MATLAB®

The Language of Technical Computing

Computation

Visualization

Programming

The MATH WORKS Inc.

Using MATLAB Graphics

Version 5.3

How to Contact The MathWorks:

508-647-7000	Phone

508-647-7001	Fax

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

`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`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
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Using MATLAB Graphics

Printing History:			
	December 1996	First printing	New for 5.0
	June 1997	Revised for 5.1	(Online version)
	January 1998	Revised for 5.2	(Online version)
	January 1999	Revised for 5.3	Release 11 (Online version)

Contents

1 Introduction

2 Basic Plotting

3 Labeling Graphs

4 Specialized Graphs

5 Creating 3-D Graphs

6 Defining the View

7 Lighting as a Visualization Tool

8 Volume Visualization Techniques

9 Creating 3-D Models with Patches

10 Displaying Bit-Mapped Images

11 Printing MATLAB Graphics

12 Handle Graphics

13 Figure Properties

14 Axes Properties

1

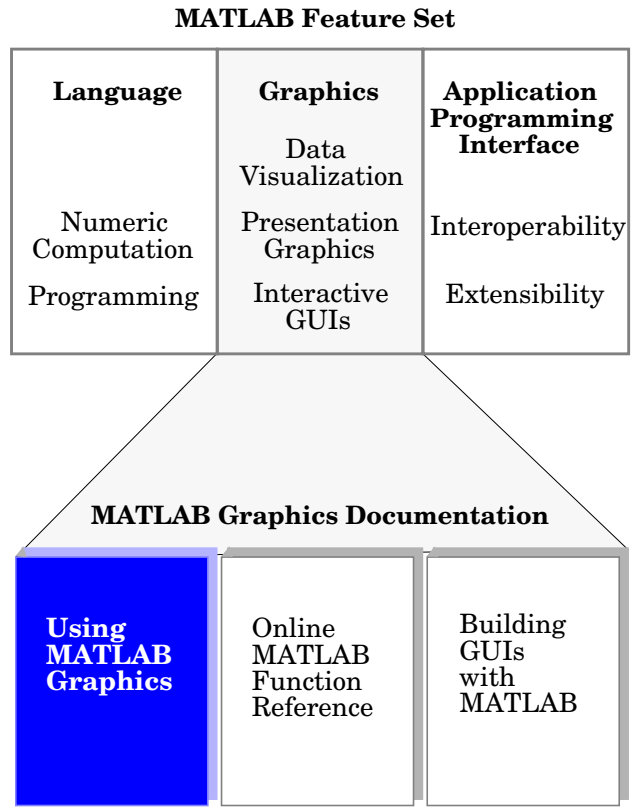
Introduction

Graphing Data with MATLAB

MATLAB provides extensive functionality for graphing a wide variety of data. This online manual is a collection of topics that explore all aspects of graphing with MATLAB, including plots, presentation graphics, and advanced visualization. The following table directs you to the topic areas that are most relevant to specific tasks.

Task	Relevant Topic Areas
Displaying vector data as line plots, bar graphs, histograms, and other specialized graphs.	Basic Plotting Specialized Graphs
Displaying surface plots of matrix data, visual representation of mathematical functions.	Creating 3-D Graphs Defining the View Lighting as a Visualization Tool
Displaying volume data	Volume Visualization Techniques Defining the View Lighting as a Visualization Tool
Displaying 3-D modeling data, creating objects with polygons.	Patch Objects for 3-D Modeling Defining the View Lighting as a Visualization Tool
Displaying bit-mapped images	Displaying Bit-Mapped Images
Labeling axes, adding text annotations, printing and exporting to files	Labeling Graphs Printing MATLAB Graphics
Programming graphics M-files	Handle Graphics Figure Properties Axes Properties

This manual presents the visualization component of the MATLAB documentation set, as is illustrated in the following picture.



High-Level Graphics

MATLAB provides a set of high-level graphing routines. These routines implement commonly used techniques for displaying data, such as line plots in rectangular and polar coordinates, bar and histogram graphs, contour plots, mesh and surface plots, and animation. In addition, you can control color and shading, axis labeling, and the general appearance of graphs. High-level commands automatically control plot characteristics such as axis scaling and

line color to produce acceptable graphs without requiring you to manipulate low-level properties.

Handle Graphics

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

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

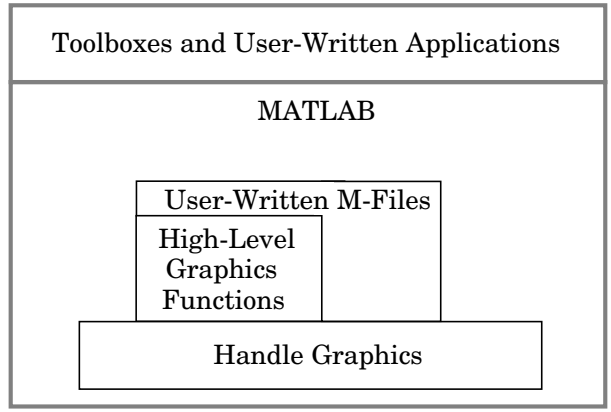
Building Interactive GUIs

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

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

How It All Fits Together

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



Where to Begin

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

2

Basic Plotting

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

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

Step	Typical Code

1 Prepare your data	`x = 0:0.2:12;`
	`y1 = bessel(1,x);`
	`y2 = bessel(2,x);`
	`y3 = bessel(3,x);`

2 Select window and position plot region within window	`figure(1)` `subplot(2,2,1)`
3 Call elementary plotting function	`h = plot(x,y1,x,y2,x,y3);`
4 Select line and marker characteristics	`set(h,'LineWidth',2,{'LineStyle'},{'--';':';'-.'})` `set(h,{'Color'},{'r';'g';'b'})`
5 Set axis limits, tick marks, and grid lines	`axis([0 12 –0.5 1])` `grid on`

6 Annotate the graph with axis labels, legend, and text	`xlabel('Time')`
	`ylabel('Amplitude')`
	`legend(h,'First','Second','Third')`
	`title('Bessel Functions')`
	`[y,ix] = min(y1);`
	`text(x(ix),y,'First Min \rightarrow',...`
	`'HorizontalAlignment','right')`

7 Export graph	`print –depsc -tiff -r200 myplot`

The following sections describe elementary plotting and provide examples of the options available. See Printing MATLAB Graphics for information on printing your graph.

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	Used to Create
`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

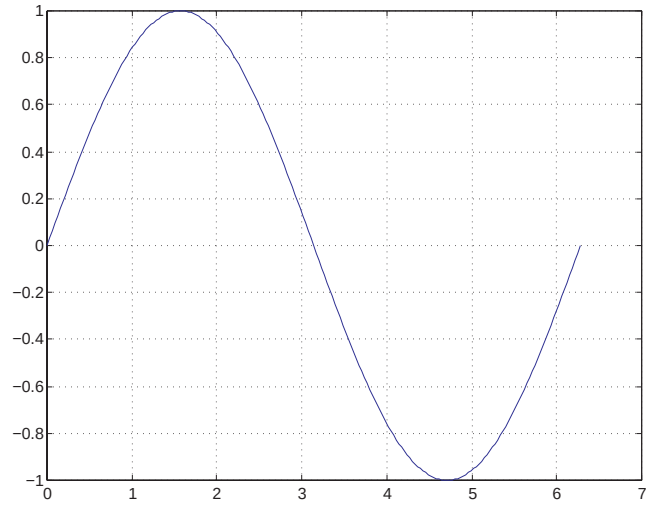
Creating Plots

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, 2π] in increments of π/100 and then use this vector to evaluate the sine function over that range. MATLAB plots the vector on the *x*-axis and the value of the sine function on the *y*-axis.

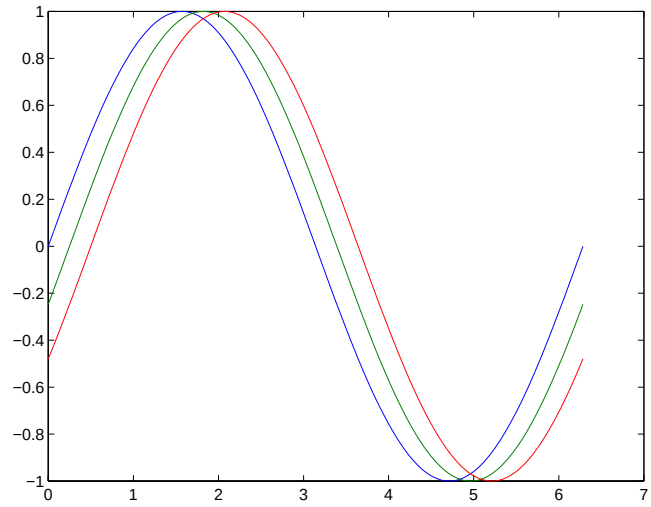
```
t = 0:pi/100:2*pi;
y = sin(t);
plot(t,y)
grid on
```

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

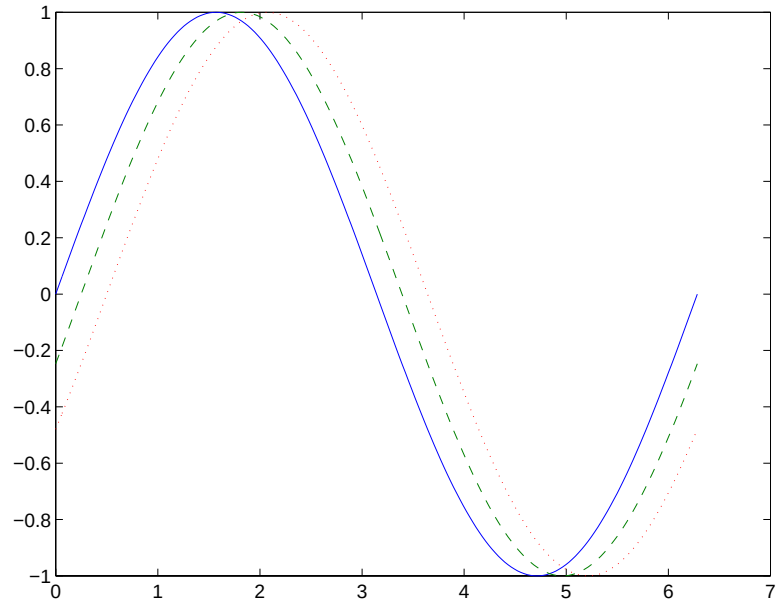
```
y2 = sin(t–0.25);
y3 = sin(t–0.5);
plot(t,y,t,y2,t,y3)
```



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 `LineSpec` discussion for more information.

Specifying the Color and Size of Lines

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

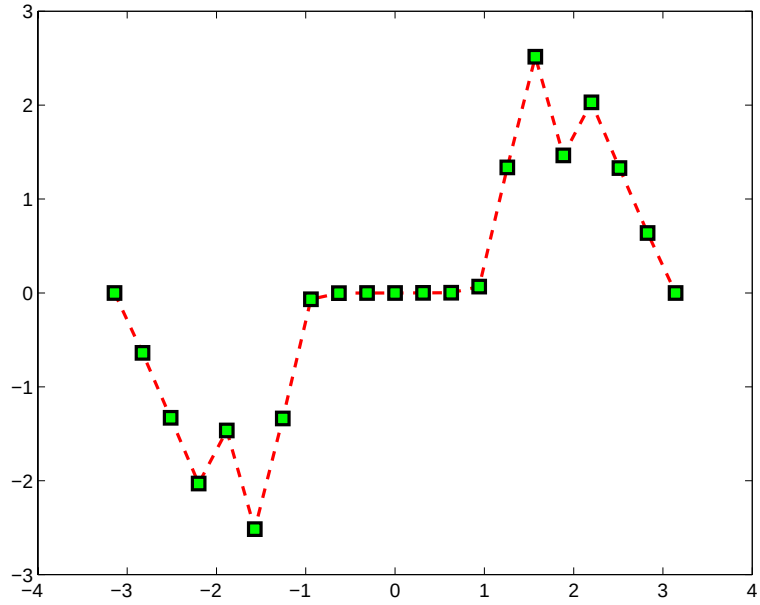
- `LineWidth` – specifies the width of the line in units of points.
- `MarkerEdgeColor` – specifies the color of the marker or the edge color for filled markers (circle, square, diamond, pentagram, hexagram, and the four triangles).
- `MarkerFaceColor` – specifies the color of the face of filled markers.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



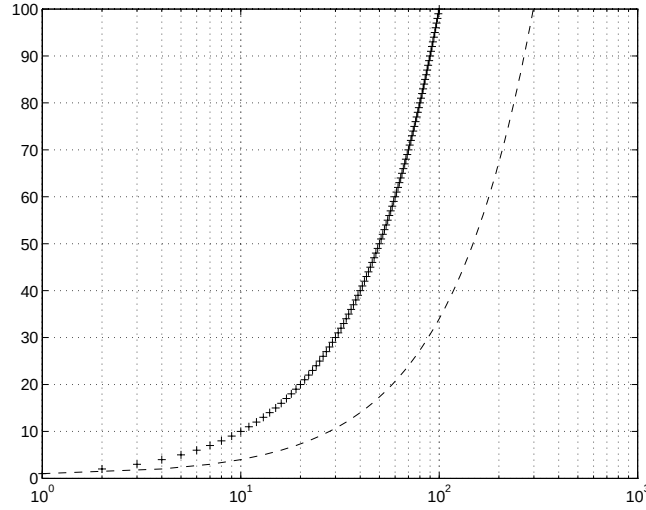
Adding Plots to an Existing Graph

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

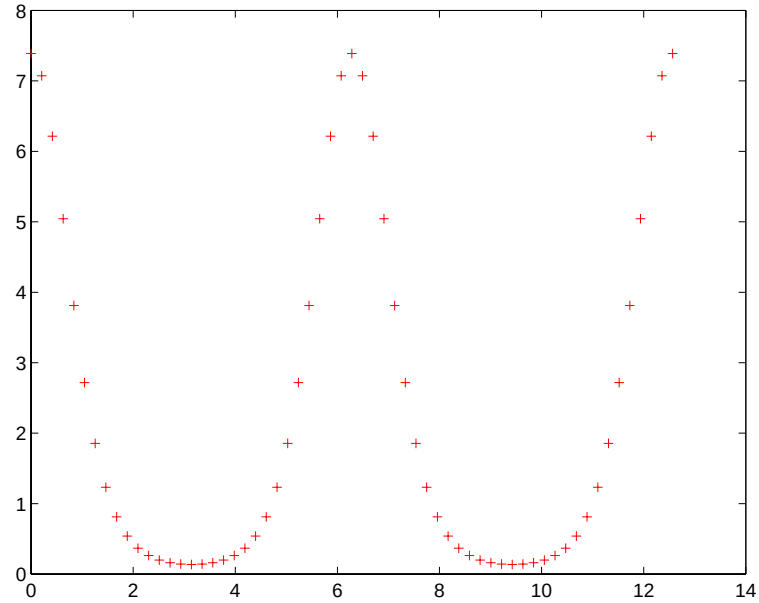
To plot a marker at each data point without connecting the markers with lines, use a specification that does not contain a line style. For example, given two vectors,

```
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:

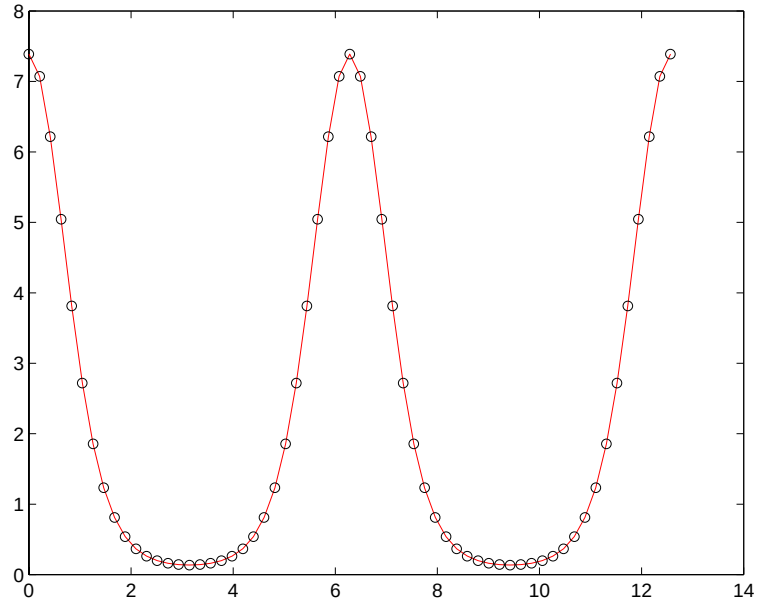


See `LineSpec` for a list of available line styles, markers, and colors.

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.

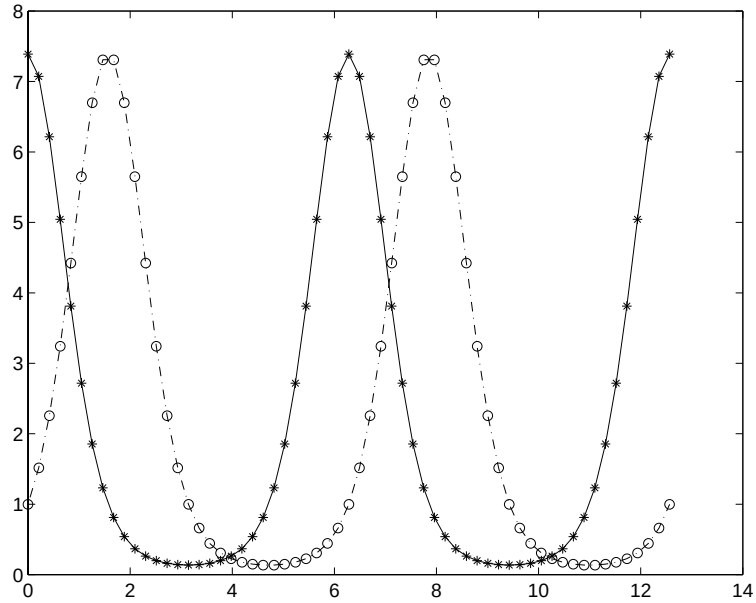
```
x = 0:pi/15:4*pi;
y = exp(2*cos(x));
plot(x,y,'-r',x,y,'ok')
```



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 `LineStyle` property. For example, the command,

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

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

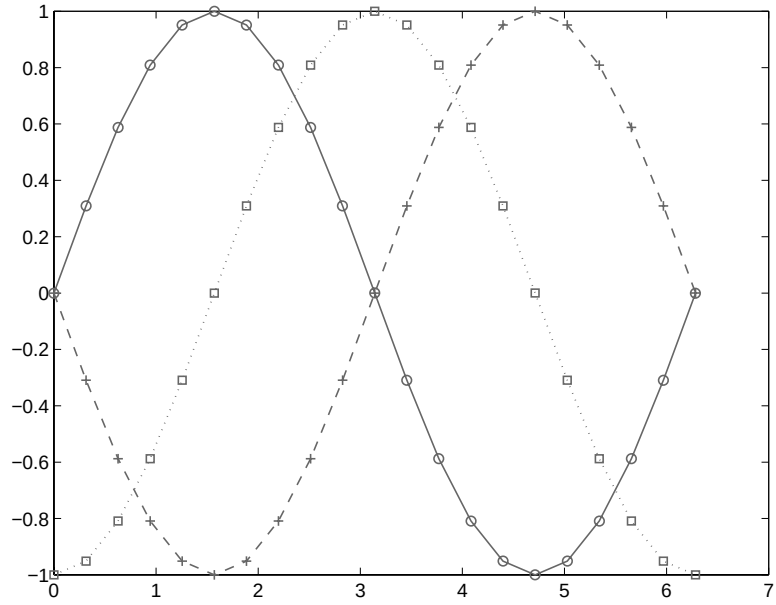
To set the default line color to dark gray, use the statement

```
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 multi-line 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 for more information.

Line Plots of Matrix Data

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

```
plot(Y)
```

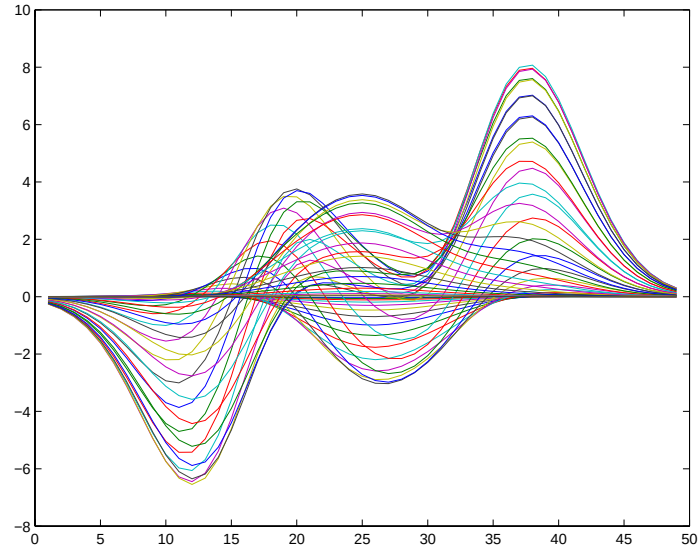
MATLAB draws one line for each column of the matrix. The *x*-axis is labeled with the row index vector, `1:m`, where `m` is the number of rows in `Y`. For example,

```
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.



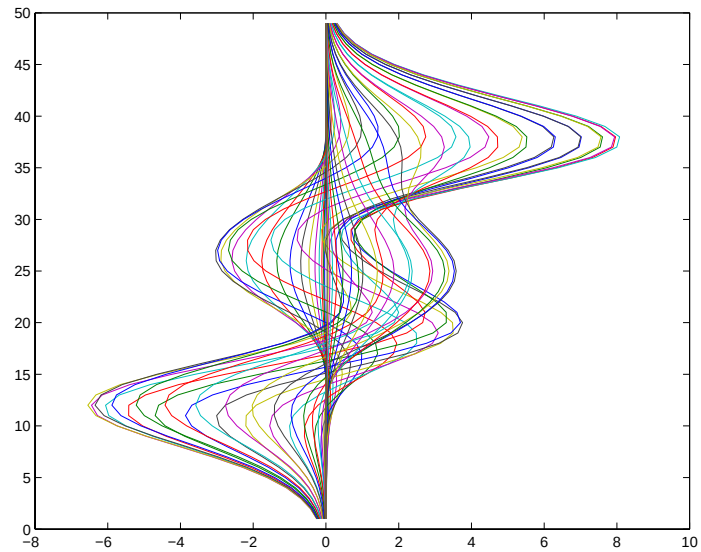
In general, if `plot` is used with two arguments and if either `X` or `Y` has more than one row or column, then

- 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,

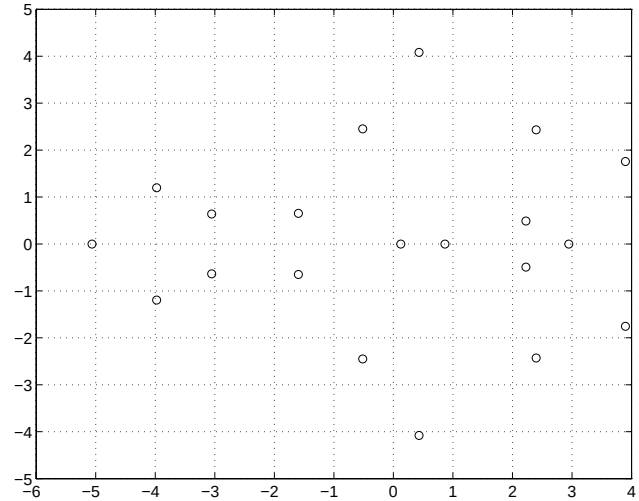
```
plot(Z)
```

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)
```

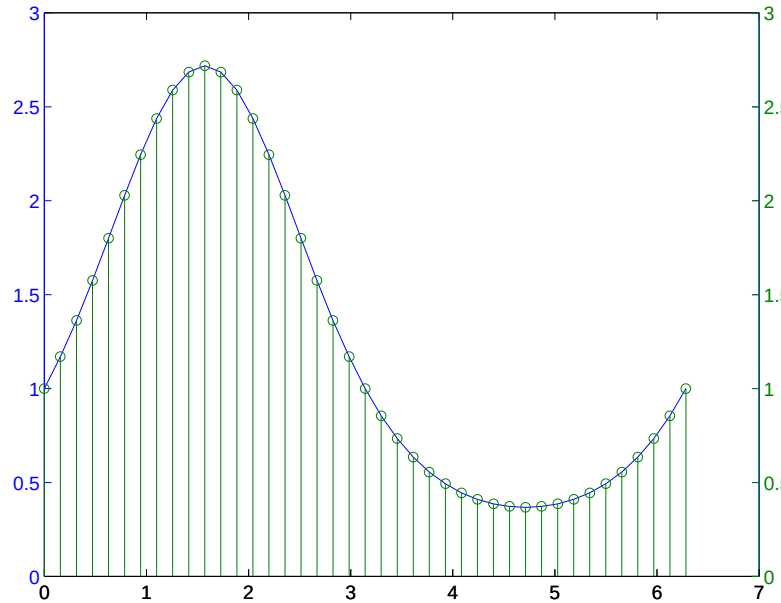


To plot more than one complex matrix, there is no shortcut; the real and imaginary parts must be taken explicitly.

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:

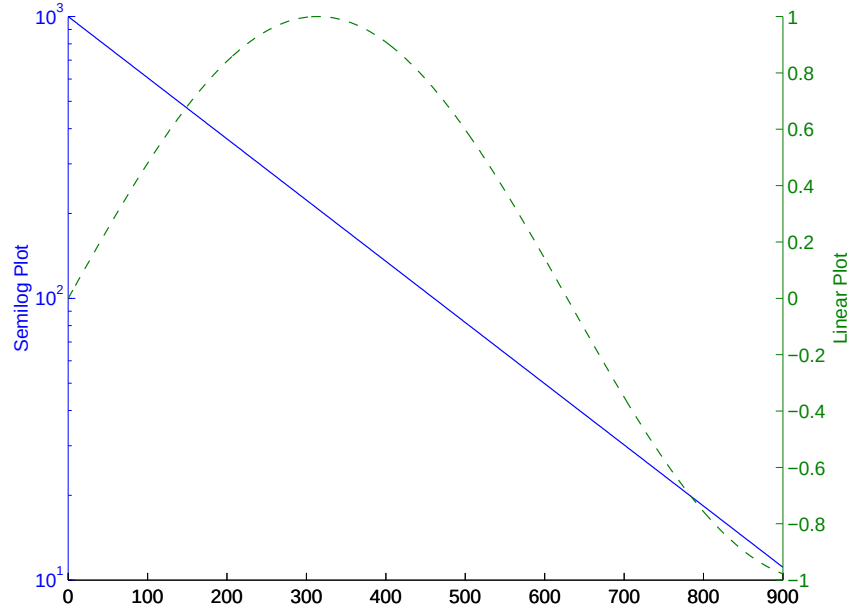
```
axes(haxes(1))
ylabel('Semilog Plot')
```

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 multi-axes plots for an example that employs double *x*- and *y*-axes.

See `LineSpec` for additional line properties.

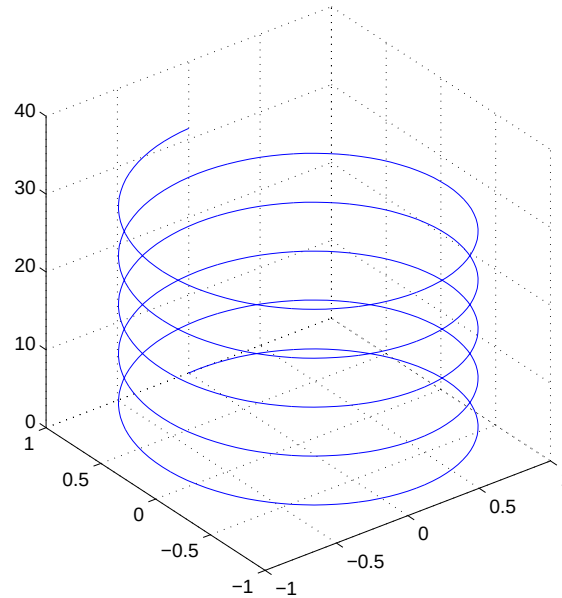
Line Plots of 3-D Data

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
```

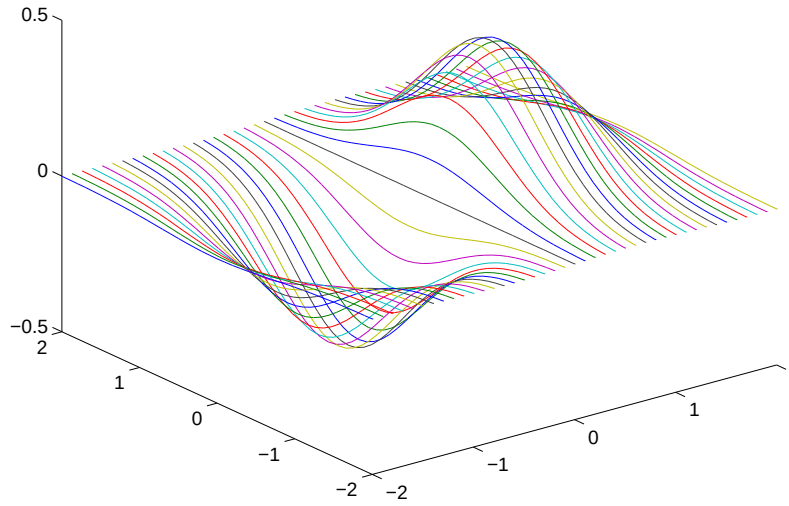


Plotting Matrix Data

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.



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 specify your own values for axis limits and tick marks by overriding MATLAB's values. 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.

Related Information

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

Axis Limits and Ticks

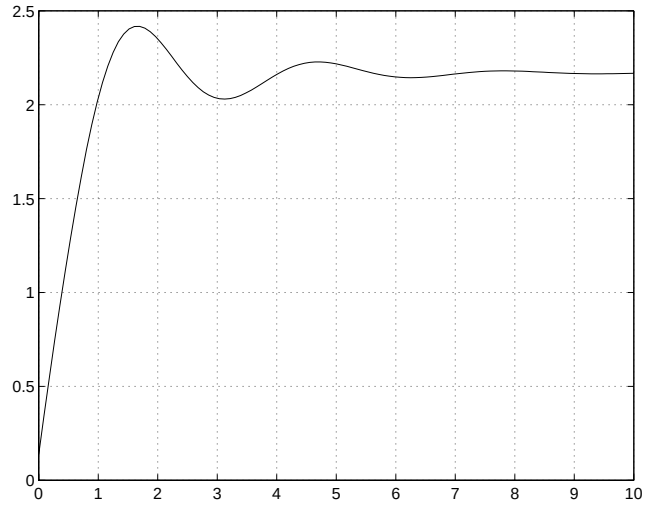
MATLAB selects axis limits based on the range of the plotted data. You can specify the limits manually using the `axis` command. Call `axis` with the new limits defined as a four-element vector.

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

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

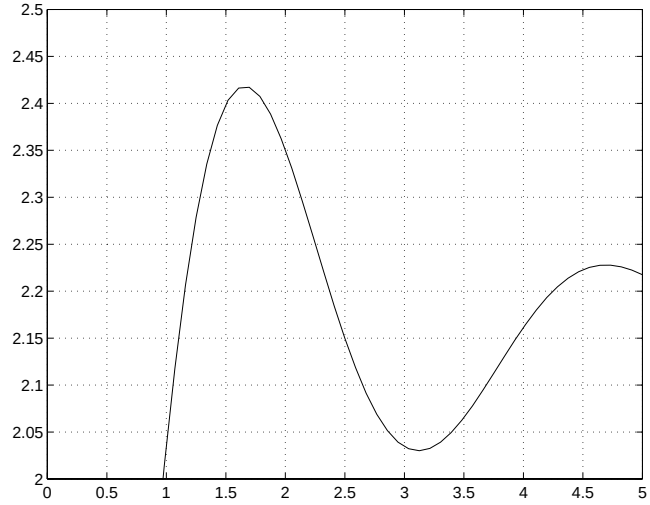
Semiautomatic Limits

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])
```



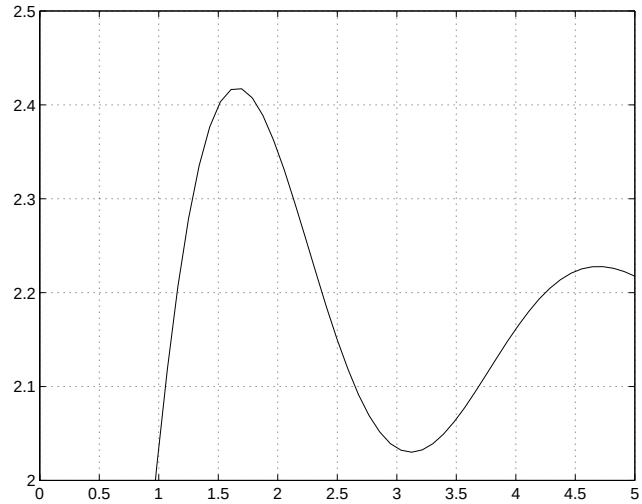
Axis Tick Marks

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'})
```

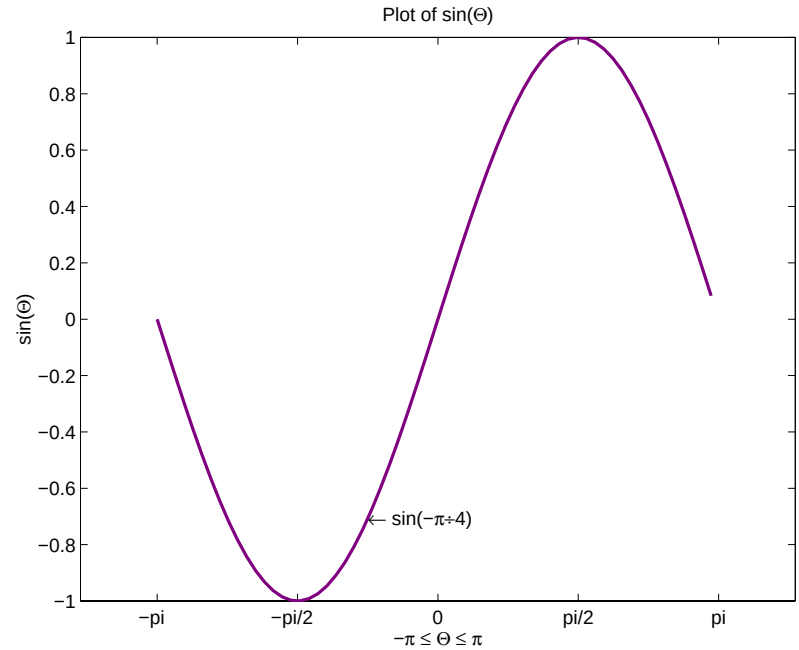
These commands (`xlabel`, `ylabel`, `title`, `text`) add axis labels and draw an arrow that points to the location on the graph where y = sin(–pi/4):

```
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.

