj(gca,'Type','line','Color',[0 0 1]),...
    'Color',[0.5,0,0.5],'LineWidth',2)
```



The Greek symbols are created using TeX character sequences.

Setting Aspect Ratio

By default, MATLAB displays graphs in a rectangular axes that has the same aspect ratio as the figure window. This makes optimum use of space available

for plotting. MATLAB provides control over the aspect ratio with the `axis` command.

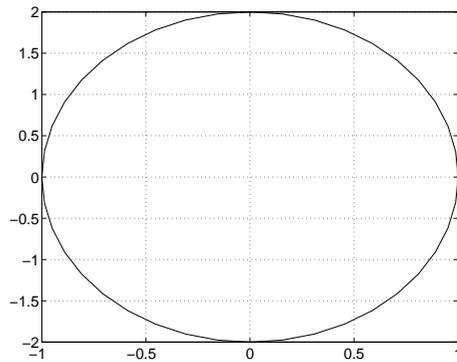
For example,

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

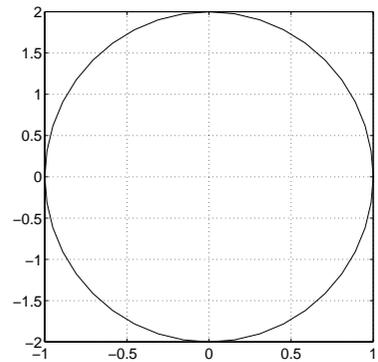
produces a graph with the default aspect ratio. The command

```
axis square
```

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



axis normal

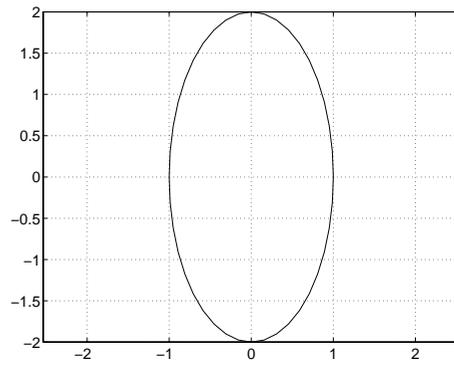


axis square

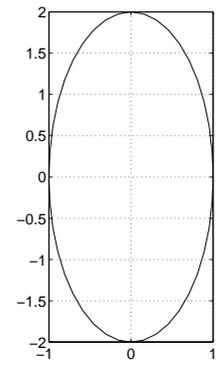
The square axes has one data unit in x to equal two data units in y . If you want the x - and y -data units to be equal, use the command

```
axis equal
```

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



axis equal



axis equal tight

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

```
axis equal tight
```

Figure Windows

MATLAB directs graphics output to a window that is separate from the command window. In MATLAB this window is referred to as a *figure*. The characteristics of this window are controlled by your computer's windowing system and MATLAB figure properties (see a description of each property). See "Figure Properties" on page 8-1 for some examples illustrating how to use figure properties.

Graphics functions automatically create new figure windows if none currently exist. If a figure already exists, MATLAB uses that window. If multiple figures exist, one is designated as the *current figure* and is used by MATLAB (this is generally the last figure used or the last figure you clicked the mouse in).

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

```
figure
```

creates a new window and makes it the current figure. You can make an existing figure current by clicking on it with the mouse or by passing its handle (the number indicated in the window title bar), as an argument to `figure`.

```
figure(h)
```

Displaying Multiple Plots per Figure

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

`subplot(m,n,i)` breaks the figure window into an m -by- n matrix of small subplots and selects the i 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.

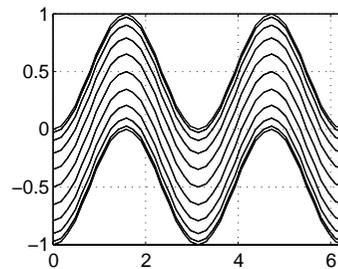
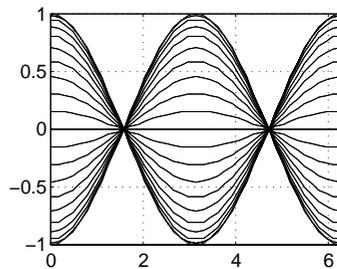
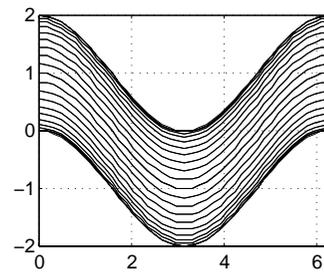
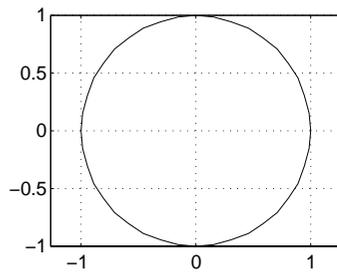
```
t = 0:pi/20:2*pi;
[x,y] = meshgrid(t);

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 for more information.

Specifying the Target Axes

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

You can make an axes current in three ways:

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

For example,

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

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

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

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

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

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

Default Color Scheme

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

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

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

You can examine the `colordef.m` M-file to determine what properties it sets (enter type `colordef` at the MATLAB prompt).

Formatting Graphs

Overview (p. 3-2)

Summary of the options for formatting graphs.

Adding Titles to Graphs (p. 3-3)

Ways to add a title to a graph.

Adding Legends to Graphs (p. 3-6)

Add, position, modify, and remove legends from graphs.

Adding Axis Labels to Graphs (p. 3-11)

Various ways to add labels to graphs.

Adding Text Annotations to Graphs
(p. 3-17)

Techniques for adding text to graphs, including alignment, symbols and Greek letters, using variables in text strings, multiline text, and text background color.

Adding Arrows and Lines to Graphs
(p. 3-31)

Adding callout arrows and lines to graphs.

Adding Plots of Basic Statistics to
Graphs (p. 3-33)

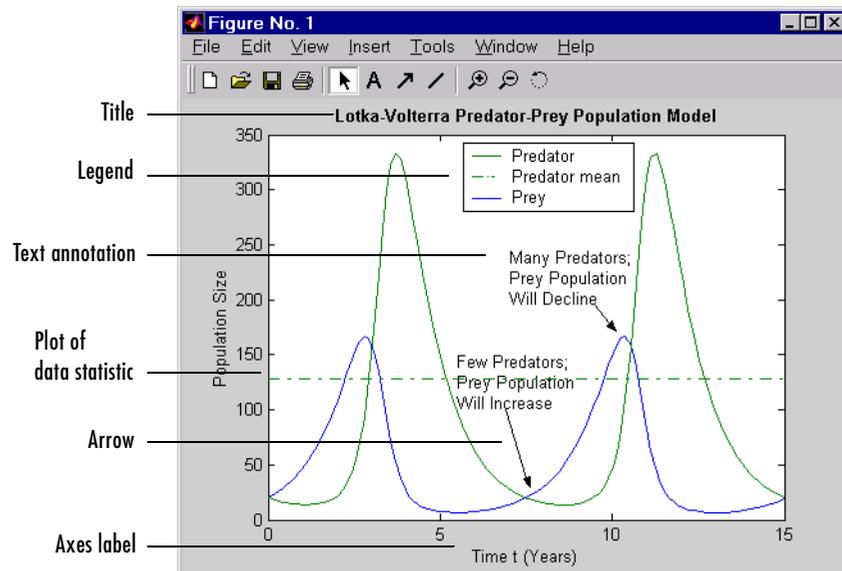
Data Statistics Tool enables you to plot statistical data about your graph.

Overview

When creating presentation graphics, you may want to add labels and annotations to your graph to help explain your data. MATLAB provides mechanisms that let you:

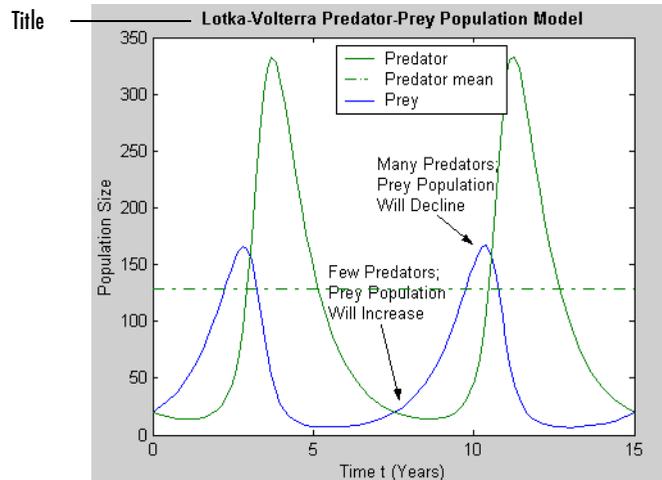
- Add a title at the top of an axes
- Add a legend
- Add labels to axes
- Add free-form text annotations anywhere on your graph
- Add arrows and lines anywhere on your graph
- Add plots of basic data statistics, such as the maximum, minimum, and mean

The following figure shows a graph that uses all of these labels and annotations. Click on any of the labels and annotations in this figure to get more information about how to create the label or annotation.



Adding Titles to Graphs

In MATLAB, a title is a text string at the top of an axes. Titles typically define the subject of the graph.



There are several ways to add a title to a graph:

- “Using the Title Option on the Insert Menu” on page 3-3
- “Using the Property Editor to Add a Title” on page 3-4
- “Using the title Function” on page 3-5

Note While you can use text annotations to create a title for your graph, it is not recommended. Titles are anchored to the top of the axes they describe; text annotations are not. If you move or resize your axes, the title remains at the top. Additionally, if you cut a title and then paste it back into a figure, the title will no longer be anchored to the axes.

Using the Title Option on the Insert Menu

To add a title to a graph using the **Insert** menu:

- 1 Click the **Insert** menu in the Figure window menu bar and choose **Title**. MATLAB opens a text entry box at the top of the axes.

Note When you select the **Title** option, MATLAB enables plot editing mode automatically.

- 2 Enter the text of the label.
- 3 When you are finished entering text, click anywhere in the figure background to close the text entry box around the title. If you click on another object in the figure, such as an axes or line, you close the title text entry box and also automatically select the object you clicked on.

To change the font used in the title to bold, you must edit the title. You can edit the title as you would any other text object in a graph. See “Editing Text Annotations” on page 3-24 for more information.

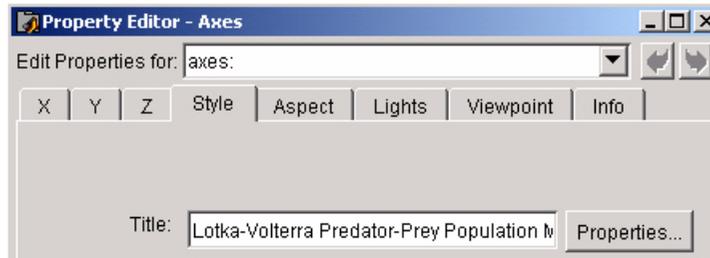
Using the Property Editor to Add a Title

To add a title to a graph using the Property Editor:

- 1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu
- 2 Double-click on the axes in the graph. This starts the Property Editor. You can also start the Property Editor by right-clicking on the axes and selecting **Properties** from the context menu.

The Property Editor displays the set of property panels specific to axes objects. Titles are a property of axes objects.

- 3 Select the **Style** panel and type in the text of your title in the **Title** text entry box.



4 Click **Apply**.

The title you create is a text object and, as such, you can change the font, font style, position and many other aspects of its format. To view the properties associated with a text object, click the **Edit** button next to the Title text entry box. For more information about text object properties, see “Editing Text Annotations” on page 3-24.

Using the title Function

To add a title to a graph at the MATLAB command prompt or from an M-file, use the `title` function. The `title` function lets you specify the value of title properties at the time you create it.

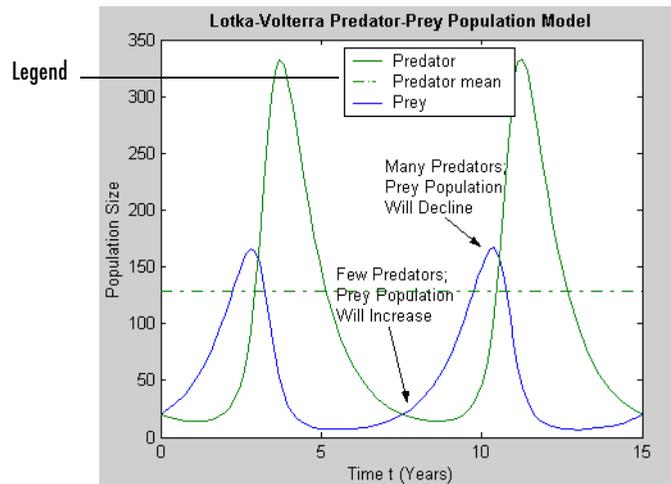
For example, the following code adds a title to the current axes and sets the value of the `FontWeight` property to bold.

```
title('Lotka-Volterra Predator-Prey Population Model',...
      'FontWeight','bold')
```

To edit a title from the MATLAB command prompt or from an M-file, use the `set` function. See “Setting and Querying Object Properties” in the “Handle Graphics Objects” chapter for more information.

Adding Legends to Graphs

Legends identify each data set plotted in your graph. In a legend, MATLAB includes a small sample of the line or marker used to represent each data set in the graph, in the same color and style as it appears in the graph. MATLAB also includes a text label to identify each data set.



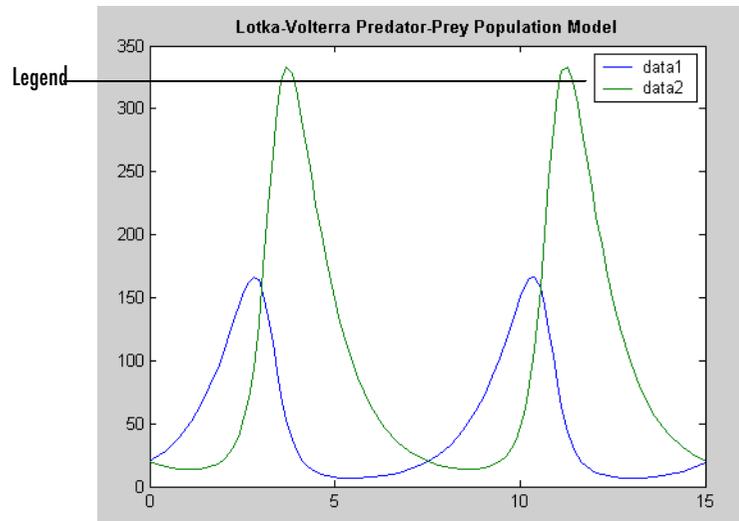
This section includes these topics:

- “Using the Legend Option on the Insert Menu” on page 3-7
- “Using the Legend Function” on page 3-7
- “Positioning the Legend” on page 3-8
- “Editing the Legend” on page 3-9
- “Removing the Legend” on page 3-10

Note The legend is implemented as a separate axes overlaying the axes it describes. The legend axes is not anchored to the main axes. If you resize or move the main axes, or create new subplots, you may need to reposition the legend.

Using the Legend Option on the Insert Menu

To add a legend to a graph, click on the **Insert** menu and choose **Legend**. MATLAB creates a legend, placing it in the upper right corner of the plot.



MATLAB creates text labels to identify each data set in the graph, using data1 to identify the first data set, data2 to identify the second data set, and so on. To learn how to customize a legend, such as changing its position or changing the text labels, see “Positioning the Legend” on page 3-8 and “Editing the Legend” on page 3-9.

Using the Legend Function

To add a legend to a graph at the MATLAB command prompt or from an M-file, use the legend function. You must specify the text labels when you create a legend using the legend function.

For example, the following code adds a legend to the current axes.

```
legend('Y1 Predator', 'Y2 Prey')
```

The legend function lets you specify many other aspects of the legend, such as its position. For more information, see the legend function reference information.

Positioning the Legend

There are two ways to change the position of a legend in a graph, depending on whether plot editing mode is enabled.

If Plot Editing Mode Is Not Enabled

- 1 Move the mouse over the legend and press and hold down the left mouse button. MATLAB changes the cursor to the indicate possible directions of movement.
- 2 With the mouse button still pressed, move the legend anywhere in the graph.
- 3 Release the mouse button.

If Plot Editing Mode Is Enabled

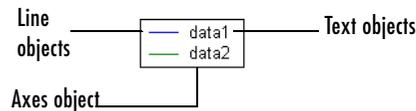
- 1 Right-click on the legend. This selects the legend and triggers the display of the context menu for the legend.
- 2 Select **Unlock Axes Position** from the context menu. (The legend is an axes object.)

Note If the context menu does not include the **Unlock Axes Position** option, you probably selected the text labels in the legend or the line objects and not the legend axes. Every object in a legend is individually selectable. Make sure you have selected the legend axes.

- 3 Move the cursor back over the legend axes (it should still be selected) and press and hold down either mouse button. MATLAB changes the cursor to indicate possible directions of movement.
- 4 Move the legend anywhere in the graph.
- 5 Release the mouse button.

Editing the Legend

A legend is implemented as a separate axes object containing one or more line objects, representing samples of the plots in the graph, and one or more text objects, representing the labels for each data set plotted in a graph.



You can edit a legend when plot editing mode is enabled or when it is not enabled.

Editing a Legend in Plot Editing Mode

When you enable plot editing mode, you can edit any of the objects that make up a legend as you would any other axes, line, or text object in a graph. For example, if you double-click on the legend axes, the Property Editor displays the set of property panels for axes objects. Change the value of an axes property and click **Apply**.

If you double-click on a text label in a legend, MATLAB opens a text editing box around all the text labels in the legend. You can edit any of the text labels in the legend. To access the properties of these text objects, right-click on a text label and select **Properties** from the context-sensitive pop-up menu.

Editing a Legend When Plot Editing Mode Is Not Enabled

When plot editing mode is not enabled, you can still edit the text labels in a legend:

- 1 Double-click on a text label in the legend.

MATLAB opens a text edit box around the text label you selected. All the other text labels are temporarily hidden. You can only edit one text label at a time.

- 2 Make changes to the text label and then click anywhere in the figure outside of the text edit box when you are finished.

MATLAB automatically resizes the legend box to fit long or multiline labels.

Resizing a Legend

To resize a legend:

- 1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu
- 2 Right-click on its axes and select **Unlock Axes Position** from the pop-up menu.
- 3 Move the cursor back to the legend axes, which is selected, and grab one of the selection handles. MATLAB changes the cursor to indicate possible directions.

Note If the text labels extend past the legend axes border, you cannot grab the selection handles on the right side of the legend axes. Resize the axes from the left side to fit the new text labels.

- 4 Drag the selection handle to resize the legend.

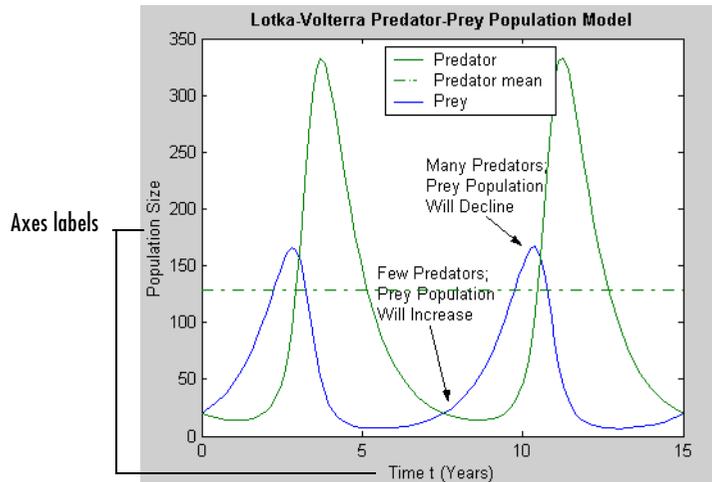
Removing the Legend

If you have enabled plot editing mode, you can remove a legend by clicking on it and choosing the **Cut** option on the **Edit** menu. You can also remove a legend by right-clicking on it and selecting **Cut** from the context-sensitive pop-up menu.

If plot editing mode is not enabled, you can remove a legend by selecting the **Legend** option on the **Insert** menu. The **Legend** options acts as a toggle switch — selecting it alternately adds or removes a legend.

Adding Axis Labels to Graphs

In MATLAB, an axes label is a text string aligned with the x -, y -, or z -axis in a graph. Axis labels can help explain the meaning of the units that each axis represents.



Note While you can use free-form text annotations to create axes labels, it is not recommended. Axis labels are anchored to the axes they describe; text annotations are not. If you move or resize your axes, the labels automatically move with the axes. Additionally, if you cut a label and then paste it back into a figure, the label will no longer be anchored to the axes.

To add axes labels to a graph, you can use any of these mechanisms:

- “Using the Label Options on the Insert Menu” on page 3-12
- “Using the Property Editor to Add Axis labels” on page 3-12
- “Using Axis-Label Commands” on page 3-15

Using the Label Options on the Insert Menu

- 1 Click on the **Insert** menu and choose the label option that corresponds to the axes you want to label: **X Label**, **Y Label**, or **Z Label**. MATLAB opens a text entry box along the axes, or around an existing axes label.

Note MATLAB opens up a horizontal text editing box for the Y- and Z- axes labels and automatically rotates the label into alignment with the axes when you finish entering text.

- 2 Enter the text of the label, or edit the text of an existing label.
- 3 Click anywhere else in the figure background to close the text entry box around the label. If you click on another object in the figure, such as an axes or line, you close the label text entry box but also automatically select the object you clicked on.

Note After you use the **Insert** menu to add an axes label, plot edit mode is enabled in the figure, if it was not already enabled.

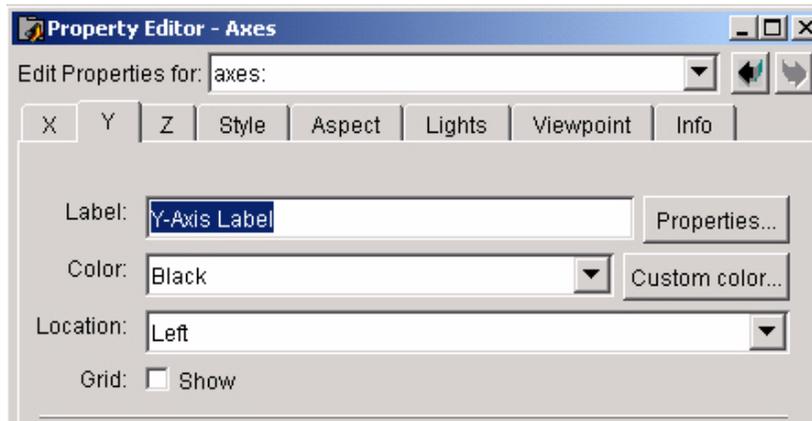
Using the Property Editor to Add Axis labels

To add labels to a graph using the Property Editor:

- 1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu
- 2 Start the Property Editor by either double-clicking on the axes in the graph or by right-clicking on the axes and selecting **Properties** from the context-sensitive pop-up menu.

The Property Editor displays the set of property panels specific to axes objects.

- 3 Select the **X**, **Y**, or **Z** panel, depending on which axis label you want to add. Enter the label text in the **Label** text entry box.

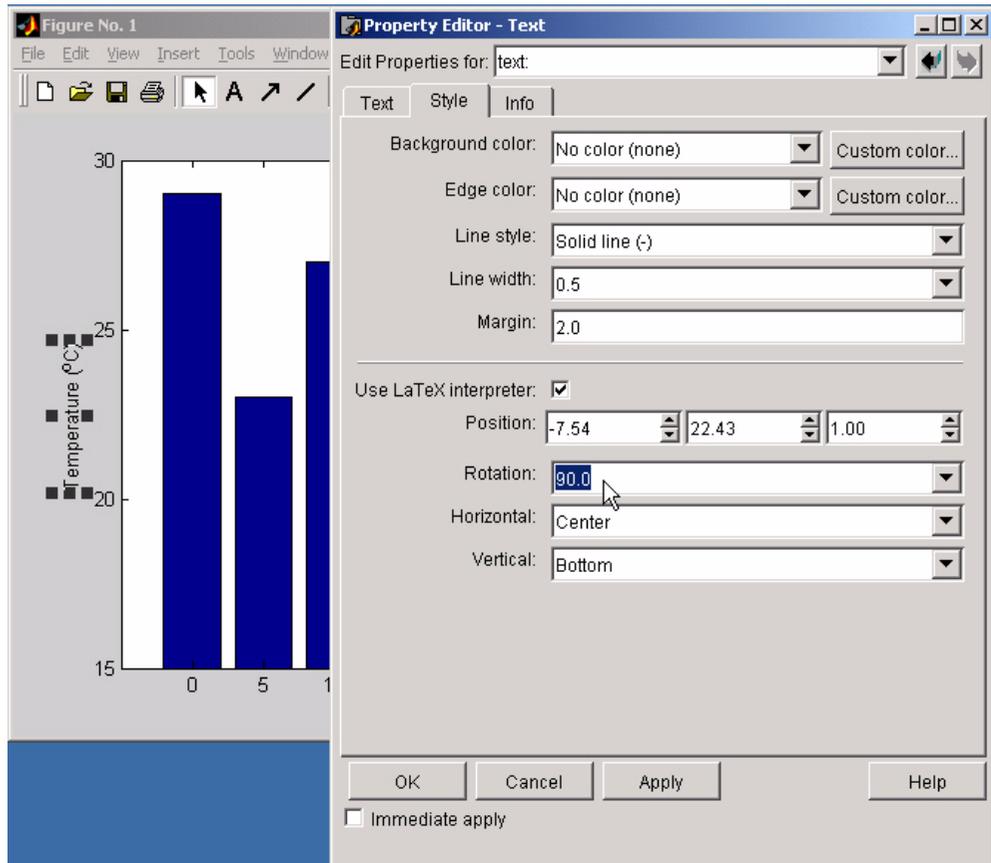


- 4 Click **Apply**.

Rotating Axis Labels

You can rotate axis labels using the Property Editor:

- 1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.
- 2 Display the Property Editor by selecting (left-click) the axis label you want to rotate. Right-click on the selected text then choose **Properties** from the pop-up menu.

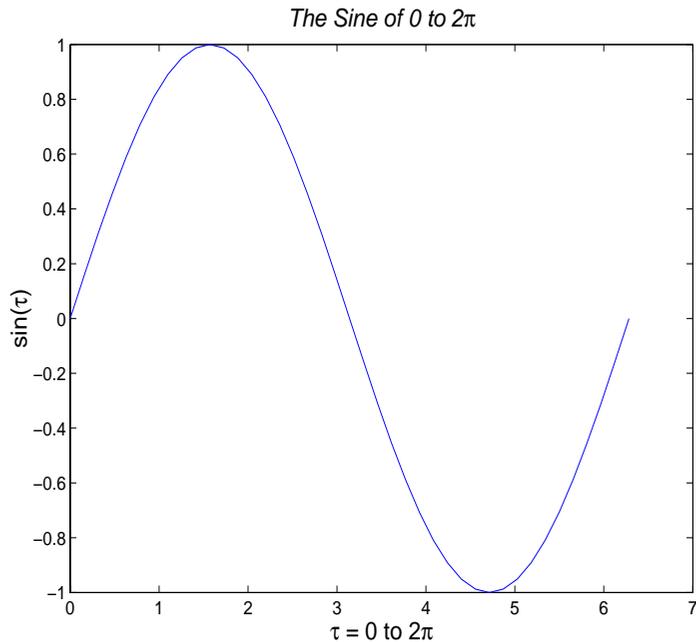


- 3 Select the **Style** panel and enter a value for the orientation in the **Rotation** text field. A value of 0 degrees orients the label in the horizontal position.
- 4 Click **Apply**.
- 5 With the left mouse button down on the selected label, drag the text to the desired location and release.

Using Axis-Label Commands

You can add x -, y -, and z -axis labels using the `xlabel`, `ylabel`, and `zlabel` 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}','FontSize',16)
```



The labeling commands automatically position the text string appropriately. MATLAB interprets the characters immediately following the backslash “\” as TeX commands. These commands draw symbols such as Greek letters and arrows.

See the text `String` property for a list of TeX character sequences. See also the `textlabel` function for converting MATLAB expressions to TeX symbols.

Rotating Axis Labels Using Commands

Axis labels are text objects that you can rotate by specifying a value for the object's `Rotation` property. The handles of the x -, y -, and z -axis labels are stored in the axes `XLabel`, `YLabel`, and `ZLabel` properties respectively.

Therefore, to rotate the y -axis so that the text is horizontal:

- 1 Get the handle of the text object using the axes `YLabel` property:
- 2 Set the `Rotation` property to 0.0 degrees

For example, this statement rotates the text of the y -axis label on the current axes:

```
set(get(gca, 'YLabel'), 'Rotation', 0.0)
```

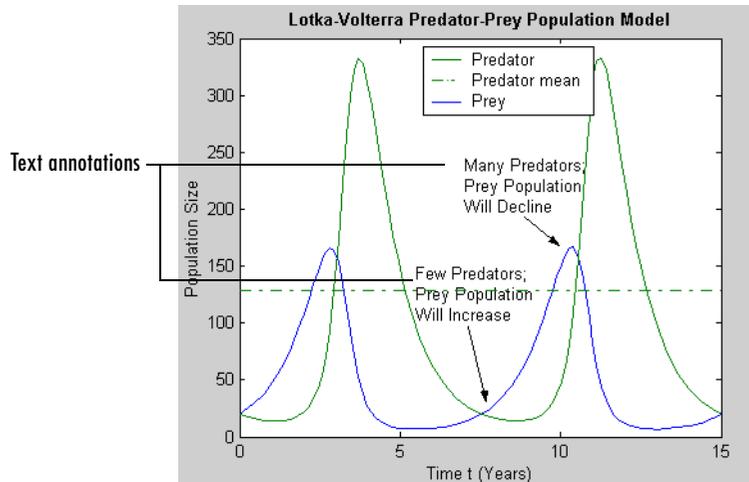
Repositioning Axis Labels

You can reposition an axis label using

- 1 Start plot editing mode by selecting **Edit Plot** from the figure **Tools** menu.
- 2 Select the text of the label you want to reposition (handles appear around the text object).
- 3 With the left mouse button down on the selected label, drag the text to the desired location and release.

Adding Text Annotations to Graphs

You can add free-form text annotations anywhere in a MATLAB figure to help explain your data or bring attention to specific points in your data sets.



If you enable plot editing mode, you can create text annotations by clicking in an area of the graph or the figure background and entering text. You can also add text annotations from the command line, using the `text` or `gtext` command.

Using plot editing mode or `gtext` make it easy to place a text annotation anywhere in graph. Use the `text` command when you want to position a text annotation at a specific point in a data set.

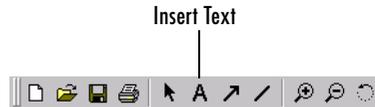
Note Text annotations created using the `text` or `gtext` command are anchored to the axes. Text annotations created in plot edit mode are not. If you move or resize your axes, you will have to reposition your text annotations.

Creating Text Annotations in Plot Editing Mode

Note Add text annotations after you are finished moving or resizing your axes. Text annotations created in plot edit mode are not anchored to axes. If you move or resize an axes, you will have to move the text annotations as well.

To add a text annotation to a graph:

- 1 Click on the **Insert** menu and choose the **Text** option or click the Insert Text button in the figure window toolbar.



MATLAB changes the cursor to a text insertion cursor.

Note After you use insert text, plot edit mode is enabled in the figure, if it was not already enabled.

- 2 Position the cursor where you want to add a text annotation in the graph and click. MATLAB opens a text editing box at that point in the graph.
- 3 Enter text.
- 4 Click anywhere in the figure background to close the text entry box. If you click on another object in the figure, such as an axes or line, you close the text entry box but also automatically select the object you clicked on.

Creating Text Annotations with the `text` or `gtext` Command

To create a text annotation using the `text` function, you must specify the text and its location in the graph, using x- and y-coordinates. You specify the coordinates in the units of the graph.

For example, the following code creates text annotations at specific points in the Lotka-Volterra Predator-Prey Population Model graph.

```
str1(1) = {'Many Predators;'};
str1(2) = {'Prey Population'};
str1(3) = {'Will Decline'};
text(7,220,str1)

str2(1) = {'Few Predators;'};
str2(2) = {'Prey Population'};
str2(3) = {'Will Increase'};
text(5.5,125,str2)
```

This example also illustrates how to create multi-line text annotations with cell arrays.

Calculating the Position of Text Annotations

You can also calculate the positions of text annotations in a graph. The following code adds annotations at three data points on a graph.

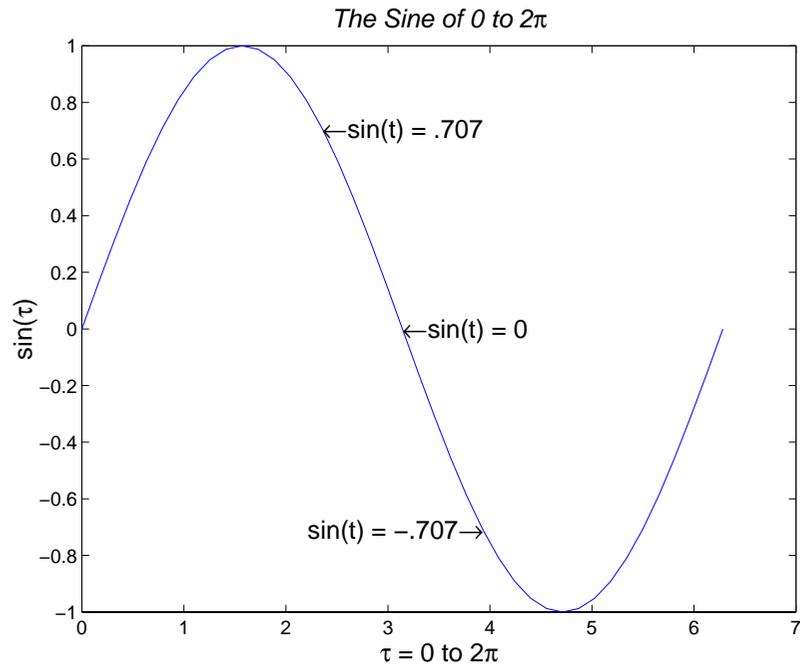
```
text(3*pi/4,sin(3*pi/4),...
    '\leftarrow sin(t) = .707',...
    'FontSize',16)

text(pi,sin(pi),'\leftarrow sin(t) = 0',...
    'FontSize',16)

text(5*pi/4,sin(5*pi/4),'sin(t) = -.707\rightarrow',...
    'HorizontalAlignment','right',...
    'FontSize',16)
```

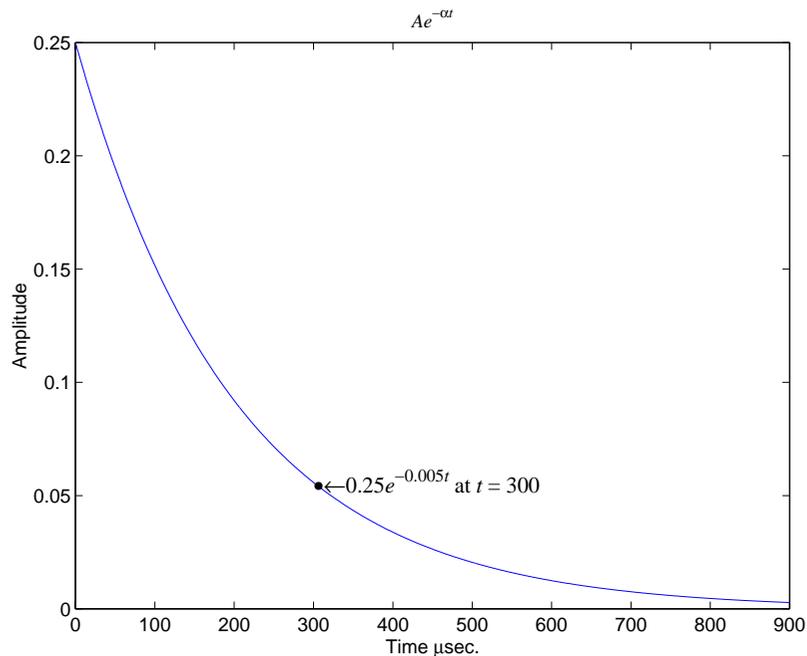
The `HorizontalAlignment` of the text string `'sin(t) = -.707 \rightarrow'` is set to `right` to place it on the left side of the point `[5*pi/4, sin(5*pi/4)]` on the graph. For more information about aligning text annotations, see “Text Alignment” on page 3-22.

Defining Symbols. For information on using symbols in text strings, see “Mathematical Symbols, Greek Letters, and TeX Characters” on page 3-25.



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{\ite}^{\{-0.005{\itt}\}} at {\itt} =
300', ...
'FontSize', 14)
```

This statement defines the text Position property as

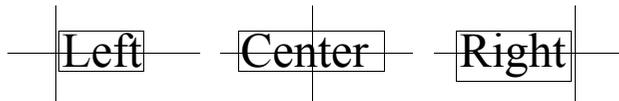
$$x = 300, y = 0.25e^{-0.005 \times 300}$$

The default text alignment places this point to the left of the string and centered vertically with the rectangle defined by the text Extent property. The following section provides more information about changing the default text alignment.

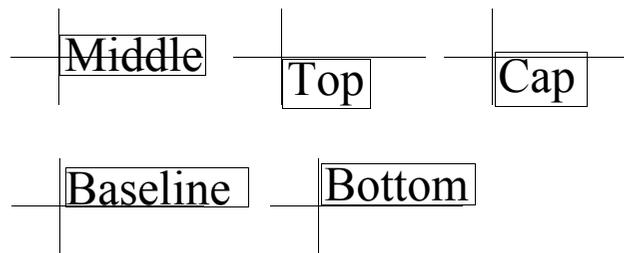
Text Alignment

The `HorizontalAlignment` and the `VerticalAlignment` properties control the placement of the text characters with respect to the specified x -, y -, and z -coordinates. The following diagram illustrates the options for each property and the corresponding placement of the text.

Text `HorizontalAlignment` property viewed with the `VerticalAlignment` property set to `middle` (the default).



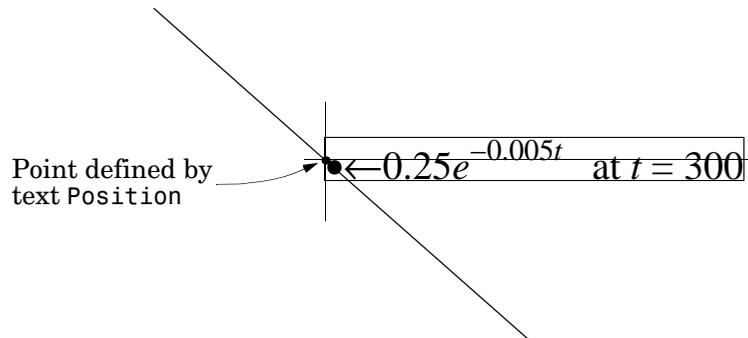
Text `VerticalAlignment` property viewed with the `HorizontalAlignment` property set to `left` (the default).



The default alignment is:

- `HorizontalAlignment = left`
- `VerticalAlignment = middle`

MATLAB does not place the text `String` exactly on the specified `Position`. For example, the previous section showed a plot with a point annotated with text. Zooming in on the plot enables you to see the actual positioning of the text.



The small dot is the point specified by the text `Position` property. The larger dot is the bullet defined as the first character in the text `String` property.

Example – Aligning Text

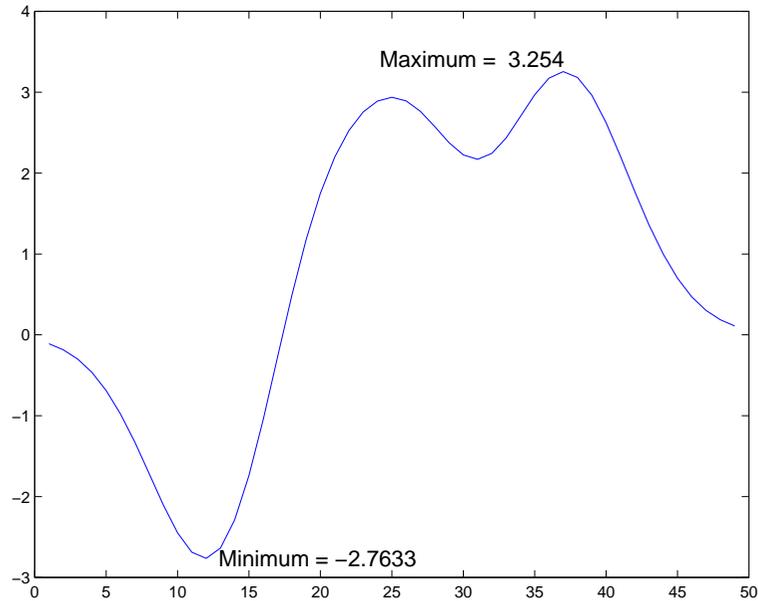
Suppose you want to label the minimum and maximum values in a plot with text that is anchored to these points and that displays the actual values. This example uses the plotted data to determine the location of the text and the values to display on the graph. One column from the peaks matrix generates the data to plot.

```
Z = peaks;
h = plot(Z(:,33));
```

The first step is to find the indices of the minimum and maximum values to determine the coordinates needed to position the text at these points (`get`, `find`). Then create the string by concatenating the values with a description of what the values are.

```
x = get(h,'XData'); % Get the plotted data
y = get(h,'YData');
imin = find(min(y) == y); % Find the index of the min and max
imax = find(max(y) == y);
text(x(imin),y(imin),[' Minimum = ',num2str(y(imin))],...
     'VerticalAlignment','middle',...
     'HorizontalAlignment','left',...
     'FontSize',14)
text(x(imax),y(imax),['Maximum = ',num2str(y(imax))],...
     'VerticalAlignment','bottom',...
     'HorizontalAlignment','right',...)
```

```
'FontSize', 14)
```



The text function positions the string relative to the point specified by the coordinates, in accordance with the settings of the alignment properties. For the minimum value, the string appears to the right of the text position point; for the maximum value the string appears above and to the left of the text position point. The text always remains in the plane of the computer screen, regardless of the view.

Editing Text Annotations

You can edit any of the text labels or annotations in a graph:

- 1 Start plot edit mode.
- 2 Double-click on the string. Or right-click on the string and select **String** from the context-sensitive pop-up menu.

An editing box appears around the text.

- 3 Make any changes to the text.
- 4 Click anywhere outside of the text edit box to end text editing.

Note To create special characters in text, such as Greek letters or mathematical symbols, use TeX sequences – see the text string property. If you create special characters by using the **Edit Font Properties** dialog box and selecting the Symbol font family, you will not be able to edit that text object using MATLAB commands.

Mathematical Symbols, Greek Letters, and TeX Characters

You can include mathematical symbols and Greek letters in text using TeX-style characters sequences. This section describes how to construct a TeX (LaTeX) character sequence used.

Available Symbols and Greek Letters

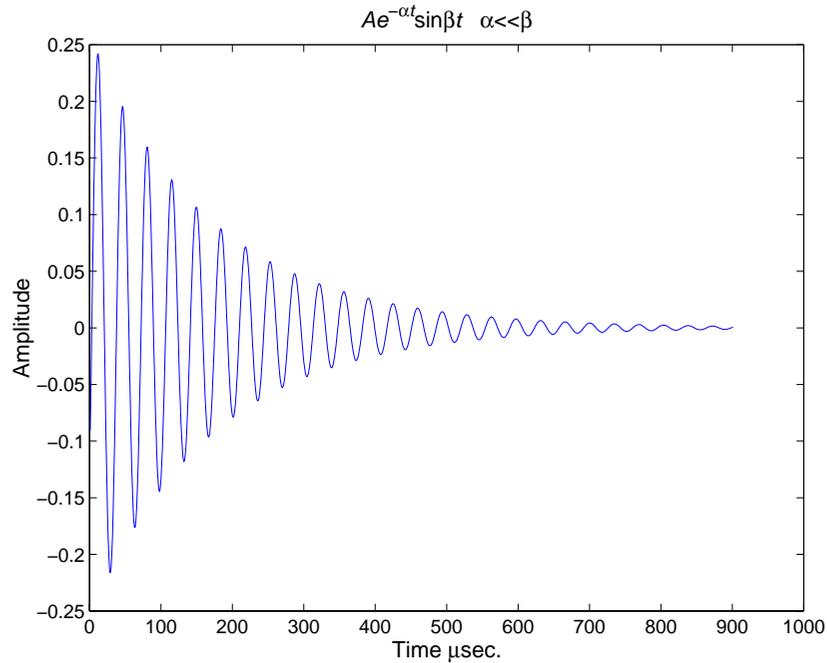
For a list of symbols and the character sequences used to define them, see the table of available TeX characters.

In general, you can define text that includes symbols and Greek letters using the text function, assigning the character sequence to the String property of text objects. You can also include these character sequences in the string arguments of the title, xlabel, ylabel, and zlabel commands.

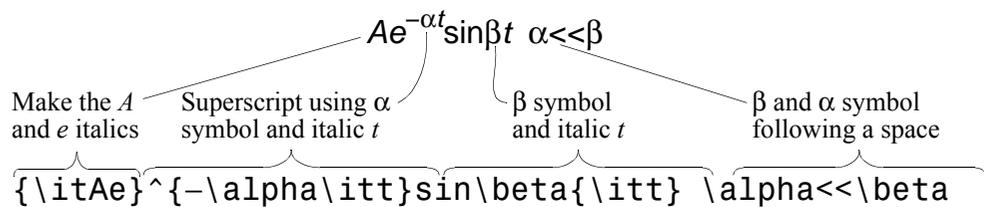
Example – Using a Mathematical Expression to Title a Graph

This example uses TeX character sequences to create graph labels. The following statements add a title and x - and y -axis labels to an existing graph.

```
title({'\itAe}^{\-\alpha\itt}\sin\beta{\itt} \alpha<<\beta'})
xlabel('Time \musec.')
ylabel('Amplitude')
```



The backslash character “\” precedes all TeX character sequences. Looking at the string defining the title illustrates how to use these characters.



Controlling the Interpretation of TeX character

The text `Interpreter` property controls the interpretation of TeX characters. If you set this property to `none`, MATLAB interprets the special characters literally.

Using Character and Numeric Variables in Text

Any string variable is a valid specification for the text String property. This section illustrates some how to use matrix, cell array, and numeric variables as arguments to the text function.

Character Variables

For example, each row of the matrix `PersonalData` contains specific information about a person (note that all but the longest row is padded with a space so that each has the same number of columns).

```
PersonalData = ['Jack Straw  '; '489 Main St.'; 'Wichita KN  '];
```

To display the data, index into the desired row.

```
text(x1,y1,['Name: ',PersonalData(1,:)])
text(x2,y2,['Address: ',PersonalData(2,:)])
text(x3,y3,['City and State: ',PersonalData(3,:)])
```

Cell Arrays

Using a cell array enables you to create multi-line text with a single text object. Each cell does not need to be the same number of characters. For example, the following statements,

```
key(1)={'{\itAe}^{\alpha\itt}\sin\beta{\itt}'};
key(2)={'Time in \musec'};
key(3)={'Amplitude in volts'};
text(x,y,key)
```

produce this output.

```
Ae-αtsinβt
Time in μsec
Amplitude in volts
```

Numeric Variables

You can specify numeric variables in text strings using the `num2str` (number to string) function. For example, if you type on the command line,

```
x = 21;
['Today is the ',num2str(x),'st day.']
```

MATLAB concatenates the three separate strings into one.

```
Today is the 21st day.
```

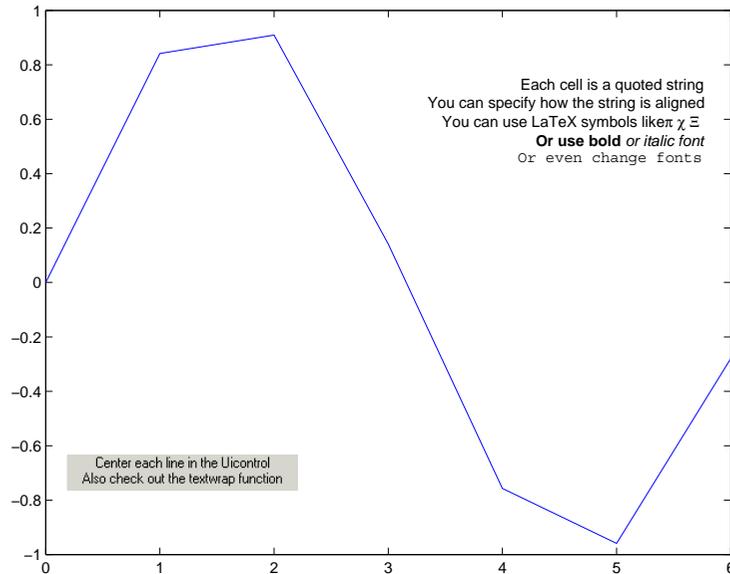
Since the result is a valid string, you can specify it as a value for the `text` String property.

```
text(xcoord,ycoord,['Today is the ',num2str(x),'st day.'])
```

Example - Multiline Text

MATLAB supports multiline text strings using cell arrays. Simply define a string variable as a cell array with one line per cell. This example defines two cell arrays, one used for a `uicontrol` and the other as `text`.

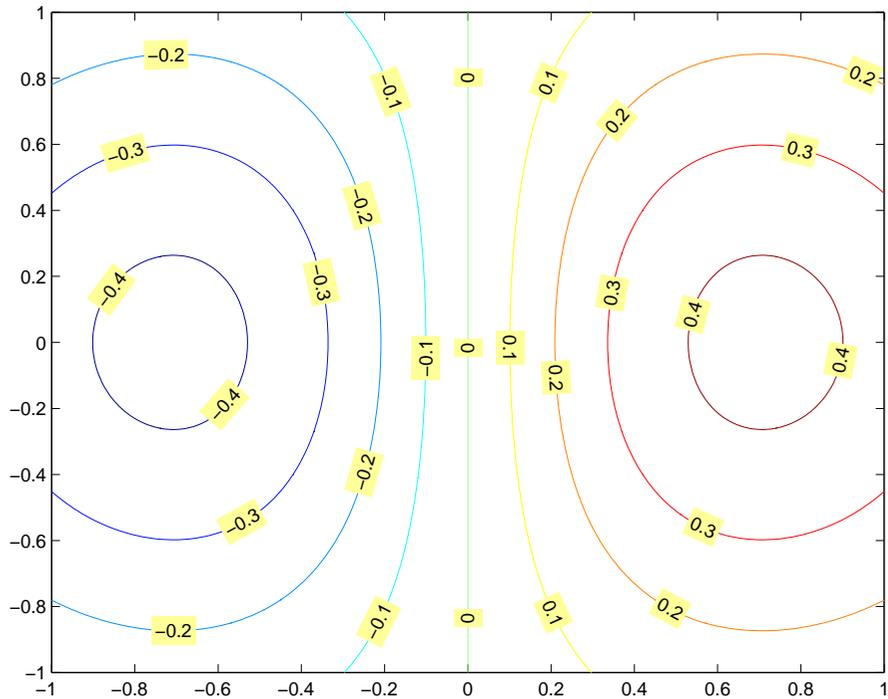
```
str1(1) = {'Center each line in the Uicontrol'};  
str1(2) = {'Also check out the textwrap function'};  
  
str2(1) = {'Each cell is a quoted string'};  
str2(2) = {'You can specify how the string is aligned'};  
str2(3) = {'You can use LaTeX symbols like \pi \chi \Xi'};  
str2(4) = {'\bfOr use bold \rm\itor italic font\rm'};  
str2(5) = {'\fontname{courier}Or even change fonts'};  
plot(0:6,sin(0:6))  
uicontrol('Style','text','Position',[80 80 200 30],...  
         'String',str1);  
text(5.75,sin(2.5),str2,'HorizontalAlignment','right')
```



Drawing Text in a Box

When you use the `text` command to display a character string, the string's position is defined by a rectangle called the Extent of the text. You can display this rectangle either as a box or a filled area. For example, you can highlight contour labels to make the text easier to read.

```
[x,y] = meshgrid(-1:.01:1);
z = x.*exp(-x.^2-y.^2);;
[c,h]=contour(x,y,z);
h = clabel(c,h);
set(h, 'BackgroundColor', [1 1 .6])
```

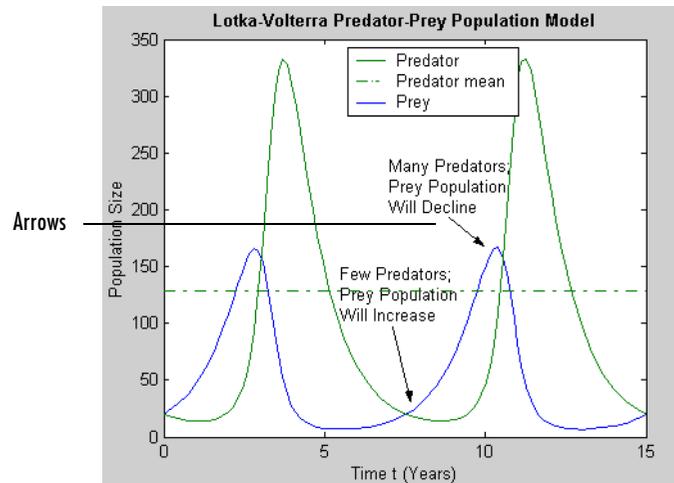


For additional features, see the following text properties:

- `BackgroundColor` – color of the rectangle’s interior (none by default).
- `EdgeColor` – color of the rectangle’s edge (none by default).
- `LineStyle` – style of the rectangle’s edge line (first set `EdgeColor`).
- `LineWidth` – width of the rectangle’s edge line (first set `EdgeColor`).
- `Margin` – increase the size of the rectangle by adding a margin to the text extent.

Adding Arrows and Lines to Graphs

With plot editing mode enabled, you can add arrows and lines anywhere in a figure window.



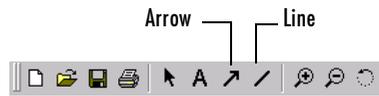
You can also use arrow characters (TeX characters) to create arrows using the text command. However, arrows created this way can only point to the left or right, horizontally. See “Calculating the Position of Text Annotations” on page 3-19 for an example.

Creating Arrows and Lines in Plot Editing Mode

Note Add arrow and line annotations after you are finished moving or resizing your axes. Arrows and lines annotations are not anchored to the axes. If you move or resize the axes, you will have to reposition your arrow and line annotations as well.

To add an arrow or line annotation to a graph:

- 1 Click on the **Insert** menu and choose the **Arrow** or **Line** option, or click the Arrow or Line buttons in the figure window toolbar.



MATLAB changes the cursor to a cross-hair style.

- 2 Position the cursor in the figure where you want to start the line or arrow and press either mouse button. Hold the button down and move the mouse to define the length and direction of the line or arrow.
- 3 Release the mouse button.

Note After you add an arrow or line, plot edit mode is enabled in the figure, if it was not already enabled.

Editing Arrows and Line Annotations

You can edit the appearance of arrow and line annotations using the Property Editor.

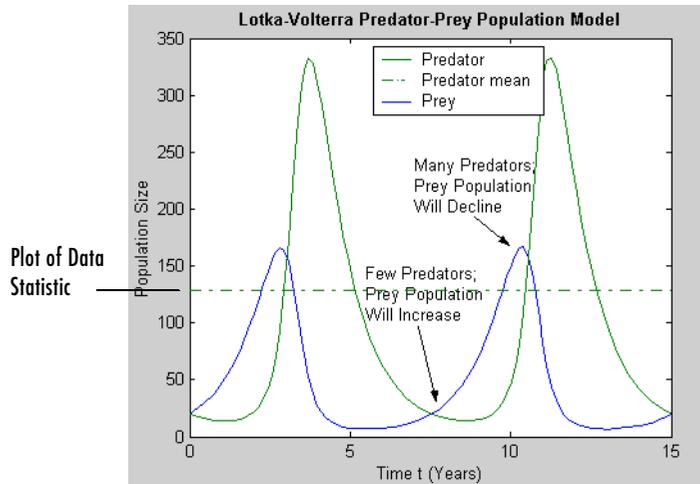
With plot editing mode enabled, double-click on the arrow or line annotation to start the Property Editor, if it is not already started. The Property Editor displays the set of panels specific to arrow or line objects. Use the fields in the panels to change the style, width, color, and many other characteristics of the arrow or line annotation object.

Adding Plots of Basic Statistics to Graphs

The MATLAB Data Statistics tool:

- Calculates basic statistics about the central tendency and variability of data plotted in a graph.
- Plots these statistics in the graph

For example, the following figure includes a plot of the mean of the Predator y-data.



The following sections provide more information about using the Data Statistics tool:

- “Example - Plotting the Mean of a Data Set” on page 3-34
- “Formatting Plots of Data Statistics” on page 3-36
- “Statistics Plotted by the Data Statistics Tool” on page 3-36
- “Viewing Statistics for Multiple Plots” on page 3-37
- “Saving Statistics to the MATLAB Workspace” on page 3-38

Example - Plotting the Mean of a Data Set

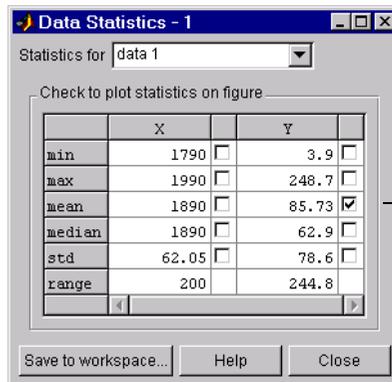
To add a plot of the mean of a data set to a graph:

- 1 Plot your data. For example, use these commands to plot historical population data from the United States census.

```
load census  
plot(cdate, pop, '+' )
```

- 2 Select the **Data Statistics** option from the figure window **Tools** menu.

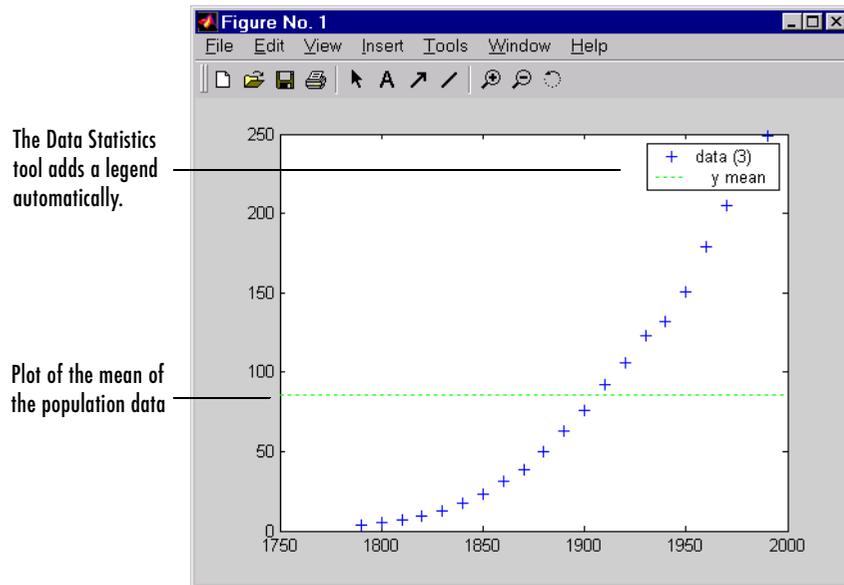
The Data Statistics tool calculates basic statistics on the x -data and y -data of the plot in the graph and displays the results in a dialog box.



Select the statistic you want to plot by clicking in its check box.

- 3 Select the statistic you want to plot in your graph by clicking in the check box next to the value.

For example, to add a plot of the mean of the population data (y -data) to the graph, click in the check box next to the value (as shown in the figure). The Data Statistics tool adds the plot of the mean to the graph.



Using a Legend with Data Statistics

When you activate the Data Statistics tool, it calculates statistics for the plotted data and automatically adds a legend to the graph, if the graph doesn't already have one.

Initially, the legend only includes entries for the data sets plotted in the graph. In the legend, each data set is identified by its tag. (A tag is a user-defined text string that can be associated with any graphics object. For information about creating tags, see "Identifying Objects in a Graph" on page 1-20.) If there are data sets in the graph that do not have tags, the Data Statistics tool creates a label for them, using `data 1` to identify the first plot, `data 2` to identify the second plot, and so on.

When you add a plot of one or more statistics to the graph, the Data Statistics tool adds an entry in the legend for the new plot. The Data Statistics tool assigns the plotted statistic a descriptive name that identifies it in the legend. In the example, the plotted statistic has the name `y mean` in the legend.

Formatting Plots of Data Statistics

The Data Statistics tool uses color and line style to distinguish the plots of statistics from the other plots in a graph. However, like any other plot in a graph, you can change these characteristics.

Note Do not edit the format of the plots of data statistics, until you are finished adding them to a graph. If you edit a plot of data statistics, delete the plot, and then add it back, any formatting you did to the plotted statistics will be lost.

To modify the properties of a plotted statistic:

- 1 Enable plot editing mode in the figure window.
- 2 Double-click on the plot of the statistic. This starts the MATLAB Property Editor, which provides access to properties of the line object used to plot the statistic.

You can also access a subset of these properties by right-clicking on the plot. This brings up the plot's context menu, which includes options for specifying line width, line style, and color.

- 3 Change the properties of the plot and click **Apply**.

Statistics Plotted by the Data Statistics Tool

Note You can only use the Data Statistics tool to generate statistics for two-dimensional data (vectors and matrices).

The following table lists the statistics calculated by the Data Statistics tool. The table includes the name of the MATLAB function used to calculate the

statistic. For more information about these statistical functions, see the “Basic Data Analysis Functions” in the “Data Analysis and Statistics” chapter.

Statistic	Description	MATLAB Function
Maximum	The largest value in the data set	max
Minimum	The smallest value in the data set	min
Mean	The average of all the values in the data set	mean
Median	The middle value in the data set	median
Range	The interval between the lowest value and the highest value in the data set. The Data Statistics tool does not plot the range statistic.	n/a
Standard deviation	A measure characterizing the amount of variation among the values in the data set Note: The Data Statistics tool uses two lines to plot the standard deviation in a graph. The lines represent the boundaries of one standard deviation on either side of the mean of the data set.	std

Automatic Updating of Statistics

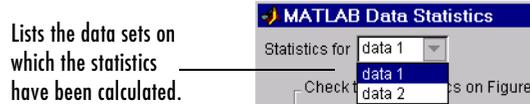
If you have the Data Statistics tool displayed and you change the x -data or y -data of a plot, the Data Statistics tool automatically regenerates the statistics for that plot.

Viewing Statistics for Multiple Plots

The Data Statistics tool calculates basic statistics for every 2-D plot in a graph, but displays the statistics for only one plot at time.

To view the statistics for a particular plot in a graph:

- 1 Click the **Statistics for** menu in the Data Statistics dialog box.



This menu lists all the data sets plotted in the graph, identifying each data set by its tag. (A tag is a user-defined text string that can be associated with any graphics object.) For plots in the graph that do not have tags, the Data Statistics tool uses data1 to identify the first plot, data2 to identify the second plot, and so on.

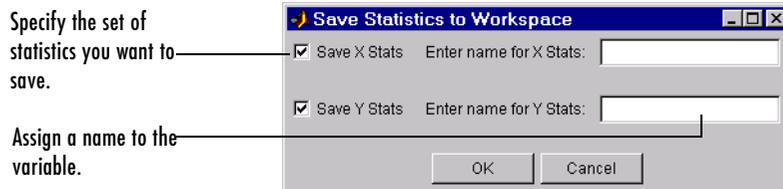
- 2 Select a plot from the list. The Data Statistics tool updates the values displayed in the dialog box.

Saving Statistics to the MATLAB Workspace

To save the statistics generated by the Data Statistics tool to the MATLAB workspace, follow this procedure:

Note You must repeat this procedure for each plot in a graph containing multiple plots.

- 1 Click the **Save to Workspace** button.
- 2 In the **Save Statistics to Workspace** dialog box, specify which sets of statistics you want to save, *x*-data or *y*-data, and specify the names you want to assign to the variables in which the statistics will be stored.



The Data Statistics tool saves each set of statistics in a structure. For example, if you save the set of statistics on the x-data in the census in the variable `census_dates`, the contents of the structure looks like this.

```
census_dates =  
  
    min: 1790  
    max: 1990  
    mean: 1890  
median: 1890  
    std: 62.0484  
    range: 200
```


Creating Specialized Plots

Bar and Area Graphs (p. 4-2)	View results over time, comparing results, and displaying individual contribution to a total amount.
Pie Charts (p. 4-14)	Individual contribution to a total amount.
Histograms (p. 4-17)	Distribution of data values.
Discrete Data Graphs (p. 4-22)	Stem and staircase plots of discrete data.
Direction and Velocity Vector Graphs (p. 4-31)	Compass, feather, and quiver plots show direction and magnitude.
Contour Plots (p. 4-37)	Indicate locations of equal data values.
Interactive Plotting (p. 4-48)	User-selectable data point (using mouse) for plotting.
Animation (p. 4-50)	Show an additional data dimension by sequencing plots.

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.

Function	Description
bar	Displays columns of m -by- n matrix as m groups of n vertical bars
barh	Displays columns of m -by- n matrix as m groups of n horizontal bars
bar3	Displays columns of m -by- n matrix as m groups of n vertical 3-D bars
bar3h	Displays columns of m -by- n matrix as m groups of n horizontal 3-D bars
area	Displays vector data as stacked area plots

Types of Bar Graphs

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

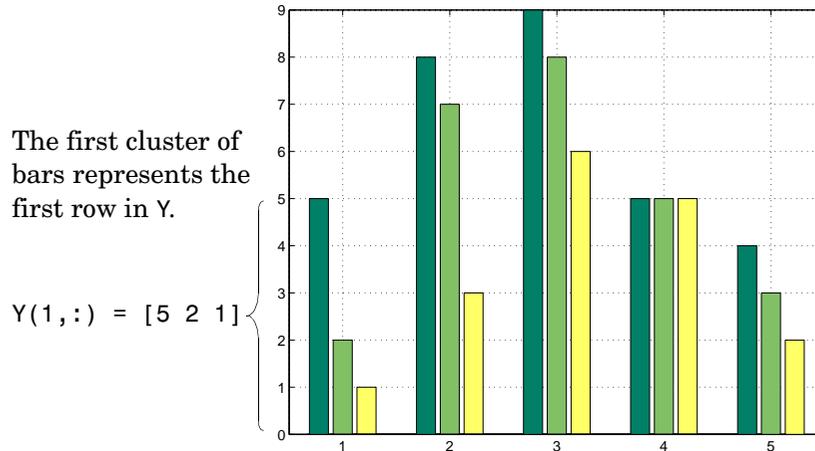
By default, a bar graph represents each element in a matrix as one bar. Bars in a 2-D bar graph, created by the bar function, are distributed along the x -axis with each element in a column drawn at a different location. All elements in a row are clustered around the same location on the x -axis.

For example, define Y as a simple matrix and issue the `bar` statement in its simplest form.

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

```
bar(Y)
```

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

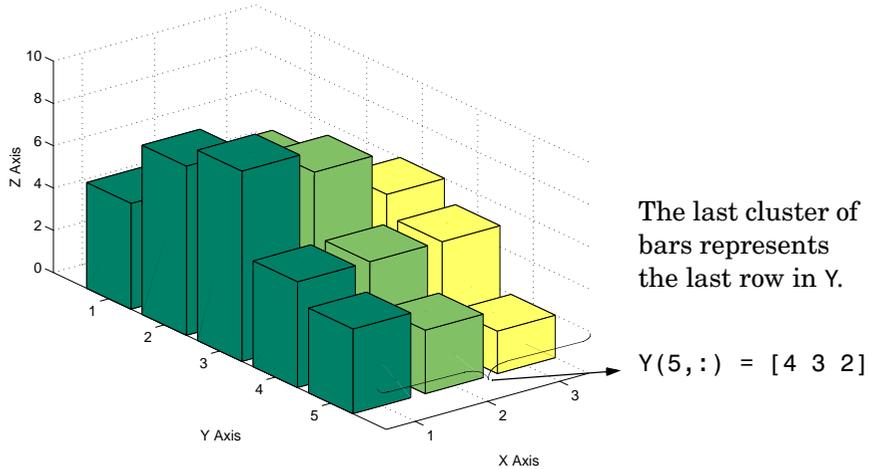


Detached 3-D Bars

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

```
bar3(Y)
```

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



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

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

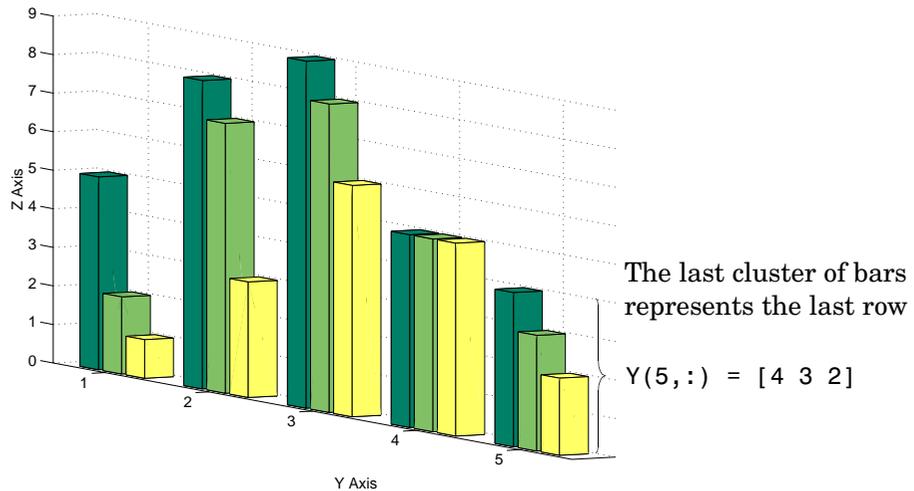
```
xlabel('X Axis')
ylabel('Y Axis')
zlabel('Z Axis')
set(gca, 'XTick', [1 2 3])
```

Grouped 3-D Bars

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

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

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



Stacked Bar Graphs to Show Contributing Amounts

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

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

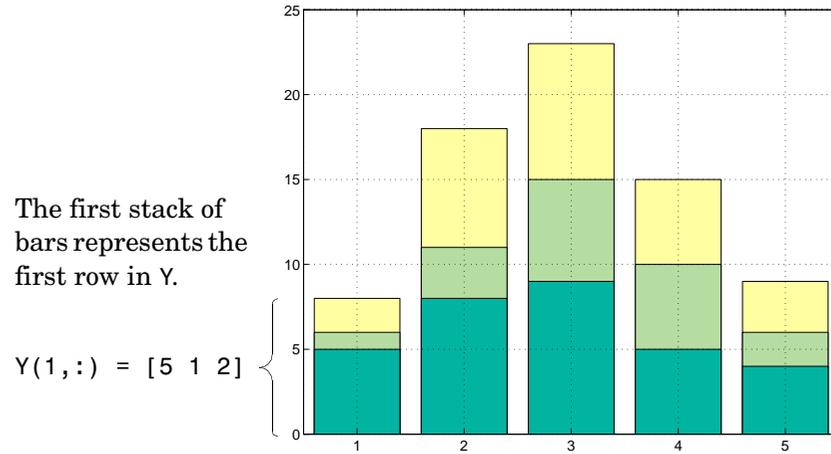
Redefining Y

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

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

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

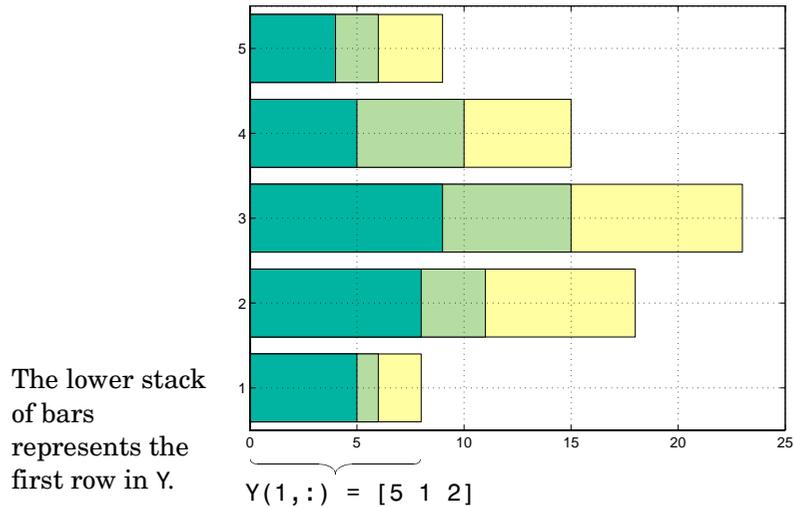
creates a 2-D stacked bar graph, where all elements in a row correspond to the same x location.



Horizontal Bar Graphs

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

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



Specifying X-Axis Data

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

For example, given temperature data,

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

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

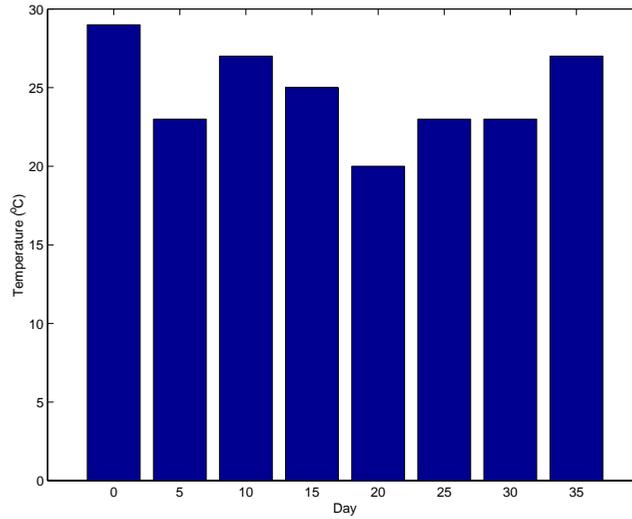
```
days = 0:5:35;
```

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

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

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

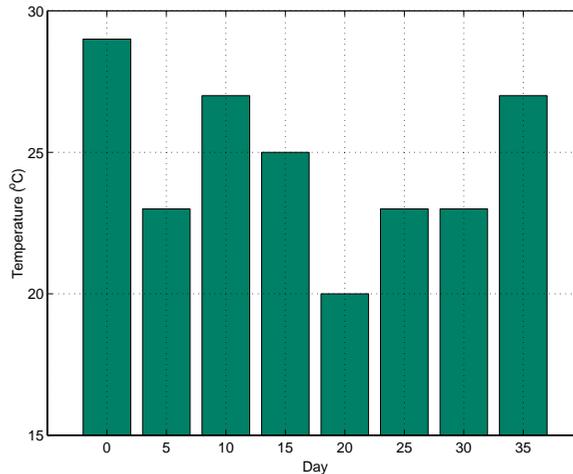
```
xlabel('Day')
ylabel('Temperature (^{o}C)')
```



Setting Y-Axis Limits

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

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



Overlaying Plots on Bar Graphs

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

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

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

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

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

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

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

Overlaying a Line Plot on the Bar Graph

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

```
h1 = gca;
```

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

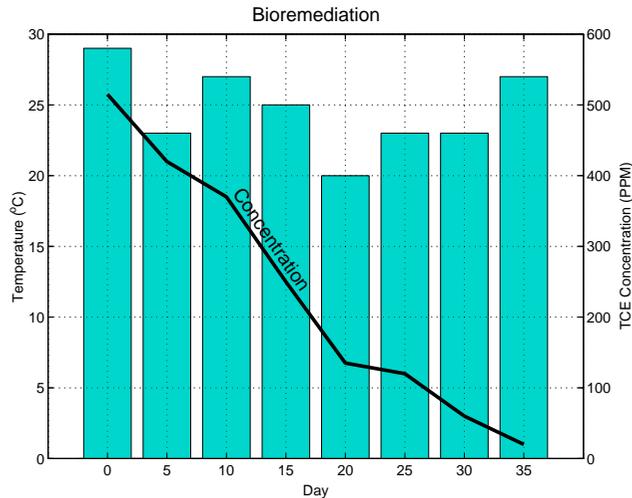
```
h2 = axes('Position',get(h1,'Position'));
plot(days,TCE,'LineWidth',3)
```

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

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

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

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



Annotating the Graph. These statements annotate the graph.

```
text(11,380,'Concentration','Rotation',-55,'FontSize',16)
```

```
ylabel('TCE Concentration (PPM)')
title('Bioremediation', 'FontSize', 16)
```

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

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

Area Graphs

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

Area Graphs Showing Contributing Amounts

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

Using this matrix,

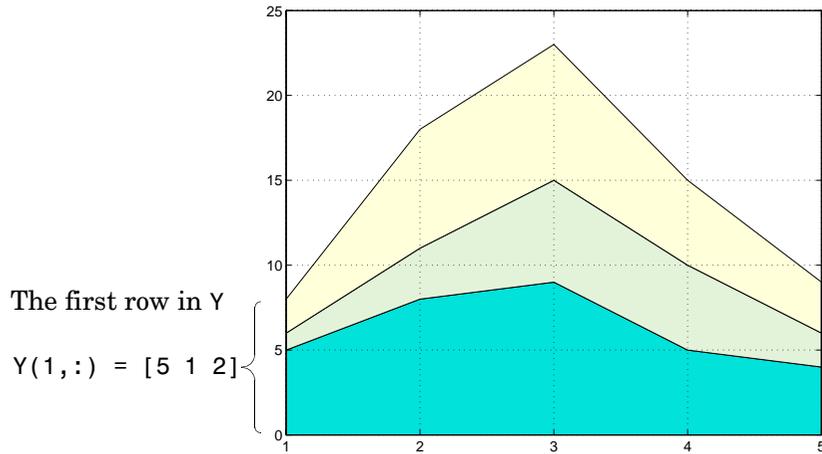
```
Y = [5 1 2
      8 3 7
      9 6 8
      5 5 5
      4 2 3];
```

the statement,

```
area(Y)
```

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

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



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

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

Comparing 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 `patch` for a complete list of properties.

```
area(x, sales, 'FaceColor', [.5 .9 .6], ...
```

```

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

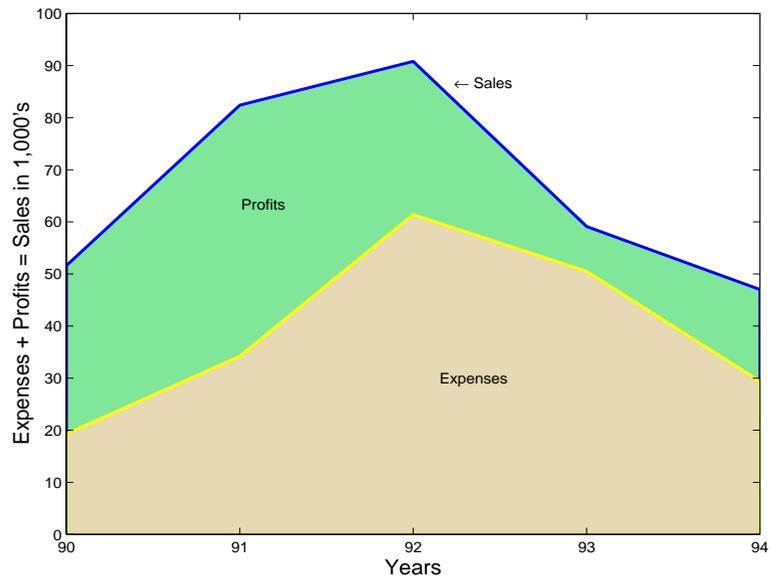
```

To annotate the graph, use the statements

```

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

```



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.

Example – Pie Chart

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

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

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

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

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

First, create a vector containing zeros.

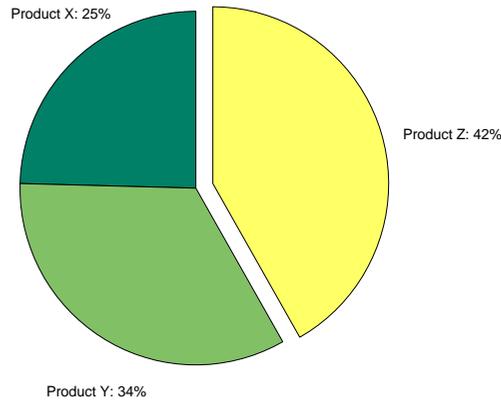
```
explode = zeros(size(x));
```

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

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

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

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



Labeling the 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(textObjs, {'Extent'});
newExt = cat(1, val1{:});
offset = sign(oldExt(:,1)).*(newExt(:,3) - oldExt(:,3))/2;
```

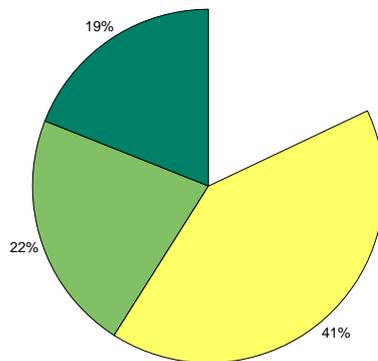
```
pos = get(textObjs, {'Position'});  
textPos = cat(1, pos{:});  
textPos(:,1) = textPos(:,1)+offset;  
set(textObjs,{'Position'},num2cell(textPos,[3,2]))
```

Removing a Piece from a Pie Charts

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 histogram functions show the distribution of data values. The functions that create histograms are `hist` and `rose`.

Function	Description
<code>hist</code>	Displays data in a Cartesian coordinate system
<code>rose</code>	Displays data in a polar coordinate system

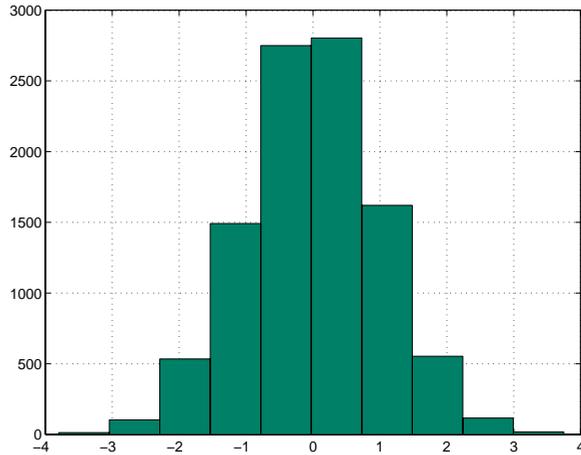
The histogram functions count the number of elements within a range and display each range as a rectangular bin. The height (or length when using `rose`) of the bins represents the number of values that fall within each range.

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`.

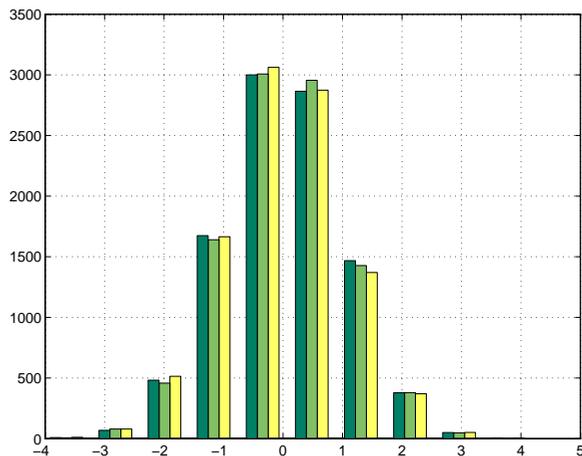


Matrix Input Argument

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

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

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



Histograms in Polar Coordinates

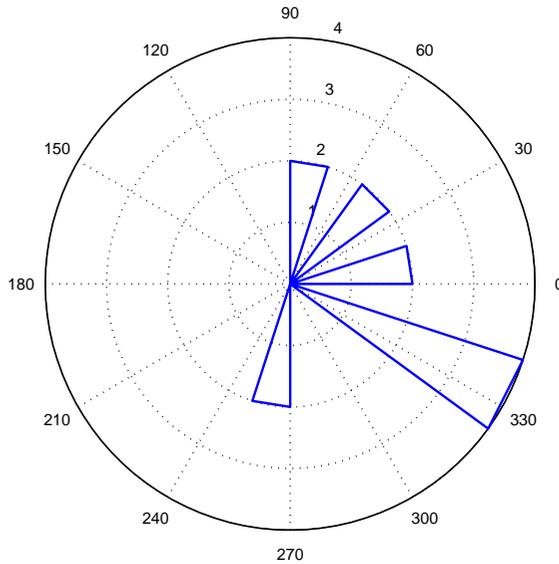
A rose plot is a histogram created in a polar coordinate system. For example, consider samples of the wind direction taken over a 12-hour period.

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

To display this data using the rose function, convert the data to radians; then use the data as an argument to the rose function. Increase the `LineWidth` property of the line to improve the visibility of the plot (`findobj`).

```
wdir = wdir * pi/180;
rose(wdir)
hline = findobj(gca, 'Type', 'line');
set(hline, 'LineWidth', 1.5)
```

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



Specifying Number of Bins

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

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

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

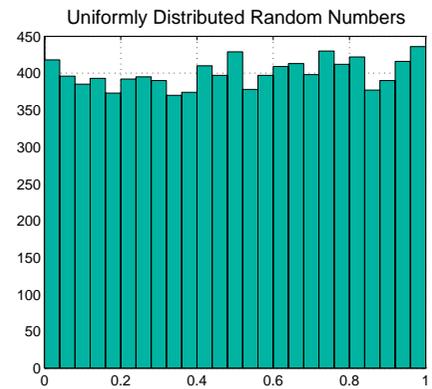
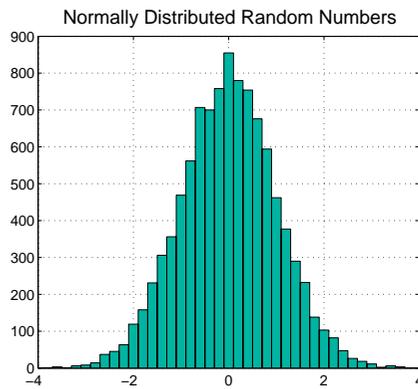
The first histogram displays the data distribution resulting from the `randn` function. The locations on the *x*-axis and number of bins depend on the vector `x`.

```
x = min(yn):.2:max(yn);  
subplot(1,2,1)
```

```
hist(yn,x)
title('Normally Distributed Random Numbers','FontSize',16)
```

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

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



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

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

Discrete Data Graphs

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 section, are also suitable for displaying discrete data.)

- “Two-Dimensional Stem Plots” on page 4-22 – compares 2-D stem and line plots and shows techniques for customizing the stems.
- “Combining Stem Plots with Line Plots” on page 4-25 – combination line and stem plot with legend.
- “Three-Dimensional Stem Plots” on page 4-26 – 3-D stem plot of an FFT and a combination 3-D stem and line plot.
- “Stairstep Plots” on page 4-29 – plotting a mathematical function with a stairstep plot.

The following table lists the commands described in this section.

Function	Description
stem	Displays a discrete sequence of y -data as stems from x -axis
stem3	Displays a discrete sequence of z -data as stems from xy -plane
stairs	Displays a discrete sequence of y -data as steps from x -axis

Two-Dimensional Stem Plots

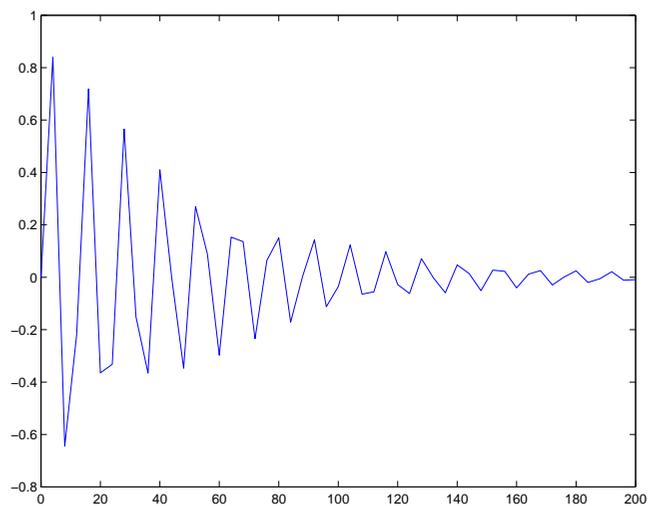
A stem plot displays data as lines (stems) terminated with a marker symbol at each data value. In a 2-D graph, stems extend from the x -axis.

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

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

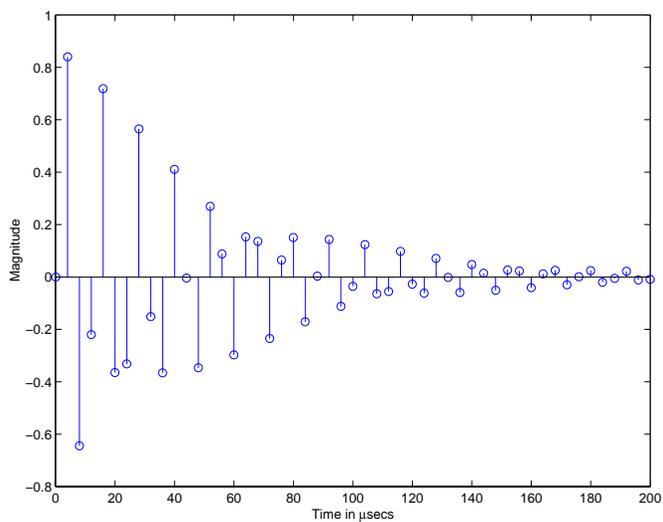
yields a vector of discrete values for y at given values of t . A line plot shows the data points connected with a straight line.

```
plot(t,y)
```



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

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



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

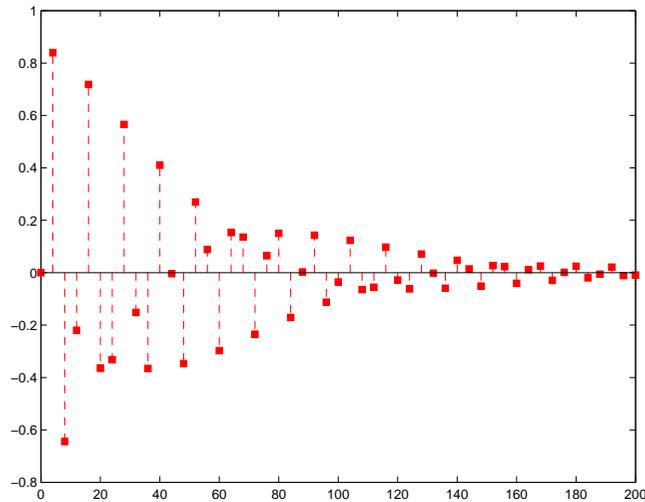
```
xlabel('Time in \musecs')  
ylabel('Magnitude')
```

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

Customizing the Graph

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

```
stem(t,y,'--sr','fill')
```



Setting the aspect ratio of the x - and y -axis to 2:1 improves the utility of the graph. You can do this by setting the aspect ratio of the plot box using `pbaspect`.

```
pbaspect([2,1,1])
```

This is equivalent to setting the `PlotBoxAspectRatio` property directly.

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

See `LineStyle` for a list of line styles and marker types.

Combining Stem Plots with Line Plots

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

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

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

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

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

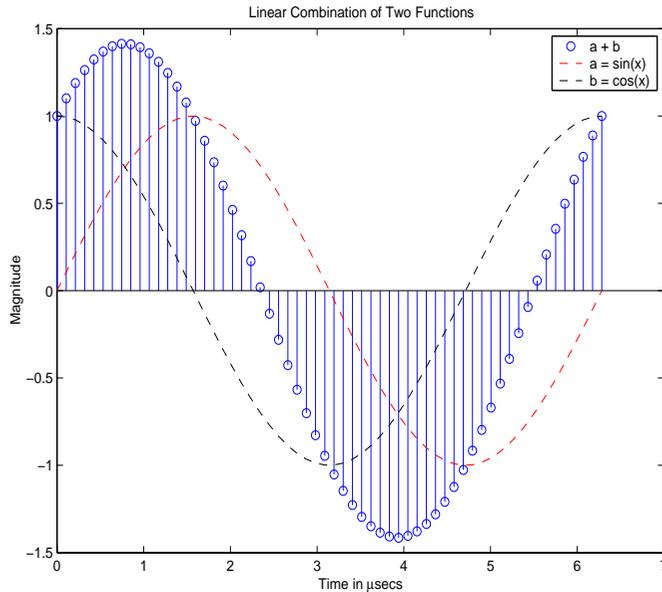
```
hold on  
plot_handles = plot(x,a,'--r',x,b,'--g');  
hold off
```

Use `legend` to annotate the graph. The stem and plot handles passed to `legend` identify 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.

Example – 3-D Stem Plot of an FFT

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

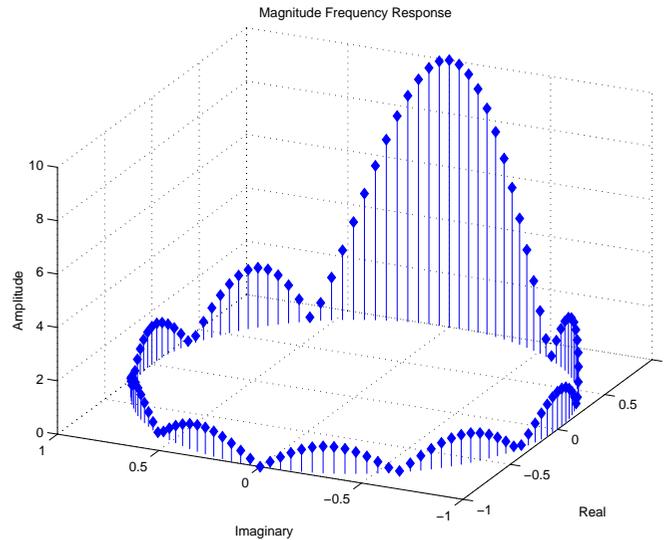
```
th = (0:127)/128*2*pi;  
x = cos(th);  
y = sin(th);
```

and the magnitude frequency response of a step function

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

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

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



Label the Graph

Label the graph with the statements

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

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

```
rotate3d on
```

Example – Combining Stem and Line Plots

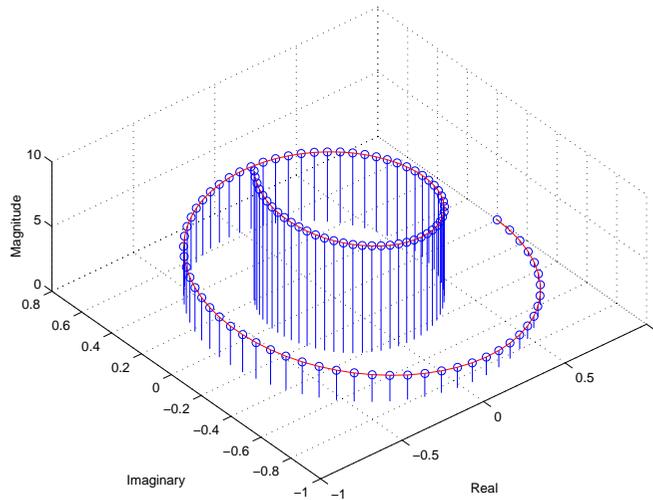
Three-dimensional stem plots work well when visualizing discrete functions that do not output a large number of data points. For example, you can use

`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,'r')
hold off
view(-39.5,62)
```



Label the Graph

Add axes labels, with the statements

```
xlabel('Real')
ylabel('Imaginary')
zlabel('Magnitude')
```

Stairstep Plots

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

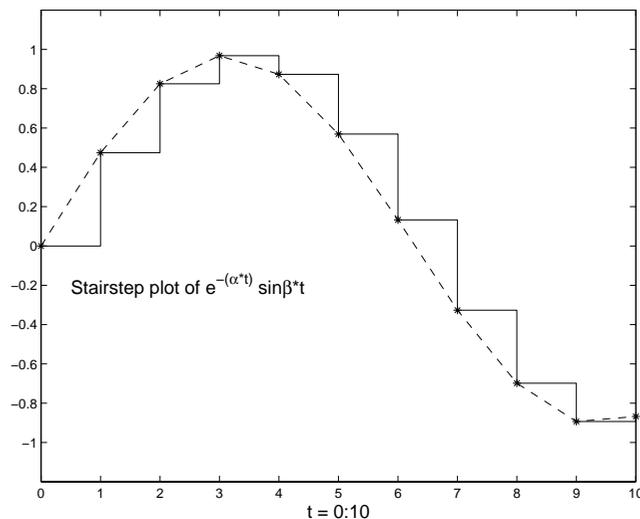
Example – Stairstep Plot of a Function

For example, define a function f that varies over time,

```
alpha = 0.01;
beta = 0.5;
t = 0:10;
f = exp(-alpha*t).*sin(beta*t);
```

Use `stairs` to display the function as a stairstep plot and a linearly interpolated function.

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



Annotate the graph and set the axes limits.

```
label = 'Stairstep plot of e^{-(\alpha*t)} sin\beta*t';  
text(0.5,-0.2,label,'FontSize',14)  
xlabel('t = 0:10','FontSize',14)  
axis([0 10 -1.2 1.2])
```

Direction and Velocity Vector Graphs

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

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

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.

Example – Compass Plot of Wind Direction and Speed

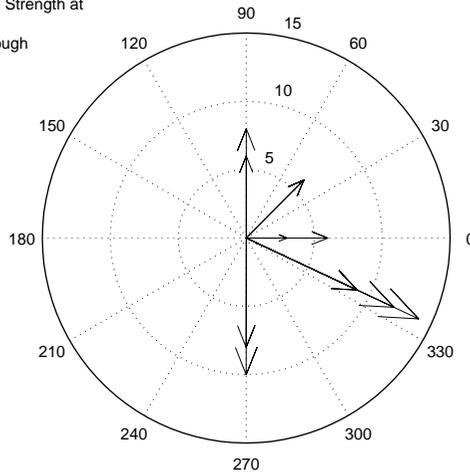
This example shows a compass plot indicating the wind direction and strength during a 12-hour period. Two vectors define the wind direction and strength.

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

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

```
rdir = wdir * pi/180;
[x,y] = pol2cart(rdir,knots);
compass(x,y)
```

Wind Direction and Strength at
Logan Airport for
Nov. 3 at 1800 through
Nov. 4 at 0600



Create text to annotate the graph.

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

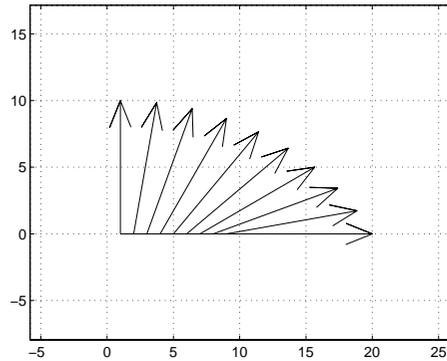
Feather Plots

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

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

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

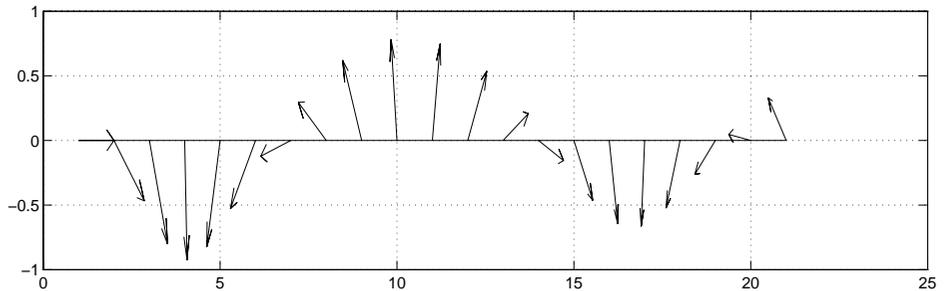
```
[u,v] = pol2cart(theta*pi/180,r*10);
feather(u,v)
axis equal
```



Plotting Complex Numbers

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

```
t = 0:0.5:10;    % Time limits
s = 0.05+i;      % Spiral rate
Z = exp(-s*t);  % Compute decaying exponential
feather(Z)
```



Printing the Graph

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

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

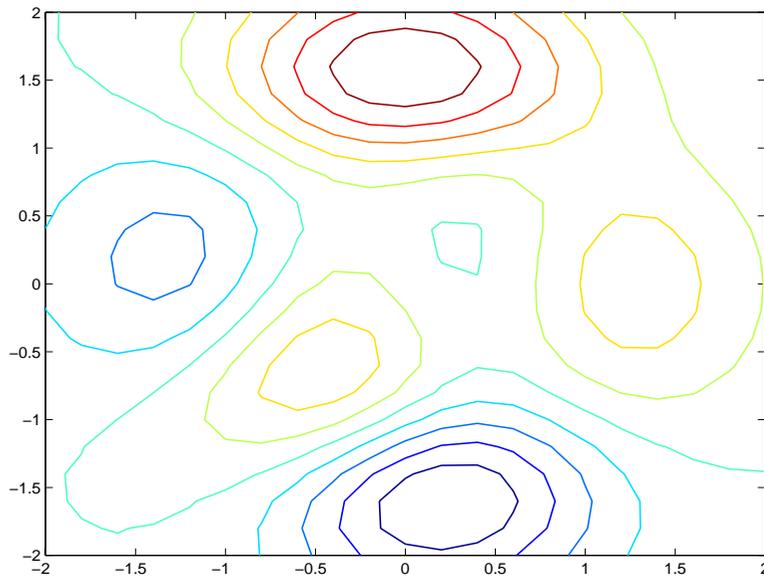
In this mode, MATLAB prints the figure as it appears on screen.

Two-Dimensional Quiver Plots

The quiver function shows vectors at given points in two-dimensional space. The vectors are defined by x and y components.

A quiver plot is useful when displayed with another plot. For example, create 10 contours of the peaks function (see the “Contour Plots” section for more information).

```
n = -2.0:.2:2.0;  
[X,Y,Z] = peaks(n);  
contour(X,Y,Z,10)
```

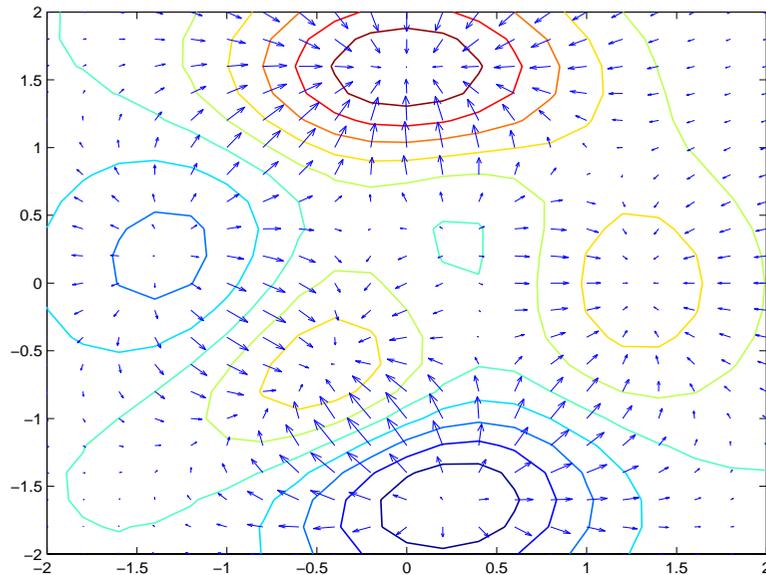


Now use gradient to create the vector components to use as inputs to quiver.

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

Set hold to on and add the contour plot.

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



Three-Dimensional Quiver Plots

Three-dimensional quiver plots (`quiver3`) display vectors consisting of (u,v,w) components at (x,y,z) locations. For example, you can show the path of a projectile as a function of time,

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

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.

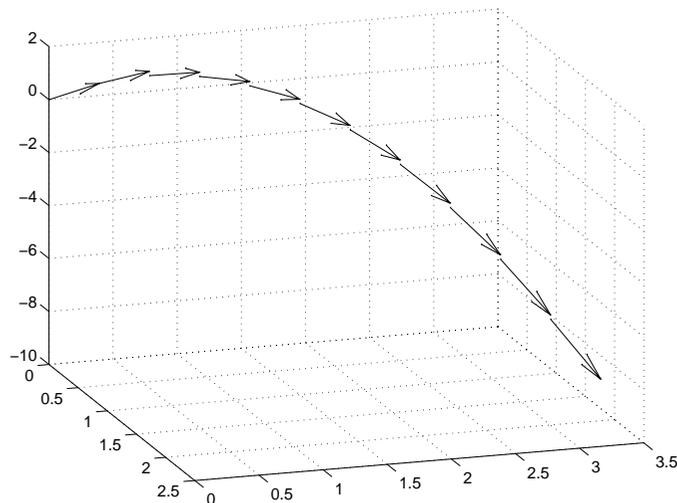
```
t = 0:.1:1;  
z = vz*t + 1/2*a*t.^2;
```

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)  
view([70 18])
```



Contour Plots

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

Function	Description
<code>clabel</code>	Generates labels using the contour matrix and displays the labels in the current figure.
<code>contour</code>	Displays 2-D isolines generated from values given by a matrix Z .
<code>contour3</code>	Displays 3-D isolines generated from values given by a matrix Z .
<code>contourf</code>	Displays a 2-D contour plot and fills the area between the isolines with a solid color.
<code>contourc</code>	Low-level function to calculate the contour matrix used by the other contour functions.
<code>meshc</code>	Creates a mesh plot with a corresponding 2-D contour plot.
<code>surf</code>	Creates a surface plot with a corresponding 2-D contour plot.

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.

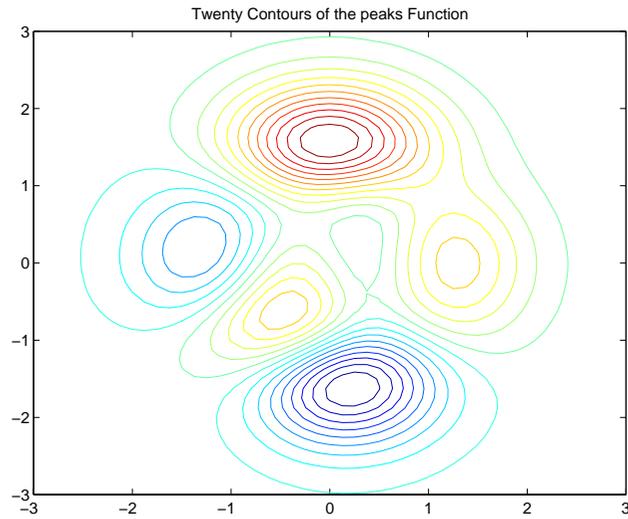
To explicitly set the number of contour levels displayed by the functions, you specify a second optional argument.

Contour Plot of the Peaks Function

The statements,

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

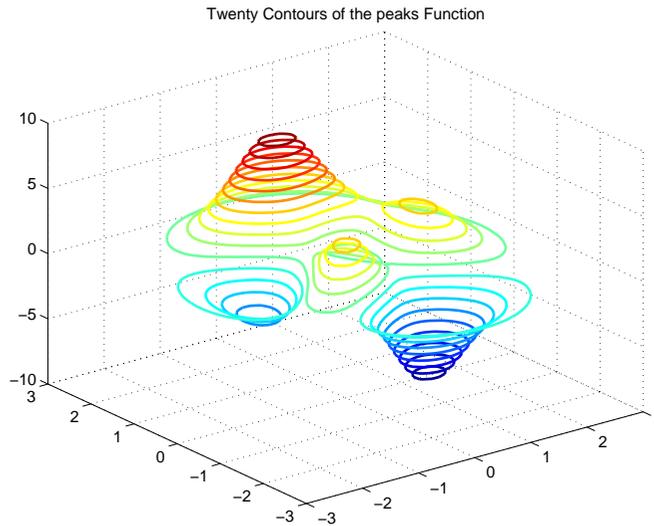
display 20 contours of the peaks function in a 2-D view.



The statements

```
[X,Y,Z] = peaks;  
contour3(X,Y,Z,20)  
h = findobj('Type','patch');  
set(h,'LineWidth',2)  
title('Twenty Contours of the peaks Function')
```

display 20 contours of the peaks function in a 3-D view and increase the line width to 2 points.



Labeling Contours

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

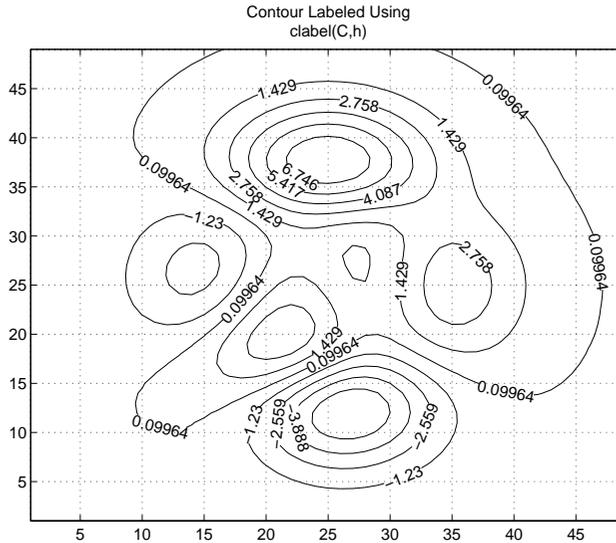
For example, display 10 contour levels of the peaks function,

```
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')
```

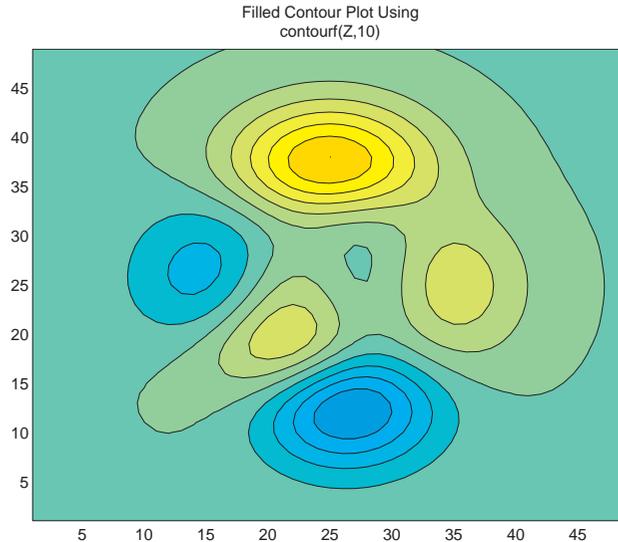
displays a crosshair cursor when your cursor is inside the figure. Pressing any mouse button labels the contour line closest to the center of the crosshair.

Filled Contours

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

```
Z = peaks;  
[C,h] = contourf(Z,10);  
caxis([-20 20])
```

```
title({'Filled Contour Plot Using','contourf(Z,10)'})
```



Drawing a Single Contour Line at a Desired Level

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

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

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

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

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

```
Z = peaks(X,Y);  
contour3(X,Y,Z)
```

To display only one contour level at $Z = 1$, define v as $[1 \ 1]$.

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

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 $v = 0.2$	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
	Columns 8 through 14	3.0000	0	1.0000	1.0359	1.0000	0.2000	1.6669
Three vertices at $v = 0$		1.0002	3.0000	2.9991	2.0000	1.0018	5.0000	3.0000
	Columns 15 through 21	1.2324	2.0000	2.8240	2.3331	0.4000	2.0000	2.6130
		2.0000	1.3629	2.0000	3.0000	5.0000	2.8530	2.0000
	Columns 22 through 28	2.0000	1.4290	2.0000	0.6000	2.0000	2.4020	2.0000
		1.5261	2.0000	2.8530	5.0000	2.5594	2.0000	1.6892
	Columns 29 through 35	1.6255	2.0000	0.8000	2.0000	2.1910	2.0000	1.8221
Five vertices at $v = 0.8$		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;
```

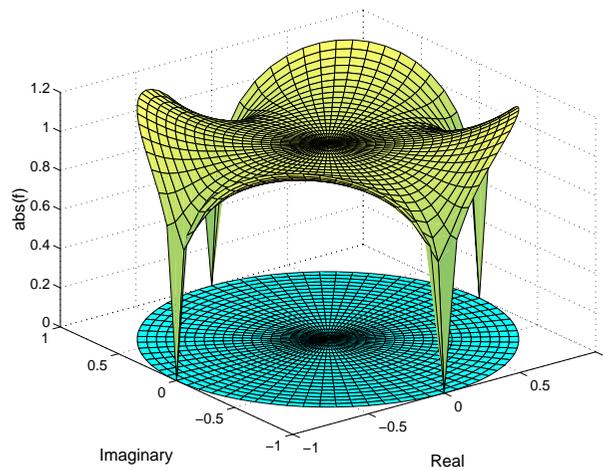
X , Y , and Z are points inside the circle.

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

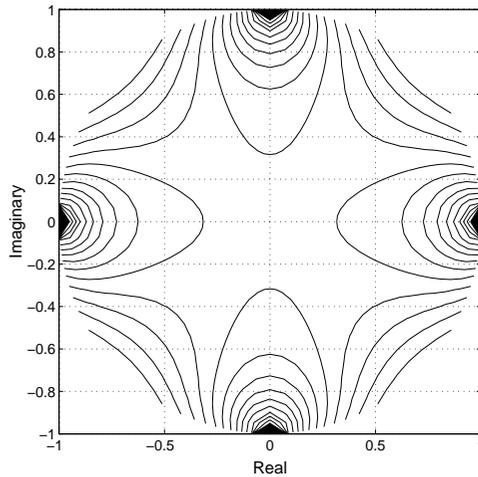
These statements add labels.

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

Contours in Cartesian Coordinates

These statements display a contour of the surface in Cartesian coordinates and label the x - and y -axis.

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



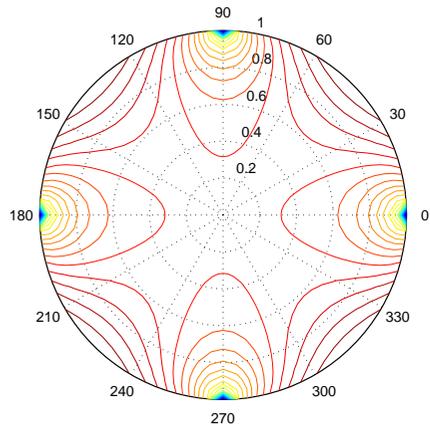
Contours on a Polar Axis

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

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

With hold on, display the contour on the polar grid.

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



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 for more information.

Example – Selecting Plotting Points from the Screen

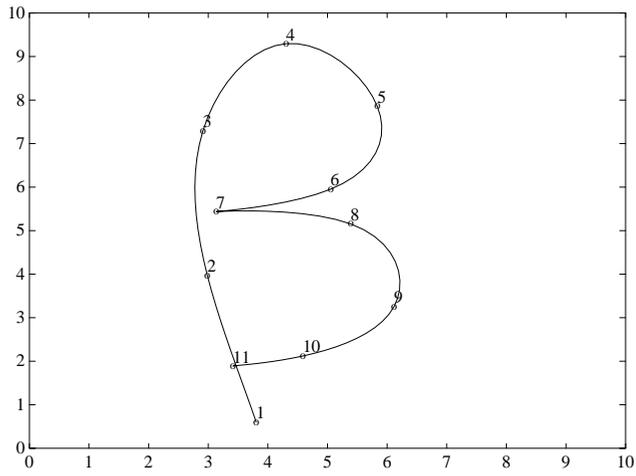
This example illustrates the use of `ginput` with the `spline` function to create a curve by interpolating in two dimensions.

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 $1/10$ of the original spacing.

```
axis([0 10 0 10])
hold on
% Initially, the list of points is empty.
xy = [];
n = 0;
% Loop, picking up the points.
disp('Left mouse button picks points.')
disp('Right mouse button picks last point.')
but = 1;
while but == 1
    [xi,yi,but] = ginput(1);
    plot(xi,yi,'ro')
    n = n+1;
    xy(:,n) = [xi;yi];
end
% Interpolate with a spline curve and finer spacing.
t = 1:n;
ts = 1: 0.1: n;
xys = spline(t,xy,ts);

% Plot the interpolated curve.
plot(xys(1,:),xys(2,:),'b-');
hold off
```

This plot shows some typical output.



Animation

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

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

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

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

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

Movies

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

- Use `getframe` to generate each movie frame.
- Use `movie` to run the movie a specified number of times at the specified rate.

Typically, you use `getframe` in a `for` loop to assemble the array of movie frames. `getframe` returns a structure having the following fields:

- `cdata` – image data in a `uint8` matrix. The matrix has dimensions of height-by-width on indexed-color systems and height-by-width-by-3 on truecolor systems.
- `colormap` – the colormap in an `n-by-3` matrix, where `n` is the number of colors. On truecolor systems, the `colormap` field is empty.

See `image` for more information on images.

Example – Visualizing an FFT as a Movie

This example illustrates the use of movies to visualize the quantity $\text{fft}(\text{eye}(n))$, 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

Create the movie in a for loop calling `getframe` to capture the graph. Since the `plot` command resets the axes properties, call `axis equal` within the loop before `getframe`.

```
for k = 1:16
    plot(fft(eye(k+16)))
    axis equal
    M(k) = getframe;
end
```

Running the Movie

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

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

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

Movies that Include the Entire Figure

If you want to capture the contents of the entire figure window (for example, to include GUI components in the movie), specify the figure's handle as an argument to the `getframe` command. For example, suppose you want to add a slider to indicate the value of k in the previous example.

```
h = uicontrol('style','slider','position',...
    [10 50 20 300],'Min',1,'Max',16,'Value',1)
for k = 1:16
    plot(fft(eye(k+16)))
    axis equal
    set(h,'Value',k)
    M(k) = getframe(gcf);
end
```

In this example, the movie frame contains the entire figure. To play so that it looks like the original figure, make the playback axes fill the figure window.

```
clf
axes('Position',[0 0 1 1])
movie(M,30)
```

Erase Modes

You can select the method MATLAB uses to redraw graphics objects. One event that causes MATLAB to redraw an object is changing the properties of that object. You can take advantage of this behavior to create animated sequences. A typical scenario is to draw a graphics object, then change its position by respecifying the x -, y -, and z -coordinate data by a small amount with each pass through a loop.

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

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

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

Example – Animating with Erase Modes

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

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

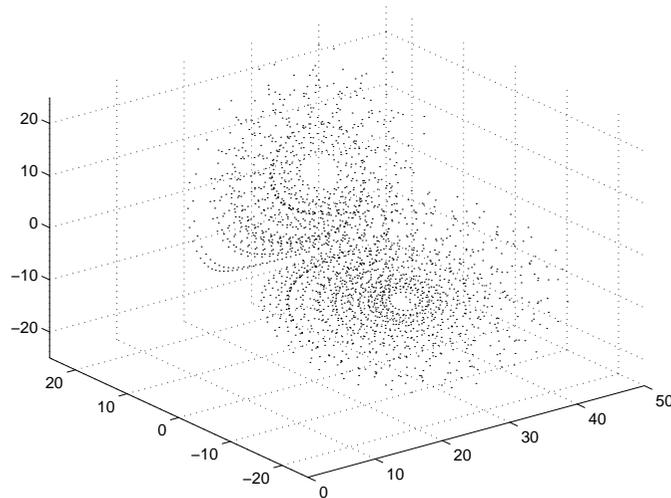
with a vector valued function $y(t)$ and a matrix A , which depends upon y .

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

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 = 0.01;
p = plot3(y(1),y(2),y(3),'.', ...
    'EraseMode','none','MarkerSize',5); % Set EraseMode to none
axis([0 50 -25 25 -25 25])
hold on
for i=1:4000
    A(1,3) = y(2);
    A(3,1) = -y(2);
    ydot = A*y;
    y = y + h*ydot;
    % Change coordinates
    set(p,'XData',y(1),'YData',y(2),'ZData',y(3))
    drawnow
    i=i+1;
end
```

The `plot3` statement sets `EraseMode` to `none`, indicating that the points already plotted should not be erased when the plot is redrawn. In addition, the handle of the plot object is saved. Within the `for` loop, a `set` statement references the plot object and changes its internally stored coordinates for the new location. While this manual cannot show the dynamically evolving output, this picture shows a snapshot.



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

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

Displaying Bit-Mapped Images

Overview (p. 5-2)	File formats and image commands.
Images in MATLAB (p. 5-4)	Specific information about images in MATLAB.
Image Types (p. 5-6)	Types of images supported in MATLAB.
Working with 8-Bit and 16-Bit Images (p. 5-11)	Operations you can perform on nondouble image data.
Reading, Writing, and Querying Graphics Image Files (p. 5-15)	Working with standard image file formats in MATLAB.
Displaying Graphics Images (p. 5-18)	Commands for displaying a matrix as an image.
The Image Object and Its Properties (p. 5-23)	Properties of MATLAB image objects.
Printing Images (p. 5-28)	Printing images in proper proportions.
Converting the Data or Graphic Type of Images (p. 5-29)	Converting between image types.

Overview

MATLAB provides commands for reading, writing, and displaying several types of graphics file formats for images. As with MATLAB-generated images, once a graphics file format image is displayed, it becomes a Handle Graphics image object. MATLAB supports the following graphics file formats:

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

For information concerning the bit depths and image types supported for these formats, see `imread` and `imwrite`.

MATLAB supports three different numeric classes for image display: double-precision floating-point (`double`), 16-bit unsigned integer (`uint16`), and 8-bit unsigned integer (`uint8`). The image display commands interpret data values differently depending on the numeric class the data is stored in.

This chapter discusses the different data and image types you can use, and includes details on how to: read, write, work with, and display graphics images; how to alter the display properties and aspect ratio of an image during display; how to print an image; and how to convert the data type or graphics format of an image.

This table lists the functions discussed in this chapter.

Function	Purpose	Function Group
<code>axis</code>	Plot axis scaling and appearance	Display
<code>image</code>	Display image (create image object)	Display
<code>imagesc</code>	Scale data and display as image	Display
<code>imread</code>	Read image from graphics file	File I/O

Function	Purpose	Function Group
<code>imwrite</code>	Write image to graphics file	File I/O
<code>imfinfo</code>	Get image information from graphics file	Utility
<code>ind2rgb</code>	Convert indexed image to RGB image	Utility

Images in MATLAB

The basic data structure in MATLAB is the *array*, an ordered set of real or complex elements. This object is naturally suited to the representation of *images*, real-valued, ordered sets of color or intensity data. (MATLAB does not support complex-valued images.)

MATLAB stores most images as two-dimensional arrays (i.e., matrices), in which each element of the matrix corresponds to a single pixel in the displayed image. For example, an image composed of 200 rows and 300 columns of different colored dots would be stored in MATLAB as a 200-by-300 matrix. Some images, such as RGB, require a three-dimensional array, where the first plane in the 3rd dimension represents the red pixel intensities, the second plane represents the green pixel intensities, and the third plane represents the blue pixel intensities.

This convention makes working with graphics file format images in MATLAB similar to working with any other type of matrix data. For example, you can select a single pixel from an image matrix using normal matrix subscripting.

```
I(2,15)
```

This command returns the value of the pixel at row 2, column 15 of the image I.

Bit Depth Support

MATLAB supports reading the most commonly used bit depths (bits per pixel) of any of the supported graphics file formats. When the data is in memory, it can be stored as `uint8`, `uint16`, or `double`. For details on which bit depths are appropriate for each supported format, see `imread` and `imwrite`.

Data Types

This section introduces you to the different data types that MATLAB uses to store images. Details on the inner workings of the storage for 8- and 16-bit images are included in “Working with 8-Bit and 16-Bit Images” on page 5-11.

By default, MATLAB stores most data in arrays of class `double`. The data in these arrays is stored as double precision (64-bit) floating-point numbers. All MATLAB functions and capabilities work with these arrays.

For images stored in one of the graphics file formats supported by MATLAB, however, this data representation is not always ideal. The number of pixels in

such an image may be very large; for example, a 1000-by-1000 image has a million pixels. Since each pixel is represented by at least one array element, this image would require about 8 megabytes of memory if it were stored as class `double`.

To reduce memory requirements, MATLAB supports storing image data in arrays of class `uint8` and `uint16`. The data in these arrays is stored as 8-bit or 16-bit unsigned integers. These arrays require one-eighth or one-fourth as much memory as data in `double` arrays.

Image Types

In MATLAB, an image consists of a data matrix and possibly a colormap matrix. Three basic image types are used in MATLAB, each differing in the way that the data matrix elements are interpreted:

- Indexed images
- Intensity (or grayscale) images
- RGB (or truecolor) images

This section discusses how MATLAB represents each of these image types.

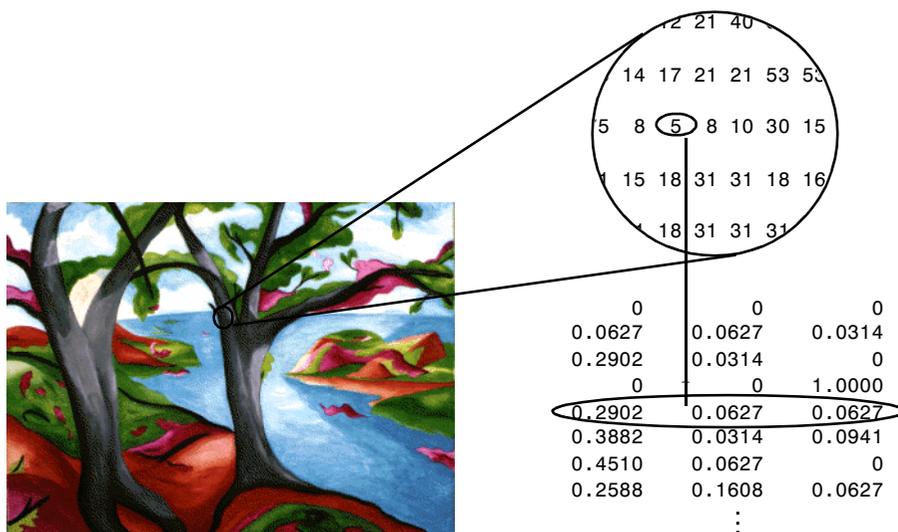
Indexed Images

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

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

A colormap is often stored with an indexed image and is automatically loaded with the image when you use the `imread` function. However, you are not limited to using the default colormap—you can use any colormap that you choose. The description for the property `CDataMapping` describes how to alter the type of mapping used.

The next figure illustrates the structure of an indexed image. The pixels in the image are represented by integers, which are pointers (indices) to color values stored in the colormap.

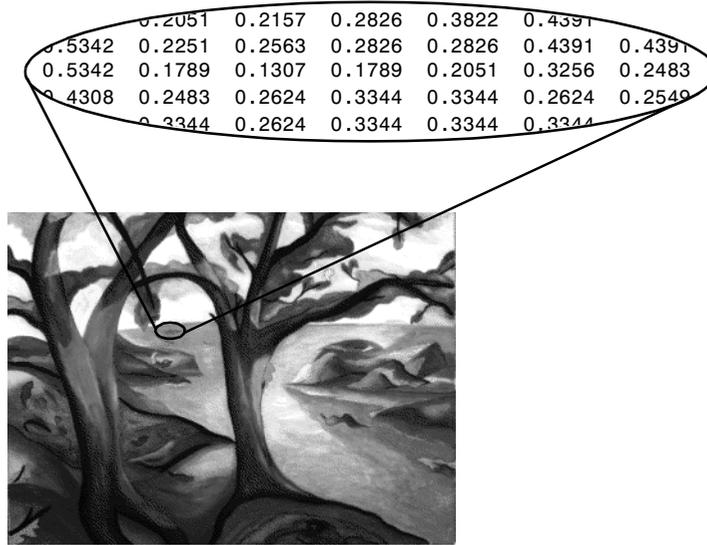


The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8` or `uint16`, there is an offset—the value 0 points to the first row in the colormap, the value 1 points to the second row, and so on. The offset is also used in graphics file formats, to maximize the number of colors that can be supported. In the image above, the image matrix is of class `double`. Because there is no offset, the value 5 points to the fifth row of the colormap.

Intensity Images

An intensity image is a data matrix, I , whose values represent intensities within some range. MATLAB stores an intensity image as a single matrix, with each element of the matrix corresponding to one image pixel. The matrix can be of class `double`, `uint8`, or `uint16`. While intensity images are rarely saved with a colormap, MATLAB uses a colormap to display them. In essence, MATLAB handles intensity images as indexed images.

This figure depicts an intensity image of class `double`.



To display an intensity image, use the `imagesc` (“image scale”) function, which enables you to set the range of intensity values. `imagesc` scales the image data to use the full colormap. Use the two-input form of `imagesc` to display an intensity image. For example,

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

The second input argument to `imagesc` specifies the desired intensity range. The function `imagesc` displays `I` by mapping the first value in the range (usually 0) to the first colormap entry, and the second value (usually 1) to the last colormap entry. Values in between are linearly distributed throughout the remaining colormap colors.

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

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

To display a matrix `A` with an arbitrary range of values as an intensity image, use the single-argument form of `imagesc`. With one input argument, `imagesc` maps the minimum value of the data matrix to the first colormap entry, and

maps the maximum value to the last colormap entry. For example, these two lines are equivalent.

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

RGB (Tricolor) Images

An RGB image, sometimes referred to as a “tricolor” image, is stored in MATLAB as an m -by- n -by-3 data array that defines red, green, and blue color components for each individual pixel. RGB images do not use a palette. The color of each pixel is determined by the combination of the red, green, and blue intensities stored in each color plane at the pixel’s location. Graphics file formats store RGB images as 24-bit images, where the red, green, and blue components are 8 bits each. This yields a potential of 16 million colors. The precision with which a real-life image can be replicated has led to the nickname “tricolor image”.

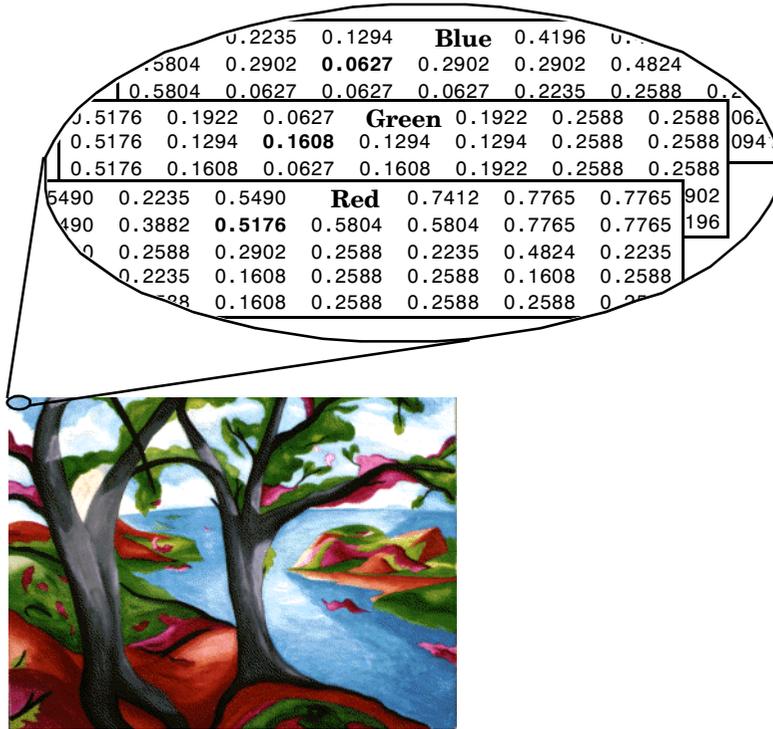
An RGB MATLAB array can be of class `double`, `uint8`, or `uint16`. In an RGB array of class `double`, each color component is a value between 0 and 1. A pixel whose color components are (0,0,0) displays as black, and a pixel whose color components are (1,1,1) displays as white. The three color components for each pixel are stored along the third dimension of the data array. For example, 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 tricolor image `RGB`, use the `image` function. For example,

```
image(RGB)
```

If MATLAB is running on a computer that does not have hardware support for tricolor image display, MATLAB uses color approximation and dithering to display an approximation of the image. See “Dithering Tricolor on Indexed Color Systems” for more information.

The next figure shows an RGB image of class `double`.



To determine the color of the pixel at (2,3), you would look at the RGB triplet stored in (2,3,1:3). Suppose (2,3,1) contains the value 0.5176, (2,3,2) contains 0.1608, and (2,3,3) contains 0.0627. The color for the pixel at (2,3) is

0.5176 0.1608 0.0627

Working with 8-Bit and 16-Bit Images

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 or 16-bit unsigned integers by using the numeric classes `uint8` or `uint16`, respectively. An image whose data matrix has class `uint8` is called an 8-bit image; an image whose data matrix has class `uint16` is called a 16-bit image.

The `image` function can display 8- or 16-bit images directly without converting them to double precision. However, `image` interprets matrix values slightly differently when the image matrix is `uint8` or `uint16`. The specific interpretation depends on the image type.

8-Bit and 16-Bit Indexed Images

If the class of `X` is `uint8` or `uint16`, 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`, `uint8`, or `uint16`.

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

The colormap index offset for `uint8` and `uint16` data is intended to support standard graphics file formats, which typically store image data in indexed form with a 256-entry colormap. The offset allows you to manipulate and display images of this form in MATLAB using the more memory-efficient `uint8` and `uint16` arrays.

Because of the offset, you must add 1 to convert a `uint8` or `uint16` indexed image to `double`. For example,

```
X64 = double(X8) + 1;  
or  
X64 = double(X16) + 1;
```

Conversely, subtract 1 to convert a `double` indexed image to `uint8` or `uint16`.

```
X8 = uint8(X64 - 1);  
or  
X16 = uint16(X64 - 1);
```

The order of operations must be as shown, because most MATLAB mathematical operations cannot be performed on `uint8` and `uint16` arrays.

8-Bit and 16-Bit Intensity Images

Whereas the range of double image arrays is usually $[0, 1]$, the range of 8-bit intensity images is usually $[0, 255]$ and the range of 16-bit intensity images is usually $[0, 65535]$. Use the following command to display an 8-bit intensity image with a gray scale colormap.

```
imagesc(I,[0 255]); colormap(gray);
```

To convert an intensity image from double to `uint16`, first multiply by 65535.

```
I16 = uint16(round(I64*65535));
```

Conversely, divide by 65535 after converting a `uint16` intensity image to double.

```
I64 = double(I16)/65535;
```

8-Bit and 16-Bit RGB Images

The color components of an 8-bit RGB image are integers in the range $[0, 255]$ rather than floating-point values in the range $[0, 1]$. A pixel whose color components are $(255,255,255)$ displays as white. The `image` command displays an RGB image correctly whether its class is `double`, `uint8`, or `uint16`.

```
image(RGB);
```

To convert an RGB image from double to `uint8`, first multiply by 255.

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

Conversely, divide by 255 after converting a `uint8` RGB image to double.

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

To convert an RGB image from double to `uint16`, first multiply by 65535.

```
RGB16 = uint16(round(RGB64*65535));
```

Conversely, divide by 65535 after converting a `uint16` RGB image to double.

```
RGB64 = double(RGB16)/65535;
```

Mathematical Operations Support for uint8 and uint16

The following MATLAB mathematical operations support uint8 and uint16 data: `conv2`, `convn`, `fft2`, `fftn`, `sum`. In these cases, the output is always double.

If you attempt to perform an unsupported operation on one of these arrays, you will receive an error. For example,

```
BW3 = BW1 + BW2
??? Function '+' not defined for variables of class 'uint8'.
```

Most of the functions in the Image Processing Toolbox accept uint8 and uint16 input. If you plan to do sophisticated image processing on uint8 or uint16 data, you should consider adding the Image Processing Toolbox to your MATLAB computing environment.

Other 8-Bit and 16-Bit Array Support

MATLAB supports several other operations on uint8 and uint16 arrays, including:

- Reshaping, reordering, and concatenating arrays using the functions `reshape`, `cat`, `permute`, and the `[]` and `'` operators
- Saving and loading uint8 and uint16 arrays in MAT-files using `save` and `load`. (Remember that if you are loading or saving a graphics file format image, you must use the commands `imread` and `imwrite`, instead.)
- Locating the indices of nonzero elements in uint8 and uint16 arrays using `find`. However, the returned array is always of class `double`.
- Relational operators

Summary of Image Types and Numeric Classes

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

Image Type	double Data	uint8 or uint16 Data
Indexed	Image is an m -by- n array of integers in the range $[1, p]$. Colormap is a p -by-3 array of floating-point values in the range $[0, 1]$.	Image is an m -by- n array of integers in the range $[0, p - 1]$. Colormap is a p -by-3 array of floating-point values in the range $[0, 1]$.
Intensity	Image is an m -by- n array of floating-point values that are linearly scaled by MATLAB to produce colormap indices. The typical range of values is $[0, 1]$. Colormap is a p -by-3 array of floating-point values in the range $[0, 1]$ and is typically grayscale.	Image is an m -by- n array of integers that are linearly scaled by MATLAB to produce colormap indices. The typical range of values is $[0, 255]$ or $[0, 65535]$. Colormap is a p -by-3 array of floating-point values in the range $[0, 1]$ and is typically grayscale.
RGB (Truicolor)	Image is an m -by- n -by-3 array of floating-point values in the range $[0, 1]$.	Image is an m -by- n -by-3 array of integers in the range $[0, 255]$ or $[0, 65535]$.

Reading, Writing, and Querying Graphics Image Files

In its native form, a graphics file format image is not stored as a MATLAB matrix, or even necessarily as a matrix. Most graphics files begin with a header containing format-specific information tags, and continue with bitmap data that can be read as a continuous stream. For this reason, you cannot use the standard MATLAB I/O commands `load` and `save` to read and write a graphics file format image.

MATLAB provides special functions for reading and writing image data from graphics file formats. To read a graphic file format image use `imread`; to write a graphic file format image, use `imwrite`; to obtain information about the nature of a graphics file format image, use `imfinfo`.

This table gives a clearer picture of which MATLAB commands should be used with which image types.

Procedure	Function(s) to Use
Load or Save a Matrix as a MAT-file	<code>load</code> <code>save</code>
Load or Save Graphics File Format Image, e.g. BMP, TIFF	<code>imread</code> <code>imwrite</code>
Display Any Image Loaded Into MATLAB	<code>image</code> <code>imagesc</code>
Utilities	<code>imfinfo</code> <code>ind2rgb</code>

Reading a Graphics Image

The function `imread` reads an image from any supported graphics image file in any of the supported bit depths. Most of the images that you will read are 8-bit. When these are read into memory, MATLAB stores them as class `uint8`. The main exception to this rule is that MATLAB supports 16-bit data for PNG and TIFF images. If you read a 16-bit PNG or TIFF image, it will be stored as class `uint16`.

Note For indexed images, `imread` always reads the colormap into an array of class `double`, even though the image array itself may be of class `uint8` or `uint16`.

For our discussion here we will show one of the most basic syntax uses of `imread`. This code reads the image `ngc6543a.jpg`.

```
RGB = imread('ngc6543a.jpg');
```

You can write (save) image data using the `imwrite` function. The statements

```
load clown
imwrite(X,map,'clown.bmp')
```

create a BMP file containing the clown image.

Writing a Graphics Image

When you save an image using `imwrite`, the default behavior is to automatically reduce the bit depth to `uint8`. Many of the images used in MATLAB are 8-bit, and most graphics file format images do not require double-precision data. One exception to the MATLAB rule for saving the image data as `uint8` is that PNG and TIFF images may be saved as `uint16`. Since these two formats support 16-bit data, you may override the MATLAB default behavior by specifying `uint16` as the data type for `imwrite`. The following example shows writing a 16-bit PNG file using `imwrite`.

```
imwrite(I,'clown.png','BitDepth',16);
```

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:

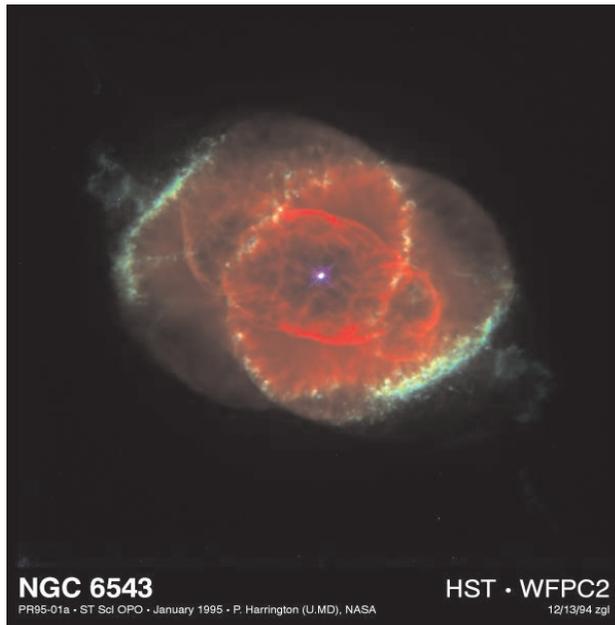
- Name of the file, including the directory path if the file is not in the current directory
- File format
- Version number of the file format

- File modification date
- File size in bytes
- Image width in pixels
- Image height in pixels
- Number of bits per pixel
- Image type: RGB (truecolor), intensity (grayscale), or indexed

Displaying Graphics Images

To display a graphics file image, use either `image` or `imagesc`. For example, assuming `RGB` is an image,

```
figure('Position',[100 100 size(RGB,2) size(RGB,1)]);  
image(RGB); set(gca,'Position',[0 0 1 1])
```



(This image was created with 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. orkowski (University of Maryland), and NASA.)

Summary of Image Types and Display Methods

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

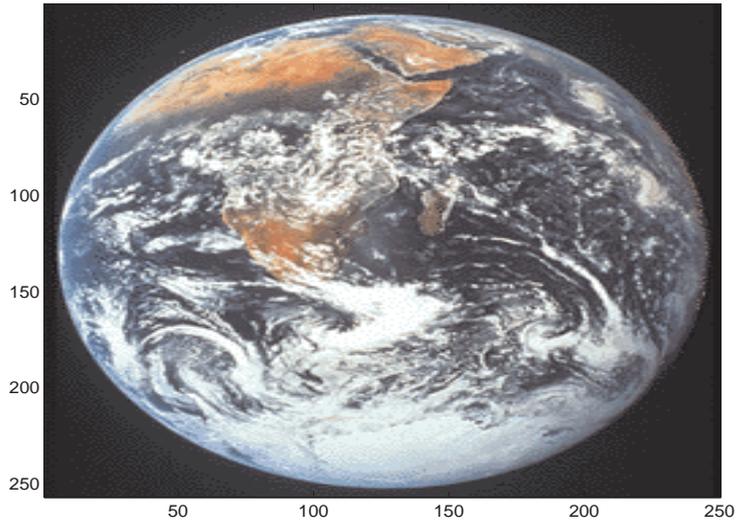
Image Type	Display Commands	Uses Colormap Colors
Indexed	<code>image(X); colormap(map)</code>	Yes
Intensity	<code>imagesc(I,[0 1]); colormap(gray)</code>	Yes
RGB (truecolor)	<code>image(RGB)</code>	No

Controlling Aspect Ratio and Display Size

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

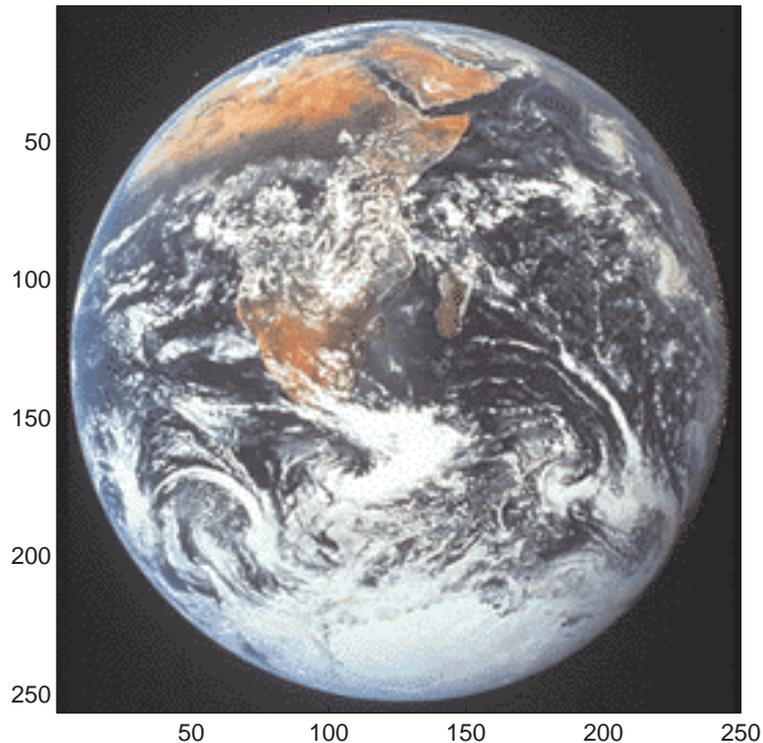
For example, these commands display the earth image in the `demos` directory using the default figure and axes positions.

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



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

```
axis image
```



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

Sometimes you 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 shown.



The Image Object and Its Properties

The commands `image` and `imagesc` create image objects. Image objects are children of axes objects, as are line, patch, surface, and text objects. Like all Handle Graphics objects, the image object has a number of properties you can set to fine-tune its appearance on the screen. The most important properties of the image object with respect to appearance are `CData`, `CDataMapping`, `XData`, `YData`, and `EraseMode`. For detailed information about these and all of the properties of the image object, please see `image`.

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

CDataMapping

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

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

direct
```

Intensity images are displayed by setting the `CDataMapping` property to `'scaled'`. In this case the `CData` values are linearly scaled to form `colormap`

indices. The scale factors are controlled by the axes `CLim` property. The `imagesc` function creates an image object whose `CDataMapping` property is set to 'scaled', and it also adjusts the `CLim` property of the parent axes. For example,

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

scaled

get(gca,'CLim')
ans =

[0 1]
```

XData and YData

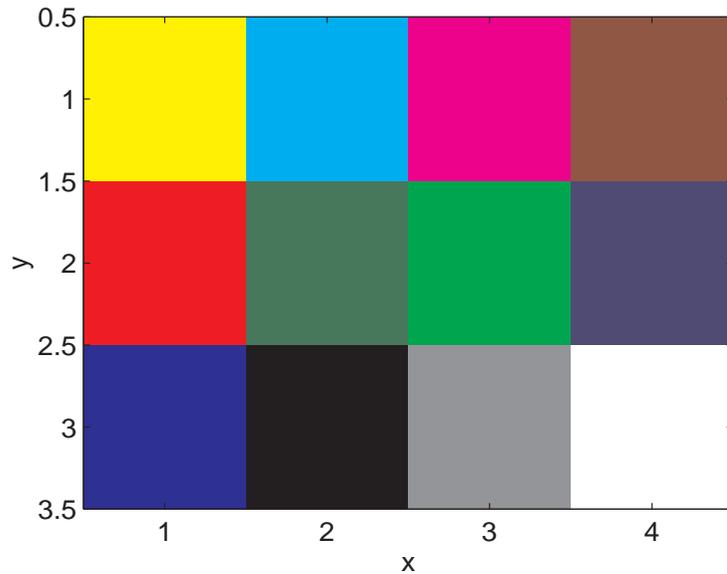
The `XData` and `YData` properties control the coordinate system of the image. For an m -by- n image, the default `XData` is `[1 n]` and the default `YData` is `[1 m]`. These settings imply the following:

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

For example, the statements,

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

produce the picture.



The XData and YData properties of the resulting image object have the default values shown below.

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

```
1 4
```

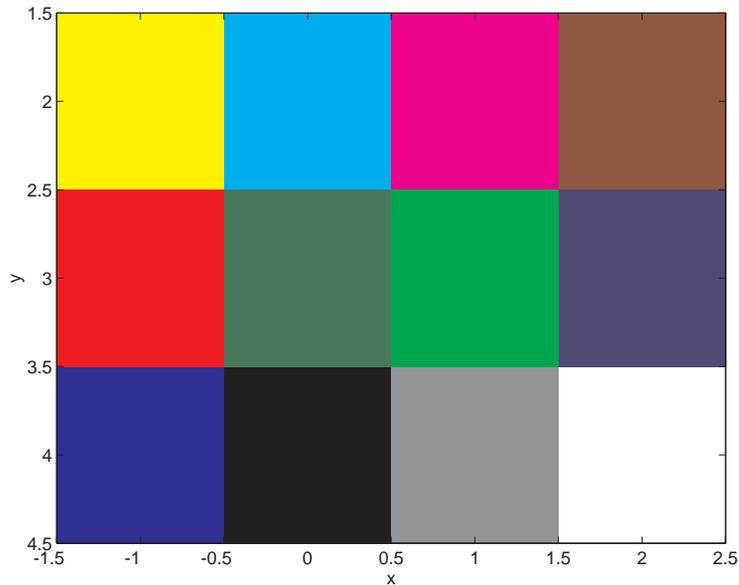
```
get(h,'YData')
ans =
```

```
1 3
```

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

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

produce the picture.



EraseMode

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

You can achieve fast and visually smooth updates of displayed images as you change the image `CData` by setting the image object `EraseMode` property to `'none'`. With this setting, MATLAB does not take the time to erase the displayed image — it immediately draws the updated image when the `CData` changes.

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

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

Rather than creating a new image object each time through the loop, this code simply changes the `CData` of the image object (which was created on the first line using the `image` command). 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.

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')
```

Converting the Data or Graphic Type of Images

Sometimes you will want to perform operations that are not supported for `uint8` or `uint16` arrays. To do this, convert the data to double precision using the `double` function. For example,

```
BW3 = double(BW1) + double(BW2);
```

Keep in mind that converting between data types changes the way MATLAB and the toolbox interpret the image data. If you want the resulting array to be interpreted properly as image data, you need to rescale or offset the data when you convert it. (See the earlier sections “Image Types” and “8-Bit and 16-Bit Indexed Images” for more information about offsets.)

For certain operations, it is helpful to convert an image to a different image type. For example, if you want to filter a color image that is stored as an indexed image, you should first convert it to RGB format. To do this efficiently, use the `ind2rgb` function. (which originated in the Image Processing Toolbox). When you apply the filter to the RGB image, MATLAB filters the intensity values in the image, as is appropriate. If you attempt to filter the indexed image, MATLAB simply applies the filter to the indices in the indexed image matrix, and the results may not be meaningful.

You can also perform certain conversions just using MATLAB syntax. For example, if you want to convert a grayscale image to RGB, you can concatenate three copies of the original matrix along the third dimension.

```
RGB = cat(3,I,I,I);
```

The resulting RGB image has identical matrices for the red, green, and blue planes, so the image displays as shades of gray.

Sometimes you will want to change the graphics format of an image, perhaps for compatibility with another software product. This process is very straightforward. For example, to convert an image from a BMP to a PNG, load the BMP using `imread`, set the data type to `uint8`, `uint16`, or `double`, and then save the image using `imwrite`, with 'PNG' specified as your target format. See `imread` and `imwrite` for the specifics of which bit depths are supported for the different graphics formats, and for how to specify the format type when writing an image to file.

Printing and Exporting

Overview of Printing and Exporting (p. 6-2)	Introduction to basic operations, interfaces, parameters, and defaults associated with printing and exporting
How to Print or Export (p. 6-9)	Step-by-step instructions for printing a figure to a printer or to a file, and for exporting a figure to a graphics-format file or to the clipboard
Examples of Basic Operations (p. 6-23)	Examples that provide you with the information you need to submit a simple print or export job
Changing a Figure's Settings (p. 6-31)	How to change the default settings for parameters, such as figure size, paper orientation, background color, and rendering method
Choosing a Graphics Format (p. 6-57)	Factors to consider when choosing a graphics format for exporting to a file, and information about commonly used formats
Choosing a Printer Driver (p. 6-68)	Factors to consider when using a nondefault print driver, and information specific to drivers supported by MATLAB
Troubleshooting (p. 6-78)	Solutions to frequently asked questions and common problems encountered while printing or exporting graphics

Overview of Printing and Exporting

This section is an introduction to the graphics printing capabilities provided with MATLAB and how to make use of them. It covers

- “Print and Export Operations”
- “Graphical User Interfaces”
- “Command Line Interface” on page 6-3
- “Specifying Parameters and Options” on page 6-5
- “Default Settings and How to Change Them” on page 6-6

Print and Export Operations

There are four basic operations that you can perform in printing or transferring figures you’ve created in MATLAB.

Operation	Description
Print	Send a figure from the screen directly to the printer.
Print to File	Write a figure to a PostScript file to be printed later.
Export to File	Export a figure in graphics format to a file, so that you can import it into an application.
Export to Clipboard	Copy a figure to the Windows clipboard, so that you can paste it into an application.

Graphical User Interfaces

You interact with the MATLAB print and export tools using either Microsoft Windows or UNIX graphical user interfaces or with MATLAB commands. The table below lists the dialog boxes you need to print and export and summarizes how to open them from the figure window.

Dialog Box	How to Open	Description
Print (Windows and UNIX)	File->Print or <code>printdlg</code> function	Send figure to the printer, select the printer, print to file, and several other options
Printing Options	Click Options on UNIX Print dialog	Set some of the most commonly used print settings (UNIX only)
Page Setup	File->Page Setup or <code>pagesetupdlg</code> function	Set properties to be associated with the figure when printed or exported
Print Preview	File->Print Preview or <code>printpreview</code> function	View and adjust the final output
ExZport	File->Export	Export the figure in graphics format to a file
Copy Options	Edit->Copy Options	Set format, figure size, and background color for Copy to Clipboard
Figure Copy Template	File->Preferences	Change text, line, axes, and UI control properties

You can open the **Print**, **Page Setup**, and **Print Preview** dialog boxes from a program or from the command line with the `printdlg`, `pagesetupdlg`, and `printpreview` functions.

Command Line Interface

You can print a MATLAB figure from the command line or from a program. Use the `set` function to set the properties that control how the printed figure looks. Use the `print` function to start the print or export operation.

Modifying Properties with set

The `set` function changes the values of properties that control the look of a figure. These properties are stored with the figure. When you change one of the properties, the new value is saved with the figure and affects the look of the figure each time you print it until you change the setting again.

To change the print properties of the current figure, the `set` command has the form

```
set(gcf, 'Property1', value1, 'Property2', value2, ...)
```

where `gcf` is a call that returns the handle of the current figure, and each property-value pair consists of a named property followed by the value to which the property is set.

For example,

```
set(gcf, 'PaperUnits', 'centimeters', 'PaperType', 'A4', ...)
```

sets the units of measure and the paper size. “Changing a Figure’s Settings” on page 6-31 describes commonly used print properties. The Figure Properties reference page contains a complete list of the properties.

Examining Properties with get

You can also use the `get` function to retrieve the value of a specific property.

```
a = get(gcf, 'Property')
```

Printing and Exporting with print

The `print` function performs any of the four actions shown in the table below. You control what action is taken, depending on the presence or absence of certain arguments.

Action	Print Command
Print a figure to a printer	<code>print</code>
Print a figure to a file for later printing	<code>print filename</code>

Action	Print Command
Copy a figure in graphics format to the clipboard	<code>print -dfileformat</code>
Export a figure to a graphics format file that you can later import into an application	<code>print -dfileformat filename</code>

You can also include optional arguments with the `print` command. For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, use

```
print -f2 -r600 -depsc spline2d
```

The functional form of this command is

```
print('-f2', '-r600', '-depsc', 'spline2d');
```

Specifying Parameters and Options

The table below lists parameters you can modify for the figure to be printed or exported. To change one of these parameters, use the **Page Setup** or **Printing Options** (UNIX only) dialog boxes, or use the `set` or `print` function.

See “Changing a Figure’s Settings” on page 6-31 for more detailed instructions.

Parameter	Description
Figure size	Set the size of the figure on the printed page
Figure position	Set the position of the figure on the printed page
Paper size	Select printer paper, specified by dimension or type
Paper orientation	Specify the way the figure is oriented on the page
Position mode	Specify the figure position yourself or have MATLAB determine position automatically
Graphics format	Select the format for exported data (e.g., EPS, JPEG)
Resolution	Specify how finely your figure is to be sampled

Parameter	Description
Renderer	Select the software that processes your graphics data
Renderer mode	Specify the renderer yourself or have MATLAB determine which renderer to use automatically
Axes tick marks	Keep axes tick marks and limits as shown or have MATLAB adjust depending on figure size
Background color	Keep background color as shown on the screen or force it to white
Line and text color	Keep line and text objects as shown on screen or print them in black and white
UI controls	Show or hide all user interface controls in the figure
Bounding box	Leave space between outermost objects in the plot and the edges of its background area
CMYK	Automatically convert RGB values to CMYK values
Character set encoding	Select character set for PostScript printers

Default Settings and How to Change Them

If you have not changed the default print and export settings, MATLAB prints or exports the figure:

- 8-by-6 inches with no window frame
- Centered on portrait format 8.5-by-11 inch paper if available
- Using white for the figure and axes background color
- With the ticks and limits of the axes scaled to accommodate the printed size

Setting Defaults for a Figure

In general, to change the property settings for a specific figure, follow the instructions given in the section, “Changing a Figure’s Settings” on page 6-31.

Any settings you change with the **Page Setup**, **Print**, and **Printing Options** dialog boxes, or with the `set` function are saved with the figure and affect each printing of the figure until you change the settings again.

The settings you change with the **Figure Copy Template Preferences** and **Copy Options Preferences** panels alter the figure as it displays on the screen.

Setting Defaults for the Session

MATLAB enables you to set the session defaults for figure properties. Set the session default for a property using the syntax

```
set(0, 'DefaultFigure $propertyname$ ', 'value')
```

where *propertyname* is one of the named figure properties. This example sets the paper orientation for all subsequent print operations in the current MATLAB session.

```
set(0, 'DefaultFigurePaperOrientation', 'landscape')
```

The Figure Properties reference page contains a complete list of the properties.

Setting Defaults Across Sessions

MATLAB enables you to set the session-to-session defaults for figure properties, the print driver, and the print function.

Print Device and Print Command. Set the default print driver and the default print command in your `printopt.m` file. This file contains instructions for changing these settings and for displaying the current defaults. Open `printopt.m` in your editor by typing the command

```
edit printopt
```

Scroll down about 40 lines until you come to this comment line and make your changes after this line.

```
%--> Put your own changes to the defaults here (if needed)
```

For example, to change the default driver, first find the line that sets `dev`, and then replace the text string with an appropriate value. So, to set the default driver to HP LaserJet III, modify the line to read,

```
dev = '-dljet3';
```

For the full list of values for `dev`, see the “Drivers” section of the `print` reference page.