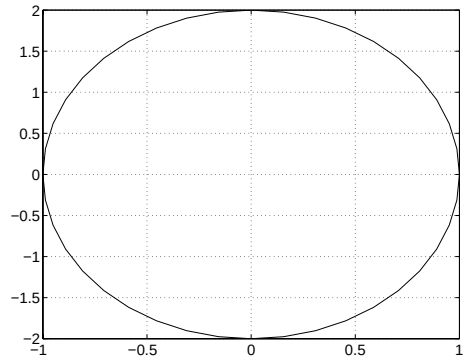
For example,

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

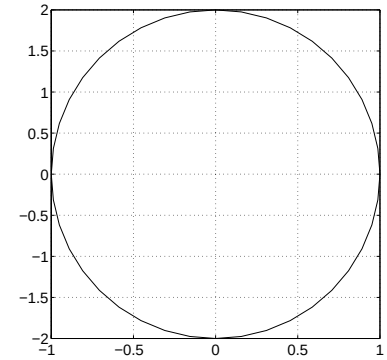
produces a graph with the default aspect ratio. The command

```
axis square
```

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



`axis normal`

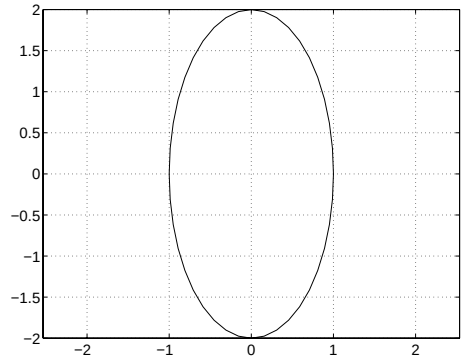


`axis square`

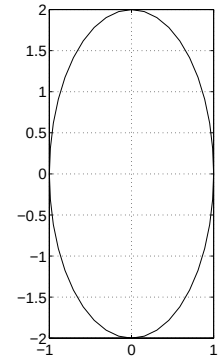
The square axes has one data unit in *x* to equal two data units in *y*. If you want the *x*- and *y*-data units to be equal, use the command:

```
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.

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

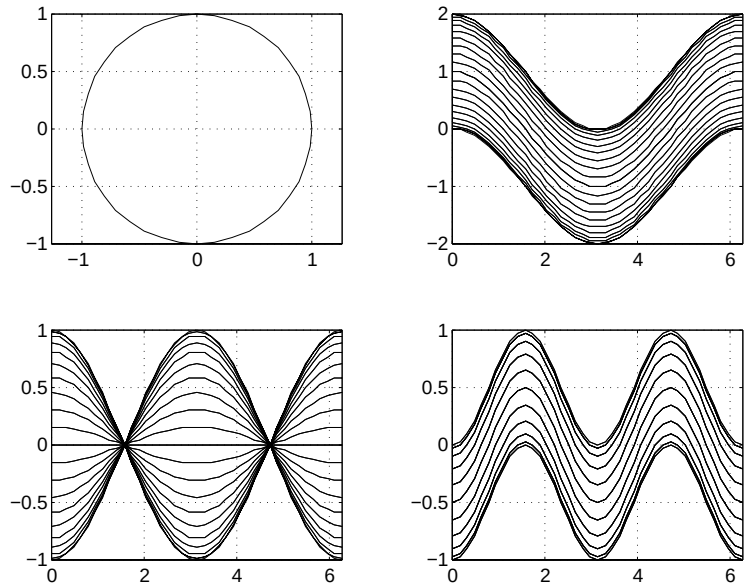
You can display multiple plots in the same figure window and print them on the same piece of paper with the `subplot` function.

`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).

3

Labeling Graphs

This topic area provides information on commands that enable you to label your graph.

Graph Labeling Commands

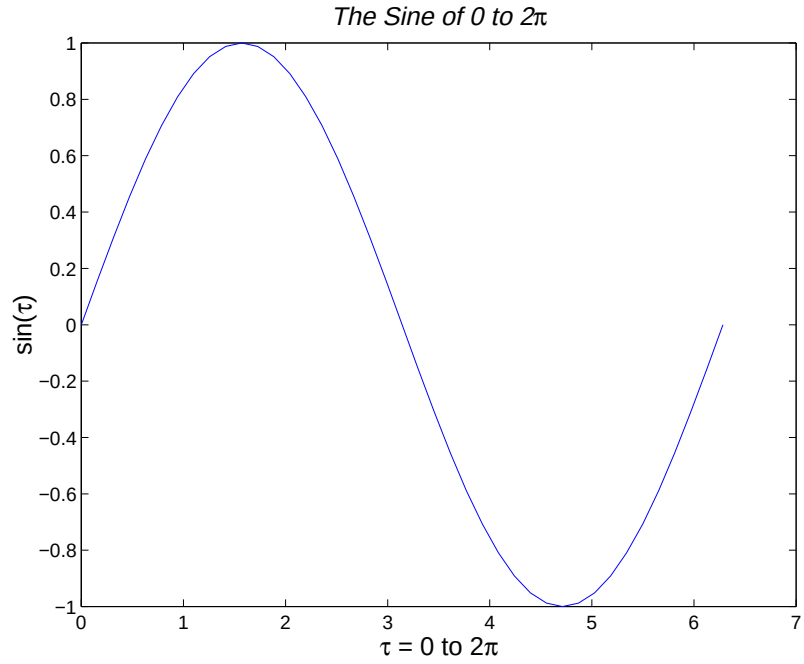
MATLAB provides commands to label each axis and place text at arbitrary locations on the graph. These commands include those displayed here.

Command	Purpose
`title`	Adds a title to the graph
`xlabel`	Adds a label to the *x*-axis
`ylabel`	Adds a label to the *y*-axis
`zlabel`	Adds a label to the *z*-axis
`legend`	Adds a legend to an existing graph
`text`	Displays a text string at a specified location
`gtext`	Places text on the graph using the mouse

Labeling Individual Axes

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

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

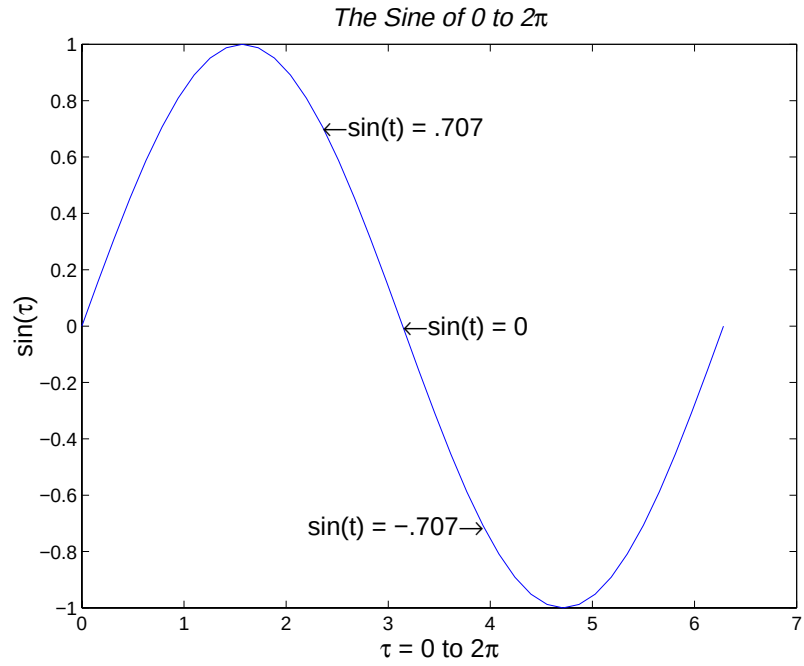
Adding Text Strings to a Graph

You can place a text string at any location on the plot using the `text` function. This function positions the text string in the data space of the

plot. For example, to label three data points on a graph, create three text strings:

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

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

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

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



Placing Text Interactively

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

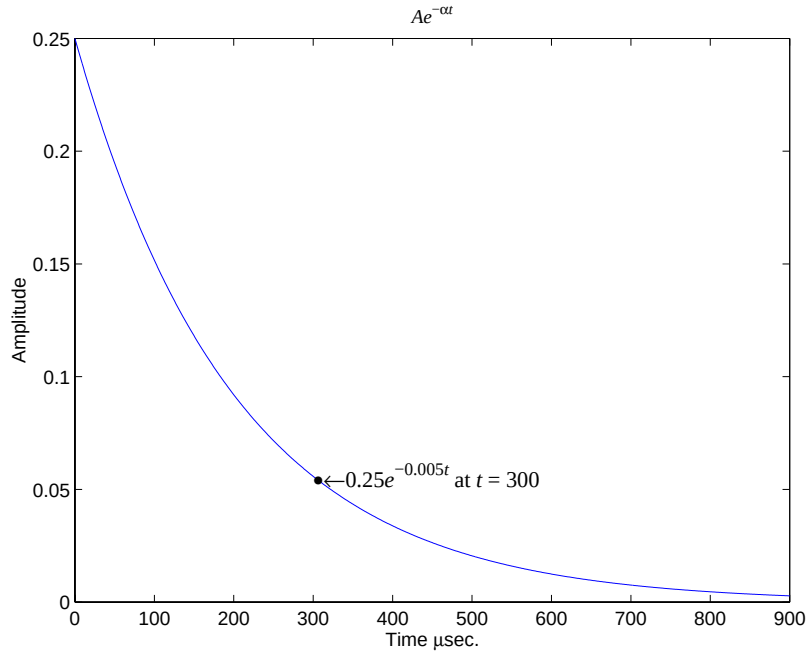
`gtext` is a convenient way to annotate your graph if you do not want precise positioning of the text. It works only on 2-D graphs.

You can also use Plot Editor to place text interactively on your graph. See the Plot Editor documentation for more information.

Positioning Text on Graphs

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)
```

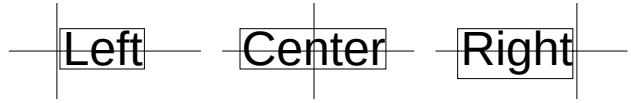
This statement defines the text `Position` property as x = 300, $y = 0.25e^{-0.005 \times 300}$. The default text alignment places this point to the left of the string and centered vertically with the rectangle defined by the text `Extent` property. Text Alignment describes additional alignment options.

Text Alignment

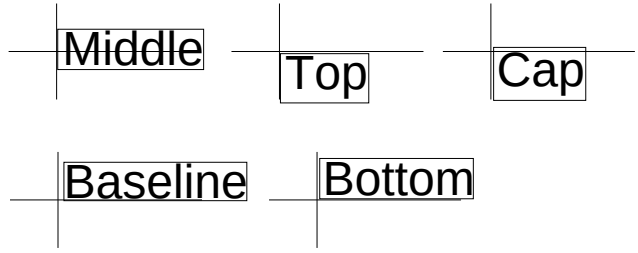
The `HorizontalAlignment` and the `VerticalAlignment` properties control the placement of the text characters with respect to the specified

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).



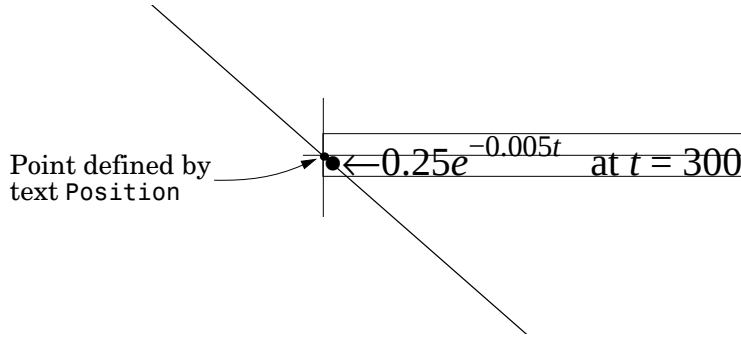
Text `VerticalAlignment` property viewed with the `HorizontalAlignment` property set to `left` (the default).



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.

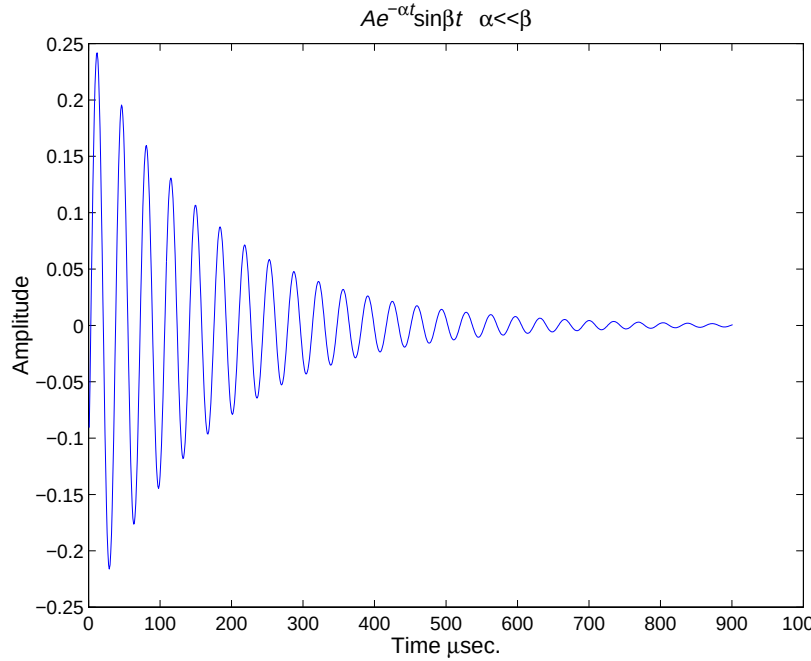


The small dot is the point specified by the text `Position` property. The larger dot is the bullet defined as the first character in the text `String` property.

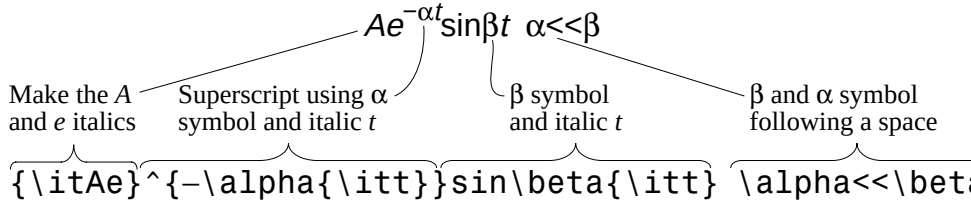
Specifying TeX Characters

Text objects support a subset of TeX characters that enable you to use symbols in the title and axis labels. For example,

```
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:



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

You can also use the `title`, `xlabel`, `ylabel`, and `zlabel` functions to add the labels. In most cases, these functions are easier to use, but affect only the current axes. Obtaining the text handle from the axes is useful in

M-files and MATLAB-based applications where you cannot be sure the intended target is the current axes.

See the text `String` property for a list of available TeX characters.

Using Variables in Text Strings

Any string variable is a valid specification for the text `String` property. 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,:)])
```

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 – Aligning Text

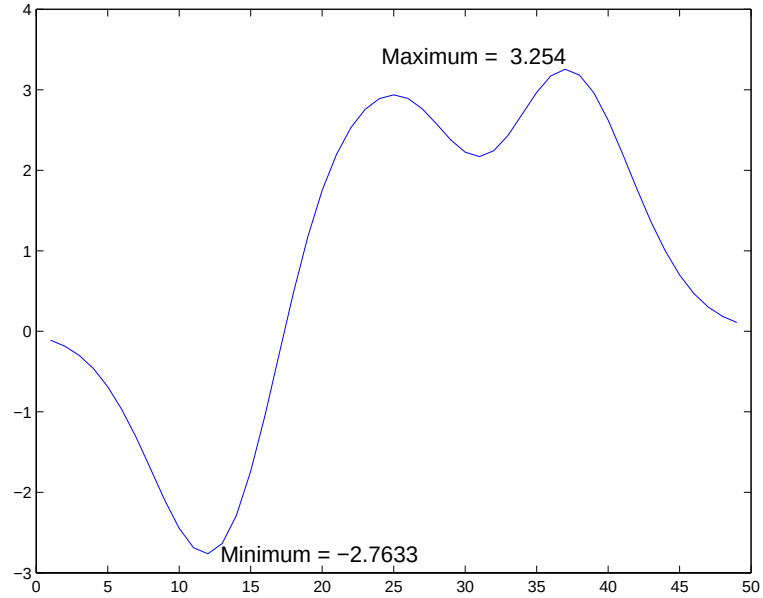
Suppose you want to label the minimum and maximum values in a plot with text that is anchored to these points and that displays the actual values. This example uses the plotted data to determine the location of

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)
```

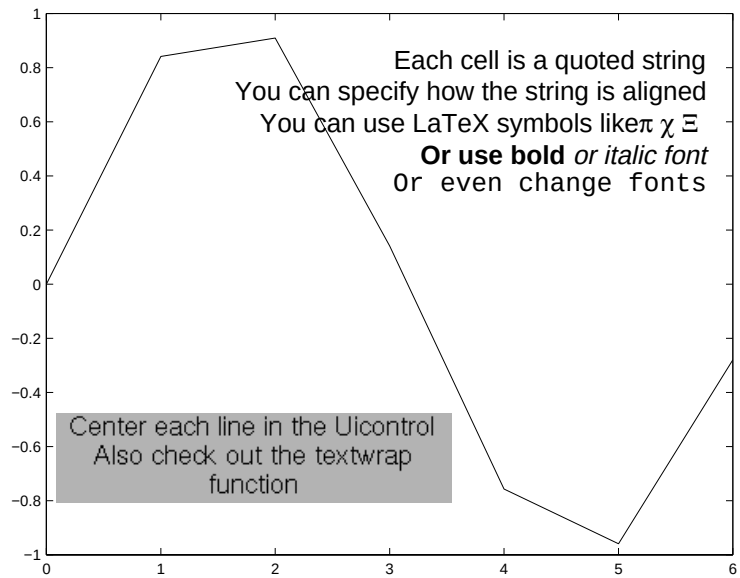


The `text` function positions the string relative to the point specified by the coordinates, in accordance with the settings of the alignment properties. For the minimum value, the string appears to the right of the text position point; for the maximum value the string appears above and to the left of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

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 80 250 65],...
          'String',str1);
text(5.75,sin(2.5),str2,'HorizontalAlignment','right')
```



4

Specialized Graphs

MATLAB supports a variety of graph types that enable you to present information effectively. The type of graph you select depends, to a large extent, on the nature of your data. The following list can help you select the appropriate graph.

- Bar and area graphs are useful to view results over time, comparing results, and displaying individual contribution to a total amount.
- Pie charts show individual contribution to a total amount.
- Histograms show the distribution of data values.
- Stem and stairstep plots display discrete data.
- Compass, feather, and quiver plots display direction and velocity vectors
- Contour plots show equivalued regions in data.
- Interactive plotting enable you to select data points to plot with the pointer.
- Animations add an addition 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.

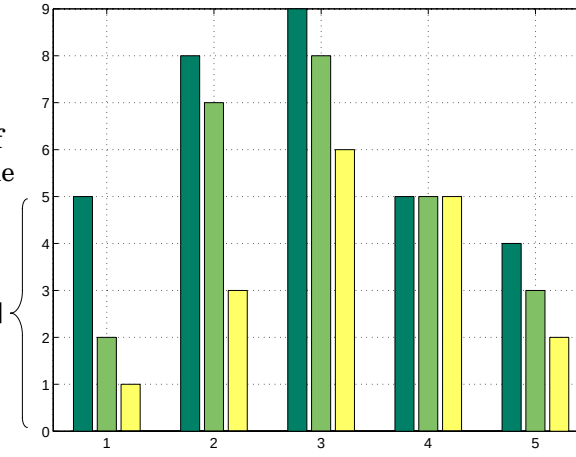
For example, define `Y` as a simple matrix and issue the `bar` statement in its simplest form:

```
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.

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

`Y(1,:) = [5 2 1]`

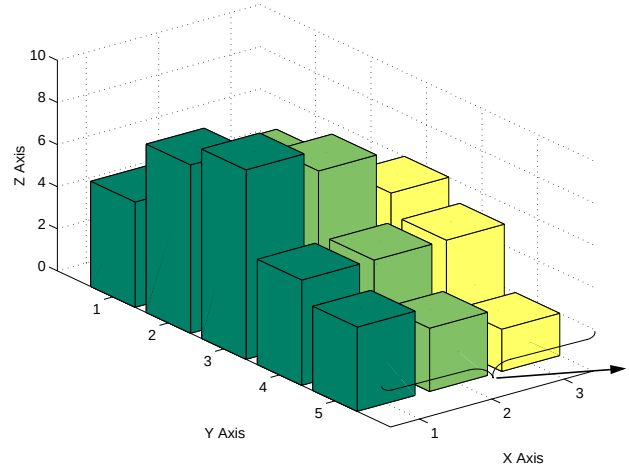


Detached 3-D Bars

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

```
bar3(Y)
```

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



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

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

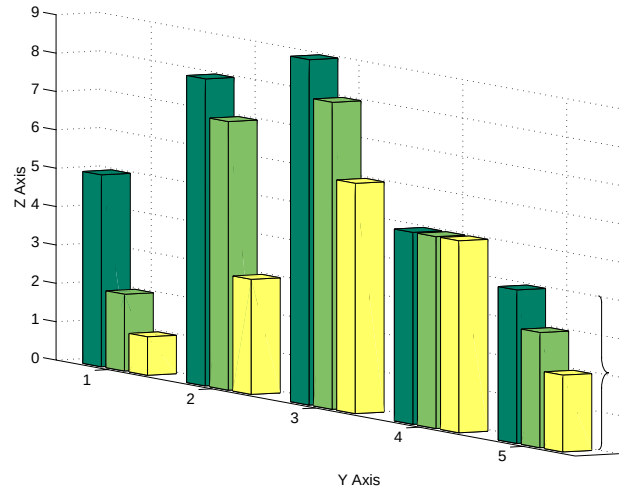
```
xlabel('X Axis')
ylabel('Y Axis')
zlabel('Z Axis')
set(gca,'XTick',[1 2 3])
```

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.



The last cluster of bars represents the last row

`Y(5,:) = [4 3 2]`

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.

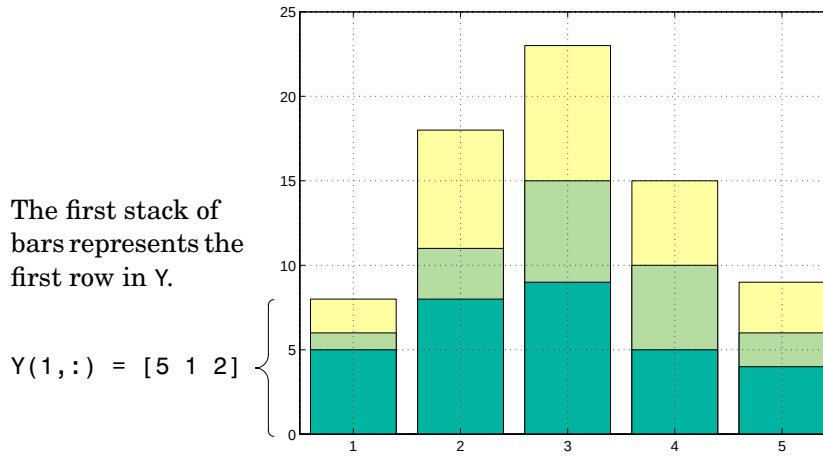
Stacked bar graphs display one bar per row of a matrix. The bars are divided into *n* segments, where *n* is the number of columns in the matrix. For vertical bar graphs, the height of each bar equals the sum of the elements in the row. Each segment is equal to the value of its respective element. Redefining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

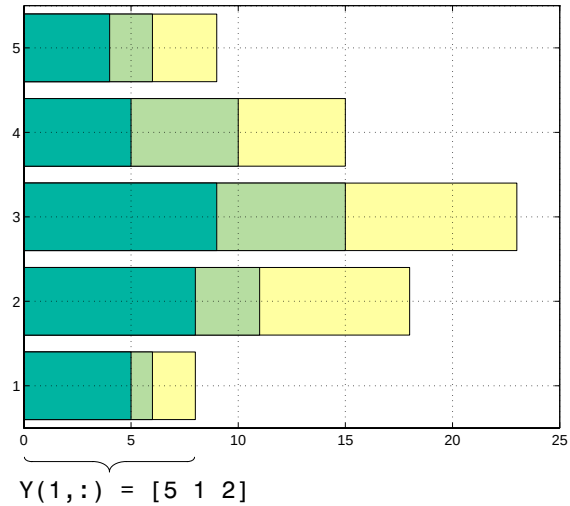
creates a 2-D stacked bar graph, where all elements in a row correspond to the same x location.



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
```



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

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,

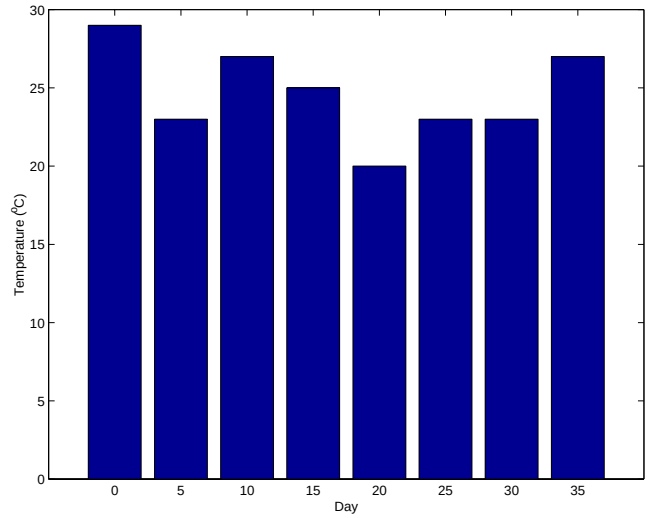
```
days = 0:5:35;
```

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.

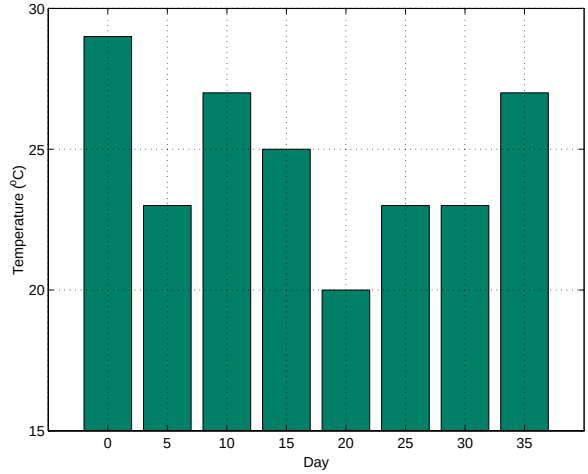
```
xlabel('Day')
ylabel('Temperature (^{o}C)')
```



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.