Note If you set `dev` to be a graphics format, such as `-djpeg`, MATLAB exports the figure rather than printing it.

Figure Properties. Set the session-to-session default for a property by including this command in your `startup.m` file:

```
set(0, 'DefaultFigurepropertyname', 'value')
```

where *propertyname* is one of the named figure properties. For example,

```
set(0, 'DefaultFigureInvertHardcopy', 'off')
```

keeps the figure background in the screen color.

This is the same command you use to change a session default, except that it executes automatically every time MATLAB is invoked.

Note Arguments specified with the `print` command override properties set using MATLAB commands or the **Page Setup** dialog box, which in turn override any MATLAB default settings specified in `printopt.m` or `startup.m`.

How to Print or Export

This section covers the following topics to show you the steps you need to take to produce a printed or exported figure:

- “Printing a Figure”
- “Printing to a File” on page 6-13
- “Exporting to a File” on page 6-15
- “Exporting to the Windows Clipboard” on page 6-20

Before you print or export a figure, preview the image to be exported by selecting **Print Preview** from the figure window’s **File** menu. Use the **Page Setup** dialog box or the `set` function to adjust the look of the exported figure. See “Changing a Figure’s Settings” on page 6-31 for details.

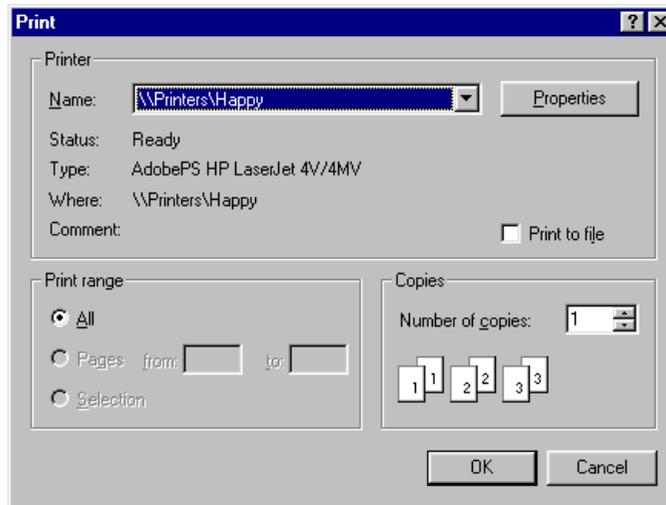
Printing a Figure

This section tells you how to print your figure to a printer:

- “Using the Graphical User Interface on Windows” on page 6-10
- “Using the Graphical User Interface on UNIX” on page 6-11
- “Using MATLAB Commands” on page 6-13

Using the Graphical User Interface on Windows

MATLAB for Windows uses the standard Windows **Print** dialog box, which normally comes with Windows software products. To open the Windows **Print** dialog box, select **Print** from the figure window's **File** menu.



- To print a figure, first select a printer from the list box, then click **OK**.
- To save it to a file, click the **Print to file** check box, click on **OK**, and when the **Print to File** window appears, enter the filename you want to save the figure to. MATLAB creates the file in your current working directory.

Settings you can change in the Windows **Print** dialog box are as follows:

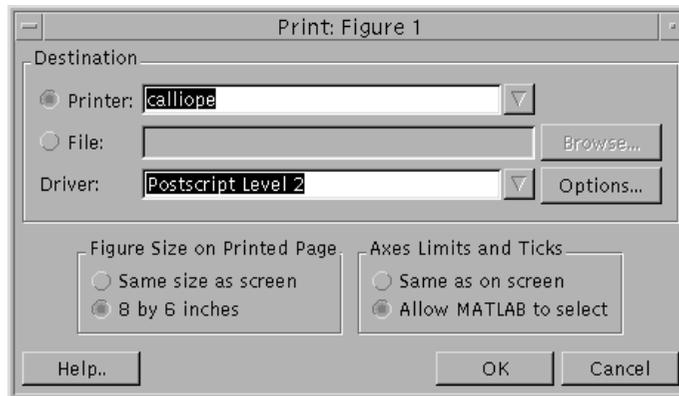
Properties. To make changes to settings specific to a printer, click the **Properties** button. This opens the Windows **Document Properties** window.

Print range. You can only select **All** in this panel. The selection does not affect your printed output.

Copies. Enter the number of copies you want to print.

Using the Graphical User Interface on UNIX

MATLAB for UNIX has a **Print** dialog box and an associated **Printing Options** dialog box. To open the **Print** dialog box, select **Print** from the figure window's **File** menu.



- To print a figure, click the **Printer** button and select a printer from the list box. You can select a driver from the **Driver** list box if you don't want the default driver.
- To save it to a file, click the **File** button, enter a filename, and browse for the directory you want the file saved in.

Settings you can change in the UNIX **Print** dialog box are as follows:

Figure Size on Printed Page. If you want the printed plot to have the same size as it does on your screen, select **Same size as screen**. If you want the printed output to have the default size of 8-by-6 inches, select **8 by 6 inches**.

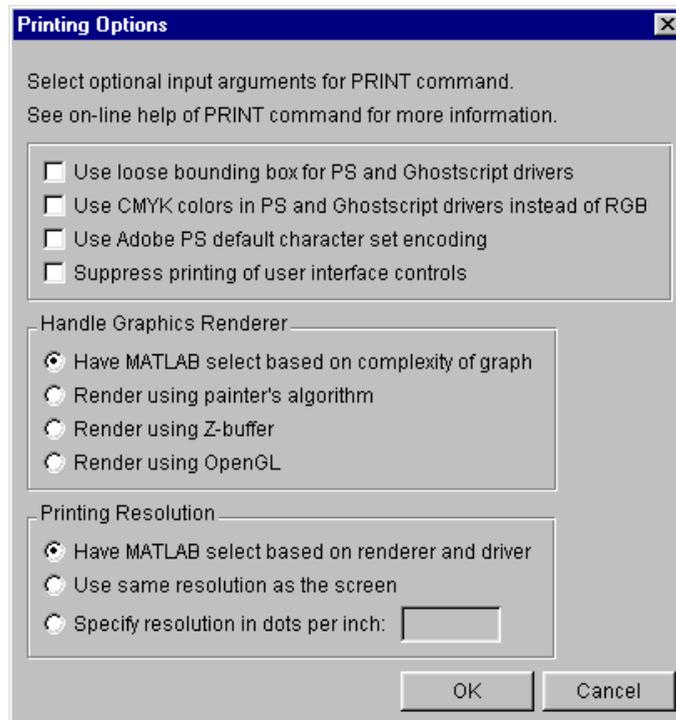
See “Setting the Figure Size and Position” on page 6-34 for more information.

Axes Limits and Ticks. To force MATLAB to print the same number of ticks and the same limit values for the axes as are used on the screen, select **Same as on screen**. To let MATLAB scale the limits and ticks of the axes based on the size of the printed figure, select **Allow MATLAB to select**.

See “Setting the Axes Ticks and Limits” on page 6-47 for more information.

UNIX Printing Options Dialog Box

To open the **UNIX Printing Options** dialog box, click the **Options** button on the **UNIX Print** dialog box (see figure shown above).



Settings you can change in the **Printing Options** dialog box are as follows:

Use loose bounding box for PS and Ghostscript drivers. Select this box to leave a little space between the outermost objects in the plot and the edges of the plot's background.

Use CMYK colors in PS and Ghostscript drivers instead of RGB. Select this box to produce output in CMYK (cyan, magenta, yellow, black) color space instead of RGB (red, green blue). This is for PostScript printers and drivers only.

Use Adobe PS default character set encoding. Select this box to have MATLAB use the default character set that is supported by all PostScript printers. This option is provided because some early PostScript printers do not support the

PostScript operator `ISOLatin1Encoding`, which MATLAB uses when it generates PostScript files. If your printer does not support this operator, you may notice problems in the text of MATLAB printouts.

Suppress printing of user interface controls. Select this box to prevent any user interface controls that you added to the plot from appearing in the printed plot.

Handle Graphics Renderer. Select one of the radio buttons to have MATLAB select the renderer, or to select a specific renderer from those that MATLAB supports. See “The Default Renderer for MATLAB” on page 6-43 for information on how MATLAB selects a renderer.

Printing Resolution. Select one of the radio buttons to either have MATLAB select the resolution for your printout, to set the resolution to that used for the screen display, or to enter a specific resolution value. See “Default Resolution and When You Can Change It” on page 6-45 for information on how MATLAB selects a resolution setting.

Using MATLAB Commands

Use the `print` function to print from the MATLAB command line or from a program. See “Printing and Exporting with `print`” on page 6-4 for more information.

To send the current or most recently active figure to a printer, simply type

```
print
```

The Printing Options table on the `print` reference page shows a full list of options that you can use with the `print` function. For example, the following command prints Figure No. 2 with 600 dpi resolution, using the Canon BubbleJet BJ200 printer driver:

```
print -f2 -r600 -dbj200
```

Printing to a File

Instead of sending your figure to the printer right now, you have the option of “printing” it to a file, and then sending the file to the printer later on. You can also append additional figures to the same file using the `print` command.

This section tells you how to save your figure to a file

- “Using the Graphical User Interface on Windows”
- “Using the Graphical User Interface on UNIX”
- “Using MATLAB Commands”

Using the Graphical User Interface on Windows

- 1 To open the **Print** dialog box, select **Print** from the figure window’s **File** menu.
- 2 Select the check box labeled **Print to file**, and click on the **OK** button.
- 3 The **Print to File** dialog box appears, allowing you to specify the output directory and filename.

Using the Graphical User Interface on UNIX

- 1 To open the **Print** dialog box, select **Print** from the figure window’s **File** menu.
- 2 Select the radio button labeled **File**, and either fill in or browse for the directory and filename.

Using MATLAB Commands

To print the figure to a PostScript file, type

```
print filename
```

If you don’t specify the filename extension, MATLAB uses an extension that is appropriate for the print driver being used.

You can also include an `-options` argument when printing to a file. For example, to append the current figure to an existing file, type

```
print -append filename
```

The only way to append to a file is by using the `print` function. There is no dialog box that enables you to do this.

Note If you print a figure to a file, the file can only be printed and cannot be imported into another application. If you want to create a figure file that you can import into an application, see the next section, “Exporting to a File.”

Appending Additional Figures to a File. Once you have printed one figure to a PostScript file, you can append other figures to that same file using the `-append` option of the `print` function. You can only append using the `print` function.

This example prints Figure No. 2 to PostScript file, `myfile.ps`, and then appends Figure No. 3 to the end of the same file.

```
print -f2 myfile
print -f3 -append myfile
```

Exporting to a File

Export a figure in a graphics format to a file if you want to import it into another application, such as a word processor. You can export to a file from the Windows or UNIX **Export** dialog box or from the command line.

This section tells you how to export your figure to a file

- “Using the Graphical User Interface” on page 6-16
- “Using MATLAB Commands” on page 6-16

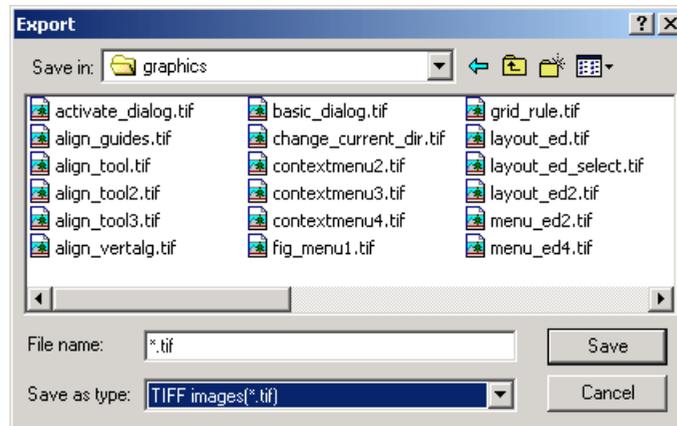
It also covers

- “Exporting with `getframe`” on page 6-17
- “Saving Multiple Figures to an AVI File” on page 6-18
- “Importing MATLAB Graphics into Other Applications” on page 6-18

For further information, see “Choosing a Graphics Format” on page 6-57.

Using the Graphical User Interface

Use the **Export** dialog box to select a file format, specify a filename, and locate the directory to which you want to save your file. To open the **Export** dialog box, select **Export** from the figure window's **File** menu.



Select the graphics file format you want the exported figure to have using the **Save as type** list box. For information on the graphics file formats supported by MATLAB, see “Choosing a Graphics Format” on page 6-57.

Using MATLAB Commands

Use the print function to print from the MATLAB command line or from a program. See “Printing and Exporting with print” on page 6-4 for basic information on printing from the command line.

To export the current or most recently active figure, type

```
print -dfileformat filename
```

where `fileformat` is a graphics format supported by MATLAB and `filename` is the name you want to give to the export file. MATLAB selects the filename extension, if you don't specify it.

You can also specify a number of options with the print function. These are shown in the Printing Options table on the print reference page.

For example, to export Figure No. 2 to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, type

```
print -f2 -r600 -depsc spline2d
```

Graphics file formats are explained in more detail in the sections, “Choosing a Graphics Format” on page 6-57 and “Description of Selected Graphics Formats” on page 6-63.

Exporting with `getframe`

You can use the `getframe` function with `imwrite` to export a graphic. `getframe` is often used in a loop to get a series of frames (figures) with the intention of creating a movie.

Some of the benefits of using this export method over using `print` are

- You can use `getframe` to capture a portion of the figure, rather than the whole figure.
- `imwrite` offers greater flexibility for setting format-specific options, such as the bit depth and compression.

The drawbacks of using this method are that `imwrite` uses built-in MATLAB formats only. Therefore, you will not have access to the Ghostscript formats available to you when exporting with the `print` function or **Export** menu. Also, this technique is limited to screen resolution.

How to Use `getframe` and `imwrite`. Use `getframe` to capture a figure and `imwrite` to save it to a file. `getframe` returns a structure containing the fields `cdata` and `colormap`. The `colormap` field is empty on true color displays. The following example captures the current figure and exports it to a PNG file.

```
I = getframe(gcf);  
imwrite(I.cdata, 'myplot.png');
```

You should use the proper syntax of `imwrite` for the type of image captured. In the example above, the image is captured from a true color display. Since the `colormap` field is empty, it is not passed to `imwrite`.

Example — Exporting a Figure Using `getframe` and `imwrite`. This example offers device independence — it will work for either RGB-mode or indexed-mode monitors.

```
X=getframe(gcf);  
if isempty(X.colormap)  
    imwrite(X.cdata, 'myplot.bmp')  
else  
    imwrite(X.cdata, X.colormap, 'myplot.tif')  
end
```

For information about available file formats and format-specific options, see the `imwrite` reference page. For information about creating a movie from a series of frames, see the reference pages for `getframe` and `movie`, or see “Movies” in Chapter 4, “Creating Specialized Plots.”

Saving Multiple Figures to an AVI File

You can also save multiple figures to an AVI file using the MATLAB `avifile` and `addframe` functions. AVI files can be used for animated sequences and do not need MATLAB to run, but do require an AVI viewer. For more information, see “Exporting MATLAB Graphs in AVI Format,” in the “Development Environment” section of the Using MATLAB documentation.

Importing MATLAB Graphics into Other Applications

You can include MATLAB graphics in a wide variety of applications for word processing, slide preparation, modification by a graphics program, presentation on the Internet, and so on. In general, the process is the same for all applications:

- 1** Use MATLAB to create the figure you want to import into another application.
- 2** Export the MATLAB figure to one of the supported graphics file formats, selecting a format that is both appropriate for the type of figure and supported by the target application. See “Choosing a Graphics Format” on page 6-57 for help.
- 3** Use the import features of the target application to import the graphics file.

Edit Before You Export. Vector graphics may be fully editable in a few high-end applications, but most applications do not support editing beyond simple resizing. Bitmaps cannot be edited with quality results unless you use a software package devoted to image processing. In general, you should try to make all the necessary settings while your figure is still in MATLAB.

Importing into Microsoft Applications. To import your exported figure into a Microsoft application, select **Picture** from the **Insert** menu. Then select **From File** and navigate to your exported file. If you use the clipboard to perform your export operations, you can take advantage of the recommended MATLAB settings for Word and PowerPoint.

Example — Importing an EPS Graphic into LaTeX. This example shows how to import an EPS file named `peaks.eps` into LaTeX.

```
\documentclass{article}

\usepackage{graphicx}

\begin{document}

\begin{figure}[h]
\centerline{\includegraphics[height=10cm]{peaks.eps}}
\caption{Surface Plot of Peaks}
\end{figure}

\end{document}
```

EPS graphics can be edited after being imported to LaTeX. For example, you can specify the height in any LaTeX-compatible dimension. To set the height to 3.5 inches, use the command

```
height=3.5in
```

You can use the `angle` function to rotate the graph. For example, to rotate the graph 90 degrees, add

```
angle=90
```

to the same line of code that sets the height, i.e., `[height=10cm,angle=90]`.

Exporting to the Windows Clipboard

You can export a figure to the Windows clipboard using one of two graphics formats: EMF color vector or BMP 8-bit color bitmap.

By default, MATLAB chooses the graphics format for you, based on the rendering method used to display the figure. For figures rendered with OpenGL or Z-buffer, MATLAB uses the BMP format. For figures rendered with Painter's, the EMF format is used. For information about how MATLAB selects a rendering method, see "The Default Renderer for MATLAB" on page 6-43.

To override the selection by MATLAB, specify the format of your choice using either the Windows **Copy Options Preferences** dialog box, or the `-d` switch in the print command.

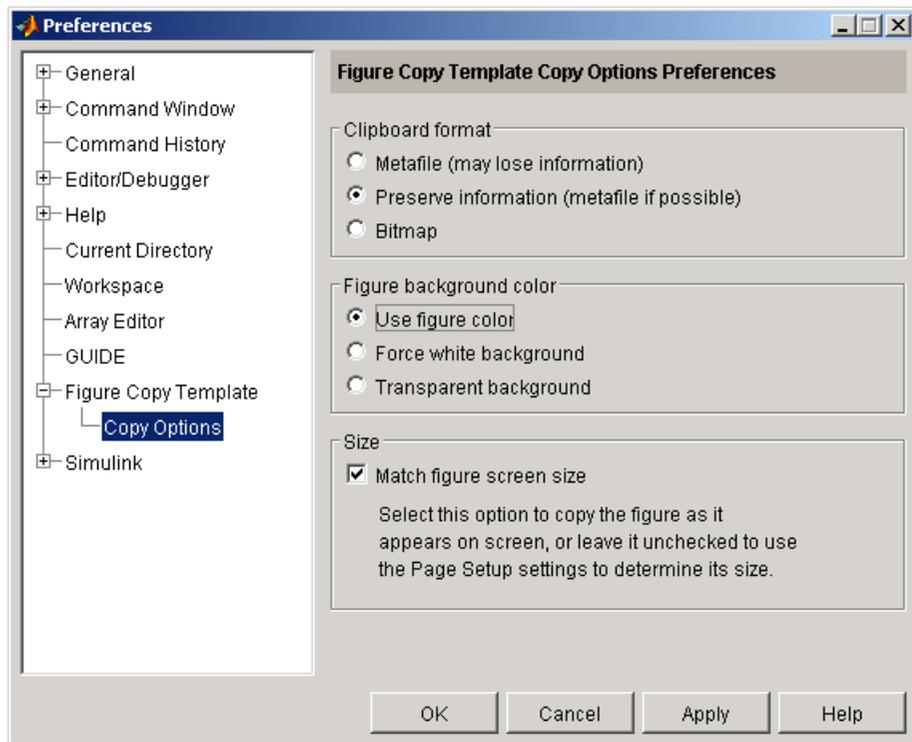
You can export to the clipboard:

- "Using the Graphical User Interface on Windows"
- "Using MATLAB Commands" on page 6-22

Using the Graphical User Interface on Windows

Before you export the figure to the clipboard, you can use the **Copy Options Preferences** dialog box to select a nondefault graphics format, or to adjust the certain figure settings. These settings become the new defaults for all figures exported to the clipboard.

To open the **Copy Options Preferences** dialog box, select **Copy Options** from the figure window's **Edit** menu. Any changes you make with this dialog box affect only the clipboard copy of the figure; they do not affect the way the figure looks on the screen.



Settings you can change in the **Copy Options Preferences** dialog box are as follows:

Clipboard format. To copy the figure in EMF color vector format, select **Metafile**. To use BMP 8-bit color bitmap format, select **Bitmap**. Or, to have MATLAB select the format for you, select **Preserve information**. MATLAB uses the metafile format, whenever possible.

Figure background color. To keep the background color the same as it appears on the screen, select **Use figure color**. To make the background white, select **Force white background**. For a background that is transparent, for example, a slide background to frame the axes part of a figure, select **Transparent background**.

Size. Select **Match figure screen size** to copy the figure as it appears on the screen, or leave it unselected to use the **Page Setup** settings to determine its size.

- 1** Open the **Copy Options Preferences** dialog box if you need to make any changes to those preferences used in copying to the clipboard.
- 2** Click **OK** to set the new preferences. These will be used for all future figures exported to the clipboard.
- 3** Select **Copy Figure** from the figure window's **Edit** menu to copy the figure to the clipboard.

Using MATLAB Commands

Export to the clipboard using the `print` function with a graphics format, but no filename. You must use one of the following clipboard formats: `-dbitmap`, or `-dmeta`. These switches create a Windows Bitmap (BMP) or an Enhanced Metafile (EMF), respectively.

For example, to export the current figure to the clipboard in Enhanced Metafile format, type

```
print -dmeta
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Examples of Basic Operations

This section provides step-by-step instructions for common printing and exporting tasks. Each printing example tells you how to perform the task from the print menus and from the command line. You can perform some tasks from the command line and others only from the menus.

The examples presented here are

- “Printing a Figure at Screen Size”
- “Printing with a Specific Paper Size” on page 6-24
- “Printing a Centered Figure” on page 6-25
- “Exporting in a Specific Graphics Format” on page 6-26
- “Exporting in EPS Format with a TIFF Preview” on page 6-27
- “Exporting a Figure to the Clipboard” on page 6-28

Printing a Figure at Screen Size

By default, MATLAB prints your figure at 8-by-6 inches. This size includes the area delimited by the background. This example shows how to print or export your figure the same size it is displayed on your screen.

Using the Graphical User Interface

- 1** Resize your figure window to the size you want it to be when printed.
- 2** Select **Page Setup** from the figure window’s **File** menu, and select the **Size and Position** tab.
- 3** In the **Mode** panel, select **Use screen size, centered on page**.
- 4** Click **OK**.
- 5** Open the **Print** dialog box and print the figure.

Using MATLAB Commands

Set the `PaperPositionMode` property to `auto` before printing the figure.

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

If later you want to print the figure at its original size, set `PaperPositionMode` back to `'manual'`.

Printing with a Specific Paper Size

By default, MATLAB uses 8.5-by-11 inch paper. This example shows how to change the paper size to 8.5-by-14 inches by selecting a paper type (`Legal`).

Using the Graphical User Interface

- 1 Select **Page Setup** from the figure window's **File** menu, and select the **Paper** tab.
- 2 Select the `Legal` paper type from the list under **Paper size**. The width and height fields update to 8.5 and 14, respectively.
- 3 Make sure that **Units** is set to inches.
- 4 Click **OK**.
- 5 Open the **Print** dialog box and print the figure.

Using MATLAB Commands

Set the `PaperUnits` property to inches, and the `PaperType` property to `Legal`.

```
set(gcf, 'PaperUnits', 'inches');  
set(gcf, 'PaperType', 'Legal');
```

Alternatively, you can set the `PaperSize` property to the size of the paper, in the specified units.

```
set(gcf, 'PaperUnits', 'inches');  
set(gcf, 'PaperSize', [8.5 14]);
```

Printing a Centered Figure

This example sets the size of a figure to 5.5-by-3 inches and centers it on the paper.

Using the Graphical User Interface

- 1 Select **Page Setup** from the figure window's **File** menu, and select the **Size and Position** tab.
- 2 Make sure **Use manual size and position** is selected.
- 3 Enter 5.5 in the **Width** field and 3 in the **Height** field.
- 4 Make sure that **Units** field is set to inches.
- 5 Click **Center**.
- 6 Click **OK**.
- 7 Open the **Print** dialog box and print the figure.

Using MATLAB Commands

- 1 Start by setting PaperUnits to inches.

```
set(gcf, 'PaperUnits', 'inches')
```
- 2 Use PaperSize to return the size of the current paper.

```
papersize = get(gcf, 'PaperSize')
```

```
papersize =  
      8.5000   11.0000
```
- 3 Initialize variables to the desired width and height of the figure.

```
width = 5.5;           % Initialize a variable for width.  
height = 3;           % Initialize a variable for height.
```

- 4 Calculate a left margin that centers the figure horizontally on the paper. Use the first element of `papersize` (width of paper) for the calculation.

```
left = (papersize(1) - width)/2
```

```
left =  
    1.5000
```

- 5 Calculate a bottom margin that centers the figure vertically on the paper. Use the second element of `papersize` (height of paper) for the calculation.

```
bottom = (papersize(2) - height)/2
```

```
bottom =  
    4
```

- 6 Set the figure size and print.

```
myfiguresize = [left, bottom, width, height];  
set(gcf, 'PaperPosition', myfiguresize);  
print
```

Exporting in a Specific Graphics Format

Export a figure to a graphics-format file when you want to import it at a later time into another application such as a word processor.

Using the Graphical User Interface

- 1 Select **Export** from the figure window's **File** menu.
- 2 Use the **Save in** field to navigate to the directory in which you want to save your file.
- 3 Select a graphics format from the **Save as type** list.
- 4 Enter a filename in the **File name** field. An appropriate file extension, based on the format you chose, is displayed.
- 5 Click **Save** to export the figure.

Using MATLAB Commands

From the command line, you must specify the graphics format as an option. See the print reference page for a complete list of graphics formats and their corresponding option strings.

This example exports a figure to an EPS color file, `myfigure.eps`, in your current directory.

```
print -depsc myfigure
```

This example exports Figure No. 2 at a resolution of 300dpi to a 24-bit JPEG file, `myfigure.jpg`.

```
print -djpeg -f2 -r300 myfigure
```

This example exports a figure at screen size to a 24-bit TIFF file, `myfigure.tif`.

```
set(gcf, 'PaperPositionMode', 'auto') % Use screen size
print -dtiff myfigure
```

Exporting in EPS Format with a TIFF Preview

Use the `print` function to export a figure in EPS format with a TIFF preview. When you import the figure, the application can display the TIFF preview in the source document. The preview is color if the exported figure is color, and black and white if the exported figure is black and white.

This example exports a figure to an EPS color format file, `myfigure.eps`, and includes a color TIFF preview.

```
print -depsc -tiff myfigure
```

This example exports a figure to an EPS black-and-white format file, `myfigure.eps`, and includes a black-and-white TIFF preview.

```
print -deps -tiff myfigure
```

Exporting a Figure to the Clipboard

Export a figure to the clipboard in graphics-format when you want to paste it into another Windows application such as a word processor.

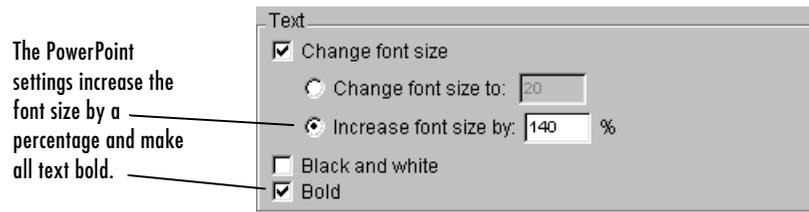
Using the Graphical User Interface on Windows

This example exports a figure to the clipboard in Enhanced Metafile (EMF) format. Figure settings are chosen that would make the exported figure suitable for use in a PowerPoint slide. Note that changing the settings modifies the figure displayed on the screen.

- 1 Create a figure containing text. You can use the following code.

```
x = -pi:0.01:pi;
h = plot(x, sin(x));
title('Sine Plot');
```

- 2 Select **Preferences** from the figure window's **File** menu. Then select **Figure Copy Template** from the **Preferences** dialog box.
- 3 In the **Figure Copy Template Preferences** panel, click the **PowerPoint** button. The MATLAB suggested settings for PowerPoint are added to the template.

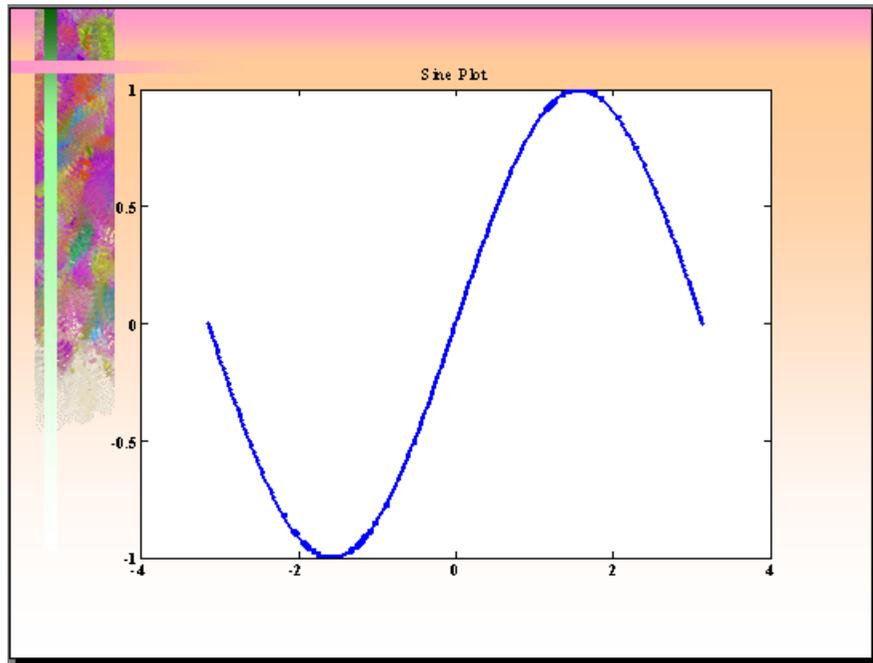


The **Text** panel after the **PowerPoint** button is clicked

- 4 In the **Lines** panel, change the **Custom width** to 4 points.
- 5 In the **Uicontrols and axes** panel, select **Keep axes limits and tick spacing** to prevent MATLAB from possibly rescaling ticks and limits when you export.

- 6** Click **Apply to Figure**. The changes appear in the figure window.

If you don't like the way your figure looks with the new settings, you can restore it to its original settings by clicking the **Restore Figure** button.
- 7** In the left pane of the **Preferences** dialog box, expand the **Figure Copy Template** topic. Select **Copy Options**.
- 8** In the **Copy Options** panel, select **Metafile** to tell MATLAB to export the figure in EMF format.
- 9** Check that **Transparent background** is selected. This choice makes the figure background transparent and allows the slide background to frame the axes part of the figure.
- 10** Clear **Match figure screen size** check box so that you can use your own figure size settings.
- 11** Click **OK**.
- 12** Select **Page Setup** from the figure window's **File** menu.
- 13** In the **Size and Position** tab, set **Width** to 10 and **Height** to 7.5. Make sure that **Units** are set to inches.
- 14** Click **OK**.
- 15** Select **Copy Figure** from the **Edit** menu. Your figure is now exported to the clipboard and can be pasted into another Windows application such as a PowerPoint slide.



Using MATLAB Commands

Use the `print` function and one of two clipboard formats (`-dmeta`, `-dbitmap`) to export a figure to the clipboard. Do *not* specify a filename.

This example exports a figure to the clipboard in Enhanced Metafile (EMF) format.

```
print -dmeta
```

This example exports a figure to the clipboard in Bitmap (BMP) 8-bit color format.

```
print -dbitmap
```

Changing a Figure's Settings

The table below shows parameters that you can set before submitting your figure to the printer.

Column 1 of the table lists all parameters that you can change.

Column 2 shows the default setting that MATLAB uses.

Column 3 shows which dialog box to use to set that parameter. If you can make this setting on only one platform, this is noted in parentheses: (W) for Windows, and (U) for UNIX.

Some dialog boxes have tabs at the top to enable you to select a certain category. These categories are denoted in the table below using the format `<dialogbox>/<tabname>`. For example, “**Page Setup/Size ...**” in this column means to use the **Page Setup** dialog box, selecting the **Size and Position** tab.

Column 4 shows how to set the parameter using the MATLAB print or set function. When using print, the table shows the appropriate command option (for example, `print -loose`). When using set, it shows the property name to set along with the type of object (for example, (Line) for line objects).

Parameter	Default	Dialog Box	PRINT Command or SET Property
Select figure	last active window	none	<code>print -fhandle</code>
Select printer	system default	Print	<code>print -pprinter</code>
Figure size	8-by-6 inches	Page Setup/Size ...	PaperSize (Figure) PaperUnits (Figure)
Position on page	0.25 in. from left, 2.5 in. from bottom	Page Setup/Size ...	PaperPosition (Figure) PaperUnits (Figure)
Position mode	manual	Page Setup/Size ...	PaperPositionMode (Figure)
Paper type	letter	Page Setup/Paper	PaperType (Figure)

Parameter	Default	Dialog Box	PRINT Command or SET Property
Paper orientation	portrait	Page Setup/Paper	PaperOrientation (Figure)
Renderer	selected by MATLAB	Page Setup/Axes ...	print -zbuffer -painters -opengl
Renderer mode	Auto	Page Setup/Axes ...	RendererMode (Figure)
Resolution	depends on driver or graphics format	Print Properties (W) Printing Options (U)	print -resolution
Axes tick marks	recompute	Page Setup/Axes ...	XTickMode, etc. (Axes)
Background color	force to white	Page Setup/Axes ...	Color (Figure) InvertHardCopy (Figure)
Font size	as in the figure	Fig. Copy Template	FontSize (Text)
Bold font	regular font	Fig. Copy Template	FontWeight (Text)
Line width	as in the figure	Fig. Copy Template	LineWidth (Line)
Line style	black or white	Fig. Copy Template	LineStyle (Line)
Line and text color	black and white	Page Setup/Lines ...	Color (Line, Text)
CMYK color	RGB color	Printing Options (U)	print -cmyk
UI controls	printed	Page Setup/Axes ...	print -noui
Bounding box	tight	Printing Options (U)	print -loose
Copy background	transparent	Copy Options (W)	see “Background color”
Copy size	same as screen size	Copy Options (W)	see “Figure Size”

Selecting the Figure

By default, MATLAB prints the current figure. If you have more than one figure open, the current figure is the last one that was active. To make a different figure active, click on it to bring it to the foreground.

Using MATLAB Commands

Specify a figure handle using the command

```
print -handle
```

This example sends Figure No. 2 to the printer. A figure's number is usually its handle.

```
print -f2
```

Selecting the Printer

You can select the printer you want to use with the **Print** dialog box or with the `print` function.

Using the Graphical User Interface

- 1 Select **Print** from the figure window's **File** menu.
- 2 Select the printer from the list box near the top of the **Print** dialog box.
- 3 Click **OK**.

Using MATLAB Commands

You can select the printer using the `-P` switch of the `print` function.

This example prints Figure No. 3 to a printer called Calliope.

```
print -f3 -PCalliope
```

If the printer name has spaces in it, put quotes around the `-P` option, as shown here.

```
print "-Pmy local printer"
```

Using a Network Print Server. On Windows NT, Windows 2000, and Windows XP systems, you can print to a network print server using the form shown here for a printer named trinity.

```
print -P\\PRINTERS\trinity
```

This form is not supported on Windows 98 or Windows ME. On these platforms, you can print to a network printer only if you install a network printer using the **Add Printer** dialogs. When installed in this manner, these network printers work without the use of the \\server\printer notation, as they look the same as local printers.

Setting the Figure Size and Position

The default output figure size is 8 inches wide by 6 inches high, which maintains the aspect ratio (width to height) of the MATLAB figure window. The figure's default position is centered both horizontally and vertically when printed to a paper size of 8.5-by-11 inches.

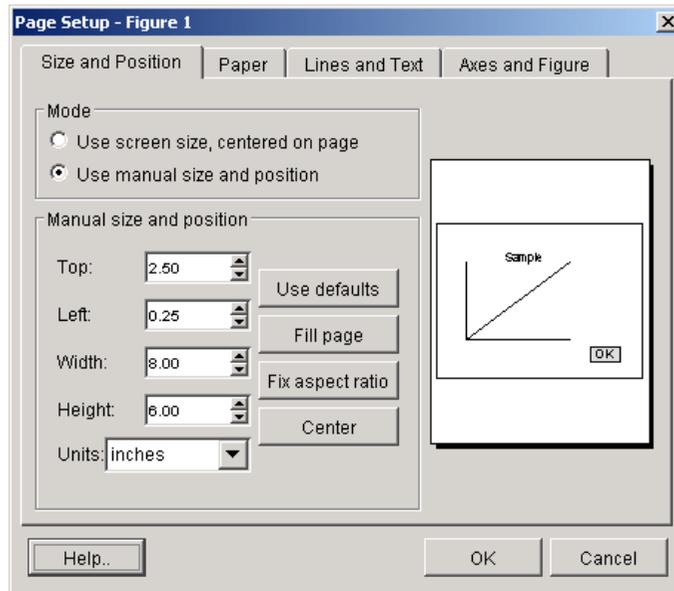
You can change the size and position of the figure:

- “Using the Graphical User Interface”
- “Using MATLAB Commands” on page 6-37

Using the Graphical User Interface

Select **Page Setup** from the figure window's **File** menu to open the **Page Setup** dialog box. Click the **Size and Position** tab to make changes to the size and position of your figure on the printed page.

Use the text edit boxes on the left to enter new dimensions for your figure. Or use the graphical user interface at the right to drag the borders and location of the “sample” figure with your mouse.



Settings you can change in the **Size and Position** window are as follows:

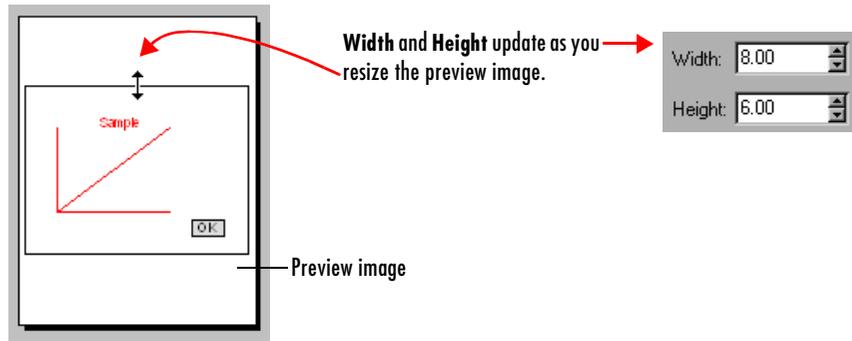
Mode. Choose whether you want the figure to be the same size as it is displayed on your screen, or you want to manually change its size using the options in the **Size and Position** window.

The next two panels are enabled only when you select the **Use manual size and position** mode.

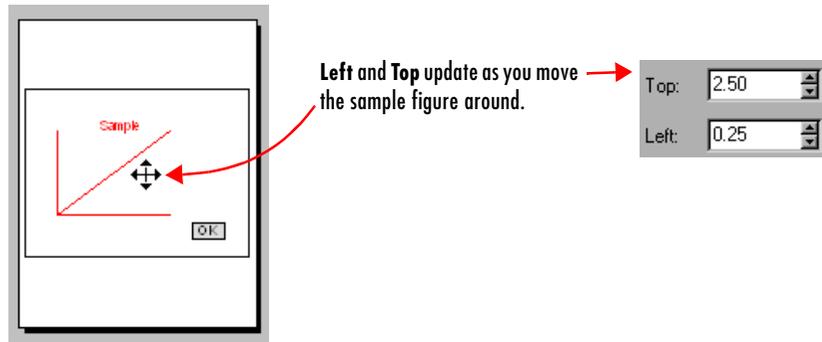
Manual size and position. Enter the measurements and units for the size and position of the figure.

Graphical User Interface. Use the “Sample” figure at the right of the dialog box to move and resize your MATLAB figure interactively.

To set the width and height interactively, use the mouse to drag the edges of the “Sample” figure to the desired size.



To set the margins of the figure (offsets from the left and top edges of the paper), drag the entire “Sample” figure to a new position with the mouse.



If you want the figure resized to fill the paper, click **Fill page**. Note that **Fill page** may alter the aspect ratio of your image. To get the maximum figure size without altering the aspect ratio, select **Fix aspect ratio**.

Note Changes you make using **Page Setup** affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

To print your figure with a specific size or position, make sure that the `PaperPositionMode` property is set to `manual` (the default). Then set the `PaperPosition` property to the desired size and position.

The `PaperPosition` property references a four-element row vector that specifies the position and dimensions of the printed output. The form of the vector is

```
[left bottom width height]
```

where

- `left` specifies the distance from the left edge of the paper to the left edge of the figure.
- `bottom` specifies the distance from the bottom of the paper to the bottom of the figure.
- `width` and `height` specify the figure's width and height.

The MATLAB default values for `PaperPosition` are

```
[0.25 2.5 8.0 6.0]
```

This example sets the figure size to a width of 4 inches and height of 2 inches, with the origin of the figure positioned 2 inches from the left edge of the paper and 1 inch from the bottom edge.

```
set(gcf, 'PaperPositionMode', 'manual');  
set(gcf, 'PaperUnits', 'inches');  
set(gcf, 'PaperPosition', [2 1 4 2]);
```

Note `PaperPosition` specifies a bottom margin, rather than a top margin as **Page Setup** does. When you set the top margin using **Page Setup**, MATLAB uses this setting to calculate the bottom margin, and updates the `PaperPosition` property appropriately.

Setting the Paper Size or Type

Set the paper size by specifying the dimensions or by choosing from a list of predefined paper types. If you do not set a paper size or type, MATLAB uses the default paper size of 8.5-by-11 inches.

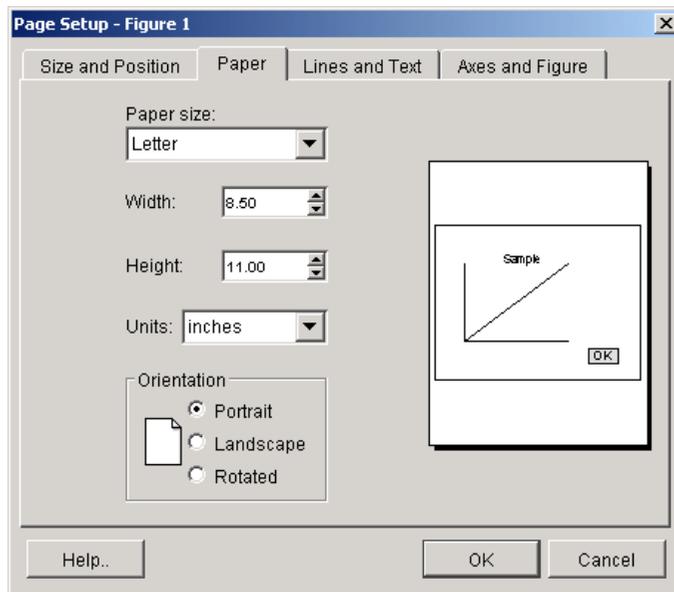
Paper-size and paper-type settings are interrelated — if you set a paper type, MATLAB updates the paper size. For example, if you set the paper type to US Legal, MATLAB updates the width of the paper to 8.5 inches and the height to 14 inches.

You can change the paper size and orientation:

- “Using the Graphical User Interface”
- “Using MATLAB Commands” on page 6-39

Using the Graphical User Interface

Select **Page Setup** from the figure window’s **File** menu to open the **Page Setup** dialog box. Click the **Paper** tab to make changes to the paper type and orientation of the figure on the printed page.



Settings you can change in the **Paper** window are as follows:

Paper size. Select a paper type from the list under **Paper size**. If there is no paper type with suitable dimensions, enter your own dimensions in the **Width** and **Height** fields. Make sure **Units** is set appropriately to inches, centimeters, points, or normalized.

Orientation. Select how you want the figure to be oriented on the printed page. The illustration under “Setting the Paper Orientation” on page 6-39 shows the three types of orientation you can choose from.

Note Changes you make using **Page Setup** affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

Set the `PaperType` property to one of the built-in MATLAB paper types, or set the `PaperSize` property to the dimensions of the paper.

When you select a paper type, the unit of measure is not automatically updated. We recommend that you set the `PaperUnits` property first.

For example, these commands set the units to centimeters and the paper type to A4.

```
set(gcf, 'PaperUnits', 'centimeters');  
set(gcf, 'PaperType', 'A4');
```

This example sets the units to inches and sets the paper size of 5-by-7 inches.

```
set(gcf, 'PaperUnits', 'inches');  
set(gcf, 'PaperSize', [5 7]);
```

If you set a paper size for which there is no matching paper type, the `PaperType` property is automatically set to '`<custom>`'.

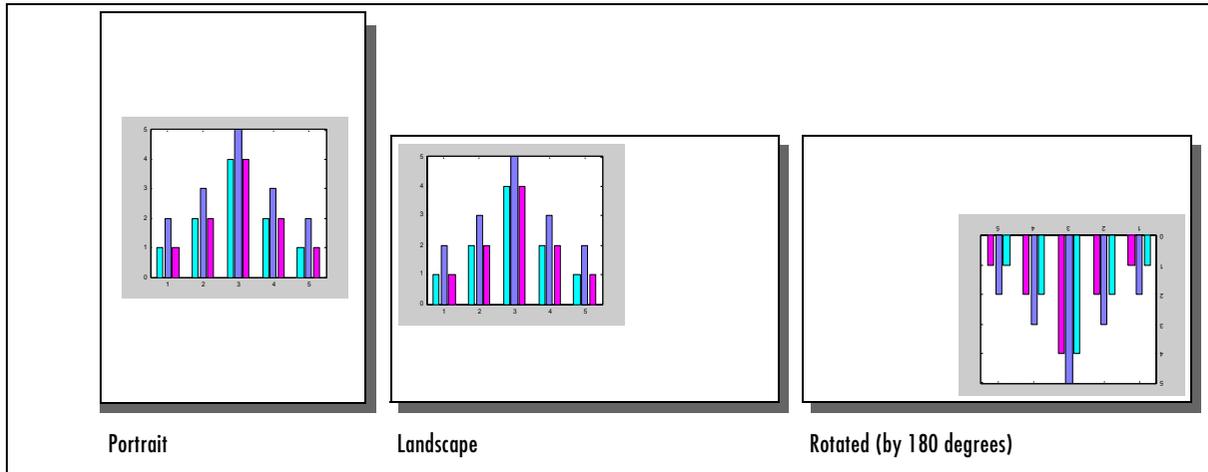
Setting the Paper Orientation

Paper orientation refers to how the paper is oriented with respect to the figure. The choices are **Portrait** (the default), **Landscape**, and **Rotated**.

You can change the orientation of the figure:

- “Using the Graphical User Interface”
- “Using MATLAB Commands” on page 6-41

The figure below shows the same figure printed using the three different orientations.



Note The rotated orientation is not supported by all printers. When the printer does not support it, landscape is used.

Using the Graphical User Interface

- 1 Select **Page Setup** from the figure window’s **File** menu and select the **Paper** tab. (See “Using the Graphical User Interface” on page 6-38).
- 2 Select the appropriate option button under **Orientation**.
- 3 Click **OK**.

Using MATLAB Commands

Use the `PaperOrientation` figure property or the `orient` function. Use the `orient` function if you always want your figure centered on the paper.

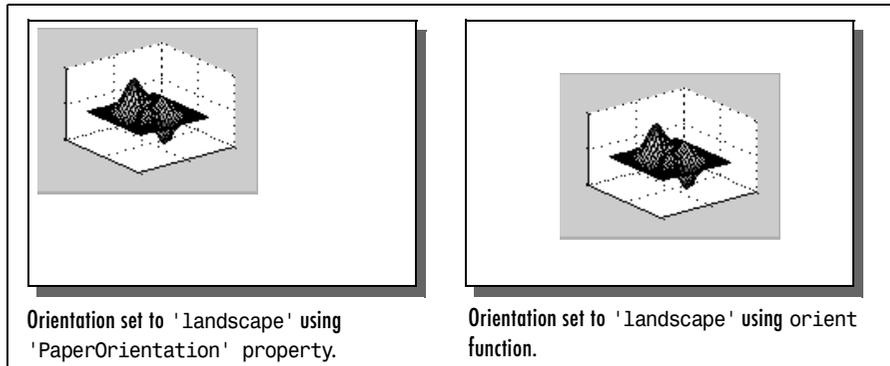
The following example sets the orientation to landscape:

```
set(gcf, 'PaperOrientation', 'landscape');
```

Centering the Figure. If you set the `PaperOrientation` property from portrait to either of the other two orientation schemes, you may find that what was previously a centered image is now positioned near the paper's edge. You can either adjust the position (use the `PaperPosition` property), or you can use the `orient` function, which always centers the figure on the paper.

The `orient` function takes the same argument names as `PaperOrientation`. For example,

```
orient rotated;
```



Selecting a Renderer

A renderer is software and/or hardware that processes graphics data (such as vertex coordinates) to display, print, or export a figure. You can change the renderer that MATLAB uses when printing a figure:

- “Using the Graphical User Interface” on page 6-44
- “Using MATLAB Commands” on page 6-44

Renderers Supported by MATLAB

MATLAB supports three rendering methods with the following characteristics:

Painter's

- Draws figures using vector graphics
- Generally produces higher resolution results
- The fastest renderer when the figure contains only simple or small graphics objects
- The only renderer possible when printing with the HPGL print driver or exporting to an Adobe Illustrator file
- The best renderer for creating PostScript or EPS files
- Cannot render figures that use RGB color for patch or surface objects
- Does not show lighting or transparency

Z-buffer

- Draws figures using bitmap (raster) graphics
- Faster and more accurate than Painter's
- Can consume a lot of system memory if MATLAB is displaying a complex scene
- Shows lighting, but not transparency

OpenGL

- Draws figures using bitmap (raster) graphics
- Generally faster than Painter's or Z-buffer
- In some cases, enables MATLAB to access graphics hardware that is available on some systems
- Shows both lighting and transparency

For more detailed information about the rendering methods, see **Renderer** on the “Figure Properties” reference pages.

The Default Renderer for MATLAB

By default, MATLAB automatically selects the best rendering method, based on the attributes of the figure (its complexity and the settings of various Handle Graphics properties) and in some cases, the printer driver or file format used.

In general, MATLAB uses

- Painter's for line plots, area plots (bar graphs, histograms, etc.), and simple surface plots
- Z-buffer when the computer screen is not truecolor or when the `opengl` function was called with `selection_mode` set to `neverselect`
- OpenGL for complex surface plots using interpolated shading and any figure using lighting

The `RendererMode` property tells MATLAB whether to automatically select the renderer based on the contents of the figure (when set to `auto`), or to use the `Renderer` property that you have indicated (when set to `manual`).

Reasons for Manually Setting the Renderer

Two reasons to set the renderer yourself are

- To make your printed or exported figure look the same as it did on the screen. The rendering method used for printing and exporting the figure is not always the same method used to display the figure.
- To avoid unintentionally exporting your figure as a bitmap within a vector format. For example, MATLAB typically renders high-complexity plots using OpenGL or Z-buffer. If you export a high-complexity figure to the EPS or EMF vector formats without specifying a rendering method, MATLAB might use OpenGL or Z-buffer, each of which creates bitmap graphics.

Storing a bitmap in a vector file can generate a very large file that takes a long time to print. If you use one of these formats and want to make sure that your figure is saved as a vector file, be sure to set the rendering method to Painter's.

Using the Graphical User Interface

- 1** Open the **Page Setup** dialog box by selecting **Page Setup** from the figure window's **File** menu. Select the **Axes and Figure** tab.
- 2** Under **Figure renderer**, select the desired rendering method from the list box.
- 3** Click **OK**.

Using MATLAB Commands

You can use the `Renderer` property or a switch with the `print` function to set the renderer for printing or exporting. These two lines each set the renderer for the current figure to Z-buffer.

```
set(gcf, 'Renderer', 'zbuffer');
```

or

```
print -zbuffer
```

The first example saves the new value of `Renderer` with the figure; the second example only affects the current print or export operation.

Note that when you set the `Renderer` property, the `RenderMode` property is automatically reset from `auto` (the factory default) to `manual`.

Setting the Resolution

Resolution refers to how accurately your figure is rendered when printed or exported. Higher resolutions produce higher quality output. The specific definition of resolution depends on whether your figure is output as a bitmap or as a vector graphic.

You can change the resolution that MATLAB uses to print a figure

- “Using the Graphical User Interface on Windows” on page 6-46
- “Using the Graphical User Interface on UNIX” on page 6-47
- “Using MATLAB Commands” on page 6-47

Default Resolution and When You Can Change It

The default resolution depends on the renderer used and the graphics format or printer driver specified. The following two tables summarize the default resolutions and whether you can change them.

Resolutions Used with Graphics Formats

Graphics Format	Default Resolution	Can Be Changed?
Built-in MATLAB export formats, (except for EMF, EPS, and ILL)	150 dpi (always use OpenGL or Z-buffer)	Yes
EMF export format (Enhanced Metafile)	150 dpi	Yes
EPS (Encapsulated PostScript)	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
ILL export format (Adobe Illustrator)	72 dpi (always uses Painter's)	No
Ghostscript export formats	72 dpi (always uses OpenGL or Z-buffer)	No

Resolutions Used with Printer Drivers

Printer Driver	Default Resolution	Can Be Changed?
Windows and PostScript drivers	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
Ghostscript driver	150 dpi, if OpenGL or Z-buffer; 864 dpi if Painter's	Yes
HPGL driver	1116 dpi (always uses Painter's)	Yes

Choosing a Setting

You may need to determine your resolution requirements through experimentation, but you can also use the following guidelines.

For Printing. The default resolution of 150 dpi is normally adequate for typical laser-printer output. However, if you are preparing figures for high-quality printing, such as a textbook or color brochures, you may want to use 200 or 300 dpi. The resolution you can use may be limited by the printer's capabilities.

For Exporting. If you are exporting your figure, base your decision on the resolution supported by the final output device. For example, if you will import your figure into a word processing document and print it on a printer that supports a maximum resolution setting of 300 dpi, you may want to export your figure using 300 dpi to get a precise one-to-one correspondence between pixels in the file and dots on the paper.

Note The only way to set resolution when exporting is with the print function.

Impact of Resolution on Size and Memory Needed

Resolution affects file size and memory requirements. For both printing and exporting, the higher the resolution setting, the longer it takes for MATLAB or your printer to render your figure.

Using the Graphical User Interface on Windows

To set the resolution for built-in MATLAB printer drivers on Windows systems,

- 1 From the **Print** dialog box, click **Properties**. This opens a new dialog box that may differ from one printer to another.
- 2 You may be able to set the resolution from this dialog, or you may have to click on **Advanced** to get to a dialog box that enables you to do this.
- 3 Set the resolution, and then click **OK**. (The resolution setting may be labeled by another name, such as "Print Quality.")

Using the Graphical User Interface on UNIX

To set the resolution for built-in MATLAB printer drivers on UNIX systems,

- 1** From the UNIX **Print** dialog box, click **Options**. This opens the **Printing Options** dialog box.
- 2** Under the **Printing Resolution** panel, select either **Use same resolution as the screen** or **Specify resolution in dots per inch**.
- 3** If you select **Specify resolution in dots per inch**, enter a value in the **Specify resolution in dots per inch** text box.
- 4** Click **OK**.

Using MATLAB Commands

If you use a Windows printer driver, you can only set the resolution using the Windows **Document Properties** dialog box.

Otherwise, to set the resolution for printing or exporting, the syntax is

```
print -rnumber
```

where number is the number of dots per inch. To print or export a figure using screen resolution, set number to 0 (zero).

This example prints the current figure with a resolution of 100 dpi,

```
print -r100
```

This example exports the current figure to a TIFF file using screen resolution,

```
print -r0 -dtiff myfile.tif
```

Setting the Axes Ticks and Limits

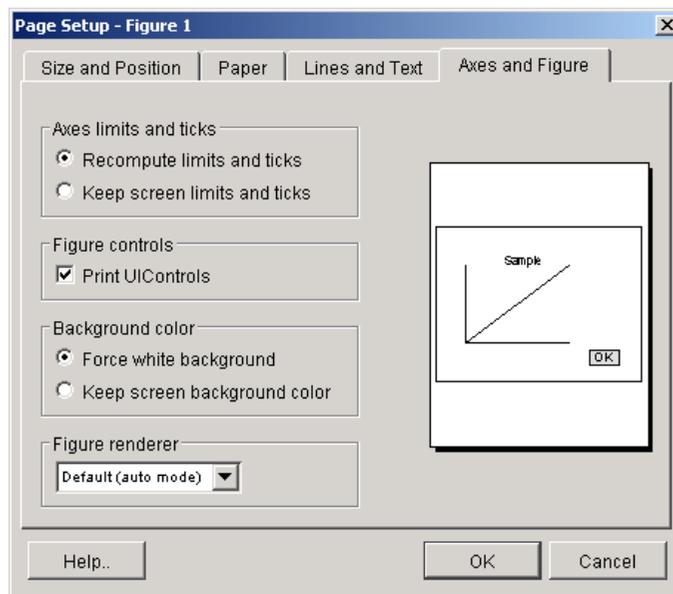
The MATLAB default output size, 8-by-6 inches, is normally larger than the screen size. If the size of your printed or exported figure is different from its size on the screen, MATLAB scales the number and placement of axes tick marks to suit the output size. This section shows you how to lock them so that they are the same as they were when displayed.

You can change the resolution that MATLAB uses to print a figure

- “Using the Graphical User Interface”
- “Using MATLAB Commands” on page 6-49

Using the Graphical User Interface

Select **Page Setup** from the figure window’s **File** menu to open the **Page Setup** dialog box. Select the **Axes and Figure** tab to make changes to the axes, UI controls, background color, or renderer selection.



Settings you can change in the **Axes and Figure** window are as follows:

Axes limits and ticks. If the size of your printed or exported figure is different from its size on the screen, MATLAB scales the number and placement of axes tick marks to suit the output size. Select **Keep screen limits and ticks** to lock them so that they are the same as they were when displayed.

Figure controls. By default, user interface controls are included in your printed or exported figure. Clear the **Print UIControls** check box to exclude them. (See “Excluding User Interface Controls” on page 6-55).

Background color. You can keep the background the same as is shown on the screen when printed, or change the background to be white. (See “Setting the Background Color” on page 6-49).

Figure renderer. Set the renderer to Painter's, Z-buffer, or OpenGL, or let MATLAB decide which one to use, depending on the characteristics of the figure. (See “Selecting a Renderer” on page 6-41).

Note Changes you make using **Page Setup** affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

To set the `XTickMode`, `YTickMode`, and `ZTickMode` properties to manual, type

```
set(gca, 'XTickMode', 'manual');  
set(gca, 'YTickMode', 'manual');  
set(gca, 'ZTickMode', 'manual');
```

Setting the Background Color

There are two types of background color settings in a figure: the axes background and the figure background. The default displayed color of both backgrounds is gray, but you can set them to any of several colors.

Regardless of the background colors in your displayed figure, by default, MATLAB always changes them to white when you print or export. This section shows you how to retain the displayed background colors in your output.

Using the Graphical User Interface

To retain the background color on a per figure basis,

- 1 Open the **Page Setup** dialog box by selecting **Page Setup** from the figure window's **File** menu. Select the **Axes and Figure** tab.
- 2 Select **Keep screen background color**.
- 3 Click **OK**.

If you are exporting your figure using the clipboard, use the **Copy Options** panel of the **Preferences** dialog box.

Using MATLAB Commands

To retain your background colors, use

```
set(gcf, 'InvertHardCopy', 'off');
```

The following example sets the figure background color to blue, the axes background color to yellow, and then sets `InvertHardCopy` to off so that these colors will appear in your printed or exported figure.

```
set(gcf, 'color', 'blue');  
set(gca, 'color', 'yellow');  
set(gcf, 'InvertHardCopy', 'off');
```

Setting Line and Text Characteristics

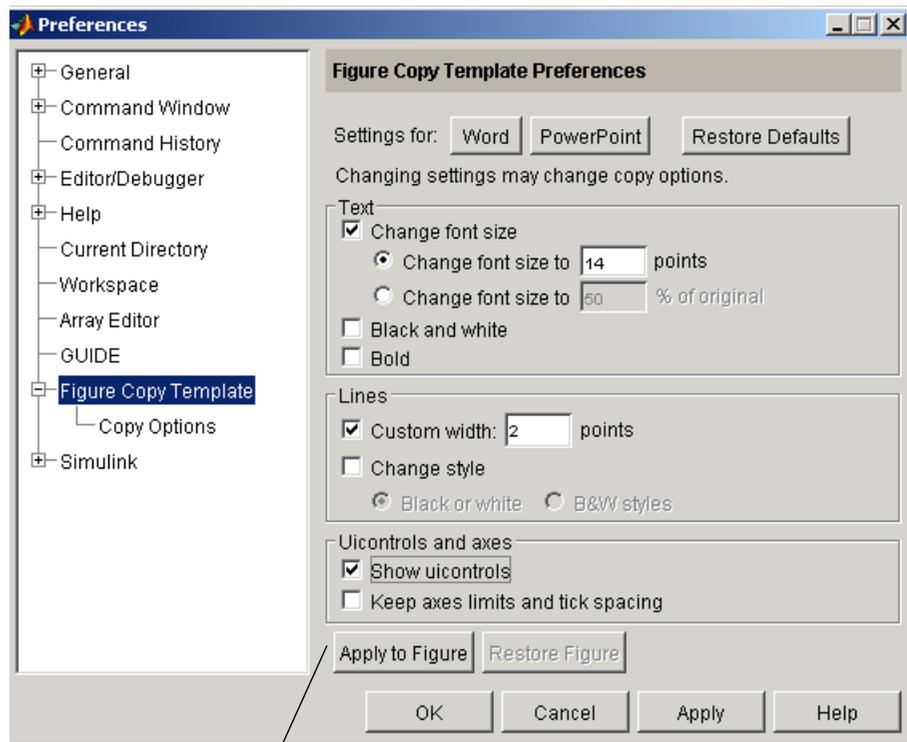
If you transfer your figures to Word or PowerPoint applications, you can set line and text characteristics to values recommended for those applications. The **Figure Copy Template Preferences** dialog box provides **Word** and **PowerPoint** options to make these settings, or you can set certain line and text characteristics individually.

You can change line and text characteristics:

- “Using the Graphical User Interface”
- “Using MATLAB Commands” on page 6-52

Using the Graphical User Interface

To open **Figure Copy Template Preferences**, select **Preferences** from the figure window’s **File** menu, and then click on **Figure Copy Template** in the left pane.



Note the difference between **Apply to Figure** and **Apply**. Use **Apply to Figure** to modify the figure in the figure window. Use **Apply** or **OK** to save your preferences.

Settings you can change in the **Figure Copy Template Preferences** dialog box are as follows:

Word or PowerPoint. Click **Word** or **PowerPoint** to apply settings recommended for MATLAB.

Text. Use options in the **Text** panel to modify the appearance of all text in the figure. You can change the font size, change color text to black and white, and change the font style to bold.

Lines. Use the **Lines** panel to modify the appearance of all lines in the figure. Options include

- **Custom width** — Change the line width.
- **Change style (Black or white)** — Change colored lines to black or white.
- **Change style (B&W styles)** — Change solid lines to different line styles (e.g., solid, dashed, etc.), and black or white color.

UIControls and axes. If your figure includes user interface controls, you can choose to show or hide them by clicking on **Show uicontrols**. Also, to keep axes limits and ticks as they appear on the screen, click **Keep axes limits and tick spacing**. To allow MATLAB to scale axes limits and ticks based on the size of the printed figure, clear this box.

Note Changes you make using **Page Setup** affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

You can use the `set` function on selected graphics objects in your figure to change individual line and text characteristics.

For example, to change line width to 1.8 and line style to a dashed line, use

```
lineobj = findobj('type', 'line');  
set(lineobj, 'linewidth', 1.8);  
set(lineobj, 'linestyle', '--');
```

To change the font size to 15 points and font weight to bold, use

```
textobj = findobj('type', 'text');  
set(textobj, 'fontunits', 'points');  
set(textobj, 'fontsize', 15);  
set(textobj, 'fontweight', 'bold');
```

Setting the Line and Text Color

When colored lines and text are dithered to gray by a black-and-white printer, it does not produce good results for thin lines and the thin lines that make up text characters. You can, however, force all line and text objects in the figure to

print in black and white, thus improving their appearance in the printed copy. When you select this setting, the lines and text are printed all black or all white, depending on the background color.

The default is to leave lines and text in the color that appears on the screen.

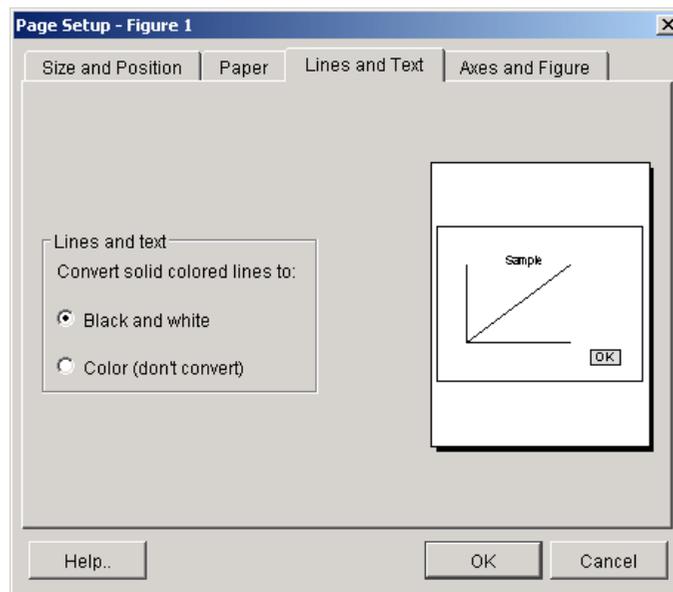
Note Your background color may not be the same as what you see on the screen. See the **Axes and Figure** tab for an option that preserves the background color when printing.

You can change the resolution that MATLAB uses to print a figure:

- “Using the Graphical User Interface”
- “Using MATLAB Commands” on page 6-54

Using the Graphical User Interface

Select **Page Setup** from the figure window's **File** menu to open the **Page Setup** dialog box. Select the **Lines and Text** tab to make changes to the color of all lines and text on the printed page.



Settings you can change in the **Lines and Text** window are as follows:

Lines and text. To have colored lines and text printed as black and white, select **Black and white**. To print them in color, select **Color (don't convert)**.

Note Changes you make using **Page Setup** affect the printed output only. They do not alter the figure displayed on your screen.

Using MATLAB Commands

There is no equivalent MATLAB command that will set line and text color depending on background color. Set the color of lines and text using the `set` function on either Line or Text objects in your figure.

This example sets all lines and text to black:

```
set(findobj('type', 'line'), 'color', 'black');  
set(findobj('type', 'text'), 'color', 'black');
```

Setting CMYK Color

By default, MATLAB produces color output in the RGB color space (red, green, blue). If you plan to publish and print MATLAB figures using printing industry standard four-color separation, you may want to use the CMYK color space (cyan, magenta, yellow, black).

Using the Graphical User Interface on UNIX

- 1 Select **Print** from the figure window's **File** menu.
- 2 Click **Options**. This opens the **Printing Options** dialog box.
- 3 Select **Use CMYK colors in PS and Ghostscript drivers instead of RGB**.
- 4 Click **OK**.

Using MATLAB Commands

Use the `-cmyk` option with the `print` function. This example prints the current figure in CMYK using a PostScript Level II color printer driver.

```
print -dpsc2 -cmyk
```

Excluding User Interface Controls

User interface controls are objects that you create and add to a figure. For example, you can add a button to a figure that, when clicked, conveniently runs another M-file. By default, user interface controls are included in your printed or exported figure. This section shows how to exclude them.

Using the Graphical User Interface

- 1 Open the **Page Setup** dialog box by selecting **Page Setup** from the figure window's **File** menu. Select the **Axes and Figure** tab.
- 2 Under **Figure controls**, clear the **Print UIControls** check box.
- 3 Click **OK**.

Using MATLAB Commands

Use the `-noui` switch. This example specifies a color PostScript driver and excludes UI controls.

```
print -dpsc -noui
```

This example exports the current figure to a color EPS file and excludes UI controls.

```
print -depesc -noui myfile.eps
```

Producing Uncropped Figures

In most cases, MATLAB crops the background tightly around the objects in the figure. Depending on the printer driver or file format you use, you may be able to produce uncropped output. An uncropped figure has increased background area and is often desirable for figures that contain UI controls.

The setting you make in MATLAB changes the PostScript `BoundingBox` property saved with the figure.

Using the Graphical User Interface on UNIX

You can only make the uncropped setting on a per figure basis:

- 1 Select **Print** from the figure window's File menu.
- 2 From the UNIX **Print** dialog box, click **Options**. This opens the **Printing Options** dialog box.
- 3 Select **Use loose bounding box for PS and Ghostscript drivers**.
- 4 Click **OK**.

Using MATLAB Commands

Use the `-loose` option with the `print` function. For Windows, the uncropped option is only available if you print to a file.

This example exports the current figure, uncropped, to an EPS file.

```
print -deps -loose myfile.eps
```

Choosing a Graphics Format

A graphics file format is a specification for storing and organizing data in a file. MATLAB supports many different graphics file formats. Some are built into MATLAB and others are Ghostscript formats. File formats also differ in color support, graphics style (bitmap or vector), and bit depth.

This section provides information to help you decide which graphics format to use when exporting your figure to a file or to the Windows clipboard. It covers

- “Frequently Used Graphics Formats” on page 6-58
- “Factors to Consider in Choosing a Format” on page 6-59
- “Properties Affected by Choice of Format” on page 6-61
- “Impact of Rendering Method on the Output” on page 6-63
- “Description of Selected Graphics Formats” on page 6-63
- “How to Specify a Format for Exporting” on page 6-66

Before deciding on a graphics format, check what formats are supported by your target application and platform. See the print reference page for a complete list of graphics formats supported in MATLAB. Once you decide on which format to use in exporting your figure, follow the instructions in “Exporting to a File” on page 6-15 or “Exporting to the Windows Clipboard” on page 6-20.

Frequently Used Graphics Formats

Here are some of the more frequently used graphics formats. For a complete list, see the Graphics Format table on the print reference page. For a more complete description of these formats, see “Description of Selected Graphics Formats” on page 6-63.

Format	Description	Command Line -device Parameter
EPS color, and black and white	Export line plots or simple graphs to a file. Note. An EPS file does not display within some applications unless you add a TIFF preview image to it. See the example “Exporting in EPS Format with a TIFF Preview” on page 6-27.	-deps (black and white) -depsec (color) -depsec -tiff (TIFF preview)
JPEG 24-bit	Export plots with surface lighting or transparency to a file. This format can be displayed by most Web browsers.	-djpeg -djpegnumber, where <i>number</i> is the compression.
TIFF 24-bit bitmap color	Export plots with surface lighting or transparency to a file. Widely available. A good format to choose if you are not sure what formats your application supports.	-dtiff
BMP 8-bit color bitmap	Export a figure to the clipboard (Windows only).	-dbitmap
EMF color vector format	Export a figure to the clipboard (Windows only).	-dmeta

Factors to Consider in Choosing a Format

There are at least five main factors to consider when choosing a graphics format to use in exporting a figure:

- Implementation — Is it a built-in MATLAB or Ghostscript format?
- Graphics Format — Is it bitmap or vector graphics format?
- Bit Depth — What bit depth does the format offer?
- Color Support — What color support does it have?
- Model/Publication — Is it a Simulink model or specific publication type?

The Graphics Format table shown on the print reference page provides information on the first four of these items for each format that MATLAB supports.

Built-In MATLAB or Ghostscript Formats

Some graphics formats are built-in MATLAB formats and others are provided by Ghostscript. In some cases (such as the Windows Bitmap format), the format is available both as a built-in format and a Ghostscript format. In general, when this is the case, we recommend that you choose the MATLAB format, especially if you plan to read the image back into MATLAB later.

The choice of MATLAB versus Ghostscript is important when any of these properties affect your output:

- Font support
- Resolution
- Importing back into MATLAB

Bitmap or Vector Graphics

MATLAB file formats are created using either bitmap or vector graphics. Bitmap formats store graphics as matrices of pixels. Vector formats use drawing commands to store graphics as geometric objects. Whether to use a bitmap or vector format depends mostly on the type of objects in your figure.

The choice of bitmap versus vector graphics is important when any of these properties or capabilities affect your output:

- Degree of complexity
- Lighting and transparency

- Line and text quality
- File size
- Resizing after import

Bit Depth

Bit depth is the number of bits a format uses to store each pixel. This determines the number of colors the exported figure can contain.

Bit depth applies mostly to bitmap graphics. An 8-bit image uses 8 bits per pixel (bpp), enabling it to define 2^8 , or 256 unique colors. The other supported bit depths are 1-bit (2 colors), 4-bit (16 colors), and 24-bit (16 million colors).

In vector files that don't normally have a bit depth, the color of objects is specified by drawing commands stored in the file. However, vector files can contain bitmaps under the following conditions:

- Image objects saved in vector formats are always saved as bitmaps, regardless of the rendering method used.
- For vector files created using the OpenGL or Z-buffer renderer, everything in the figure is saved as a bitmap.

The Graphics Format table on the print reference page indicates the bit depth of each format. If file size is not critical, make sure you choose a format with a bit depth that supports the number of colors or shades of gray in your displayed figure.

Color Support

Each graphics format can produce either color, grayscale, or monochrome output. Check the Graphics Format table to see the level of color support for each format type. To preserve the color in your exported file, you must select a color graphics format. Color is also affected by bit depth.

Simulink Models

Simulink models can only be exported to EPS or a Ghostscript format. Note that you can only use the print function to export a model, not the **Export** dialog box.

High Resolution or Web Publications

If you want to use a figure in a journal or other publication, use a format that enables you to set a high resolution. We recommend using either TIFF or EPS.

If you want to use a figure in a Web publication, you should use either the PNG or the JPEG format. Note that if you need a GIF image, you can export your figure as a TIFF file and convert it to a GIF using another software application.

Properties Affected by Choice of Format

The figure properties listed in this section are affected when you select a graphics format when exporting to a file or the Windows clipboard.

Font Support

Ghostscript formats support a limited number of fonts. If you use an unsupported font, MATLAB substitutes Courier. See “PostScript and Ghostscript Supported Fonts” on page 6-71 for more information.

Resolution

Generally, higher resolution means higher quality. Your choice of resolution should be based in part on the device to which you will ultimately print it. Experimentation with different resolution settings can be helpful.

You cannot change the resolution of a Ghostscript format. The resolution is low (72 dpi) and may not be appropriate for publications.

Importing Into MATLAB

If you want to read an exported figure back into MATLAB, it is best to use one of the built-in MATLAB formats.

Degree of Complexity

Bitmaps are preferable for high-complexity plots, where complexity is determined by the number of polygons, the number of polygons with interpolated shading, the number of markers, the presence of truecolor images, and other factors. An example of a high-complexity plot is a surface plot that uses interpolated shading.

Vector formats are preferable for most 2-D plots and for some low-complexity surface plots.

Lighting and Transparency

Surface lighting and transparency are only supported by bitmap graphics formats. If you use a vector format, the lighting and transparency disappear. Note that of the two renderers intended for bitmaps (OpenGL and Z-buffer) only OpenGL supports transparency.

Note If you export to an EPS (vector) file using the Painter's renderer and include a TIFF preview, the preview image is a bitmap and will show lighting or transparency when displayed on your screen. Remember that the underlying format vector file, which is what normally gets printed, does not support these features.

Lines and Text

Generally, vector formats create better lines and text than bitmap formats.

File Size

In general, bitmap formats produce smaller files for complex plots than vector formats, and vector formats produce smaller files for simple plots than bitmap formats.

You can calculate the size of a figure exported to an uncompressed bitmap by multiplying the figure size by its resolution and the bit depth of the chosen format. For example, if a figure is 2 inches by 3 inches and has a resolution of 100 dpi (dots per inch), it will consist of $(2 \times 100) \times (3 \times 100)$, or 60,000 pixels. If exported to an 8-bit file, it uses 480,000 bits, or 60KB. If exported to a 24-bit file, it uses three times the number of bytes, or 180KB.

Vector format file size is affected by the complexity and number of objects in your figure. As the complexity and number of objects increase, the number of drawing commands increases.

Resizing After Import

You can resize a vector graphics figure after importing it into another software application without losing quality. (Not all applications that support vector formats enable you to resize them.)

This is not true of bitmap formats. Resizing a bitmap causes round-off errors that result in jagged edges and degradation of picture quality. This degradation is particularly obvious in lines and text and is highly discouraged.

Color

The Graphics Format table on the print reference page indicates the color support and bit depth of each format. If file size is not critical, make sure you choose a format with a bit depth that supports the number of colors or shades of gray in your displayed figure.

Impact of Rendering Method on the Output

If you specify a bitmap format when exporting, the exported file will always contain a bitmap regardless of your current renderer setting. If you have the renderer set to Painter's, which normally produces a vector format, that setting is ignored under these circumstances.

Vector format files, however, can store your figure as a vector or bitmap graphic depending on the renderer used to export it. If you do not specify a rendering method and MATLAB chooses the OpenGL or Z-buffer renderer, your exported vector file will contain a bitmap. If you want your figure exported as a vector graphic, be sure to set the rendering method to Painter's.

Description of Selected Graphics Formats

This section contains details about some of the export file formats MATLAB supports. For information about formats not listed here, consult a graphics file format reference.

Formats covered in this section are

- “Adobe Illustrator 88 Files” on page 6-64
- “EMF Files” on page 6-64
- “EPS Files” on page 6-64
- “TIFF Files” on page 6-66
- “JPEG Files” on page 6-66

Adobe Illustrator 88 Files

Adobe Illustrator (ILL) is a vector format that is fully compatible with Adobe Illustrator software. An Illustrator file created in MATLAB can be further processed with Adobe Illustrator running on any platform. (Note that when you view it in Illustrator, it will have no template.)

By default, Illustrator files are color and saved in portrait orientation. The Illustrator group command is used to give the illustrations a hierarchy similar to that of the Handle Graphics or Simulink graphic.

Some limitations of the Illustrator format are

- Interpolated patches and surfaces cannot be created. The color of each polygon will be determined by the average of the CData values for all of the polygon's vertices.
- Images cannot be exported in this format.
- The resolution setting of 72 dpi cannot be changed.
- No fonts are downloaded to the Illustrator file. Any unavailable fonts will be substituted with available fonts.

EMF Files

Enhanced Metafiles (EMF) are vector files similar in nature to Encapsulated PostScript (EPS), capable of producing near publication-quality graphics. EMF is an excellent format to use if you plan to import your image into a Microsoft application and want the flexibility to edit and resize your image once it has been imported. It is the only MATLAB supported vector format you can edit from within a Microsoft application. (Note that your editing ability is limited. For the best results, do all of your editing in MATLAB.)

A drawback of using EMF files is that they are generally only supported by Windows-based applications.

EPS Files

The Encapsulated PostScript (EPS) vector format is the most reliable and consistent file format MATLAB supports. It is widely recognized in desktop publishing and word processing packages on both UNIX and Windows platforms. EPS is the only MATLAB supported export format that can produce CMYK output. (PostScript printer drivers also support this feature.)

This format is your best choice for producing publication-quality graphics. It may not be appropriate for figures containing interpolated shading because it creates a very large file that is difficult to print. For such figures, use the TIFF format with a high resolution setting. For more information about format choices, see “Bitmap or Vector Graphics” on page 6-59.

When imported into Microsoft applications, an EPS file will not display unless you add a TIFF preview image to it.

The preview image is simple to add (see the next section, “Creating a Preview Image”). However, if you print your file to a nonPostScript printer, the TIFF preview is used as the printed image. The resolution of the preview image is 72 dpi, resulting in much lower quality than the EPS image. If there is no preview image, your printout to a nonPostScript printer contains an error message in place of the graphic. Many high-end graphics packages, like Adobe Illustrator, can print an EPS file to a nonPostScript printer.

When using EPS files in Microsoft applications, figures cannot be edited; they can only be annotated.

Note The best vector format to use with Microsoft applications is EMF. See “EMF Files” on page 6-64.

EPS format has limited font support. When MATLAB exports a graphic to the EPS file format, it does not try to determine whether the fonts you have used in your axes text objects are supported by the EPS format. Unsupported fonts are substituted with Courier.

Creating a Preview Image. You cannot create TIFF preview images using the graphical user interface. Use the `print` command with the `-tiff` switch. For example, to create an EPS Level 2 image with TIFF preview in file `myfile.eps`, type

```
print -depsc2 -tiff myfile.eps
```

TIFF Files

The Tagged Image File Format (TIFF) is a very widely used bitmap format and can produce publication-quality graphics if you use a high-resolution setting (such as 200 or 300 dpi).

TIFF is a good format to choose if you are not sure what formats your target application supports, or if you want to import the graphic into more than one application without having to export it to several different formats. It can also be imported into most image-processing applications and converted to other formats, if necessary. For example, MATLAB does not produce GIF files (due to patent restrictions), but there are many applications that can convert TIFF files to GIF.

JPEG Files

The Joint Photographic Experts Group (JPEG) bitmap format is one of the dominant formats used in Web graphics. The 24-bit version MATLAB supports more colors than the popular GIF format.

JPEG files always use JPEG compression. This is a lossy compression scheme, meaning that some data is thrown away during compression. When you export to a JPEG image, you can set the amount of compression to use. The more compression you use, the more data is thrown away. The compression amount is referred to as JPEG quality, where the highest setting results in the highest quality image, but the lowest amount of compression.

Setting JPEG Quality. You cannot set the quality using the graphical user interface. Use the `print` command with the `-djpeg` format switch, including the desired quality value as a suffix. This example exports to a JPEG file using a quality setting of 100.

```
print -djpeg100 myfile.jpg
```

By default, MATLAB uses a quality setting of 75. Possible values are from 1 to 100. Note that the highest setting of 100 still results in some data loss, although the result is usually visually indistinguishable from the original.

How to Specify a Format for Exporting

To select a graphics format to use when exporting, choose a format from the Graphics Format table on the print reference page, and specify that format in either the **Export** dialog box or in the MATLAB `print` function.

Using the Graphical User Interface

When exporting your figure to a file,

- 1** Select **Export** from the figure window's **File** menu.
- 2** Select a format from the **Save as type** list box.
- 3** Enter the filename you want to use and browse for the directory to save the file in.
- 4** Click **Save**.

Using MATLAB Commands

To specify a nondefault graphics format for the figure you are exporting, include the `-d` switch with the `print` command. For example, to export the current figure to file `spline2d.eps`, with 600 dpi resolution, and using the EPS color graphics format, type

```
print -r600 -depsc spline2d
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Choosing a Printer Driver

A MATLAB printer driver formats your figure into instructions that your printer understands. See the Printer Driver table on the print reference page for a complete list of drivers. Specifying the printer driver does not change the selected printer.

This section provides information to help you decide which printer driver to use when printing your figure. It covers

- “Types of Printer Drivers”
- “Factors to Consider in Choosing a Driver” on page 6-69
- “Driver-Specific Information” on page 6-72
- “How to Specify the Printer Driver to Use” on page 6-76

Types of Printer Drivers

There are two main types of MATLAB printer drivers: built-in MATLAB, and Ghostscript.

Built-in MATLAB Drivers

Built-in MATLAB drivers are written specifically for MATLAB and include Windows, PostScript, and HPGL.

MATLAB provides built-in Windows printer drivers so that your print requests can work with the Windows Print Manager. The Print Manager enables you to monitor printer queues and control various aspects of the printing process.

HPGL support is provided for the HP 7475A plotter and fully compatible plotters. HPGL files can also be imported into documents of other applications, such as Microsoft Word.

Ghostscript Drivers

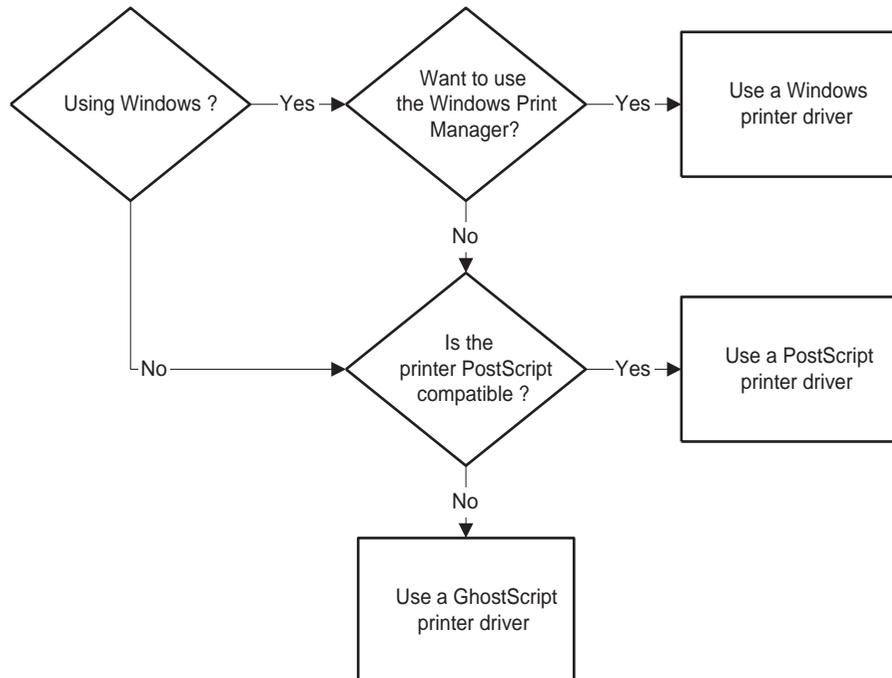
Ghostscript drivers use Ghostscript, a free software conversion program, to convert your figure into printer-model-specific instructions. Examples of Ghostscript drivers are Epson and HP.

Factors to Consider in Choosing a Driver

The choice of printer driver depends upon several considerations:

- What platform you are using
- What kind of printer you have
- What color model you want to use
- What font support you need
- Any driver-specific settings you need

The flow chart below gives an overview of how to choose a driver based on the platform you are using and the type of printer you have.



Deciding What Type of Printer Driver to Use

Platform Considerations

On Windows, you can use any of the driver types shown in the figure above. If you use the Windows driver, you can use the Windows Print Manager.

On UNIX, you can use either PostScript or Ghostscript drivers.

On either platform, if you have a PostScript-compatible printer, it is better to use a PostScript driver than a Ghostscript driver. PostScript is less prone to printing errors than Ghostscript.

Printer Type

Printer support is different among the Windows, PostScript, and Ghostscript drivers. Consult the manual for your printer to see what driver to use.

Windows drivers support most printer models, but sometimes the printer's native driver is incompatible with the MATLAB Windows driver. If you are getting printing errors, see "Trouble with Native Drivers on Windows" on page 6-74. If lines and text in your figure are not printing with the desired color scheme, see "Correcting Color Results with Windows Drivers" on page 6-73.

Some Ghostscript drivers are specific to certain printer models. For example, MATLAB provides different drivers to support the HP DeskJet 500, 500C, and 550C models, plus a generic driver for the series. When this is the case, try the model-specific driver first. If that doesn't work, try the generic driver.

Color Model

By default, MATLAB uses a black-and-white driver. The built-in MATLAB and Ghostscript drivers print both color and black and white. The Printer Drivers table on the print reference page indicates which drivers are color.

Colored surfaces and images print in grayscale when you use a black-and-white driver. Colored lines and text can be printed in color, grayscale, or black and white, depending on the color support of the driver and color capability of your printer.

Font Support

In MATLAB, the fonts supported for printing depend upon the MATLAB printer driver you specify and sometimes upon which platform you are using.

PostScript and Ghostscript Supported Fonts. The table below lists the fonts supported by the MATLAB PostScript and Ghostscript drivers. This same set of fonts is supported on both Windows and UNIX. If you use a font that is not on this list, it will be substituted with Courier.

AvantGarde	Helvetica-Narrow	Times-Roman
Bookman	NewCenturySchlbk	ZapfChancery
Courier	Palatino	ZapfDingbats
Helvetica	Symbol	

If you set the font using the `set` function, use the names exactly as shown above. This example sets the font of the current text object to Helvetica-Narrow using MATLAB commands.

```
set(gca, 'FontName', 'Helvetica-Narrow');
```

If you use the **Property Editor** dialog box (available under **Axes Properties** or **Current Object Properties** on the **Edit** menu) to set the font, the list of available fonts shows those that are supported by your system. If you choose one that is not in the table above, your resulting file will use Courier.

Windows Drivers Supported Fonts. The MATLAB Windows drivers support any system-supported font. To see the list of fonts installed on your system, open the **Font name** list on the **Text** or **Style** tab of the Property Editor.

If you use the `set` function to set fonts, type in the name just as it appears in the Property Editor. For example, if you have the Script font installed on your system, set the title of your figure to Script using the following code.

```
h = get(gca, 'Title');
set(h, 'FontName', 'Script');
```

HPGL Driver Supported Fonts. HPGL drivers support only one font. However, you can set its size and color.

Settings That Are Driver Specific

Some print settings are only supported by specific drivers. This table summarizes the settings and which driver supports them.

Setting	Driver(s)
Appending figures to a PostScript file	PostScript
BoundingBox (setting figure to print uncropped)	PostScript, Ghostscript
CMYK	PostScript
Resolution set with user interface	PostScript, Windows
Resolution set with print function	PostScript, Ghostscript

Driver-Specific Information

This section provides additional information about the various types of printer drivers available to MATLAB users. It covers the following topics:

- “Setting the Windows Driver”
- “Correcting Color Results with Windows Drivers” on page 6-73
- “Trouble with Native Drivers on Windows” on page 6-74
- “Level 1 or Level 2 PostScript Drivers” on page 6-74
- “Early PostScript 1 Printers” on page 6-74
- “Background Fills in HPGL Drivers” on page 6-75
- “Color Selection in HPGL Drivers” on page 6-75
- “Limitations of HPGL Drivers” on page 6-76

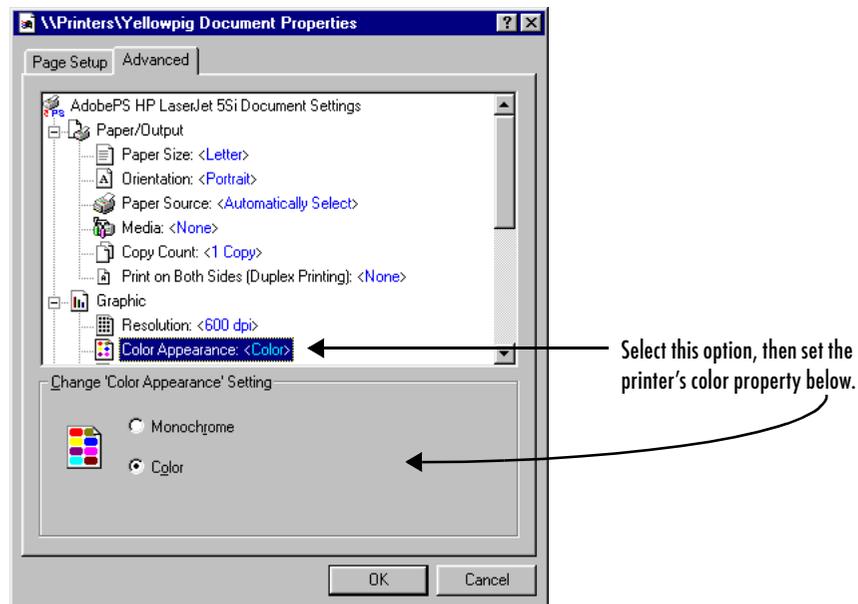
Setting the Windows Driver

When you specify a Windows driver (`-dwin` or `-dwinc`), MATLAB interprets this to mean that the print request will use the Windows Print Manager. It also means that MATLAB will assign the default Windows driver based on your current printer’s color property setting. In other words, MATLAB does not differentiate between `-dwin` or `-dwinc` in `printopt.m` and you might not get the expected output color.

There are two ways to ensure that MATLAB uses `-dwin` or `-dwinc`: specify the driver when you print, or use the Windows **Document Properties** dialog box to set the default driver.

You can use the printer's **Document Properties** dialog box to set the default driver for all print requests. This dialog box sets the printer's color property, which in turn sets the default Windows driver.

To access this dialog box, click the **Options** button on the Windows **Print** or **Print Setup** dialog box. See your Windows documentation if you need help with this dialog box. **Document Properties** dialog boxes vary from printer to printer. The figure below shows an example of one.



Windows Document Properties Dialog Box for an HP LaserJet 55i

Correcting Color Results with Windows Drivers

Sometimes, even when you use the Windows **Document Properties** dialog box, you can receive incorrect color results because some Windows printers return inaccurate information about their color property setting.

If this happens, you can use the figure window's **Preferences** dialog box to override the color property. This setting is used as the default setting for all future print operations:

- 1 Select **Preferences** from the figure window's **File** menu or from the **File** menu on the MATLAB command window.
- 2 Select **General Preferences**.
- 3 Under the **Figure Window Printing** panel, select **Always send as color**.
- 4 Click **OK**.

Trouble with Native Drivers on Windows

Occasionally, printing problems are due to a bug in the native printer driver or an incompatibility between the native printer driver and the MATLAB driver.

If you are having trouble, try installing a different native printer driver. A newer version may be available from the manufacturer or from a different vendor. You may also be able to use the native driver from a different printer, such as an earlier model from the same manufacturer.

If this doesn't help, try using a PostScript or Ghostscript driver.

Level 1 or Level 2 PostScript Drivers

Choosing between the Level 1 or Level 2 MATLAB PostScript drivers does not affect the quality of your output. Make the choice based on what your printer supports and on any file size or speed concerns.

Level 1 PostScript produces good results on a Level 2 printer, but Level 2 PostScript will not print properly on a Level 1 printer.

Level 2 PostScript files are generally smaller and render more quickly than Level 1 files. If your printer supports Level 2 PostScript, use one of the Level 2 drivers. If your printer does not support Level 2, or if you're not sure, use a Level 1 driver.

Early PostScript 1 Printers

If you have an early PostScript 1 printer, such as some of the PostScript printers manufactured before 1990, you may notice problems in the text of MATLAB printouts. Your printer may not support the `ISOlatin1Encoding`

operator that MATLAB uses for PostScript files. If this is the case, use Adobe's PostScript default character-set encoding:

- UNIX only: Open the **Print** dialog box and click **Options**. This opens the **Printing Options** dialog box. Select **Use Adobe PS default character set encoding**.
- Specify the `-adobecset` option with the `print` command.

Background Fills in HPGL Drivers

The HPGL driver cannot do background fills. Therefore, you should ensure that your figure is set to print with a white background (the default), and that any lines and text in your figure will be drawn in a color dark enough to be seen on a white background. For more information about background color, see "Setting the Background Color" on page 6-49.

Color Selection in HPGL Drivers

The HP 7475A plotter supports six pens, none of which can be white. If MATLAB tries to draw in white while rendering in HPGL mode, the driver ignores all drawing commands until a different color is chosen.

Pen 1, which is assumed to be black, is used for drawing axes. The remaining pens are used for the first five colors specified in the `ColorOrder` property of the current axes object. If `ColorOrder` specifies fewer than five colors, the unspecified pens are not used.

For Simulink systems, which ordinarily use a maximum of eight colors, the six pens available on the plotter are assumed to be

- Pen 1: black
- Pen 2: red
- Pen 3: green
- Pen 4: blue
- Pen 5: cyan
- Pen 6: magenta

If you attempt to draw a MATLAB object containing a color that is not a known pen color, the driver chooses the nearest approximation to the unlisted color.

Limitations of HPGL Drivers

The HPGL driver has these limitations:

- Display colors and plotted colors sometimes differ.
- Areas (faces on mesh and surface plots, patches, blocks, and arrowheads) are not filled.
- There is no hidden line or surface removal.
- Text is printed in the plotter's default font.
- Line width is determined by pen width.
- Images and UI controls cannot be plotted.
- Interpolated edge lines between two vertices are drawn with the pen whose color best matches the average color of the two vertices.
- Figures cannot be rendered using Z-buffer or OpenGL; this driver always uses the Painter's algorithm.

How to Specify the Printer Driver to Use

If you need to use a driver other than the default driver for your system, choose a new driver from the Printer Driver table on the `print` reference page, and set it either as a new default or just for the current figure you are working on.

Setting the Default Driver for All Figures

If you do not indicate a specific printer driver, MATLAB uses the default driver specified by the variable `dev` in the `printopt.m` file. The factory default driver depends on the platform.

Platform	Factory Default Printer Driver	Driver Code
Windows	Black-and-white Windows	-dwin
UNIX	Black-and-white Level II PostScript	-dps2

To change the default driver for all figures, edit `printopt.m` and change the value for `dev` to match one of the driver codes listed in the Printer Drivers table

on the print reference page. See “Setting Defaults Across Sessions” on page 6-7 for instructions.

Setting a Driver for the Current Figure Only

You can change the printer driver using a UNIX dialog box or from the MATLAB command line.

Using the Graphical User Interface on UNIX. To specify a printer driver for the current figure:

- 1** From the figure window’s **File** menu, select **Print**.
- 2** Select a printer driver from the **Driver** list.
- 3** Click **OK** to print your figure.

Using MATLAB Commands. To specify a nondefault printer driver for the figure you are printing, include the `-d` switch with the `print` command. For example, to print the current figure using the MATLAB built-in Windows color printer driver, `winc`, type

```
print -dwinc
```

Note When printing, the `print -d` option specifies a printer driver. When exporting, the `print -d` option specifies a graphics format.

Troubleshooting

This section lists some common problems you might encounter when printing or exporting your figure. Refer to the appropriate section listed below:

- **Printing Problems**

- “Printer Drivers” on page 6-79
- “Default Settings” on page 6-80
- “Line Style” on page 6-80
- “Color vs. Black and White” on page 6-81
- “Printer Selection” on page 6-82
- “Rotated Text” on page 6-82
- “ResizeFcn Warning” on page 6-82

- **Exporting Problems**

- “Background Color” on page 6-82
- “Default Settings” on page 6-83
- “Microsoft Word” on page 6-83
- “File Format” on page 6-84
- “Size of Exported File” on page 6-84
- “Making Movies” on page 6-85
- “Extended Operations” on page 6-85

- **General Problems**

- “Background Color” on page 6-86
- “Default Settings” on page 6-86
- “Dimensions of Output” on page 6-86
- “Axis and Tick Labels” on page 6-87
- “UI Controls” on page 6-88
- “Cropping” on page 6-88
- “Text Object Font” on page 6-88

If you don’t find your problem listed here, try searching the Knowledge Base maintained by the MathWorks Technical Support Department. Go to <http://www.mathworks.com/support> and enter a topic in the search field.

Printing Problems

Printer Drivers

I'm using a Windows printer driver and have been encountering problems such as segmentation violations, general protection faults, application errors, and unexpected output.

Try one of the following solutions:

- Check the table of drivers in the print reference page to see if there are other drivers you can try.
 - If your printer is PostScript compatible, try printing with one of the MATLAB built-in PostScript drivers.
 - If your printer is not PostScript compatible, see if one of the MATLAB built-in Ghostscript devices is appropriate for your printer model. These devices use Ghostscript to convert PostScript files into other formats, such as HP LaserJet and Canon BubbleJet.
- Contact the printer vendor to obtain a different native printer driver. The behavior you are experiencing may occur only with certain versions of the native printer driver. If this doesn't help and you are using Windows, try reinstalling the drivers that were shipped with your Windows installation disk.
- Export the figure to a graphics-format file, and then import it into another application before printing it. For information about exporting figures with MATLAB, see "Exporting to a File" on page 6-15.

When I use the print function with the -deps switch, I receive this error message.

```
Encapsulated PostScript files cannot be sent to the printer.  
File saved to disk under name 'figure2.eps'
```

As the error message indicates, your figure was saved to a file. EPS is a graphics file format and cannot be sent to a printer using a printer driver. To send your figure directly to a printer, try using one of the PostScript driver switches. See the table of drivers in the print reference page. To print an EPS file, you must first import it into a word processor or other software program.

Default Settings

My printer uses a different default paper type than the MATLAB default type of “letter.” How can I change the default paper type so that I won’t have to set it for each new figure?

You can set the default value for any property by adding a line to `startup.m`. Adding the following line sets the default paper type to A4.

```
set(0, 'DefaultFigurePaperType', 'A4');
```

In your call to `set`, combine the word `Default` with the name of the object `Figure` and the property name `PaperType`.

I set the paper orientation to landscape, but each time I go to print a new figure, the orientation setting is portrait again. How can I change the default orientation so that I won’t have to set it for each new figure?

See the explanation for the question above. Adding the following line to `startup.m` sets the default paper orientation to landscape.

```
set(0, 'DefaultFigurePaperOrient', 'landscape')
```

Line Style

My figure contains lines that use broken line styles. However, these lines print as solid lines.

Microsoft Windows 98 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 98 drivers produce solid lines, regardless of the setting of `LineStyleOrder`.

There are various ways you can work around this problem:

- Change the MATLAB default line width to 1 pixel wide, by adding this line to the `[MATLAB Settings]` section of your `MATLAB.INI` file.

```
ThinLineStyle=1
```

This will result in thin lines, but the lines will print with the specified styles.

- Set the figure's `Renderer` property to `OpenGL`.

```
set(gcf, 'Renderer', 'OpenGL')
```

The printed output will match the displayed figure. See the section on “Selecting a Renderer” on page 6-41 for more information about the OpenGL renderer.

- Use a PostScript or Ghostscript printer driver. These drivers bypass the Windows Print Manager. See “Choosing a Printer Driver” on page 6-68.

Color vs. Black and White

I want the lines in my figure to print in black, but they keep printing in color.

You must be using a color printer driver. You can specify a black-and-white driver using the `print` function or the **Page Setup** dialog box to force the lines for the current figure to print in black. See “Setting the Line and Text Color” on page 6-52 for instructions.

A white line in my figure keeps coming out black when I print it.

There are two things that can cause this to happen. Most likely, the line is positioned over a dark background. By default, MATLAB inverts your background to white when you print, and changes any white lines over the background to black. To avoid this, retain your background color when you print. See “Setting the Background Color” on page 6-49.

The other possibility is that you are using a Windows printer driver and the printer is sending inaccurate color information to MATLAB. See “Correcting Color Results with Windows Drivers” on page 6-73.

I am using a color printer, but my figure keeps printing in black and white.

By default, MATLAB uses a black-and-white printer driver. You need to specify a color printer driver. For instructions, see “Choosing a Printer Driver” on page 6-68. If you are already using a Windows color driver, the printer may be returning inaccurate information about its color property. See “Correcting Color Results with Windows Drivers” on page 6-73.

Printer Selection

I have more than one printer connected to my system. How do I specify which one to print my figure with?

You can use either the **Print** dialog box, or the MATLAB print function, specifying the printer with the `-P` switch. For instructions using either method, see “Selecting the Printer” on page 6-33.

Rotated Text

I have some rotated text in my figure. It looks fine on the screen, but when I print it, the resolution is poor.

You are probably using bitmapped fonts, which don't rotate well. Try using TrueType fonts instead.

ResizeFcn Warning

I get a warning about my ResizeFcn being used when I print my figure.

By default, MATLAB resizes your figure when converting it to printer coordinates. Therefore, MATLAB calls any `ResizeFcn` you have created for the figure and issues a warning. You can avoid this warning by setting the figure to print at screen size.

Exporting Problems

Background Color

I generated a figure with a black background and selected “Use figure color” from the Copy Options panel of the Preferences dialog box. But when I exported my figure, its background was changed to white.

You must have exported your figure to a file. The settings in **Copy Options** only apply to figures copied to the clipboard.

There are two ways to retain the displayed background color: use the **Page Setup** dialog box or set the `InvertHardCopy` property to `off`. See “Setting the Background Color” on page 6-49 for instructions on either method.

Default Settings

I want to export all of my figures using the same size. Is there some way to do this so that I don't have to set the size for each individual figure?

You can set the default value for any property by adding a line to `startup.m`. Adding the following line sets the default figure size to 4-by-3 inches.

```
set(0, 'DefaultFigurePaperPosition', [0 0 4 3]);
```

In your call to `set`, combine the word `Default` with the name of the object `Figure` and the property name `PaperPosition`.

I use the clipboard to export my figures as metafiles. Is there some way to force all of my copy operations to use the metafile format?

Use the **Copy Options** panel of the **Preferences** dialog box. Any settings made here, including whether MATLAB copies your figure as a metafile or bitmap, apply to all copy operations. See “Exporting to the Windows Clipboard” on page 6-20 for instructions.

Microsoft Word

I exported my figure to an EPS file, and then tried to import it into my Word document. My printout has an empty frame with an error message saying that my EPS picture was not saved with a preview and will only print to a PostScript printer. How do I include a TIFF preview?

Use the `print` command with the `-tiff` switch. For example:

```
print -deps -tiff filename
```

Note that if you print to a nonPostScript printer with Word, the preview image is used for printing. This is a low-resolution image that lacks the quality of an EPS graphic. For more information about preview images and other aspects of EPS files, see “EPS Files” on page 6-64.

When I try to resize my figure in Word, its quality suffers.

You must have used a bitmap format. Bitmap files generally do not resize well. If you are going to export using a bitmap format, try to set the figure's size while it's still in MATLAB. See “Setting the Figure Size and Position” on page 6-34 for instructions.

As an alternative, you can use one of the vector formats, EMF or EPS. Figures exported in these formats can be resized in Word without affecting quality.

I exported my figure as an EMF to the clipboard. When I paste it into Word, some of the labels are printed incorrectly.

This problem occurs with some versions of Word and Windows. Try editing the labels in Word.

File Format

I tried to import my exported figure into a word processing document, but I got an error saying the file format is unrecognized.

There are two likely causes: you used the `print` function and forgot to specify the export format, or your word processing program does not support the export format. Include a format switch when you use the `print` function; simply including the file extension is not sufficient. For instructions, see “Exporting to a File” on page 6-15.

If this does not solve your problem, check what formats the word processor supports.

I tried to append a figure to an EPS file, and received an error message.

You cannot append figures to an EPS file. The `-append` option is only valid for PostScript files, which should not be confused with EPS files. PostScript is a printer driver; EPS is a graphics file format.

Of the supported export formats, only HDF supports storing multiple figures, but you must use the `imwrite` function to append them. For an example, see the reference page for `imwrite`.

Size of Exported File

I've always used the EPS format to export my figures, but recently it started to generate huge files. Some of my files are now several megabytes!

Your graphics have probably become complicated enough that MATLAB is using the OpenGL or Z-buffer renderer instead of the Painter's renderer. It does this to improve display time or to handle attributes that Painter's cannot, such as lighting. However, using OpenGL or Z-buffer causes a bitmap to be stored in your EPS file, which sometimes leads to a large file.

There are two ways to fix the problem. You can specify the Painter's renderer when you export to EPS, or you can use a bitmap format, such as TIFF. The best renderer and type of format to use depend upon the figure. See "Bitmap or Vector Graphics" on page 6-59 if you need help deciding. For information about the rendering methods and how to set them, see the section on "Selecting a Renderer" on page 6-41.

Making Movies

I am processing a large number of frames in MATLAB. I would like these frames to be saved as individual files for later conversion into a movie. How can I do this?

Use `getframe` to capture the frames, `imwrite` to write them to a file, and `movie` to create a movie from the files. For more information about using `getframe` and `imwrite` to capture and write the frames, see "Exporting with `getframe`" on page 6-17. For more information about creating a movie from the captured frames, see the reference page for `movie`.

You can also save multiple figures to an AVI file. AVI files can be used for animated sequences that do not need MATLAB to run. However, they do require an AVI viewer. For more information, see "Creating an AVI Format Movie" in the "Development Environment" section of the Using MATLAB documentation.

Extended Operations

There are some Export operations that cannot be performed using the Export dialog box.

You need to use the `print` function to do any of the following operations:

- Export to a supported file format not listed in the **Export** dialog box. The formats not available from the **Export** dialog box include HDF, some variations of BMP and PCX, and the raw data versions of PBM, PGM, and PPM.
- Specify a resolution.
- Specify one of the following options: TIFF preview, loose bounding box for EPS files, compression quality for JPEG files, CMYK output on Windows.
- Perform batch exporting.

General Problems

Background Color

When I output my figure, its background is changed to white. How can I get it to have the displayed background color?

By default, when you print or export a figure, MATLAB inverts the background color to white. There are two ways to retain the displayed background color: use the **Page Setup** dialog box or set the `InvertHardCopy` property to `off`. See “Setting the Background Color” on page 6-49 for instructions on either method.

If you are exporting your figure to the clipboard, you can also use the **Copy Options** panel of the **Preferences** dialog box. Setting the background here sets it for all figures copied to the clipboard.

Default Settings

I need to produce diagrams for publications. There is a list of requirements that I must meet for size of the figure, fonts types, etc. How can I do this easily and consistently?

You can set the default value for any property by adding a line to `startup.m`. As an example, the following line sets the default axes label font size to 12.

```
set(0, 'DefaultAxesFontSize', 12);
```

In your call to `set`, combine the word `Default` with the name of the object `Axes` and the property name `FontSize`.

Dimensions of Output

The dimensions of my output are huge. How can I make it smaller?

Check your settings for figure size and resolution, both of which affect the output dimensions of your figure.

The default figure size is 8-by-6 inches. You can use the **Page Setup** dialog box or the `PaperPosition` property to set the figure size. See “Setting the Figure Size and Position” on page 6-34.

The default resolution depends on the export format or printer driver used. For example, built-in MATLAB bitmap formats, like TIFF, have a default resolution of 150 dpi. You can change the resolution by using the print function and the `-r` switch. For default resolution values and instructions on how to change them, see “Setting the Resolution” on page 6-44.

I selected “Match Screen Size” from the Page Setup menu, but my output looks a little bigger, and my font looks different.

You probably output your figure using a higher resolution than your screen uses. Set your resolution to be the same as the screen’s.

As an alternative, if you are exporting your figure, see if your application enables you to select a resolution. If so, import the figure at the same resolution it was exported with. For more information about resolution and how to set it when exporting, see “Setting the Resolution” on page 6-44.

Axis and Tick Labels

When I resize my figure below a certain size, my x-axis label and the bottom half of the x-axis tick labels are missing from the output.

Your figure size may be too small to accommodate the labels. Labels are positioned a fixed distance from the x -axis. Since the x -axis itself is positioned a relative distance away from the window’s edge, the label text may not fit. Try using a larger figure size or smaller fonts. For instructions on setting the size of your figure, see “Setting the Figure Size and Position” on page 6-34. For information about setting font size, see the Text Properties reference page.

In my output, the x-axis has fewer ticks than it did on the screen.

MATLAB has rescaled your ticks because the size of your output figure is different from its displayed size. There are two ways to prevent this: select **Keep screen limits and ticks** from the **Axes and Figure** tab of the **Page Setup** dialog box, or set the `XTickMode`, `YTickMode`, and `ZTickMode` properties to manual. See “Setting the Axes Ticks and Limits” on page 6-47 for details.

UI Controls

My figure contains UI Controls. How do I prevent them from appearing in my output?

Use the `print` function with the `-noui` switch. For details, see “Excluding User Interface Controls” on page 6-55.

Cropping

I can't output my figure using the uncropped setting (i.e., a loose `BoundingBox`).

Only PostScript printer drivers and the EPS export format support uncropped output. There is a workaround for Windows printer drivers, however. Using the `print` function, save your figure to a file that can be printed later. For an example see “Producing Uncropped Figures” on page 6-55.

Text Object Font

I have a problem with text objects when printing with a PostScript printer driver or exporting to EPS. The fonts are correct on the screen, but are changed in the output.

You have probably used a font that is not supported by EPS and PostScript. All unsupported fonts are converted to Courier. See “PostScript and Ghostscript Supported Fonts” on page 6-71 for the list of the supported fonts.

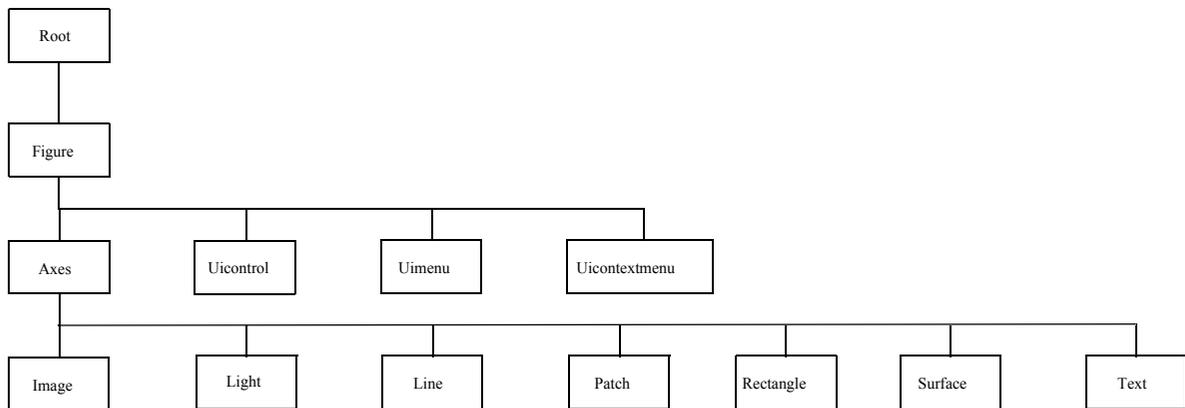
Handle Graphics Objects

Graphics Object Hierarchy (p. 7-2)	Illustration of the graphics object hierarchy.
Types of Graphics Objects (p. 7-3)	Overview of the various graphics objects.
Object Properties (p. 7-8)	What is a property and what do you do with it.
Graphics Object Creation Functions (p. 7-11)	Functions that construct graphics objects.
Setting and Querying Property Values (p. 7-17)	How to set and query property values and how to return to original (factory default) values.
Setting Default Property Values (p. 7-22)	How MATLAB determines what values to use for a given object's properties. How to define default values.
Accessing Object Handles (p. 7-29)	Obtain the handles of existing objects.
Controlling Graphics Output (p. 7-36)	Control target window for graphics output.
Saving Handles in M-Files (p. 7-47)	How to manage object handles within a graphics M-file.
Properties Changed by Built-In Functions (p. 7-48)	List of the properties that are changed by MATLAB built-in functions.
Callback Properties for Graphics Objects (p. 7-51)	Execute functions when the events described in this section occur on graphics objects.
Function Handle Callbacks (p. 7-53)	Function handles provide advantages for specifying callbacks. These advantages are illustrated through a GUI example.

Graphics Object Hierarchy

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.

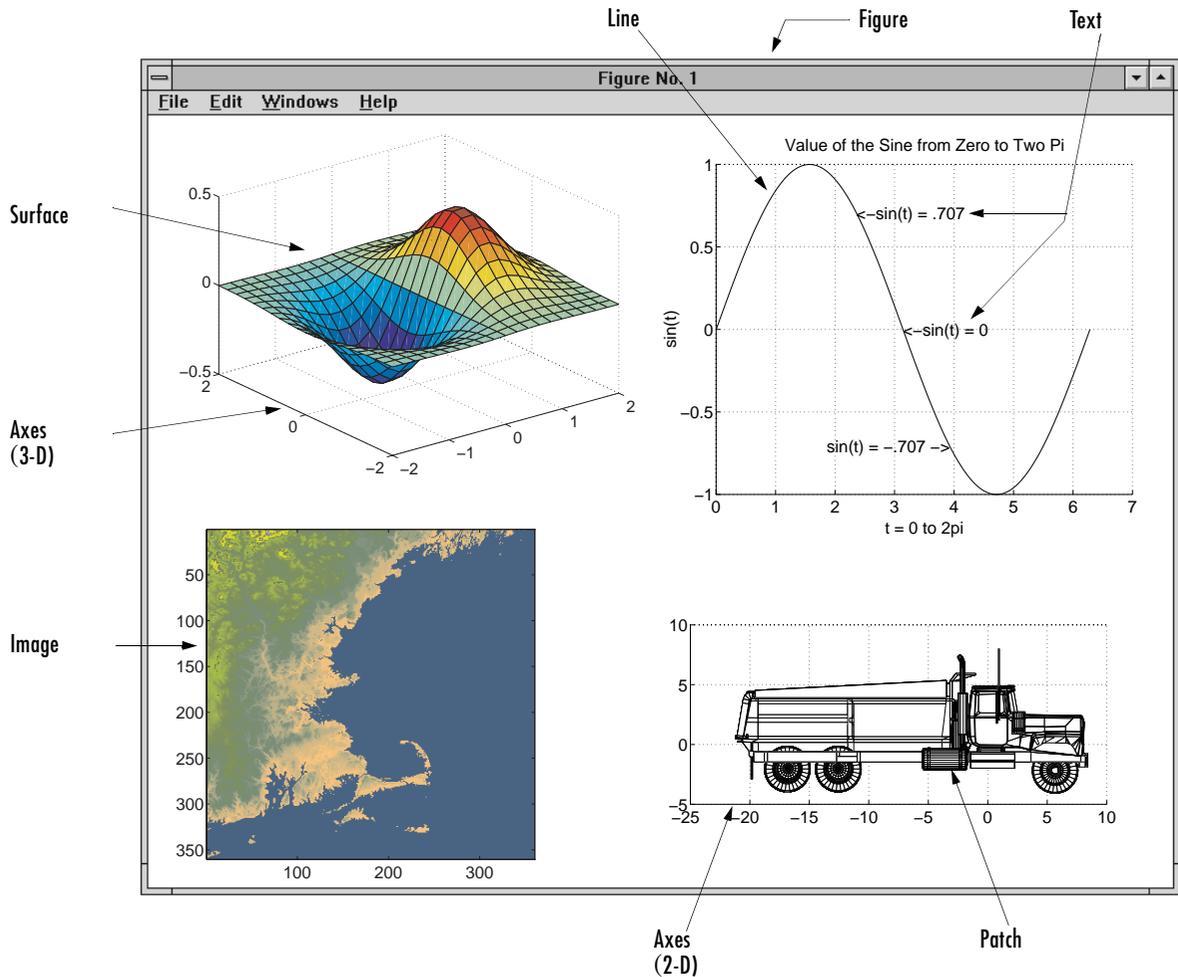
These objects are organized into a tree-structured hierarchy.



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.

Types of Graphics Objects

Graphics objects are interdependent so 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).

Handle Graphics Objects

The following list summarizes the Handle Graphics objects.

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

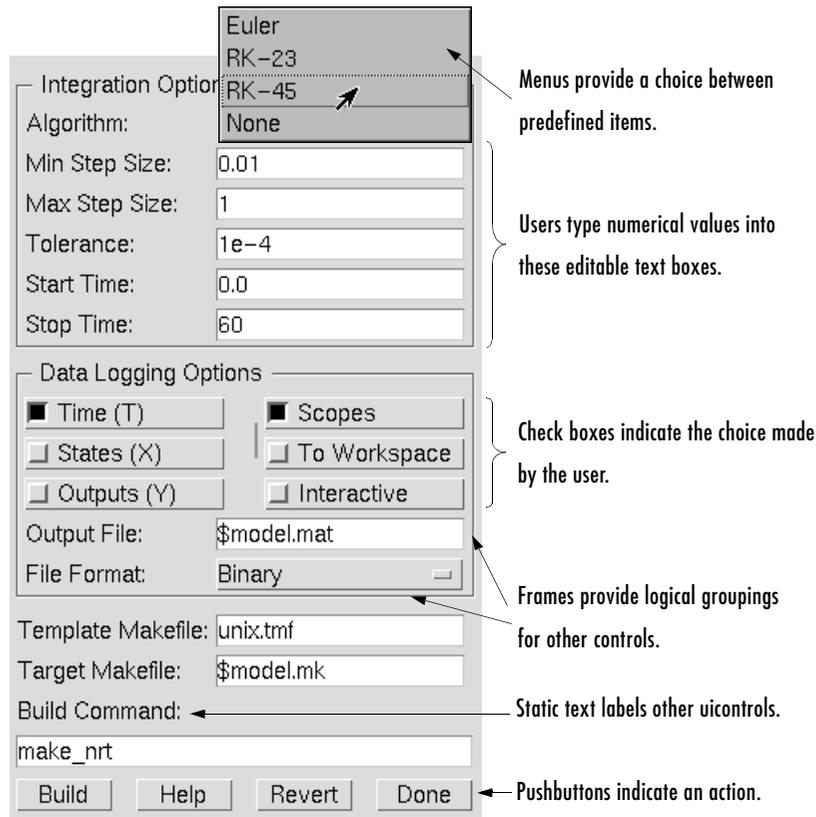
Figure objects are the individual windows on the root screen where MATLAB displays graphics. MATLAB places no limits on the number of figure windows you can create (your computer may, however). All figures are children of the root and all other graphics objects are descendants of figures.

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.

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.

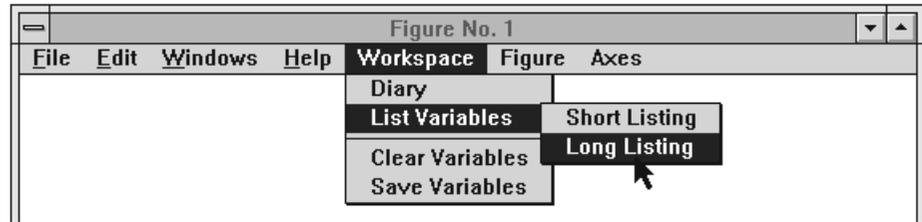
You can use uicontrols in combinations to construct control panels and dialog boxes. Pop-up menus, editable text boxes, check boxes, pushbuttons, static text, and frames compose this particular example.



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 objects 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 objects 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).

Image

A MATLAB image consists of a data matrix and possibly a colormap. There are three basic image types that differ in the way that data matrix elements are interpreted as pixel colors — indexed, intensity, and truecolor. Since images are strictly 2-D, you can view them only at the default 2-D view.

Light

Light objects define light sources that affect all patch and surface objects within the axes. You cannot see lights, but you can set properties that control the style of light source, color, location, and other properties common to all graphics objects.

Line

Line objects are the basic graphics primitives used to create most 2-D and some 3-D plots. High-level functions `plot`, `plot3`, and `loglog` (and others) create line objects. The coordinate system of the 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.

Rectangle

Rectangle objects are 2-D filled areas having a shape that can range from a rectangle to an ellipse. Rectangles are useful for creating flow-chart type drawings.

Surface

Surface objects are 3-D representations of matrix data created by plotting the value of each matrix element as a height above the x - y plane. Surface plots are composed of quadrilaterals whose vertices are specified by the matrix data. MATLAB can draw surfaces with solid or interpolated colors or with only a mesh of lines connecting the points. The coordinate system of the 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.

Changing Values

You can query the current value of any property and specify most property values (although some are set by MATLAB and are read only). Property values apply uniquely to a particular instance of an object; setting a value for one object does not change this value for other objects of the same type.

Default Values

You can set default values that affect all subsequently created objects. Whenever you do not define a value for a property, either as a default or when you create the object, MATLAB uses “factory-defined” values.

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, as illustrated in the following table.

Property	Information Contained
BusyAction	Controls the way MATLAB handles callback routine interruption defined for the particular object
ButtonDownFcn	Callback routine that executes when button press occurs
Children	Handles of all this object's children objects
Clipping	Mode that enables or disables clipping (meaningful only for axes children)
CreateFcn	Callback routine that executes when this type of object is created
DeleteFcn	Callback routine that executes when you issue a command that destroys the object
HandleVisibility	Allows you to control the availability of the object's handle from the command line and from within callback routines
Interruptible	Determines whether a callback routine can be interrupted by a subsequently invoked callback routine
Parent	The object's parent
Selected	Indicates whether object is selected
SelectionHighlight	Specifies whether object visually indicates the selection state
Tag	User-specified object label
Type	The type of object (figure, line, text, etc.)

Property	Information Contained
UserData	Any data you want to associate with the object
Visible	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 patches and surfaces.
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.
rectangle	2-D filled area having a shape that can range from a rectangle to an ellipse.
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.
uicontextmenu	Context menu that you can associate with other graphics object.

Function	Object Description
<code>uicontrol</code>	Programmable user-interface device, such as pushbutton, slider, or listbox.
<code>uimenu</code>	Programmable menu appearing at the top of a figure window.

All object creation functions have a similar format.

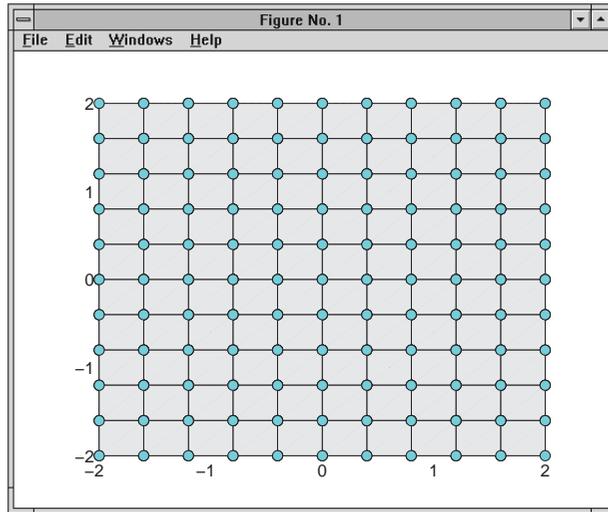
```
handle = function('propertyname',propertyvalue,...)
```

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

This code evaluates a mathematical function and creates three graphics objects using the property values specified as arguments to the `figure`, `axes`, and `surface` commands. MATLAB uses default values for all other properties.

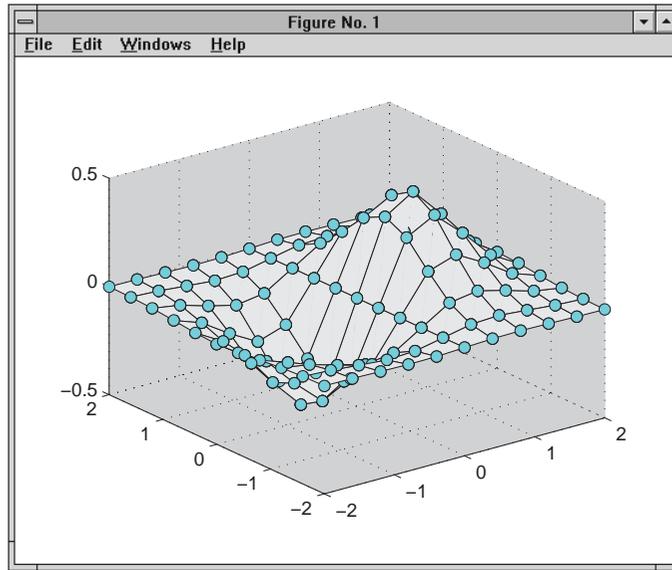
```
[x,y] = meshgrid([-2:.4:2]);  
Z = x.*exp(-x.^2-y.^2);  
fh = figure('Position',[350 275 400 300],'Color','w');  
ah = axes('Color',[.8 .8 .8],'XTick',[-2 -1 0 1 2],...  
         'YTick',[-2 -1 0 1 2]);  
sh = surface('XData',x,'YData',y,'ZData',Z,...  
           'FaceColor',get(ah,'Color')+1,...  
           'EdgeColor','k','Marker','o',...  
           'MarkerFaceColor',[.5 1 .85]);
```



Note that the surface function does not use a 3-D view like the high-level surf functions. Object creation functions simply add new objects to the current axes without changing axes properties, except the Children property, which now includes the new object and the axis limits (XLim, YLim, and ZLim), if necessary.

You can change the view using the camera commands or use the view command.

```
view(3)
```



Parenting

By default, all statements that create graphics objects do so in the current figure and the current axes (if the object is an axes child). However, you can specify the parent of an object when you create it. For example, the statement,

```
axes('Parent',figure_handle,...)
```

creates an axes in the figure identified by `figure_handle`. You can also move an object from one parent to another by redefining its `Parent` property.

```
set(gca,'Parent',figure_handle)
```

High-Level Versus Low-Level

The MATLAB 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 or axes `NextPlot` property.

For example, if you call the `line` function,

```
line('XData',x,'YData',y,'ZData',z,'Color','r')
```

MATLAB draws a red line in the current axes using the specified data values. If there is no axes, MATLAB creates one. If there is no figure window in which to create the axes, MATLAB creates it as well.

If you call the `line` function a second time, MATLAB draws the second line in the current axes without erasing the first line. This behavior is different from high-level functions like `plot` that delete graphics objects and reset all axes properties (except `Position` and `Units`). You can change the behavior of high-level functions using the `hold` command or changing the setting of the axes `NextPlot` property.

See “Controlling Graphics Output” on page 7-36 for more information on this behavior and on using the `NextPlot` property.

Simplified Calling Syntax

Object creation functions have convenience forms that allow you to use a simpler syntax. For example,

```
text(.5,.5,.5,'Hello')
```

is equivalent to,

```
text('Position',[.5 .5 .5],'String','Hello')
```

Note that using the convenience form of an object creation function can cause subtle differences in behavior when compared to formal property name/property value syntax.

A Note About Property Names

By convention, MATLAB documentation capitalizes the first letter of each word that makes up a property name, such as `LineStyle` or `XTickLabelMode`. While this makes property names easier to read, MATLAB does not check for uppercase letters. In addition, you need use only enough letters to identify the name uniquely, so you can abbreviate most property names.

In M-files, however, using the full property name can prevent problems with future releases of MATLAB if a shortened name is no longer unique because of the addition of new properties.

Setting and Querying Property Values

The `set` and `get` functions specify and retrieve the value of existing graphics object properties. They also enable you to list possible values for properties that have a fixed set of values. (You can also use the Property Editor to set many property values. See “Using the Property Editor” on page 1-15 for more information.)

The basic syntax for setting the value of a property on an existing object is

```
set(object_handle, 'PropertyName', 'NewPropertyValue')
```

To query the current value of a specific object’s property, use a statement like

```
returned_value = get(object_handle, 'PropertyName');
```

Property names are always quoted strings. Property values depend on the particular property.

See “Accessing Object Handles” on page 7-29 and the `findobj` command for information on finding the handles of existing object.

Setting Property Values

You can change the properties of an existing object using the `set` function and the handle returned by the creating function. For example, this statement moves the *y*-axis to the right side of the plot on the current axes.

```
set(gca, 'YAxisLocation', 'right')
```

If the handle argument is a vector, MATLAB sets the specified value on all identified objects.

You can specify property names and property values using structure arrays or cell arrays. This can be useful if you want to set the same properties on a number of objects. For example, you can define a structure to set axes properties appropriately to display a particular graph.

```
view1.CameraViewAngleMode = 'manual';  
view1.DataAspectRatio = [1 1 1];  
view1.ProjectionType = 'Perspective';
```

To set these values on the current axes, type

```
set(gca, view1)
```

Listing Possible Values

You can use `set` to display the possible values for many properties without actually assigning a new value. For example, this statement obtains the values you can specify for line object markers.

```
set(obj_handle, 'Marker')
```

MATLAB returns a list of values for the Marker property for the type of object specified by `obj_handle`. Braces indicate the default value.

```
[ + | o | * | . | x | square | diamond | v | ^ | > | < | pentagram  
| hexagram | {none} ]
```

To see a list of all settable properties along with possible values of properties that accept string values, use `set` with just an object handle.

```
set(object_handle)
```

For example, for a surface object, MATLAB returns

```
CData  
CDataScaling: [ {on} | off]  
EdgeColor: [ none | {flat} | interp ] ColorSpec.  
EraseMode: [ {normal} | background | xor | none ]  
FaceColor: [ none | {flat} | interp | texturemap ] ColorSpec.  
LineStyle: [ {-} | -- | : | -. | none ]  
.  
.  
.  
Visible: [ {on} | off ]
```

If you assign the output of the `set` function to a variable, MATLAB returns the output as a structure array. For example,

```
a = set(gca);
```

The field names in `a` are the object's property names and the field values are the possible values for the associated property. For example,

```
a.GridLineStyle  
ans =
```

```
'_'  
'--'
```

```

'.'
'-'
'none'

```

returns the possible 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.

Querying Property Values

Use `get` to query the current value of a property or of all the object's properties. For example, check the value of the current axes `PlotBoxAspectRatio` property.

```

get(gca, 'PlotBoxAspectRatio')

ans =
     1     1     1

```

MATLAB lists the values of all properties, where practical. However, for properties containing data, MATLAB lists the dimensions only (for example, `CurrentPoint` and `ColorOrder`).

```

AmbientLightColor = [1 1 1]
Box = off
CameraPosition = [0.5 0.5 2.23205]
CameraPositionMode = auto
CameraTarget = [0.5 0.5 0.5]
CameraTargetMode = auto
CameraUpVector = [0 1 0]
CameraUpVectorMode = auto
CameraViewAngle = [32.2042]
CameraViewAngleMode = auto
CLim: [0 1]
CLimMode: auto
Color: [0 0 0]
CurrentPoint: [ 2x3 double]
ColorOrder: [ 7x3 double]

```

```
·  
·  
·  
Visible = on
```

Querying Individual Properties

You can obtain the data from the property by getting that property individually.

```
get(gca, 'ColorOrder')  
ans =  
      0      0      1.0000  
      0      0.5000      0  
      1.0000      0      0  
      0      0.7500      0.7500  
      0.7500      0      0.7500  
      0.7500      0.7500      0  
      0.2500      0.2500      0.2500
```

Returning a Structure

If you assign the output of `get` to a variable, MATLAB creates a structure array whose field names are the object property names and whose field values are the current values of the named property.

For example, if you plot some data, `x` and `y`,

```
h = plot(x,y);
```

and get the properties of the line object created by `plot`,

```
a = get(h);
```

you can access the values of the line properties using the field name. This call to the `text` command places the string 'x and y data' at the first data point and colors the text to match the line color.

```
text(x(1),y(1), 'x and y data', 'Color', a.Color)
```

If `x` and `y` are matrices, `plot` draws one line per column. To label the plot of the second column of data, reference that line.

```
text(x(1,2),y(1,2), 'Second set of data', 'Color', a(2).Color)
```

Querying Groups of Properties

You can define a cell array of property names and conveniently use it to obtain the values for those properties. For example, suppose you want to query the values of the axes “camera mode” properties. First define the cell array.

```
camera_props(1) = { 'CameraPositionMode' };
camera_props(2) = { 'CameraTargetMode' };
camera_props(3) = { 'CameraUpVectorMode' };
camera_props(4) = { 'CameraViewAngleMode' };
```

Use this cell array as an argument to obtain the current values of these properties.

```
get(gca,camera_props)
ans =
    'auto' 'auto' 'auto' 'auto'
```

Factory-Defined Property Values

MATLAB defines values for all properties, which are used if you do not specify values as arguments or as defaults. You can obtain a list of all factory-defined values with the statement.

```
a = get(0,'Factory');
```

`get` returns a structure array whose field names are the object type and property name concatenated together, and field values are the factory value for the indicated object and property. For example, this field,

```
UimenuSelectionHighlight: 'on'
```

indicates that the factory value for the `SelectionHighlight` property on `uimenu` objects is `on`.

You can get the factory value of an individual property with,

```
get(0,'FactoryObjectTypePropertyName')
```

For example,

```
get(0,'FactoryTextFontName')
```

Setting Default Property Values

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.

How MATLAB Searches for Default Values

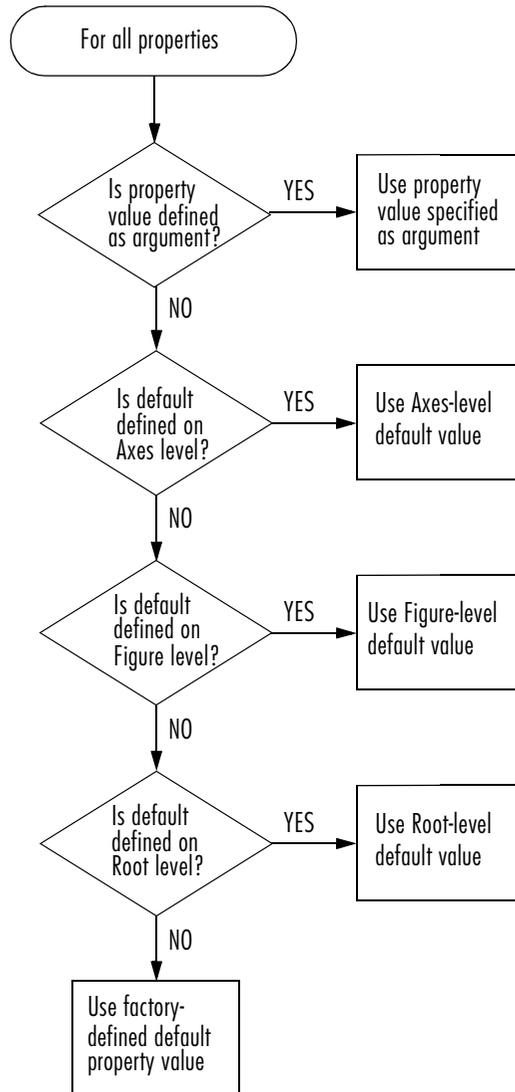
MATLAB searches for a default value beginning with the current object and continuing through the object’s ancestors until it finds a user-defined default value or until it reaches the factory-defined value. Therefore, a search for property values is always satisfied.

The closer to the root of the hierarchy you define the default, the broader its scope. If you specify a default value for line objects on the root level, MATLAB uses that value for all lines (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.



Defining Default Values

To specify default values, create a string beginning with the word `Default` followed by the object type and finally the object property. For example, to specify a default value of 1.5 points for the line `LineWidth` 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 surface `EdgeColor`,

```
set(0, 'DefaultSurfaceEdgeColor', 'k')
h = surface(peaks);
set(gcf, 'DefaultSurfaceEdgeColor', 'g')
set(h, 'EdgeColor', 'default')
```

Since a default value for surface `EdgeColor` exists on the figure level, MATLAB encounters this value first and uses it instead of the default `EdgeColor` defined on the root.

Removing Default Values

Specifying a property value of `'remove'` gets rid of user-defined default values. The statement,

```
set(0, 'DefaultSurfaceEdgeColor', 'remove')
```

removes the definition of the default `Surface EdgeColor` from the root.

Setting Properties to Factory-Defined Values

Specifying a property value of 'factory' sets the property to its factory-defined value. (The property descriptions provides access to the factory settings for properties having predefined sets of values.)

For example, these statements set the `EdgeColor` of surface `h` to black (its factory setting) regardless of what default values you have defined.

```
set(gcf, 'DefaultSurfaceEdgeColor', 'g')
h = surface(peaks);
set(h, 'EdgeColor', 'factory')
```

Reserved Words

Setting a property value to `default`, `remove`, or `factory` produces the effect described in the previous sections. To set a property to one of these words (e.g., a text or `uicontrol` `String` property set to the word `Default`), you must precede the word with the backslash character. For example,

```
h = uicontrol('Style','edit','String','\Default');
```

Examples – Setting Default LineStyles

The `plot` function cycles through the colors defined by the axes `ColorOrder` property when displaying multiline plots. If you define more than one value for the axes `LineStyleOrder` property, MATLAB increments the 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

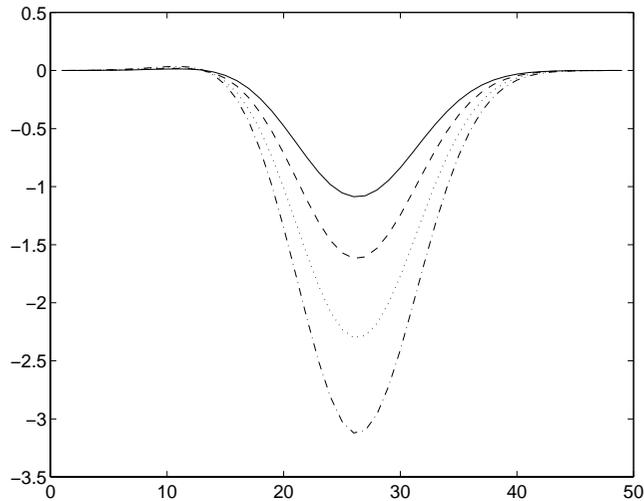
This example creates a figure with a white plot (axes) background color, then sets default values for axes objects on the root level.

```
whitebg('w') %create a figure with a white color scheme
set(0, 'DefaultAxesColorOrder', [0 0 0], ...
      'DefaultAxesLineStyleOrder', '-|---|:|-.')
```

Whenever you call `plot`,

```
Z = peaks; plot(1:49,Z(4:7,:))
```

it uses one color for all data plotted because the axes `ColorOrder` contains only one color, but cycles through the `LineStyleOrder` defined for `LineStyleOrder`.



Second Example

This example sets default values on more than one level in the hierarchy. These statements create two axes in one figure window, setting default values on the figure level and the axes level.

```
t = 0:pi/20:2*pi;
s = sin(t);
c = cos(t);
% Set default value for axes Color property
figh = figure('Position',[30 100 800 350],...
             'DefaultAxesColor',[.8 .8 .8]);

axh1 = subplot(1,2,1); grid on
% Set default value for line LineStyle property in first axes
set(axh1,'DefaultLineStyle',' .')
line('XData',t,'YData',s)
line('XData',t,'YData',c)
text('Position',[3 .4],'String','Sine')
text('Position',[2 .3],'String','Cosine',...
```

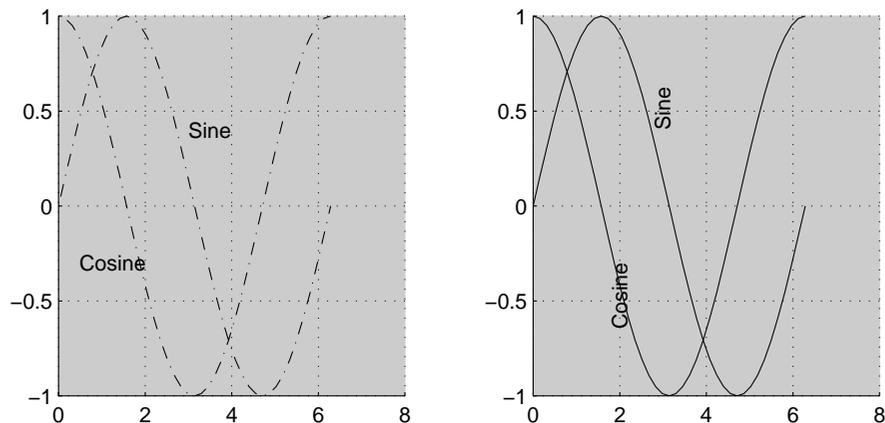
```

    'HorizontalAlignment','right')

axh2 = subplot(1,2,2); grid on
% Set default value for text Rotation property in second axes
set(axh2,'DefaultTextRotation',90)
line('XData',t,'YData',s)
line('XData',t,'YData',c)
text('Position',[3 .4],'String','Sine')
text('Position',[2 .3],'String','Cosine',...
    'HorizontalAlignment','right')

```

Issuing the same line and text statements to each subplot region results in a different display, reflecting different default settings.



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.

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. The "Protecting Figures and Axes" section in this chapter provides for more information on how object handles are hidden from normal access.

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

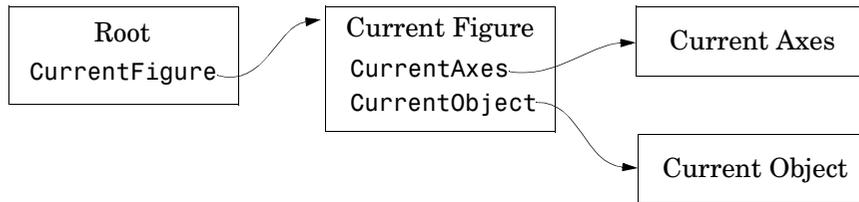
The figure `IntegerHandle` property controls which type of handle the figure receives.

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.

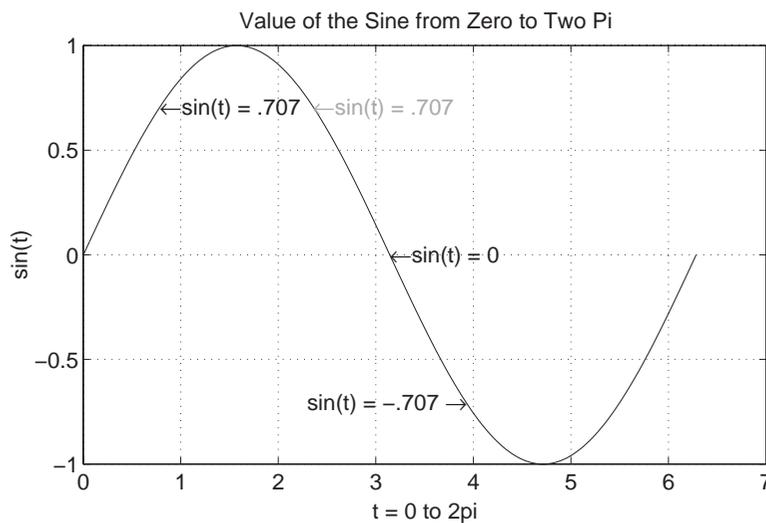
See “Controlling Graphics Output” on page 7-36 for information on how to prevent users from accessing the handles of graphics objects that you want to protect.

Searching for Objects by Property Values – `findobj`

The `findobj` function provides a means to traverse the object hierarchy quickly and obtain the handles of objects having specific property values. 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.

Example – Finding Objects

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 text `String` property.

```
text_handle = findobj('String','\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 in the hierarchy. In the previous example, you know the text object of interest is in the current axes so you can type

```
text_handle = findobj(gca,'String','\leftarrow sin(t) = .707');
```

Copying Objects

You can copy objects from one parent to another using the `copyobj` function. The new object differs from the original object only in the value of its `Parent` property and its handle; it is otherwise a clone of the original. You can copy a number of objects to a new parent, or one object to a number of new parents as long as the result maintains the correct parent/child relationship.

When you copy an object having children objects, MATLAB copies all children as well.

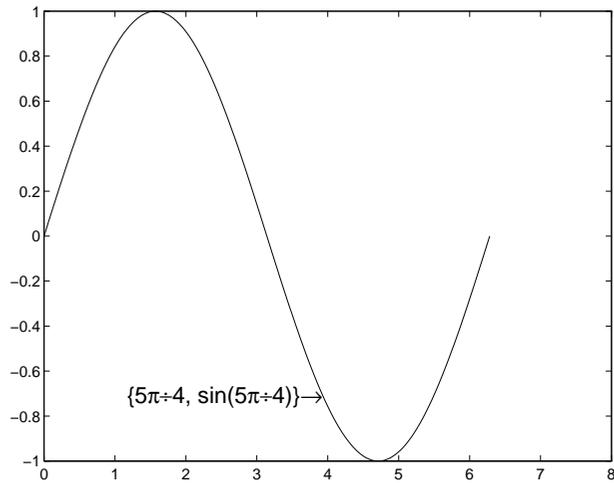
Example — Copying Objects

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\div4, sin(5\pi\div4)\}\rightarrow',...  
     'Position',[5*pi/4,sin(5*pi/4),0],...  
     'HorizontalAlignment','right')
```

In this statement, the `text` function:

- Labels the data point with the string $\{5\pi \div 4, \sin(5\pi \div 4)\}$ using TeX commands to draw a right-facing arrow and mathematical symbols.
- Specifies the Position in terms of the data being plotted.
- Places the data point to the right of the text string by changing the HorizontalAlignment to right (the default is left).

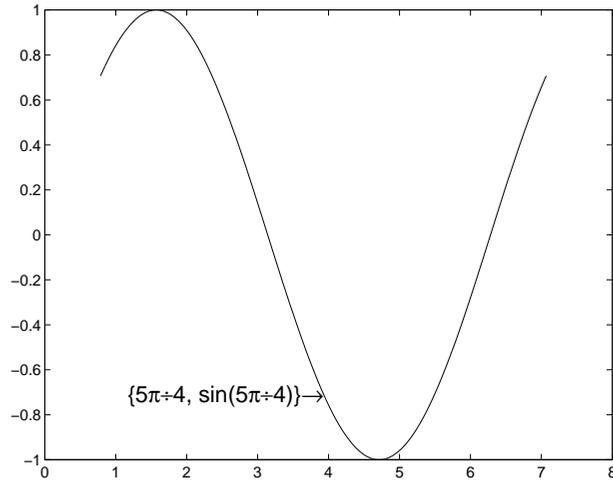


To label the same point with the same string in another plot, copy the text using `copyobj`. Since the last statement did not save the handle to the text object, you can find it using `findobj` and the 'String' property.

```
text_handle = findobj('String',...
    '\{5\pi\div4,\sin(5\pi\div4)\}\rightarrow');
```

After creating the next plot, add the label by copying it from the first plot.

```
copyobj(text_handle,gca).
```



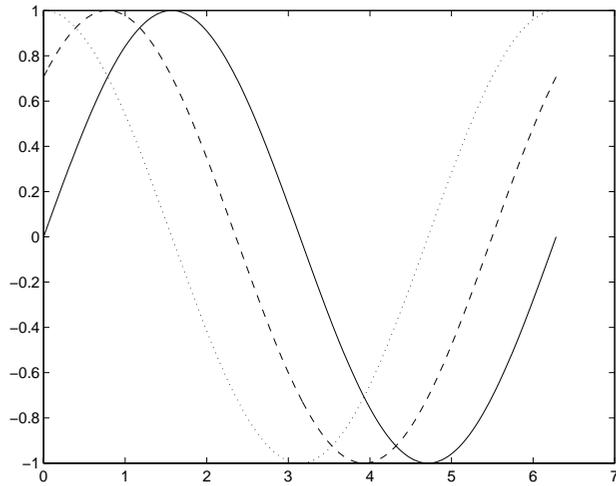
This particular example takes advantage of the fact that text objects define their location in the axes' data space. Therefore the text `Position` property did not need to change from one plot to another.

Deleting Objects

You can remove a graphics object with the `delete` command, using the object's handle as an argument. For example, you can delete the current axes (and all of its descendants) with the statement

```
delete(gca)
```

You can use `findobj` to get the handle of a particular object you want to delete. For example, to find the handle of the dotted line in this multiline plot,



use `findobj` to locate the object whose `LineStyle` property is `'.'`

```
line_handle = findobj('LineStyle',':');
```

then use this handle with the `delete` command.

```
delete(line_handle)
```

You can combine these two statements, substituting the `findobj` statement for the handle.

```
delete(findobj('LineStyle',':'))
```

Controlling Graphics Output

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 `uicontrol` and `uimenu` functions have a convenient syntax that enables you to specify the parent as the first argument,

```
uicontrol(Figure_handle,...)
uimenu(parent_menu_handle,...)
```

or you can set the `Parent` property.

Preparing Figures and Axes for Graphics

By default, commands that generate graphics output display the graphics objects in the current figure without clearing or resetting figure properties. However, if the graphics objects are axes children, MATLAB clears the axes and resets most axes properties to their default values before displaying the objects.

You can change this behavior by setting the figure and axes `NextPlot` properties.

Using `NextPlot` to Control Output Target

MATLAB high-level graphics functions check the 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 you develop a MATLAB-based application, controlling MATLAB drawing behavior is essential to creating a program that behaves predictably.

This table summarizes the possible values for the `NextPlot` property.

NextPlot	Figure	Axes
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 <code>clf</code> .	Remove all child objects, but do not reset axes properties. Equivalent to <code>cla</code> .
replace	Remove all child objects and reset figure properties to their defaults. Equivalent to <code>clf reset</code> .	Remove all child objects and reset axes properties to their defaults. Equivalent to <code>cla reset</code> . (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.

Targeting Graphics Output with newplot

MATLAB provides the `newplot` function to simplify the process of writing graphics M-files that conform to the settings of the NextPlot properties.

`newplot` checks the values of the NextPlot properties and takes the appropriate action based on these values. You should place `newplot` at the beginning of any M-file that calls object creation functions.

When your M-file calls `newplot`, the following possible actions occur:

- 1** `newplot` checks the current figure's NextPlot property:
 - If there are no figures in existence, `newplot` creates one and makes it the current figure.
 - If the value of NextPlot is `add`, `newplot` makes the figure the current figure.
 - If the value of NextPlot is `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 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 line styles instead of using different colors for multiline plots.

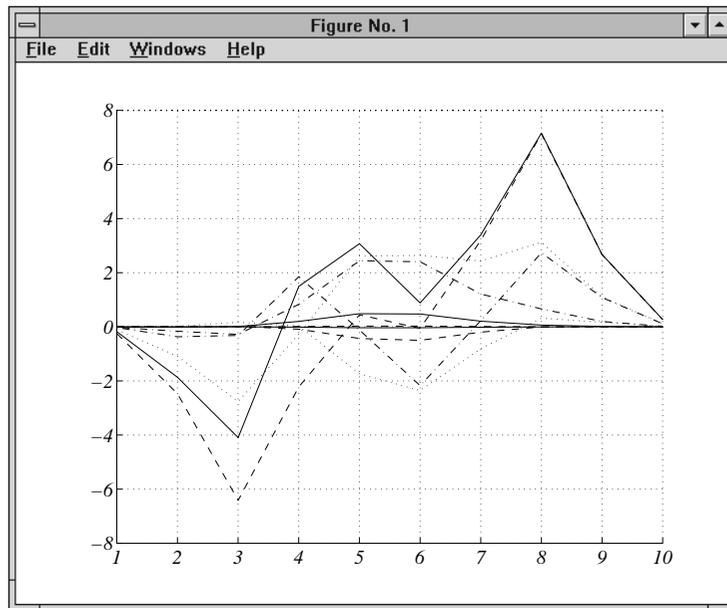
```
function my_plot(x,y)
cax = newplot; % newplot returns handle of current axes
LSO = ['- ' ;'--' ;':' ;'-.'];
set(cax,'FontName','Times','FontAngle','italic')
set(get(cax,'Parent'),'MenuBar','none') %
line_handles = line(x,y,'Color','b');
style = 1;
for i = 1:length(line_handles)
    if style > length(LSO), style = 1;end
    set(line_handles(i),'LineStyle',LSO(style,:))
    style = style + 1;
end
grid on
```

The function `my_plot` uses the high-level line function syntax to plot the data. This provides the same flexibility in input argument dimension that the built-in `plot` function supports. The `line` function does not check the value of the figure or axes `NextPlot` property. However, because `my_plot` calls `newplot`, it behaves the same way the high-level `plot` function does – with default values in place, `my_plot` clears and resets the axes each time you call it.

`my_plot` uses the handle returned by `newplot` to access the target figure and axes. This example sets axes font properties and disables the figure's menu bar. Note how the figure handle is obtained via the axes Parent property.

This picture shows typical output for the `my_plot` function.

```
my_plot(1:10,peaks(10))
```



Basic Plotting M-file Structure

This example illustrates the basic structure of graphics M-files:

- Call `newplot` early to conform to the `NextPlot` properties and to obtain the handle of the target axes.
- Reference the axes handle returned by `newplot` to set any axes properties or to obtain the figure's handle.
- Call object creation functions to draw graphics objects with the desired characteristics.

The MATLAB default settings for the `NextPlot` properties facilitate writing M-files that adhere to the standard behavior: reuse the figure window, but clear and reset the axes with each new graph. Other values for these properties allow you to implement different behaviors.

Replacing Only the 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 the MATLAB 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; % ishold tests the current hold state
LSO = ['- ' ; '- - ' ; ' : ' ; '- . '];
```

```
if nargin == 2
    hlines = line(x,y,'Color','k');
    if ~hold_state % Change view only if hold is off
        view(2)
    end
elseif nargin == 3
    hlines = line(x,y,z,'Color','k');
    if ~hold_state % Change view only if hold is off
        view(3)
    end
end
ls = 1;
for hindex = 1:length(hlines)
    if ls > length(LSO),ls = 1;end
    set(hlines(hindex),'LineStyle',LSO(ls,:))
    ls = ls + 1;
end
```

If hold is on when you call `my_plot3`, it does not change the view. If hold is off, `my_plot3` sets the view to 2-D or 3-D, depending on whether there are two or three input arguments.

Protecting Figures and Axes

There are situations in which it is important to prevent particular figures or axes from becoming the target for graphics output (i.e., preventing them from becoming the `gcf` or `gca`). An example is a figure containing the `uicontrols` that implement a user interface.

You can prevent MATLAB from drawing into a particular figure or axes by removing its handle from the list of handles that are visible to the `newplot` function, as well as any other functions that either return or implicitly reference handles (i.e., `gca`, `gcf`, `gco`, `cla`, `clf`, `close`, and `findobj`). Two properties control handle hiding: `HandleVisibility` and `ShowHiddenHandles`.

HandleVisibility Property

`HandleVisibility` is a property of all objects. It controls the scope of handle visibility within three different ranges. Property values can be:

- on — The object's handle is available to any function executed on the MATLAB command line or from an M-file. This is the default setting.

- **callback** — The object's handle is hidden from all functions executing on the command line, even if it is on the top of the screen stacking order. However, during callback routine execution (MATLAB statements or functions that execute in response to user action), the handle is visible to all functions, such as `gca`, `gcf`, `gco`, `findobj`, and `newplot`. This setting enables callback routines to take advantage of the MATLAB handle access functions, while ensuring that users typing at the command line do not inadvertently disturb a protected object.
- **off** — The object's handle is hidden from all functions executing on the command line and in callback routines. This setting is useful when you want to protect objects from possibly damaging user commands.

For example, if a GUI accepts user input in the form of text strings, which are then evaluated (using the `eval` function) from within the callback routine, a string such as `'close all'` could destroy the GUI. To protect against this situation, you can temporarily set `HandleVisibility` to `off` on key objects.