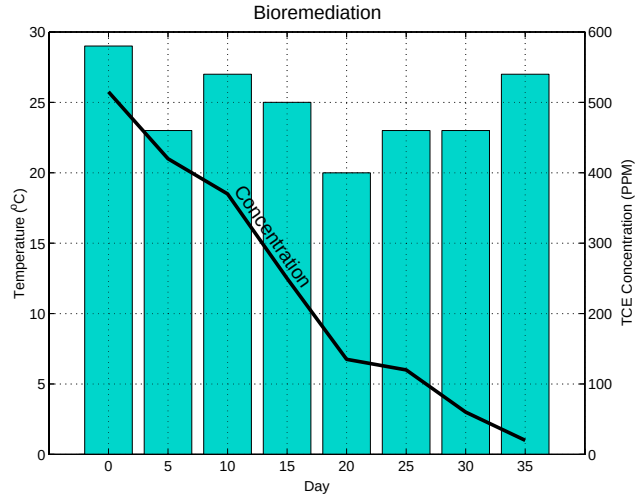
```
h2 = axes('Position',get(h1,'Position'));
plot(days,TCE,'LineWidth',3)
```

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,

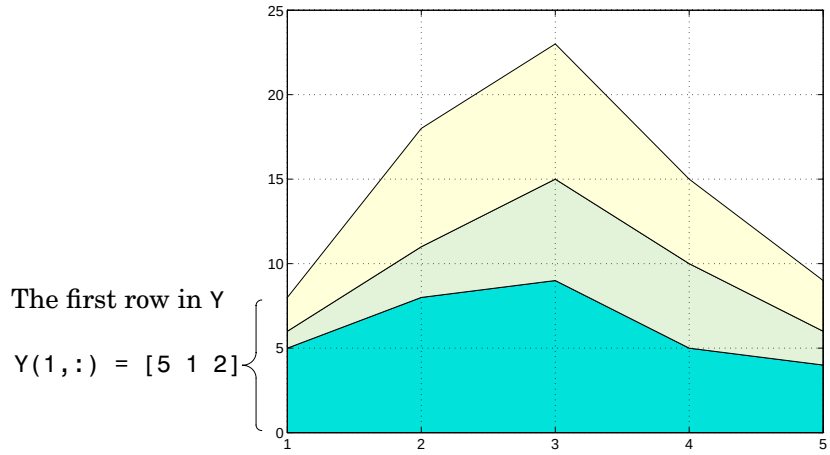
```
Y = [5 1 2
     8 3 7
     9 6 8
     5 5 5
     4 2 3];
```

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

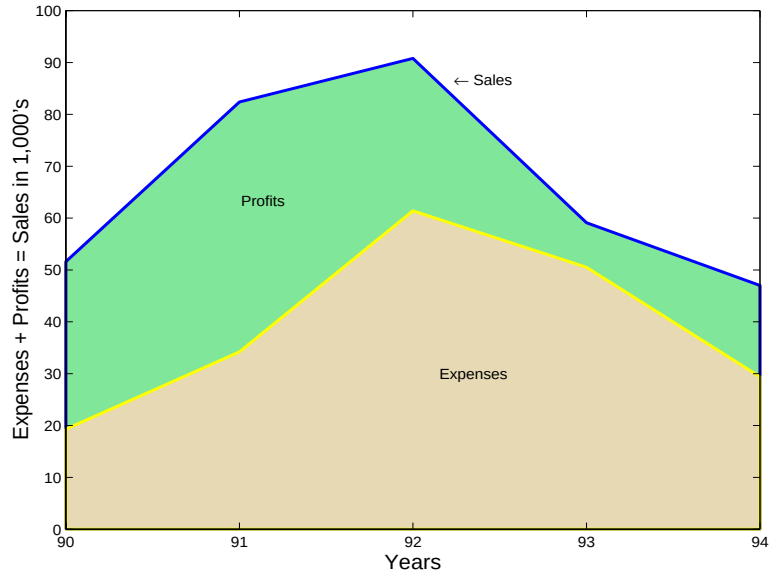
```
profits = [19.3 34.2 61.4 50.5 29.4];
```

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)
```


100
90
80
70
60
50
40
30
20
10
0
90
91
92
93
94
Years
Expenses + Profits = Sales in 1,000's
← Sales
Profits
Expenses

Pie Charts

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

Here is an example using the `pie` function to visualize the contribution 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.

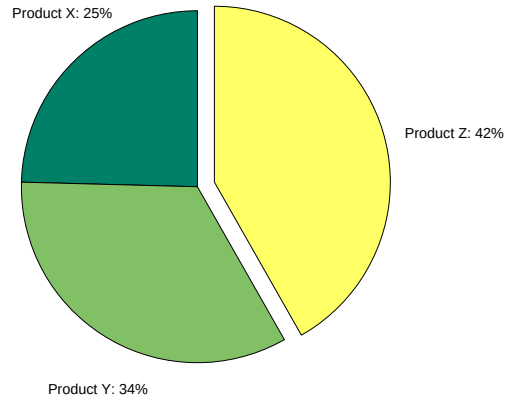
```
explode = zeros(size(x));
```

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.

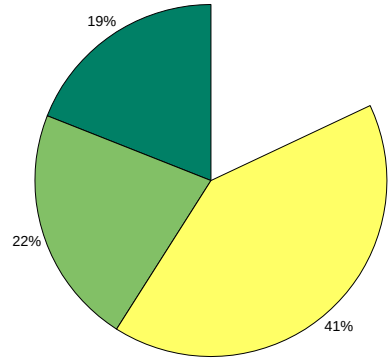
```
val1 = get(textObjs, {'Extent'});
newExt = cat(1, val1{:});
offset = sign(oldExt(:,1)).*(newExt(:,3)-oldExt(:,3))/2;
pos = get(textObjs, {'Position'});
textPos =  cat(1, pos{:});
textPos(:,1) = textPos(:,1)+offset;
set(textObjs,{'Position'},num2cell(textPos,[3,2]))
```

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 $x_i / sum(x_i)$, where x_i is an element of x. The normalized value specifies the fractional part of each pie slice.

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

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



Histograms

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

Function	Description
`hist`	Displays data in a Cartesian coordinate system
`rose`	Displays data in a polar coordinate system

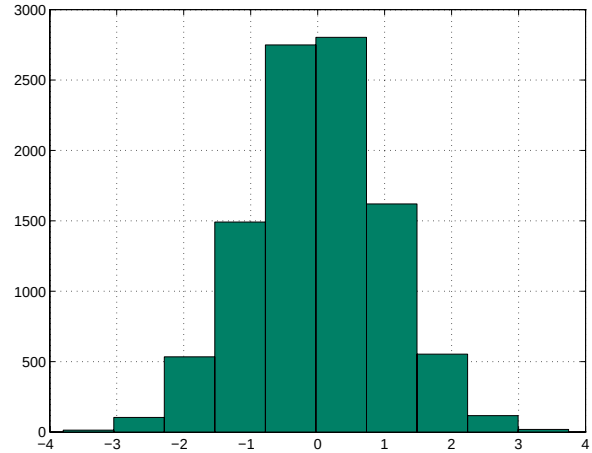
The histogram functions count the number of elements within a range and display each range as a rectangular bin. The height (or length when using `rose`) of the bins represents the number of values that fall within each range.

Histograms in Cartesian Coordinate Systems

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

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

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

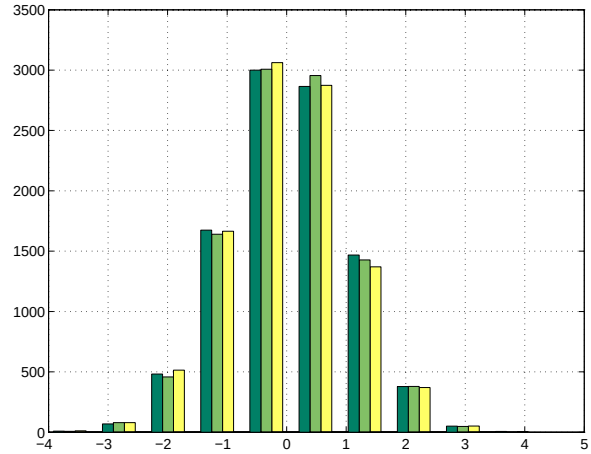


Matrix Input Argument

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 Coordinate Systems

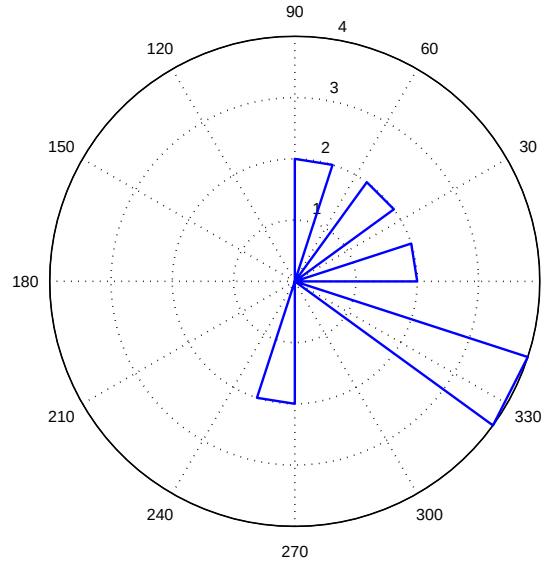
A `rose` plot is a histogram created in a polar coordinate system. For example, consider samples of the wind direction taken over a 12-hour period.

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

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

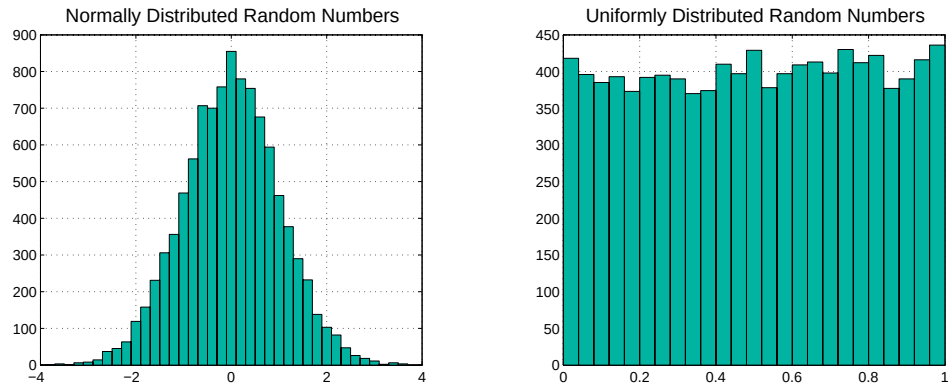
```
yn = randn(10000,1);
yu = rand(10000,1);
```

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.)

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

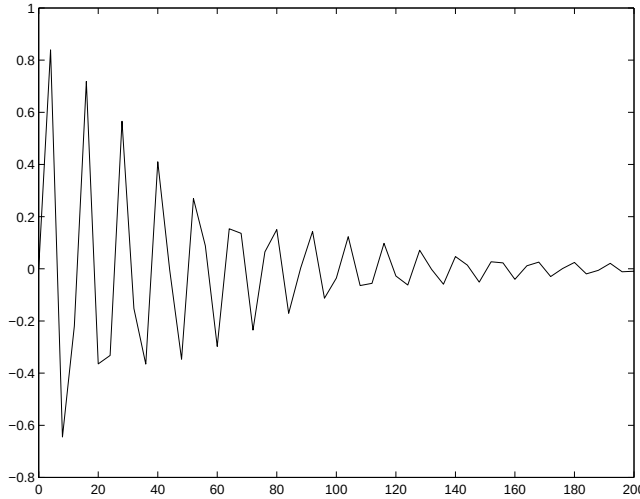
A stem plot displays data as lines (stems) terminated with a marker symbol at each data value. In a 2-D graph, stems extend from the x-axis.

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);
```

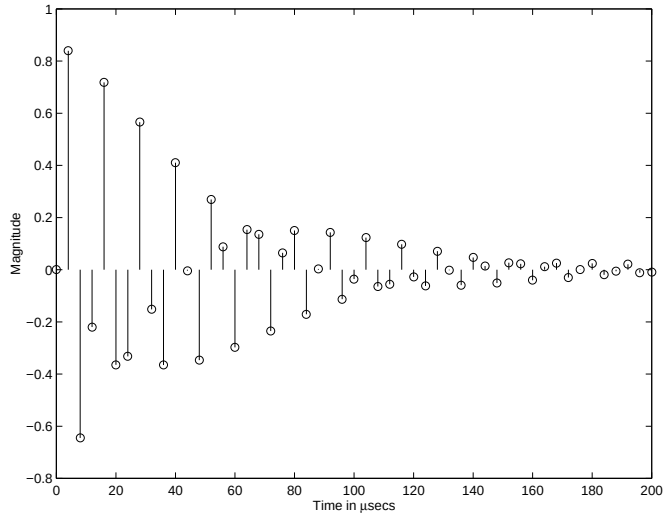
yields a vector of discrete values for `y` at given values of `t`. A line plot shows the data points connected with a straight line.

```
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.

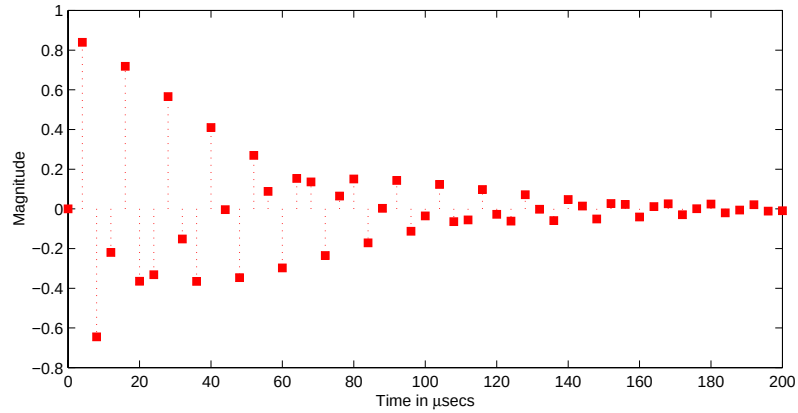
```
xlabel('Time in \musecs')
ylabel('Magnitude')
```

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 `PlotBoxApectRatio` property directly.

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

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

Combining Stem Plots with Line Plots

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

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

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

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

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

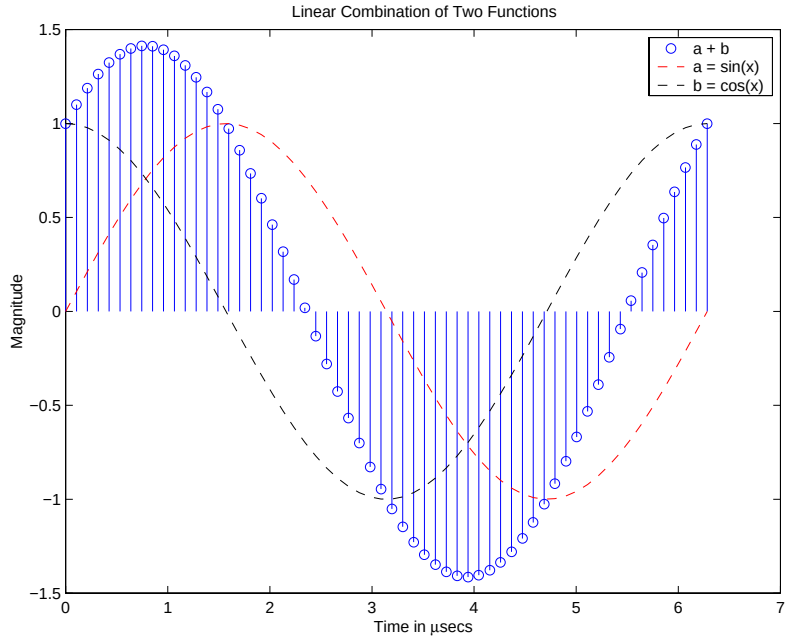
```
hold on
plot_handles = plot(x,a,'--r',x,b,'--g');
hold off
```

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

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

Labeling the axes and creating a title finishes the graph.

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



Three-Dimensional Stem Plots

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

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

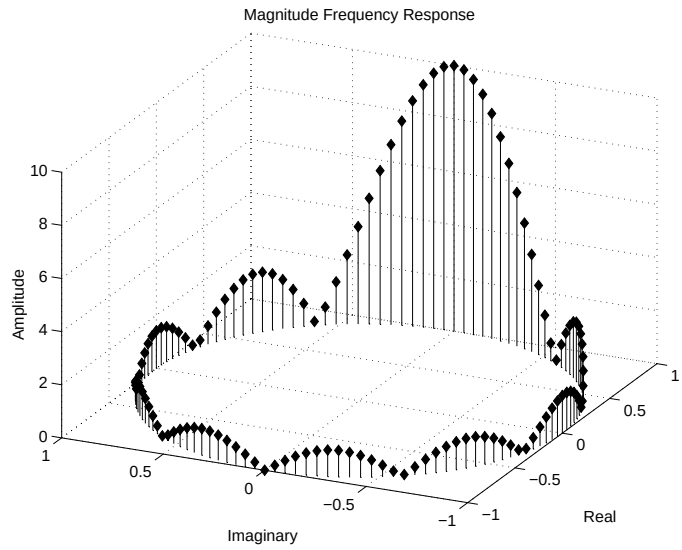
```
th = (0:127)/128*2*pi;
x = cos(th);
y = sin(th);
```

and the magnitude frequency response of a step function

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

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

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



Label the Graph

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
```

Combining Stem and Line Plots

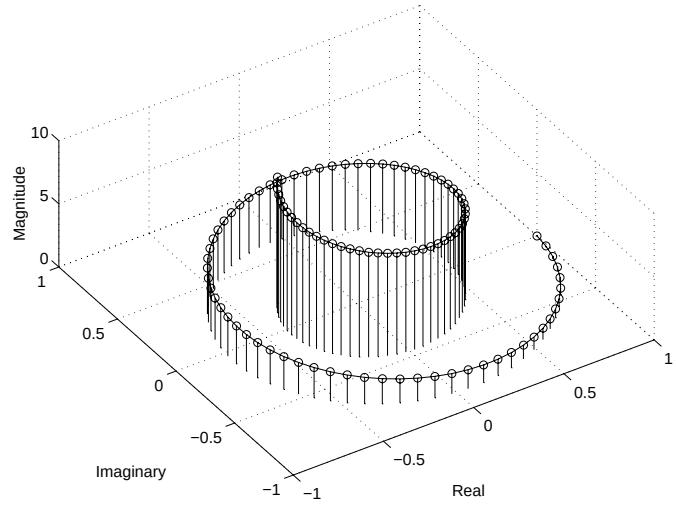
Three-dimensional stem plots work well when visualizing discrete functions that do not output a large number of data points. For example, you can use

`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,'k')
hold off
```



Label the Graph

Add axes labels, with the statements:

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

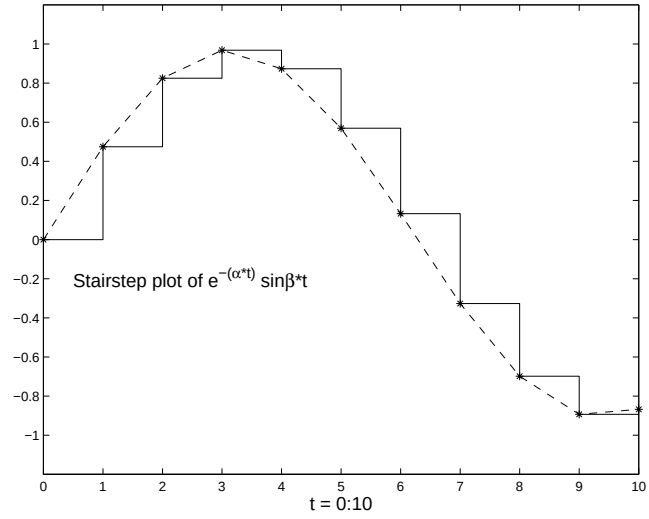
Stairstep Plots

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. 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

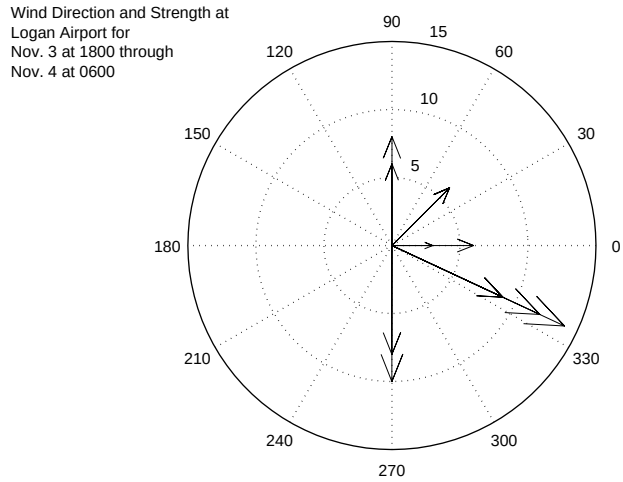
The `compass` function shows vectors emanating from the origin of a graph. The function takes Cartesian coordinates and plots them on a circular grid.

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

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

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

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



Create text to annotate the graph:

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

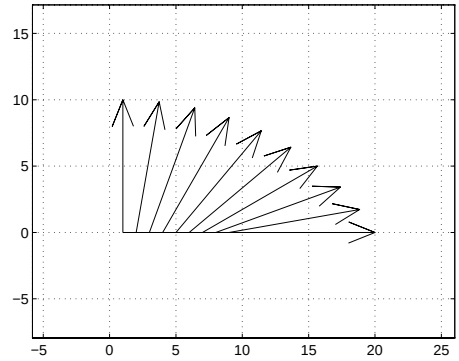
Feather Plots

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

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

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

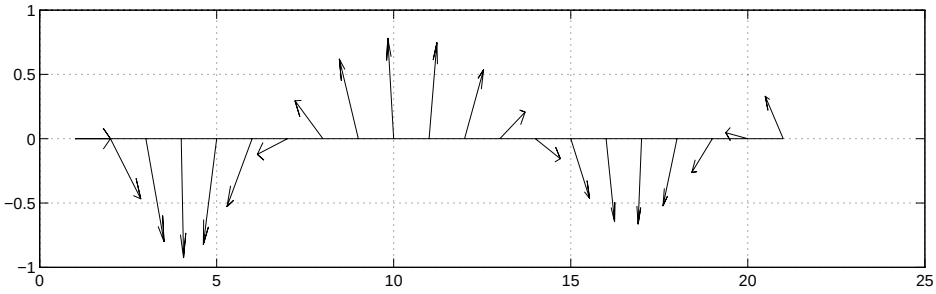
```
[u,v] = pol2cart(theta*pi/180,r*10);
feather(u,v)
axis equal
```



Plotting Complex Numbers

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

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



Printing the Graph

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

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

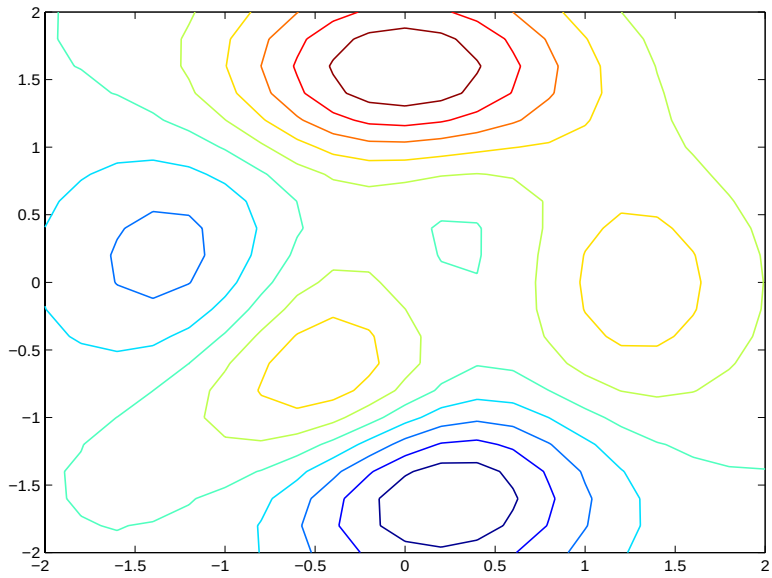
In this mode, MATLAB prints the figure as it appears on screen.

Two-Dimensional Quiver Plots

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

A quiver plot is useful when displayed with another plot. For example, create 10 contours of the `peaks` function (see more information on contour plotting).

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

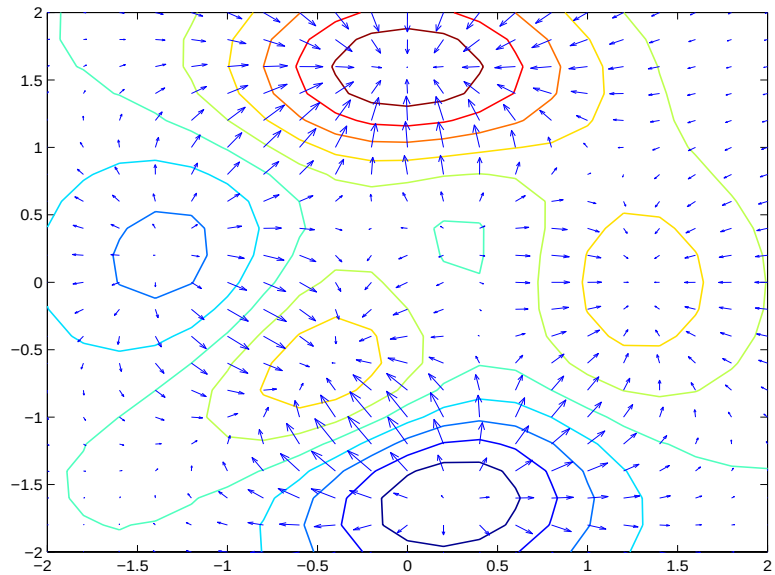


Now use `gradient` to create the vector components to use as inputs to `quiver`.

```
[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 `vz` and `a`.

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

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

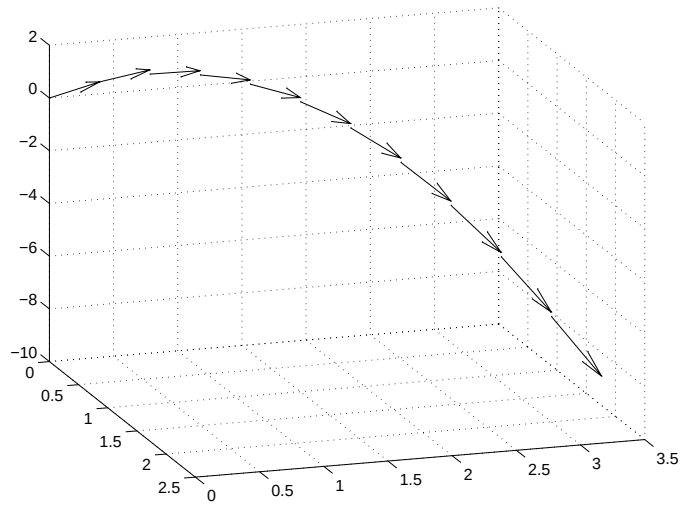
```
t = 0:.1:1;
z = vz*t + 1/2*a*t.^2;
```

Calculate the position in the x and y directions.

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

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

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



Contour Plots

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

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

Two other functions also create contours. `meshc` displays a contour in addition to a mesh, and `surfc` displays a contour in addition to a surface.

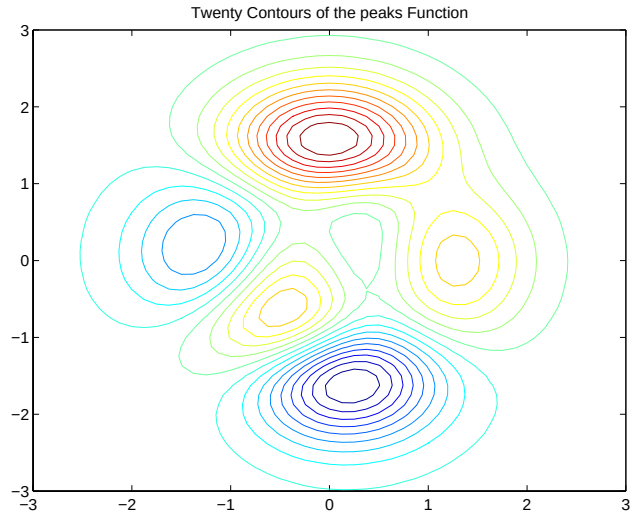
Creating Simple Contour Plots

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

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

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

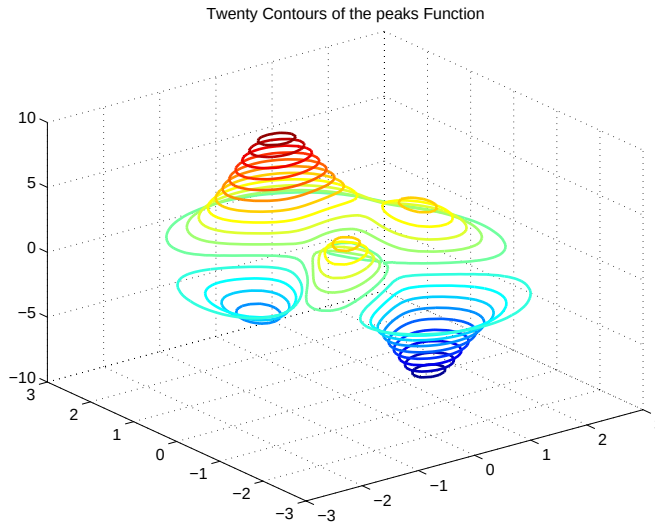
displays 20 contours of the `peaks` function in a 2-D view.



The statements

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

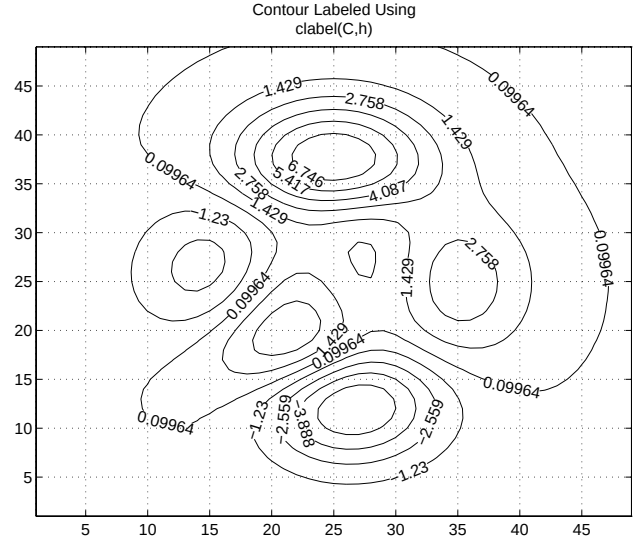
For example, display 10 contour levels of the `peaks` function,

```
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')
```

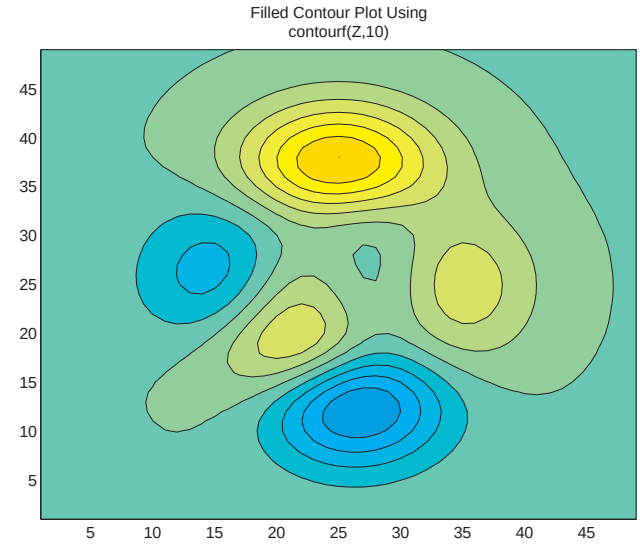
displays a crosshair cursor when your cursor is inside the figure. Pressing any mouse button labels the contour line closest to the center of the crosshair.

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

Three vertices at v = 0

Five vertices at v = 0.8

```
Columns 1 through 7
-0.2000    1.8165    2.0000    2.1835         0    1.0003    2.0000
3.0000    1.0000    1.0367    1.0000    3.0000    1.0000    1.1998

Columns 8 through 14
3.0000         0    1.0000    1.0359    1.0000    0.2000    1.6669
1.0002    3.0000    2.9991    2.0000    1.0018    5.0000    3.0000

Columns 15 through 21
1.2324    2.0000    2.8240    2.3331    0.4000    2.0000    2.6130
2.0000    1.3629    2.0000    3.0000    5.0000    2.8530    2.0000

Columns 22 through 28
2.0000    1.4290    2.0000    0.6000    2.0000    2.4020    2.0000
1.5261    2.0000    2.8530    5.0000    2.5594    2.0000    1.6892

Columns 29 through 35
1.6255    2.0000    0.8000    2.0000    2.1910    2.0000    1.8221
2.0000    2.5594    5.0000    2.2657    2.0000    1.8524    2.0000

Column 36
2.0000
2.2657
```

The circled values begin each definition of a contour line.