```
user_input = get(editbox_handle,'String');
set(gui_handles,'HandleVisibility','off')
eval(user_input)
set(gui_handles,'HandleVisibility','commandline')
```

Values Returned by `gca` and `gcf`. When a protected figure is topmost on the screen, but has 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. You can, therefore, use `gcf` to specify the figure handle from within the close request function.

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, the statements,

```
set(0, 'ShowHiddenHandles', 'yes')
delete(get(0, 'Children'))
```

unconditionally delete all figures.

Handle Validity Versus Handle Visibility

All handles remain valid regardless of whether they are visible or not. If you know an object's handle, you can set and get its properties. By default, figure handles are integers that are displayed at the top of the window.

You can provide further protection to figures by setting the `IntegerHandle` property to `off`. MATLAB then uses a floating-point number for figure handles.

Saving Handles in M-Files

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 because of user interaction.

Save Information First

It is a good practice to save relevant information about the MATLAB state in the beginning of your M-file. For example, you can begin an M-file with

```
cax = newplot;  
cfig = get(cax, 'Parent');  
hold_state = ishold;
```

rather than querying this information each time you need it. Remember that utility commands like `ishold` obtain the values they return whenever called. (The `ishold` command issues a number of `get` commands and string compares (`strcmp`) to determine the hold state.)

If you are temporarily going to alter the hold state within the M-file, you should save the current values of the `NextPlot` properties so you can reset them later.

```
ax_nextplot = lower(get(cax, 'NextPlot'));  
fig_nextplot = lower(get(cfig, 'NextPlot'));  
.  
.  
.  
set(cax, 'NextPlot', ax_nextplot)  
set(cfig, 'NextPlot', fig_nextplot)
```

Properties Changed by Built-In Functions

To achieve their intended effect, many built-in functions change axes properties, which can then affect the workings of your M-file. This table lists the MATLAB built-in graphics functions and the properties they change. Note that these properties change only if `hold` is off.

Function	Axes Property: Set To
<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

Function	Axes Property: Set To
image (high-level)	Box: on Layer: top CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XDir: normal XLim: [0 size(CData,1)]+0.5 XLimMode: manual YDir: reverse YLim: [0 size(CData,2)]+0.5 YLimMode: manual
loglog	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: log YScale: log
plot	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view

Function	Axes Property: Set To
plot3	CameraPosition: 3-D view CameraTarget: 3-D view CameraUpVector: 3-D view CameraViewAngle: 3-D view XScale: linear YScale: linear ZScale: linear
semilogx	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: log YScale: linear
semilogy	Box: on CameraPosition: 2-D view CameraTarget: 2-D view CameraUpVector: 2-D view CameraViewAngle: 2-D view XScale: linear YScale: log

Callback Properties for Graphics Objects

A *callback* is a function that executes when a specific event occurs on a graphics object. You specify a callback by setting the appropriate property of the object. This section describes the events (specified via properties) for which you can define callbacks. See “Function Handle Callbacks” on page 7-53 for information on how to define callbacks.

Graphics Object Callbacks

All graphics objects have three properties for which you can define callback routines:

- `ButtonDownFcn` — Executes when users click the left mouse button while the cursor is over the object or within a 5-pixel border around the object.
- `CreateFcn` — Executes during object creation after all properties are set.
- `DeleteFcn` — Executes just before deleting the object.

Uicontrol, Uimenu, and Uicontextmenu Callbacks

`Uicontrol`, `uimenu`, and `uicontextmenu` objects have a `Callback` property through which you define the function to execute when users activate these devices (e.g., click on a push button or select a menu).

Figures Callbacks

Figures have additional properties that execute callback routines with the appropriate user action. Only the `CloseRequestFcn` has a callback defined by default:

- `CloseRequestFcn` — Executes when a request is made to close the figure (by a `close` command, by the window manager menu or by quitting MATLAB)
- `KeyPressFcn` — Executes when users press a key while the cursor is within the figure window
- `ResizeFcn` — Executes when users resize the figure window
- `WindowButtonDownFcn` — Executes when users click a mouse button while the cursor is over the figure background, a disabled `uicontrol`, or the axes background

- `WindowButtonMotionFcn` — Executes when users move the mouse button within the figure window (but not over menus or title bar).
- `WindowButtonUpFcn` — Executes when users release the mouse button, after having pressed the mouse button within the figure.

Function Handle Callbacks

Handle Graphics objects have a number of properties for which you can define callback functions. When a specific event occurs (e.g., a user clicks on a push button or deletes a figure), the corresponding callback function executes. You can specify the value of a callback property as a

- String that is a MATLAB command or the name of an M-file
- Cell array of strings
- Function handle or a cell array containing a function handle and additional arguments

This section illustrates how to define function handle callbacks for Handle Graphics objects. For information on function handles and how to use them, see the function handle reference page.

Function Handle Syntax

In Handle Graphics, functions that you want to use as function handle callbacks must define at least two input arguments in the function definition:

- The handle of the object generating the callback
- The event data structure (currently empty)

MATLAB passes these two arguments implicitly whenever the callback executes. For example, consider the following statements that are contained in a single M-file.

```
function myGui
% Create a figure and specify a callback
figure('WindowButtonDownFcn',@myCallback)
.
.
.
% Callback subfunction header defines two input arguments
function myCallback(obj,eventdata)
```

The first statement creates a figure and assigns a function handle to its `WindowButtonDownFcn` property (created by using the `@` symbol before the function name). This function handle points to the subfunction `myCallback`.

The definition of `myCallback` must specify the two required input arguments in its function definition line.

Passing Additional Input Arguments

You can define the callback function to accept additional input arguments by adding them to the function definition. For example,

```
function myCallback(obj,eventdata,arg1,arg2)
```

When using additional arguments for the callback function, you must set the value of the property to a cell array (i.e., enclose the function handle and arguments in curly braces). For example,

```
figure('WindowButtonDownFcn',{@myCallback,arg1,arg2})
```

Defining Callbacks as a Cell Array of Strings — Special Case

Defining a callback as a cell array of strings is a special case because MATLAB treats it differently from a simple string. Setting a callback property to a string causes MATLAB to evaluate that string in the base workspace when the callback is invoked. However, setting a callback to a cell array of strings behaves as follows:

- The cell array must contain the name of an M-file that is on the MATLAB path as the first string element.
- The M-file callback must define at least two arguments (the handle of the callback object and an empty matrix).
- Any additional strings in the cell array are passed to the M-file callback as arguments.

For example,

```
figure('WindowButtonDownFcn',{myCallback,arg1})
```

requires you to define a function M-file that uses three arguments,

```
function myCallback(obj,eventdata,arg1)
```

Why Use Function Handle Callbacks

Using function handles to specify callbacks provides some advantages over the use of strings, which must be either MATLAB commands or the name of an M-file that will be on the MATLAB path at run-time.

Single File for All Code

Function handles enable you to use a single M-file for all callbacks. This is particularly useful when creating graphical user interfaces since you can include both the layout commands and callbacks in one file.

For information on how to access subfunctions, see the “Evaluating a Function Through Its Handle” section of Programming and Data Types in the MATLAB documentation.

Keeping Variables in Scope

When MATLAB evaluates function handles, the same variables are in scope as when the function handle was created. (In contrast, callbacks specified as strings are evaluated in the base workspace.) This simplifies the process of managing global data, such as object handles in a GUI.

For example, suppose you create a GUI with a list box that displays workspace variables and a push button whose callback creates a plot using the variables selected in the list box. The push button callback needs the handle of the list box to query the names of the selected variables. Here’s what to do.

Create the list box and save the handle:

```
h_listbox = uicontrol('Style','listbox',... etc.);
```

Pass the list box handle to the push button’s callback, which is defined in the same M-file:

```
h_plot_button = uicontrol('Style','pushbutton',...  
    'Callback',{@plot_button_callback,h_listbox},...,etc.);
```

The handle of the list box is now available in the plot button’s callback without relying on global variables or using `findobj` to search for the handle. See “Example — Using Function Handles in a GUI” on page 7-56 for an example that uses this technique.

Callback Object Handle and Event Data

MATLAB passes additional information to the callback when executed. Currently this information includes the handle of the callback object, however, later MATLAB releases will include additional event data.

Function Handles Stay in Scope

A function handle can point to a function that is not in scope at the time of execution. For example, the function may be a subfunction in another M-file.

For a general discussion of the advantages function handles provide, see the “Benefits of Function Handles” section of Programming and Data Types in the MATLAB documentation.

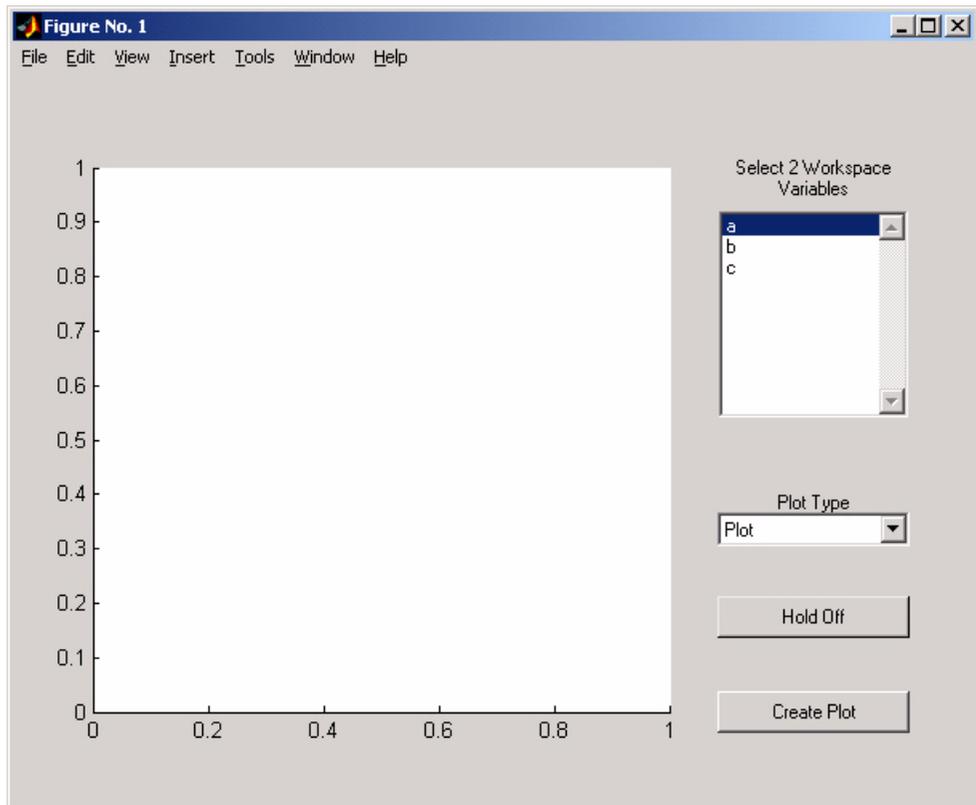
Example – Using Function Handles in a GUI

This example shows how to create a simple GUI that plots variables that exist in the base workspace. It is defined in a single M-file that contains both the layout commands and the callbacks. This example uses function handles to specify callback functions. See “Function Handle Callbacks” on page 7-53 for more information on the use of function handle callbacks.

Note The following link executes MATLAB commands and is designed to work within the MATLAB Help browser.

Click this link to display the example code in the MATLAB editor.

Here is what the GUI looks like.



The GUI Layout

The first step is to define each component in the GUI and save the handles.

```
function plot_vars
% Define the GUI layout
h_figure = figure('Units','characters',...
    'Position',[72 38 120 35],...
    'Color',get(0,'DefaultUicontrolBackgroundColor'),...
    'HandleVisibility','callback');
h_axes = axes('Units','characters',...
    'Position',[10 4.5 72 26],...
    'Parent',h_figure);
h_listbox_label = uicontrol(h_figure,...
```

```
        'Style','text',...
        'Units','characters',...
        'Position',[88 29 24 2],...
        'String','Select 2 Workspace Variables');
h_listbox = uicontrol(h_figure,...
    'Style','listbox',...
    'Units','characters',...
    'Position',[88 18.5 24 10],...
    'BackgroundColor','white',...
    'Max',10,'Min',1,...
    'Callback',@listbox_callback);
h_popup_label = uicontrol(h_figure,..
    'Style','text',...
    'Units','characters',...
    'Position',[88 13 24 2],...
    'String','Plot Type');
h_popup = uicontrol(h_figure,...
    'Style','popupmenu',...
    'Units','characters',...
    'Position',[88 12 24 2],...
    'BackgroundColor','white',...
    'String',{'Plot','Bar','Stem'});
h_hold_toggle = uicontrol(h_figure,..
    'Style','toggle',...
    'Units','characters',...
    'Position',[88 8 24 2],...
    'String','Hold State',...
    'Callback',{@hold_toggle_callback,h_axes});
h_plot_button = uicontrol(h_figure,...
    'Style','pushbutton',...
    'Units','characters',...
    'Position',[88 3.5 24 2],...
    'String','Create Plot',...
    'Callback',{@plot_button_callback,h_listbox,h_popup,h_axes});
```

Initialize the GUI

The list box and the hold toggle button need to be initialized before the GUI is ready to use, which is accomplished by executing their callbacks. Note that you

must specify all the arguments when calling these functions since we are not evaluating function handles here.

```
% Initialize list box and make sure
% the hold toggle is set correctly
listbox_callback(h_listbox,[])
hold_toggle_callback(h_hold_toggle,[],h_axes)
```

The Callback Functions

Only the list box, toggle button, and plot push button have callbacks.

List Box Callback. The list box callback takes advantage of the callback object handle (first argument) generated by MATLAB to set the String property to the current list of workspace variables.

Note that for simplicity, the contents of the list box is updated every time the user selects an item. A separate update button would be a more robust approach.

```
% Callback for list box
function listbox_callback(obj,eventdata)
% Load workspace vars into listbox
vars = evalin('base','who');
set(obj,'String',vars)
```

Toggle Button Callback. The toggle button callback requires two additional arguments — the handles of the GUI figure and axes. We can use the handles saved when we created the figure and axes (`h_figure` and `h_axes`) because function handle callbacks will execute within the context of this M-file.

We want the GUI to call the `hold` command, but `hold` operates only on the current figure. Our GUI figure cannot become the current figure because we've hidden its handle. To implement the functionality of `hold`, this callback sets the axes `NextPlot` property directly.

```
% Callback for hold state toggle button
function hold_toggle_callback(obj,eventdata,h_axes)
button_state = get(obj,'Value');
if button_state == get(obj,'Max')
% toggle button is pressed
set(h_axes,'NextPlot','add')
set(obj,'String','Hold On')
```

```
elseif button_state == get(obj,'Min')
    % toggle button is not pressed
    set(h_axes,'NextPlot','replace')
    set(obj,'String','Hold Off')
end
```

Plot Button Callback. The plot button callback performs three tasks:

- Gets the names of the variables selected by the user in the list box
- Gets the type of plot selected by the user in the popup menu
- Constructs and evaluates the plotting command in the MATLAB base workspace

```
% Callback for plot button
function
plot_button_callback(obj,eventdata,h_listbox,h_popup,h_axes)
% Get workspace variables
vars = get(h_listbox,'String');
var_index = get(h_listbox,'Value');
if length(var_index) ~= 2
    errordlg('You must select two variables',...
            'Incorrect Selection','modal')
    return
end
% Get data from base workspace
x = evalin('base',vars{var_index(1)});
y = evalin('base',vars{var_index(2)});
% Get plotting command
selected_cmd = get(h_popup,'Value');
% Make the GUI axes current
axes(h_axes)
% Call appropriate command based on what user selected
switch selected_cmd
case 1 % user selected plot
    plot(x,y)
case 2 % user selected bar
    bar(x,y)
case 3 % user selected stem
    stem(x,y)
end
```

Using the GUI

Select some variables from the workspace and overlay stem and bar plots.

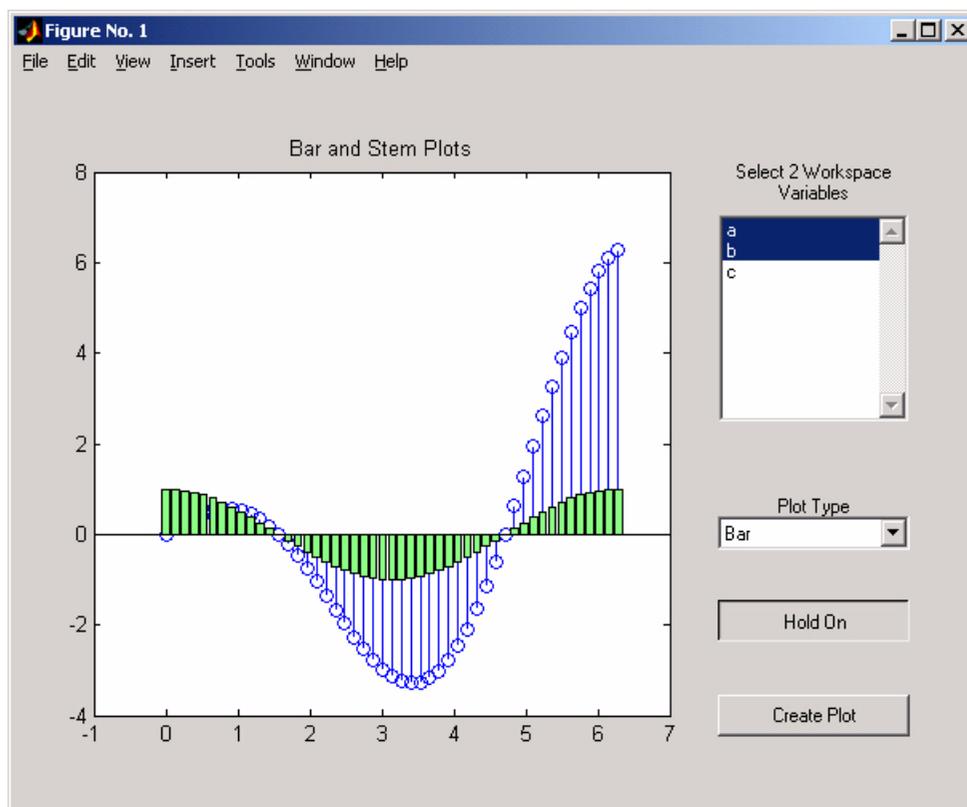


Figure Properties

Figure Objects (p. 8-2)	What is a figure and what are its properties.
Positioning Figures (p. 8-3)	Properties used to position figures and how they are measured.
Controlling How MATLAB Uses Color (p. 8-7)	Properties that control how MATLAB uses system color resources.
Selecting Drawing Methods (p. 8-15)	How to select rendering methods and when to use double buffering and backing store.
Specifying the Figure Pointer (p. 8-18)	How to select from predefined pointers or define customer pointers.
Interactive Graphics (p. 8-23)	Properties that define figure callbacks and contain figure state.

Figure Objects

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 section discusses some of the features that are implemented through figure properties and provides examples of how to use these features. The table in the figure reference page listing all properties provides an overview of the characteristics affected by figure properties.

Positioning Figures

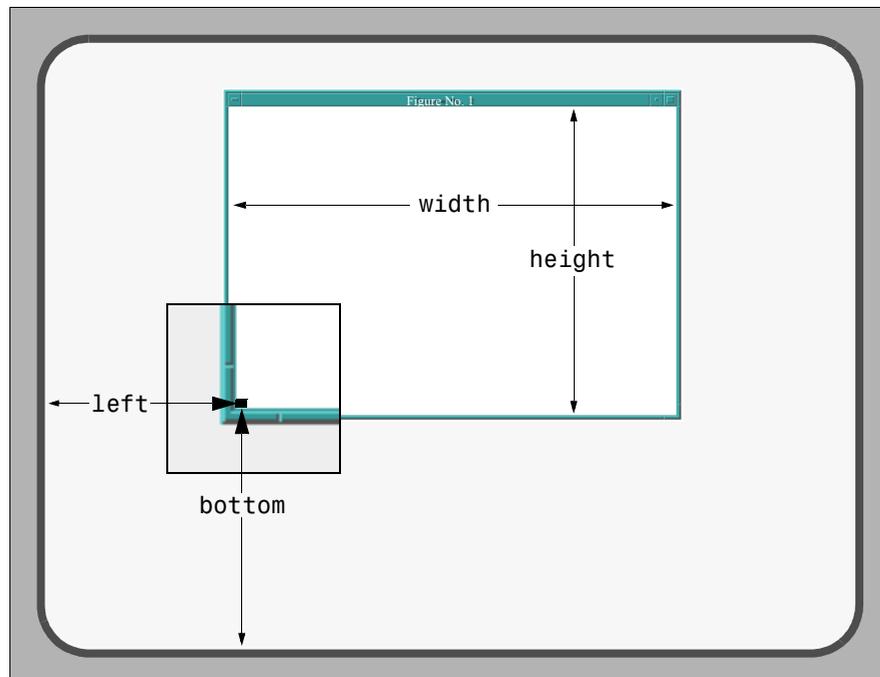
The figure `Position` property controls the size and location of the figure window on the root screen. At startup, MATLAB determines the size of your computer screen and defines a default value for `Position`. This default creates figures about one-quarter of the screen's size and places them centered 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} |
 characters]
```

with `pixels` being the default. These choices allow you to specify the figure size and location in absolute units (such as inches) if you want the window to always be a certain size, or in units relative to the screen size (such as pixels). `characters` are units that enable you to define the location and size of the figure in units that are based on the size of the default system font.

Determining Screen Size

Whatever units you use, it is important to know the extent of the screen in those units. You can obtain this information from the root `ScreenSize` property. For example,

```
get(0, 'ScreenSize')
ans =
    1    1 1152  900
```

In this case, the screen is 1152 by 900 pixels. MATLAB returns the `ScreenSize` in the units determined by the root `Units` property. For example,

```
set(0, 'Units', 'normalized')
```

normalizes the values returned by `ScreenSize`.

```
get(0, 'ScreenSize')
ans =
    0    0    1    1
```

Defining the figure `Position` in terms of the `ScreenSize` in normalized units makes the specification independent of variations in screen size. This is useful if you are writing an M-file that is to be used on different computer systems.

Example – Specifying Figure Position

Suppose you want to define two figure windows that occupy the upper third of the computer screen (e.g., one for uicontrols and the other to display data). To position the windows precisely, you must consider the window borders when calculating the size and offsets to specify for the `Position` properties.

The figure `Position` property does not include the window borders, so this example uses a width of 5 pixels on the sides and bottom and 30 pixels on the top.

```
bdwidth = 5;  
topbdwidth = 30;
```

Ensure root units are pixels and get the size of the screen

```
set(0, 'Units', 'pixels')  
scnsize = get(0, 'ScreenSize');
```

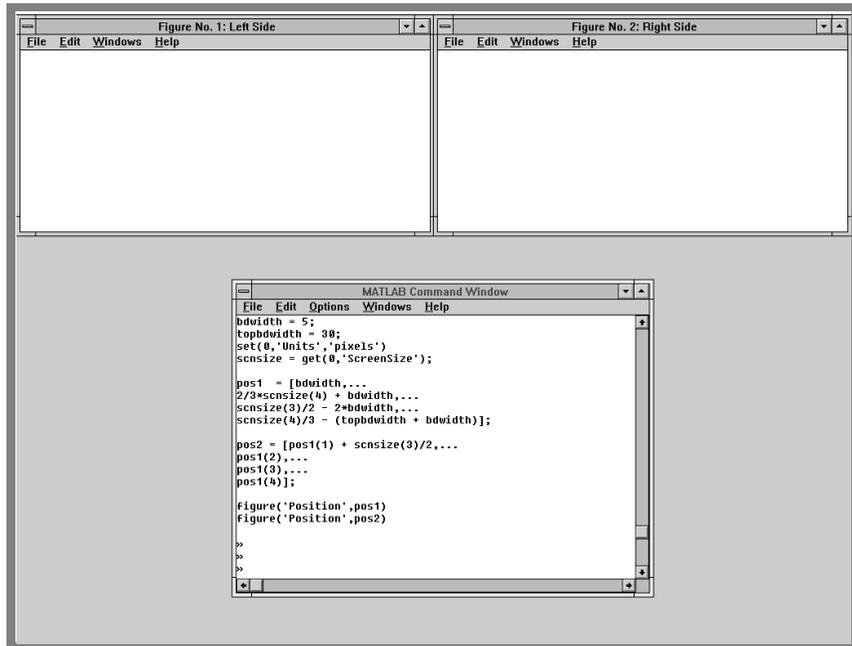
Define the size and location of the figures

```
pos1 = [bdwidth, ...  
        2/3*scnsize(4) + bdwidth, ...  
        scnsize(3)/2 - 2*bdwidth, ...  
        scnsize(4)/3 - (topbdwidth + bdwidth)];  
pos2 = [pos1(1) + scnsize(3)/2, ...  
        pos1(2), ...  
        pos1(3), ...  
        pos1(4)];
```

Create the figures

```
figure('Position', pos1)  
figure('Position', pos2)
```

The two figures now occupy the top third of the screen.



Controlling How MATLAB Uses 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 those listed in the following table.

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	The property that 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	The property that determines whether MATLAB uses the current dither colormap or creates one based on the colors specified for existing graphics objects.

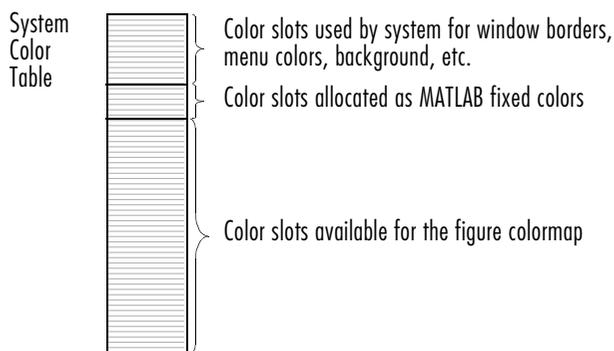
Indexed Color Displays

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, and so on.

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.

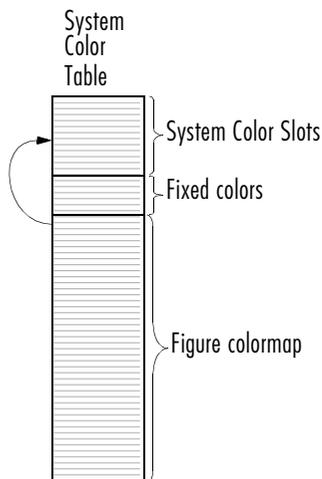
More Details

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).

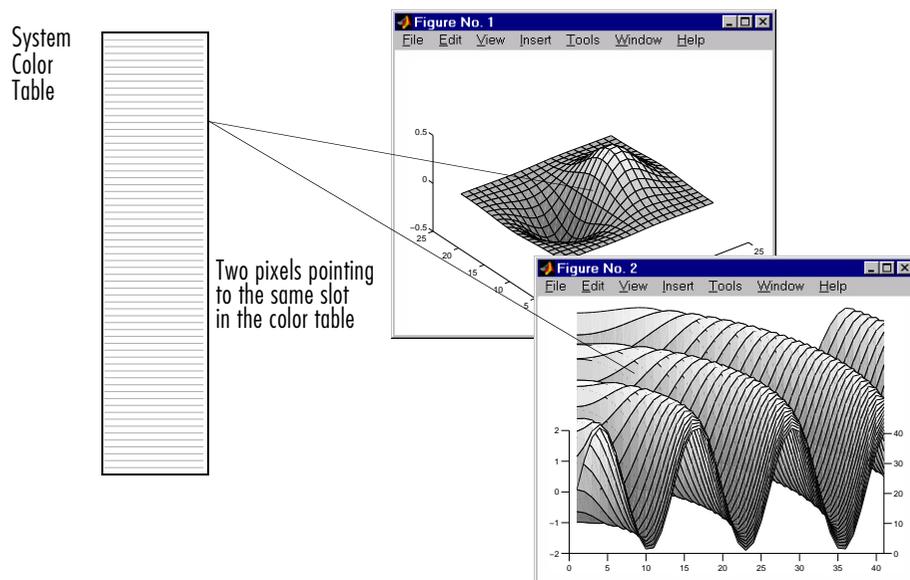
Nonactive Figures and Shared Colors

Overview

Set `ShareColors` to on to conserve resources and to off to allow rapid colormap change.

More Detail

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 `spinmap` 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 re-rendering (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 `spinmap` M-file for an example of this technique.

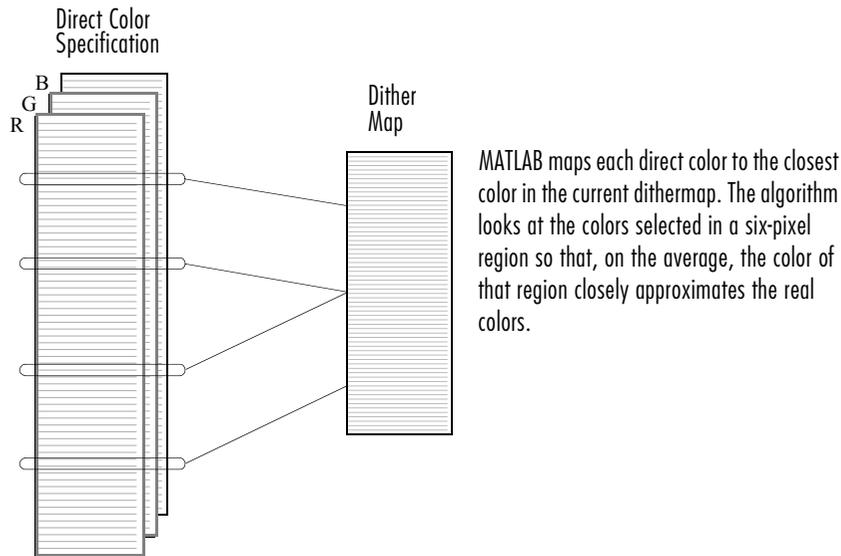
Dithering Truecolor on Indexed Color Systems

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.

More Detail

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. Indexed 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.

Selecting Drawing Methods

MATLAB enables you to select different techniques for drawing graphics. The combination of settings you select depends on the type of graphics you are producing.

There are four figure properties that affect how MATLAB draws graphics:

- `BackingStore` – allows faster redrawing when obscured figure windows are exposed.
- `DoubleBuffer` – produces flash-free rendering for simple animations.
- `Renderer` and `RendererMode` – specifies different rendering methods or allows MATLAB to make the selection.

Backing Store

Overview

Set `BackingStore` to `on` to produce fast redraws of previously obscured windows. Disable `BackingStore` to use less system memory.

More Details

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.

Double Buffering

Overview

Set `DoubleBuffer` to `on` when animating lines rendered in painters with `EraseMode` set to `normal`.

More Details

Double buffering is the process of drawing into an off-screen pixel buffer and then blitting the buffer contents to the screen once the drawing is complete (instead of drawing directly to the screen where the process of drawing is visible as it progresses). Double buffering generally produces flash-free rendering for simple animations (such as those involving lines, as opposed to objects containing large numbers of polygons).

The figure `DoubleBuffer` property accepts the values `on` and `off`, with `off` being the default. You can select double buffering only when the figure `Renderer` property is set to `painters`. `Zbuffer` always uses double buffering and ignores this property. `OpenGL` does not use double buffering.

Use double buffering with the animated object's `EraseMode` property set to `normal`.

Selecting a Renderer

Overview

MATLAB automatically selects the best renderer based on the complexity of the graphics objects and the options available on your system.

More Details

A renderer is the software that processes graphics data (such as vertex coordinates) into a form that MATLAB can use to draw into the figure. MATLAB supports three renderers:

- `Painters`
- `Zbuffer`
- `OpenGL`

Painters

`Painters` method is faster when the figure contains only simple or small graphics. It cannot be used with lighting.

Z-Buffer

Z-buffering is the process of determining how to render each pixel by drawing only the front-most object, as opposed to drawing all objects back to front,

redrawing objects that obscure those behind. The pixel data is buffered and then blitted to the screen all at once.

Z-buffering is generally faster for more complex graphics, but 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 from Z-Buffer. You can select the resolution of the PostScript file produced by the `print` command using the `-r` option. By default, MATLAB prints Z-buffered figures at a medium resolution of 150 dpi (the default with `Renderer` set to `painters` is 864 dpi).

The size of the file generated from a Z-buffer figure does not depend on its contents, just the size of the figure. To decrease the file size, make the `PaperPosition` property smaller before printing (or set `PaperPositionMode` to `auto` and resize the figure window).

OpenGL

OpenGL is available on many computer systems. It is generally faster than either `painters` or `zbuffer` and in some cases enables MATLAB to use the system's graphics hardware (which results in significant speed increase). See the figure `Renderer` property for more information.

Limitations of OpenGL. OpenGL has two limitations when compared to `painters` and `zbuffer`:

- OpenGL does not interpolate colors within the figure colormap; all color interpolation is performed through the RGB color cube, which may produce unexpected results.
- OpenGL does not support Phong lighting.

Specifying the Figure Pointer

MATLAB indicates the position of the pointer (cursor) within the figure window using a graphical symbol. You can select a pointer from 15 predefined symbols (see table below) or you can define your own symbol. By convention, each of the predefined symbols has a purpose associated with it (although MATLAB enforces no rules for the use of any symbols).

You specify the pointer symbol by setting the value of the figure `Pointer` property. For example, this statement sets the pointer in the current figure (`gcf`) to an arrow.

```
set(gcf, 'Pointer', 'arrow')
```

The following table shows the predefined symbols, the associated specifier, and describes typical use.

Purpose	Specifier	Typical Symbol
Locate a point on a graphics object	crosshair	+
Select a point anywhere in the figure	arrow	
Indicate the system is busy	watch	
Resize an object from the top-left corner	topl	
Resize an object from the top-right corner	topr	
Resize an object from the bottom-left corner	botl	
Resize an object from the bottom-right corner	botr	
View the actual hot spot	circle	
Locate a point	cross	

Purpose	Specifier	Typical Symbol
Use as popular symbol	fleur	
Resize an object from the left side	left	
Resize an object from the right side	right	
Resize an object from the top	top	
Resize an object from the bottom	bottom	
Align a point with other objects on the display	fullcross	
See the next section for information on defining your own pointer shape	custom	

Defining Custom Pointers

When you set the Pointer property to custom, MATLAB displays the pointer you define using the `PointerShapeCData` and the `PointerShapeHotSpot` properties. Custom pointers are 16-by-16 pixels, where each pixel can be either black, white, or transparent.

Specify the pointer by creating a 16-by-16 matrix containing elements that are:

- 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 `PointerShapeCData` property. MATLAB displays the defined pointer whenever the pointer is in the figure window.

The `PointerShapeHotSpot` property specifies the pixel that indicates the pointer location. MATLAB then stores this location in the root `PointerLocation` property. Set the `PointerShapeHotSpot` property to a two-element vector specifying the row and column indices in the `PointerShapeCData` matrix that 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.

Example – Two Custom Pointers

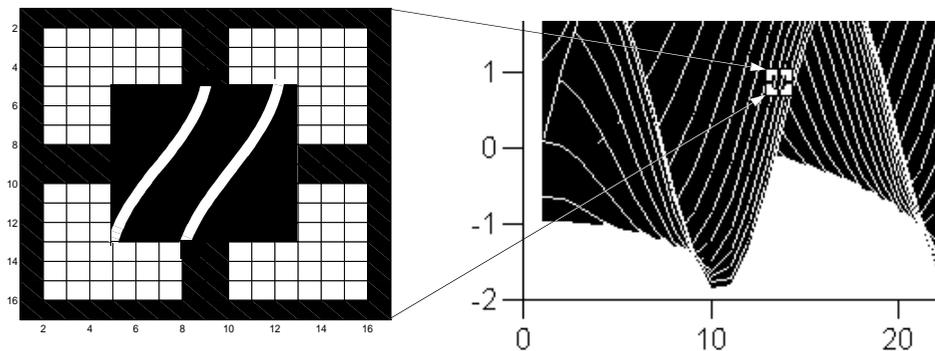
One way to create a custom pointer is to assign values to a 16-by-16 matrix by hand, as illustrated in the following example.

First, initialize the matrix, setting all values to 2. Create a black border 1 pixel wide. Add alignment marks.

```
P = ones(16)+1;
P(1,:) = 1; P(16,:) = 1;
P(:,1) = 1; P(:,16) = 1;
P(1:4,8:9) = 1; P(13:16,8:9) = 1;
P(8:9,1:4) = 1; P(8:9,13:16) = 1;
P(5:12,5:12) = NaN; % Create a transparent region in the center
set(gcf, 'Pointer', 'custom', 'PointerShapeCData', P, ...
    'PointerShapeHotSpot', [9 9])
```

The last statement sets the Pointer property to custom, assigns the matrix to the PointerShapeCData property, and selects the “hot spot” as element (9,9).

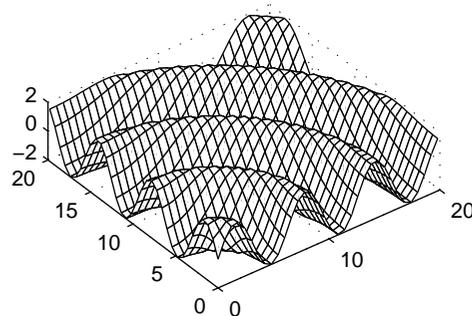
MATLAB now uses the custom pointer within the figure window.



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);
```

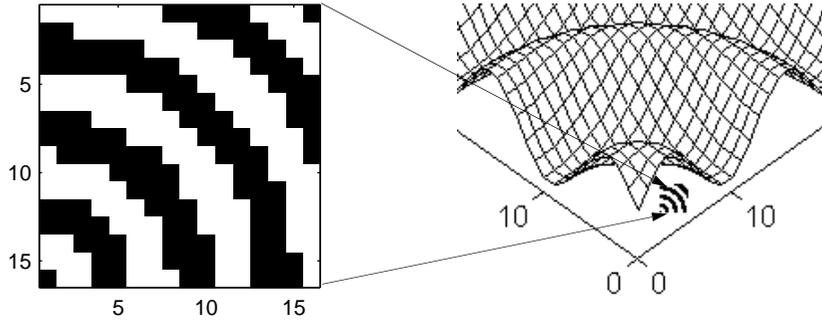
produces an interesting surface.



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.



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
- CreateFcn
- DeleteFcn
- KeyPressFcn
- Interruptible
- ResizeFcn
- WindowButtonDownFcn, WindowButtonMotionFcn, and WindowButtonUpFcn

Properties that contain information about the MATLAB state:

- CurrentAxes
- CurrentCharacter
- CurrentMenu
- CurrentObject
- CurrentPoint
- SelectionType

The online manual, *Creating Graphical User Interfaces*, provides information on creating programs that incorporate interactive graphics.

Axes Properties

Axes Objects (p. 9-2)	What is an axes and what are its properties.
Labeling and Appearance Properties (p. 9-3)	Properties that affect general appearance of the axes.
Positioning Axes (p. 9-5)	How axes are positioned within a figure.
Multiple Axes per Figure (p. 9-7)	How to use axes to place text outside the graph axes and how to use multiple axes within a figure to achieve different views.
Individual Axis Control (p. 9-10)	Properties that control the x -, y -, and z -axis individually.
Using Multiple X and Y Axes (p. 9-16)	Multiple axes on a single graph.
Automatic-Mode Properties (p. 9-19)	Properties that are set automatically with each graph.
Colors Controlled by Axes (p. 9-22)	Axes colors and color limits (<code>caxis</code>) to control the mapping of data to colormaps.

Axes Objects

Axes are the parents of image, light, line, patch, rectangle, 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 section discusses some of the features that are implemented through axes properties and provides examples of how to use these features.

The table in the axes reference page listing all axes properties provides an overview of the characteristics affected by these properties.

Axis Labels

The individual axis labels are text objects whose handles are normally hidden from the command line (their `HandleVisibility` property is set to `callback`). You can use the `xlabel`, `ylabel`, `zlabel`, and `title` functions to create axis labels. However, these functions affect only the current axes. If you are labeling axes other than the current axes by referencing the axes handle, then you must obtain the text object handle from the corresponding axes property. For example,

```
get(axes_handle, 'XLabel')
```

returns the handle of the text object used as the x -axis label. Obtaining the text handle from the axes is useful in M-files and MATLAB-based applications where you cannot be sure the intended target is the current axes.

The following statements define the x - and y -axis labels and title for the axes above.

```
set(get(axes_handle, 'XLabel'), 'String', 'Values of X')
set(get(axes_handle, 'YLabel'), 'String', 'Values of Y')
set(get(axes_handle, 'Title'), 'String', '\fontname{times}\itZ =
f(x,y)')
```

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)
```

Positioning Axes

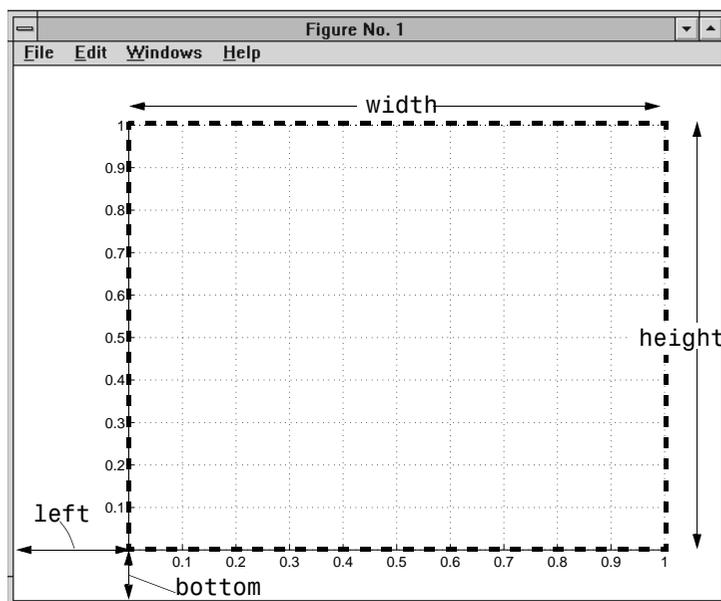
The axes `Position` property controls the size and location of an axes within a figure. The default axes has the same aspect ratio (ratio of width to height) as the default figure and fills most of the figure, leaving a border around the edges. However, you can define the axes position as any rectangle and place it wherever you want within a figure.

The Position Vector

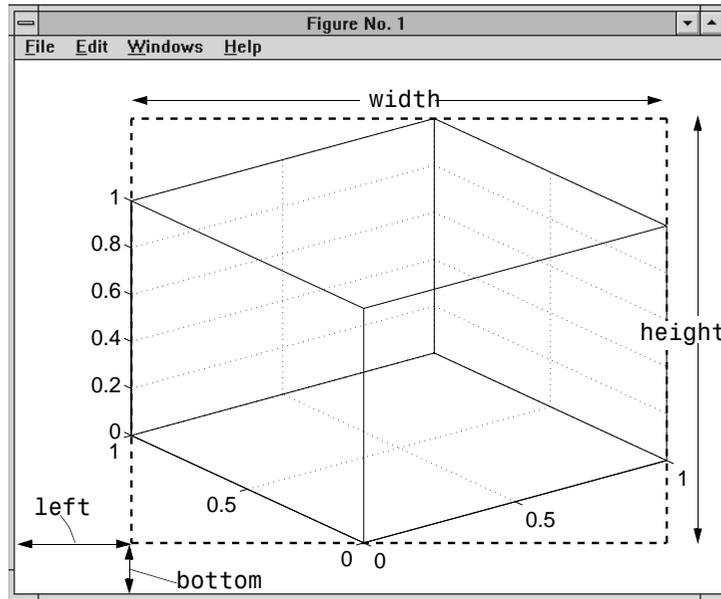
MATLAB defines the axes `Position` property as a vector.

```
[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.

Position Units

The axes Units property determines the units of measurement for the Position property. Possible values for this property are

```
set(gca, 'Units')
[ inches | centimeters | {normalized} | points | pixels ]
```

with normalized being the default. Normalized units map the lower-left corner of the figure to the point (0,0) and the upper-right corner to (1.0,1.0), regardless of the size of the figure. Normalized units cause axes to resize automatically whenever you resize the figure. All other units are absolute measurements that remained fixed as you resize the figure.

Multiple Axes per Figure

The `subplot` function creates multiple axes in one figure by computing values for `Position` that produce the specified number of axes.

The `subplot` function is useful for laying out a number of graphs equally spaced in the figure. However, overlapping axes can create some other useful effects. The following two sections provides examples:

- “Placing Text Outside the Axes” on page 9-7
- “Multiple Axes for Different Scaling” on page 9-8

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.

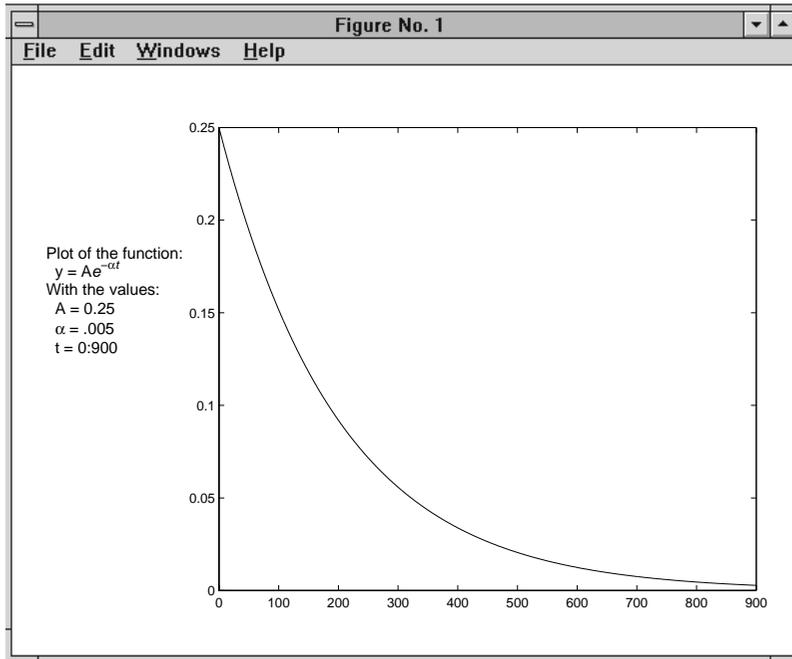
Now plot some data in the current axes. The last axes created is the current axes so MATLAB directs graphics output there.

```
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\ite}^{\-alpha{\itt}}'};
str(3) = {'With the values:'};
str(3) = {' A = 0.25'};
str(4) = {' \alpha = .005'};
str(5) = {' t = 0:900'};
set(gcf, 'CurrentAxes', h)
```

```
text(.025,.6,str,'FontSize',12)
```

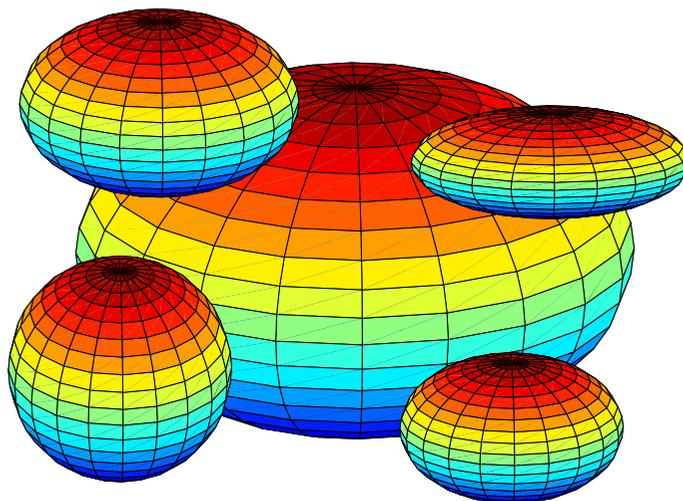


Multiple Axes for Different Scaling

You can create multiple axes to display graphics objects with different scaling without changing the data that defines these objects (which would be required to display them in a single axes).

```
h(1) = axes('Position',[0 0 1 1]);  
sphere  
h(2) = axes('Position',[0 0 .4 .6]);  
sphere  
h(3) = axes('Position',[0 .5 .5 .5]);  
sphere  
h(4) = axes('Position',[.5 0 .4 .4]);  
sphere  
h(5) = axes('Position',[.5 .5 .5 .3]);  
sphere
```

```
set(h, 'Visible', 'off')
```



Each sphere is defined by the same data. However, 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`.

This section discusses the following properties.

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.

Setting Axis Limits

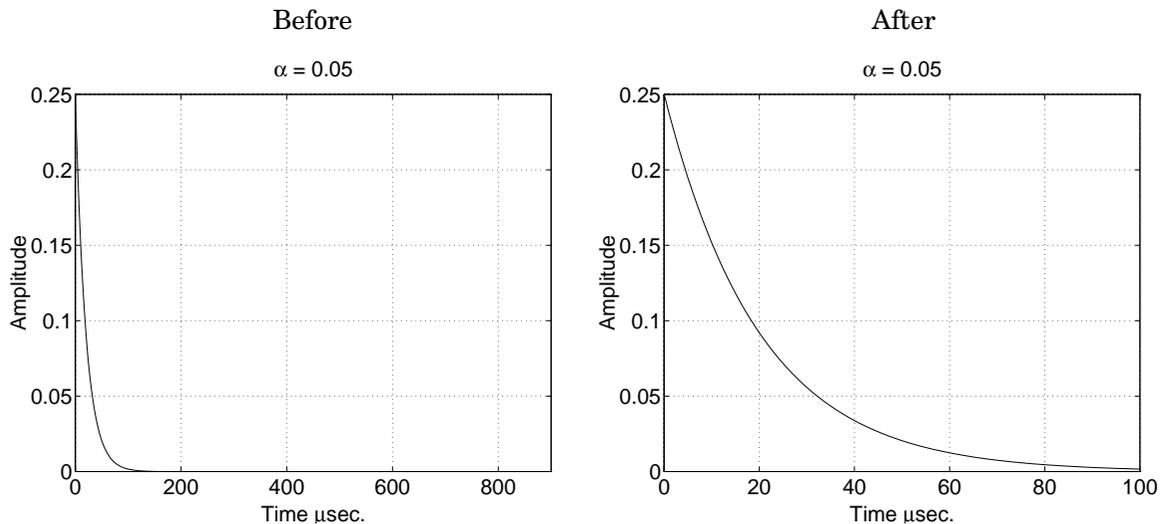
MATLAB determines the limits automatically for each axis based on the range of the data. You can override the selected limits by specifying the `XLim`, `YLim`, or `ZLim` property. For example, consider a plot of the function $Ae^{-\alpha t}$ evaluated with $A = 0.25$, $\alpha = 0.05$, and $t = 0$ to 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. If the handle of the axes is `axes_handle`, then following statement,

```
set(axes_handle,'XLim',[0 100])
```

creates the plot on the right.



You can use the `axis` command to set limits on the current axes only.

Semiautomatic Limits

You can specify either the minimum or maximum value for an axis limit and allow the other limit to autorange. Do this by setting an explicit value for the manual limit and `Inf` for the automatic limit. For example, the statement,

```
set(axes_handle, 'XLim', [0 Inf])
```

sets the `XLimMode` property to `auto` and allows MATLAB to determine the maximum value for `XLim`. Similarly, the statement,

```
set(axes_handle, 'XLim', [-Inf 800])
```

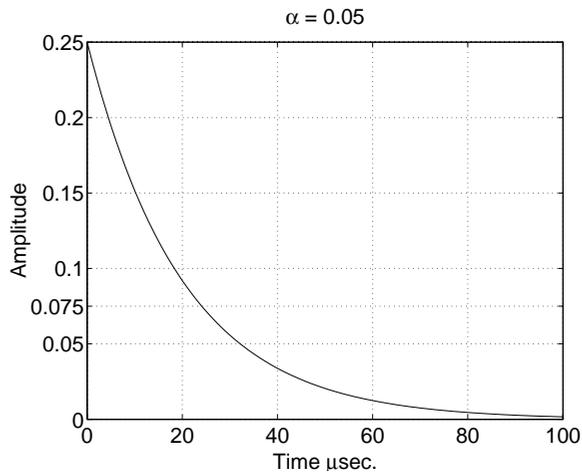
sets the `XLimMode` property to `auto` and allows MATLAB to determine the minimum value for `XLim`.

Setting Tick Mark Locations

MATLAB selects the tick mark location based on the data range to produce equally spaced ticks (for linear graphs). You can specify alternative locations for the tick marks by setting the `XTick`, `YTick`, and `ZTick` properties.

For example, if the value 0.075 is of interest for the amplitude of the function $Ae^{-\alpha t}$, specify tick marks to include that value.

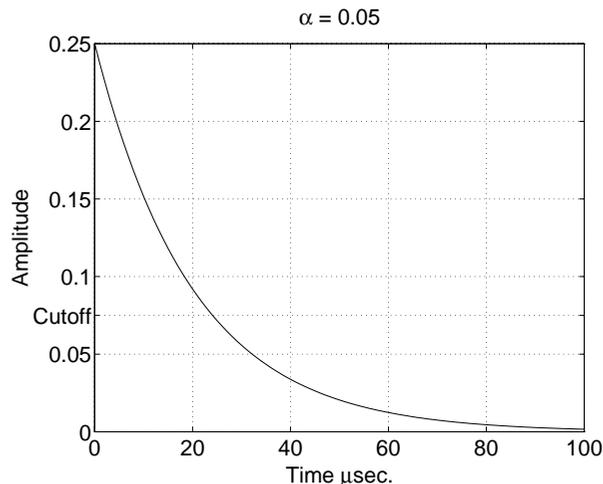
```
set(gca, 'YTick', [0 0.05 0.075 0.1 0.15 0.2 0.25])
```



You can change tick labeling from numbers to strings using the `XTickLabel`, `YTickLabel`, and `ZTickLabel` properties.

For example, to label the y -axis value of 0.075 with the string `Cutoff`, you can specify all y -axis labels as a string, separating each label with the “|” character.

```
set(gca, 'YTickLabel', '0|0.05|Cutoff|0.1|0.15|0.2|0.25')
```



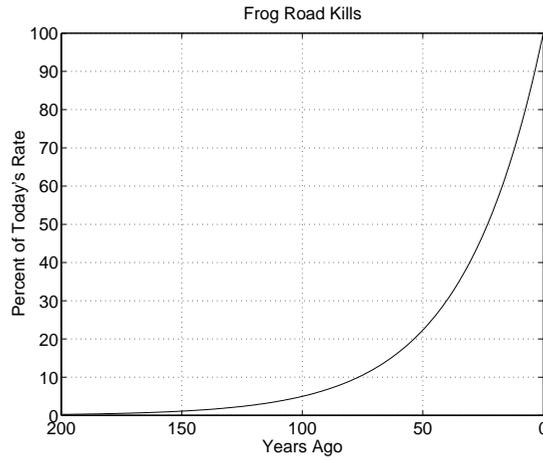
Changing Axis Direction

The `XDir`, `YDir`, and `ZDir` properties control the direction of increasing values on the respective axis. In the default 2-D view, the x -axis values increase from left to right and the y -axis values increase from bottom to top. The z -axis points out of the screen.

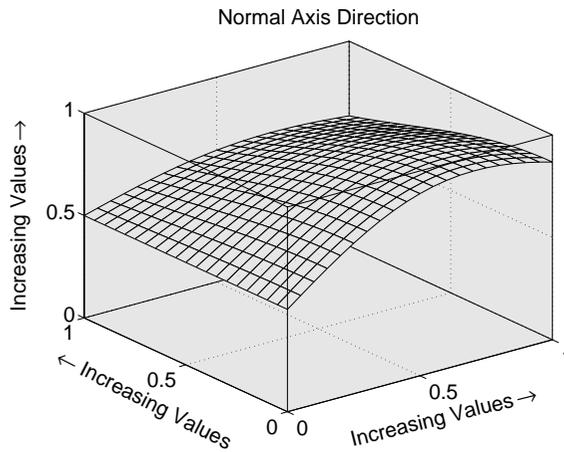
You can change the direction of increasing values by setting the associated property to reverse. For example, setting `XDir` to reverse,

```
set(gca, 'XDir', 'reverse')
```

produces a plot whose x -axis decreases from left to right.



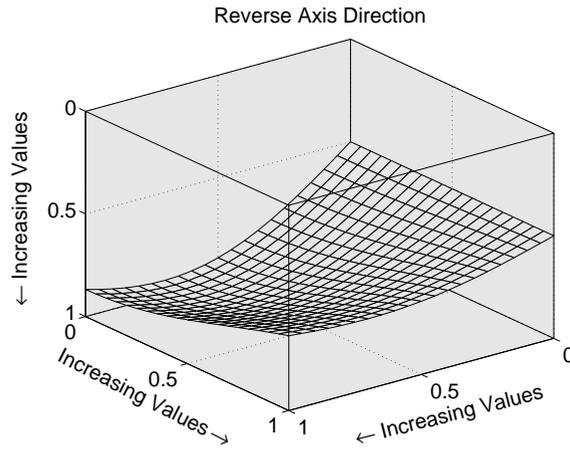
In the 3-D view, the y-axis increases from front to back and the z-axis increases from bottom to top.



Setting the x-, y-, and z-directions to reverse,

```
set(gca, 'XDir', 'rev', 'YDir', 'rev', 'ZDir', 'rev')
```

yields



Using Multiple X and Y Axes

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.

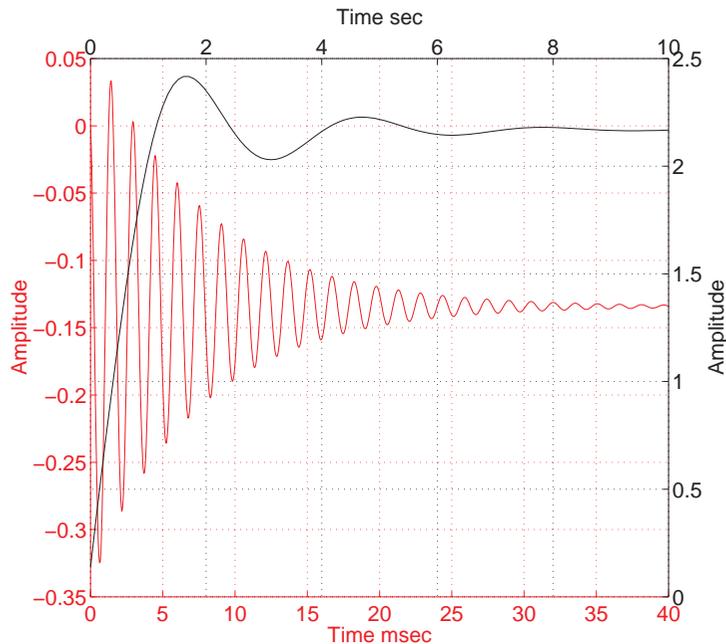
```
h11 = line(x1,y1,'Color','r');
ax1 = gca;
set(ax1,'XColor','r','YColor','r')
```

Next, create another axes at the same location as the first, placing the x -axis on top and the y -axis on the right. Set the axes `Color` to `none` to allow the first axes to be visible and color code the x - and y -axis to match the data.

```
ax2 = axes('Position',get(ax1,'Position'),...
           'XAxisLocation','top',...
           'YAxisLocation','right',...
           'Color','none',...
           'XColor','k','YColor','k');
```

Draw the second set of data in the same color as the x - and y -axis.

```
h12 = line(x2,y2,'Color','k','Parent',ax2);
```



Creating Coincident Grids

Since the two axes are completely independent, MATLAB determines tick mark locations according to the data plotted in each. It is unlikely the gridlines will coincide. This produces a somewhat confusing looking graph, even though the two grids are drawn in different colors. However, if you manually specify tick mark locations, you can make the grids coincide.

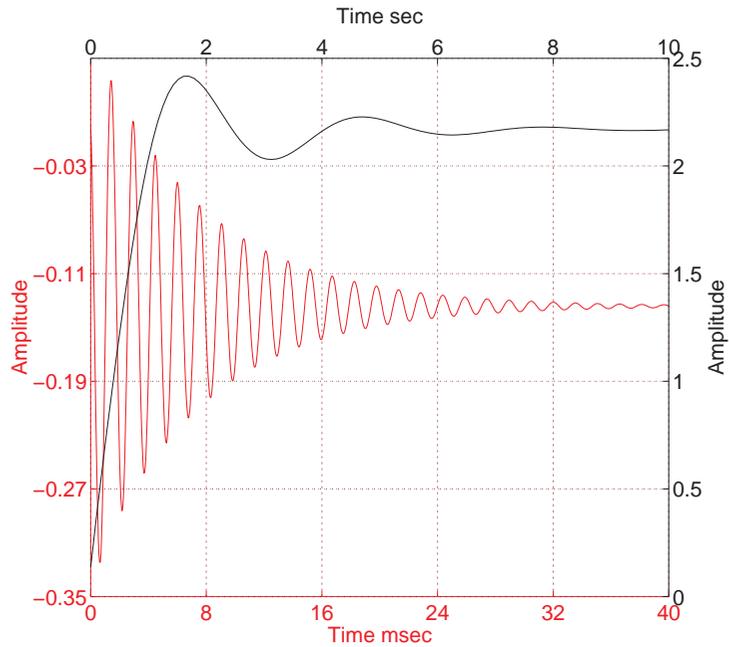
The key is to specify the same number of tick marks along corresponding axis lines (it is also necessary for both axes to be the same size). The following graph of the same data uses six tick marks per axis, equally spaced within the original limits. To calculate the tick mark location, obtain the limits 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, 'XTick', [xlims(1):xinc:xlims(2)], ...
      'YTick', [ylims(1):yinc:ylims(2)])
```

The resulting graph is visually simpler, even though the y-axis on the left has rather odd tick mark values.



Automatic-Mode Properties

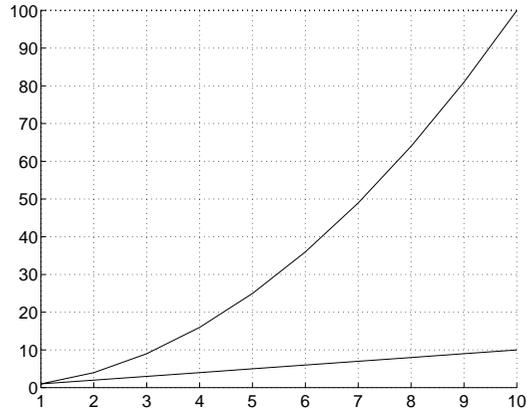
While object creation routines that create axes children do not explicitly change axes properties, some axes properties are under automatic control when their associated mode property is set to auto (which is the default). The following table lists the automatic-mode properties.

Mode 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
TickDirMode	Direction of axis tick marks (in for 2-D, out for 3-D)
XLimMode YLimMode ZLimMode	Limits of the respective x , y , and z axes
XTickMode YTickMode ZTickMode	Tick mark spacing along the respective x , y , and z axes
XTickLabelMode YTickLabelMode ZTickLabelMode	Tick mark labels along the respective x , y , and z axes

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]`.



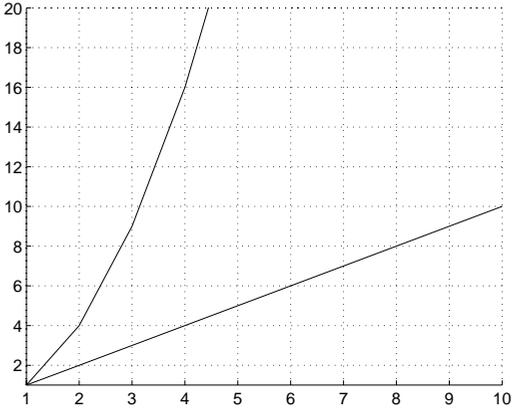
This is because `YLimMode` is `auto`, which always causes MATLAB to recompute the axis limits.

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.



Colors Controlled by Axes

Axes properties specify the color of the axis lines, tick marks, labels, and the background. Properties also control the color of the lines drawn by plotting routines and how image, patch, and surface objects obtain colors from the figure colormap.

The axes properties discussed in this section are listed in the following table.

Property	Characteristic it Controls
Color	Axes background color
XColor, YColor, ZColor	Color of the axis lines, tick marks, gridlines and labels
Title	Title text object handles
XLabel, YLabel, Zlabel	Axis label text object handles
CLim	Controls mapping of graphic object CData to the figure colormap
CLimMode	Automatic or manual control of CLim property
ColorOrder	Line color autcycle order
LineStyleOrder	Line styles autcycle order (not a color, but related to ColorOrder)

Specifying Axes Colors

The default axes background color is set up by the `colordef` command, which is called in your startup file. However, you can easily define your own color scheme.

See `InvertHardCopy` 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')
```

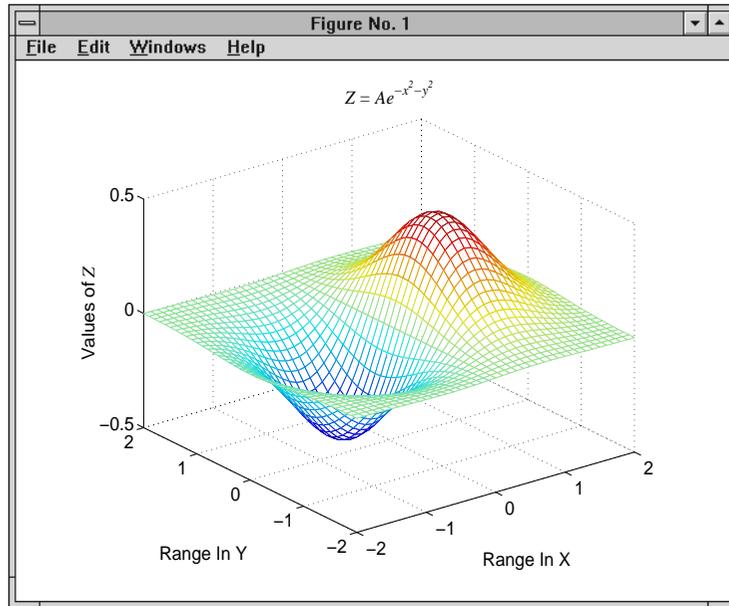
Next, change the color of the text objects used for the title and axis labels.

```
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 the following figure.



You can define default values for the appropriate properties and put these definitions in your `startup.m` file. Titles and axis labels are text objects, so you must set a default color for all text objects, which is a good idea anyway 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.

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 `[cmin cmax]` specifying the `CData` value to map to the first color in the colormap (`cmin`) 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

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

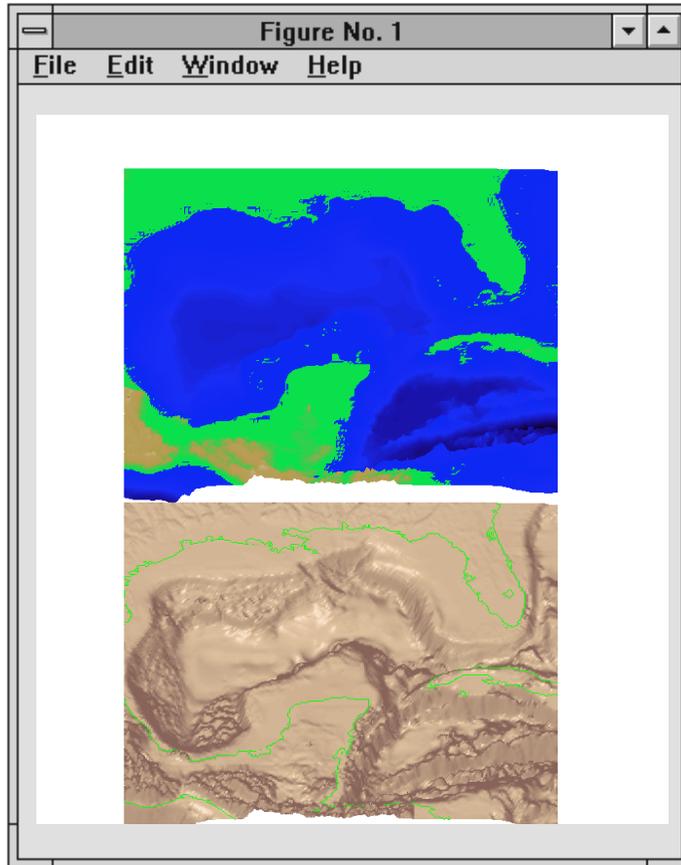
`cm_length` is the length of the colormap. When `CLimMode` is `auto`, MATLAB sets `CLim` to the range of the `CData` of all graphics objects within the axes. However, you can set `CLim` to span any range of values. This allows individual axes within a single figure to use different portions of the figure's colormap. You can create colormaps with different regions, each used by a different axes.

See the `caxis` command for more information on color limits

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 `CLim` that cause each surface to use the section of the colormap containing the appropriate colors.

To calculate the new values for `CLim`, you need to know:

- The total length of the colormap (`CmLength`)
- The beginning colormap slot to use for each axes (`BeginSlot`)
- The ending colormap slot to use for each axes (`EndSlot`)

- The minimum and maximum CData values of the graphic objects contained in the axes. That is, the values of the axes CLim property determined by MATLAB when CLimMode is auto (CDmin and CDmax).

First, define 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.

```
CmLength = size(get(gcf,'Colormap'),1);% Colormap length
BeginSlot1 = 1; % Begining slot
EndSlot1 = size(Lightingmap,1); % Ending slot
BeginSlot2 = EndSlot1+1;
EndSlot2 = CmLength;
CLim1 = get(ax1,'CLim');% CLim values for each axis
CLim2 = get(ax2,'CLim');
```

Defining a Function to Calculate CLim Values

Computing new values for CLim involves determining the portion of the colormap you want each axes to use relative to the total colormap size and scaling its CLim range accordingly. You can define a MATLAB function to do this.

```
function CLim = newclim(BeginSlot,EndSlot,CDmin,CDmax,CmLength)
% Convert slot number and range
% to percent of colormap
PBeginSlot = (BeginSlot - 1) / (CmLength - 1);
PEndSlot = (EndSlot - 1) / (CmLength - 1);
PCmRange = PEndSlot - PBeginSlot;
% Determine range and min and max
```

```
%           of new CLim values
DataRange  = CDmax - CDmin;
ClimRange  = DataRange / PCmRange;
NewCmin    = CDmin - (PBeginSlot * ClimRange);
NewCmax    = CDmax + (1 - PEndSlot) * ClimRange;
CLim       = [NewCmin, NewCmax];
```

The input arguments are identified in the bulleted list above. 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 determines 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 Function

Use the `newclim` M-file to set the CLim values of each axes. The statement,

```
set(ax1, 'CLim', newclim(65, 120, clim1(1), clim1(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.

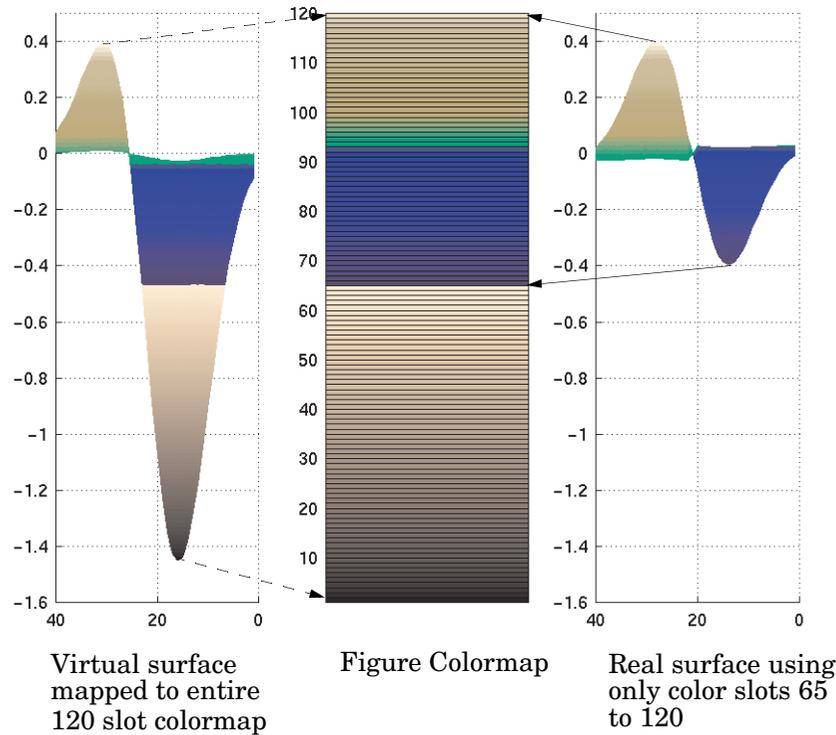
```
set(ax2, 'CLim', newclim(1, 64, clim1(1), clim1(2)))
```

How the Function Works

MATLAB enables you to specify any values for the axes CLim property, even if these values do not correspond to the CData of the graphics objects displayed in the axes. 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.



Defining the Color of Lines 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 `colordef` command defines various color order schemes for different background colors. `colordef` is typically called in the `matlabrc` file, which is executed during MATLAB startup.

Defining Your Own ColorOrder

You can redefine `ColorOrder` to be any m -by-3 matrix of RGB values, where m is the number of colors. However, high-level functions like `plot` and `plot3` reset most axes properties (including `ColorOrder`) to the defaults each time you call them. To use your own `ColorOrder` definition you must take one of the following three steps:

- Define a default `ColorOrder` on the figure or root level, or
- Change the axes `NextPlot` property to `add` or `replacechildren`, 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 existing plots.

By default, `NextPlot` is set to `replace`, which is equivalent to a `cla reset` command (i.e., delete all axes children and reset all properties, except `Position`, to their defaults). If you set `NextPlot` to `replacechildren`,

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

MATLAB deletes the axes children, but does not reset axes properties. This is equivalent to a `cla` command without the `reset`.

After setting `NextPlot` to `replacechildren`, you can redefine the `ColorOrder` property and call `plot` and `plot3` without affecting the `ColorOrder`.

Setting `NextPlot` to `add` is the equivalent of issuing the `hold on` command. This setting prevents MATLAB from resetting the `ColorOrder` property, but it does not clear the axes children with each call to a plotting function.

Using the line Function. The behavior of the `line` function depends on its calling syntax. When you use the informal form (which does not include any explicit property definitions).

```
line(x,y,z)
```

`line` obeys the `ColorOrder` property, but does not clear the axes with each invocation or change the view to 3-D (as `plot3` does). However, `line` can be useful for creating your own plotting functions where you do not want the automatic behavior of `plot` or `plot3`, but you do want multiline graphs to use a particular `ColorOrder`.

Line Styles Used for Plotting – `LineStyleOrder`

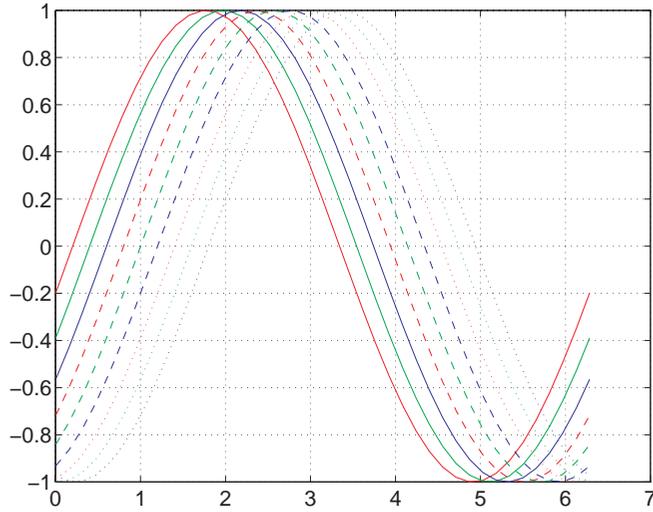
The axes `LineStyleOrder` property is analogous to the `ColorOrder` property. It specifies the line styles to use for multiline plots created with the `plot` and `plot3` functions. MATLAB increments the line style only after using all of the colors in the `ColorOrder` property. It then uses all the colors again with the second line style, and so on.

For example, define a default `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.

3-D Visualization

This section discusses visualization techniques and illustrates the application of these techniques to specific types of data.

Creating 3-D Graphs	3-D line and surface graphs
Defining the View	Control camera, zooming, projection, compose a scene, control the aspect ratio of the axes
Lighting as a Visualization Tool	Lighting effects you can employ to add realism and improve shape definition in 3-D views
Transparency	Various techniques for making objects translucent
Creating 3-D Models with Patches	Define 3-D shell representations of physical shapes using patch objects
Volume Visualization Techniques	Visualize gridded 3-D volume data (both scalar and vector)

Related Information

The following section provide information that is useful in understanding the techniques described in this section.

Graphics	<p>Fundamentals of plotting in MATLAB, including standard plotting routines (line plots, pie charts, histograms, etc.), graph formatting and annotation.</p> <p>It also covers graphics object hierarchy, manipulating object properties, and the use of important figure and axes properties.</p>
----------	--



Creating 3-D Graphs

A Typical 3-D Graph (p. 10-2)

The steps to follow to create a typical 3-D graph.

Line Plots of 3-D Data (p. 10-3)

Lines plots of data having x -, y -, and z -coordinates.

Representing a Matrix as a Surface
(p. 10-5)

Graphing matrix (2-D array) data on a rectangular grid.

Coloring Mesh and Surface Plots
(p. 10-13)

Techniques for coloring surface and mesh plots, including colormaps, truecolor, and texture mapping.

A Typical 3-D Graph

This table illustrates typical steps involved in producing 3-D scenes containing either data graphs or models of 3-D objects. Example applications include pseudocolor surfaces illustrating the values of functions over specific regions and objects drawn with polygons and colored with light sources to produce realism. Usually, you follow either step 4 or step 5.

Step	Typical Code
1 Prepare your data	<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>
4 Set colormap and shading algorithm	<code>colormap hot shading interp set(h, 'EdgeColor', 'k')</code>
5 Add 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>
6 Set viewpoint	<code>view([30,25]) set(gca, 'CameraViewAngleMode', 'Manual')</code>
7 Set axis limits and tick marks	<code>axis([5 15 5 15 -8 8]) set(gca, 'ZTickLabel', 'Negative Positive')</code>
8 Set aspect ratio	<code>set(gca, 'PlotBoxAspectRatio', [2.5 2.5 1])</code>
9 Annotate the graph with axis labels, legend, and text	<code>xlabel('X Axis') ylabel('Y Axis') zlabel('Function Value') title('Peaks')</code>
10 Print graph	<code>set(gcf, 'PaperPositionMode', 'auto') print -dps2</code>

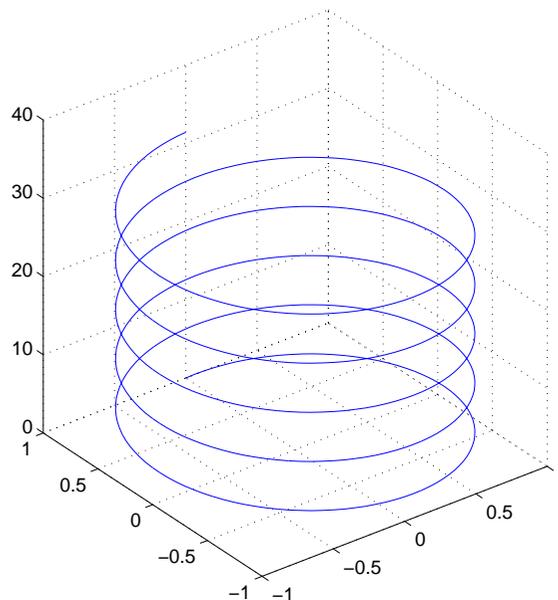
Line Plots of 3-D Data

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

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

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

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



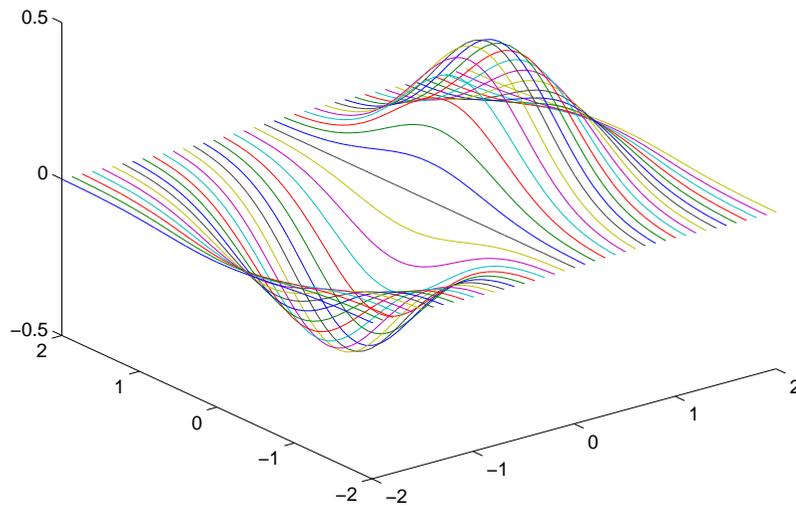
Plotting Matrix Data

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

```
[X,Y] = meshgrid([-2:0.1:2]);
```

```
Z = X.*exp(-X.^2-Y.^2);  
plot3(X,Y,Z)  
grid on
```

Notice how MATLAB cycles through line colors.



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

```
mesh(Z)
```

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

```
surf(Z)
```

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

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

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

Visualizing Functions of Two Variables

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

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

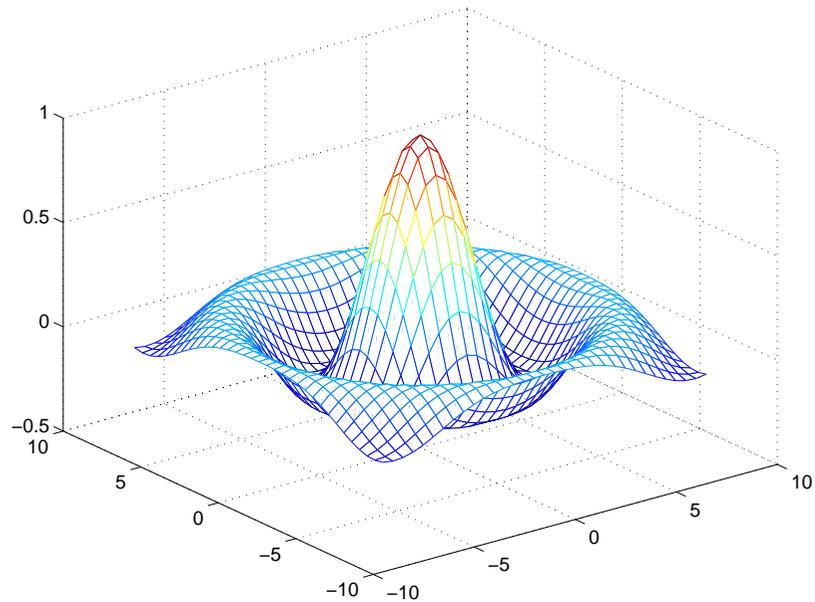
To illustrate the use of `meshgrid`, consider the $\sin(r)/r$ or `sinc` function. To evaluate this function between -8 and 8 in both x and y , you need pass only one vector argument to `meshgrid`, which is then used in both directions.

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

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

Forming the `sinc` function and plotting Z with `mesh` results in the 3-D surface.

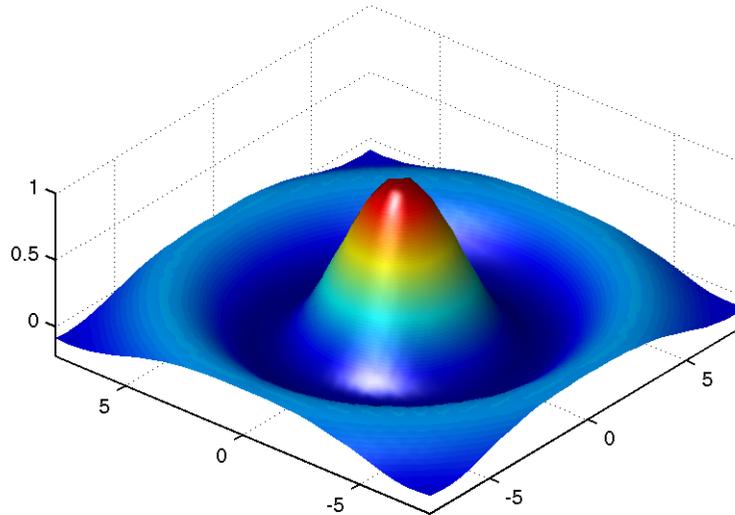
```
Z = sin(R)./R;  
mesh(X,Y,Z)
```



Emphasizing Surface Shape

MATLAB provides a number of techniques that can enhance the information content of your graphs. For example, this graph of the sinc function uses the same data as the previous graph, but employs lighting and view adjustment to emphasize the shape of the graphed function (`daspect`, `axis`, `camlight`, `view`).

```
surf(X,Y,Z,'FaceColor','interp',...
      'EdgeColor','none',...
      'FaceLighting','phong')
daspect([5 5 1])
axis tight
view(-50,30)
camlight left
```



See the `surf` function for more information on surface plots.

Surface Plots of Nonuniformly Sampled Data

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

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

Example— Displaying Nonuniform Data on a Surface

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

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

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

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

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

- 3 Now use these points to generate a uniformly spaced grid.

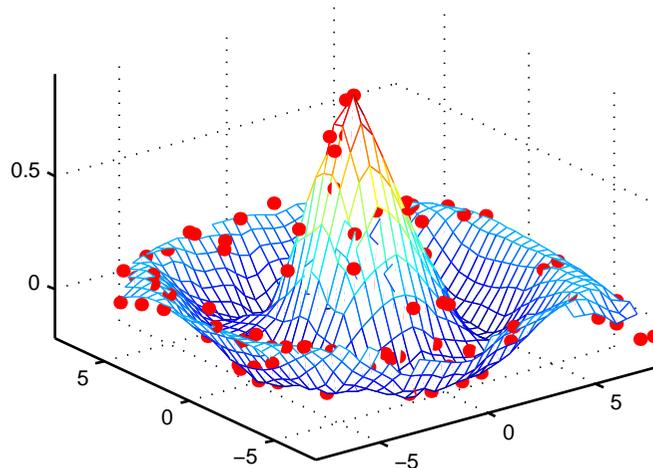
```
[X,Y] = meshgrid(xlin,ylin);
```

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

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

- 5 Plotting the interpolated and the nonuniform data produces

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



Parametric Surfaces

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

$$\text{mesh}(x, y, Z, C)$$

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

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

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

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

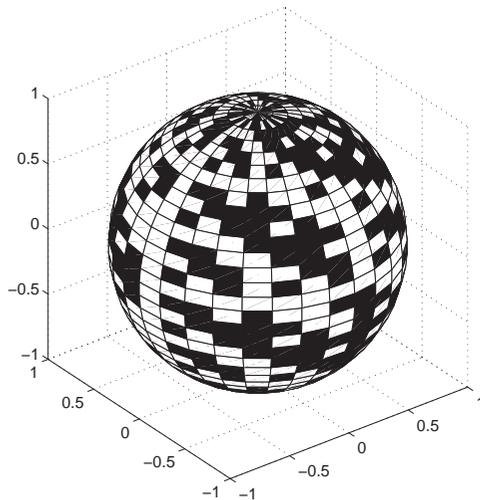
$$\text{mesh}(X, Y, Z, C)$$

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

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

This example uses spherical coordinates to draw a sphere and color it with the pattern of pluses and minuses in a Hadamard matrix, an orthogonal matrix used in signal processing coding theory. The vectors θ and ϕ are in the range $\pi \leq \theta \leq 2\pi$ and $\pi/2 \leq \phi \leq 3\pi/2$. Because 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
```

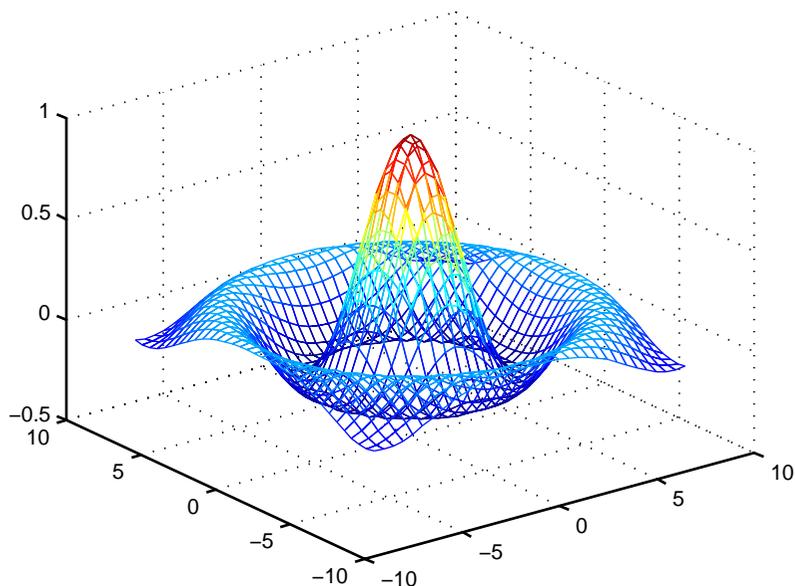


Hidden Line Removal

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 predefined range of colors called a *colormap*.

Coloring Techniques

There are three coloring techniques:

- Indexed Color – MATLAB colors the surface plot by assigning each data point an index into the figure's colormap. The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated).
- Truecolor – MATLAB colors the surface plot using the explicitly specified colors (i.e., the RGB triplets). The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated). To be rendered accurately, truecolor requires computers with 24-bit displays; however, MATLAB simulates truecolor on indexed systems. See the shading command for information on the types of shading.
- Texture Mapping – Texture mapping displays a 2-D image mapped onto a 3-D surface.

Types of Color Data

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

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

Colormaps

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

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

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

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

RGB Color Components

This table lists some representative RGB color definitions.

Red	Green	Blue	Color
0	0	0	black
1	1	1	white
1	0	0	red
0	1	0	green
0	0	1	blue
1	1	0	yellow
1	0	1	magenta
0	1	1	cyan
0.5	0.5	0.5	gray
0.5	0	0	dark red

Red	Green	Blue	Color
1	0.62	0.40	copper
0.49	1	0.83	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)`.

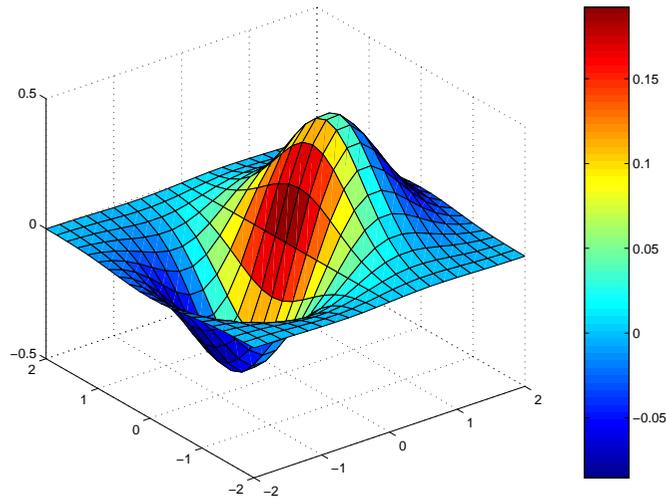
If you use long colormaps (> 64 colors) in each of several figures windows, it may become necessary for the operating system to swap in different color lookup tables as the active focus is moved among the windows. See "Controlling How MATLAB Uses Color" in the "Figure Properties" chapter for more information 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. Note how the colorbar indicates the mapping of data value to color with the axis labels.



Indexed Color Surfaces – Direct and Scaled Colormapping

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

Direct Mapping

Direct mapping uses the color data directly as indices into the colormap. For example, a value of 1 points to the first color in the colormap, a value of 2 points to the second color, and so on. If the color data is noninteger, MATLAB rounds it 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.

```
colormap_index = fix((color_data-cmin)/(cmax-cmin)*cm_length)+1
```

`cm_length` is the length of the colormap.

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

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

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

See `surface` for more information on specifying color data.

Specifying Indexed Colors

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

With two matrix arguments, the statement

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

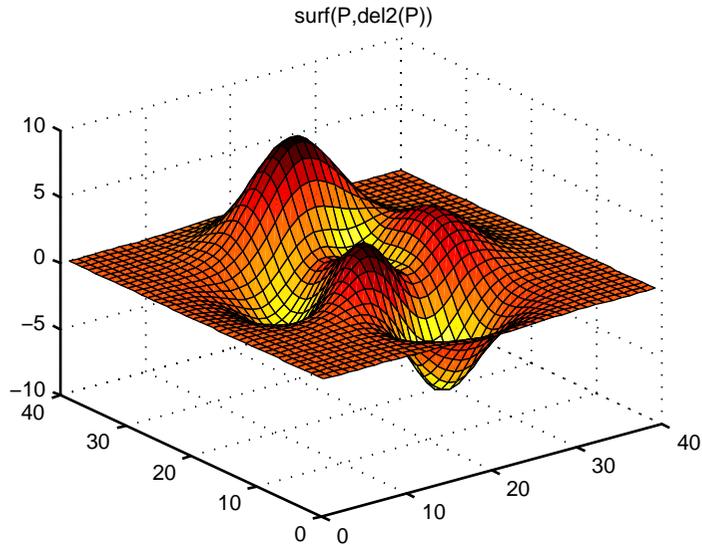
independently specifies the color using the second argument.

Example - Mapping Surface Curvature to Color

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

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

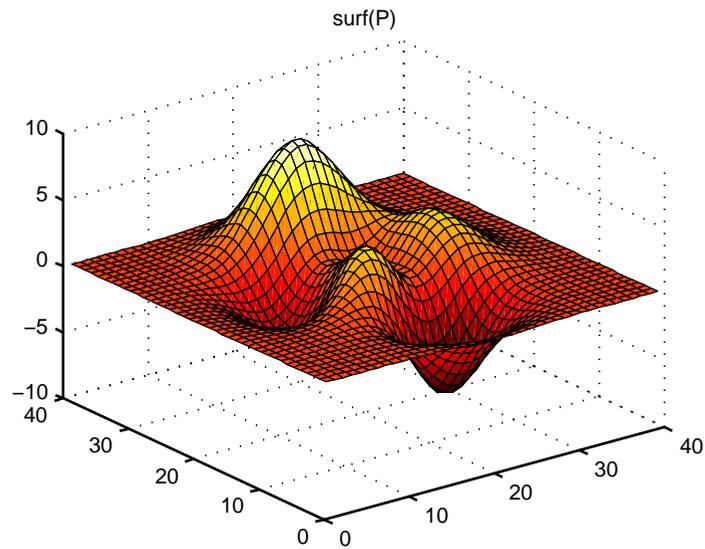
Creating a color array by applying the Laplacian to the data is useful because it causes regions with similar curvature to be drawn in the same color.



Compare this surface coloring with that produced by the statements

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

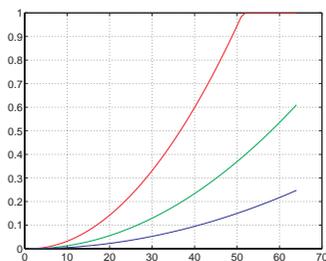
which use the same colormap, but maps regions with similar z value (height above the x-y plane) to the same color.



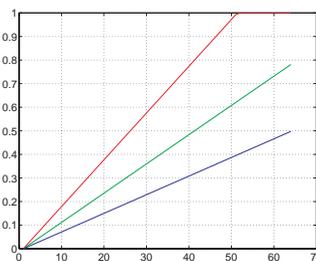
Altering Colormaps

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

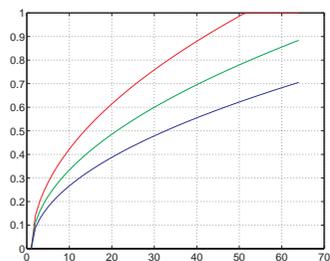
`brighten(copper,-0.5)`



`copper`



`brighten(copper,0.5)`



NTSC Color Encoding

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

```
b = .30*red + .59*green + .11*blue
   = sum(diag([.30 .59 .11])*map')';
```

Using the nonlinear grayscale map,

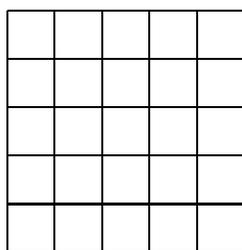
```
colormap([b b b])
```

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

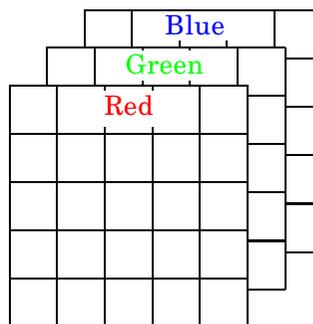
Truecolor Surfaces

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

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



m -by- n matrix defining surface plot



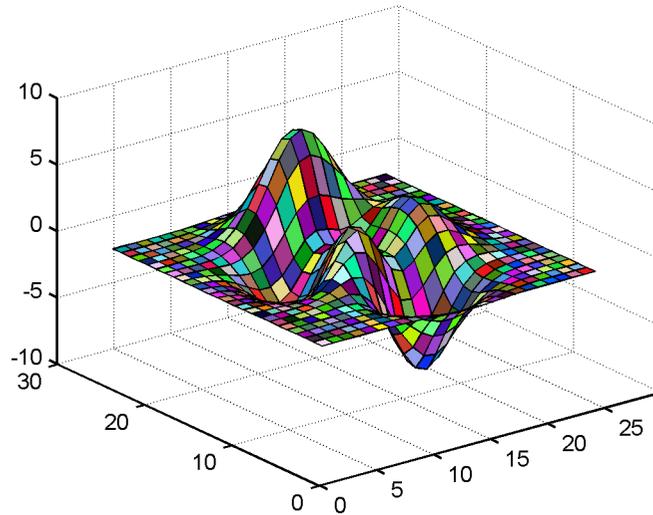
Corresponding m -by- n -by-3 matrix specifying truecolor for the surface plot

For example, the statements

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

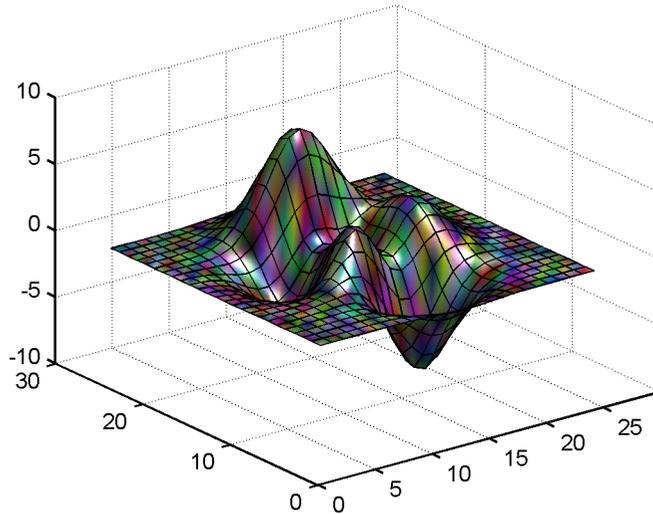
```
C(:,:,3) = rand(25);  
surf(Z,C)
```

create a plot of the peaks matrix with random coloring.



You can set surface properties as with indexed color.

```
surf(Z,C,'FaceColor','interp','FaceLighting','phong')  
camlight right
```



Rendering Method for Truecolor

MATLAB always uses either OpenGL or the zbuffer 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 `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 “Dithering Truecolor on Indexed Color Systems” in the “Figure Properties” chapter for more information on the use of a `dithermap`.

Texture Mapping

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

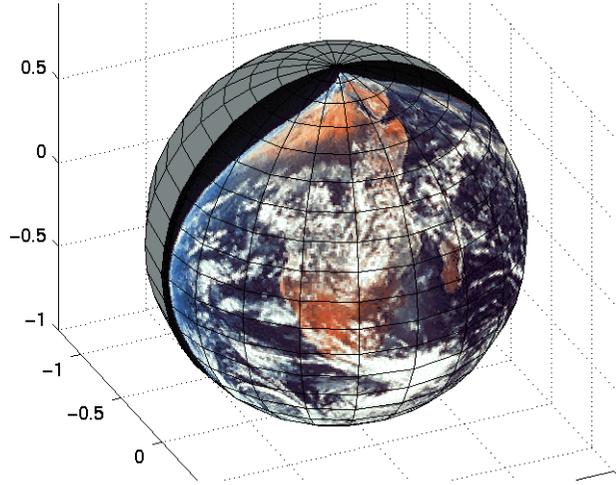
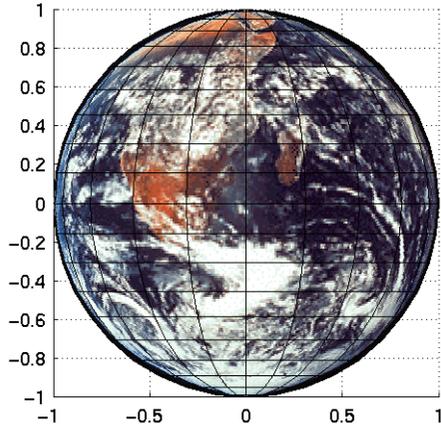
Texture mapping allows the dimensions of the color data array to be different from the data defining the surface plot. You can apply an image of arbitrary size to any surface. MATLAB interpolates texture color data so that it is mapped to the entire surface.

Example – Texture Mapping a Surface

This example creates a spherical surface using the sphere function and texture maps it with an image of the earth taken from space. Because the earth image is a view of earth from one side, this example maps the image to only one side of the sphere, padding the image data with 1s. In this case, the image data is a 257-by-250 matrix so it is padded equally on each side with two 257-by-125 matrices of 1s by concatenating the three matrices 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');
hemisphere = [ones(257,125),...
              X,...
              ones(257,125)];
set(h,'CData',flipud(hemisphere),'FaceColor','texturemap')
colormap(map)
axis equal
view([90 0])
set(gca,'CameraViewAngleMode','manual')
view([65 30])
```



Defining the View

Viewing Overview (p. 11-2)	Overview of topics covered in this chapter.
Setting the Viewpoint with Azimuth and Elevation (p. 11-4)	Using the simple azimuth and elevation view model to define the viewpoint. Includes definition and examples.
Defining Scenes with Camera Graphics (p. 11-8)	Using the camera view model to control 3-D scenes. Illustration defines terms.
View Control with the Camera Toolbar (p. 11-9)	Camera tools provide a set of functionality for manipulating 3-D scenes.
Camera Graphics Functions (p. 11-21)	Functions that control the camera view model.
Example — Dollying the Camera (p. 11-22)	Example illustrates how to reposition a scene when the user clicks over an image.
Example — Moving the Camera Through a Scene (p. 11-24)	Example illustrates how to move a camera through a scene along a path traced by a stream line. Also show how to move a light source with the camera.
Low-Level Camera Properties (p. 11-30)	Description of the graphic object properties that control the camera.
View Projection Types (p. 11-36)	Orthographic and perspective project types compared and illustrated. Also, the interaction between camera properties and projection type.
Understanding Axes Aspect Ratio (p. 11-41)	How MATLAB determines the axes aspect ratio for graphs. Also, how you can specify aspect ratio.
Axes Aspect Ratio Properties (p. 11-46)	Axes properties that control the aspect ratio and how to set them to achieve particular results.

Viewing Overview

The *view* is the particular orientation you select to display your graph or graphical scene. The term viewing refers to the process of displaying a graphical scene from various directions, zooming in or out, changing the perspective and aspect ratio, flying by, and so on.

This section describes how to define the various viewing parameters to obtain the view you want. Generally, viewing is applied to 3-D graphs or models, although you may want to adjust the aspect ratio of 2-D views to achieve specific proportions or make a graph fit in a particular shape.

MATLAB viewing is composed of two basic areas:

- Positioning the viewpoint to orient the scene
- Setting the aspect ratio and relative axis scaling to control the shape of the objects being displayed

Positioning the Viewpoint

- **Setting the Viewpoint** — discusses how to specify the point from which you view a graph in terms of azimuth and elevation. This is conceptually simple, but does have limitations.
- **Defining Scenes with Camera Graphics, View Control with the Camera Toolbar, and Camera Graphics Functions** — describe how to compose complex scenes using the MATLAB camera viewing model.
- **Dollying the Camera and Moving the Camera Through a Scene** — illustrate programming techniques for moving the view around and through scenes.
- **Low-Level Camera Properties** — lists the graphics properties that control the camera and illustrates the effects they cause.

Setting the Aspect Ratio

- **View Projection Types** — discusses orthographic and perspective projection types and illustrates their use.
- **Understanding Axes Aspect Ratio and Axes Aspect Ratio Properties** — describe how MATLAB sets the aspect ratio of the axes and how you can select the most appropriate setting for your graphs.

Default Views

MATLAB automatically sets the view when you create a graph. The actual view that MATLAB selects depends on whether you are creating a 2- or 3-D graph. See “Default Viewpoint Selection” on page 11-31 and “Default Aspect Ratio Selection” on page 11-47 for a description of how MATLAB defines the standard view.

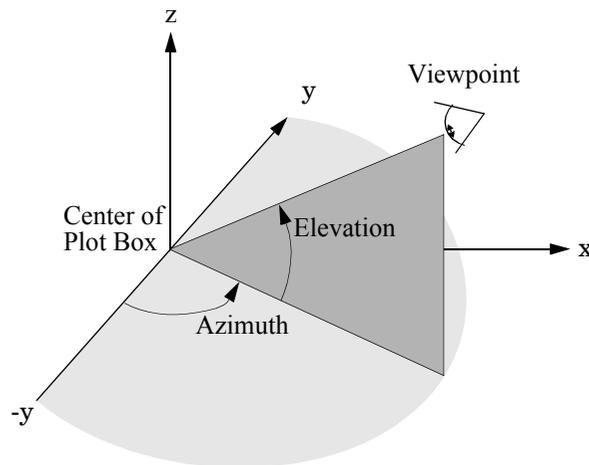
Setting the Viewpoint with Azimuth and Elevation

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

Azimuth and Elevation

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.



Default 2-D and 3-D Views

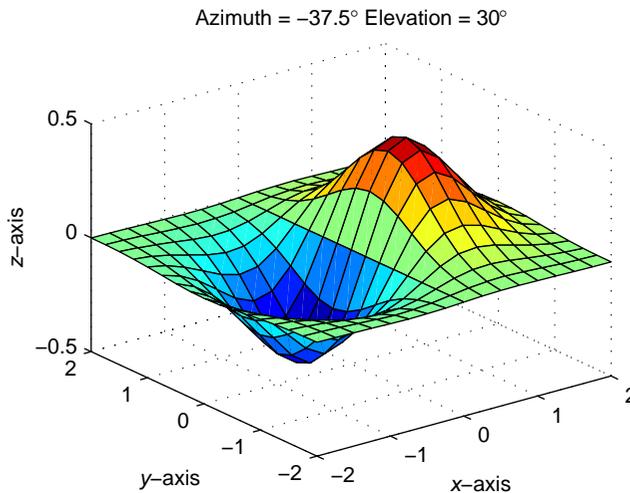
MATLAB automatically selects a viewpoint that is determined by whether the plot is 2-D or 3-D:

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

Examples of Views Specified with Azimuth and Elevation

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

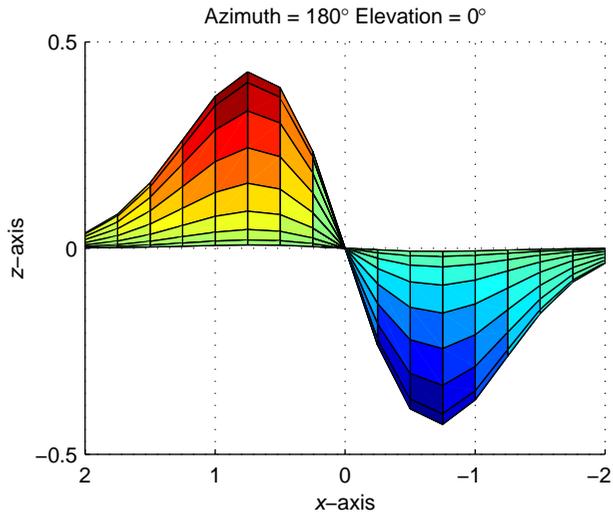
```
[X,Y] = meshgrid([-2:.25:2]);  
Z = X.*exp(-X.^2 -Y.^2);  
surf(X,Y,Z)
```



The statement

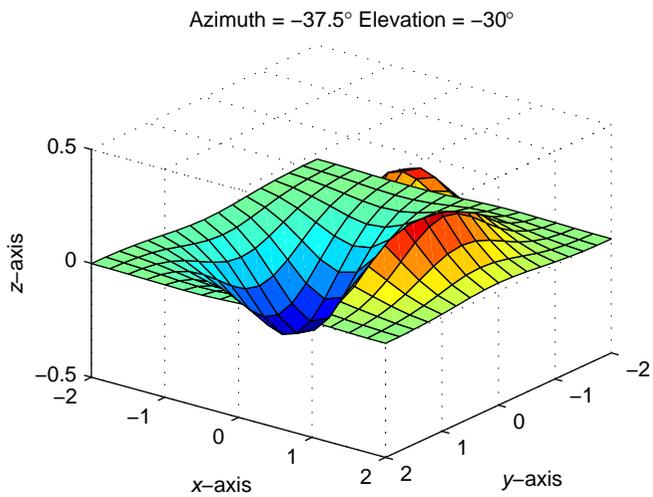
```
view([180 0])
```

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



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

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



Limitations of Azimuth and Elevation

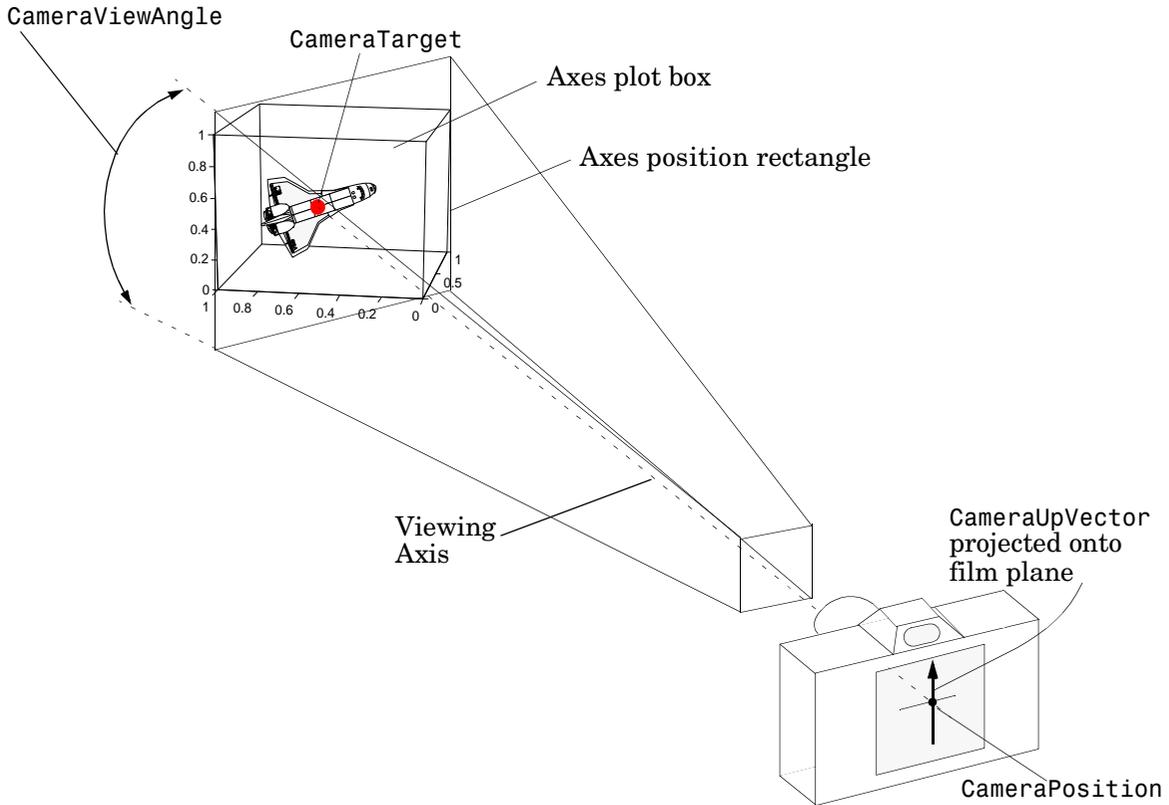
Specifying the viewpoint in terms of azimuth and elevation is conceptually simple, but it has limitations. It does not allow you to specify the actual position of the viewpoint, just its direction, and the z -axis is always pointing up. It does not allow you to zoom in and out on the scene or perform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

MATLAB camera graphics provides greater control than the simple adjustments allowed with azimuth and elevation. The following sections discuss how to use camera properties to control the view.

Defining Scenes with Camera Graphics

When you look at the graphics objects displayed in an axes, you are viewing a scene from a particular location in space that has a particular orientation with regard to the scene. MATLAB provides functionality, analogous to that of a camera with a zoom lens, that enables you to control the view of the scene created by MATLAB.

This picture illustrates how the camera is defined in terms of properties of the axes.



View Control with the Camera Toolbar

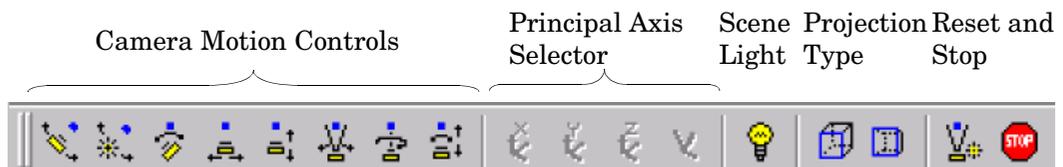
The Camera Toolbar enables you to perform a number of viewing operations interactively. To use the Camera Toolbar:

- Display the toolbar by selecting **Camera Toolbar** from the figure window's **View** menu.
- Select the type of camera motion control you want to use.
- Position the cursor over the figure window and click, hold down the right mouse button, then move the cursor in the desired direction.

MATLAB updates the display immediately as you move the mouse.

Camera Toolbar

The toolbar contains the following parts:



- Camera Motion Controls — these tools select which camera motion function to enable. You can also access the camera motion controls from the **Tools** menu.
- Principal Axis Selector — some camera controls operate with respect to a particular axis. These selectors enable you to select the principal axis or to select non-axis constrained motion. The selectors are grayed out when not applicable to the currently selected function. You can also access the principal axis selector from the **Tools** menu.
- Scene Light — The scene light button toggles a light source on or off in the scene (one light per axes).
- Projection Type — You can select orthographic or perspective projection types.
- Reset and Stop — Reset returns the scene to the standard 3-D view. Stop causes the camera to stop moving (this can be useful if you apply too much

cursor movement). You can also access the an expanded set of reset functions from the **Tools** menu.

Principal Axes

The principal axis of a scene defines the direction that is oriented upward on the screen. For example, a MATLAB surface plot aligns the up direction along the positive z axis.

Principal axes constrain camera-tool motion along axes that are (on the screen) parallel and perpendicular to the principal axis that you select. Specifying a principal axis is useful if your data is define with respect to a specific axis. Z is the default principal axis, since this matches the MATLAB default 3-D view.

Three of the camera tools (Orbit, Pan/Tilt, and Walk) allow you to select a principal axis, as well as axis-free motion. On the screen, the axes of rotation are determined by a vertical and a horizontal line, both of which pass through the point defined by the CameraTarget property and are parallel and perpendicular to the principal axis.

For example, when the principal axis is z, movement occurs about:

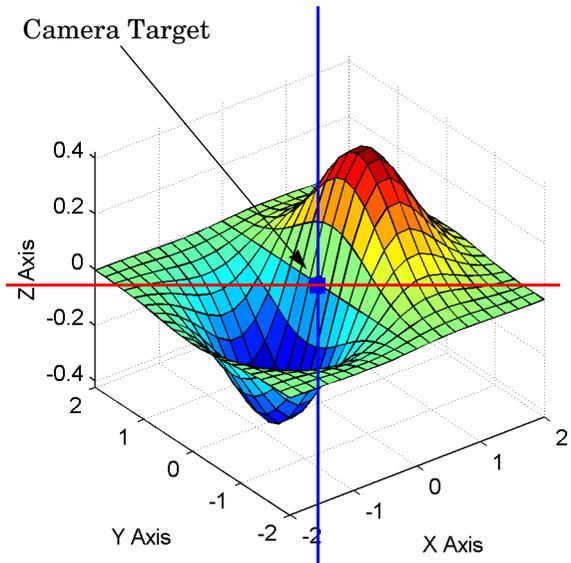
- A vertical line that passes through the camera target and is parallel to the z axis
- A horizontal line that passes through the camera target and is perpendicular to the z axis

This means the scene (or camera, as the case may be) moves in an arc whose center is at the camera target. The following picture illustrates the rotation axes for a z principal axis.

Horizontal cursor motion results in rotation about the (blue) vertical axis.

Vertical cursor motion causes rotation about the (red) horizontal axis.

Cursor Motion

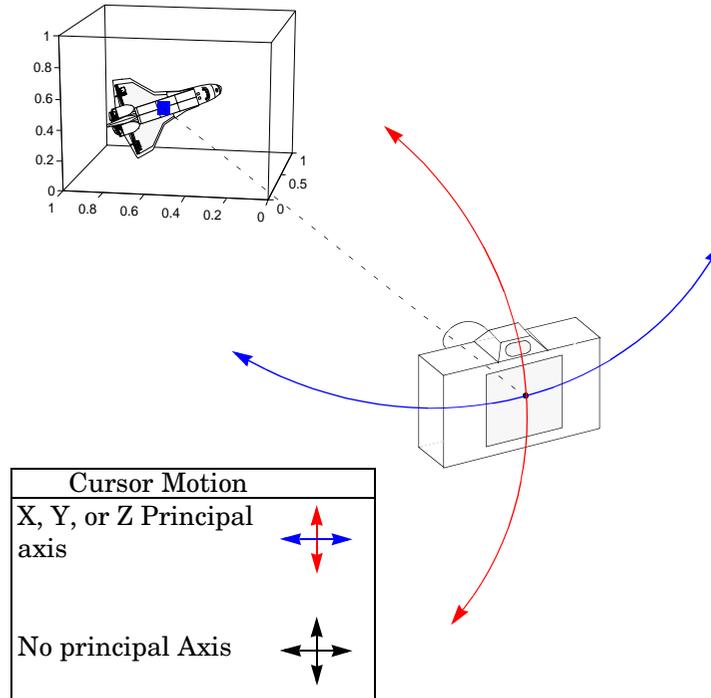


The axes of rotation always pass through the camera target.

Optimizing for 3-D Camera Motion

When you create a plot, MATLAB displays it with an aspect ratio that fits the figure window. This behavior may not create an optimum situation for the manipulation of 3-D graphics as it can lead to distortion as you move the camera around scene. To avoid possible distortion, it is best to switch to a 3-D visualization mode (enabled from the command line with the command `axis vis3d`). When using the camera toolbar, MATLAB automatically switches to the 3-D visualization mode, but warns you first with the following dialog box.

Graphics Properties. Orbit camera changes the `CameraPosition` property while keeping the `CameraTarget` fixed.



Orbit Scene Light



The scene light is a light source that is placed with respect to the camera position. By default, the scene light is positioned to the right of the camera (i.e., `camlight right`). Orbit Scene Light changes the light's offset from the camera position. There is only one scene light, however, you can add other lights using the light command.

Toggle the scene light on and off by clicking on the yellow light bulb icon.

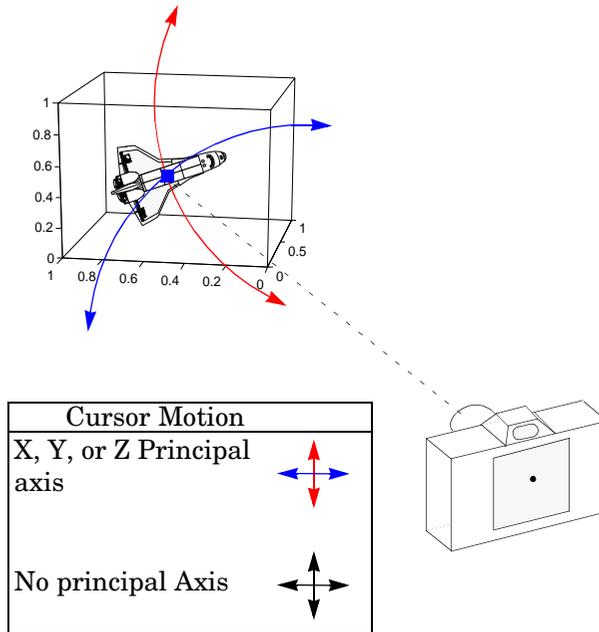
Graphics Properties. Orbit Scene Light moves the scene light by changing the light's Position property.

Pan/Tilt Camera



Pan/Tilt Camera moves the point in the scene that the camera points to while keeping the camera fixed. The movement occurs in an arc about the z-axis by default. You can select x-, y-, z-, or free-axis rotation using the Principal Axes Selectors.

Graphics Properties. Pan/Tilt Camera moves the point in the scene that the camera is pointing to by changing the CameraTarget property.

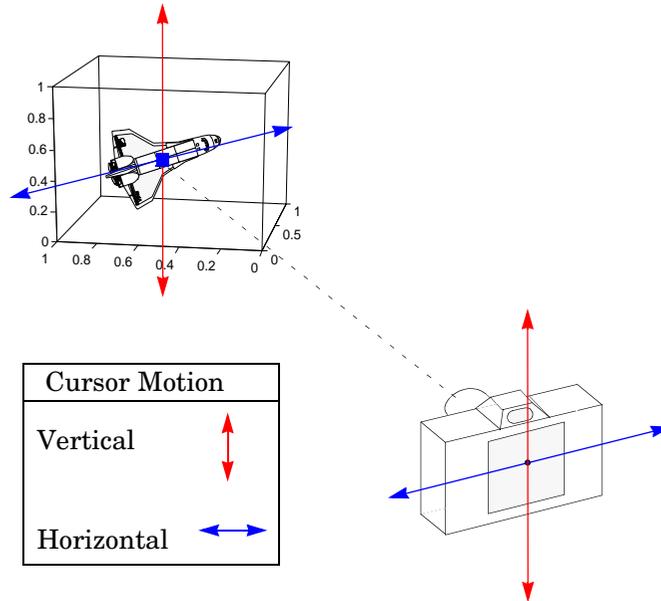


Move Camera Horizontally/Vertically



Moving the cursor horizontally or vertically (or any combination of the two) moves the scene in the same direction.

Graphics Properties. The horizontal and vertical movement is achieved by moving the `CameraPosition` and the `CameraTarget` in unison along parallel lines.

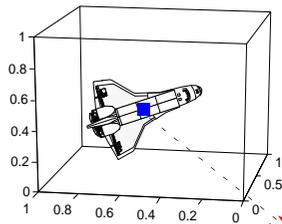


Move Camera Forward and Backwards

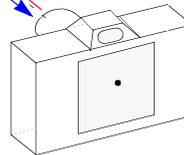


Moving the cursor up or to the right moves the camera towards the scene. Moving the cursor down or to the left moves the camera away from the scene. It is possible to move the camera through objects in the scene and to the other side of the camera target.

Graphics Properties. This function moves the CameraPosition along the line connecting the camera position and the camera target.



Cursor Motion	
Closer to target	
Farther from target	

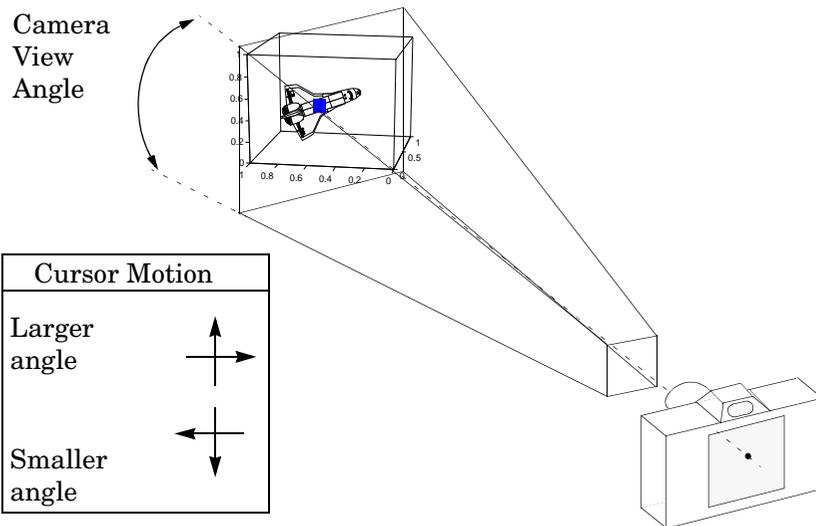


Zoom Camera



Zoom Camera makes the scene larger as you move the cursor up or to the right and smaller as you move the cursor down or to the left. Zooming does not move the camera and therefore cannot move the viewpoint through objects in the scene.

Graphics Properties. Zoom is implemented by changing the `CameraViewAngle`. The larger the angle, the smaller the scene appears, and vice versa.

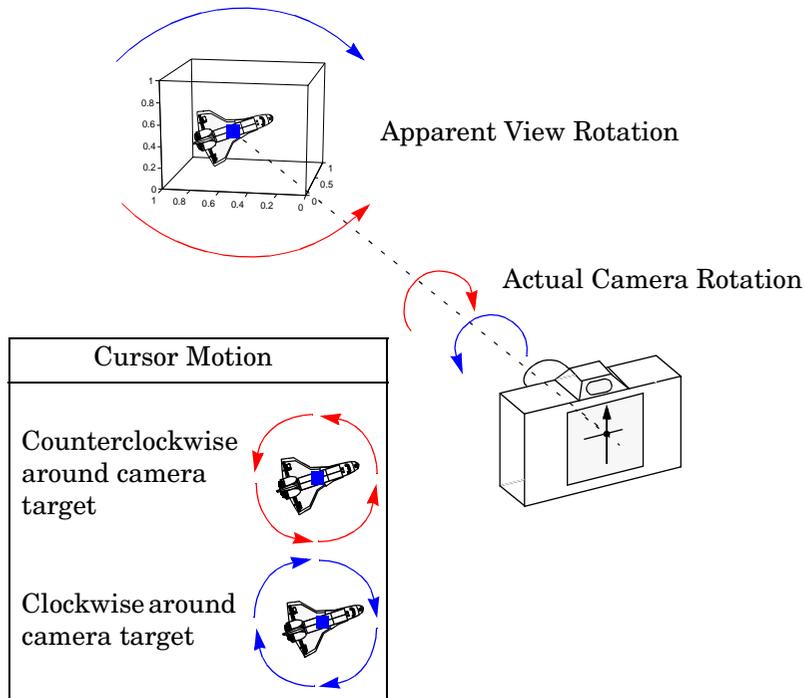


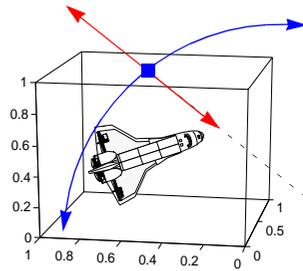
Camera Roll



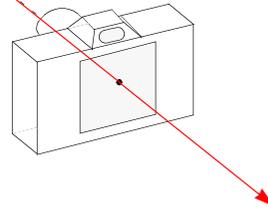
Camera Roll rotates the camera about the viewing axis, thereby rotating the view on the screen.

Graphics Properties. Camera Roll changes the CameraUpVector





Cursor Motion	
Towards or away from scene	↕
Pan left or right	↔



Camera Graphics Functions

The following table lists MATLAB functions that enable you to perform a number of useful camera maneuvers. The individual command descriptions provide information on using each one.

Functions	Purpose
camdolly	Move camera position and target
camlookat	View specific objects
camorbit	Orbit the camera about the camera target
campan	Rotate the camera target about the camera position
campos	Set or get the camera position
camproj	Set or get the projection type (orthographic or perspective)
camroll	Rotate the camera about the viewing axis
camtarget	Set or get the camera target location
camup	Set or get the value of the camera up vector
camva	Set or get the value of the camera view angle
camzoom	Zoom the camera in or out on the scene

Example – Dollying the Camera

In the camera metaphor, a dolly is a stage that enables movement of the camera side to side with respect to the scene. The `camdolly` command implements similar behavior by moving both the position of the camera and the position of the camera target in unison (or just the camera position if you so desire).

This example illustrates how to use `camdolly` to explore different regions of an image.

Summary of Techniques

This example:

- Uses `ginput` to obtain the coordinates of locations on the image
- Uses the `camdolly data coordinates` option to move the camera and target to the new position based on coordinates obtained from `ginput`
- Uses `camva` to zoom in and to fix the camera view angle, which is otherwise under automatic control

Implementation

First load the Cape Cod image and zoom in by setting the camera view angle (using `camva`).

```
load cape
image(X)
colormap(map)
axis image
camva(camva/2.5)
```

Then use `ginput` to select the x - and y -coordinates of the camera target and camera position.

```
while 1
    [x,y] = ginput(1);
    if ~strcmp(get(gcf, 'SelectionType'), 'normal')
        break
    end
    ct = camtarget;
    dx = x - ct(1);
```

```
dy = y - ct(2);  
camdolly(dx,dy,ct(3),'movetarget','data')  
drawnow  
end
```

Example – Moving the Camera Through a Scene

A fly-through is an effect created by moving the camera through three dimensional space, giving the impression that you are flying along with the camera as if in an aircraft. You can fly-through of regions of a scene that may be otherwise obscured by objects in the scene or you can fly by a scene by keeping the camera focused on a particular point.

To accomplish these effects you move the camera along a particular path, the x axis for example, in a series of steps. To produce a fly-through, move both the camera position and the camera target at the same time.

The following example makes use of the fly-through effect to view the interior of an isosurface drawn within a volume defined by a vector field of wind velocities. This data representing air currents over North America.

See `coneplot` for a fixed visualization of the same data.

Summary of Techniques

This example employs a number of visualization techniques. It uses:

- Isosurfaces and `conplots` to illustrate the flow through the volume
- Lighting to illuminate the isosurface and cones in the volume
- Stream lines to define a path for the camera through the volume
- Coordinated motion of the camera position, camera target, and light together

Graphing the Volume Data

The first step is to draw the isosurface and plot the air flow using cone plots.

See `isosurface`, `isonormals`, `reducepatch`, and `coneplot` for information on using these commands.

Setting the data aspect ratio (`daspect`) to `[1, 1, 1]` before drawing the cone plot enables MATLAB to calculate the size of the cones correctly for the final view.

```
load wind
wind_speed = sqrt(u.^2 + v.^2 + w.^2);

hpatch = patch(isosurface(x,y,z,wind_speed,35));
isonormals(x,y,z,wind_speed,hpatch)
```

```

set(hpatch, FaceColor , red , EdgeColor , none );

[f vt] = reducepatch(isosurface(x,y,z,wind_speed,45),0.05);
daspect([1,1,1]);
hcone = coneplot(x,y,z,u,v,w,vt(:,1),vt(:,2),vt(:,3),2);
set(hcone, 'FaceColor', 'blue', 'EdgeColor', 'none');

```

Setting Up the View

You need to define viewing parameters to ensure the scene displays correctly.

- Selecting a perspective projection provides the perception of depth as the camera passes through the interior of the isosurface (`camproj`).
- Setting the camera view angle to a fixed value prevents MATLAB from automatically adjusting the angle to encompass the entire scene as well as zooming in to the desired amount (`camva`).

```

camproj perspective
camva(25)

```

Specifying the Light Source

Positioning the light source at the camera location and modifying the reflectance characteristics of the isosurface and cones enhances the realism of the scene.

- Creating a light source at the camera position provides a “headlight” that moves along with the camera through the isosurface interior (`camlight`).
- Setting the reflection properties of the isosurface gives the appearance of a dark interior (`AmbientStrength` set to 0.1) with highly reflective material (`SpecularStrength` and `DiffuseStrength` set to 1).
- Setting the `SpecularStrength` of the cones to 1 makes them highly reflective.

```

hlight = camlight('headlight');
set(hpatch, 'AmbientStrength', .1, ...
    'SpecularStrength', 1, ...
    'DiffuseStrength', 1);
set(hcone, 'SpecularStrength', 1);
set(gcf, 'Color', 'k')

```

Selecting a Renderer

Because this example uses lighting, MATLAB must use either `zbuffer` or, if available, OpenGL renderer settings. The OpenGL renderer is likely to be much faster displaying the animation; however, you need to use `gouraud` lighting with OpenGL, which is not as smooth as `phong` lighting, which you can use with the `zbuffer` renderer. The two choices are

```
lighting gouraud
set(gcf, 'Renderer', 'OpenGL')
```

or for `zbuffer`

```
lighting phong
set(gcf, 'Renderer', 'zbuffer')
```

Defining the Camera Path as a Stream Line

Stream lines indicate the direction of flow in the vector field. This example uses the `x`, `y`, and `z` coordinate data of a single stream line to map a path through the volume. The camera is then moved along this path. The steps include:

- Create a stream line starting at the point `x = 80`, `y = 30`, `z = 11`
- Get the `x`, `y`, and `z` coordinate data of the stream line.
- Delete the stream line (note that you could also use `stream3` to calculate the stream line data without actually drawing the stream line).

```
hsline = streamline(x,y,z,u,v,w,80,30,11);
xd = get(hsline, 'XData');
yd = get(hsline, 'YData');
zd = get(hsline, 'ZData');
delete(hsline)
```

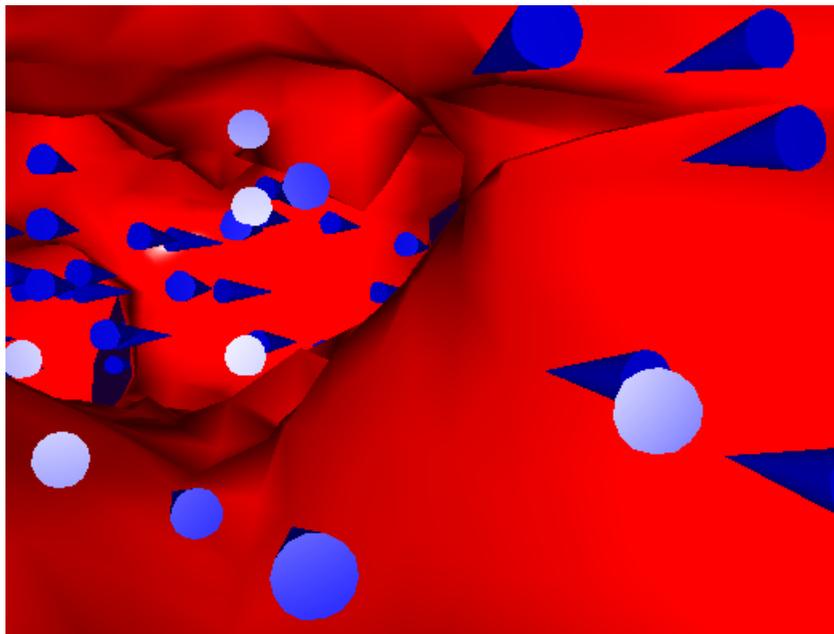
Implementing the Fly-Through

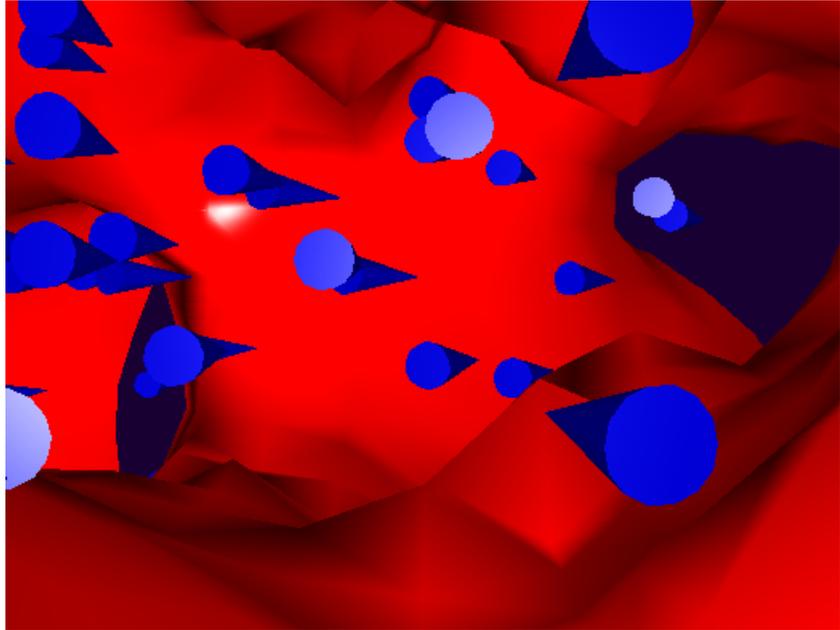
To create a fly-through, move the camera position and camera target along the same path. In this example, the camera target is placed five elements further along the `x`-axis than the camera. Also, a small value is added to the camera target `x` position to prevent the position of the camera and target from becoming the same point if the condition `xd(n) = xd(n+5)` should occur:

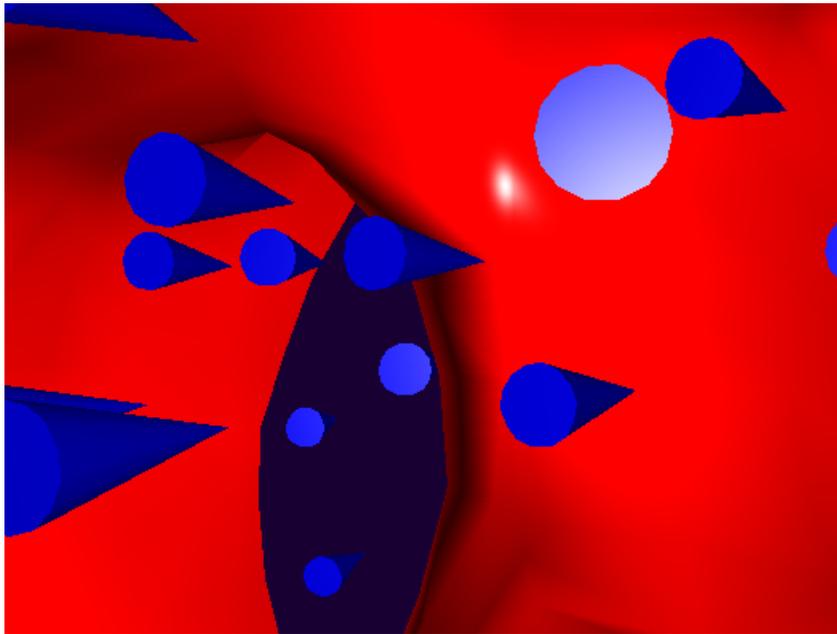
- Update the camera position and camera target so that they both move along the coordinates of the stream line.
- Move the light along with the camera.
- Call `drawnow` to display the results of each move.

```
for i=1:length(xd)-50
    campos([xd(i),yd(i),zd(i)])
    camtarget([xd(i+5)+min(xd)/100,yd(i),zd(i)])
    camlight(hlight,'headlight')
    drawnow
end
```

These snapshots illustrate the view at values of `i` equal to 10, 110, and 185.







Low-Level Camera Properties

Camera graphics is based on a group of axes properties that control the position and orientation of the camera. In general, the camera commands make it unnecessary to access these properties directly.

Property	What It Is
CameraPosition	Specifies the location of the viewpoint in axes units.
CameraPositionMode	In automatic mode, MATLAB determines the position based on the scene. In manual mode, you specify the viewpoint location.
CameraTarget	Specifies the location in the axes pointed to by the camera. Together with the CameraPosition, it defines the viewing axis.
CameraTargetMode	In automatic mode, MATLAB specifies the CameraTarget as the center of the axes plot box. In manual mode, you specify the location.
CameraUpVector	The rotation of the camera around the viewing axis is defined by a vector indicating the direction taken as up.
CameraUpVectorMode	In automatic mode, MATLAB orients the up vector along the positive y -axis for 2-D views and along the positive z -axis for 3-D views. In manual mode, you specify the direction.
CameraViewAngle	Specifies the field of view of the “lens.” If you specify a value for CameraViewAngle, MATLAB overrides stretch-to-fill behavior (see the “Understanding Axes Aspect Ratio” section of this chapter).
CameraViewAngleMode	In automatic mode, MATLAB adjusts the view angle to the smallest angle that captures the entire scene. In manual mode, you specify the angle. Setting CameraViewAngleMode to manual overrides stretch-to-fill behavior.
Projection	Selects either an orthographic or perspective projection.

Default Viewpoint Selection

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

By default, MATLAB:

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

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

Moving In and Out on the Scene

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

Moving Through a Scene

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

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

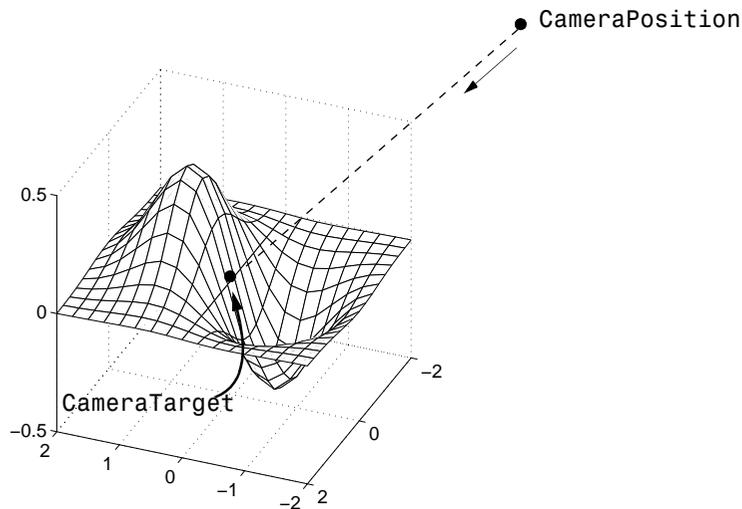
If the `Projection` is set to perspective, the amount of perspective distortion increases as the camera gets closer to the target and decreases as it gets farther away.

Example — Moving Toward or Away from the Target

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

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

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



Note that setting the `CameraViewAngleMode` to `manual` overrides MATLAB stretch-to-fill behavior and may cause an abrupt change in the aspect ratio. See

the “Understanding Axes Aspect Ratio” section of this chapter for more information on stretch-to-fill.

Making the Scene Larger or Smaller

Adjusting the `CameraViewAngle` property makes the view of the scene larger or smaller. Larger angles cause the view to encompass a larger area, thereby making the objects in the scene appear smaller. Similarly, smaller angles make the objects appear larger.

Changing `CameraViewAngle` makes the scene larger or smaller without affecting the position of the camera. This is desirable if you want to zoom in without moving the viewpoint past objects that will then no longer be in the scene (as could happen if you changed the camera position). Also, changing the `CameraViewAngle` does not affect the amount of perspective applied to the scene, as changing `CameraPosition` does when the figure `Projection` property is set to perspective.

Revolving Around the Scene

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

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

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

Rotation Without Resizing of Graphics Objects

When `CameraViewAngleMode` is `auto`, MATLAB calculates the `CameraViewAngle` so that the scene is as large as can fit in the axes position rectangle. This causes an apparent size change during rotation of the scene. To

prevent resizing during rotation, you need to set the `CameraViewAngleMode` to `manual` (which happens automatically when you specify a value for the `CameraViewAngle` property). To do this in the `orbit` function, add the statement

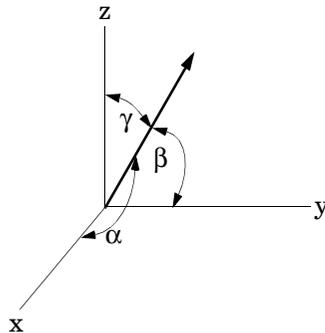
```
set(gca, 'CameraViewAngleMode', 'manual')
```

Rotation About the Viewing Axis

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

The vector defined by the `CameraUpVector` property forms one axis of the camera's coordinate system. Internally, MATLAB determines the actual orientation of the camera up vector by projecting the specified vector onto the plane that is normal to the camera direction (i.e., the viewing axis). This simplifies the specification of the `CameraUpVector` property 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.

```
XComponent = cos( $\alpha \times (\text{pi} \div 180)$ );  
YComponent = cos( $\beta \times (\text{pi} \div 180)$ );  
ZComponent = cos( $\gamma \times (\text{pi} \div 180)$ );
```

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

Example — Calculating a Camera Up Vector

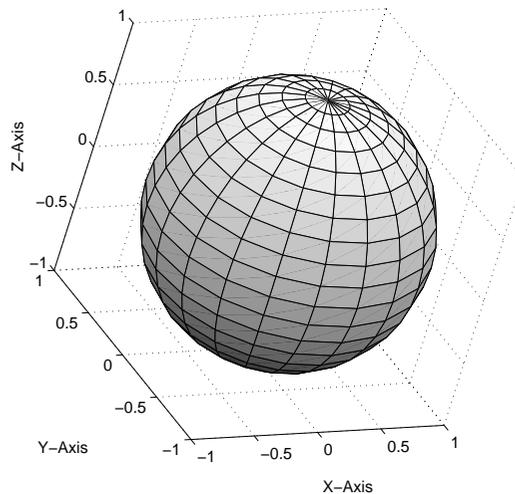
To specify an up vector that makes an angle of 30° with the z -axis and lies in the y - z plane, use the expression

```
upvec = [cos( $90 \times (\text{pi}/180)$ ), cos( $60 \times (\text{pi}/180)$ ), cos( $30 \times (\text{pi}/180)$ )];
```

and then set the CameraUpVector property.

```
set(gca, 'CameraUpVector', upvec)
```

Drawing a sphere with this orientation produces



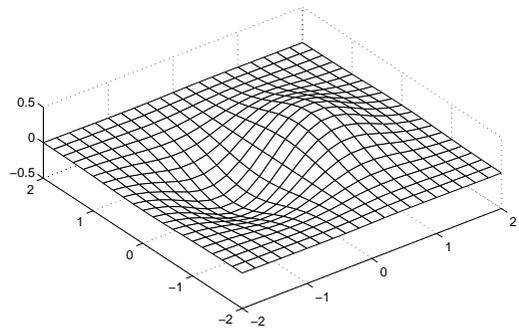
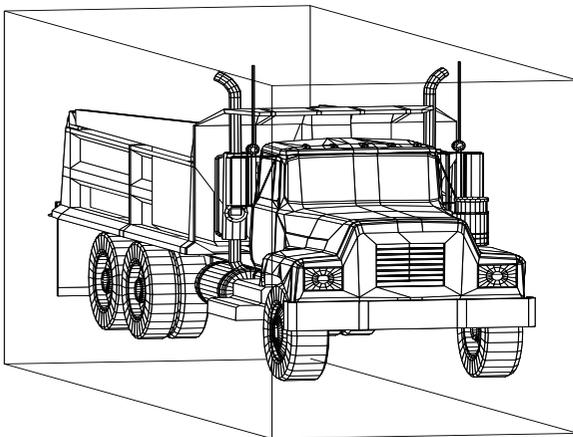
View Projection Types

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 frustrum of a pyramid (a pyramid whose apex has been cut off parallel to the base). Distance causes foreshortening; objects further from the camera appear smaller. This projection type is useful when you want to display realistic views of real objects.

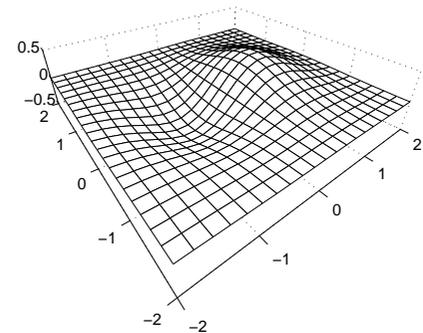
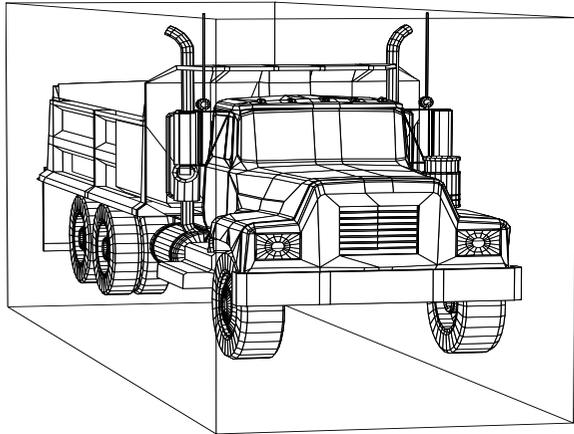
By default, MATLAB displays objects using orthographic projection. You can set the projection type using the `camproj` command.

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



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

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



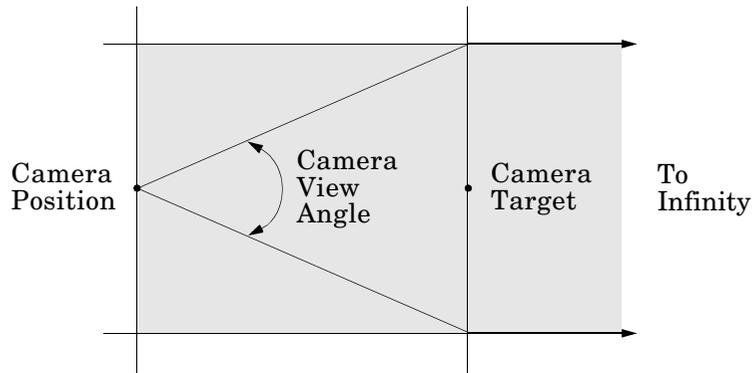
Projection Types and Camera Location

By default, MATLAB adjusts the `CameraPosition`, `CameraTarget`, and `CameraViewAngle` properties to point the camera at the center of the scene and to include all graphics objects in the axes. If you position the camera so that there are graphics objects behind the camera, the scene displayed can be affected by both the axes `Projection` property and the figure `Renderer`

property. This table summarizes the interactions between projection type and rendering method.

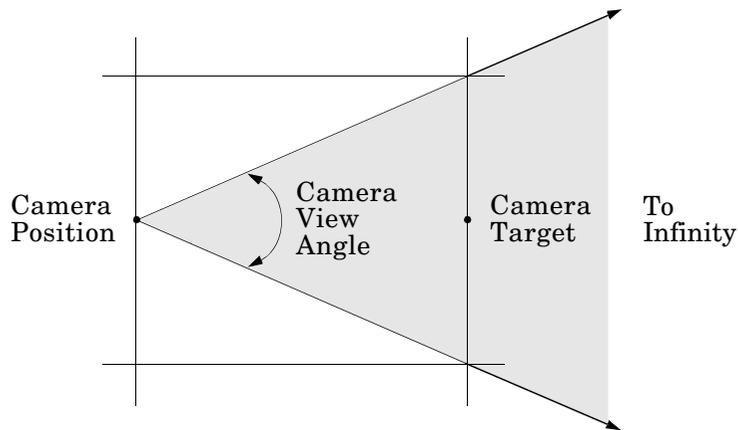
	Orthographic	Perspective
Z-buffer	CameraViewAngle determines extent of scene at CameraTarget	CameraViewAngle determines extent of scene from CameraPosition to infinity
Painters	All objects display regardless of CameraPosition	Not recommended if graphics objects are behind the CameraPosition

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



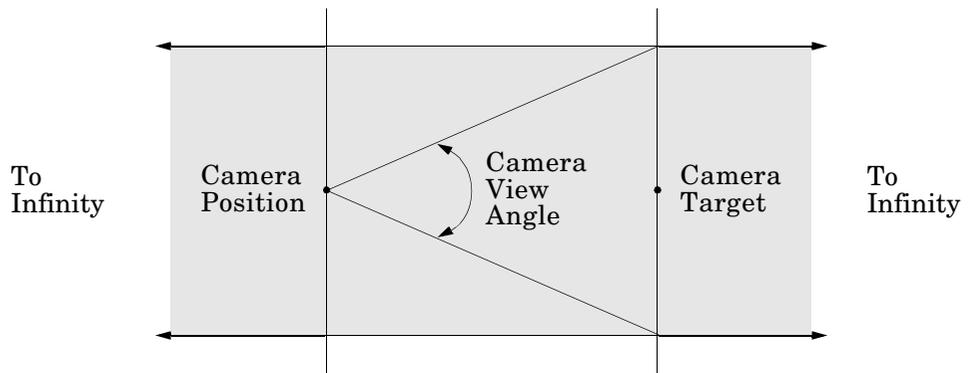
Orthographic projection and Z-buffer renderer

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.



Orthographic projection and painters renderer

Printing 3-D Scenes

The same effects described in the previous section occur in hardcopy output. However, because of the differences in the process of rendering to the screen

and to a printing format, MATLAB may render in Z-buffer and generate printed output in painters. You may need to specify Z-buffer printing explicitly to obtain the results displayed on the screen (use the `-zbuffer` option with the `print` command).

Additional Information

See the “Basic Printing and Exporting” chapter and the “Selecting a Renderer” section in the “Figure Properties” chapter for information on printing and rendering methods.

Understanding Axes 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 or size of the plotted data, and then draws the axes to fit the space available for display. Axes aspect ratio properties control how MATLAB performs the scaling required to create a graph.

This section discusses MATLAB 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 possibly the aspect ratio (the ratio of width to height) 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 of the figure window, or you may want graphs always to be a particular size on a printed page.

Specifying Axis Scaling

The `axis` command enables you to adjust the scaling of graphs. By default, MATLAB finds the maxima and minima of the plotted data and chooses appropriate axes ranges. You can override the defaults by setting axis limits.

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

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

- `axis auto` returns the axis scaling to its default, automatic mode. `v = axis` saves the scaling of the axes of the current plot in vector `v`. For subsequent graphics commands to have these same axis limits, follow them with `axis(v)`.

- `axis manual` freezes the scaling at the current limits. If you then set `hold on`, subsequent plots use the current limits. Specifying values for axis limits also sets axis scaling to manual.
- `axis tight` sets the axis limits to the range of the data.
- `axis ij` places MATLAB into its “matrix” axes mode. The coordinate system origin is at the upper-left corner. The *i*-axis is vertical and is numbered from top to bottom. The *j*-axis is horizontal and is numbered from left to right.
- `axis xy` places MATLAB into its default Cartesian axes mode. The coordinate system origin is at the lower-left corner. The *x*-axis is horizontal and is numbered from left to right. The *y*-axis is vertical and is numbered from bottom to top.

Specifying Aspect Ratio

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

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

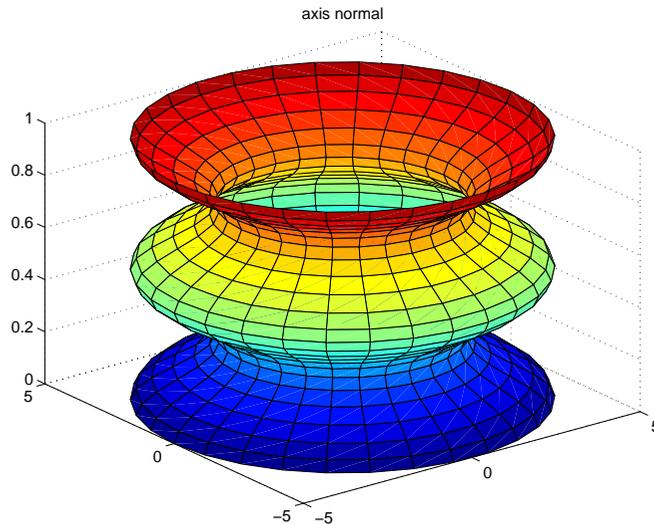
The `axis` command works by manipulating axes graphics object properties.

Example – axis Command Options

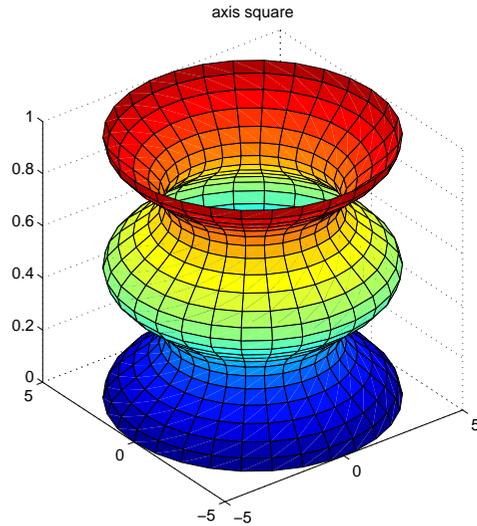
The following three pictures illustrate the effects of three axis options on a cylindrical surface created with the statements

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

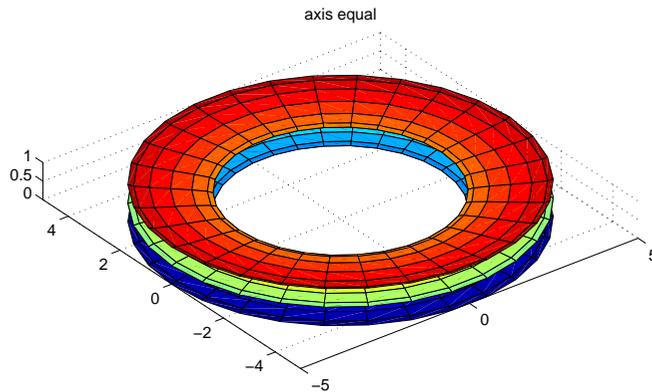
`axis normal` is the default behavior. MATLAB automatically sets the axis limits to span the data range along each axis and stretches the plot to fit the figure window.



`axis square` creates an 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.



Additional Commands for Setting Aspect Ratio

You can control the aspect ratio of your graph in three ways:

- Specifying the relative scales of the x, y, and z axes (data aspect ratio).
- Specifying the shape of the space defined by the axes (plot box aspect ratio).
- Specifying the axis limits.

The following commands enable you to set these values.

Command	Purpose
<code>daspect</code>	Set or query the data aspect ratio
<code>pbaspect</code>	Set or query the plot box aspect ratio
<code>xlim</code>	Set or query <i>x</i> -axis limits
<code>ylim</code>	Set or query <i>y</i> -axis limits
<code>zlim</code>	Set or query <i>z</i> -axis limits

See the “Axes Aspect Ratio Properties” section in this chapter for a list of the axes properties that control aspect ratio.

Axes Aspect Ratio Properties

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

Property	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 auto, 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 on to see the box). Specifying a value for <code>PlotBoxAspectRatio</code> overrides stretch-to-fill behavior.
<code>PlotBoxAspectRatioMode</code>	In auto, 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: <i>[left offset, bottom offset, width, height]</i> .
<code>XLim, YLim, ZLim</code>	Sets the minimum and maximum limits of the respective axes.
<code>XLimMode, YLimMode, ZLimMode</code>	In auto, MATLAB selects the axis limits.

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

Much of the data visualized with MATLAB is either:

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

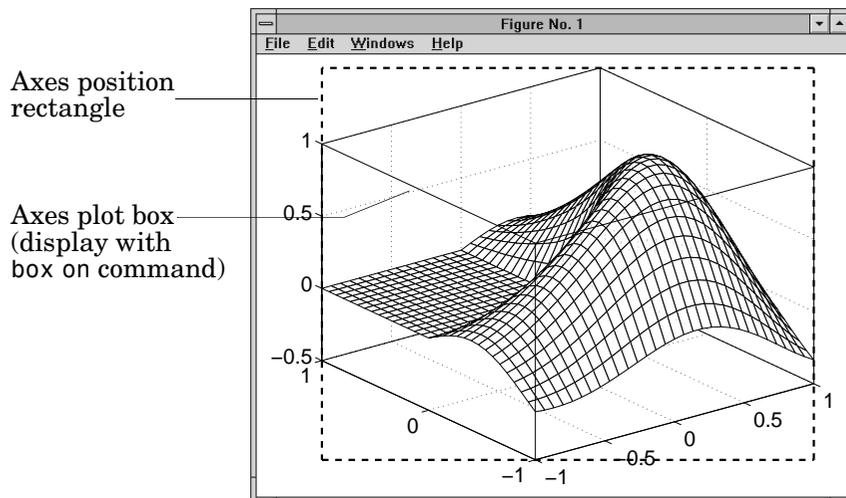
In the first case, it is generally desirable to select axis limits that provide good resolution in each axis direction and to fill the available space. Real-world objects, on the other hand, need to be represented accurately in proportion, regardless of the angle of view.

Default Aspect Ratio Selection

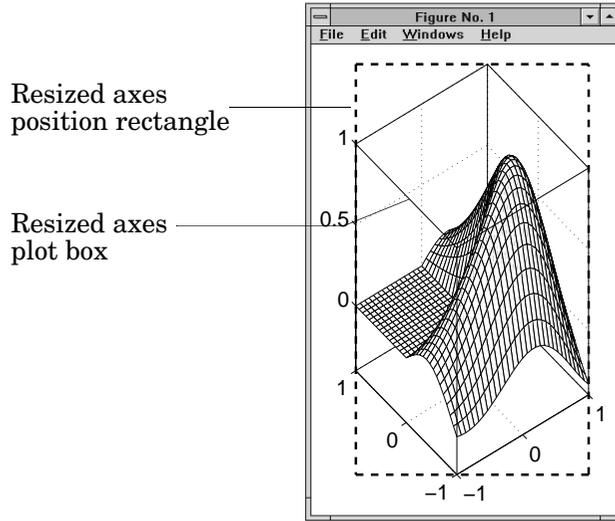
There are two key elements to MATLAB default behavior — normalizing the axes size to the window size and stretch-to-fill.

The axes `Position` property specifies the location and dimensions of the axes. The third and fourth elements of the `Position` vector (width and height) define a rectangle in which MATLAB draws the axes (indicated by the dotted line in the following pictures). MATLAB stretches the axes to fill this rectangle.

The default value for the axes `Units` property is normalized to the parent figure dimensions. This means the shape of the figure window determines the shape of the position rectangle. As you change the size of the window, MATLAB reshapes the position rectangle to fit it.



The view is the 2-D projection of the plot box onto the screen.

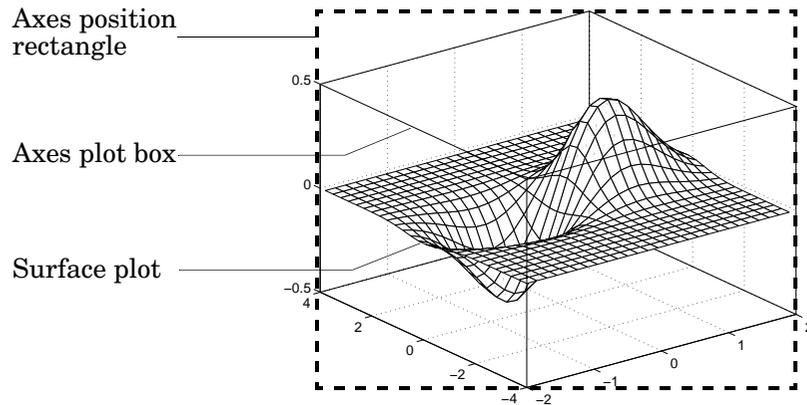


As you can see, reshaping the axes to fit into the figure window can change the aspect ratio of the graph. MATLAB applies stretch-to-fill so the axes fill the position rectangle and in the process may distort the shape. This is generally desirable for graphs of numeric data, but not for displaying objects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

Example — MATLAB Defaults

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

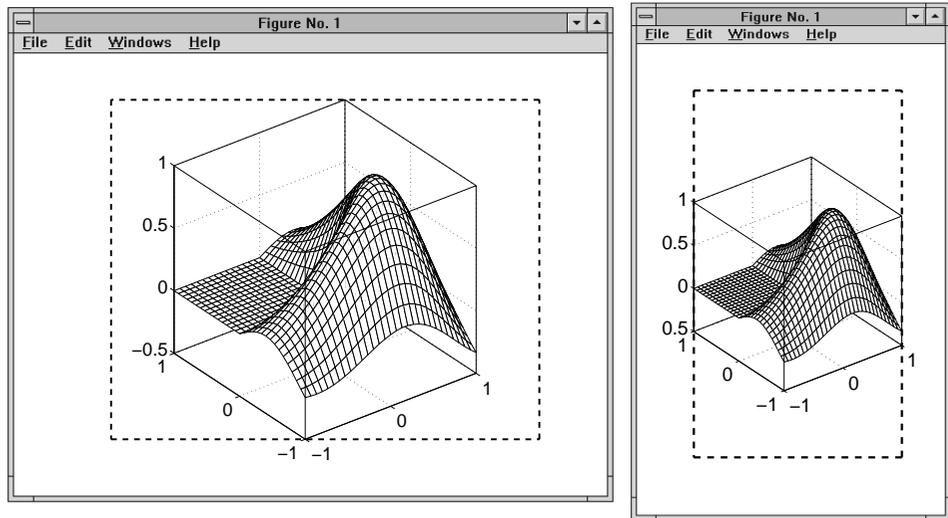
Overriding Stretch-to-Fill

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.

Effects of Setting Aspect Ratio Properties

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

Data Aspect Ratio

The DataAspectRatio property controls the ratio of the axis scales. For a mesh plot of the function, $z = xe^{(-x^2 - y^2)}$ evaluated over the range $-2 \leq x \leq 2$, $-4 \leq y \leq 4$

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

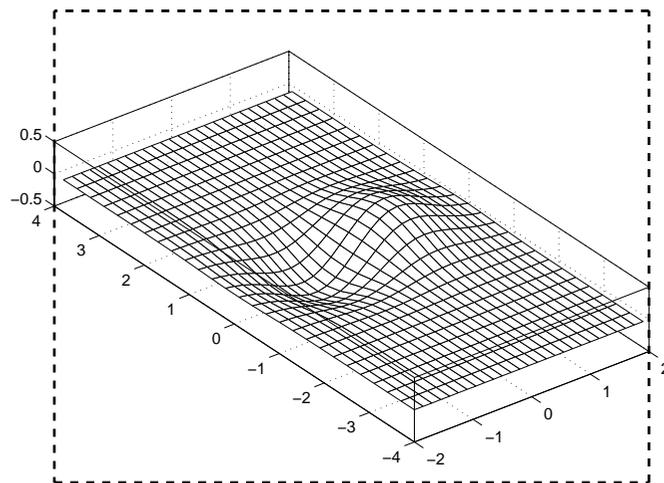
the values are

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

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

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

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



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

Plot Box Aspect Ratio

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

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

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

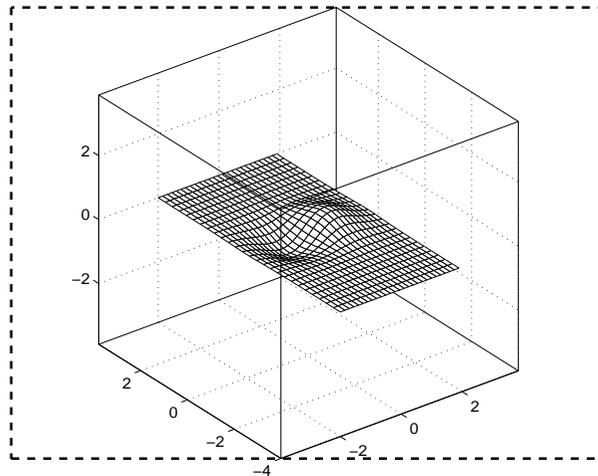
The `PlotBoxAspectRatio` property controls the shape of the axes plot box. MATLAB sets this property to `[1 1 1]` by default and adjusts the `DataAspectRatio` property so that graphs fill the plot box if stretching is on, or until reaching a constraint if stretch-to-fill has been overridden.

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

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

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

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



Adjusting Axis Limits

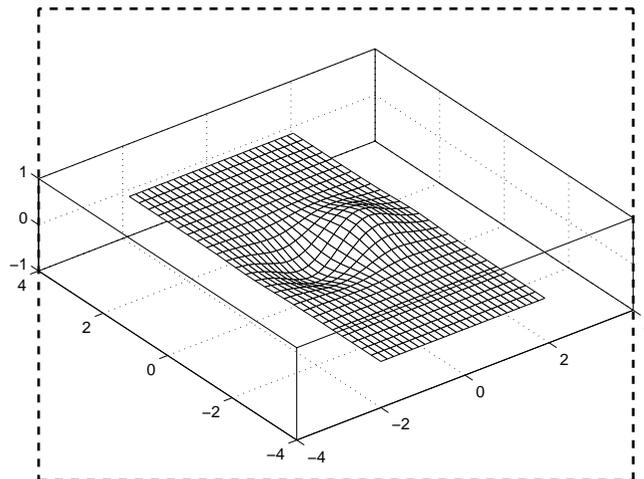
MATLAB enables you to set the axis limits to whichever values you want. However, specifying a value for `DataAspectRatio`, `PlotBoxAspectRatio`, and the axis limits, overconstrains the axes definition. For example, it is not possible for MATLAB to draw the axes if you set these values.