Changing the Offset of a Contour

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

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

The first handle belongs to the mesh or surface. The remaining handles belong to the contours you want to change. To raise the contour plane, 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;
```

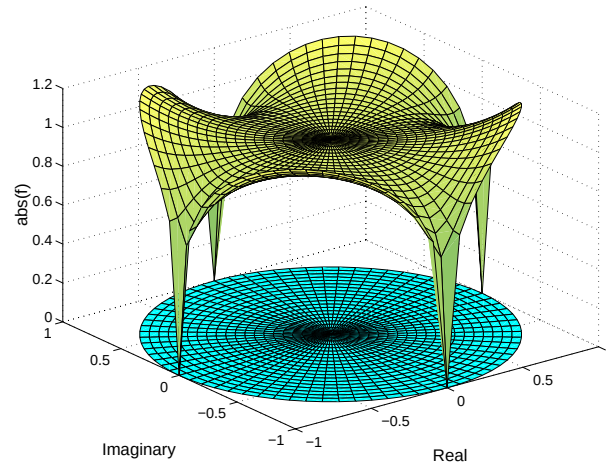
`X`, `Y`, and `Z` are points inside the circle.

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

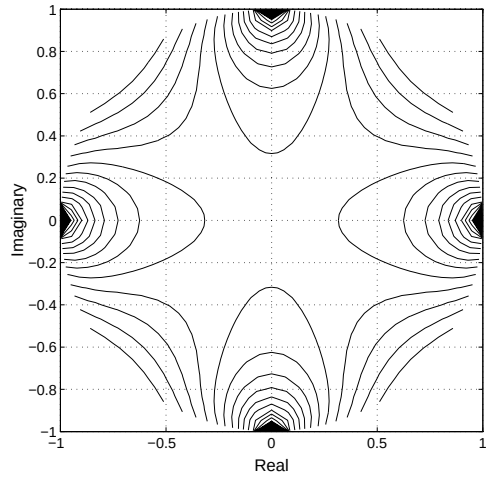
These statements add labels:

```
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);
```



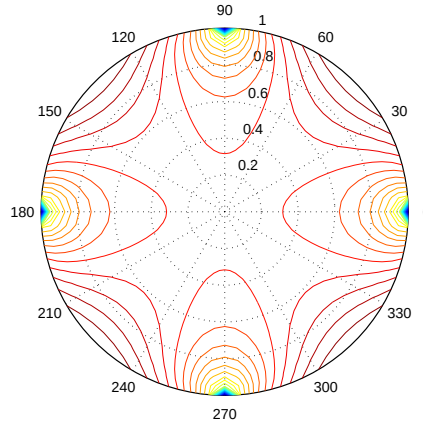
Contours on a Polar Axis

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)
```


90
1
120
60
0.8
0.6
150
30
0.4
0.2
180
0
210
330
240
300
270

Interactive Plotting

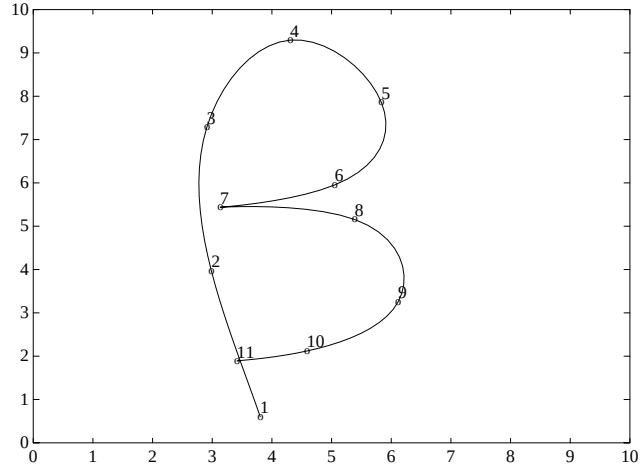
The `ginput` function enables you to use the mouse or the arrow keys to select points to plot. `ginput` returns the coordinates of the pointer's position; either the current position or the position when a mouse button or key is pressed. See the `ginput` function in the online MATLAB Function Reference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

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

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

```
axis([0 10 0 10])
hold on

% Initially, the list of points is empty.
x = [];
y = [];
n = 0;

% Loop, picking up the points.
disp('Left mouse button picks points.')
disp('Right mouse button picks last point.')
but = 1;
while but == 1
    [xi,yi,but] = ginput(1);
    plot(xi,yi,'go')
    n = n+1;
    x(n,1) = xi;
    y(n,1) = yi;
end

% Interpolate with two splines and finer spacing.
t = 1:n;
ts = 1: 0.1: n;
xs = spline(t,x,ts);
ys = spline(t,y,ts);

% Plot the interpolated curve.
plot(xs,ys,'c-');
hold off
```

This plot shows some typical output.


10
9
8
7
6
5
4
3
2
1
0
0 1 2 3 4 5 6 7 8 9 10
4
5
3
6
7
8
2
9
11
10
1

Animation

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

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

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

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

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

Movies

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

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

Visualizing an FFT

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

Creating the Movie

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

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

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

The statement,

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

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

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

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

If you plan to convert the MATLAB movie to another format (such as QuickTime) and you want to include the axis labels in this new format, you should capture the figure, not just the axes. If you are using `moviein` to pre-allocate the movie matrix, be sure to specify the same figure handle that you use with `getframe`.

Running the Movie

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

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

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

Full-Figure Movies

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

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

Erase Modes

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

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

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

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

Example

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

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

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

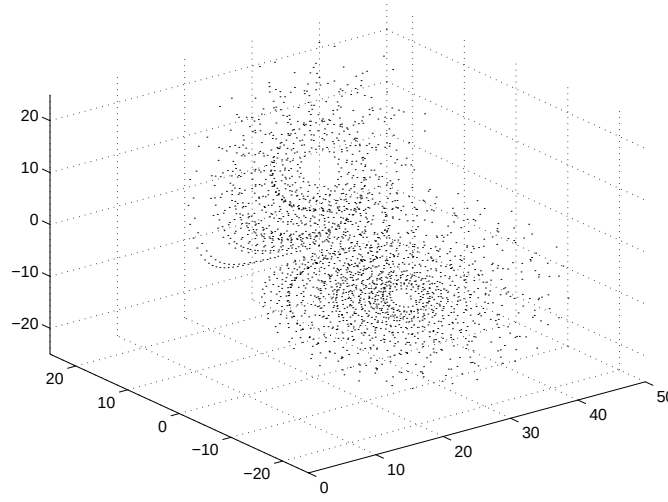
$$A(y) = \begin{bmatrix} -\frac{8}{3} & 0 & y(2) \\ 0 & -10 & 10 \\ -y(2) & 28 & -1 \end{bmatrix}$$

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

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

```
A = [ -8/3 0 0; 0 -10 10; 0 28 -1 ];
y = [35 -10 -7]';
h = 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;
    set(p,'XData',y(1),'YData',y(2),'ZData',y(3)) % Change coordinates
    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.

5

Creating 3-D Graphs

Building a 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 4a or step 4b.

Step	Typical Code
1 Prepare your data	`Z = peaks(20);`
2 Select window and position plot region within window	`figure(1)` `subplot(2,1,2)`
3 Call 3-D graphing function	`h = surf(Z);`

4a Set colormap and shading algorithm	`colormap hot`
	`shading interp`
	`set(h,'EdgeColor','k')`

4b Add lighting	`light('Position',[-2,2,20])`
	`lighting phong`
	`material([0.4,0.6,0.5,30])`
	`set(h,'FaceColor',[0.7 0.7 0],...`
	`  'BackFaceLighting','lit')`

5 Set viewpoint	`view([30,25])` `set(gca,'CameraViewAngleMode','Manual')`
6 Set axis limits and tick marks	`axis([5 15 5 15 –8 8])` `set(gca'ZTickLabel','Negative||Positive')`
7 Set aspect ratio	`set(gca,'PlotBoxAspectRatio',[2.5 2.5 1])`

8 Annotate the graph with axis labels, legend, and text	`xlabel('X Axis')`
	`ylabel('Y Axis')`
	`zlabel('Function Value')`
	`title('Peaks')`

9 Print graph	`set(gcf,'PaperPositionMode','auto')` `print –dps2`

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

Mesh and Surface Plots

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

```
mesh(Z)
```

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

```
surf(Z)
```

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

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

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

Visualizing Functions of Two Variables

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

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

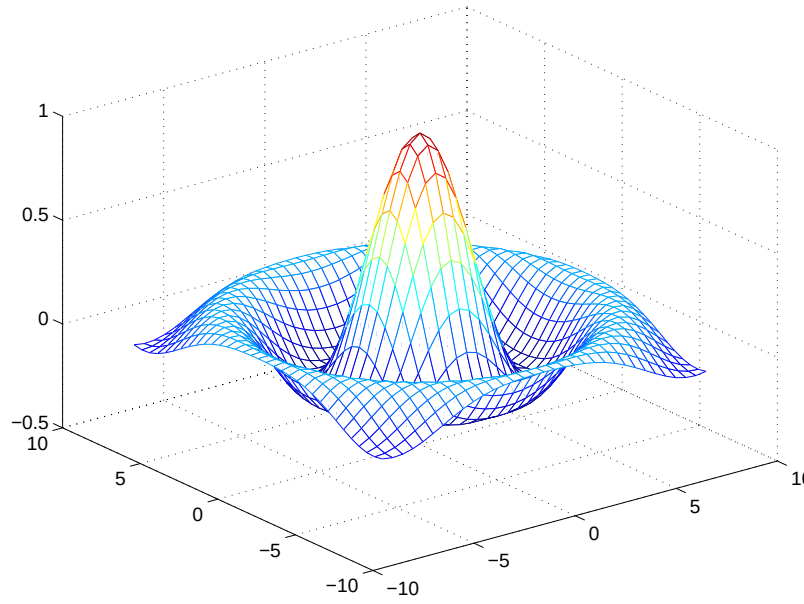
To illustrate the use of `meshgrid`, consider the *sin(r)/r* or *sinc* function. To evaluate this function between –8 and 8 in both *x* and *y*, you need pass only one vector argument to `meshgrid`, which is then used in both directions:

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

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

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

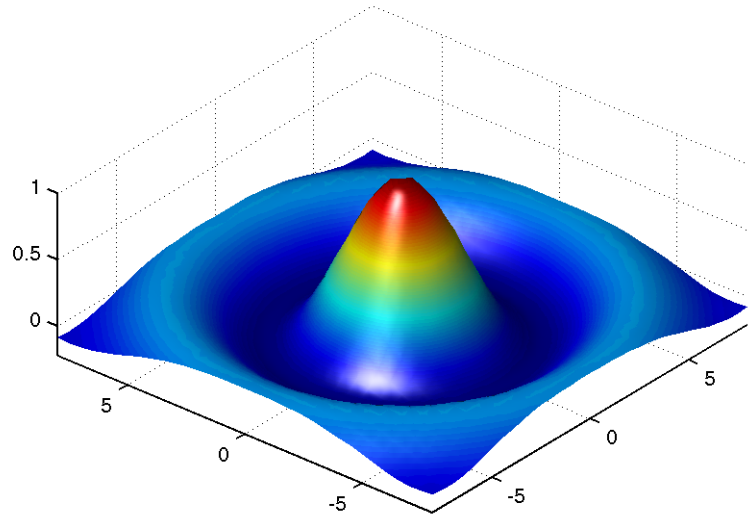
```
Z = sin(R)./R;
mesh(Z)
```



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.

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

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

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

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

Now use these points to generate a uniformly spaced grid.

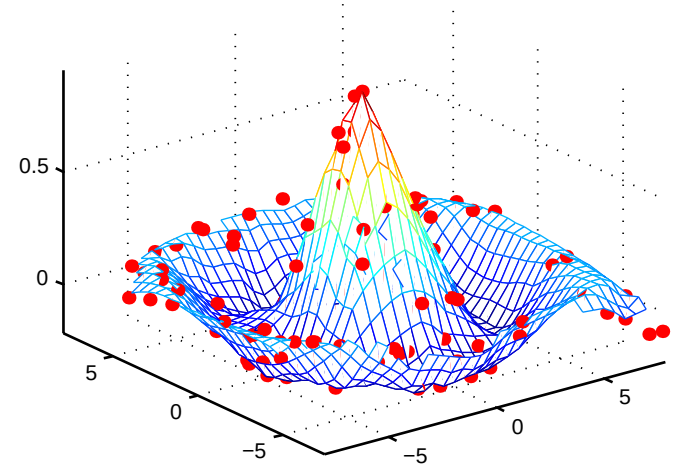
```
[X,Y] = meshgrid(xlin,ylin);
```

The key to this process is to use `griddata` to interpolate the values of the function at the uniformly spaced points, based on the values of the function at the original 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');
```

Plotting the interpolated and the nonuniform data produces

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



Parametric Surfaces

The functions that draw surfaces can take two additional vector or matrix arguments to describe surfaces with specific *x* and *y* data. If `Z` is an *m*-by-*n* matrix, `x` is an *n*-vector, and `y` is an *m*-vector, then

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

describes a mesh surface with vertices having color `C(i,j)` and located at the points

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

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

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

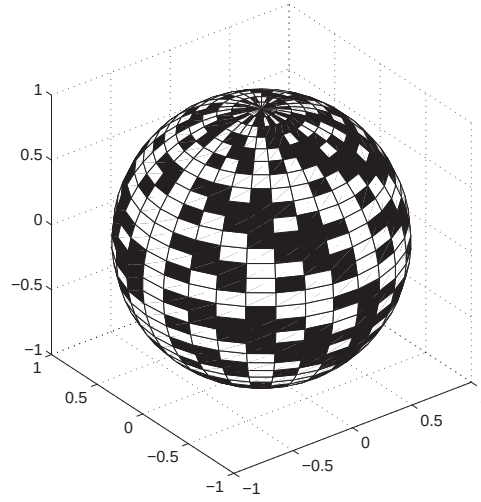
```
mesh(X,Y,Z,C)
```

describes a mesh surface with vertices having color `C(i,j)` and located at the points:

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

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

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

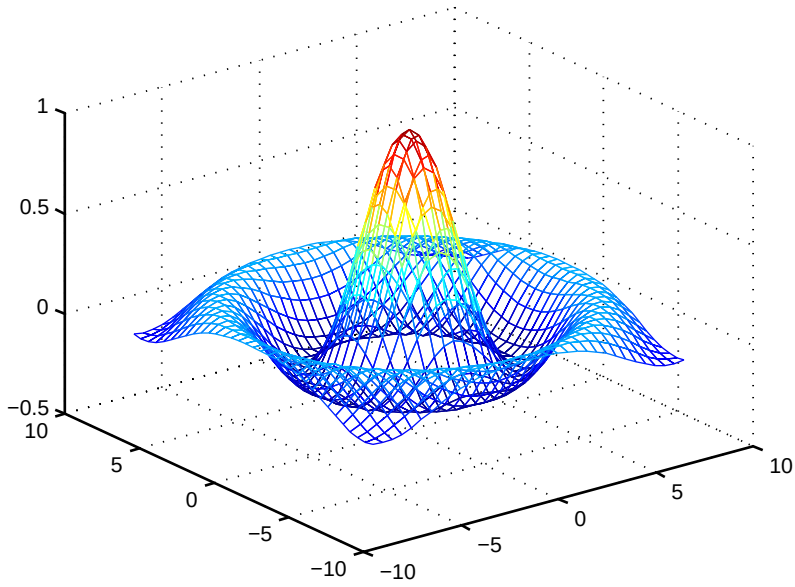


Hidden Line Removal

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

```
hidden off
```

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



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*.

There are two coloring techniques:

- Indexed Color – MATLAB colors the surface plot by assigning each data point an index into the figure's colormap. The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated).
- Truecolor – MATLAB colors the surface plot using the explicitly specified colors (i.e., the RGB triplets). The way MATLAB applies these colors depends on the type of shading used (faceted, flat, or interpolated). To be 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.

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, MATLAB 's default colormap contains 64 colors and the 57^{th} color is red.

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

This table lists some representative RGB color definitions.

Red	Green	Blue	Color
0	0	0	black
1	1	1	white
1	0	0	red
0	1	0	green
0	0	1	blue
1	1	0	yellow
1	0	1	magenta
0	1	1	cyan
0.5	0.5	0.5	gray
0.5	0	0	dark red
1	0.62	0.40	copper
0.49	1	0.83	aquamarine

You can create colormaps with MATLAB '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)`.

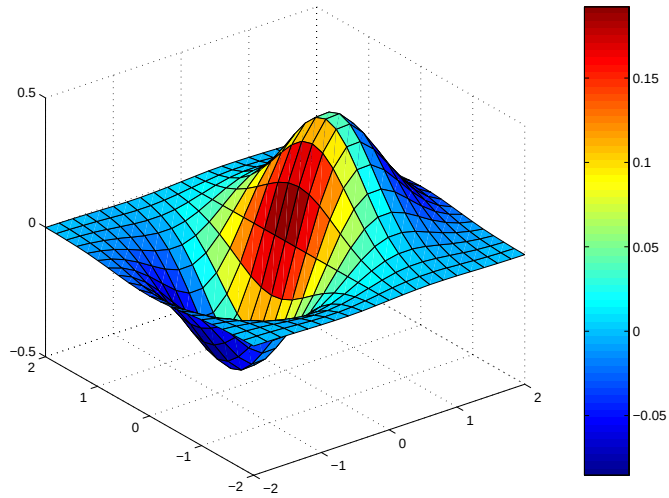
If you use long colormaps (> 64 colors) in each of several figures windows, it may become necessary for the operating system to swap in different color lookup tables as the active focus is moved among the windows. See Controlling How MATLAB Uses Color for more information on how MATLAB manages color.

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 Colors – 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 Calculating Color Limits 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)
```

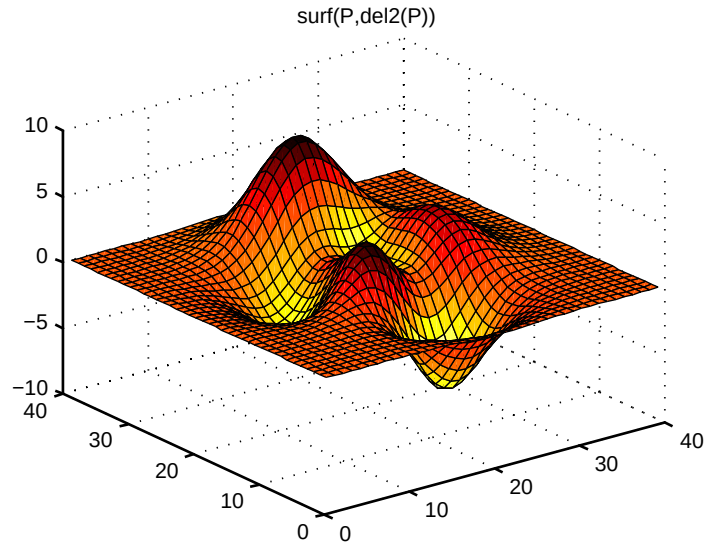
independently specifies the color using the second argument.

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
```

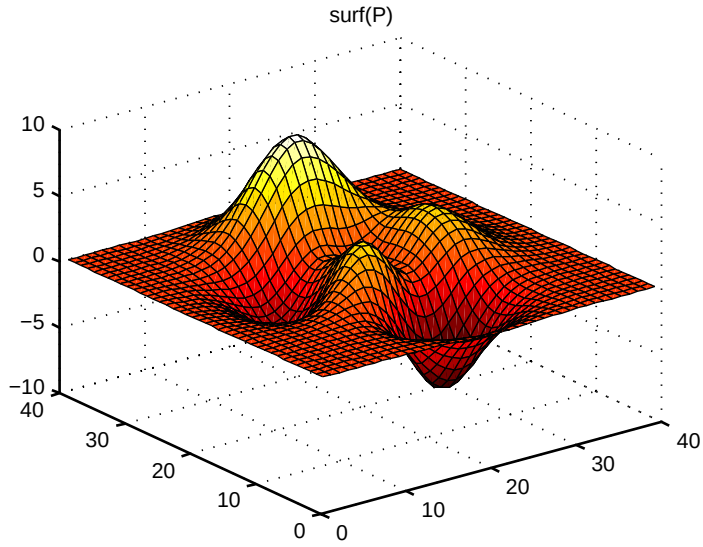
Creating a color array by applying the Laplacian to the data is useful because it causes regions with similar curvature to be drawn in the same color.



Compare this surface coloring with that produced by the statements

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

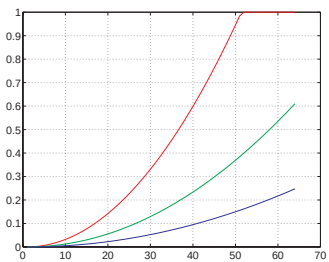
which use the same colormap, but maps regions with similar z value (height above the x-y plane) to the same color.



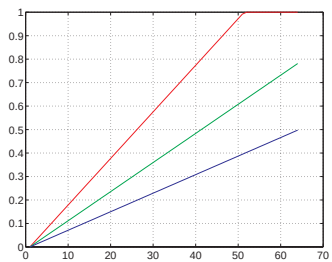
Altering Colormaps

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`.

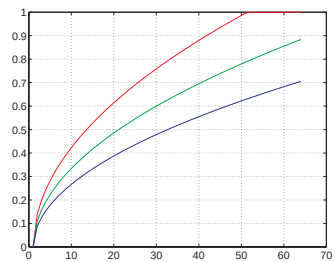
`brighten(copper,−0.5)`



`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,

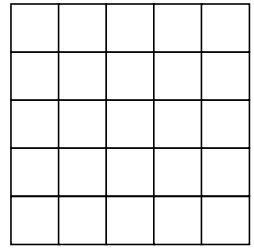
```
colormap([b b b])
```

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

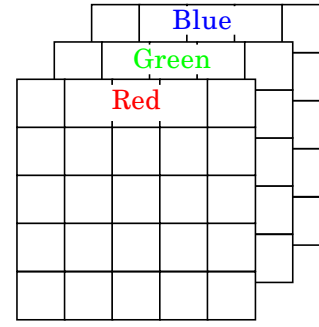
Truecolor

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

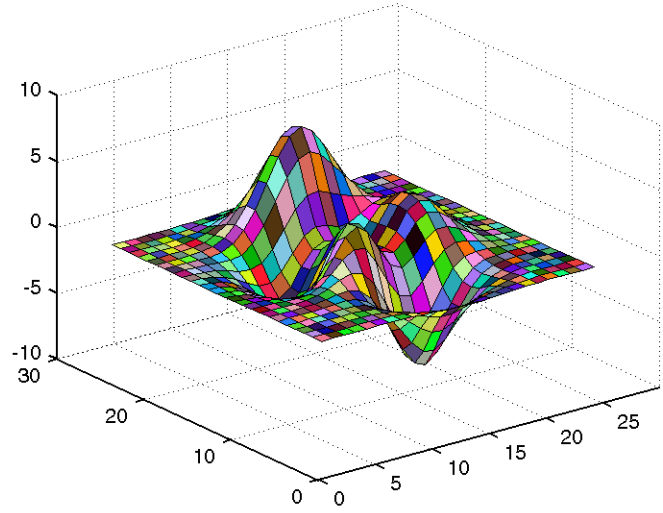


Corresponding *m*-by-*n*-by-3 matrix specifying truecolor for the surface plot

For example, the statements

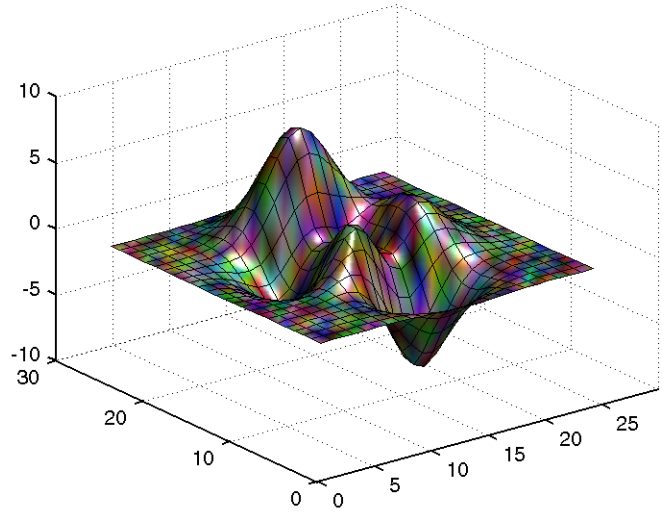
```
Z = peaks(25);
C(:,:,1) = rand(25);
C(:,:,2) = rand(25);
C(:,:,3) = rand(25);
surf(Z,C)
```

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

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

See the `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 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.

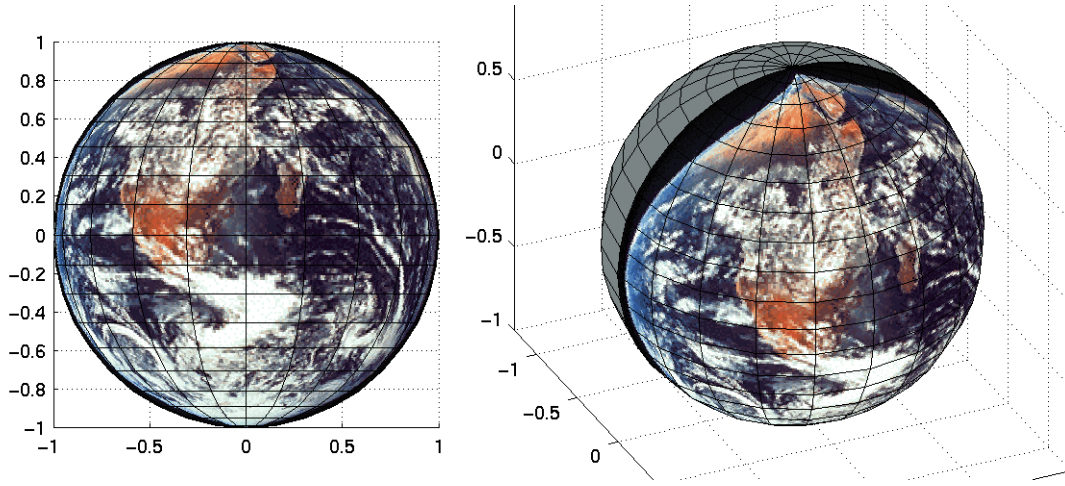
Texture mapping allows the dimensions of the color data array to be different from the data defining the surface plot. You can apply an image of arbitrary size to any surface. MATLAB interpolates texture color data so that it is mapped to the entire surface.

Example

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])
```


1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1
−1
−0.5
0
0.5
1
0.5
0
−0.5
−1
−1
−0.5
0

6

Defining the View

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 topic area describes how to define the various viewing parameters to objtain 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 comprised 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.

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 and Default Aspect Ratio Selection for a description of how MATLAB defines the standard view.

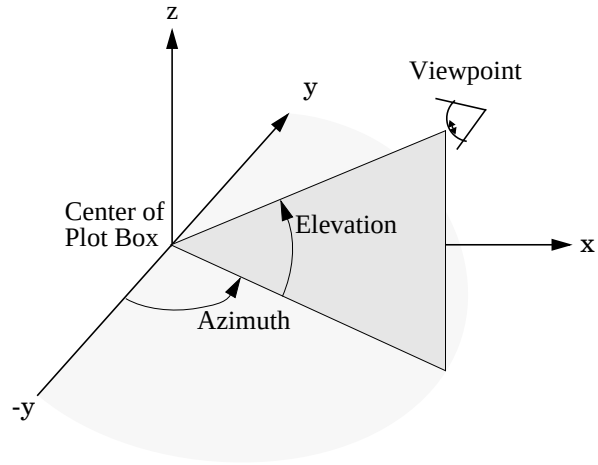
Setting the Viewpoint

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

Specifying 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.

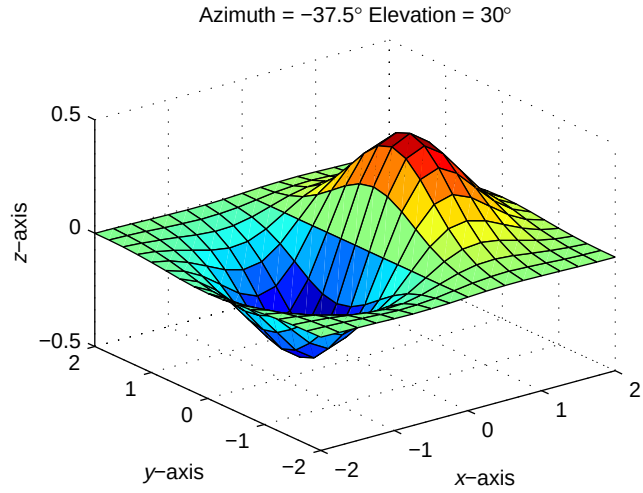


MATLAB automatically selects a viewpoint that is determined by whether the plot is 2-D or 3-D.

- 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˚.

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

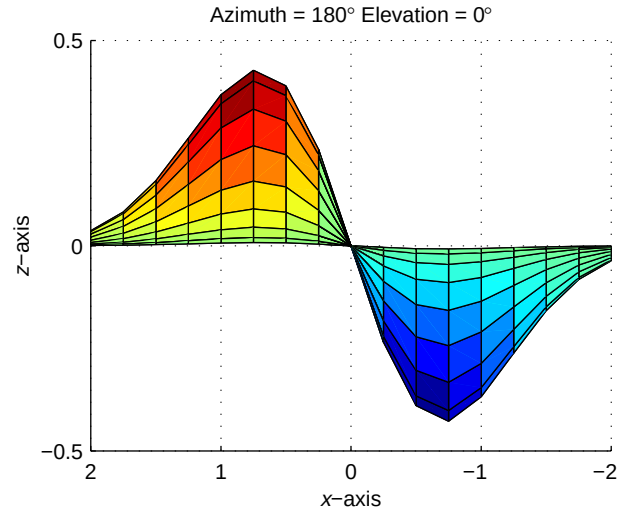
```
[X,Y] = meshgrid([–2:.25:2]);
Z = X.*exp(–X.^2 –Y.^2);
surf(X,Y,Z)
```



The statement

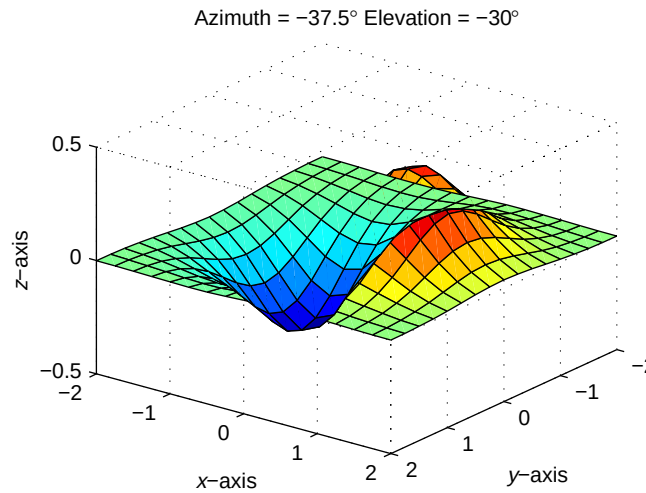
```
view([180 0])
```

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

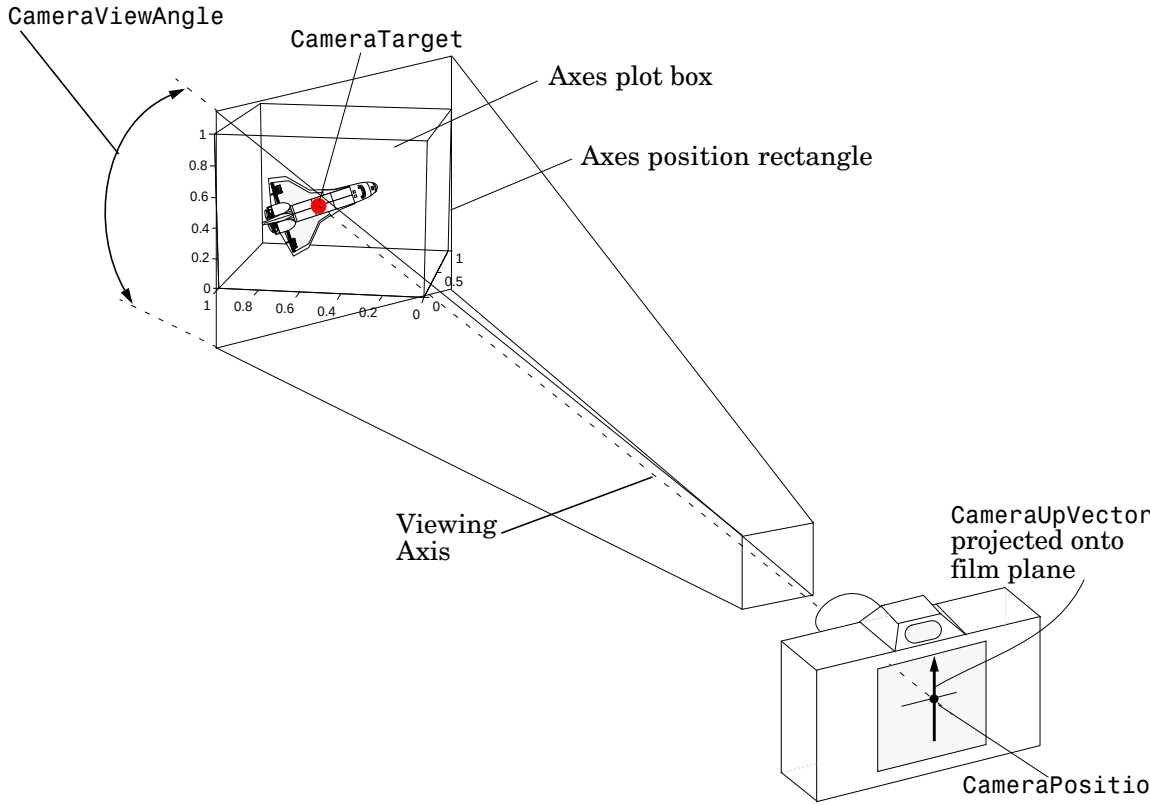
Specifying the viewpoint in terms of azimuth and elevation is conceptually simple, but it has limitations. It does not allow you to specify the actual position of the viewpoint, just its direction, and the *z*-axis is always pointing up. It does not allow you to zoom in and out on the scene or perform arbitrary rotations and translations.

MATLAB's camera graphics provides greater control than the simple adjustments allowed with azimuth and elevation. The following sections discusses 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.



Camera Graphics Commands

The following table lists MATLAB commands that enable you to perform a number of useful camera maneuvers. The individual command descriptions provide information on using each one.

Command	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 – Creating a Fly-Through

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-throughs 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-though 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 calculated 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)
```