```
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.

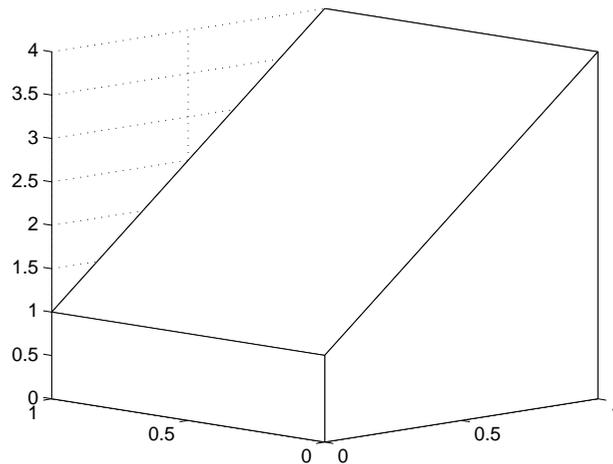


Example – Displaying Real Objects

If you want to display an object so that it looks realistic, you need to change MATLAB 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
```

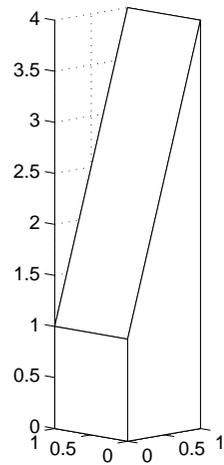
```
patch('Vertices',vertex_list,'Faces',vertex_connection,...
      'FaceColor','w','EdgeColor','k')
view(3)
```



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

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

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



Lighting as a Visualization Tool

Lighting Overview (p. 12-2)	Contains links to examples throughout the graphics documentation that illustrate the use of lighting.
Lighting Commands (p. 12-3)	Commands for creating lighting effects.
Light Objects (p. 12-4)	Creation and properties of light objects.
Adding Lights to a Scene (p. 12-5)	Examples of how to position lights and set properties.
Properties That Affect Lighting (p. 12-8)	Properties of axes, patches, and surfaces that affect lights.
Selecting a Lighting Method (p. 12-10)	Illustration of various lighting methods showing which to use.
Reflectance Characteristics of Graphics Objects (p. 12-11)	Catalog illustrating various lighting characteristics.

Lighting Overview

Lighting is a technique for adding realism to a graphical scene. It does this by simulating the highlights and dark areas that occur on objects under natural lighting (e.g., the directional light that comes from the sun). To create lighting effects, MATLAB defines a graphics object called a light. MATLAB applies lighting to surface and patch objects.

Lighting Examples

These examples illustrate the use of lighting in a visualization context.

- Tracing a stream line through a volume – sets properties of surfaces, patches, and lights. See "Example – "Creating a Fly-Through" in the "Defining the View Chapter".
- Using slice planes and cone plots – sets lighting characteristics of objects in a scene independently to achieve a desired result. See the `coneplot` function.
- Lighting multiple slice planes independently to visualize fluid flow. See the "Example - Slicing Fluid Flow Data" section in the "Volume Visualization Techniques" chapter.
- Combining single-color lit surfaces with interpolated coloring. See the "Example - Visualizing MRI Data" section in the "Volume Visualization Techniques" chapter.
- Employing lighting to reveal surface shape. The fluid flow isosurface example and the surface plot of the `sinc` function examples illustrate this technique. See the "Example - Isosurfaces in Fluid Flow Data" section in the "Volume Visualization Techniques" chapter and the "Visualizing Functions of Two Variables" section in the "Creating 3-D Graphs" chapter.

Lighting Commands

MATLAB provides commands that enable you to position light sources and adjust the characteristics of lit objects. These commands include the following.

Command	Purpose
<code>camlight</code>	Create or move a light with respect to the camera position
<code>lightangle</code>	Create or position a light in spherical coordinates
<code>light</code>	Create a light object
<code>lighting</code>	Select a lighting method
<code>material</code>	Set the reflectance properties of lit objects

You may find it useful to set light or lit-object properties directly to achieve specific results. In addition to the material in this topic area, you can explore the following lighting examples as an introduction to lighting for visualization.

Light Objects

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

- `Color` – 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 `Color` property determines the color of the directional light from the light source. The color of an object in a scene is determined by the color of the object and the light source.

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

The `Position` property specifies the location of the light source in axes data units. In the case of an light source at infinity, `Position` specifies the direction to the light source.

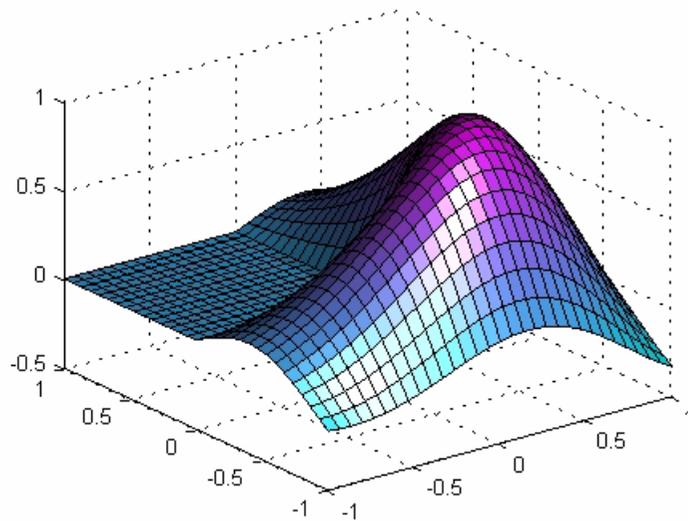
Lights affect surface and patch objects that are in the same axes as the light. These objects have a number of properties that alter the way they look when illuminated by lights.

Adding Lights to a Scene

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

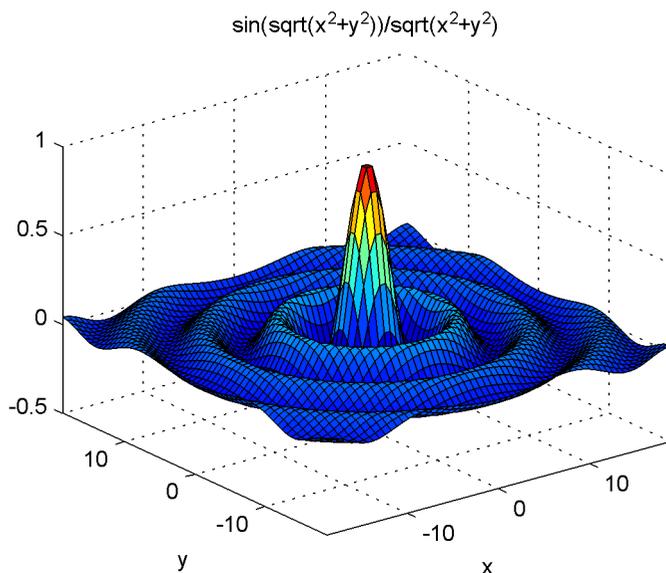


Illuminating Mathematical Functions

Lighting can enhance surface graphs of mathematical functions. For example, use the `ezsurf` command to evaluate the expression

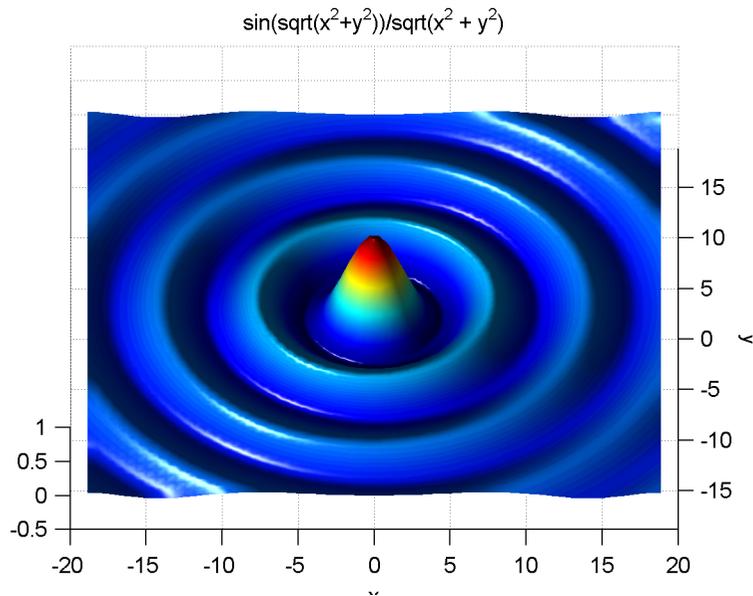
$\sin(\sqrt{x^2 + y^2}) / \sqrt{x^2 + y^2}$ over the region -6π to 6π .

```
ezsurf('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)',[-6*pi,6*pi])
```



Now add lighting using the `lightangle` command, which accepts the light position in terms of azimuth and elevation.

```
view(0,75)
shading interp
lightangle(-45,30)
set(gcf,'Renderer','zbuffer')
set(findobj(gca,'type','surface'),...
    'FaceLighting','phong',...
    'AmbientStrength',.3,'DiffuseStrength',.8,...
    'SpecularStrength',.9,'SpecularExponent',25,...
    'BackFaceLighting','unlit')
```



After obtaining surface object's handle using `findobj`, you can set properties that affect how the light reflects from the surface. See the "Properties That Affect Lighting" section of this chapter for more detailed descriptions of these properties.

Properties That Affect Lighting

You cannot see light objects themselves, but you can see their effects on any patch and surface objects present in the axes containing the light. A number of functions create these objects, including `surf`, `mesh`, `pcolor`, `fill`, and `fill3` as well as the surface and patch functions.

You control lighting effects by setting various axes, light, patch, and surface object properties. All properties have default values that generally produce desirable results. However, you can achieve the specific effect you want by adjusting the values of these properties.

Property	Effect
<code>AmbientLightColor</code>	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.
<code>AmbientStrength</code>	A patch and surface property that determines the intensity of the ambient component of the light reflected from the object.
<code>DiffuseStrength</code>	A patch and surface property that determines the intensity of the diffuse component of the light reflected from the object.
<code>SpecularStrength</code>	A patch and surface property that determines the intensity of the specular component of the light reflected from the object.
<code>SpecularExponent</code>	A patch and surface property that determines the size of the specular highlight.
<code>SpecularColorReflectance</code>	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.
<code>FaceLighting</code>	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.

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

See a description of all axes, surface, and patch object properties.

Selecting a Lighting Method

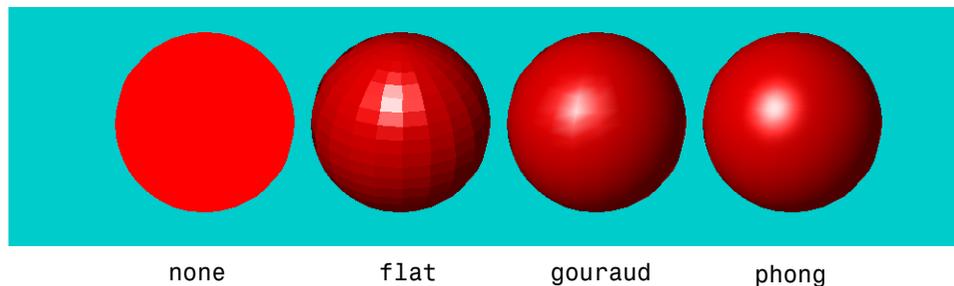
When you add lights to an axes, MATLAB determines the effects these lights have on the patch and surface objects that are displayed in that axes. There are different methods used to calculate the face and edge coloring of lit objects, and the one you select depends on the results you want to obtain.

Face and Edge Lighting Methods

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

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

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



The `lighting` command (as opposed to the `light` function) provides a convenient way to set the lighting method.

Reflectance Characteristics of Graphics Objects

You can modify the reflectance characteristics of patch and surface objects and thereby change the way they look when lights are applied to the scene. The characteristics discussed in this section include:

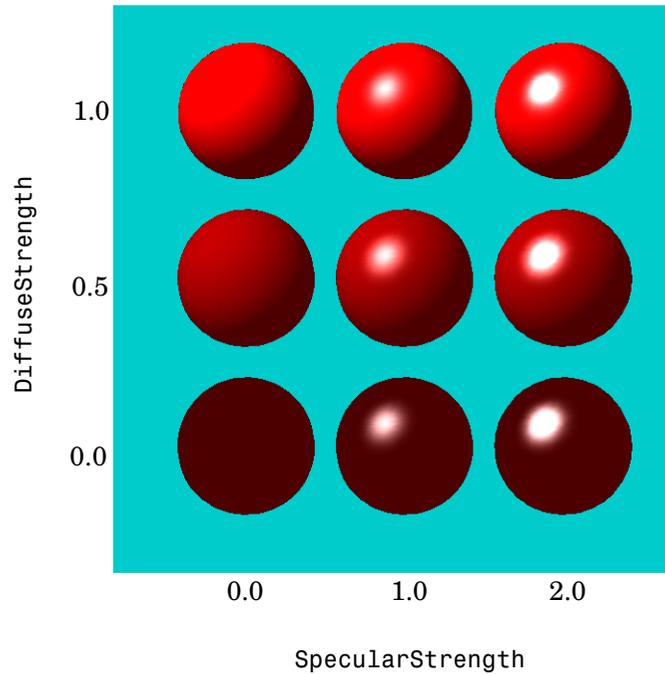
- Specular and diffuse reflection
- Ambient light
- Specular exponent
- Specular color reflectance
- Backface lighting

It is likely you will adjust these characteristics in combination to produce particular results.

Also see the `material` command for a convenient way to produce certain lighting effects.

Specular and Diffuse Reflection

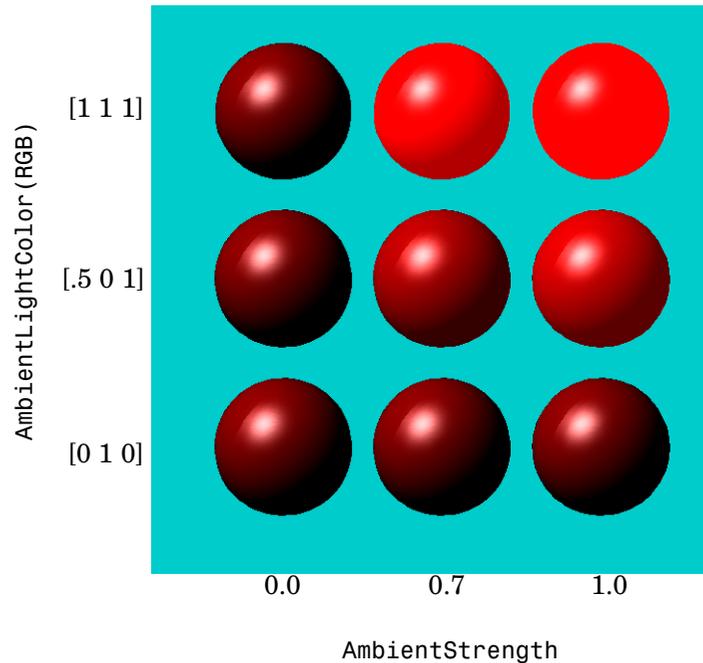
You can control the amount of specular and diffuse reflection from the surface of an object by setting the `SpecularStrength` and `DiffuseStrength` properties. This picture illustrates various settings.



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

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

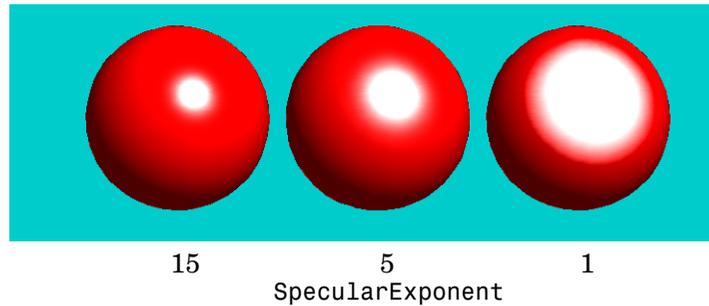


The green [0 1 0] ambient light does not affect the scene because there is no red component in green light. However, the color defined by the RGB values [0.5 0 1] does have a red component, so it contributes to the light on the sphere (but less than the white [1 1 1] ambient light).

Specular Exponent

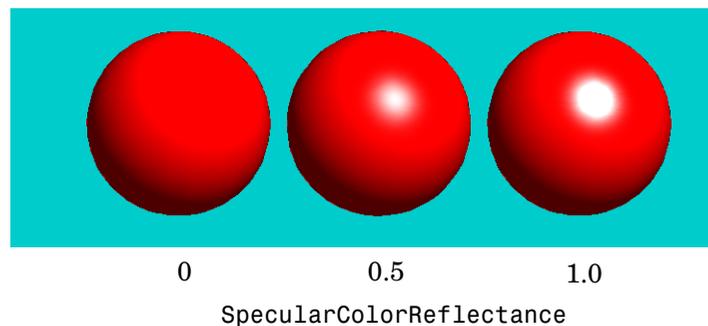
The size of the specular highlight spot depends on the value of the patch and surface object's `SpecularExponent` property. Typical values for this property range from 1 to 500, with normal objects having values in the range 5 to 20.

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



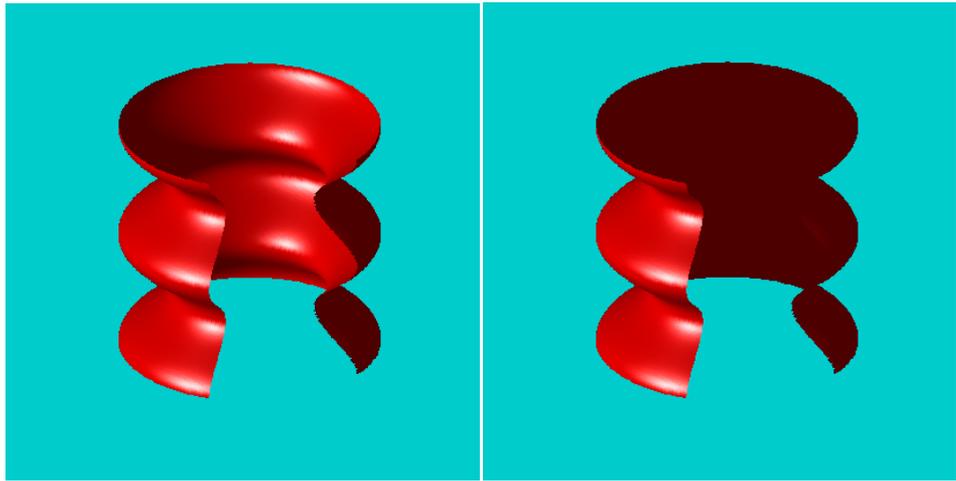
Specular Color Reflectance

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



Back Face Lighting

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



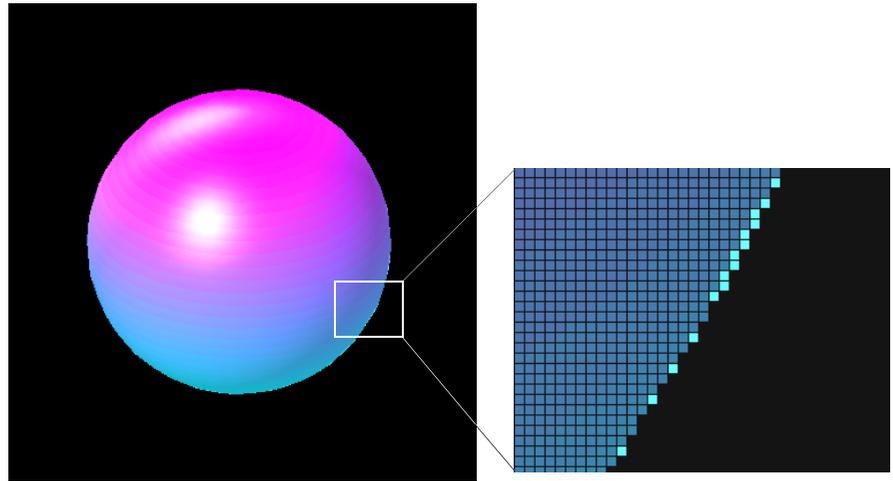
BackFaceLighting = reverselit

BackFaceLighting = unlit

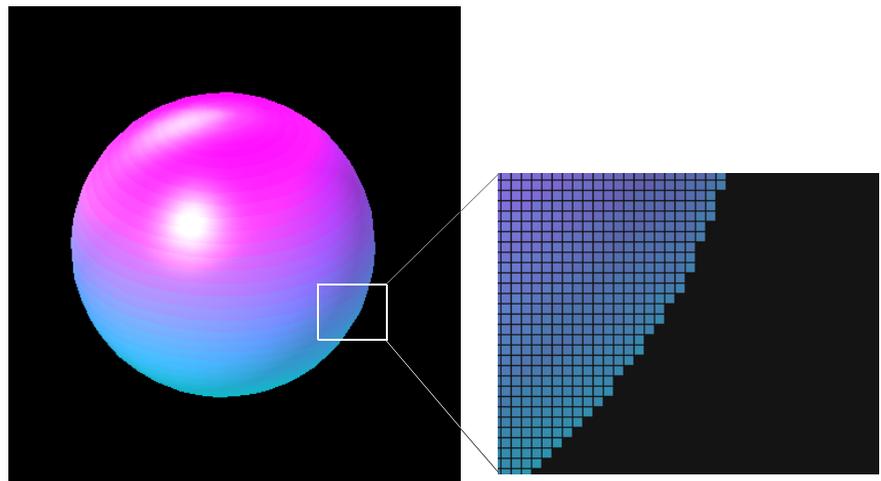
The default value for `BackFaceLighting` is `reverselit`. This setting reverses the direction of the vertex normals that face away from the camera, causing the interior surface to reflect light towards the camera. Setting `BackFaceLighting` to `unlit` disables lighting on faces with normals that point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

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

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



`BackFaceLighting = reverselit`



`BackFaceLighting = lit`

Positioning Lights in Data Space

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

```
vert =
    1    1    1
    1    2    1
    2    2    1
    2    1    1
    1    1    2
    1    2    2
    2    2    2
    2    1    2

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

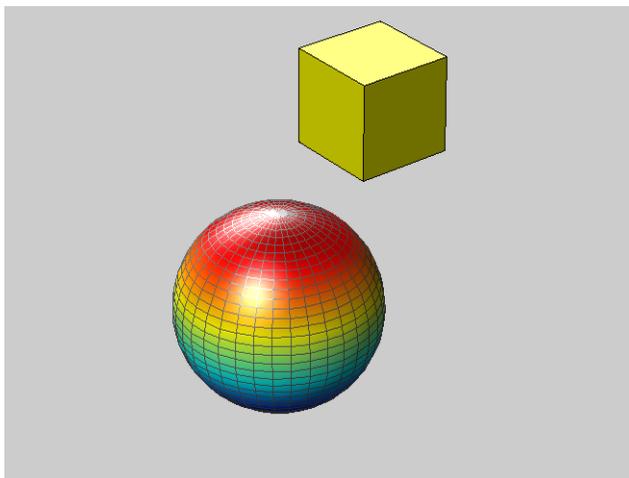
```
sphere(36);
h = findobj('Type','surface');
set(h,'FaceLighting','phong',...
    'FaceColor','interp',...
    'EdgeColor',[.4 .4 .4],...
    'BackFaceLighting','lit')
hold on
patch('faces',fac,'vertices',vert,'FaceColor','y');
light('Position',[1 3 2]);
light('Position',[-3 -1 3]);
material shiny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

Transparency

Making Objects Transparent (p. 13-2)	Overview of the object properties that specify transparency.
Specifying a Single Transparency Value (p. 13-5)	How to specify a transparency value that applies to all the faces of a graphics object.
Mapping Data to Transparency (p. 13-8)	How to use transparency as another dimension for visualizing data.
Selecting an Alphamap (p. 13-11)	Characteristics of various alphamaps and illustrations of the effects they produce.

Making Objects Transparent

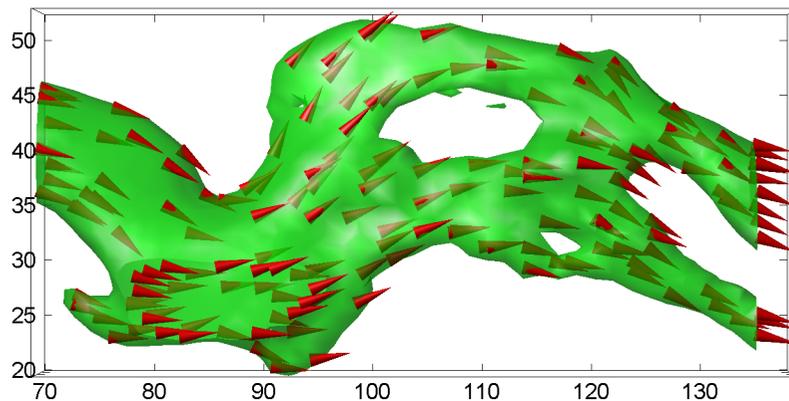
Making graphics objects semi-transparent is a useful technique in 3-D visualization to make it possible to see an object, while at the same time, see what information the object would obscure if it was completely opaque. You can also use transparency as another dimension for displaying data, much the way color is used in MATLAB.

The transparency of a graphics object determines the degree to which you can see through the object. You can specify a continuous range of transparency varying from completely transparent (i.e., invisible) to completely opaque (i.e., no transparency).

Objects that support transparency are:

- Image
- Patch
- Surface

The following picture illustrates the effect of transparency. The green isosurface (patch object) reveals the coneplot that lies in the interior.



Note You must have OpenGL available on your system to use transparency. MATLAB automatically uses OpenGL if it is available. See the figure `RendererMode` property for more information.

Specifying Transparency

Transparency values, which range from [0 1], are referred to as *alpha* values. An alpha value of 0 means completely transparent (i.e., invisible); an alpha value of 1 means completely opaque (i.e., no transparency).

MATLAB treats transparency in a way that is analogous to how it treats color for the respective objects:

- Patches and surfaces can define a single face and edge alpha value or use flat or interpolated transparency based on values in the figure's alphamap.
- Images, patches, and surfaces can define alpha data that is used as indices into the alphamap or directly as alpha values.
- Axes define alpha limits that control the mapping of object data to alpha values.
- Figures contain alphamaps, which are m-by-1 arrays of alpha values.

See the following sections for more information on color:

- “Specifying Patch Coloring” in the “Creating 3-D Models with Patches” chapter.
- “Coloring Mesh and Surface Plots” in the “Creating 3-D Graphs” chapter.

Transparency Properties

The following table summarizes the object properties that control transparency.

Properties	Purpose
AlphaData	The transparency data for image and surface objects
AlphaDataMapping	The transparency data mapping method
FaceAlpha	Transparency of the faces (patch and surface only)
EdgeAlpha	Transparency of the edges (patch and surface only)
FaceVertexAlphaData	Patch only alpha data property
ALim	Alpha axis limits
ALimMode	Alpha axis limits mode
Alphamap	Figure alphamap

Transparency Functions

There are three functions that simplify the process of setting alpha properties.

Function	Purpose
alpha	Set or query transparency properties for objects in current axes
alphamap	Specify the figure alphamap
alim	Set or query the axes alpha limits

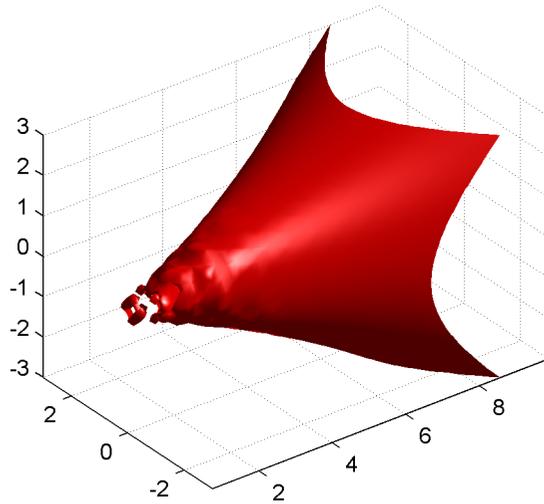
Specifying a Single Transparency Value

Specifying a single transparency value for graphics objects is useful when you want to reveal structure that is obscured with opaque objects. For patches and surfaces, use the `FaceAlpha` and `EdgeAlpha` properties to specify the transparency of faces and edges. The following example illustrates this.

Example – Transparent Isosurface

This example uses the `flow` function to generate data for the speed profile of a submerged jet within a infinite tank. One way to visualize this data is by creating an isosurface illustrating where the rate of flow is equal to a specified value.

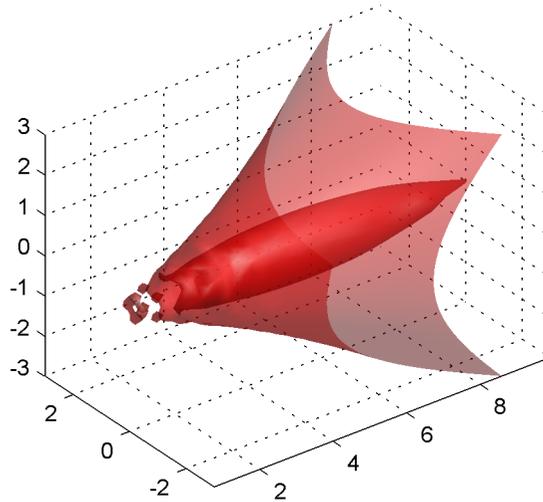
```
[x y z v] = flow;  
p = patch(isosurface(x,y,z,v,-3));  
isonormals(x,y,z,v,p);  
set(p,'facecolor','red','edgecolor','none');  
daspect([1 1 1]);  
view(3); axis tight; grid on;  
camlight; lighting gouraud;
```



Adding transparency to the isosurface reveals that there is greater complexity in the fluid flow than is visible using the opaque surface. The statement

```
alpha(.5)
```

sets the FaceAlpha value for the isosurface face to .5.



Setting a Single Transparency Value for Images

For images, the statement

```
alpha(.5)
```

sets the `AlphaData` to `.5`. When the `AlphaDataMapping` property is set to `none` (the default), setting `AlphaData` on an image causes the entire image to render with the specified alpha value.

Mapping Data to Transparency

Alpha data is analogous to color data (e.g., the `CData` property of surfaces). When you create a surface, MATLAB maps each element in the color data array to a color in the colormap. Similarly, each element in the alpha data maps to a transparency value in the alphamap.

Specify surface and image alpha data with the `AlphaData` property. For patch objects, use the `FaceVertexAlphaData` property.

You can control how MATLAB interprets alpha data with the following properties:

- `FaceAlpha` and `EdgeAlpha` – enable you to select flat or interpolated transparency rendering. If set to a single transparency value, MATLAB applies this value to all faces or edges and does not use the alpha data.
- `AlphaDataMapping` and `ALim` – determine how MATLAB maps the alpha data to the alphamap. By default, MATLAB scales the alpha data to be within the range [0 1].
- `Alphamap` – contains the actual transparency values to which the data is to be mapped.

Note that there are differences between the default values of equivalent color and alpha properties because, in contrast to color, transparency is not displayed by default. The following table highlights these differences.

Color Property	Default	Alpha Property	Default
<code>FaceColor</code>	flat	<code>FaceAlpha</code>	1 (opaque)
<code>CData</code>	equal to <code>ZData</code>	<code>AlphaData</code> and <code>FaceVertexAlphaData</code>	1 (scalar)

By default, objects have single-valued alpha data. Therefore you cannot specify flat or interp `FaceAlpha` or `EdgeAlpha` without first setting `AlphaData` to an array of the appropriate size.

The sections that follow illustrate how to use these properties to display object data as degrees of transparency.

Size of the Alpha Data Array

In order to use nonscalar alpha data, you need to specify the alpha data as an array equal in size to the:

- CData of images and surfaces
- The number of faces (flat) or the number of vertices (interpolated) defined in the FaceVertexAlphaData property of patches

Once you have specified an alpha data array of the proper size, you can select the face and edge rendering you want to use. Flat uses one transparency value per face, while interpolated performs bilinear interpolation of the values at each vertex.

Mapping Alpha Data to the Alphamap

You can control how MATLAB maps the alpha data to the alphamap using the AlphaDataMapping property. There are three possible mappings:

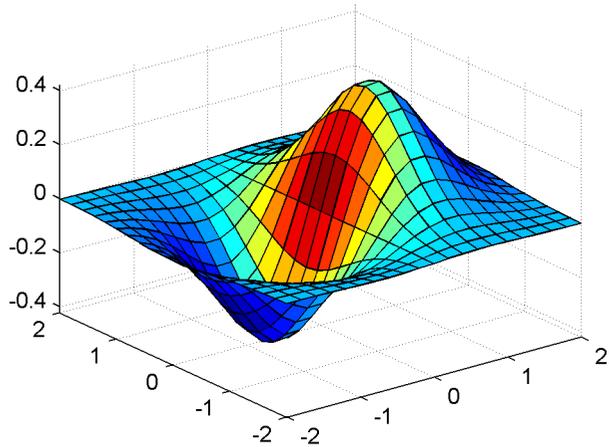
- none – Interpret the values in alpha data as transparency values (data values must be between 0 and 1, or will be clamped to 0 or 1). This is the default mapping.
- scaled – Transform the alpha data to span the portion of the alphamap indicated by the axes ALim property, linearly mapping data values to alpha values. This is the same way color data is mapped to the colormap.
- direct – Use the alpha data directly as indices into the figure alphamap.

By default, objects have scalar alpha data (AlphaData and FaceVertexAlphaData) set to the value 1.

Example: Mapping Data to Color or Transparency

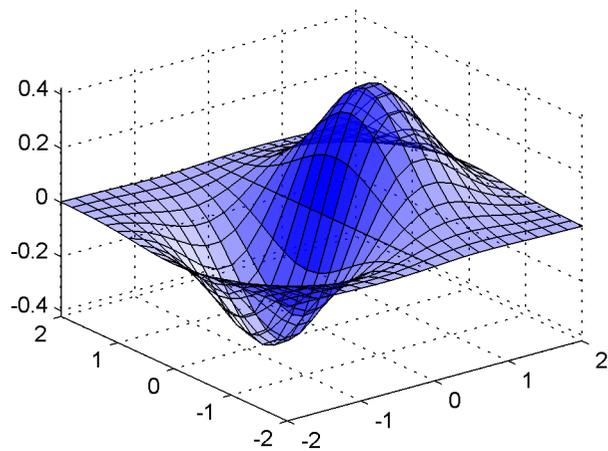
This example displays a surface plot of a functions of two variables. The color is mapped to the gradient of the z data.

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



You can map transparency to the gradient of z in a similar way.

```
surf(x,y,z,'FaceAlpha','flat',...  
      'AlphaDataMapping','scaled',...  
      'AlphaData',gradient(z),...  
      'FaceColor','blue');  
axis tight
```

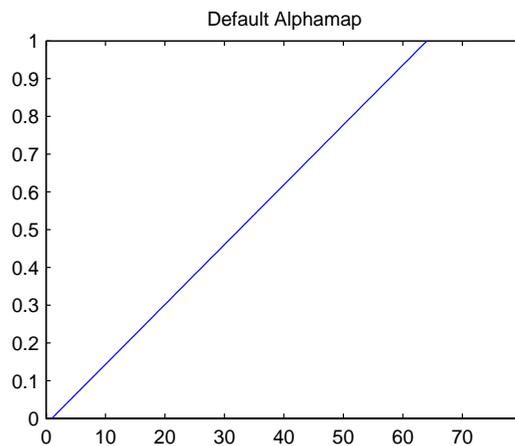


Selecting an Alphamap

An alphamap is simply an array of values ranging from 0 to 1. The size of the array can be either m-by-1 or 1-by-m.

The default alphamap contains 64 values ranging linearly from 0 to 1, as you can see in the following plot.

```
plot(get(gcf, 'Alphamap'))
```

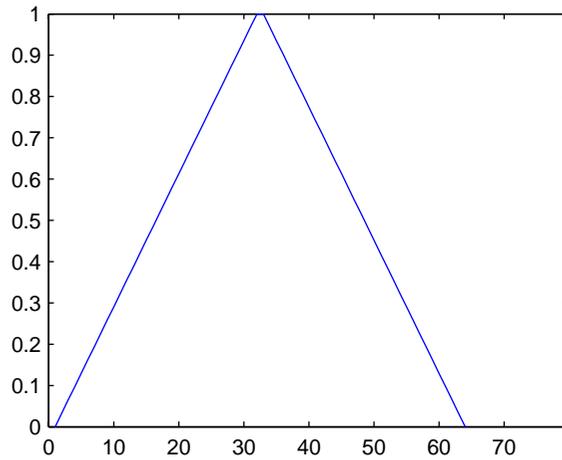


This alphamap displays the lowest alpha data values as completely transparent and the highest alpha data values as opaque.

The alphamap function creates some useful predefined alphamaps and also enables you to modify existing maps. For example,

```
plot(alphamap('vup'))
```

produces the following alphamap.

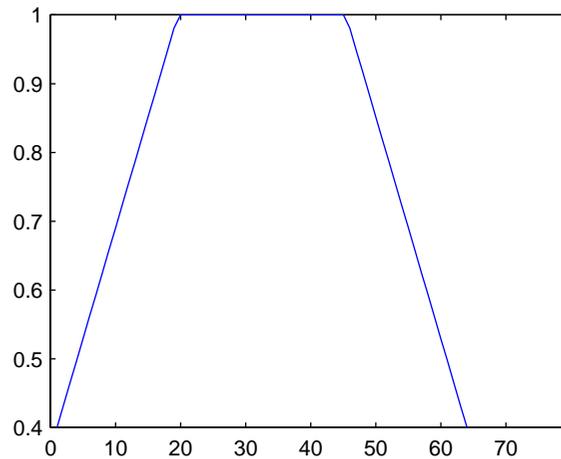


You can shift the values using the `increase` or `decrease` options. For example,

```
alphamap('increase', .4)
```

adds the value `.4` to all values in the current figure's `alphamap`. Replotting the `'vup'` `alphamap` illustrates the change. Note how the values are clamped to the range `[0 1]`.

```
plot(get(gcf, 'Alphamap'))
```



Example: Modifying the Alphamap

This example uses slice planes to examine volume data. The slice planes use the color data for alpha data and employ a rampdown alphamap (the values range from 1 to 0):

- 1 Create the volume data by evaluating a function of three variables.

```
[x,y,z] = meshgrid(-1.25:.1:-.25,-2:.2:2,-2:.1:2);
v = x.*exp(-x.^2-y.^2-z.^2);
```

- 2 Create the slice planes, set the alpha data equal to the color data, and specify interpolated FaceAlpha.

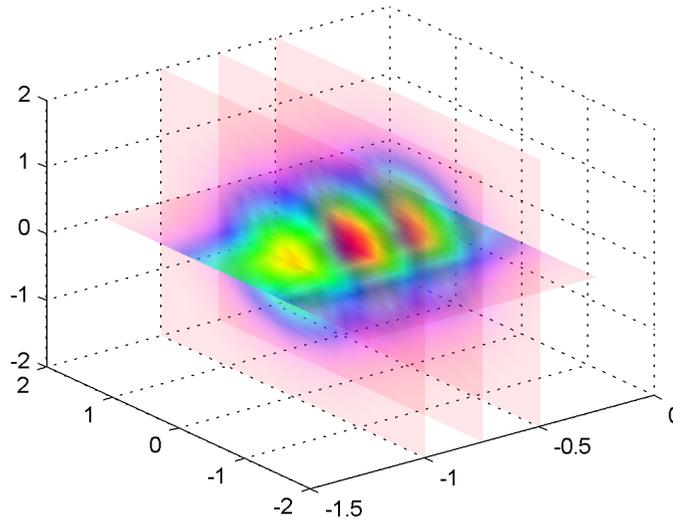
```
h = slice(x,y,z,v,[-1 -.75 -.5],[],[0]);
alpha('color')
set(h,'EdgeColor','none','FaceColor','interp',...
    'FaceAlpha','interp')
```

- 3 Install the rampdown alphamap and increase each value in the alphamap by .1 to achieve the desired degree of transparency. Specify the hsv colormap.

```
alphamap('rampdown')
alphamap('increase',.1)
```

```
colormap(hsv)
```

This colormap causes the smallest values of the function (around zero) to display with the least transparency and the greatest values to display with the most transparency. This enables you to see through the slice planes, while at the same time preserving the data around zero.



Creating 3-D Models with Patches

Introduction to Patch Objects (p. 14-2)	Overview of what a patch object is and how to define one.
Multi-Faceted Patches (p. 14-6)	Shows how to define a 3-D patch object using both x-, y-, and z-coordinate and faces/vertices data. Also illustrates flat and interpolated face coloring.
Specifying Patch Coloring (p. 14-11)	How to specify patch coloring using various patch properties.
Patch Edge Coloring (p. 14-13)	Details about how MATLAB determines patch edge coloring.
Interpreting Indexed and Truecolor Data (p. 14-16)	Specify color data that uses colormaps or defines explicit colors.

Introduction to Patch Objects

A patch graphics object is composed of one or more polygons that may or may not be connected. Patches are useful for modeling real-world objects such as airplanes or automobiles, and for drawing 2- or 3-D polygons of arbitrary shape.

In contrast, surface objects are rectangular grids of quadrilaterals and are better suited for displaying planar topographies such as the values of mathematical functions of two variables, the contours of data in a rectangular plane, or parameterized surfaces such as spheres.

A number of MATLAB functions create patch objects – `fill`, `fill3`, `isosurface`, `isocaps`, some of the contour functions, and `patch`. This section concentrates on use of the `patch` function.

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 because it generally requires less data to define the patch; vertices shared by more than one face need be defined only once. This topic area provides examples of both techniques.

Behavior of the patch Function

There are two forms of the `patch` function – high-level syntax and low-level syntax. The behavior of the `patch` function differs somewhat depending on which syntax you use.

High-Level Syntax

When you use the high-level syntax, MATLAB automatically determines how to color each face based on the color data you specify. The high-level syntax enables you to omit the property names for the x -, y -, and z -coordinates and the color data, as long as you specify these arguments in the correct order.

```
patch(x-coordinates,y-coordinates,z-coordinates,colordata)
```

However, you must specify color data so MATLAB can determine what type of coloring to use. If you do not specify color data, MATLAB returns an error.

```
patch(sin(t),cos(t))
??? Error using ==> patch
Not enough input arguments.
```

Low-Level Syntax

The low-level syntax accepts only property name/property value pairs as arguments and does not automatically color the faces unless you also change the value of the `FaceColor` property. For example, the statement

```
patch('XData',sin(t),'YData',cos(t)) % Low-level syntax
```

draws a patch with white face color because the factory default value for the `FaceColor` property is the color white.

```
get(0,'FactoryPatchFaceColor')
ans =
     1     1     1
```

See the list of patch properties in the MATLAB Function Reference and the `get` command for information on how to obtain the factory and user default values for properties.

Interpreting the Color Argument

When you use the 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 out the color, MATLAB interprets the z -coordinates as color data, and then draws a 2-D patch. For example,

```
h = patch(sin(t),cos(t),1:length(t))
```

draws a patch with all vertices at $z = 0$, colored by interpolating the vertex colors (since there is one color for each vertex), whereas

```
h = patch(sin(t),cos(t),1:length(t),'y')
```

draws a patch with vertices at increasing values of z , colored yellow.

The “Specifying Patch Coloring” section in this chapter provides more information on options for coloring patches.

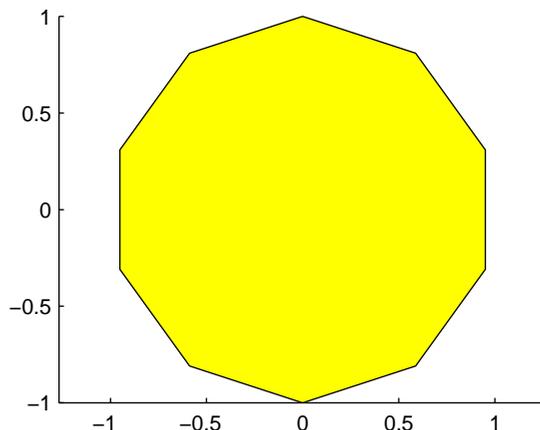
Creating a Single Polygon

A polygon is simply a patch with one face. To create a polygon, specify the coordinates of the vertices and color data with a statement of the form.

```
patch(x-coordinates,y-coordinates,[z-coordinates],colordata)
```

For example, these statements display a 10-sided polygon with a yellow face enclosed by a black edge. The `axis equal` command produces a correctly proportioned polygon.

```
t = 0:pi/5:2*pi;  
patch(sin(t),cos(t),'y')  
axis equal
```

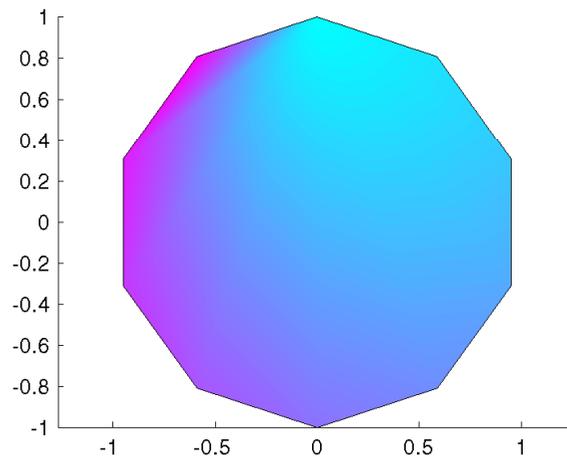


The first and last vertices need not coincide; MATLAB automatically closes each polygonal face of the patch. In fact, it is generally better to define each vertex only once, particularly if you are using interpolated face coloring.

Interpolated Face Colors

You can control many aspects of the patch coloring. For example, instead of specifying a single color, you can provide a range of numerical values that map the color at each vertex to a color in the figure colormap.

```
a = t(1:length(t)-1); %remove redundant vertex definition
patch(sin(a),cos(a),1:length(a),'FaceColor','interp')
colormap cool;
axis equal
```



MATLAB now interpolates the colors across the face of the patch. You can color the edges of the patch the same way, by setting the edge colors to be interpolated. The command is

```
patch(sin(t),cos(t),1:length(t),'EdgeColor','interp')
```

The “Specifying Patch Coloring” section in this chapter provides more information on options for coloring patches.

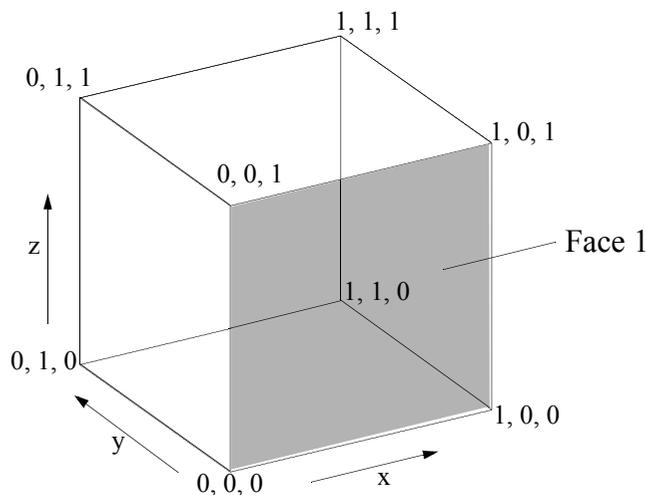
Multi-Faceted Patches

If you specify the x -, y -, and z -coordinate arguments as vectors, MATLAB draws a single polygon by connecting the points. If the arguments are matrices, MATLAB draws one polygon per column, producing a single patch with multiple faces. These faces need not be connected and can be self-intersecting.

Alternatively, you can specify the coordinates of each unique vertex and the order in which to connect them to form the faces. The examples in this section illustrate both techniques.

Example – Defining a Cube

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.



Specifying X, Y, and Z Coordinates

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.

x-coordinates	y-coordinates	z-coordinates
0 1 1 0 0 0	0 0 1 1 0 0	0 0 0 0 0 1
1 1 0 0 1 1	0 1 1 0 0 0	0 0 0 0 0 1
1 1 0 0 1 1	0 1 1 0 1 1	1 1 1 1 0 1
0 1 1 0 0 0	0 0 1 1 1 1	1 1 1 1 0 1

Face 1

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.

	Vertices			Faces			
	x	y	z				
1st vertex	0	0	0	1	2	6	5
2nd vertex	1	0	0	2	3	7	6
⋮	1	1	0	3	4	8	7
⋮	0	1	0	4	1	5	8
5th vertex	0	0	1	1	2	3	4
6th vertex	1	0	1	5	6	7	8
	1	1	1				
	0	1	1				

This data draws the first face by connecting vertices 1, 2, 6, and 5 in that order.

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)
```

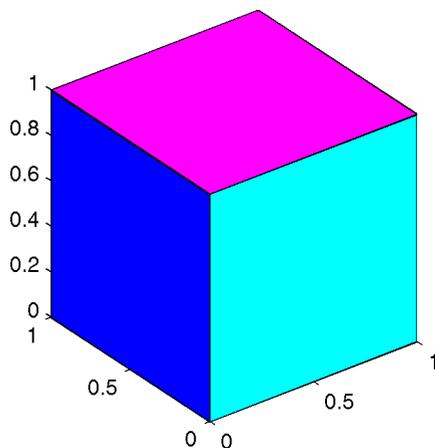
Since the formal syntax does not automatically assign face or edge colors, you must set the appropriate properties to produce patches with colors other than the default white face color and black edge color.

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

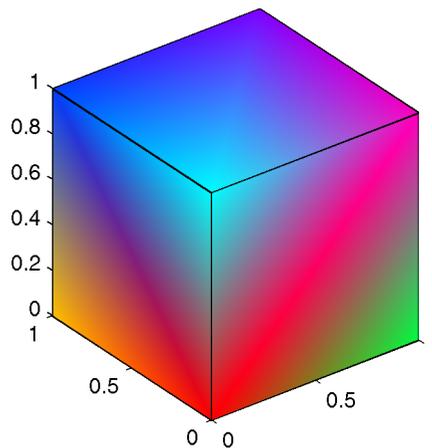
Interpolated face color means the vertex colors of each face define a transition of color from one vertex to the next. To interpolate the colors between vertices, you must specify a color for each vertex and set the `FaceColor` property to `interp`.

```
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.

FaceVertexCData =	CData(:,1) =	
1.00 0.00 0.00	1.00 1.00 0.50 0.00 1.00 0.00	
1.00 0.75 0.00	1.00 0.50 0.00 1.00 1.00 0.00	Red page
0.50 1.00 0.00	0.00 0.50 1.00 0.00 0.50 0.50	
0.00 1.00 0.25	0.00 0.00 0.50 1.00 0.00 1.00	
0.00 1.00 1.00	CData(:,2) =	
0.00 0.25 1.00	0.00 0.75 1.00 1.00 0.00 1.00	Green page
0.50 0.00 1.00	0.75 1.00 1.00 0.00 0.75 0.25	
1.00 0.00 0.75	0.25 0.00 0.00 1.00 1.00 0.00	
Red Green Blue	1.00 0.25 0.00 0.00 1.00 0.00	
	CData(:,3) =	Blue page
	0.00 0.00 0.00 0.25 0.00 1.00	
	0.00 0.00 0.25 0.00 0.00 1.00	
	1.00 1.00 0.75 1.00 0.00 1.00	
	1.00 1.00 1.00 0.75 0.25 0.75	

See “Specifying Patch Coloring” in this chapter for a discussion of coloring techniques in more detail.

Specifying Patch Coloring

Patch coloring is defined differently from surface object coloring in that patches do not automatically generate color data based on the value of the z -coordinate at each vertex. You must explicitly specify patch coloring, or MATLAB uses the default white face color and black edge color.

You can specify patch face coloring by defining:

- A single color for all faces
- One color for each face, which is used for flat coloring
- One color for each vertex, which is used for interpolated coloring

Specify the face color using either the `CData` property, if you are using x -, y -, and z -coordinates or the `FaceVertexCData` property, if you are specifying vertices and faces.

Patch Color Properties

This table summarizes the patch properties that control color (exclusive of those used when light sources are present).

Property	Purpose
<code>CData</code>	Specify single, per face, or per vertex colors in conjunction with x , y , and z 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>	Specifies whether edges are invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors

Property	Purpose
FaceColor	Specifies whether faces are invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors
MarkerEdgeColor	Specifies the color of the marker, or the edge color for filled markers
MarkerFaceColor	Specifies the fill color for markers that are closed shapes

Patch Edge Coloring

Each patch face has a bounding edge, which you can color as:

- A single color for all edges
- A flat color defined by the color of the vertex that precedes the edge
- Interpolated colors determined by the two vertices that bound the edge

Note that patch edge colors can be flat or interpolated only when you specify a color for each vertex. For flat edge coloring, MATLAB uses the color of the vertex preceding the edge to determine the color of the edge. The order in which you specify the vertices establishes which vertex colors a particular edge.

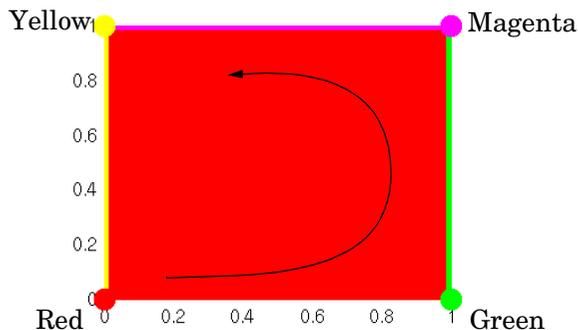
The following examples illustrate patch edge coloring:

- “Example – Specifying Flat Edge and Face Coloring”
- “Coloring Edges with Shared Vertices”

Example – Specifying Flat Edge and Face Coloring

These statements create a square patch.

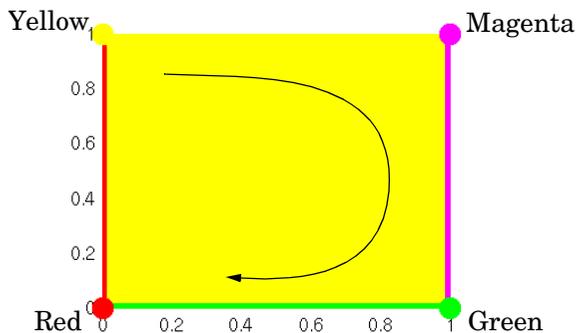
```
v = [0 0 0;1 0 0;1 1 0;0 1 0];  
f = [1 2 3 4];  
fvc = [1 0 0;0 1 0;1 0 1;1 1 0];  
patch('Vertices',v,'Faces',f,'FaceVertexCData',fvc,...  
      'FaceColor','flat','EdgeColor','flat',...  
      'Marker','o','MarkerFaceColor','flat')
```



The Faces property value, [1 2 3 4], determines the order in which MATLAB connects the vertices. In this case, the order is red, green, magenta, and yellow. If you change this order, the results can be quite different. For example, specifying the Faces property as,

```
f = [4 3 2 1];
```

changes the order to yellow, magenta, green, and red. Note that changing the order not only changes the color of the edges, but also the color of the face, which is the color of the first vertex specified.

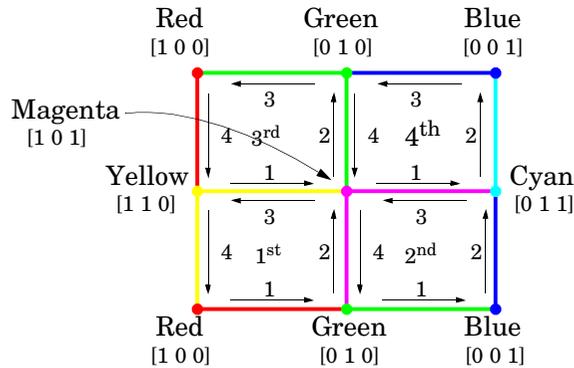


Coloring Edges with Shared Vertices

Each patch face is bound by edges, which are line segments that connect the vertices. When patches have multiple faces that share vertices, some of the

edges 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 (FaceColor set to none, EdgeColor set to flat).



The arrows indicate the order in which each edge is drawn in the first, second, third, and fourth face. The color at each vertex determines the color of the edge that follows it. Notice how the second edge in the first face would be green except that the second face drew its fourth edge from the magenta vertex. You can see similar effects in all shared edges.

For EdgeColor set to interp, MATLAB interpolates colors between adjacent vertices. In this case, the order in which you specify the vertices does not affect the edge color.

Interpreting Indexed and Truecolor Data

MATLAB interprets the patch 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.

Indexed Color Data

MATLAB interprets indexed color data as either values to scale before mapping to the colormap, or directly as indices into the colormap. You control the interpretation by setting the CDataMapping property. The default is to scale the data.

Scaled Color

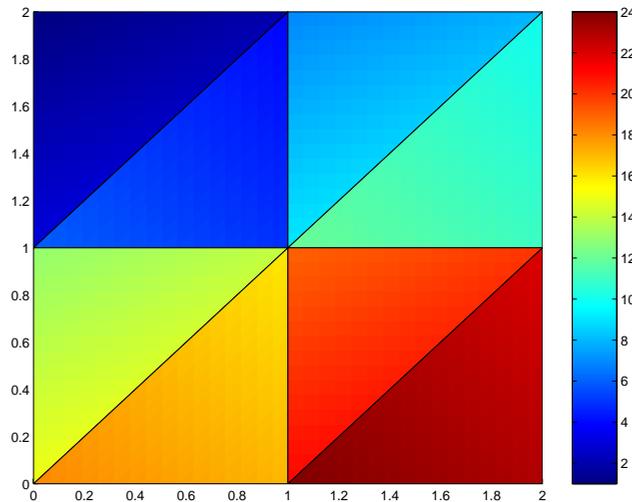
By default, MATLAB scales the color data so that the minimum value maps to the first color in the colormap, the maximum value maps to the last color in the colormap, and values in between are linearly transformed to span the colormap. This enables you to use colormaps of different sizes without changing your data and to use data in any range of values without changing the colormap.

For example, the following patch has eight triangular faces with a total of 24 (nonunique) vertices. The color data are integers that range from one to 24, but could be any values.

The variable `c` contains the color data. It is a 3-by-8 matrix, with each column specifying the colors for the three vertices of each face.

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

The color bar (colorbar) on the right side of the patch illustrates the colormap used and indicates with the vertical axis which color is mapped to the respective data value.



You can alter the mapping of color data to colormap entry using the `caxis` command. This command uses a two-element vector `[cmin cmax]` to specify what data values map to the beginning and end of the colormap, thereby shifting the color mapping.

By default, MATLAB sets `cmin` to the minimum value and `cmax` to the maximum value of the color data of all graphics objects within the axes. However, you can set these limits to span any range of values and thereby shift the color mapping. See “Calculating Color Limits” in the “Axes Properties” 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)))
```

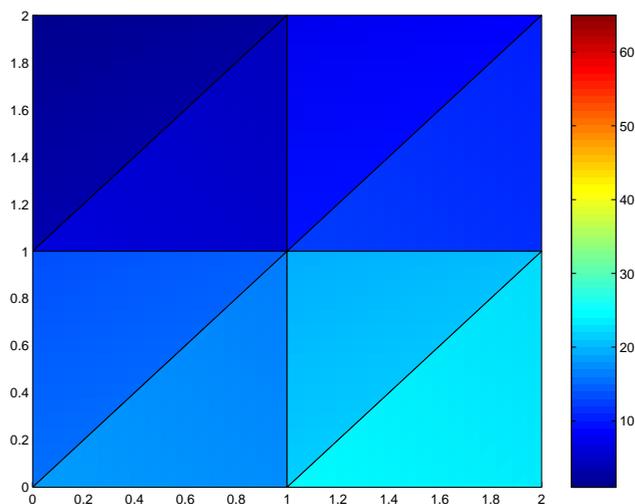
Direct Color

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).

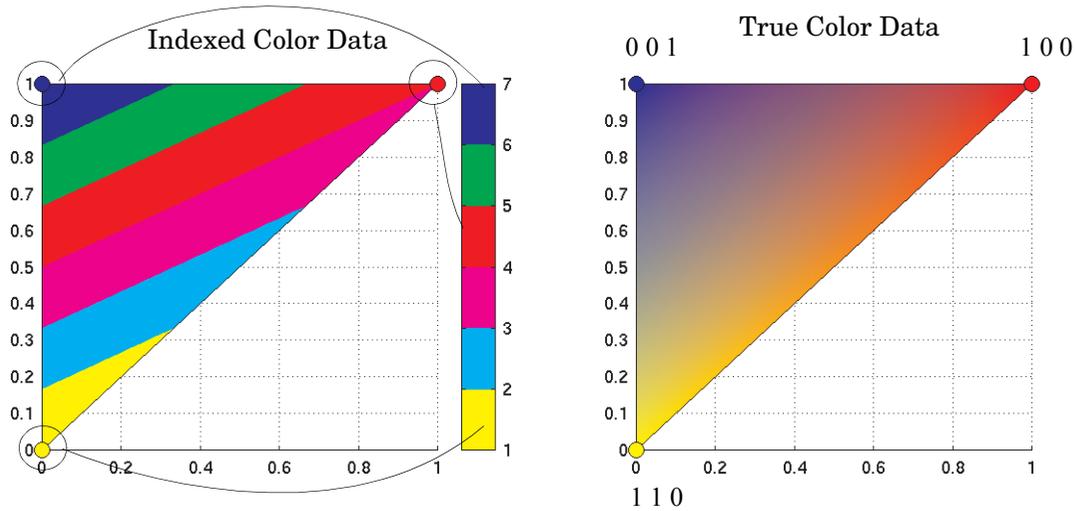
You can use truecolor on computers that do not have true color (24-bit) displays. In this case, MATLAB uses a special colormap designed to produce results that are as close as possible with the limited number of colors available. Properties control how MATLAB uses color on pseudocolor machines.

Interpolating in Indexed Color Versus Truecolor

When you specify interpolated face coloring, MATLAB determines the color of each face by interpolating the vertex colors. The method of interpolation depends on whether you specified truecolor data or indexed color data.

With truecolor data, MATLAB interpolates the numeric RGB values defined for the vertices. This generally produces a smooth variation of color across the face. In contrast, indexed color interpolation uses only colors that are defined in the colormap. With certain colormaps, the results can be quite different.

To illustrate this difference, these two patches are defined with the same vertex colors. Circular markers indicate the yellow, red, and blue vertex colors.



The patch on the left uses indexed colors obtained from the six-element colormap shown next to it. The color data maps the vertex colors to the colormap elements indicated in the picture. With this colormap, interpolating from the cyan vertex to the blue vertex can include only the colors green, red, yellow, and magenta, hence the banding.

Interpolation in RGB space makes no use of the colormap. It is simply the gradual transition from one numeric value to another. For example, interpolating from the cyan vertex to the blue vertex follows a progression similar to these values.

0 1 1, 0 0.9 1, 0 0.8 1, ... 0 0.2 1, 0 0.1 1, 0 0 1

In reality each pixel would be a different color so the incremental change would be much smaller than illustrated here.

Volume Visualization Techniques

Overview of Volume Visualization (p. 15-3)	Volume data visualization with MATLAB, including examples of available techniques.
Volume Visualization Functions (p. 15-5)	Functions used for volume visualization.
Visualizing Scalar Volume Data (p. 15-8)	Techniques available for visualizing scalar volume data.
Visualizing MRI Data (p. 15-9)	Visualize MRI data using 2- and 3-D contour slices, isosurfaces, isocaps, and lighting.
Exploring Volumes with Slice Planes (p. 15-15)	Using slice planes to scan the interior of scalar volumes.
Connecting Equal Values with Isosurfaces (p. 15-20)	Using isosurfaces to illustrate scalar fluid-flow data.
Isocaps Add Context to Visualizations (p. 15-22)	Using isocaps to improve the shape definition of isosurface plots.
Visualizing Vector Volume Data (p. 15-27)	Techniques for visualizing vector volume data, including scalar techniques, determining starting points for stream plots, and plotting subregions of volumes.
Stream Line Plots of Vector Data (p. 15-32)	Using stream lines, slice planes, and contours lines in one graph.
Displaying Curl with Stream Ribbons (p. 15-34)	Example using stream ribbon plots to display the curl of a vector field.
Displaying Divergence with Stream Tubes (p. 15-36)	Example using stream tube plots to display the divergence of a vector field. Slice planes and contour lines enhance the visualization.

Creating Stream Particle Animations (p. 15-39)	Example using stream lines and stream particles to create an animation illustrating wind currents.
Vector Field Displayed with Cone Plots (p. 15-42)	Example using cone plots, isosurfaces, lighting, and camera placement to visualize a vector field.

Overview of Volume Visualization

Volume visualization is the creation of graphical representations of data sets that are defined on three-dimensional grids. Volume data sets are characterized by multidimensional arrays of scalar or vector data. These data are typically defined on lattice structures representing values sampled in 3-D space. There are two basic types of volume data:

- *Scalar volume data* contains single values for each point.
- *Vector volume data* contains two or three values for each point, defining the components of a vector.

Examples of Volume Data

An example of scalar volume data is that produced by the `flow` M-file. The flow data represents the speed profile of a submerged jet within an infinite tank.

Typing

```
[x,y,z,v] = flow;
```

produces four 3-D arrays. The `x`, `y`, and `z` arrays specify the coordinates of the scalar values in the array `v`.

The wind data set is an example of vector volume data that represents air currents over North America. You can load this data in the MATLAB workspace with the command

```
load wind
```

This data set comprises six 3-D arrays: `x`, `y`, and `z` are the coordinate data for the arrays `u`, `v`, and `w`, which are the vector components for each point in the volume.

Selecting Visualization Techniques

The techniques you select to visualize volume data depend on what type of data you have and what you want to learn. In general,

- **Scalar data** is best viewed with isosurfaces, slice planes, and contour slices.
- **Vector data** represents both a magnitude and direction at each point, which is best displayed by stream lines (particles, ribbons, and tubes), cone plots,

and arrow plots. Most visualizations, however, employ a combination of techniques to best reveal the content of the data.

The material in these sections describe how to apply a variety of techniques to typical volume data.

Steps to Create a Volume Visualization

Creating an effective visualization requires a number of steps to compose the final scene. These steps fall into four basic categories:

- 1** Determine the characteristics of your data. Graphing volume data usually requires knowledge of the range of both the coordinates and the data values.
- 2** Select an appropriate plotting routine. The information in this section helps you select the right methods.
- 3** Define the view. The information conveyed by a complex three-dimensional graph can be greatly enhanced through careful composition of the scene. Viewing techniques include adjusting camera position, specifying aspect ratio and project type, zooming in or out, and so on.
- 4** Add lighting and specify coloring. Lighting is an effective means to enhance the visibility of surface shape and to provide a three-dimensional perspective to volume graphs. Color can convey data values, both constant and varying.

Volume Visualization Functions

MATLAB provides functions that enable you to apply a variety of volume visualization techniques. The following tables group these functions into two categories based on the type of data (scalar or vector) that each is designed to work with. The reference page for each function provides examples of the intended use.

Functions for Scalar Data

Functions	Purpose
contourslice	Draw contours in volume slice planes
isocaps	Compute isosurface end-cap geometry
isocolors	Compute the colors of isosurface vertices
isonormals	Compute normals of isosurface vertices
isosurface	Extract isosurface data from volume data
patch	Create a patch (multipolygon) graphics object
reducepatch	Reduce the number of patch faces
reducevolume	Reduce the number of elements in a volume data set
shrinkfaces	Reduce the size of each patch face
slice	Draw slice planes in volume
smooth3	Smooth 3-D data
surf2patch	Convert surface data to patch data
subvolume	Extract subset of volume data set

Functions for Vector Data

Functions	Purpose
coneplot	Plot velocity vectors as cones in 3-D vector fields
curl	Compute the curl and angular velocity of a 3-D vector field
divergence	Compute the divergence of a 3-D vector field
interpstreamspeed	Interpolate streamline vertices from vector-field magnitudes

Functions	Purpose
streamline	Draw stream lines from 2-D or 3-D vector data
streamparticles	Draw stream particles from vector volume data
streamribbon	Draw stream ribbons from vector volume data
streamslice	Draw well-spaced stream lines from vector volume data
streamtube	Draw stream tubes from vector volume data
stream2	Compute 2-D stream line data
stream3	Compute 3-D stream line data
volumebounds	Return coordinate and color limits for volume (scalar and vector)

Visualizing Scalar Volume Data

Typical scalar volume data is composed of a 3-D array of data and three coordinate arrays of the same dimensions. The coordinate arrays specify the x, y, and z coordinates for each data point.

The units of the coordinates depend on the type of data. For example, flow data might have coordinate units of inches and data units of psi.

Techniques for Visualizing Scalar Data

MATLAB supports a number of functions that are useful for visualizing scalar data:

- Slice planes provide a way to explore the distribution of data values within the volume by mapping values to colors. You can orient slice planes at arbitrary angles, as well as use nonplanar slices. (For illustrations of how to use slice planes, see `slice`, a volume slicing example, and slice planes used to show context.) You can specify the data used to color isosurfaces, enabling you to display different information in color and surface shape (see `isocolors`).
- Contour slices are contour plots drawn at specific coordinates within the volume. Contour plots enable you to see where in a given plane the data values are equal. See `contourslice` and MRI data for an example
- Isosurfaces are surfaces constructed by using points of equal value as the vertices of patch graphics objects.

Visualizing MRI Data

An example of scalar data includes Magnetic Resonance Imaging (MRI) data. This data typically contains a number of slice planes taken through a volume, such as the human body. MATLAB includes an MRI data set that contains 27 image slices of a human head. This section describes some useful techniques for visualizing MRI data.