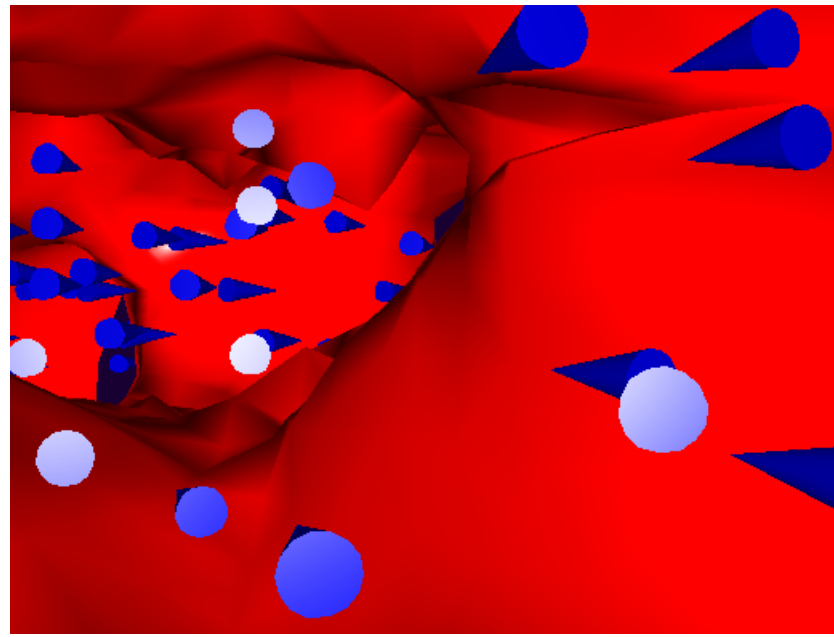
Implementing the Fly-By

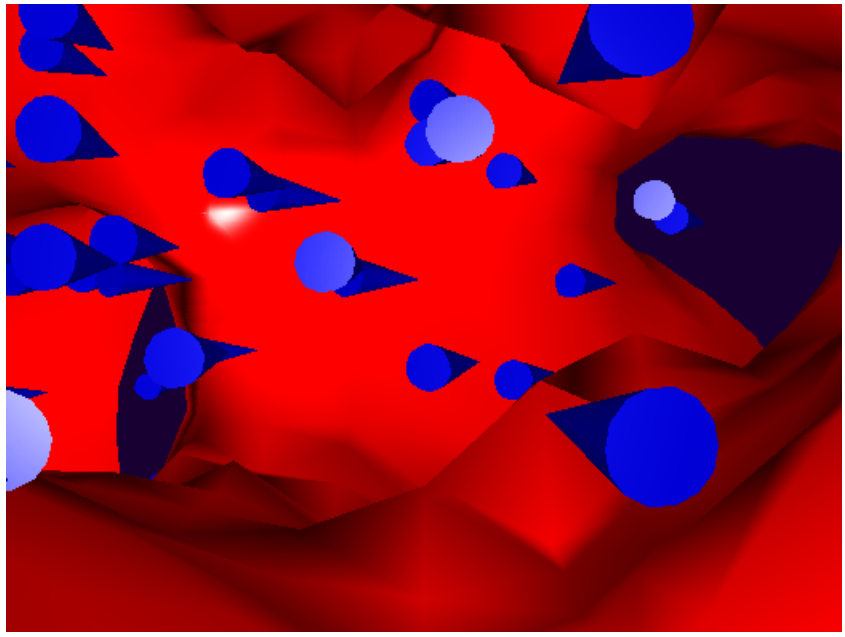
To create a fly-by, move the camera position and camera target along the same path. In this example, the camera target is placed five elements further along the x-axis than the camera. Also, a small value is added to the camera target x position to prevent the position of the camera and target from becoming the same point if the condition `xd(n) = xd(n+5)` should occur.

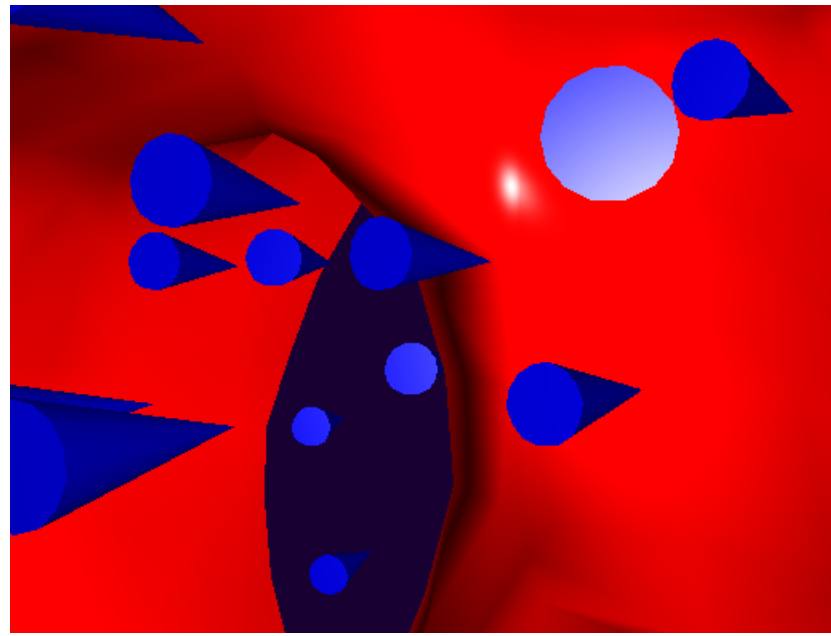
- Update the camera position and camera target so that they both move along the coordinates of the stream line.
- Move the light along with the camera.
- Call `drawnow` to display the results of each move.

```
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 snap shots 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 that the camera points to. 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 "Aspect Ratio" section).
`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 MATLAB's automatic resizing of the scene each time you move the camera by setting the `CameraViewAngleMode` to `manual`.

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

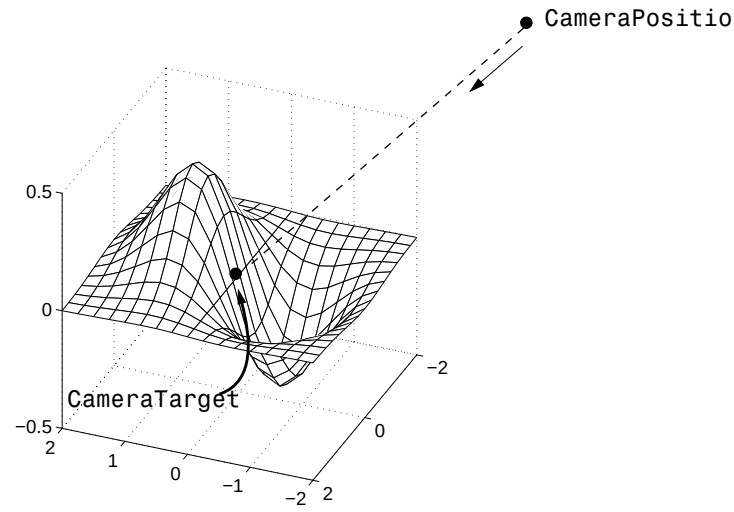
If the Projection is set to perspective, the amount of perspective distortion increases as the camera gets closer to the target and decreases as it gets farther away.

Example – Moving 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's stretch-to-fill behavior and may cause an abrupt change in the aspect ratio. See Aspect Ratio 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')
```

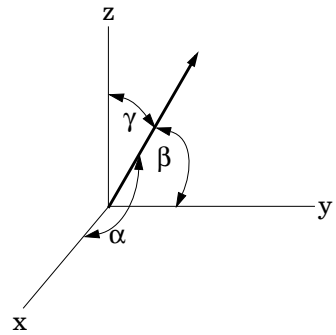
Rotation About the Viewing Axis

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(α×(pi÷180));
YComponent = cos(β×(pi÷180));
ZComponent = cos(γ×(pi÷180));
```

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

Example – Calculating a Camera Up Vector

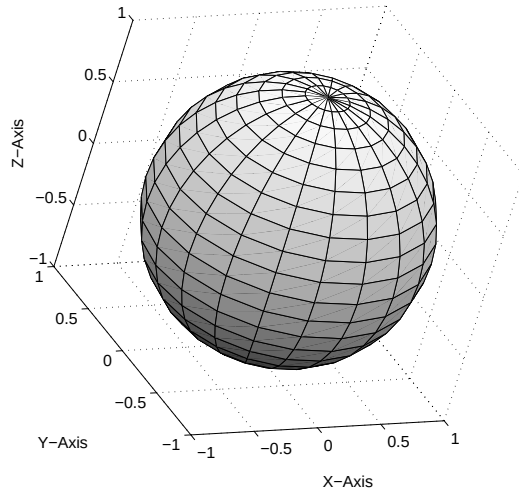
To specify an up vector that makes an angle of 30° with the *z*-axis and lies in the *y*-*z* plane, use the expression

```
upvec = [cos(90*(pi/180)),cos(60*(pi/180)),cos(30*(pi/180))];
```

and then set the `CameraUpVector` property:

```
set(gca,'CameraUpVector',upvec)
```

Drawing a sphere with this orientation produces


1
0.5
0
−0.5
−1
Z−Axis
1
0.5
0
−0.5
−1
Y−Axis
−1
−0.5
0
0.5
1
X−Axis

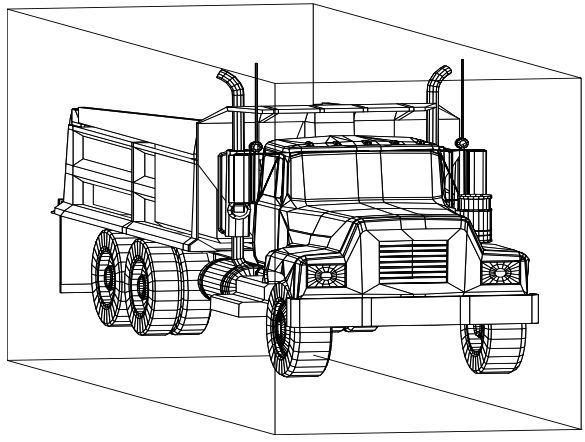
View Projection Types

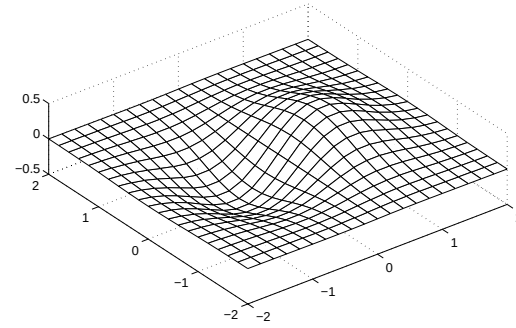
MATLAB supports both orthographic and perspective projection types for displaying 3-D graphics. The one you select depends on the type of graphics you are displaying:

- `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.

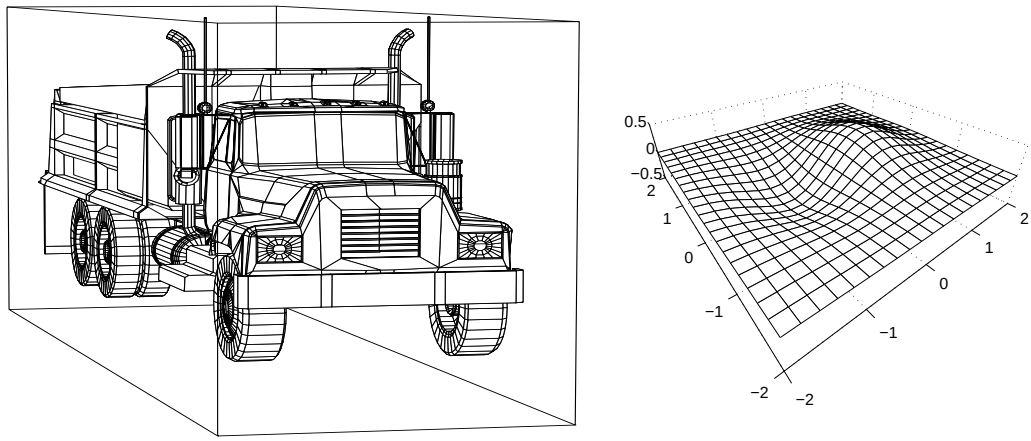
These pictures show a drawing of a dump truck (created with `patch`) and a surface plot of a mathematical function, both using orthographic projection.





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.

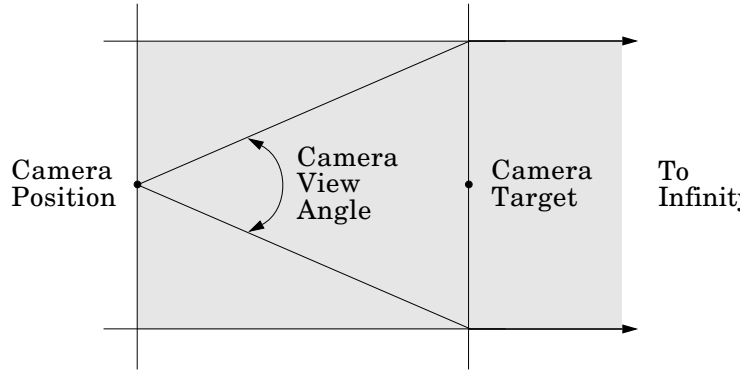


Projection Types and Camera Location

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.

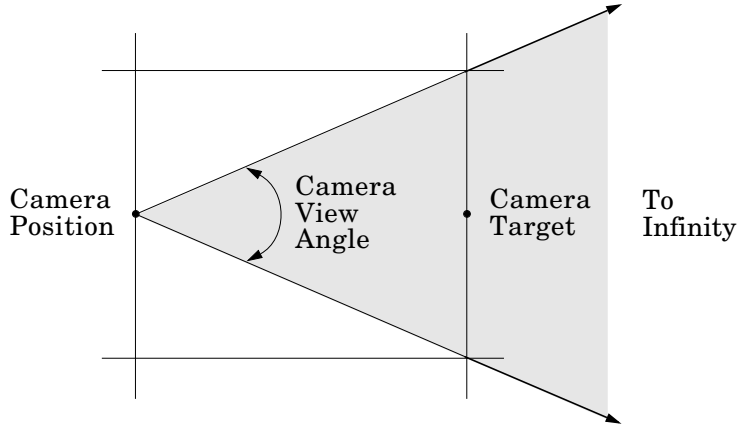
	Orthographic	**Perspective**
Z-buffer	`CameraViewAngle` determines extent of scene at `CameraTarget`	`CameraViewAngle` determines extent of scene from `CameraPosition` to infinity
Painters	All objects display regardless of `CameraPosition`	Not recommended if graphics objects are behind the `CameraPosition`

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



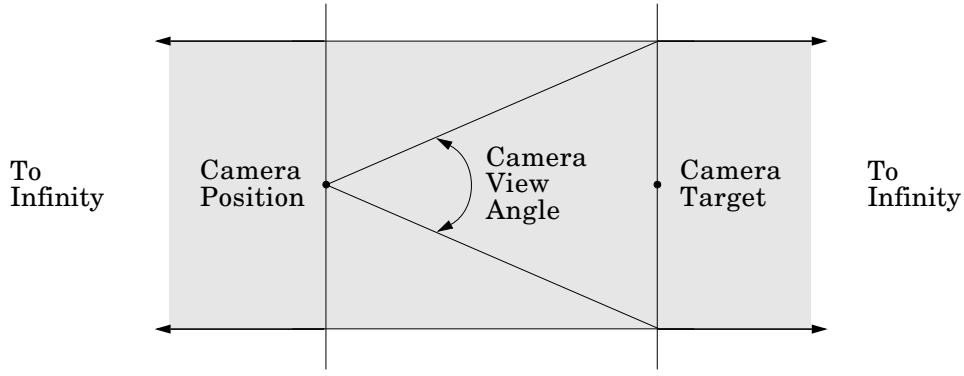
Orthographic projection and Z-buffer renderer

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 printing and renderers 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 of the plotted data and then draws the axes to fit the space available for display. The definition of axes characteristics is controlled by axes graphics object properties. You can specify values for these properties to optimize each graph.

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

Stretch-to-Fill

By default, the size of the axes MATLAB creates for plotting is normalized to the size of the figure window (but is slightly smaller to allow for borders). If you resize the figure, the size and 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`.

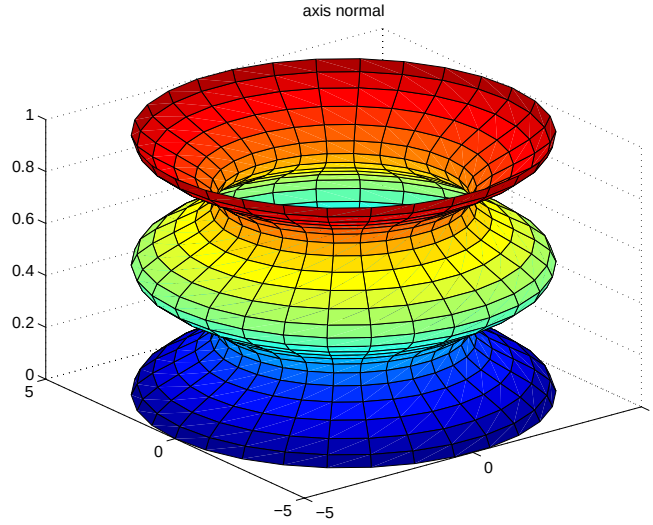
The `axis` command works by manipulating axes graphics object properties.

Example – axis Command Options

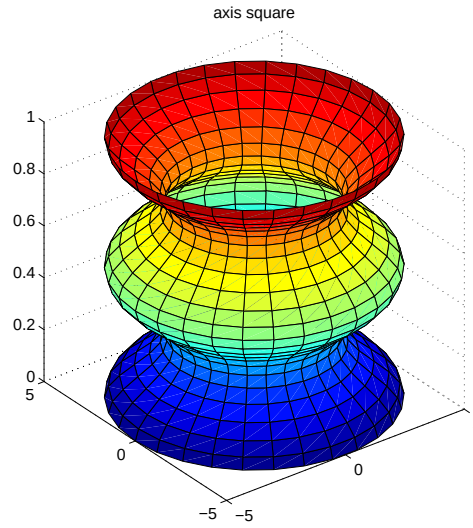
The following three pictures illustrate the effects of three `axis` options on a cylindrical surface created with the statements:

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

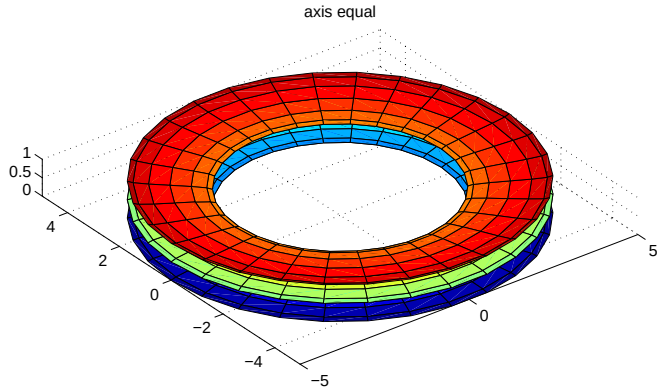
`axis normal` is the default behavior. MATLAB automatically sets the axis limits to span the data range along each axis and stretches the plot to fit the figure window.



`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
`daspect`	Set or query the data aspect ratio
`pbaspect`	Set or query the plot box aspect ratio
`xlim`	Set or query x-axis limits
`ylim`	Set or query y-axis limits
`zlim`	Set or query z-axis limits

See Aspect Ratio Properties for a list of the axes properties that control aspect ratio.

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

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

Much of the data visualized with MATLAB is either:

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

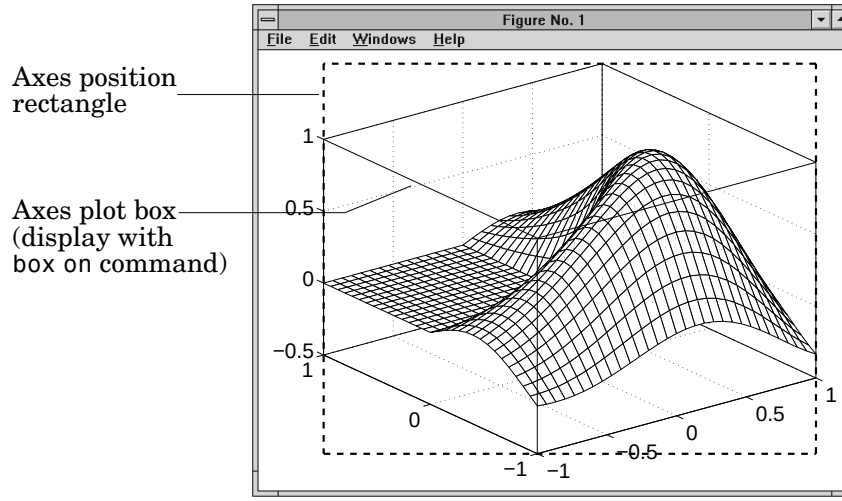
In the first case, it is generally desirable to select axis limits that provide good resolution in each axis direction and to fill the available space. Real-world objects, on the other hand, need to be represented accurately in proportion, regardless of the angle of view.

Default Aspect Ratio Selection

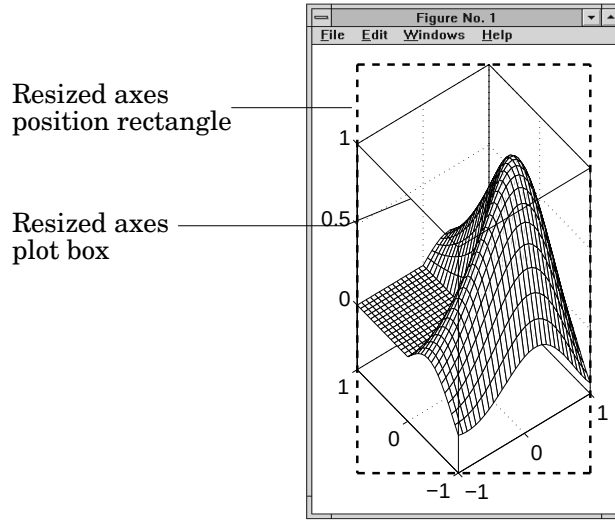
There are two key elements to MATLAB's default behavior – normalizing the axes size to the window size and stretch-to-fill.

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.

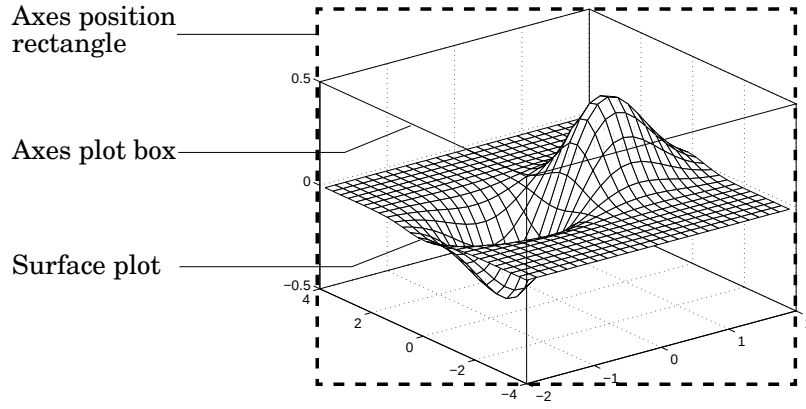


As you can see, reshaping the axes to fit into the figure window can change the aspect ratio of the graph. MATLAB applies stretch-to-fill so the axes fill the position rectangle and in the process may distort the shape. This is generally desirable for graphs of numeric data, but not for displaying objects 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 $-2 \le x \le 2$, $-4 \le y \le 4$, use the statements:

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



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.

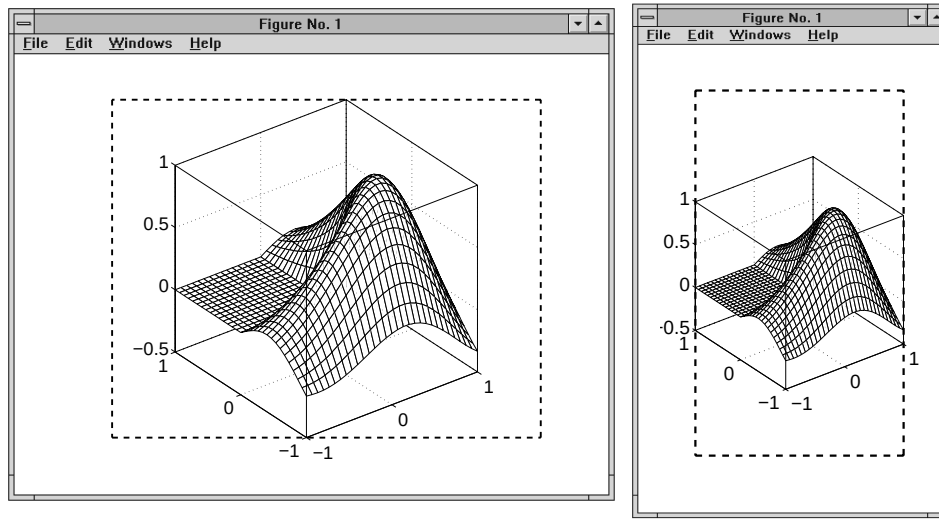
Overriding Stretch-to-Fill

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 $-2 \le x \le 2$, $-4 \le y \le 4$

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

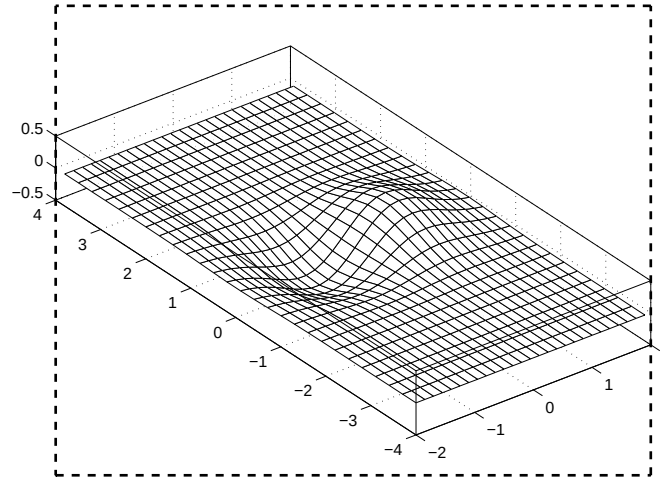
the values are:

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

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

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

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



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

Plot Box Aspect Ratio

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

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

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

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

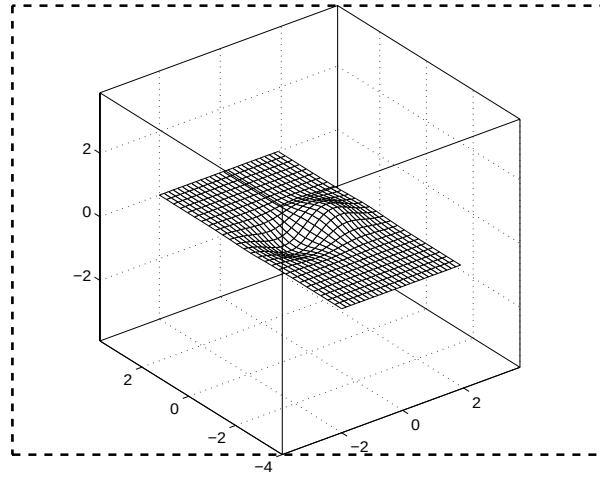
When you set the value of the `DataAspectRatio` and thereby prevent it from changing, MATLAB varies the `PlotBoxAspectRatio` instead. If you specify both the `DataAspectRatio` and the `PlotBoxAspectRatio`,

MATLAB is forced to changed the axis limits to obey the two constraints you have already defined.

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

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

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



Adjusting Axis Limits

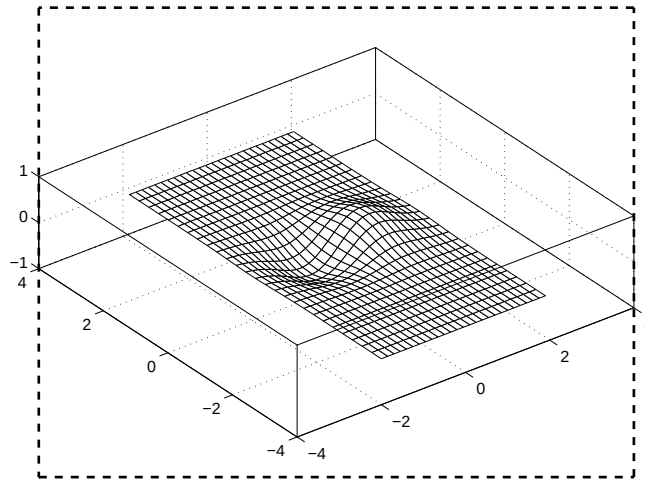
MATLAB enables you to set the axis limits to whichever values you want. However, specifying a value for `DataAspectRatio`, `PlotBoxAspectRatio`, and the axis limits, overconstrains the axes definition. For example, it is not possible for MATLAB to draw the axes if you set these values:

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

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

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

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



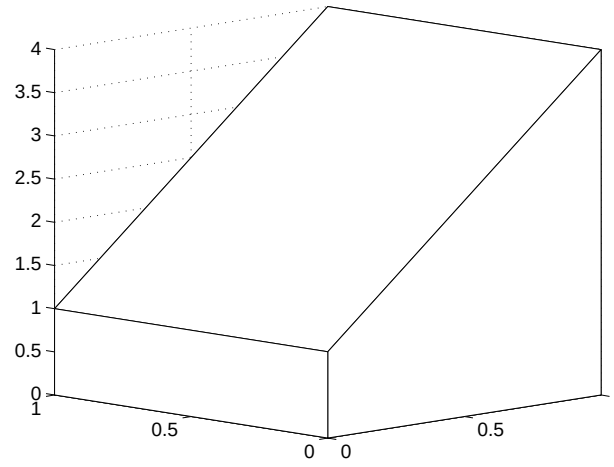
Example – Displaying Real Objects

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

```
vertex_list =

     0     0     0
     0     1     0
     1     1     0
     1     0     0
     0     0     1
     0     1     1
     1     1     4
     1     0     4
```

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

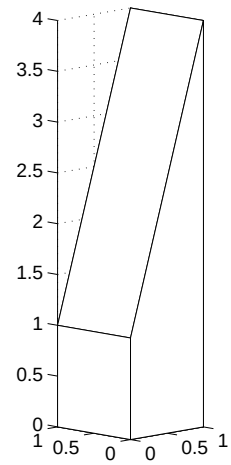
```
patch('Vertices',vertex_list,'Faces',vertex_connection,...
   'FaceColor','w','EdgeColor','k')
view(3)
```



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

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

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


4
3.5
3
2.5
2
1.5
1
0.5
0
1
0.5
0
0
0.5
1

7

Lighting as a Visualization Tool

Lighting

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

Lighting Examples

These examples illustrate the use of lighting in a visualization context.

- Tracing a stream line thought a volume – sets lighting characteristics of surfaces, patches, and lights.
- Using slice planes and cone plots – sets lighting characteristics of objects in a scene independently to achieve a desired result.
- Lighting multiple slice planes independently to visualize fluid flow.
- Combining single-color lit surfaces with interpolated coloring.
- Employing lighting to reveal surface shape – flow data example, surface plot of "sinc" function.

Lighting Commands

MATLAB provides commands that enable you to position light sources and adjust the characteristics of lit objects. These commands include:

Command	Purpose
`camlight`	Create or move a light with respect to the camera position.
`lightangle`	Create or position a light in spherical coordinates.
`light`	Create a light object.
`lighting`	Select a lighting method.
`material`	Set the reflectance properties of lit objects.

You 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 determine 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.

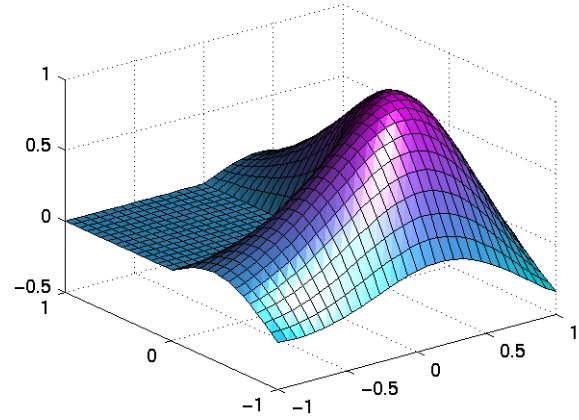
Lights affect surface and patch objects that are in the same axes as the light. These objects have a number of properties that alter the way they look when illuminated by lights.

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])
```

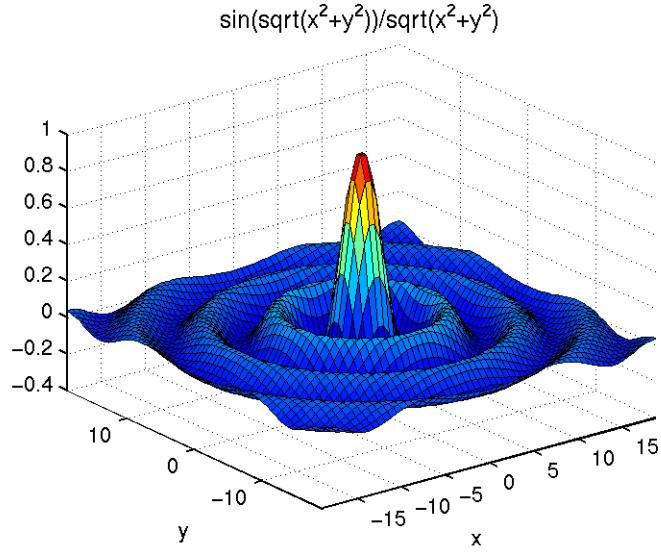
Creating a light activates a number of lighting-related properties controlling characteristics, such as the ambient light and reflectance properties of objects. It also switches to Z-buffer renderer if not already in that mode.



Illuminating Mathematical Functions

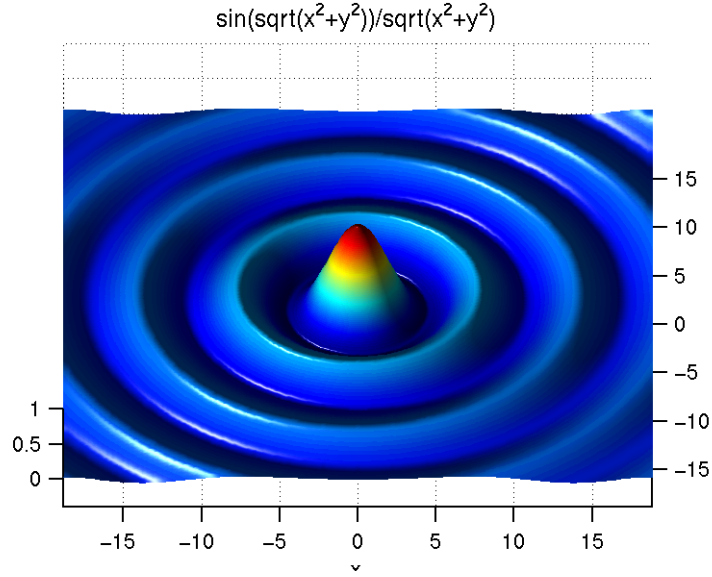
Lighting can enhance surface graphs of mathematical functions. For example, use the `ezsurf` command to evaluate the expression $\sin(\sqrt{x^2 + y^2}) \div \sqrt{x^2 + y^2}$ over the region -6π to 6π.

```
ezsurf('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)',[-6*pi,6*pi])
```



Now add lighting using the `lightangle` command, which accepts the light position in terms of azimuth and elevation.

```
view(0,75)
shading interp
lightangle(-45,30)
set(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 Properties that Affect Lighting 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
`AmbientLightColor`	An axes property that specifies the color of the background light in the scene, which has no direction and affects all objects uniformly. Ambient light effects occur only when there is a visible light object in the axes.
`AmbientStrength`	A patch and surface property that determines the intensity of the ambient component of the light reflected from the object.
`DiffuseStrength`	A patch and surface property that determines the intensity of the diffuse component of the light reflected from the object.
`SpecularStrength`	A patch and surface property that determines the intensity of the specular component of the light reflected from the object.
`SpecularExponent`	A patch and surface property that determines the size of the specular highlight.
`SpecularColorReflectance`	A patch and surface property that determines the degree to which the specularly reflected light is colored by the object color or the light source color.
`FaceLighting`	A patch and surface property that determines the method used to calculate the effect of the light on the faces of the object. Choices are either no lighting, or flat, Gouraud, or Phong lighting algorithms.

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

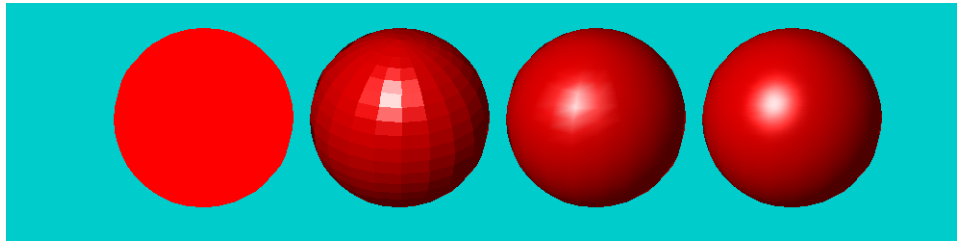
When you add lights to an axes, MATLAB determines the effects these lights have on the patch and surface objects that are displayed in that axes. There are different methods used to calculate the face and edge coloring of lit objects, and the one you select depends on the results you want to obtain.

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:

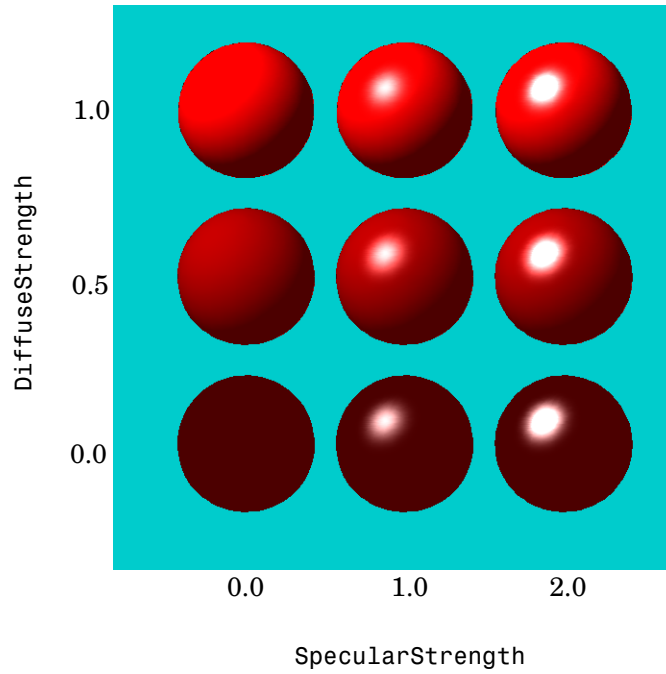
- Specular and diffuse reflection
- Ambient light
- Specular exponent
- Specular color reflectance
- Backface lighting

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

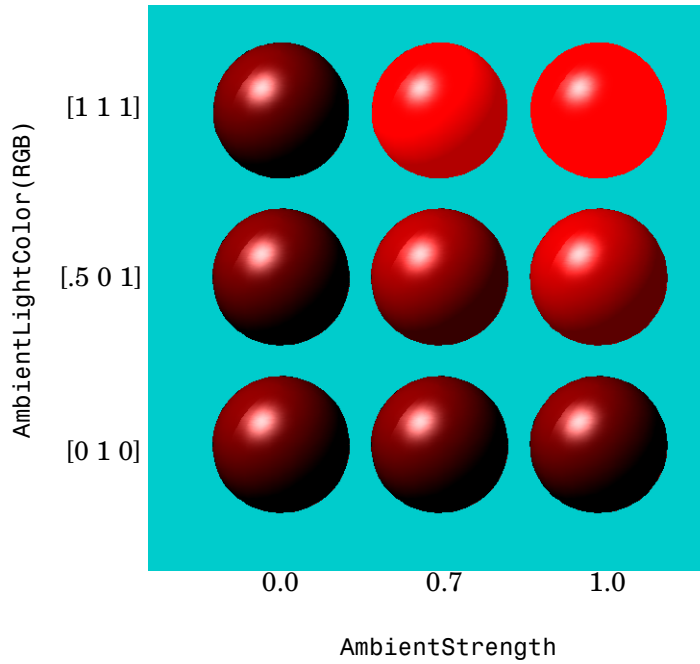
You can control the amount of specular and diffuse reflection from the surface of an object by setting the `SpecularStrength` and `DiffuseStrength` properties. This picture illustrates various settings.



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.

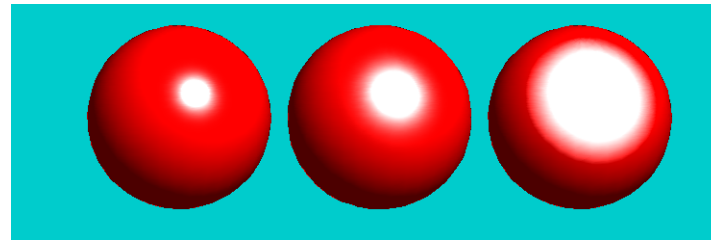


The green [0 1 0] ambient light does not affect the scene because there is no red component in green light. However, the color defined by the RGB values [.5 0 1] does have a red component, so it contributes to the light on the sphere (but less than the white [1 1 1] ambient light).

Specular Exponent

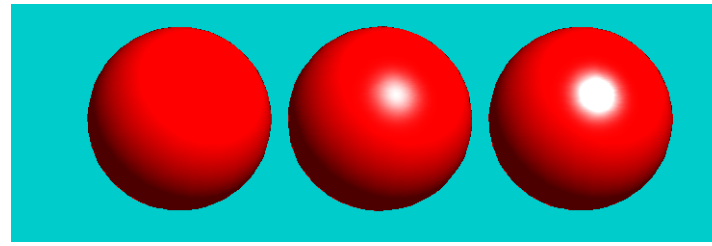
The size of the specular highlight spot depends on the value of the patch and surface object's `SpecularExponent` property. Typical values for this property range from 1 to 500, with normal objects having values in the range 5 to 20.

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



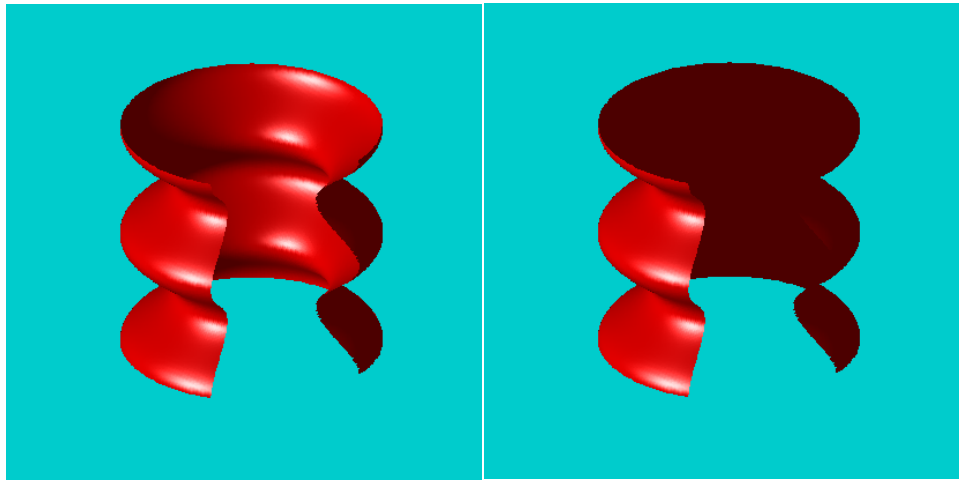
Specular Color Reflectance

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



Back Face Lighting

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

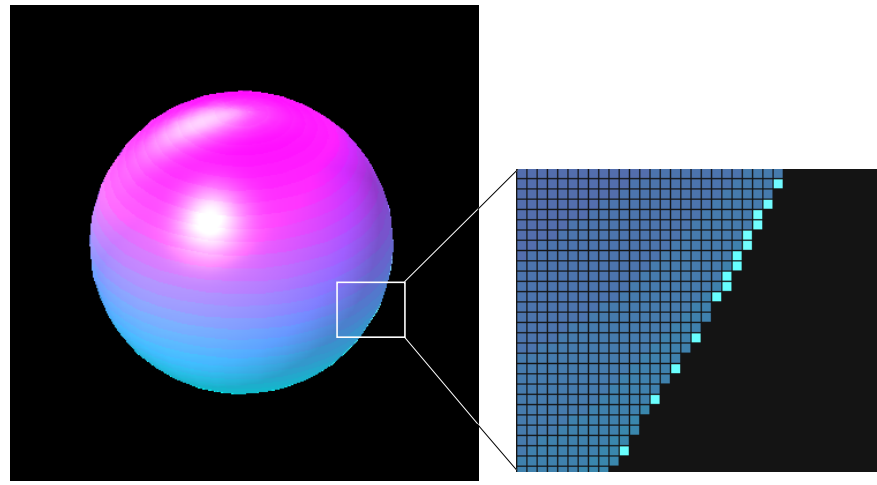


`BackFaceLighting` = `reverselit` `BackFaceLighting` = `unlit`

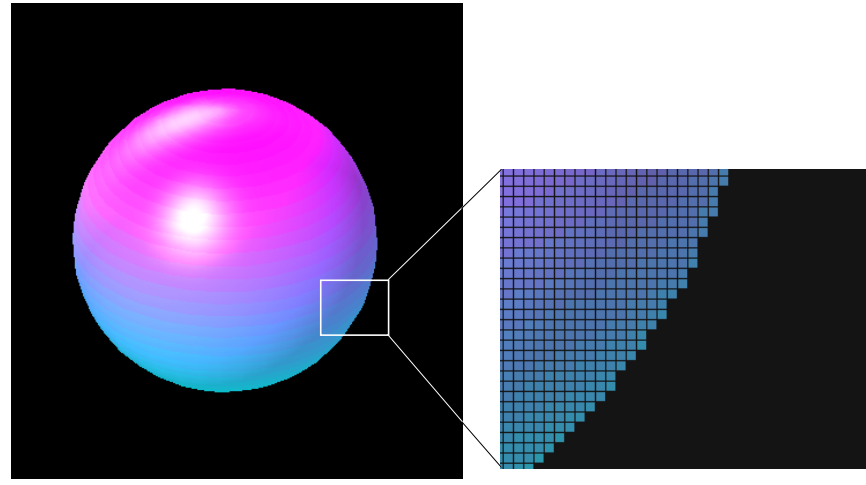
The default value for `BackFaceLighting` is `reverselit`. This setting reverses the direction of the vertex normals that face away from the camera, causing the interior surface to reflect light towards the camera. Setting `BackFaceLighting` to `unlit` disables lighting on faces with normals that point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

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

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



`BackFaceLighting` = `reverselit`



`BackFaceLighting` = `lit`

Positioning Lights in Data Space

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

```
vert =                          fac =

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

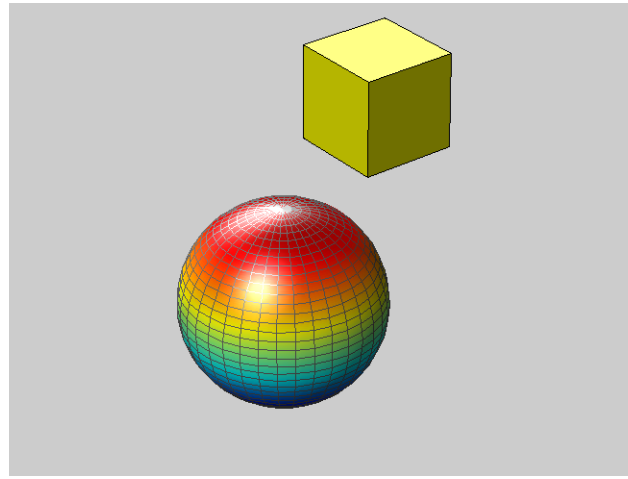
```
sphere(36);
h = findobj('Type','surface');
set(h,'FaceLighting','phong',...
      'FaceColor','interp',...
      'EdgeColor',[.4 .4 .4],...
      'BackFaceLighting','lit')
hold on
patch('faces',fac,'vertices',vert,'FaceColor','y');
light('Position',[1 3 2]);
light('Position',[-3 -1 3]);
material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

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

The patch uses `flat` `FaceLighting` (the default) to enhance the visibility of each side. The surface uses `phong` `FaceLighting` because it produces

the smoothest interpolation of lighting effects. The `material shiny` command affects the reflectance properties of both the cube and sphere (although its effects are noticeable only on the sphere because of the cube's flat shading).

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.

8

Volume Visualization Techniques

Introducing 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. Scalar data contains single values for each point. Vector data contains 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 to 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, cone plots and arrow plots. Most visualizations, however, employ a combination of techniques to best reveal the content of the data.

The sections in this topic area 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:

- Determine the characteristics of your data. Graphing volume data usually requires knowledge of the range of both the coordinates and the data values.
- Select an appropriate plotting routine. The information in this topic area helps you select the right methods.
- Define the view. The information conveyed by a complex 3-D 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.
- 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 Commands

MATLAB supports commands that enable you to apply a variety of volume visualization techniques. The following table lists these functions.

Command	Purpose
`coneplot`	Plot velocity vectors as cones in 3-D vector fields
`contourslice`	Draw contours in volume slice planes
`isocaps`	Compute isosurface end-cap geometry
`isonormals`	Compute normals of isosurface vertices
`isosurface`	Extract isosurface data from volume data
`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
`smooth3`	Smooth 3-D data
`slice`	Draw slice planes in volume
`stream2`	Compute 2-D stream line data
`stream3`	Compute 3-D stream line data
`streamline`	Draw stream lines from 2-D or 3-D vector data
`surf2patch`	Convert surface data to patch data
`subvolume`	Extract subset of volume data set

Each function name in this table links to a description of the function.

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 depends 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 slices 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).
- 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.

Example – 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. Some useful techniques for visualizing this data include displaying the data as:

- 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

The first step is to load the data and transform the data array from 4-D to 3-D.

```
load mri
D = squeeze(D);
```

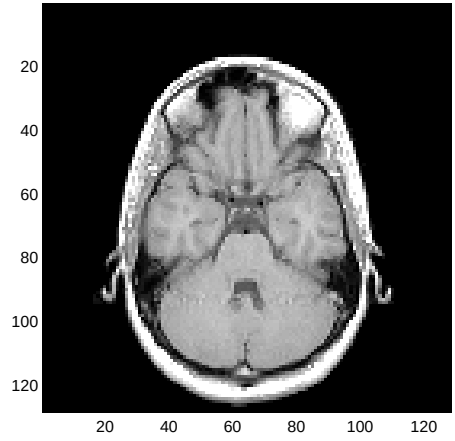
Changing the Data Format

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.

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.

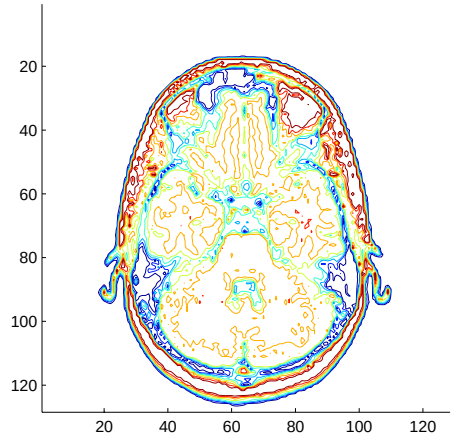
```
x = xlim;
y = ylim;
```

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 the data aspect ratio (`daspect`).

```
contourslice(D,[],[],image_num)
axis ij
xlim(x)
ylim(y)
daspect([1,1,1])
colormap('default')
```

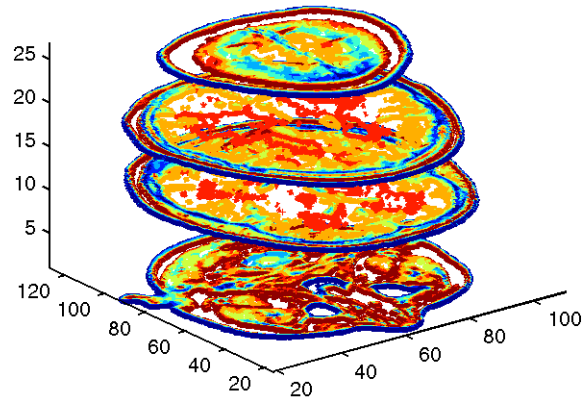
This contour plot uses the figure colormap to map color to contour value.



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

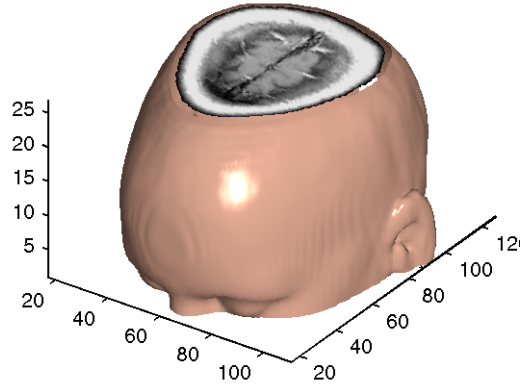
Define the view and set the aspect ratio (`view`, `axis`, `daspect`).

```
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` and the `SpecularExponent` of the isosurface to make the color of the specular reflected light closer to the color of the isosurface and then reduce the size of the specular spot.

```
lightangle(45,30); 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 `isocaps` and `subvolume` 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 slice for examples). 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 stream line 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.

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(:));
```

Creating 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)
```

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')
```

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 was selected initially by using `rotate3d` to determine the best `view`.

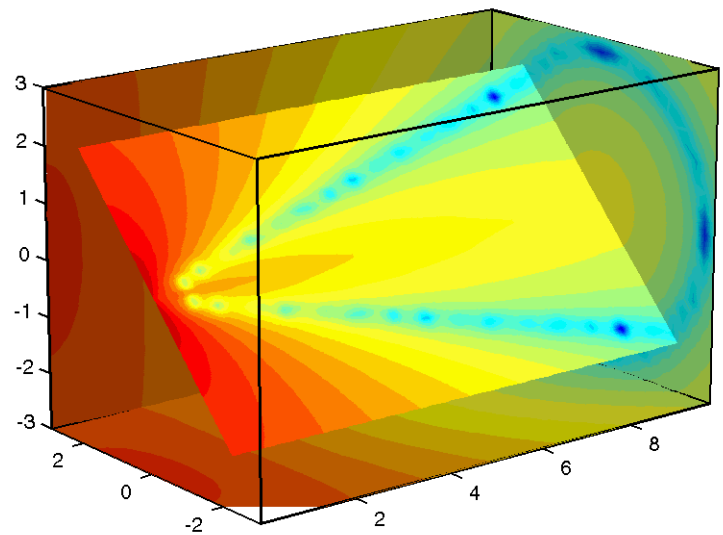
Zooming in on the scene provides a larger view of the volume (`camzoom`). Selecting a projection type of perspective gives the rectangular solid more natural proportions than the default orthographic projection (`camproj`)

```
daspect([1,1,1])
axis tight
box on
view(-38.5,16)
camzoom(1.4)
camproj perspective
```

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))
```



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)))
```

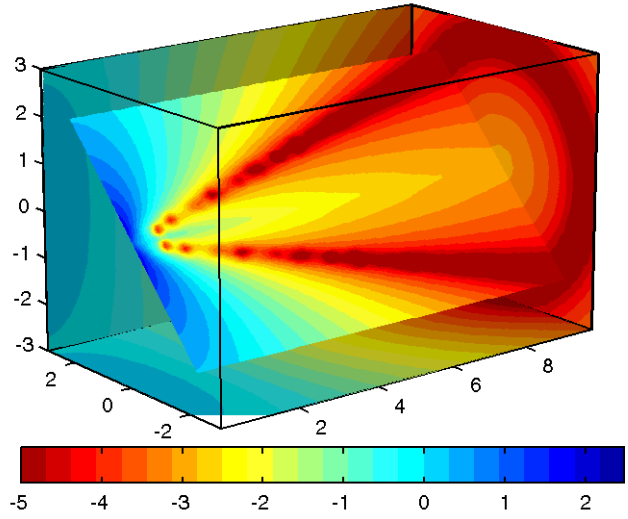
Adjusting the Color Limits

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 to same color (see `caxis` and Color Limits 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')
```


3
2
1
0
-1
-2
-3
2
0
-2
2
4
6
8
-5
-4
-3
-2
-1
0
1
2

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 a 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 generating 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.

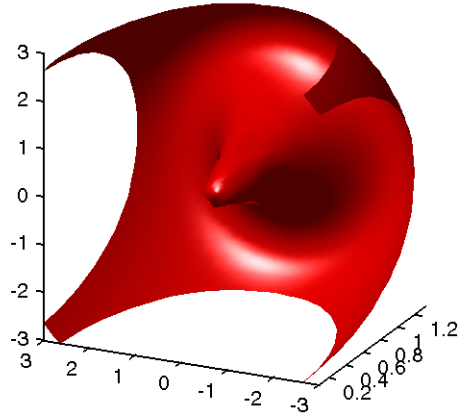
```
min(v(:))
ans =
   -11.5417
max(v(:))
ans =
   2.4832
```

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`).

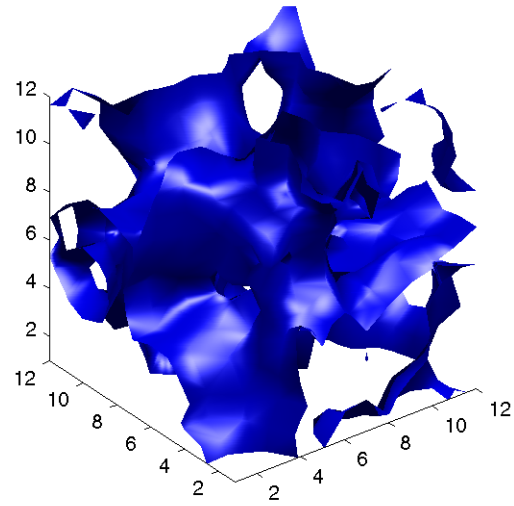
```
hpatch = patch(isosurface(x,y,z,v,0));
isonormals(x,y,z,v,hpatch)
set(hpatch,'FaceColor','red','EdgeColor','none')
daspect([1,4,4])
view([-65,20])
axis tight
camlight left; lighting phong
```



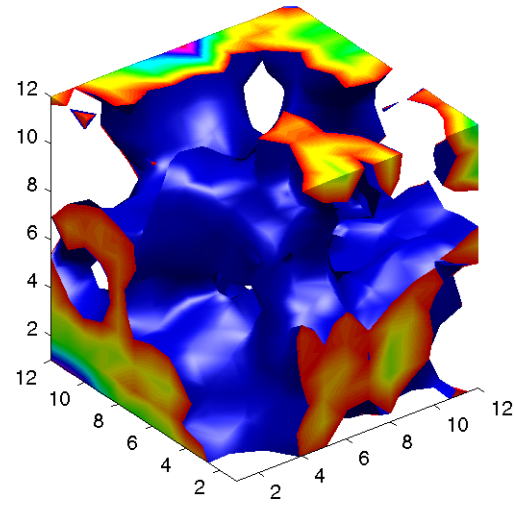
Isocaps Add Context to Visualizations

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 show the effect of adding isocaps to the same isosurface.



Other Isocap Examples

For additional examples of isocaps see:

- 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 four basic steps:

- Generate and process your volume data.
- Create the isosurface and isocaps and set patch properties to control the coloring and lighting.
- Specify the view.
- Add lights to the scene.

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);
```

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 double 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)
```

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
```

Specify 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)
```

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`).

```
camlight right
camlight left
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 two techniques that are useful for visualizing vector data:

- Stream lines (`streamline`) trace the path that a massless particle immersed in the vector field would follow.
- Cone plots (`coneplot`) represent the magnitude and direction of the data at each point by displaying a conical arrowhead or an arrow (`quiver`).

It is typically the case that stream line and cone plots 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 equal 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.

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

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(:));
```

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')
```

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)
```

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

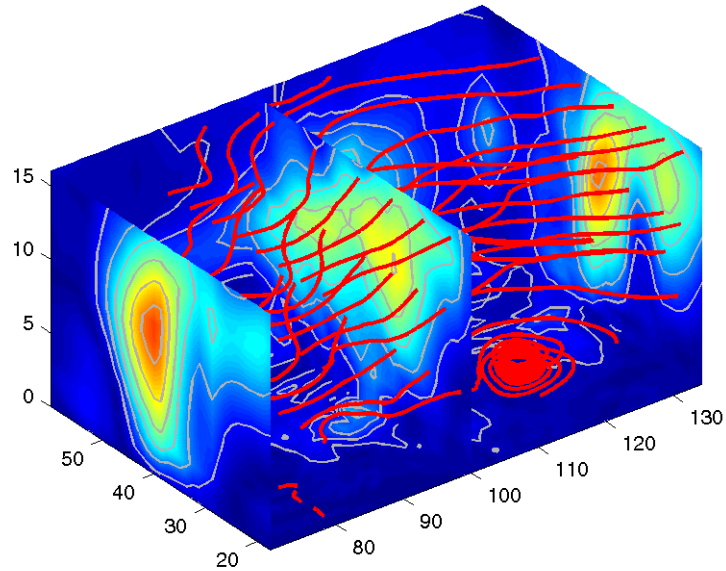
of the stream lines and set the line width and color (`meshgrid`, `streamline`, `LineWidth`, `Color`).