Example - Ways to Display MRI DATA

This example illustrates the following techniques applied to MRI data:

- A series of 2-D images representing slices through the head
- 2-D and 3-D contour slices taken at arbitrary locations within the data
- An isosurface with isocaps showing a cross section of the interior

Changing the Data Format

The MRI data, `D`, is stored as a 128-by-128-by-1-by-27 array. The third array dimension is used typically for the image color data. However, since these are indexed images (a colormap, `map`, is also loaded) there is no information in the third dimension, which you can remove using the `squeeze` command. The result is a 128-by-128-by-27 array.

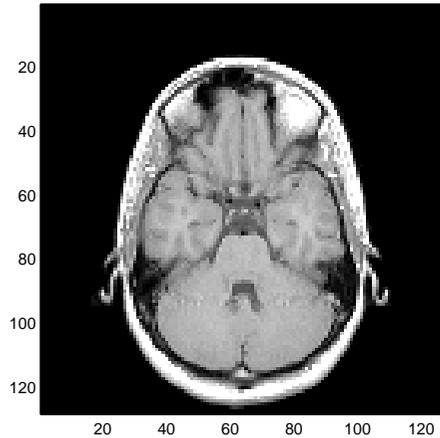
The first step is to load the data and transform the data array from 4-D to 3-D.

```
load mri
D = squeeze(D);
```

Displaying Images of MRI Data

To display one of the MRI images, use the `image` command, indexing into the data array to obtain the eighth image. Then adjust axis scaling, and install the MRI colormap, which was loaded along with the data.

```
image_num = 8;
image(D(:,:,image_num))
axis image
colormap(map)
```



Save the x and y axis limits for use in the next part of the example.

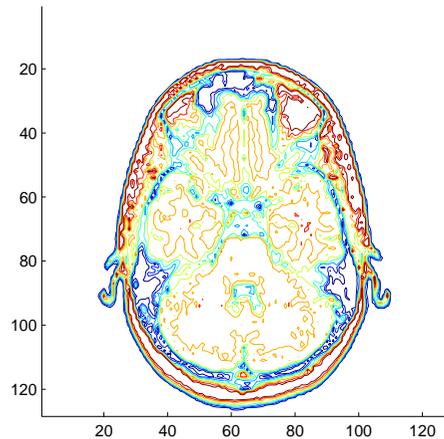
```
x = xlim;  
y = ylim;
```

Displaying a 2-D Contour Slice

You can treat this MRI data as a volume because it is a collection of slices taken progressively through the 3-D object. Use `contourslice` to display a contour plot of a slice of the volume. To create a contour plot with the same orientation and size as the image created in the first part of this example, adjust the y -axis direction (axis), set the limits (`xlim`, `ylim`), and set the data aspect ratio (`daspect`).

```
contourslice(D,[],[],image_num)  
axis ij  
xlim(x)  
ylim(y)  
daspect([1,1,1])  
colormap('default')
```

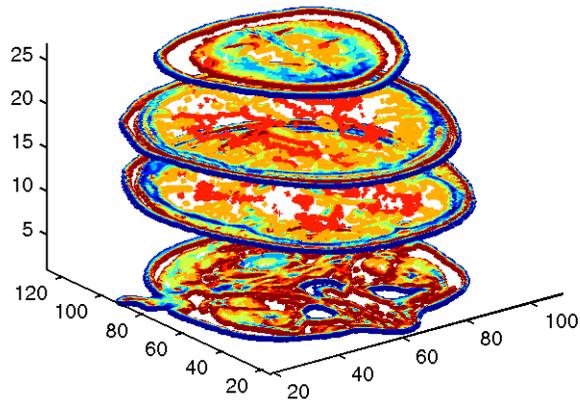
This contour plot uses the figure colormap to map color to contour value.



Displaying 3-D Contour Slices

Unlike images, which are 2-D objects, contour slices are 3-D objects that you can display in any orientation. For example, you can display four contour slices in a 3-D view. To improve the visibility of the contour line, increase the `LineWidth` to 2 points (one point equals 1/72 of an inch).

```
phandles = contourslice(D,[],[],[1,12,19,27],8);  
view(3); axis tight  
set(phandles,'LineWidth',2)
```



Displaying an Isosurface

You can use isosurfaces to display the overall structure of a volume. When combined with isocaps, this technique can reveal information about data on the interior of the isosurface.

First, smooth the data with `smooth3`; then use `isosurface` to calculate the isodata. Use `patch` to display this data as a graphics object.

```
Ds = smooth3(D);
hiso = patch(isosurface(Ds,5),...
    'FaceColor',[1,.75,.65],...
    'EdgeColor','none');
```

Adding an Isocap to Show a Cutaway Surface

Use `isocaps` to calculate the data for another patch that is displayed at the same isovalue (5) as the surface. Use the unsmoothed data (`D`) to show details of the interior. You can see this as the sliced-away top of the head.

```
hcap = patch(isocaps(D,5),...
    'FaceColor','interp',...
    'EdgeColor','none');
colormap(map)
```

Defining the View

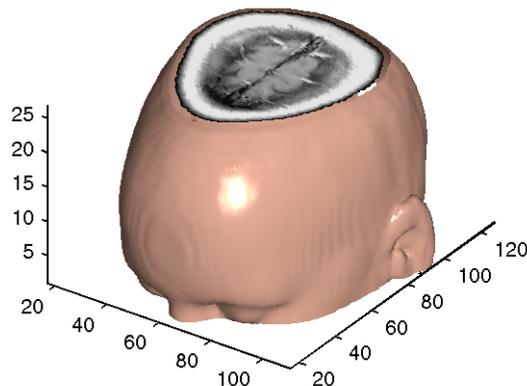
Define the view and set the aspect ratio (view, axis, daspect).

```
view(45,30)
axis tight
daspect([1,1,.4])
```

Add Lighting

Add lighting and recalculate the surface normals based on the gradient of the volume data, which produces smoother lighting (camlight, lighting, isonormals). Increase the AmbientStrength property of the isocap to brighten the coloring without affecting the isosurface. Set the SpecularColorReflectance of the isosurface to make the color of the specular reflected light closer to the color of the isosurface; then set the SpecularExponent to reduce the size of the specular spot.

```
lightangle(45,30);
set(gcf,'Renderer','zbuffer'); lighting phong
isonormals(Ds,hiso)
set(hcap,'AmbientStrength',.6)
set(hiso,'SpecularColorReflectance',0,'SpecularExponent',50)
```



The isocap uses interpolated face coloring, which means the figure colormap determines the coloring of the patch. This example uses the colormap supplied with the data.

To display isocaps at other data values, try changing the isosurface value or use the `subvolume` command. See the `isocaps` and `subvolume` reference pages for examples.

Exploring Volumes with Slice Planes

A slice plane (which does not have to be planar) is a surface that takes on coloring based on the values of the volume data in the region where the slice is positioned. Slice planes are useful for probing volume data sets to discover where interesting regions exist, which you can then visualize with other types of graphs (see the `slice` example). Slice planes are also useful for adding a visual context to the bound of the volume when other graphing methods are also used (see `coneplot` and the “Streamline Plots of Vector Data” section in this chapter for examples).

Use the `slice` command to create slice planes.

Example – Slicing Fluid Flow Data

This example slices through a volume generated by the `flow` M-file.

1. Investigate the Data

Generate the volume data with the command

```
[x,y,z,v] = flow;
```

Determine the range of the volume by finding the minimum and maximum of the coordinate data.

```
xmin = min(x(:));  
ymin = min(y(:));  
zmin = min(z(:));  
  
xmax = max(x(:));  
ymax = max(y(:));  
zmax = max(z(:));
```

2. Create a Slice Plane at an Angle to the X-Axes

To create a slice plane that does not lie in an axes plane, first define a surface and rotate it to the desired orientation. This example uses a surface that has the same `x` and `y` coordinates as the volume.

```
hslice = surf(linspace(xmin,xmax,100),...  
             linspace(ymin,ymax,100),...  
             zeros(100));
```

Rotate the surface by -45 degrees about the x axis and save the surface XData, YData, and ZData to define the slice plane; then delete the surface.

```
rotate(hslice, [-1,0,0], -45)
xd = get(hslice, 'XData');
yd = get(hslice, 'YData');
zd = get(hslice, 'ZData');
delete(hslice)
```

3. Draw the Slice Planes

Draw the rotated slice plane, setting the FaceColor to interp so that it is colored by the figure colormap and set the EdgeColor to none. Increase the DiffuseStrength to .8 to make this plane shine more brightly after adding a light source.

```
h = slice(x,y,z,v,xd,yd,zd);
set(h, 'FaceColor', 'interp', ...
    'EdgeColor', 'none', ...
    'DiffuseStrength', .8)
```

Set hold to on and add three more orthogonal slice planes at xmax, ymax, and zmin to provide a context for the first plane, which slices through the volume at an angle.

```
hold on
hx = slice(x,y,z,v,xmax,[],[]);
set(hx, 'FaceColor', 'interp', 'EdgeColor', 'none')

hy = slice(x,y,z,v,[],ymax,[]);
set(hy, 'FaceColor', 'interp', 'EdgeColor', 'none')

hz = slice(x,y,z,v,[],[],zmin);
set(hz, 'FaceColor', 'interp', 'EdgeColor', 'none')
```

4. Define the View

To display the volume in correct proportions, set the data aspect ratio to [1,1,1] (daspect). Adjust the axis to fit tightly around the volume (axis) and turn on the box to provide a sense of a 3-D object. The orientation of the axes can be selected initially using rotate3d to determine the best view.

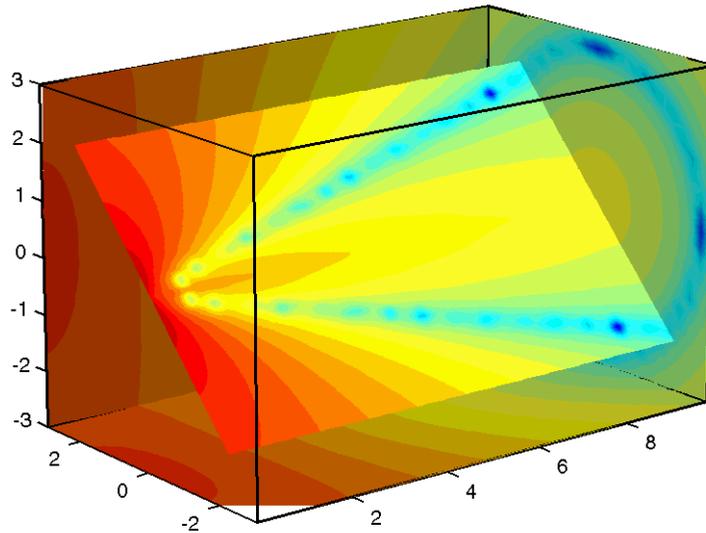
Zooming in on the scene provides a larger view of the volume (`camzoom`). Selecting a projection type of perspective gives the rectangular solid more natural proportions than the default orthographic projection (`camproj`).

```
daspect([1,1,1])
axis tight
box on
view(-38.5,16)
camzoom(1.4)
camproj perspective
```

5. Add Lighting and Specify Colors

Adding a light to the scene makes the boundaries between the four slice planes more obvious since each plane forms a different angle with the light source (`lightangle`). Selecting a colormap with only 24 colors (the default is 64) creates visible gradations that help indicate the variation within the volume.

```
lightangle(-45,45)
colormap (jet(24))
set(gcf, 'Renderer', 'zbuffer')
```



Modifying the Color Mapping

The current colormap determines the coloring of the slice planes. This enables you to change the slice plane coloring by

- Changing the colormap
- Changing the mapping of data value to color

Suppose, for example, you are interested in data values only between -5 and 2.5 and would like to use a colormap that mapped lower values to reds and higher values to blues (that is, the opposite of the default jet colormap).

Customizing the Colormap

The first step is to flip the colormap (`colormap, flipud`).

```
colormap (flipud(jet(24)))
```

Adjusting the Color Limits

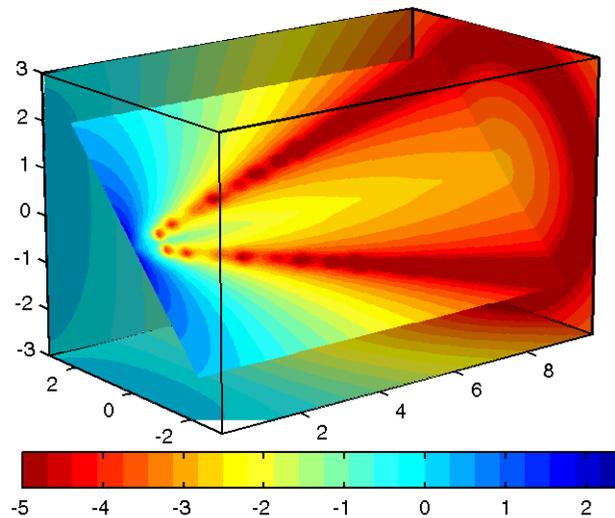
Adjusting the color limits enables you to emphasize any particular data range of interest. Adjust the color limits to range from -5 to 2.4832 so that any value

lower than the value -5 (the original data ranged from -11.5417 to 2.4832) is mapped into the same color. (See `caxis` and the "Axis Color Limits – The `CLim` Property" section in the "Axes Properties" chapter for an explanation of color mapping.)

```
caxis([-5,2.4832])
```

Adding a color bar provides a key for the data to color mapping.

```
colorbar('horiz')
```



Connecting Equal Values with Isosurfaces

Isosurfaces are constructed by creating a surface within the volume that has the same value at each vertex. Isosurface plots are similar to contour plots in that they both indicate where values are equal.

Isosurfaces are useful to determine where in a volume a certain threshold value is reached or to observe the spacial distribution of data by selecting various isovalues at which to generate a plot. The isovalue must lie within the range of the volume data.

Create isosurfaces with the `isosurface` and `patch` commands.

Example – Isosurfaces in Fluid Flow Data

This example creates isosurfaces in a volume generated by the `flow` M-file. Generate the volume data with the command,

```
[x,y,z,v] = flow;
```

To select the isovalue, determine the range of values in the volume data.

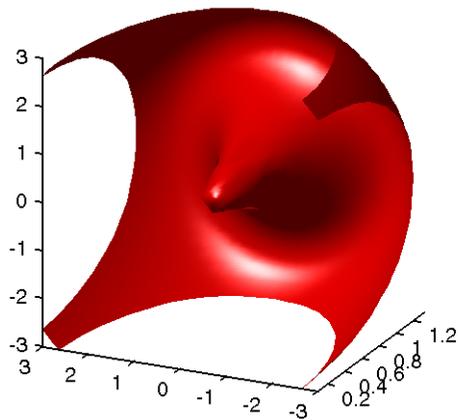
```
min(v(:))  
ans =  
    -11.5417  
max(v(:))  
ans =  
     2.4832
```

Through exploration, you can select isovalues that reveal useful information about the data. Once selected, use the isovalue to create the isosurface:

- Use `isosurface` to generate data that you can pass directly to `patch`.
- Recalculate the surface normals from the gradient of the volume data to produce better lighting characteristics (`isonormals`).
- Set the `patch` `FaceColor` to red and the `EdgeColor` to none to produce a smoothly lit surface.
- Adjust the view and add lighting (`daspect`, `view`, `camlight`, `lighting`).

```
hpatch = patch(isosurface(x,y,z,v,0));  
isonormals(x,y,z,v,hpatch)  
set(hpatch,'FaceColor','red','EdgeColor','none')
```

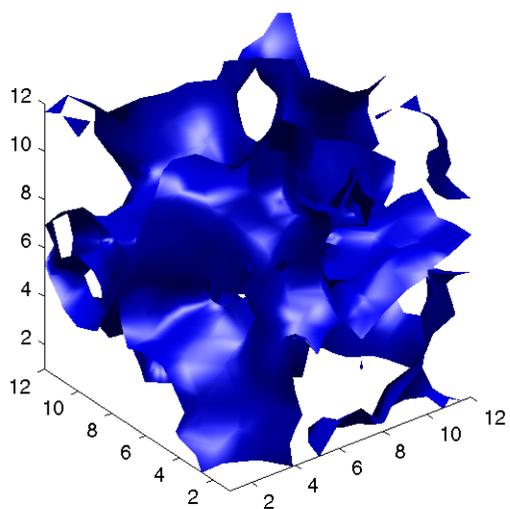
```
daspect([1,4,4])  
view([-65,20])  
axis tight  
camlight left;  
set(gcf,'Renderer','zbuffer'); lighting phong
```



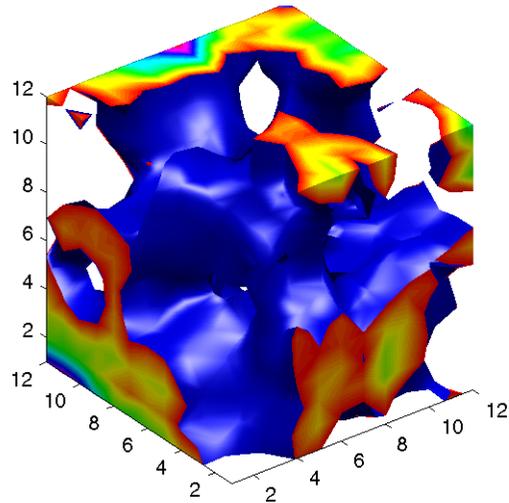
Isocaps Add Context to Visualizations

Isocaps are planes that are fitted to the limits of an isosurface to provide a visual context for the isosurface. Isocaps show a cross-sectional view of the interior of the isosurface for which it provides an *end cap*.

The following two pictures illustrate the use of isocaps. The first is an isosurface without isocaps.



The second picture shows the effect of adding isocaps to the same isosurface.



Other Isocap Applications

Some additional applications of isocaps are

- Isocaps used to show the interior of a cut-away volume.
- Isocaps used to cap the end of a volume that would otherwise appear empty.
- Isocaps used to enhance the visibility of the isosurface limits.

Defining Isocaps

Isocaps, like isosurfaces, are created as patch graphics objects. Use the `isocaps` command to generate the data to pass to `patch`. For example,

```
patch(isocaps(voldata,isoval),...  
      'FaceColor','interp',...  
      'EdgeColor','none')
```

creates isocaps for the scalar volume data `voldata` at the value `isoval`. You should create the isosurface using the same volume data and isovalue to ensure that the edges of the isocaps *fit* the isosurface.

Setting the patch `FaceColor` property to `interp` results in a coloring that maps the data values spanned by the `isocap` to `colormap` entries. You can also set other patch properties to control the effects of lighting and coloring on the `isocaps`.

Example – Adding Isocaps to an Isosurface

This example illustrates how to set coloring and lighting characteristics when working with `isocaps`. There are five basic steps:

- Generate and process your volume data.
- Create the isosurface and `isocaps` and set patch properties to control the coloring and lighting.
- Create the `isocaps` and set properties.
- Specify the view.
- Add lights to the scene.

1. Prepare the Data

This example uses a 3-D array of random (`rand`) data to define the volume data. The data is then smoothed (`smooth3`).

```
data = rand(12,12,12);  
data = smooth3(data, 'box', 5);
```

2. Create the Isosurface and Set Properties

Use `isosurface` and `patch` to create the isosurface and set coloring and lighting properties. Reduce the `AmbientStrength`, `SpecularStrength`, and `DiffuseStrength` of the reflected light to compensate for the brightness of the two light sources used to provide more uniform lighting.

Recalculate the vertex normals of the isosurface to produce smoother lighting (`isonormals`).

```
isoval = .5;  
h = patch(isosurface(data, isoval), ...  
    'FaceColor', 'blue', ...  
    'EdgeColor', 'none', ...  
    'AmbientStrength', .2, ...  
    'SpecularStrength', .7, ...  
    'DiffuseStrength', .4);
```

```
isonormals(data,h)
```

3. Create the Isocaps and Set Properties

Define the isocaps using the same data and isovalue as the isosurface. Specify interpolated coloring and select a colormap that provides better contrasting colors with the blue isosurface than those in the default colormap (colormap).

```
patch(isocaps(data,isoval),...
      'FaceColor','interp',...
      'EdgeColor','none')
colormap hsv
```

4. Define the View

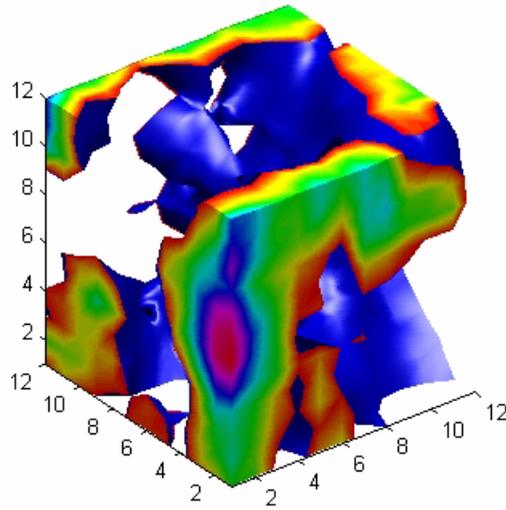
Set the data aspect ratio to [1,1,1] so that the displays in correct proportions (daspect). Eliminate white space within the axis and set the view to 3-D (axis tight, view).

```
daspect([1,1,1])
axis tight
view(3)
```

5. Add Lighting

To add fairly uniform lighting, but still take advantage of the ability of light sources to make visible subtle variations in shape, this example uses two lights; one to the left and one to the right of the camera (camlight). Use Phong lighting to produce the smoothest variation of color (lighting). Phong lighting requires the zbuffer renderer.

```
camlight right
camlight left
set(gcf,'Renderer','zbuffer');
lighting phong
```



Visualizing Vector Volume Data

Vector volume data contains more information than scalar data because each coordinate point in the data set has three values associated with it. These values define a vector that represents both a magnitude and a direction. The velocity of fluid flow is an example of vector data.

MATLAB supports a number of techniques that are useful for visualizing vector data:

- Stream lines trace the path that a massless particle immersed in the vector field would follow.
- Stream particles are markers that trace stream lines and are useful for creating stream line animations.
- Stream ribbons are similar to stream lines, except that the width of the ribbons enable them to indicate twist. Stream ribbons are useful to indicate curl angular velocity.
- Stream tubes are similar to stream lines, but you can also control the width of the tube. Stream tubes are useful for displaying the divergence of a vector field.
- Cone plots represent the magnitude and direction of the data at each point by displaying a conical arrowhead or an arrow.

It is typically the case that these functions best elucidate the data when used in conjunction with other visualization techniques, such as contours, slice planes, and isosurfaces. The examples in this section illustrate some of these techniques.

Using Scalar Techniques with Vector Data

Visualization techniques such as contour slices, slice planes, and isosurfaces require scalar volume data. You can use these techniques with vector data by taking the magnitude of the vectors. For example, the wind data set returns three coordinate arrays and three vector component arrays, u , v , w . In this case, the magnitude of the velocity vectors equals the wind speed at each corresponding coordinate point in the volume.

```
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

The array `wind_speed` contains scalar values for the volume data. The usefulness of the information produced by this approach, however, depends on what physical phenomena is represented by the magnitude of your vector data.

Specifying Starting Points for Stream Plots

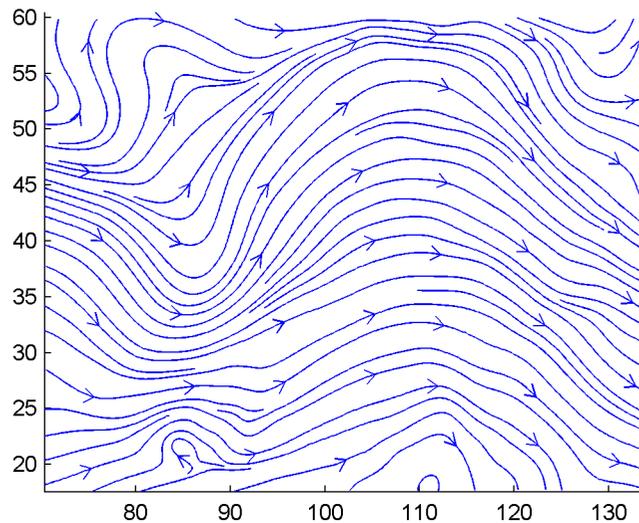
Stream plots (stream lines, ribbons, tubes, and cones or arrows) illustrate the flow of a 3-D vector field. The MATLAB stream plotting routines (`streamline`, `streamribbon`, `streamtube`, `coneplot`, `stream2`, `stream3`) all require you to specify the point at which you want to begin each stream trace.

Determining the Starting Points

Generally, knowledge of your data's characteristics help you select the starting points. Information such as the primary direction of flow and the range of the data coordinates helps you decide where to evaluate the data.

The `streamslice` function is useful for exploring your data. For example, these statements draw a slice through the vector field at a `z` value midway in the range.

```
load wind
zmax = max(z(:)); zmin = min(z(:));
streamslice(x,y,z,u,v,w,[],[],(zmax-zmin)/2)
```



This stream slice plot indicates that the flow is in the positive x direction and also enables you to select starting points in both x and y. You could create similar plots that slice the volume in the x-z plane or the y-z plane to gain further insight into your data's range and orientation.

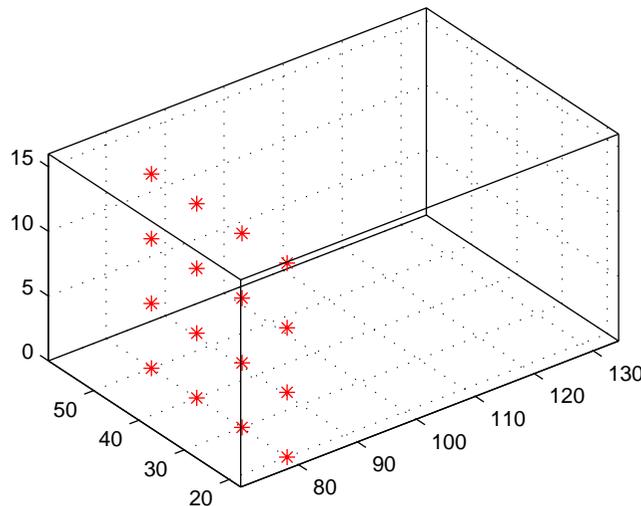
Specifying Arrays of Starting-Point Coordinates

To specify the starting point for one stream line, you need the x-, y-, and z-coordinates of the point. The `meshgrid` command provides a convenient way to create arrays of starting points. For example, you could select the following starting points from the wind data displayed in the previous stream slice.

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
```

This statement defines the starting points as all lying on $x = 80$, y ranging from 20 to 50, and z ranging from 0 to 15. You can use `plot3` to display the locations.

```
plot3(sx(:),sy(:),sz(:),'*r');
axis(volumebounds(x,y,z,u,v,w))
grid; box; daspect([2 2 1])
```



You do not need to use 3-D arrays, such as those returned by `meshgrid`, but the size of each array must be the same, and `meshgrid` provides a convenient way to generate arrays when you do not have an equal number of unique values in

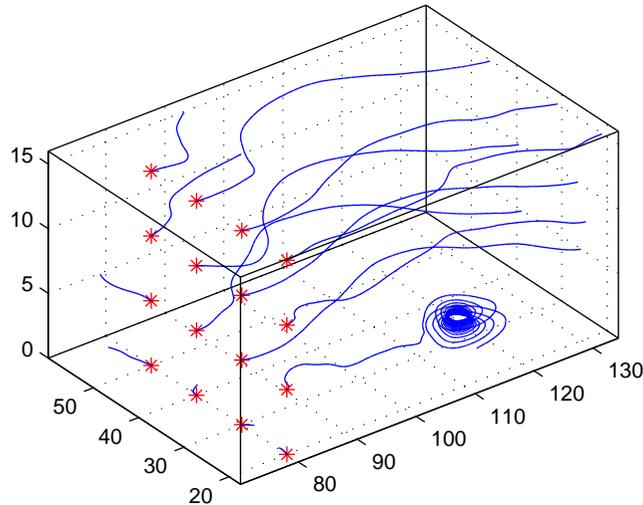
each coordinate. You can also define starting-point arrays as column vectors. For example, `meshgrid` returns 3-D arrays.

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
whos
Name      Size      Bytes  Class
sx        4x1x4      128   double array
sy        4x1x4      128   double array
sz        4x1x4      128   double array
```

In addition, you could use 16-by-1 column vectors with the corresponding elements of the three arrays comprising the coordinates of each starting point. (This is the equivalent of indexing the values returned by `meshgrid` as `sx(:)`, `sy(:)`, and `sz(:)`.)

For example, adding the stream lines produces

```
streamline(x,y,z,u,v,w,sx(:),sy(:),sz(:))
```



Accessing Subregions of Volume Data

The `subvolume` function provides a simple way to access subregions of a volume data set. `subvolume` enables you to select regions of interest based on limits rather than using the colon operator to index into the 3-D arrays that define

volumes. Consider the following two approaches to creating the data for a subvolume – indexing with the colon operator and using subvolume.

Indexing with the Colon Operator

When you index the arrays, you work with values that specify the elements in each dimension of the array.

```
load wind
xsub = x(1:10,20:30,1:7);
ysub = y(1:10,20:30,1:7);
zsub = z(1:10,20:30,1:7);
usub = u(1:10,20:30,1:7);
vsub = v(1:10,20:30,1:7);
wsub = w(1:10,20:30,1:7);
```

Using the subvolume Function

subvolume enables you to use coordinate values that you can read from the axes. For example,

```
lims = [100.64 116.67 17.25 28.75 -0.02 6.86];
[xsub,ysub,zsub,usub,vsub,wsub] = subvolume(x,y,z,u,v,w,lims);
```

You can then use the subvolume data as inputs to any function requiring vector volume data.

Stream Line Plots of Vector Data

MATLAB includes a vector data set called `wind` that represents air currents over North America. This example uses a combination of techniques:

- Stream lines to trace the wind velocity
- Slice planes to show cross-sectional views of the data
- Contours on the slice planes to improve the visibility of slice-plane coloring

1. Determine the Range of the Coordinates

Load the data and determine minimum and maximum values to locate the slice planes and contour plots (`load`, `min`, `max`).

```
load wind
xmin = min(x(:));
xmax = max(x(:));
ymax = max(y(:));
zmin = min(z(:));
```

2. Add Slice Planes for Visual Context

Calculate the magnitude of the vector field (which represents wind speed) to generate scalar data for the `slice` command. Create slice planes along the x-axis at `xmin`, 100, and `xmax`, along the y-axis at `ymax`, and along the z-axis at `zmin`. Specify interpolated face coloring so the slice coloring indicates wind speed, and do not draw edges (`sqrt`, `slice`, `FaceColor`, `EdgeColor`).

```
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
hsurfaces = slice(x,y,z,wind_speed,[xmin,100,xmax],ymax,zmin);
set(hsurfaces,'FaceColor','interp','EdgeColor','none')
```

3. Add Contour Lines to the Slice Planes

Draw light gray contour lines on the slice planes to help quantify the color mapping (`contourslice`, `EdgeColor`, `LineWidth`).

```
hcont = ...
contourslice(x,y,z,wind_speed,[xmin,100,xmax],ymax,zmin);
set(hcont,'EdgeColor',[.7,.7,.7],'LineWidth',.5)
```

4. Define the Starting Points for the Stream Lines

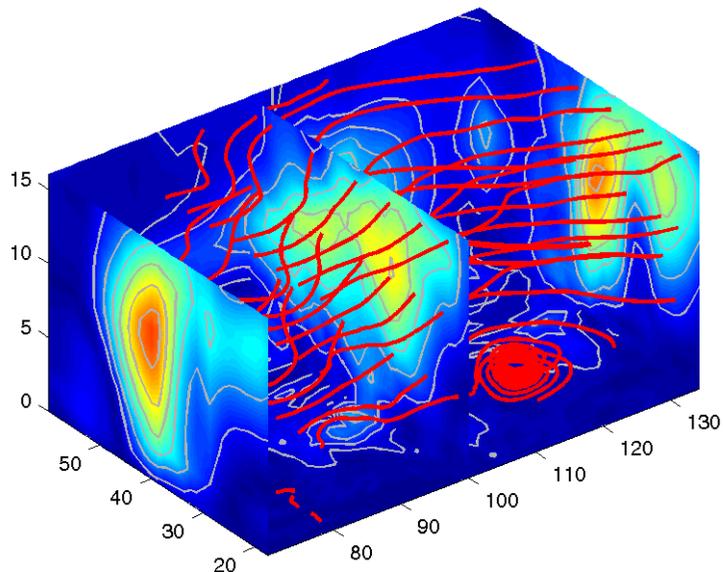
In this example, all stream lines start at an x-axis value of 80 and span the range 20 to 50 in the y direction and 0 to 15 in the z direction. Save the handles of the stream lines and set the line width and color (meshgrid, streamline, LineWidth, Color).

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
hlines = streamline(x,y,z,u,v,w,sx,sy,sz);
set(hlines,'LineWidth',2,'Color','r')
```

5. Define the View

Set up the view, expanding the z-axis to make it easier to read the graph (view, daspect, axis).

```
view(3)
daspect([2,2,1])
axis tight
```



See coneplot for an example of the same data plotted with cones.

Displaying Curl with Stream Ribbons

Stream ribbons illustrate direction of flow, similar to stream lines, but can also show rotation about the flow axis by twisting the ribbon-shaped flow line. The `streamribbon` function enables you to specify a twist angle (in radians) for each vertex in the stream ribbons.

When used in conjunction with the `curl` function, `streamribbon` is useful for displaying the curl angular velocity of a vector field. The following example illustrates this technique.

1. Select a Subset of Data to Plot

Load and select a region of interest in the wind data set using `subvolume`. Plotting the full data set first can help you select a region of interest.

```
load wind
lims = [100.64 116.67 17.25 28.75 -0.02 6.86];
[x,y,z,u,v,w] = subvolume(x,y,z,u,v,w,lims);
```

2. Calculate Curl Angular Velocity and Wind Speed

Calculate the curl angular velocity and the wind speed.

```
cav = curl(x,y,z,u,v,w);
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

3. Create the Stream Ribbons

- Use `meshgrid` to create arrays of starting points for the stream ribbons. See "Starting Points for Stream Plots" in this chapter for information on specifying the arrays of starting points.
- `stream3` calculates the stream line vertices with a step size of .5
- `streamribbon` scales the width of the ribbon by a factor of 2 to enhance the visibility of the twisting (which indicates curl angular velocity).
- `streamribbon` returns the handles of the surface objects it creates, which are then used to set the color to red (`FaceColor`), the color of the surface edges to light gray (`EdgeColor`), and slightly increase the brightness of the ambient light reflected when lighting is applied (`AmbientStrength`).

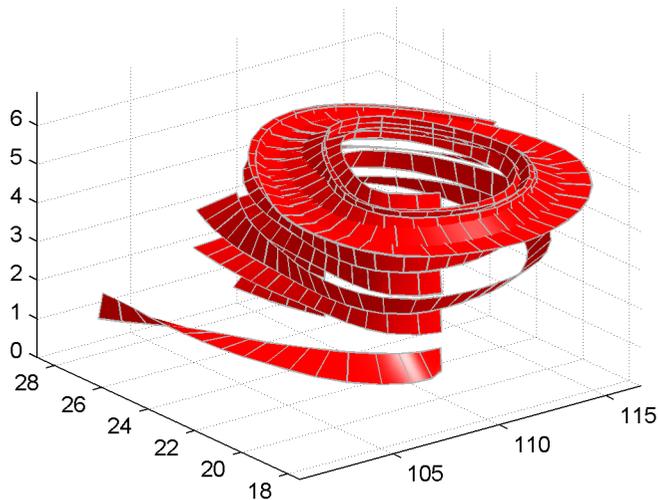
```
[sx sy sz] = meshgrid(110,20:5:30,1:5);
verts = stream3(x,y,z,u,v,w,sx,sy,sz,.5);
```

```
h = streamribbon(verts,x,y,z,cav,wind_speed,2);  
set(h,'FaceColor','r',...  
    'EdgeColor',[.7 .7 .7],...  
    'AmbientStrength',.6)
```

4. Define the View and Add Lighting

- The `volumebounds` command provides a convenient way to set axis and color limits.
- Add a grid and set the view for 3-D (`streamribbon` does not change the current view).
- `camlight` creates a light positioned to the right of the view point and `lighting` sets the lighting method to Phong (which requires the `zbuffer` renderer).

```
axis(volumebounds(x,y,z,wind_speed))  
grid on  
view(3)  
camlight right;  
set(gcf,'Renderer','zbuffer'); lighting phong
```



Displaying Divergence with Stream Tubes

Stream tubes are similar to stream lines, except the tubes have width, providing another dimension that you can use to represent information.

By default, MATLAB indicates the divergence of the vector field by the width of the tube. You can also define widths for each tube vertex and thereby map other data to width.

This example uses the following techniques:

- Stream tubes indicate flow direction and divergence of the vector field in the wind data set.
- Slice planes colored to indicate the speed of the wind currents overlaid with contour line to enhance visibility

Inputs include the coordinates of the volume, vector field components, and starting location for the stream tubes.

1. Load Data and Calculate Required Values

The first step is to load the data and calculate values needed to make the plots. These values include

- The location of the slice planes (maximum x, minimum y, and a value for the altitude)
- The minimum x value for the start of the stream tubes
- The speed of the wind (magnitude of the vector field)

```
load wind
xmin = min(x(:));
xmax = max(x(:));
ymin = min(y(:));
alt = 7.356; % z-value for slice and streamtube plane
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
```

2. Draw the Slice Planes

Draw the slice planes (`slice`) and set surface properties to create a smoothly colored slice. Use 16 colors from the `hsv colormap`.

```
hslice = slice(x,y,z,wind_speed,xmax,ymin,alt);
set(hslice,'FaceColor','interp','EdgeColor','none')
```

```
colormap hsv(16)
```

3. Add Contour Lines to Slice Planes

Add contour lines (`contourslice`) to the slice planes. Adjust the contour interval so the lines match the color boundaries in the slice planes by

- Calling `caxis` to get the current color limits
- Setting the interpolation method used by `contourslice` to `linear` to match the default used by `slice`.

```
color_lim = caxis;
cont_intervals = linspace(color_lim(1),color_lim(2),17);
hcont = contourslice(x,y,z,wind_speed,xmax,ymin,...
    alt,cont_intervals,'linear');
set(hcont,'EdgeColor',[.4 .4 .4],'LineWidth',1)
```

4. Create the Stream Tubes

Use `meshgrid` to create arrays for the starting points for the stream tubes, which begin at the minimum `x` value, range from 20 to 50 in `y`, and lie in a single plane in `z` (corresponding to one of the slice planes).

The stream tubes (`streamtube`) are drawn at the specified locations and scaled to be 1.25 times the default width to emphasize the variation in divergence (width). The second element in the vector `[1.25 30]` specifies the number of points along the circumference of the tube (the default is 20). You may want to increase this value as the tube size increases to maintain a smooth-looking tube.

Set the data aspect ratio (`daspect`) before calling `streamtube`.

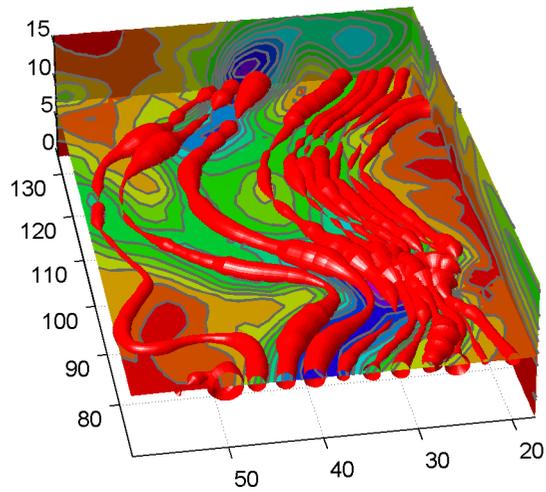
Stream tubes are surface objects therefore you can control their appearance by setting (surface properties). This example sets surface properties to give a brightly lit, red surface.

```
[sx,sy,sz] = meshgrid(xmin,20:3:50,alt);
daspect([1,1,1]) % set DAR before calling streamtube
htubes = streamtube(x,y,z,u,v,w,sx,sy,sz,[1.25 30]);
set(htubes,'EdgeColor','none','FaceColor','r',...
    'AmbientStrength',.5)
```

5. Define the View

The final step is to define the view and add lighting (view, axis, volumebounds, Projection, camlight).

```
view(-100,30)  
axis(volumebounds(x,y,z,wind_speed))  
set(gca,'Projection','perspective')  
camlight left
```



Creating Stream Particle Animations

A stream particle animation is useful for visualizing the flow direction and speed of a vector field. The “particles” (represented by any of the line markers) trace the flow along a particular stream line. The speed of each particle in the animation is proportional to the magnitude of the vector field at any given point along the stream line.

1. Specify the Starting Points of the Data Range to Plot

This example determines the region of the volume to plot by specifying the appropriate starting points. In this case, the stream plots begin at $x = 100$, y spans 20 to 50 and in the $z = 5$ plane. Note that this is not the full volume bounds.

```
load wind
[sx sy sz] = meshgrid(100,20:2:50,5);
```

2. Create Stream Lines to Indicate the Particle Paths

This example uses stream lines (`stream3`, `streamline`) to trace the path of the animated particles. This adds a visual context for the animation. Another possibility is to set the `EraseMode` property of the stream particle to `none`, which would be useful for a single trace through the volume.

```
verts = stream3(x,y,z,u,v,w,sx,sy,sz);
sl = streamline(verts);
```

3. Define the View

While all of the stream lines start in the $z = 5$ plane, the values of some spiral down to lower values. The following settings provide a clear view of the animation:

- The viewpoint (`view`) selected shows both the plane containing most stream lines as well as the spiral.
- Selecting a data aspect ratio (`daspect`) of `[2 2 0.125]` provides greater resolution in the z -direction to make the stream particles more easily visible in the spiral.
- Set the axes limits to match the data limits (`axis`) and draw the axis box (`box`).

```
view(-10.5,18)
daspect([2 2 0.125])
axis tight; box on
```

4. Calculate the Stream Particle Vertices

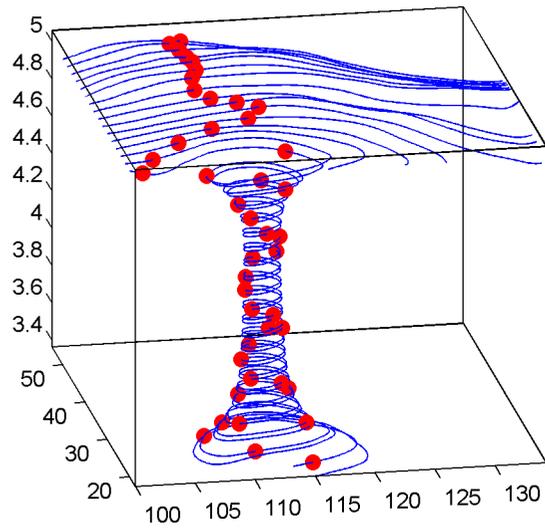
The first step is to determine the vertices along the stream line where a particle should be drawn. The `interpstreamspeed` function returns this data based on the stream line vertices and the speed of the vector data. This example scales the velocities 0.05 to increase the number of interpolated vertices.

Setting the axes `DrawMode` property to `fast` enables the animation to run faster.

The `streamparticles` function sets the following properties:

- `Animate` to 10 to run the animation 10 times
- `ParticleAlignment` to `on` to start all particle traces together
- `MarkerEdgeColor` to `none` to draw only the face of the circular marker. Animations usually run faster when marker edges are not drawn.
- `MarkerFaceColor` to `red`
- `Marker` to `o`, which draws a circular marker. You can use other line markers as well.

```
iverts = interpstreamspeed(x,y,z,u,v,w,verts,0.05);
set(gca,'drawmode','fast');
streamparticles(iverts,15,...
    'Animate',10,...
    'ParticleAlignment','on',...
    'MarkerEdgeColor','none',...
    'MarkerFaceColor','red',...
    'Marker','o');
```



Vector Field Displayed with Cone Plots

This example plots the velocity vector cones for the wind data. The graph produced employs a number of visualization techniques:

- An isosurface is used to provide visual context for the cone plots and to provide means to select a specific data value for a set of cones.
- Lighting enables the shape of the isosurface to be clearly visible.
- The use of perspective projection, camera positioning, and view angle adjustments compose the final view.

1. Create an Isosurface

Displaying an isosurface within the rectangular space of the data provides a visual context for the cone plot. Creating the isosurface requires a number of steps:

- Calculate the magnitude of the vector field, which represents the speed of the wind.
- Use `isosurface` and `patch` to draw an isosurface illustrating where in the rectangular space the wind speed is equal to a particular value. Regions inside the isosurface have higher wind speeds, regions outside the isosurface have lower wind speeds.
- Use `isonormals` to compute vertex normals of the isosurface from the volume data rather than calculate the normals from the triangles used to render the isosurface. These normals generally produce more accurate results.
- Set visual properties of the isosurface, making it red and without edges drawn (`FaceColor`, `EdgeColor`).

```
load wind
wind_speed = sqrt(u.^2 + v.^2 + w.^2);
hiso = patch(isosurface(x,y,z,wind_speed,40));
isonormals(x,y,z,wind_speed,hiso)
set(hiso,'FaceColor','red','EdgeColor','none');
```

2. Add Isocaps to the Isosurface

Isocaps are similar to slice planes in that they show a cross section of the volume. They are designed to be the end caps of isosurfaces. Using interpolated face color on an isocap causes a mapping of data value to color in the current

`colormap`. To create isocaps for the isosurface, define them at the same isovalue (`isocaps`, `patch`, `colormap`).

```
hcap = patch(isocaps(x,y,z,wind_speed,40),...
            'FaceColor','interp',...
            'EdgeColor','none');
colormap hsv
```

3. Create First Set of Cones

- Use `daspect` to set the data aspect ratio of the axes before calling `coneplot` so MATLAB can determine the proper size of the cones.
- Determine the points at which to place cones by calculating another isosurface that has a smaller isovalue (so the cones display outside the first isosurface) and use `reducepatch` to reduce number of faces and vertices (so there are not too many cones on the graph).
- Draw the cones and set the face color to blue and the edge color to none.

```
daspect([1,1,1]);
[f verts] = reducepatch(isosurface(x,y,z,wind_speed,30),0.07);
h1 = coneplot(x,y,z,u,v,w,verts(:,1),verts(:,2),verts(:,3),3);
set(h1,'FaceColor','blue','EdgeColor','none');
```

4. Create Second Set of Cones

- Create a second set of points at values that span the data range (`linspace`, `meshgrid`).
- Draw a second set of cones and set the face color to green and the edge color to none.

```
xrange = linspace(min(x(:)),max(x(:)),10);
yrange = linspace(min(y(:)),max(y(:)),10);
zrange = 3:4:15;
[cx,cy,cz] = meshgrid(xrange,yrange,zrange);
h2 = coneplot(x,y,z,u,v,w,cx,cy,cz,2);
set(h2,'FaceColor','green','EdgeColor','none');
```

5. Define the View

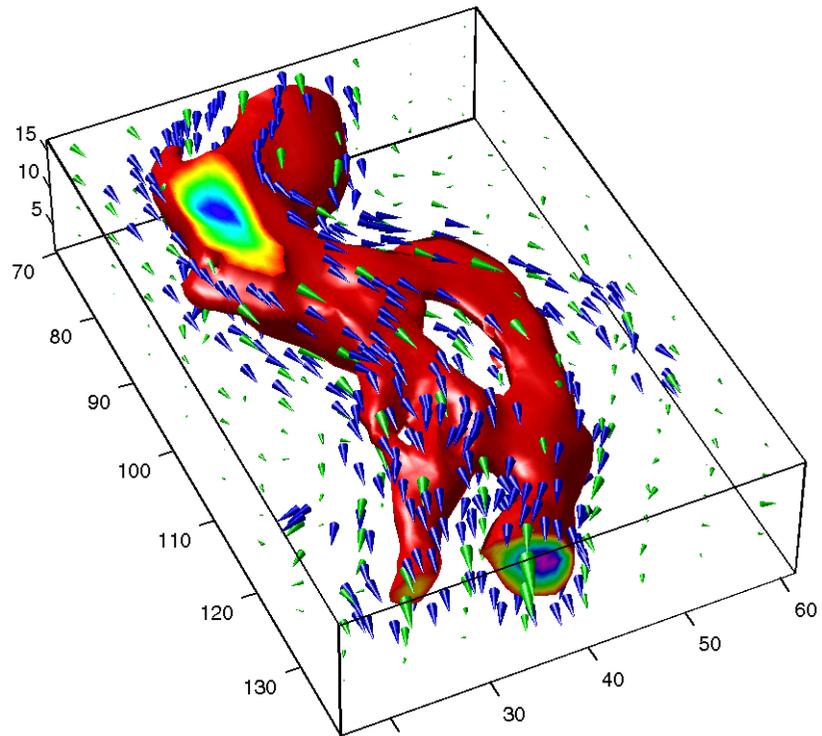
- Use the axis command to set the axis limits equal to the minimum and maximum values of the data and enclose the graph in a box to improve the sense of a volume (box).
- Set the projection type to perspective to create a more natural view of the volume. Set the view point and zoom in to make the scene larger (camproj, camzoom, view).

```
axis tight
box on
camproj perspective
camzoom(1.25)
view(65,45)
```

6. Add Lighting

Add a light source and use Phong lighting for the smoothest lighting of the isosurface (Phong lighting requires the zbuffer renderer). Increase the strength of the background lighting on the isocaps to make them brighter (camlight, lighting, AmbientStrength).

```
camlight(-45,45)
set(gcf,'Renderer','zbuffer');
lighting phong
set(hcap,'AmbientStrength',.6)
```



A

- alpha values 13-3
- ambient light 12-12
- AmbientLightColor property 12-8
 - illustration 12-12
- AmbientStrength property 12-8
 - illustration 12-12
- animation 4-50
 - erase modes for 4-52
 - movies 4-50
- annotating graphs 3-1
 - adding a legend 3-6
 - adding a title 3-3
 - adding labels 3-11
 - adding plots of data statistics 3-33
 - adding text 3-17
- annotations
 - adding 1-13, 3-18
- area 4-2, 4-11
- area graphs 4-2, 4-11
- arrays, storing images 5-4
- arrow annotations
 - adding 1-13
- arrows
 - adding to a graph 3-32
- aspect ratio 11-41–11-55
 - for realistic objects 11-54
 - properties that affect 11-46
 - specifying 11-50
- aspect ratio of figures 6-34, 6-36
 - See also* printing
- axes
 - adding labels 3-11
 - adding text 3-19
 - aspect ratio 11-41, 11-46
 - 2-D 2-23
 - 3-D 11-41
 - properties that affect 11-46
 - specifying 11-50
 - automatic modes 8-19
 - axis control 8-10
 - axis direction 8-13
 - camera properties 11-30
 - CLim property 8-25
 - color limits 8-25
 - ColorOrder property 8-30
 - colors 8-22
 - controlling the shape of 11-50
 - default aspect ratio 11-47
 - individual axis control 8-10
 - labeling 3-15
 - labels
 - font properties 8-4
 - using TeX characters 3-25
 - limits 11-41
 - example 11-53
 - locking position 1-7
 - making grids coincident 8-17
 - multi-axis 8-16
 - multiple 2-26, 8-7
 - NextPlot property 7-37
 - overlapping 8-7
 - plot box 11-8
 - position rectangle 11-31
 - positioning 8-5–8-9
 - preparing to accept graphics 7-36
 - properties
 - for labeling 8-3
 - properties of 1-8
 - protecting from output 7-42
 - scaling 11-41
 - independent 8-8
 - setting

- limits 8-11
 - line styles used for plotting 8-31
 - setting limits 2-19
 - standard plotting behavior 7-41
 - stretch-to-fill 11-41
 - target for graphics 2-28
 - tick marks 2-21
 - locating 8-12
 - units 8-6
 - unlocking position 1-7
 - with two x and y axes 8-16
- axis 5-2
- axis 11-41
- auto 11-41
 - equal 2-24, 11-42
 - ij 11-42
 - illustrated examples, 2-D 2-24
 - illustrated examples, 3-D 11-43
 - image 5-20, 11-42
 - manual 11-42
 - normal 11-42
 - square 2-24, 11-42
 - tight 2-25, 11-42
 - vis3d 11-42
 - xy 11-42
- axis labels, rotating 3-13
- azimuth of viewpoint 11-4
- default 2-D 11-5
 - default 3-D 11-5
 - limitations 11-7
- B**
- BackFaceLighting property 12-9
- illustration 12-14
- background color, of text 3-29
- backing store 9-15
- bar 4-2, 4-3
- bar graphs 4-2–4-11
- 3-D 4-3
 - grouped
 - 2-D 4-2
 - 3-D 4-4
 - horizontal 4-6
 - labeling 4-4, 4-7
 - overlaid with plots 4-9
 - stacked 4-5
- bar3 4-2, 4-3
- bar3h 4-2
- barh 4-2
- binary images 5-8
- bins, specifying for histogram 4-20
- BMP 5-2
- brighten 10-19
- buttons on toolbar 1-13, 1-14, 3-18, 3-32
- C**
- callbacks
- function handles used for 7-53
 - using function handles for 7-53
- camdolly 11-21
- camera position, moving 11-31
- camera properties 11-30
- illustration showing 11-8
- camera toolbar 11-9
- CameraPosition property 11-30
- and perspective 11-31
 - fly-by 11-31
- CameraPositionMode property 11-30
- CameraTarget property 11-30
- CameraTargetMode property 11-30
- CameraUpVector property 11-30, 11-34
- example 11-35

- CameraUpVectorMode property 11-30
- CameraViewAngle property 11-30
 - and perspective 11-33
 - zooming with 11-33
- CameraViewAngleMode property 11-30, 11-33
- camlookat property 11-21
- camorbit 11-21
- campan 11-21
- campos 11-21
- camproj 11-21
- camroll 11-21
- camtarget 11-21
- camup 11-21
- camva 11-21
- camzoom 11-21
- CData property
 - images 5-23
 - patches 14-11
- CDataMapping property 10-17
 - images 5-23
 - patches 14-11
- cla 7-37
- clabel 4-37, 4-39
- clf 7-37
- close 7-44
- close request function
 - default 7-44
- closereq.m 7-44
- CloseRequestFcn property 7-44
 - default value 7-44
 - errors in 7-45
 - overriding 7-45
- closing figures 7-44
- closing MATLAB, errors occurring when 7-45
- color limits, calculating 8-26
- color property of lights 12-4
- colorbar 10-15
- colordef 2-29
- colormap 10-14
- colormaps
 - altering 10-19
 - brightening 10-19
 - brightness component of TV signal 10-20
 - displaying 10-15
 - for surfaces 10-14
 - functions that create 10-15
 - large 9-9
 - minimum size 9-10
 - range of RGB values in 10-14
 - simulating multiple 8-25
 - size of dithermap 9-14
- ColorOrder 8-29
- colors
 - changing color scheme 8-23
 - colormaps 9-8, 10-14
 - controlled by axes 8-22
 - controlled by figure properties 9-7
 - dithering 9-12, 10-22
 - effects of dithering 9-14
 - fixed 9-8
 - indexed 10-13, 10-14
 - direct 10-16
 - scaled 10-16
 - indexed and dithering 9-12
 - interpreted by surfaces 10-13
 - mapping to data 8-25
 - NTSC encoding of 10-20
 - of patches 14-11
 - of surface plots 10-13
 - scaling algorithm 10-17
 - shared 9-11
 - size of dithermap 9-14
 - specifying Figure colors 2-28
 - specifying for surface plot, example 10-17

- truecolor 10-13
 - on indexed color systems 10-22
 - specifying 10-20
- typical RGB values 10-14
- used for plotting 8-29
- using a large number 9-9
- compass 4-31
- compass plots 4-31
- complex numbers, plotting 2-15
 - with feather 4-33
- cone plots 15-42
- contour 4-37
- contour plots 4-37
 - algorithm 4-42
 - filled 4-40
 - in polar coordinates 4-44
 - labeling 4-39
 - specifying contour levels 4-41, 4-43
- contour3 4-37
- contourc 4-37, 4-42
- contourf 4-37, 4-40
- conv2 5-13
- converting the data class of an indexed image 5-11
- convn 5-13
- coordinate system and viewpoint 11-4
- copying
 - figures 1-10
 - options 1-10
- copying graphics objects 7-32
- current
 - Axes 7-29
 - Figure 7-29
 - object 7-29
- cursors, see pointers

D

- data statistics
 - formatting plots of of 3-36
 - plotting 3-33, 3-36
- Data Statistics tool
 - example 3-34
 - interface 3-34
 - overview 1-22
 - saving to workspace 3-38
- data types
 - 8-bit integers 5-4
 - double-precision 5-4
- DataAspectRatio property 11-46
 - example 11-50
 - images 5-21
- DataAspectRatioMode property 11-46
- default
 - aspect ratio 11-47
 - of figure windows 6-34
 - azimuth
 - 2-D 11-5
 - 3-D 11-5
 - CameraPosition 11-31
 - CameraTarget 11-31
 - CameraUpVector 11-31
 - CameraViewAngle 11-31
 - CloseRequestFcn 7-44
 - elevation
 - 2-D 11-5
 - 3-D 11-5
 - factory 7-21
 - figure color scheme 2-28
 - Projection 11-31
 - property values 7-22-??
 - removing 7-24
 - search path, diagram 7-22
 - setting to factory defaults 7-25

- view 11-31
 - default line styles, setting and removing 2-11
 - de12 10-17
 - deleting graphics objects 7-34
 - deselecting objects 1-7
 - diffuse reflection 12-11
 - DiffuseStrength property 12-8
 - illustration 12-11
 - direct color mapping 10-16
 - direction cosines 11-34
 - discrete data graphs 4-22–4-30
 - stairstep plots 4-29
 - stem plots 4-22
 - dithering 9-12
 - algorithm 9-12
 - effects of 9-14
 - Dithermap property 9-12
 - DithermapMode property 9-12, 9-13
 - documentation 1-12
 - double 5-29
 - double
 - converting double to uint8 or uint16 5-11
 - converting image data to double 5-29
 - double buffering 9-15
 - double converting double to uint16 5-12
 - double converting double to uint8 5-12
- E**
- edge effects and lighting 12-15
 - EdgeColor property 12-9
 - EdgeLighting property 12-9
 - edges of patches 14-14
 - Edit menu
 - starting Property Editor 1-16
 - editing plots
 - interactively 1-5
 - efficient programming 7-47, 7-48
 - elevation of viewpoint 11-4
 - default 2-D 11-5
 - default 3-D 11-5
 - limitations 11-7
 - ending plot edit mode 1-6
 - erase modes 4-52
 - and printing 4-54
 - background 4-54
 - images 5-26
 - none 4-52
 - xor 4-55
 - errors closing MATLAB 7-45
 - examples
 - 3-D graph 10-2
 - animation 4-52
 - area graphs 4-11
 - axis 11-43
 - bar graphs 4-3
 - changing CameraPosition 11-32
 - contour plots 4-37
 - copying graphics objects 7-32
 - custom pointers 9-20
 - DataAspectRatio property 11-50
 - de12 10-17
 - direction and velocity graphs 4-31
 - direction cosines 11-34
 - discrete data graphs 4-22
 - displaying real objects 11-54
 - double axis graphs 8-16
 - finding objects handles 7-31
 - histograms 4-17
 - hold 7-42
 - line 7-39
 - linspace 10-9
 - meshgrid 10-3, 10-9
 - movies 4-51

- multiline text 3-28
 - newplot 7-39
 - object creation functions 7-12
 - of lighting 12-2
 - overlapping axes 8-7
 - parametric surfaces 10-11
 - pie charts 4-14
 - placing text dynamically 3-23
 - plot 2-2
 - complex data 2-15
 - plot3 10-3
 - PlotBoxAspectRatio property 11-51
 - plotting linestyles 8-31
 - ScreenSize property 9-5
 - setting default property values 7-25
 - simulating multiple colormaps 8-25
 - specifying figure position 9-5
 - specifying truecolor
 - surfaces 10-20
 - stretch-to-fill 11-50
 - subplot 2-26
 - text 3-18
 - texture mapping 10-23
 - unevenly sampled data 10-8
 - view 11-33
- exporting
- Enhanced Metafiles 6-64
- exporting figures 1-10
- Adobe Illustrator 6-66
 - EPS files 6-64
 - formats
 - choosing a format 6-57
 - MATLAB and GhostScript 6-59
 - vector or bitmap 6-59
 - JPEG files 6-66
 - LaTeX
 - importing example 6-19
 - lighting 6-62
 - publication quality 6-65
 - TIFF files 6-66
 - transparency 6-62
- extent of computer screen 9-4
- ## F
- FaceColor property 12-9
 - FaceLighting property 12-8
 - Faces property 14-7
 - FaceVertexCData property 14-9, 14-11
 - factory defaults 7-21
 - feather 4-31, 4-32
 - feather plots 4-32
 - fft2 5-13
 - fftn 5-13
 - figure files 1-9
 - figures
 - CloseRequestFcn 7-44
 - closing 7-44
 - copying 1-10
 - defining custom pointers 9-19
 - defining pointers 9-18
 - defining the color of 2-28
 - fixed colors 9-8
 - for plotting 2-26
 - index color properties 9-7
 - introduction to 9-2
 - NextPlot property 7-37
 - nonactive 9-11
 - opening 1-9
 - positioning 9-3
 - positioning example 9-5
 - preparing to accept graphics 7-36
 - printing
 - default figure size for printing 6-34

- protecting from output 7-42
 - rendering properties 9-15
 - saving 1-9
 - specifying pointers 9-18
 - standard plotting behavior 7-41
 - units 9-4
 - visible property 7-44
 - with multiple axes 2-26
- files
- exporting 1-10
 - figure `fig` 1-9
 - formats for figures 1-10
 - opening 1-9
 - printing 1-10
 - saving 1-9
- `fill`, properties changed by 7-48
- `fill3`, properties changed by 7-48
- `findobj` 7-31
- fixed colors 9-8
- `FixedColors` property 9-8
- Floyd-Steinberg dithering algorithm 9-12
- fly-by effect 11-31
- fonts
- axis labels 8-4
- formats for figures 1-10
- function handles
- Handle Graphics callbacks 7-53
- functions
- convenience forms 7-15
 - high-level vs. low-level 7-14
 - to create graphics objects 7-11
- G**
- `gca` 7-30
- handle visibility 7-43
- `gcf` 7-30
- handle visibility 7-43
- `gco` 7-30
- `get` 7-17
- `getframe` 4-50
- GIF graphic file format 6-66
- `ginput` 4-48
- Gouraud lighting algorithm 12-10
- gradient 4-35
- graphical input 4-48
- graphics
- M-files, structure of 7-40
- graphics file formats
- list of formats supported by MATLAB 5-2
- graphics images
- 16-bit
 - intensity 5-12
 - 8-bit
 - intensity 5-12
 - RGB 5-12
 - converting from one format to another 5-29
 - converting to RGB 5-29
 - reading from file 5-15
 - see also BMP, HDF, JPG, PCX, PNG, TIFF, XWD 5-16
 - writing to file 5-16
- graphics objects 7-3
- accessing handles 7-29
 - accessing hidden handles 7-43
 - axes 7-6
 - controlling where they draw 7-36
 - copying 7-32
 - deleting 7-34
 - editing properties 1-8, 1-17
 - figures 7-4
 - function handle callbacks 7-53
 - functions that create 7-11
 - convenience forms 7-15

- handle validity versus visibility 7-45
- HandleVisibility property 7-42
- hierarchy 7-2
- images 7-6
 - See also Image chapter
- invisible handles 7-42
- lights 7-6
- line 7-6
- patches 7-7
- properties 7-8
 - changed by functions 7-48
 - changed when created 7-13
 - common to all objects 7-9
 - factory defined 7-21
 - getting current values 7-19
 - listing possible values 7-18
 - querying in groups 7-21
 - search path for default values 7-22
 - searching for 7-31
 - setting values 7-17
- property names 7-15
- rectangle 7-7
- root 7-4
- setting parent of 7-14
- specifying tag value 1-20
- surface 7-7
- text 7-7
- uicontrol 7-4
- uimenu 7-5

graphs

- area 4-11–4-13
- bar 4-2–4-11
 - horizontal 4-6
- compass plots 4-31
- contour plots 4-37–4-46
- direction and velocity 4-31–4-36
- discrete data 4-22–4-30

- feather plots 4-32
- histograms 4-17–4-21
- labeling 3-1
- pie charts 4-14–4-16
- quiver plots 4-34
- stairstep plots 4-29
- steps to create 3-D 10-2
- with double axes 8-16

grayscale 5-17

- see also* intensity images 5-17

Greek characters

- see* text function
- using to annotate 3-15

griddata 10-9

grids, coincident 8-17

H

Hadamard matrix 10-11

Handle Graphics

- graphics objects 7-3
- hierarchy of graphics objects 7-2

handles to graphics objects 7-29

- finding 7-31

handles, saving in M-files 7-47

HandleVisibility property 7-42

HDF 5-2

help 1-12

hidden 10-12

hidden line removal 10-12

high-level functions 7-14

hist 4-17

histograms 4-17

- in polar coordinates 4-19
- labeling the bins 4-20
- rose plot 4-19
- specifying number of bins 4-20

hold 2-7
 and NextPlot 7-38
 testing state of 7-41
hold state, testing for 7-41
HorizontalAlignment property 3-22

I

image 5-2
image 5-19
 properties changed by 7-49
image types
 binary 5-8
images
 16-bit 5-11
 indexed 5-11
 8-bit 5-11
 indexed 5-11
 data types 5-4
 erase modes 5-26
 indexed 5-6
 information about files 5-16
 intensity 5-7
 numeric classes 5-2
 printing 5-28
 properties 5-23
 CData 5-23
 CDataMapping 5-23
 XData and YData 5-24
 RGB 5-9
 see also graphics images 5-12
 size and aspect ratio 5-19
 storing in MATLAB 5-4
 truecolor 5-9
 types 5-6
imagesc 5-2
imagesc 5-8

imagesc 5-8
imfinfo 5-3
imfinfo 5-16
imread 5-2
imread 5-15
imwrite 5-3
imwrite 5-16
ind2rgb 5-29
indexed color
 displays 9-7
 dithering truecolor 9-12
 surfaces 10-13
indexed images
 converting the data class of 5-11
indexed_color_surfaces 10-16
indrgb 5-3
Infs, avoiding in data 10-6
intensity images
 converting the data class of 5-12
interpolated colors
 patches 14-9
 indexed vs. truecolor 14-19
interpreter property 3-26
ishold 7-41
isosurface
 illustrating flow data 15-20

J

JPEG 5-2

L

labeling
 axes 3-11
labeling graphs 3-1, 3-11
Laplacian of a matrix 10-17

- LaTeX. See TeX
 - legend
 - adding to a graph 3-6
 - editing text labels 3-9
 - positioning in a graph 3-8
 - legend 4-25
 - legends
 - with data statistics 3-35
 - light 12-4
 - lighting 12-2–12-18
 - algorithms
 - flat 12-10
 - Gouraud 12-10
 - Phong 12-10
 - ambient light 12-12
 - backface 12-14
 - diffuse reflection 12-11
 - important properties 12-4
 - properties that affect 12-8
 - reflectance characteristics 12-11–12-14
 - specular
 - color 12-14
 - exponent 12-13
 - reflection 12-11
 - lighting command 12-10
 - limits
 - axes 2-19, 8-11
 - line styles
 - used for plotting 2-5
 - redefining 8-31
 - lines
 - adding as annotations 3-32
 - adding to existing graph 2-7
 - annotations 1-13
 - marker types 2-5
 - properties of 1-8
 - removing hidden 10-12
 - styles 2-5
 - LineStyleOrder property 8-31
 - linspace 10-8
 - locking axes position 1-7
 - loglog, properties changed by 7-49
 - low-level functions 7-14
- ## M
- mapping data to color 8-25
 - markers used for plotting 2-5
 - material command 12-11
 - mathematical functions
 - visualizing with surface plot 10-6
 - MATLAB 4 color scheme 2-29
 - MATLAB, quitting 7-45
 - matrix
 - displaying contours 4-38
 - Hadamard 10-11
 - plotting 2-13
 - representing as
 - area graph 4-11
 - bar graph 4-3
 - histogram 4-18
 - surface 10-5
 - storing images 5-4
 - mesh 10-5
 - meshc 4-43
 - meshgrid 10-6
 - M-files
 - basic structure of graphics 7-40
 - closereq 7-44
 - to set color mapping 8-28
 - using newplot 7-38
 - writing efficient 7-47
 - min
 - plotting 3-33

MinColormap property 9-9

movie 4-50, 4-51

movies 4-50

 example 4-51

moving

 objects 1-7

MRI data, visualizing 15-9

multiaxis axes 8-16

multiline text 3-28

N

navigation bar

 in Property Editor 1-18

newlink aspectratio 6-34

newplot 7-38

 example using 7-39

NextPlot property 7-37

 add 7-37

 replace 7-37

 replacechildren 7-37, 7-41

 setting plotting color order 8-30

nonuniform data, plotting 10-8

NormalMode property 12-9

NTSC color encoding 10-20

O

object properties

 editing 1-15

online help 1-12

open 1-9, 1-10

OpenGL 9-16, 9-17

 printing 6-43

opening figures 1-9

options for copying 1-10

organization of Handle Graphics 7-2

orient

 example 6-41

orthographic projection 11-36

 and Z-buffer 11-38

P

page setup 1-10

painters algorithm 9-16

paper type

 setting from the command line 6-39

paper type for printing

 setting from the command line 6-39

PaperPosition property

 example 6-37

PaperType property

 example 6-39

parametric surfaces 10-10

parent, of graphics object 7-14

patch

 behavior of function 14-2

 interpreting color 14-3

patches

 coloring 14-11

 edges 14-13

 face coloring

 flat 14-8

 interpolated 14-9

 indexed color 14-16

 direct 14-17

 scaled 14-16

 interpreting color data 14-16

 multifaceted 14-6

 single polygons 14-4

 specifying faces and vertices 14-7

 truecolor 14-19

 ways to specify 14-2

- PCX 5-2
- perspective projection 11-36
 - and Z-buffer 11-38
- Phong lighting algorithm 12-10
- pie charts 4-14
 - labeling 4-15
 - offsetting a slice 4-14
 - removing a piece 4-16
- plot 2-2
 - properties changed by 7-49
- plot box 11-8
- plot edit mode, starting and ending 1-6
- plot editing mode
 - overview 1-5
- plot3 10-3
 - properties changed by 7-50
- PlotBoxAspectRatio property 11-46
 - example 11-51
- PlotBoxAspectRatioMode property 11-46
- plottedit 1-6
- plots
 - editing object in 1-17
- plotting
 - 3-D
 - matrices 10-3
 - vectors 10-3
 - adding to existing graph 2-7
 - annotating graphs 3-1
 - area graphs 4-11
 - bar graphs 4-2
 - compass plots 4-31
 - complex data 2-15
 - contour plots 4-37
 - contours, labeling 4-39
 - creating a plot 2-2
 - data-point markers 2-5
 - elementary functions for 2-2
 - feather plots 4-32
 - interactive 4-48
 - line colors 8-29
 - line styles 2-5
 - matrices 2-13
 - multiple graphs 2-3
 - nonuniform data 10-8
 - overlying bar graphs 4-9
 - quiver plots 4-34
 - specifying line styles 2-4, 8-31
 - stairstep plots 4-29
 - stem plots 4-22
 - surfaces 10-5
 - to subaxis 2-26
 - vector data 2-2
 - windows for 2-26
- plotting statistics 3-36
- PNG 5-2
 - writing as 16-bit using imwrite 5-16
- Pointer property 9-19
- pointers
 - custom 9-19
 - example defining 9-20
 - specifying 9-18
- PointerShapeCData property 9-19
- PointerShapeHotSpot property 9-19
- polar 4-46
- polar coordinates
 - contour plots 4-44
 - rose plot 4-19
- polygons, creating with patch 14-2
- position of figure 9-3
- Position property
 - axes 8-5
 - figure 9-3
- position rectangle 11-8
- positioning axes 1-7

- positioning of axes 8-5
 - positioning text on a graph 3-19
 - preferences 1-10
 - printing
 - 3-D scenes 11-39
 - aspect ratio 6-34
 - default 6-34
 - background color 6-49
 - figure size
 - setting from the command line 6-33, 6-37, 6-77
 - fonts
 - supported for HPGL 6-71
 - supported for PostScript and GhostScript 6-71
 - supported for Windows drivers 6-71
 - images 5-28
 - MATLAB printer driver
 - definition 6-68
 - OpenGL 6-43
 - paper type
 - setting from the command line 6-39
 - PaperType property
 - example 6-39
 - PostScript
 - fonts supported for 6-71
 - quick start 6-23
 - renderer
 - methods 6-42
 - resolution
 - with painters renderer 9-17
 - with Z-buffer renderer 9-17
 - troubleshooting 6-78
 - Z-buffer 9-17
 - printing figures 1-10
 - Projection property 11-30
 - projection types 11-36–11-40
 - camera position 11-37
 - orthographic 11-36
 - perspective 11-36
 - rendering method 11-38
 - properties
 - automatic axes 8-19
 - changed by built-in functions 7-48
 - changed by object creation functions 7-13
 - defining in startup.m 7-28
 - editing 1-8, 1-15
 - for labeling axes 8-3
 - naming convention 7-15
 - See also graphics objects
 - specifying default values 7-24
 - Property Editor
 - closing 1-17
 - editing multiple objects 1-18
 - interface 1-15
 - overview 1-15
 - searching for objects 1-20
 - starting 1-16
 - using the navigation bar 1-18
 - property values
 - defaults 7-22
 - defined by MATLAB 7-21
 - getting 7-17
 - resetting to default 7-24
 - setting 7-17
 - specifying defaults 7-24
 - user defined 7-22
 - pseudocolor displays, see indexed color
- Q**
- quiver 4-31, 4-34
 - quiver plots 4-34
 - 2-D 4-34

- 3-D 4-35
 - combined with contour plot 4-35
 - displaying velocity vectors 4-36
- quiver3 4-31

R

- realism, adding with lighting 12-2
- realistic display of objects 11-54
- reflection, specular and diffuse 12-11
- Renderer property 10-22
 - and printing 9-17
- RenderMode property 10-22
- rendering
 - options 9-15
 - Z-buffer 9-16
- reset 7-37
- resizing
 - objects 1-7
- RGB
 - color values 10-14
 - converting to 5-29
 - images 5-9
 - converting the data class of 5-12
- rgbplot 10-19
- rose 4-17, 4-19
- rotating a plot 1-14
- rotating axis labels 3-13
- rotation
 - about viewing axis 11-34
 - without resizing 11-33

S

- saveas 1-9
- saving figures 1-9
- scaled color mapping 10-16

- screen extent, determining 9-4
- ScreenSize property 9-4
 - example 9-5
- selecting multiple objects 1-6
- selection button 1-6
- semilogx, properties changed by 7-50
- semilogy, properties changed by 7-50
- set 7-17
- ShareColors property 9-11
- ShowHiddenHandles property 7-43
- size of computer screen 9-4
- slice planes
 - colormapping 15-18
 - slicing a volume 15-15
- specular
 - color 12-14
 - exponent 12-13
 - highlight 12-13
 - reflection 12-11
- SpecularColorReflectance property 12-8
 - illustration 12-14
- SpecularExponent property 12-8
 - illustration 12-13
- SpecularStrength property 12-8
 - illustration 12-11
- sphere 10-23
- spline 4-48
- stairs 4-22, 4-29
- stairstep plot 4-29
- starting plot edit mode 1-6
- starting points for stream plots 15-28
- statistics
 - formatting plots of 3-36
 - plotting 3-33, 3-36
 - saving to workspace 3-38
- stem 4-22
- stem plots 4-22

- 3-D 4-26
 - overlaid with line plot 4-25
- stem3 4-22, 4-26
- stream line plots 15-32
- stream plots
 - starting points 15-28
- stretch-to-fill 11-41
 - overriding 11-49
- string variable, in text 3-27
- style property of lights 12-4
- subplot 2-26
- sum 5-13
- surf 10-5
- Surfaces
 - CData 10-23
 - coloring 10-13
 - curvature mapped to color 10-17
 - FaceColor, texturemap 10-23
 - parametric 10-10
 - plotting 10-5
 - nonuniformly sampled data 10-8
 - surf 4-43
 - symbols, TeX characters 3-25
- T**
- tag
 - specifying value of 1-20
- TeX
 - available characters 3-25
 - creating mathematical symbols 3-25
 - symbols in text 3-15, 3-26
- text
 - adding to Axes 3-18, 3-25
 - annotating graphs 3-18
 - annotations, adding 1-13
 - editing 3-24
 - for labeling plots 3-18
 - horizontal and vertical alignment 3-22
 - multiline 3-28
 - placing dynamically, example 3-23
 - placing outside of axes 8-7
 - positioning 3-21
 - TeX characters 3-26
 - using variables in 3-27
- texture mapping 10-23
- thin line styles 6-80
- three-dimensional objects, creating with patch
 - 14-2
- tick marks, on axes 2-21, 8-12
- TIFF 5-2
- title
 - adding to a graph 3-3
- toolbar
 - buttons 1-13, 1-14, 3-18, 3-32
- toolbar, camera 11-9
- truecolor
 - dithering on indexed systems 9-12
 - patches 14-19
 - rendering method used for 10-22
 - simulating 10-22
 - surface plots 10-13, 10-20
- U**
- Uicontrol graphics objects 7-4
- Uimenu graphics objects 7-5
- uint16 arrays
 - operations supported on 5-13
 - storing images 5-5
- uint16 arrays
 - converting uint16 to double 5-11, 5-12
- uint8 arrays 5-11
 - operations supported on 5-13

- storing images 5-5
- uint8 arrays
 - converting to double 5-12
 - converting uint8 to double 5-11
- units
 - axes 8-6
 - used by figures 9-4
- unlocking axes position 1-7
- unselecting objects 1-7

V

- vectors
 - determined by direction cosines 11-34
 - displaying velocity 4-36
- velocity vectors displayed with quiver 4-36
- vertex normals and back face lighting 12-15
- VertexNormals property 12-9
- VerticalAlignment property 3-22
- Vertices property 14-7
- view
 - azimuth of viewpoint 11-4
 - camera properties 11-30
 - coordinate system defining 11-4
 - definition of 11-2
 - elevation of viewpoint 11-4
 - limitation of azimuth and elevation 11-7
 - MATLAB's default behavior 11-31
 - projection types 11-36
 - rotating 1-14
 - specifying 11-30
 - specifying with azimuth and elevation 11-4
- view 11-4
 - example of rotation 11-33
 - limitations using 11-7
- viewing axis 11-8
 - moving camera along 11-32

- viewpoint, controlling 11-4, ??-11-7
- visibility of graphics objects 7-45
- visualizing
 - commands for volume data 15-5
 - mathematical functions 10-6
 - steps for volume data 15-4
 - techniques for volume data 15-3
- volume data
 - accessing subregions 15-30
 - examples of 15-3
 - MRI 15-9
 - scalar 15-8
 - slicing with plane 15-15
 - steps to visualize 15-4
 - techniques for visualizing 15-3
 - vector 15-27
 - visualizing 15-3

W

- wire frame surface 10-5, 10-12

X

- XWD 5-2

Z

- Z-buffer 9-16
 - orthographic projection 11-38
 - perspective projection 11-38
 - printing 9-17
 - rendering truecolor 10-22
- zooming by setting camera angle 11-33