```
[sx,sy,sz] = meshgrid(80,20:10:50,0:5:15);
hlines = streamline(x,y,z,y,v,w,sx,sy,sz);
set(hlines,'LineWidth',2,'Color','r')
```

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.

Vector Data 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.

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');
```

Adding 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
```

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');
```

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');
```

Define the View

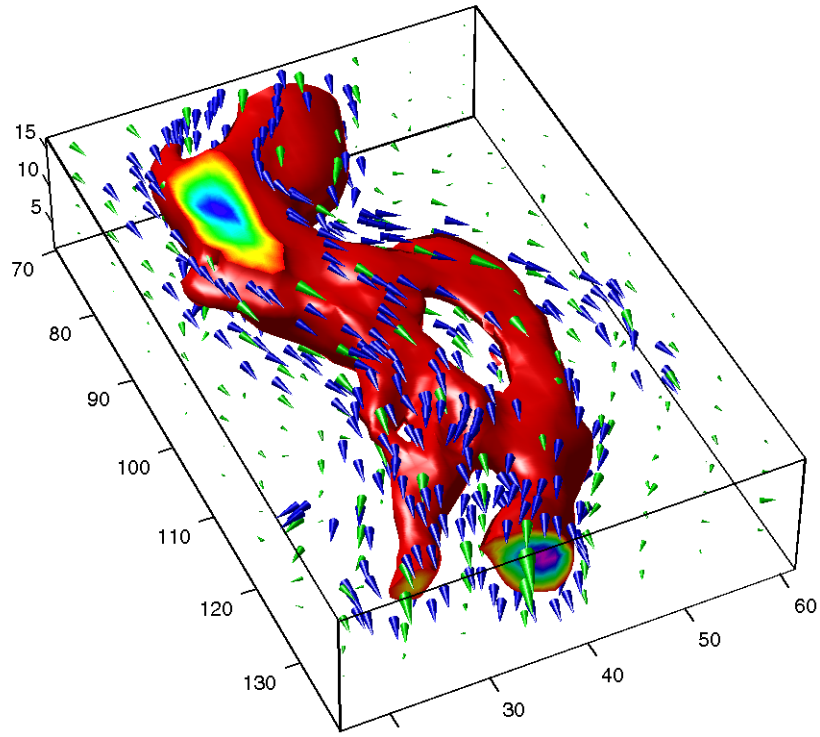
- Use the `axis` command to set the axis limits equal to the minimum and maximum values of the data and enclose the graph in a box to improve the sense of a volume (`box`).
- Set the projection type to perspective to create a more natural view of the volume. Set the view point and zoom in to make the scene larger (`camproj`, `camzoom`, `view`).

```
axis tight
box on
camproj perspective
camzoom(1.25)
view(65,45)
```

Add Lighting

Add a light source and use Phong lighting for the smoothest lighting of the isosurface. Increase the strength of the background lighting on the isocaps to make them brighter (`camlight`, `lighting`, `AmbientStrength`).

```
camlight(-45,45)
lighting phong
set(hcap,'AmbientStrength',.6)
```


15
10
5
70
80
90
100
110
120
130
30
40
50
60

9

Creating 3-D Models with Patches

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 some mathematical functions, the contours of data in a rectangular plane, or parameterized surfaces such as a sphere.

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 `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 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 provides more information on options for coloring patches.

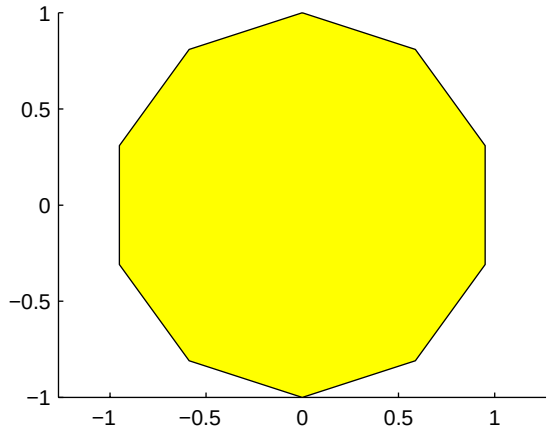
Creating a Single Polygon

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
```

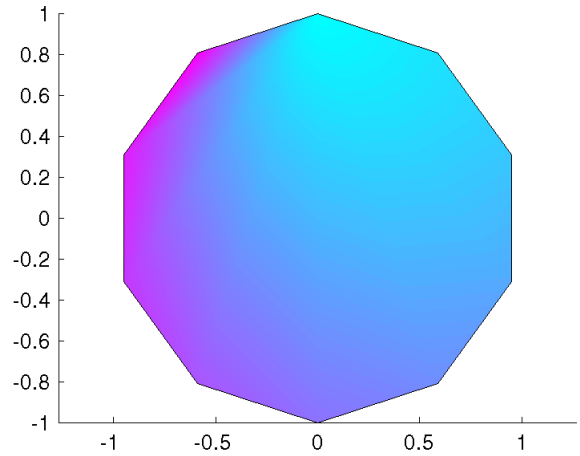


The first and last vertices need not coincide; MATLAB automatically closes each polygonal face of the patch. In fact, it is generally better to define each vertex only once, particularly if you are using interpolated face coloring.

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 provides more information on options for coloring patches.

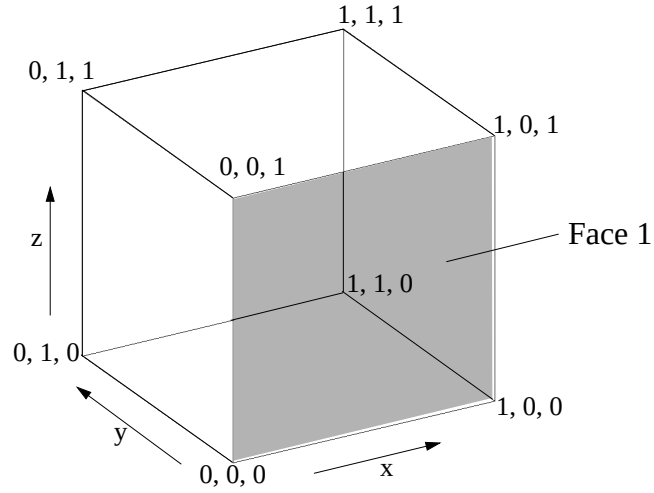
Multi-Faceted Patches

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.

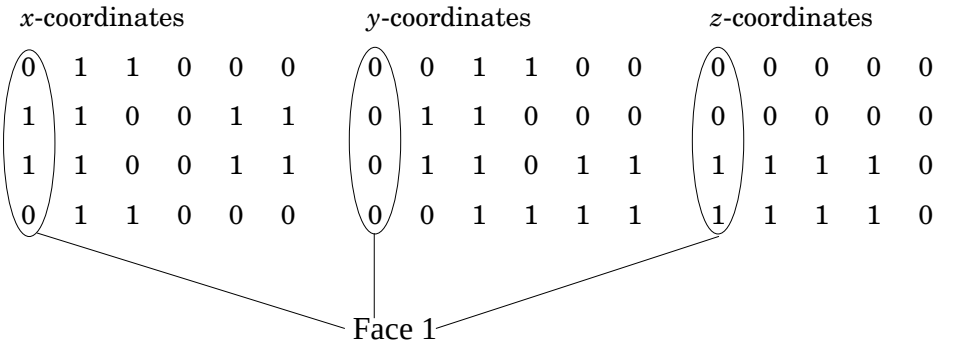
Example – Defining a Cube

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

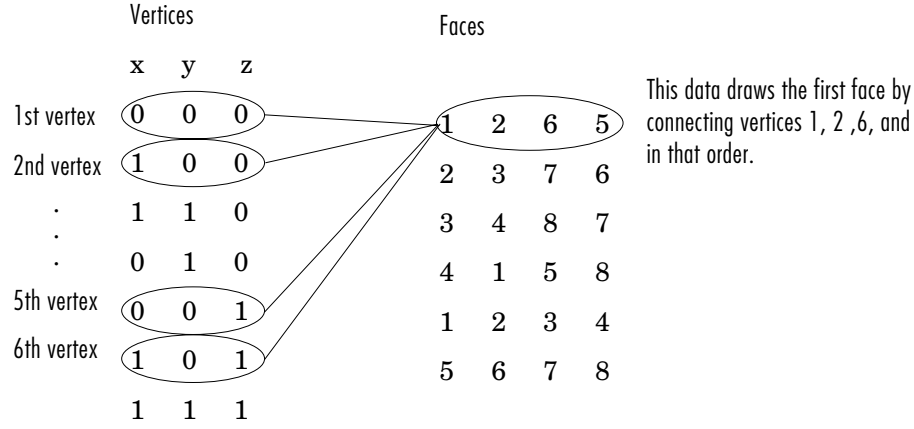
Each of the six faces has four vertices. Since you do not need to close each polygon (i.e., the first and last vertices do not need to be the same), you can define this cube using a 4-by-6 matrix for each of the *x*-, *y*-, and *z*-coordinates.



Each column of the matrices specifies a different face. Note that while there are only eight vertices, you must specify 24 vertices to define all six faces. Since each face shares vertices with four other faces, you can define the patch more efficiently by defining each vertex only once and then specifying the order in which to connect these vertices to form each face. The patch `Vertices` and `Faces` properties define patches in just this way.

Specifying Faces and Vertices

These matrices specify the cube using `Vertices` and `Faces`:



Using the vertices/faces technique can save a considerable amount of computer memory when patches contain a large number of faces. This technique requires

the formal `patch` function syntax, which entails assigning values to the `Vertices` and `Faces` properties explicitly. For example,

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix)
```

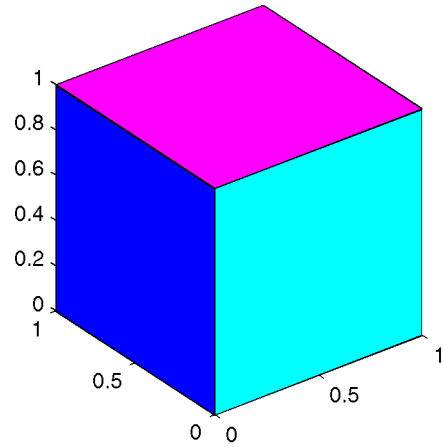
Since the formal syntax does not automatically assign face or edge colors, you must set the appropriate properties to produce patches with colors other than the default white face color and black edge color.

Flat Face Color

Flat face color is the result of specifying one color per face. For example, using the vertices/faces technique and the `FaceVertexCData` property to define color, this statement specifies one color per face and sets the `FaceColor` property to `flat`.

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix,...
      'FaceVertexCData',hsv(6),'FaceColor','flat')
```

Since true color specified with the `FaceVertexCData` property has the same format as a MATLAB colormap (i.e., an *n*-by-3 array of RGB values), this example uses the `hsv` colormap to generate the six colors required for flat shading.



Interpolated Face Color

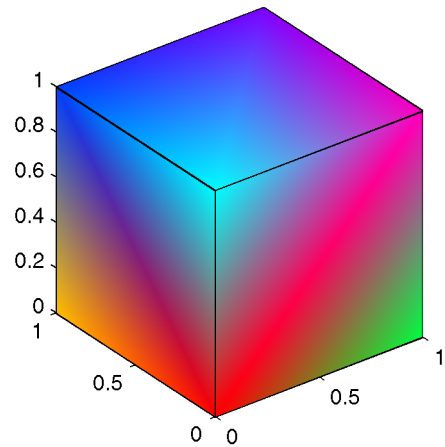
Interpolated face color means the vertex colors of each face define a transition of color from one vertex to the next. To interpolate the colors between vertices, you must specify a color for each vertex and set the `FaceColor` property to `interp`.

```
patch('Vertices',vertex_matrix,'Faces',faces_matrix,...
      'FaceVertexCData',hsv(8),'FaceColor','interp')
```

Changing to the standard 3-D view and making the axis square,

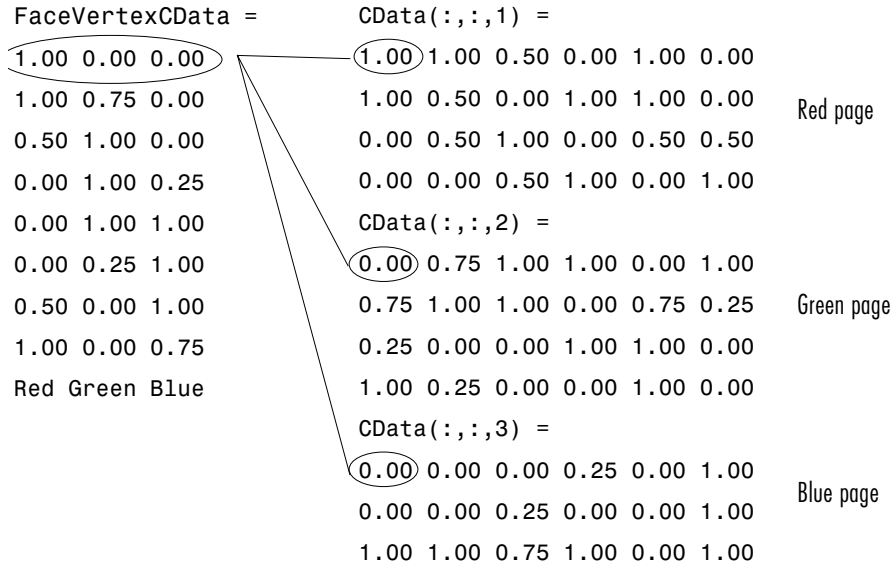
```
view(3); axis square
```

produces a cube with each face colored by interpolating the vertex colors.



To specify the same coloring using the x, y, z, c technique, c must be an *m*-by-*n*-by-3 array, where the dimensions of x, y, and z are *m*-by-*n*.

This diagram shows the correspondence between the `FaceVertexCData` and `CData` properties.



See “Specifying Patch Coloring” for a discussion of coloring techniques in more detail.

Specifying Patch Coloring

Patch objects employ a coloring scheme that is basically different from that used by surface objects in that patches do not automatically generate color data based on the value of the *z*-coordinate at each vertex. You must explicitly specify patch coloring, or MATLAB uses the default white face color and black edge color.

Patch coloring methods provide a means to display pictures of real-world objects with information superimposed on them through the use of color. For example a picture of an airplane wing can be colored to indicate the air pressure across its surface.

This table summarizes the patch properties that control color (exclusive of those used when light sources are present). See patch properties for a complete list of properties.

Property	Purpose
`CData`	Specify single, per face, or per vertex colors in conjunction with x, y, and z data.
`CDataMapping`	Specifies whether color data is scaled or used directly as indices into the figure colormap.
`FaceVertexCData`	Specify single, per face, or per vertex colors in conjunction with faces and vertices data.
`EdgeColor`	Specifies whether edges are invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors.
`FaceColor`	Specifies whether faces are invisible, a single color, a flat color determined by vertex colors, or interpolated colors determined by vertex colors.
`MarkerEdgeColor`	Specifies the color of the marker, or the edge color for filled markers.

Property	Purpose
`MarkerFaceColor`	Specifies the fill color for markers that are closed shapes.

Face and Edge Coloring

You can specify patch face coloring by defining:

- A single color for all faces
- One color for each face, which is used for flat coloring
- One color for each vertex, which is used for interpolated coloring

Specify the face color using either the `CData` property, if you are using *x*-, *y*-, and *z*-coordinates or the `FaceVertexCData` property, if you are specifying vertices and faces.

Each patch face has a bounding edge, which you can color as:

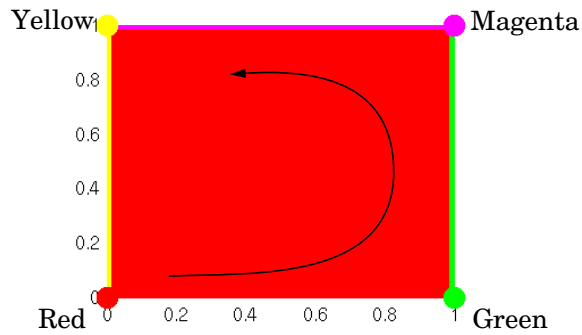
- A single color for all edges
- A flat color defined by the color of the vertex that precedes the edge
- Interpolated colors determined by the two vertices that bound the edge

Note that patch edge colors can be flat or interpolated only when you specify a color for each vertex. For flat edge coloring, MATLAB uses the color of the vertex preceding the edge to determine the color of the edge. The order in which you specify the vertices establishes which vertex colors a particular edge.

Example – Specifying Flat Edge and Face Coloring

These statements create a square patch.

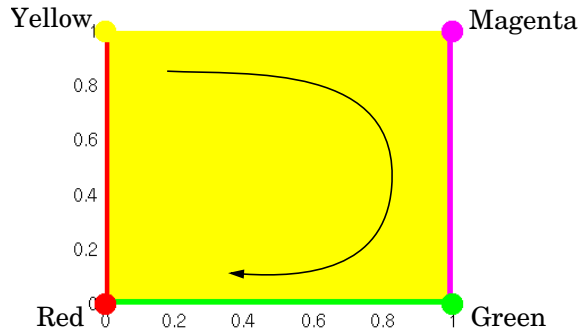
```
v  = [0 0 0;1 0 0;1 1 0;0 1 0];
f = [1 2 3 4];
fvc = [1 0 0;0 1 0;1 0 1;1 1 0];
patch('Vertices',v,'Faces',f,'FaceVertexCData',fvc,...
      'FaceColor','flat','EdgeColor','flat',...
      'Marker','o','MarkerFaceColor','flat')
```



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.

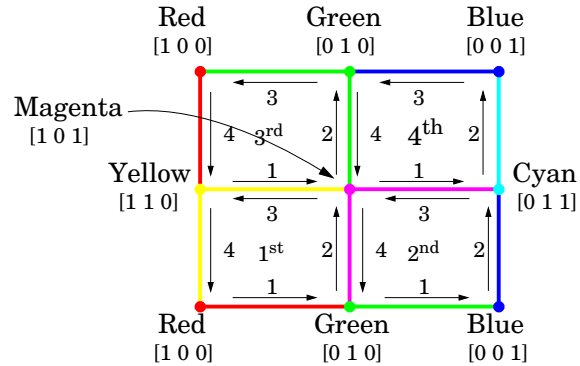


Coloring Edges with Shared Vertices

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.

How MATLAB Interprets Patch Color Data

MATLAB interprets the color data in one of two ways:

- Indexed color data – numerical values that are mapped to colors defined in the figure colormap
- Truecolor data – RGB triples that define colors explicitly and do not make use of the figure colormap

The dimensions of the color data (`CData` or `FaceVertexCData`) determine how MATLAB interprets it. If you specify only one numeric value per patch, per face, or per vertex, then MATLAB interprets the data as indexed. If there are three numeric values per patch, face, or vertex, then MATLAB interprets the data as RGB values.

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

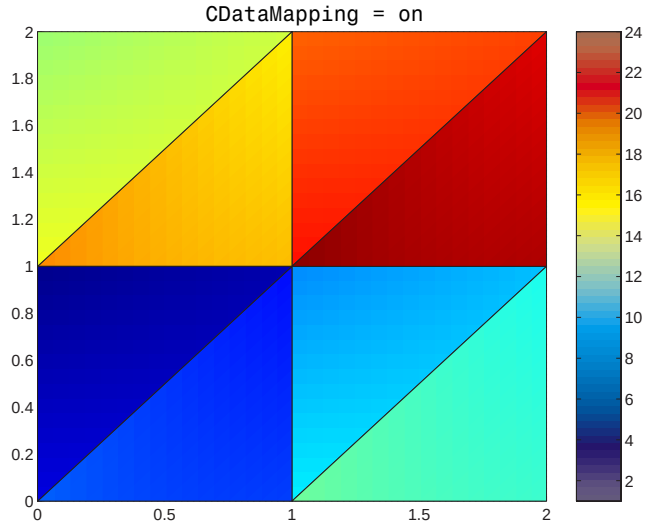
By default, MATLAB scales the color data so that the minimum value maps to the first color in the colormap, the maximum value maps to the last color in the colormap, and values in between are linearly transformed to span the colormap. This enables you to use colormaps of different sizes without changing 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 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)))
```

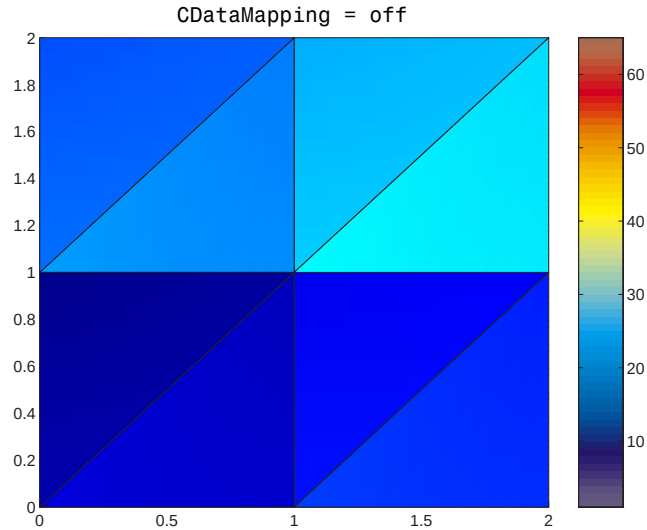
Direct Color

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 > `length(colormap)`, it maps to the last color in the colormap.

Unscaled color data is more commonly used for images where there is typically a colormap associated with a particular image.

Truecolor Patches

Truecolor is a means to specify a color explicitly with RGB values rather than pointing to an entry in the figure colormap. Truecolor generally provides a greater range of colors than can be defined in a colormap.

Using truecolor eliminates the mapping of data to colormap entries. On the other hand, you cannot change the coloring of the patch without redefining the color data (as opposed to just changing the colormap).

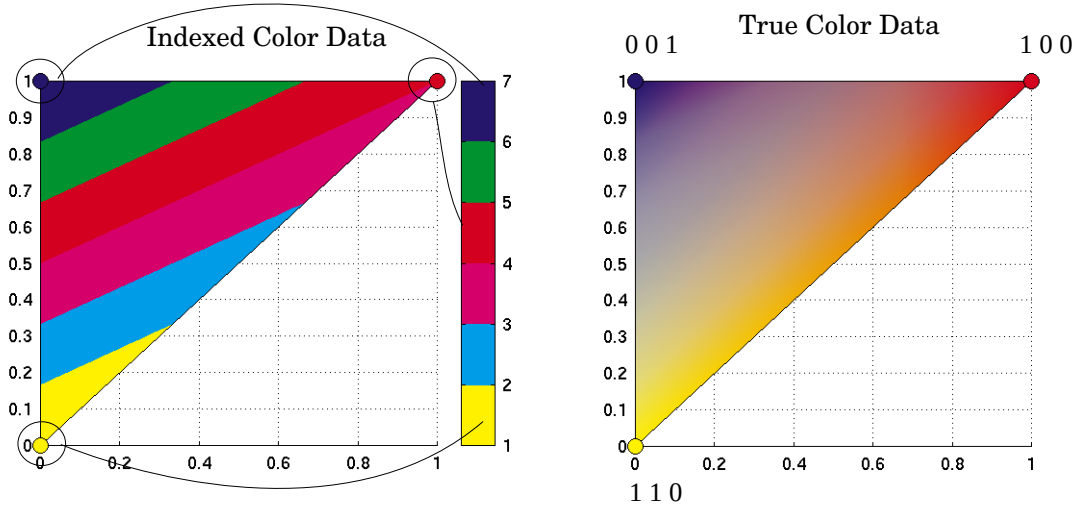
You can use truecolor on computers that do not have true color (24-bit) displays. In this case, MATLAB uses a special colormap designed to produce results that are as close as possible with the limited number of colors available. Properties control how MATLAB uses color on pseudocolor machines..

Interpolating in Indexed Color vs. Truecolor

When you specify interpolated face coloring, MATLAB determines the color of each face by interpolating the vertex colors. The method of interpolation depends on whether you specified truecolor data or indexed color data.

With truecolor data, MATLAB interpolates the numeric RGB values defined for the vertices. This generally produces a smooth variation of color across the face. In contrast, indexed color interpolation uses only colors that are defined in the colormap. With certain colormaps, the results can be quite different.

To illustrate this difference, these two patches are defined with the same vertex colors. Circular markers indicate the yellow, red, and blue vertex colors.



The patch on the left uses indexed colors obtained from the six-element colormap shown next to it. The color data maps the vertex colors to the colormap elements indicated in the picture. With this colormap, interpolating from the cyan vertex to the blue vertex can include only the colors green, red, yellow, and magenta, hence the banding.

Interpolation in RGB space makes no use of the colormap. It is simply the gradual transition from one numeric value to another. For example, interpolating from the cyan vertex to the blue vertex follows a progression similar to these values:

```
0 1 1, 0 0.9 1, 0 0.8 1, ... 0 0.2 1, 0 0.1 1, 0 0 1
```

In reality each pixel would be a different color so the incremental change would be much smaller than illustrated here.

10

Displaying Bit-Mapped Images

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
`axis`	Plot axis scaling and appearance	Display
`image`	Display image (create image object)	Display
`imagesc`	Scale data and display as image	Display
`imread`	Read image from graphics file	File I/O

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

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”.

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 of MATLAB’s 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.

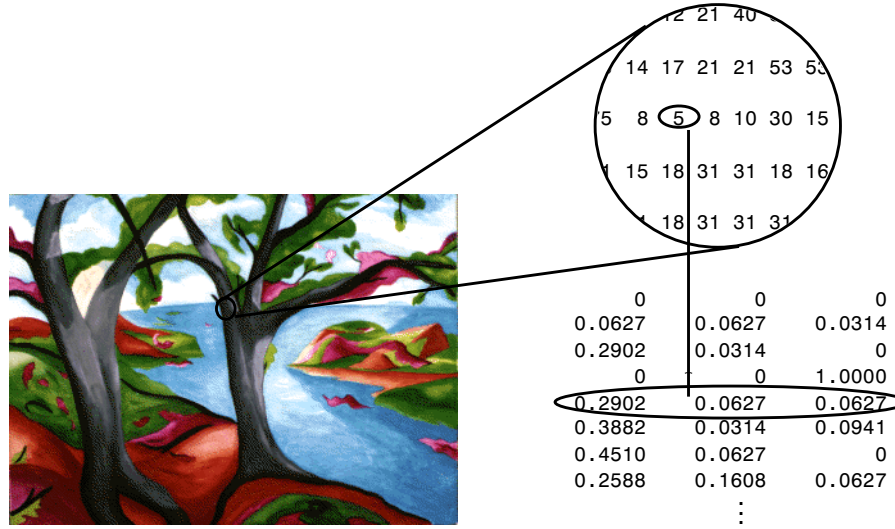
Indexed Images

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.

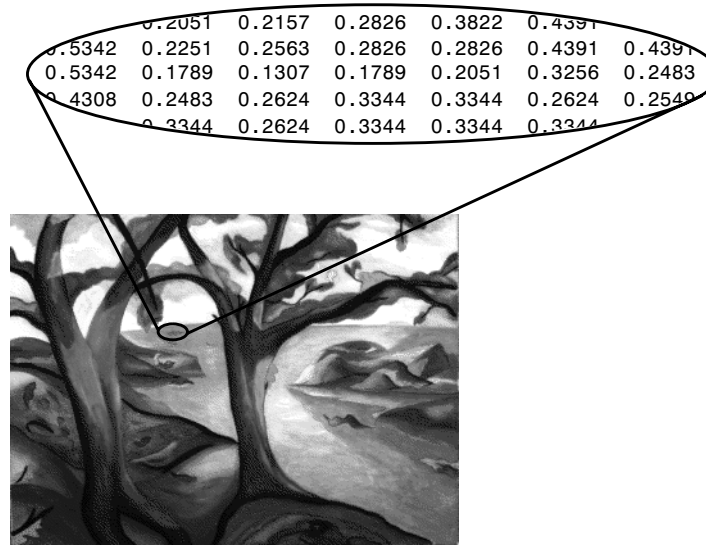


The relationship between the values in the image matrix and the colormap depends on the class of the image matrix. If the image matrix is of class `double`, the value 1 points to the first row in the colormap, the value 2 points to the second row, and so on. If the image matrix is of class `uint8` or `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 (Truecolor) Images

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

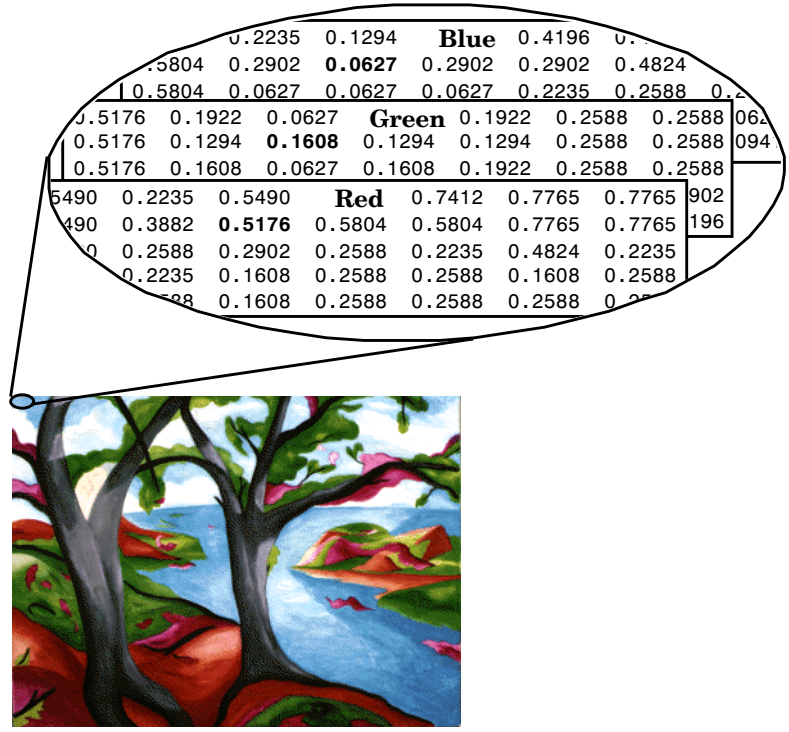
An RGB MATLAB array can be of class `double`, `uint8`, or `uint16`. In an RGB array of class `double`, each color component is a value between 0 and 1. A pixel whose color components are (0,0,0) displays as black, and a pixel whose color components are (1,1,1) displays as white. The three color components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

```
image(RGB)
```

If MATLAB is running on a computer that does not have hardware support for truecolor image display, MATLAB uses color approximation and dithering to display an approximation of the image. See "Dithering Truecolor 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 of MATLAB's 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 `unit16`.

```
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 (Truecolor)	Image is an m-by-n-by-3 array of floating-point values in the range [0, 1].	Image is an m-by-n-by-3 array of integers in the range [0, 255] or [0, 65535].

Reading, Writing, and Querying Graphics Image Files

In 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`.

See the next table for 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	`load` `save`
Load or Save Graphics File Format Image, e.g. BMP, TIFF	`imread` `imwrite`
Display Any Image Loaded Into MATLAB	`image` `imagesc`
Utilities	`imfinfo` `ind2rgb`

Reading a Graphics Image

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

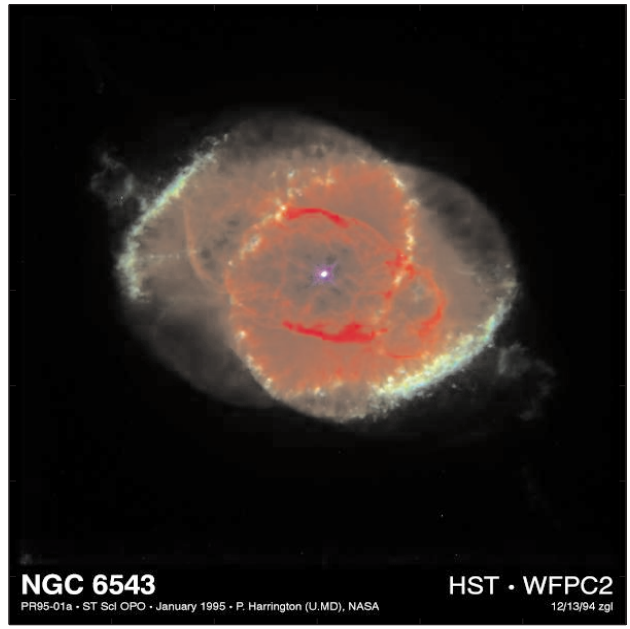
- Name of the file, including the directory path if the file is not in the current directory
- File format
- Version number of the file format

- 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,

```
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.

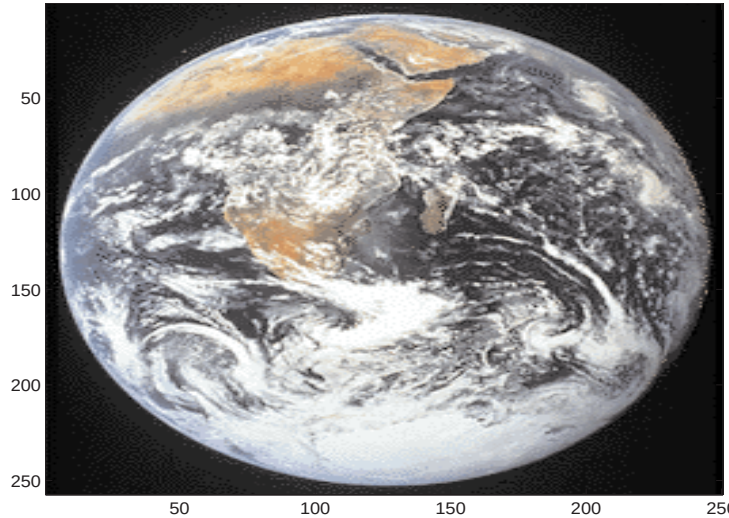
Image Type	Display Commands	Uses Colormap Colors
Indexed	`image(X); colormap(map)`	Yes
Intensity	`imagesc(I,[0 1]); colormap(gray)`	Yes
RGB (truecolor)	`image(RGB)`	No

Controlling Aspect Ratio and Display Size

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

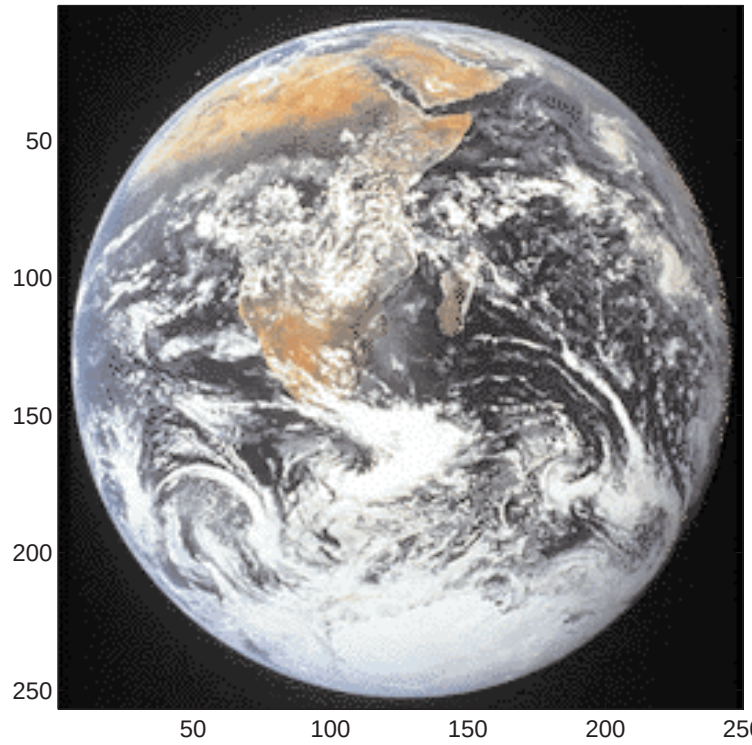
For example, these commands display the `earth` image in the `demos` directory using the default figure and axes positions.

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



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

```
axis image
```

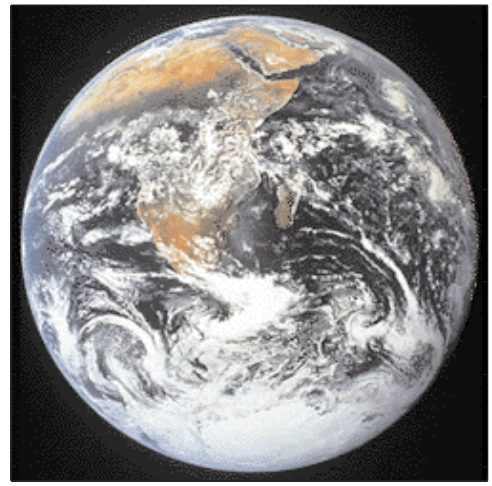


The command `axis image` works by setting the `DataAspectRatio` property of the axes object to [1 1 1]. See `axis` and `axes` for more information on how to control the appearance of axes objects.

Sometimes you 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

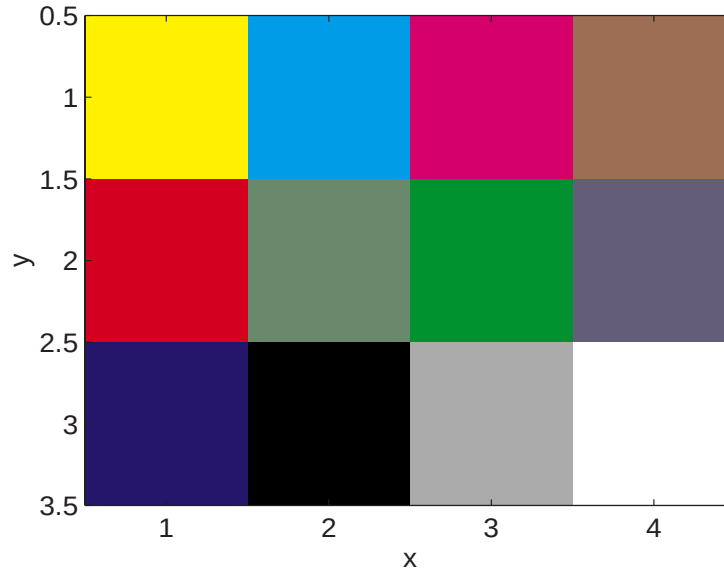
The `XData` and `YData` properties control the coordinate system of the image. For an *m*-by-*n* image, the default `XData` is `[1 n]` and the default `YData` is `[1 m]`. These settings imply the following:

- 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:



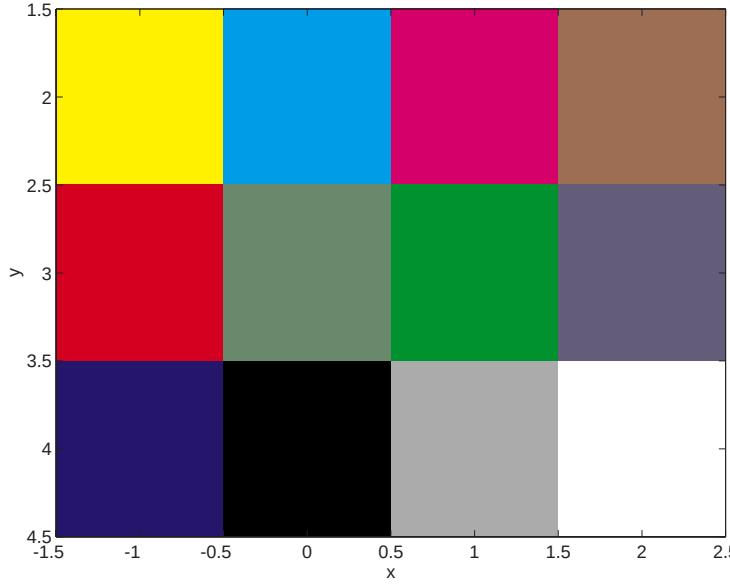
The `XData` and `YData` properties of the resulting image object have the default values shown below.

```
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 though 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 specifcs 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.

11

Printing MATLAB Graphics

Introduction

MATLAB provides a number of different methods for producing graphical output from figures. These methods include ways to:

- Print from the menu or print from the command line
- Use MATLAB's built-in print engine or use system-specific printing services
- Print directly to hardcopy or create graphics-format files to incorporate in documents or other applications

The method you use depends on what you want to accomplish. For example, the simplest way to produce output is to choose the **Print** option from the **File** menu, but if you want to print from an M-file, you need to use the `print` command. If you want to produce graphics to use in other applications, there are many options, depending on your platform and the file format you want to use. This chapter discusses all of these methods and provides guidelines for choosing among them.

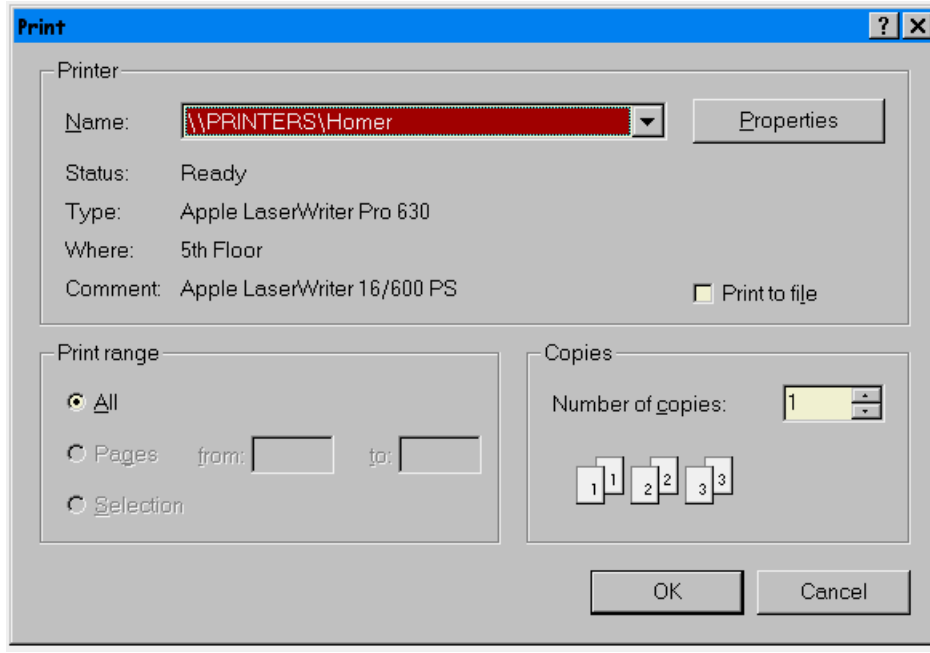
Printing from the Menu

This section discusses how to print a figure by selecting options from the **File** menu of the figure window. This is the simplest way to print in MATLAB and is the preferred method in many cases.

Printing differs depending on which platform you are running MATLAB. Read the section corresponding to the platform you are using. Also see Page Setup for information about adjusting the size, color, and location of the graphic on the page.

PC

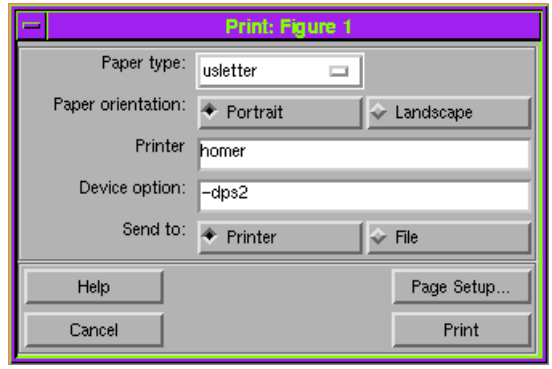
To print a figure, choose the **Print** option from the figure window's **File** menu. MATLAB brings up the Windows print dialog box



Fill in the dialog box, and then click the **OK** button to print the figure.

UNIX

To print a figure, choose the **Print** option from the figure window's **File** menu. MATLAB brings up the print dialog box:



Fill in the dialog box, and then click the button in the lower right corner of the box. This button is labeled **Print** if Send to Printer is selected and **Save** if Send to File is selected. If you are saving to a file, MATLAB displays another dialog box where you specify the filename.

See "Graphic File Formats" for a list of options for the **Device option** field.

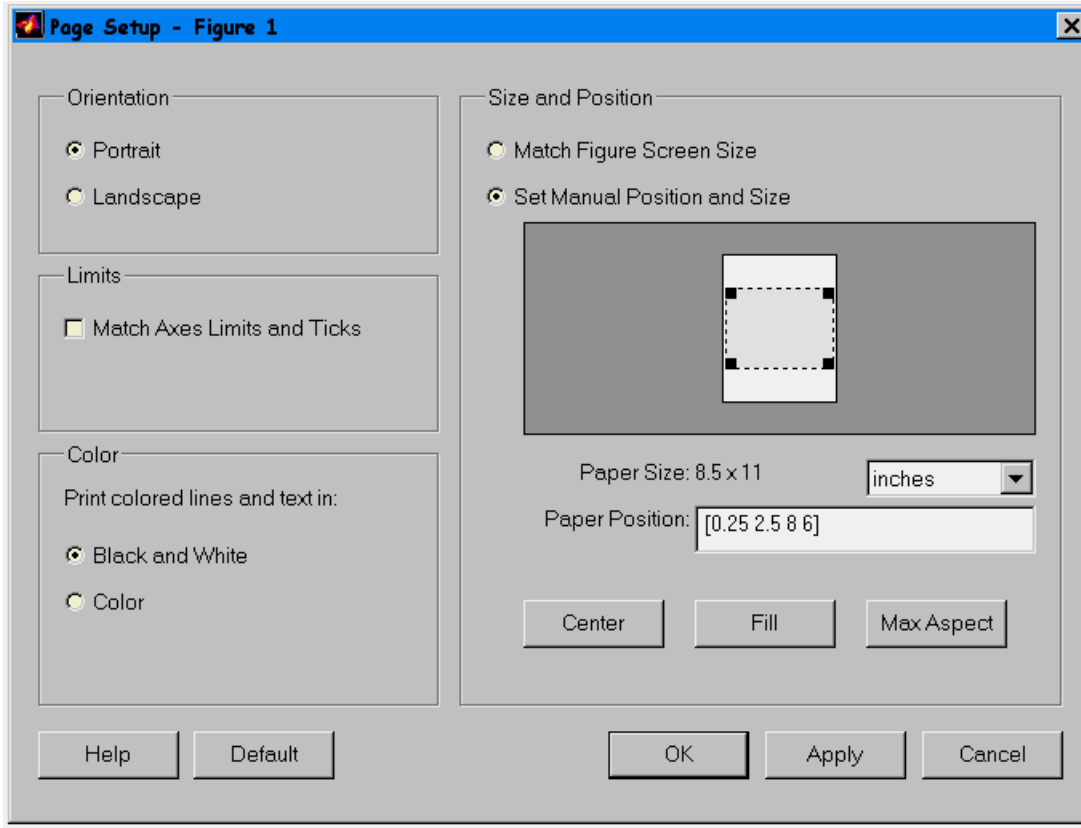
Adjusting the Size and Color of the Graphic

To adjust the size and location of the printed graphic, choose the **Page Setup...** option from the figure window's **File** menu.

The Page Setup Dialog Box

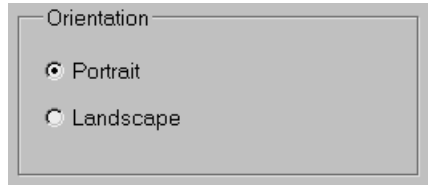
The **Page Setup** dialog box enables you to perform four categories of adjustments:

- Orientation
- Limits
- Color
- Size and Position



When you have finished making selections on the dialog box, select **OK** to accept the settings for printing and close the dialog box, **Apply** to apply the changes without closing the dialog box, **Default** to return the values to the current MATLAB defaults, or **Cancel** to dismiss the dialog box without accepting any of the changes made.

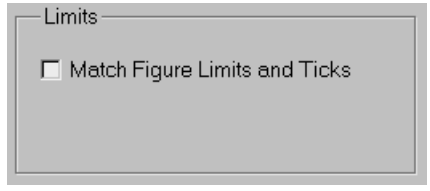
Orientation



Orientation refers to the way the figure is placed on the printed page. **Portrait** (the default) specifies that the longer paper dimension is oriented vertically; **Landscape** orients the longer paper dimension horizontally. In both cases, the value of the paper position (size and position of the figure on the page) remains the same.

Use the **Center**, **Fill**, and **Max Aspect** buttons in the **Size and Position** field to adjust the paper position.

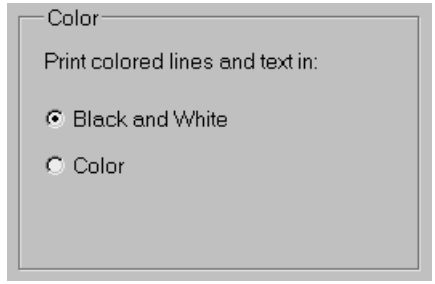
Limits



MATLAB changes the number and placement of axis tick marks when you print a figure, if the size of the printed figure is different from its size on screen. The default size of the printed figure is six by eight inches. This is generally larger than the size on screen so MATLAB takes advantage of the increased size and increases the resolution of the axes.

Check this box if you do not want MATLAB to recompute the tick marks on the printed figure. The printed figure will then have the same tick marks as on screen.

Color



The Color field controls whether lines and text are printed in color or changed to black and white.

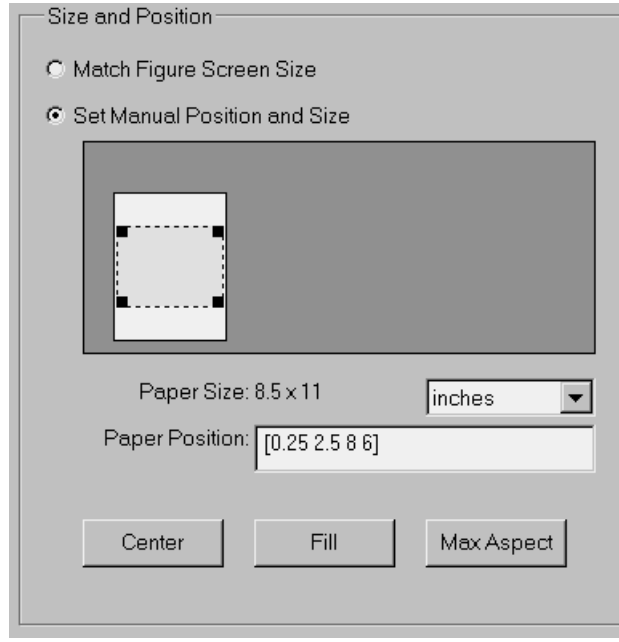
- If you select **Black and White**, MATLAB changes the color of lines and text to either black or white, depending on the background color.
- If you select **Color**, MATLAB prints lines and text in their actual color (which will be dithered on a black and white printer).

If you print from the MATLAB command line using the `print` command, options that you specify override options you set in the **Page Setup** dialog box. For example, if you select **Color** in this field in the dialog box and then print the figure with one of these command,

```
print -dwin % Windows platform
print -dps  % Unix platform
```

the resulting page is printed in black and white.

Size and Position



The **Size and Position** field provides control of both the size and the position of the figure. You can choose from two basic approaches to sizing and locating the figure on the printed page:

- **Match Figure Screen Size** – With this option, you specify the figure size (and aspect ratio) on the screen. You can do this by resizing the figure window or by setting the figure `Position` property. You must resize the figure before opening the Page Setup dialog box.
- **Set Manual Position and Size** – This option enables you to position and resize the figure on the printed page without changing the figure on screen.

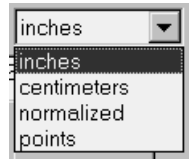
Manipulating a Figure Manually. You can change the position and size of the figure in either of two ways:

- Resize and move the dashed box, which represents the size and position of the figure on the printed page.
- Specify a new value for the paper position, which is defined by four components:

  `[distance_from_left, distance_from_bottom, width, height]`

  MATLAB interprets these values in the selected units.

Setting Units. Displaying the pull-down menu provides four options for units.



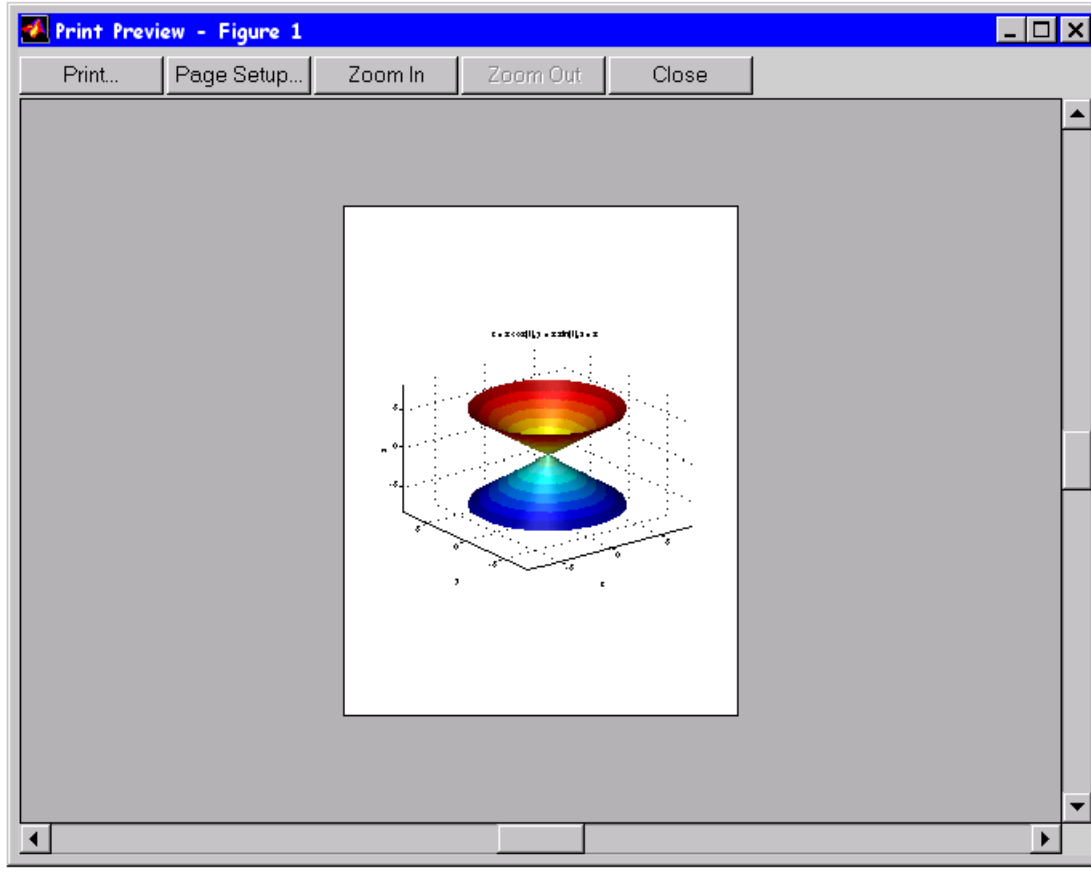
MATLAB uses these units to interpret the values specified for the paper position. All units are measured from the lower-left corner of the page. Normalized units map the lower-left corner of the page to (0,0) and the upper-right corner to (1.0,1.0).

Automatic Placement. There are three automatic positioning push buttons:

- **Center** – place the figure as currently sized in the center of the page.
- **Fill** – resize the figure to be the size and shape of the paper (note that this option may change the figure aspect ratio and thereby alter the way the plot looks).
- **Max Aspect** – make the figure as large as will fit on the page without changing the aspect ratio.

Print Preview

You can preview the way your figure will look on the printed page using the print previewer. Start the print previewer by selecting **Print Preview...** under the figure window's **File** menu.



The preview shows the size and placement of the figure on the printed page and also accurately represents color.

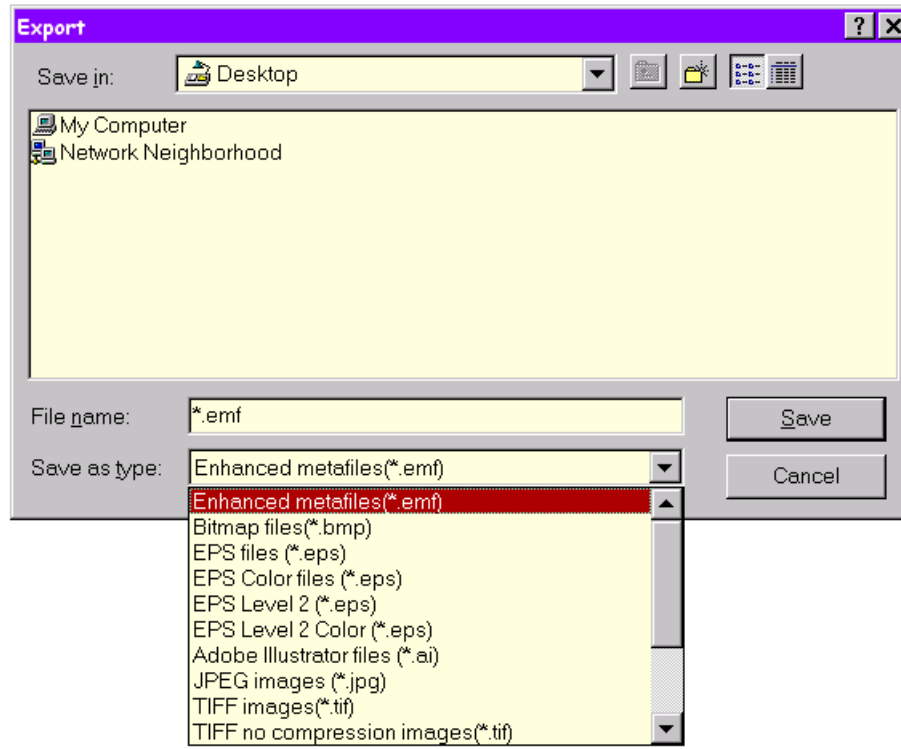
The **Zoom Out** and **Zoom In** buttons provide a full-page view and a scrolled-window close up of the graphic.

You can access the **Page Setup** dialog box directly from the previewer or you can send the graphic directly to the printer using the current page settings.

Note: When using paper types `arch-B` or larger, the print preview may be displayed in a lower resolution to conserve memory. However, this does not affect the quality of the printed page.

Exporting Figures to Graphic Files

You can export a MATLAB figure to a standard graphic file format by selecting **Export...** under the figure window's **File** menu. The **Export** dialog box enables you to save figures in a variety of standard formats.



MATLAB supports additional graphic formats via the `print` command.

Printing from the Command Line

The `print` command enables you to send the figure directly to a printer or export a figure to a variety of standard graphic file formats. In addition, you can specify options to modify the default behavior.

This section discusses:

- The `print` command
- Options for modifying the behavior of `print`

When you print from the command line, the output is controlled by `print` command options, settings specified in the **Page Setup** dialog box, and the values of figure properties. Print command arguments override **Page Setup** dialog settings, which in turn, override the default print command specified in `printopt`.

The **Page Setup** dialog actually sets figure properties, therefore property values specified on the command line after you close the dialog box take precedent over dialog box settings.

When to Use the Print Command

On MS-Windows platforms, where you can use the clipboard to copy and paste, the print command is useful if you need to:

- Export to a file type not listed in the **Export** dialog box
- Apply options not otherwise available (e.g., TIFF preview, loose bounding box, JPEG quality factor, resolution, etc.)
- Print in batch from within a M-file

The print Command

To print from the MATLAB command line, use the `print` command and specify the appropriate device type. The syntax of the `print` command is:

```
print –devicetype –options
```

If you do not specify a device type, MATLAB uses the default device for your system. For example, these commands plot a sine function and print the resulting figure on your default printer.

```
x = –pi:0.1:pi;
plot(x,sin(x))
print
```

To send the output to a file rather than to a printer, the syntax is:

```
print –devicetype –options filename
```

For example, the following command creates an Encapsulated PostScript file from the current figure, includes a TIFF preview, and names the file `fig1.eps`.

```
print –deps –tiff fig1.eps
```

Passing String Arguments to print

You can use the function form of `print` to pass string arguments to the command. This form encloses all arguments in parenthesis and places quotes around all arguments that are not variable names.

Batch Printing

You can use a for loop to create different graphs and print them in a series of files whose names are constructed from the loop counter (or any other variable for that matter):

```
for i=1:number_of_graphs
   % create or update graph
   print('–dtiff','–r200',['frame',sprintf('%02d',i)])
end
```

Specifying the Figure to Print

You can print a figure that is not the current figure by specifying the figure's handle:

```
filename = 'Figure2';
print('–f2','–dtiff',filename)
```

If the figure uses noninteger handles, you can pass a numeric variable as the first argument:

```
h = figure('IntegerHandle','off')
print(h,'-dtiff',filename)
```

The command,

```
print('-svdp','-deps',filename)
```

prints the Simulink model having the name `vdp`.

Changing Default Print Settings

The `print` command obtains default settings by calling the `printopt` function. You or your system manager can change these default values by editing the file `printopt.m`, which is found in the `toolbox/local` directory. If you are working on a multiuser system, you can make a copy of `printopt.m` and place it on your search path ahead of the MATLAB version.

The syntax for `printopt` is:

```
[pcmd,dev] = printopt
```

`pcmd` and `dev` are strings representing the operating-system command for printing and the default device type for your platform. The printing command is the actual operating-system command that MATLAB invokes after it creates the temporary file. The device type is the MATLAB command-line switch used to specify the type of device to format the output for. (If you specify a device in the `print` command, `dev` is ignored.)

This table shows the default values for `pcmd` and `dev` on each platform.

Platform	pcmd	dev
MS-Windows (with no default printer specified)	`COPY /B %s default_printer` `COPY /B %s LPT1:`	`-dwin`
UNIX (except Silicon Graphics)	`lpr -r -s`	`-dps2`
Silicon Graphics	`lp`	`-dps2`

As the table shows, the default value for `dev` on non-Windows platforms is `–dps2`, which means MATLAB produces black and white Level 2 PostScript. On Windows systems, the default value for `dev` is `–dwin`, which specifies printing through the Windows Print Manager.

Editing printopt.m

If you want to edit `printopt.m` to change the value of `pcmd` or `dev`, enter the command:

```
edit printopt
```

This command opens your text editor with the `printopt.m` file. Scroll down about 40 lines until you come to this comment line:

```
%---> Put your own changes to the defaults here (if needed)
```

On the line below this, enter the values you want to use. For example, this line sets the default device type to Level 2 color PostScript:

```
dev = '–dpsc2';
```

Graphic File Formats

MATLAB enables you to export figures to a number of standard graphic file formats via the selection of the appropriate device option for the `print` command. This section lists the devices supported by MATLAB.

Built-in Devices

When you enter a `print` command, MATLAB uses the device type returned by `printopt`. You can override the default by specifying a different device with a command-line switch.

The set of devices you can specify varies depending on your system. All systems support a core set of built-in device drivers.

MATLAB has built-in drivers for these device types:

- PostScript
- Hewlett-Packard Graphics Language (HPGL)
- Adobe Illustrator 88

This table summarizes the command-line switches for MATLAB's built-in device drivers.

Device	Description
`-dps`	Level 1 black and white PostScript
`-dpsc`	Level 1 color PostScript
`-dps2`	Level 2 black and white PostScript
`-dpsc2`	Level 2 color PostScript
`-deps`	Level 1 black and white Encapsulated PostScript (EPS)
`-depsc`	Level 1 color Encapsulated PostScript (EPS)
`-deps2`	Level 2 black and white Encapsulated PostScript (EPS)
`-depsc2`	Level 2 color Encapsulated PostScript (EPS)

Device	Description
`-dhpgl`	HPGL compatible with HP 7475A plotter
`-dill`	Adobe Illustrator 88 compatible illustration file

Output Formats Created by Ghostscript

On Windows and UNIX systems, The MathWorks distributes with MATLAB a program called Ghostscript. Ghostscript is optionally used by MATLAB's `print` command to provide support for a variety of output devices that are not supported by MATLAB's built-in drivers. When you use a Ghostscript device, MATLAB generates a Level 1 PostScript file (either color or black and white, depending on the Ghostscript device), and then calls the appropriate Ghostscript driver, which converts the output to the specified format. This output is then saved to the filename you specify in the `print` command, or sent to the printer (if you do not specify a filename).

Ghostscript is copyrighted by Aladdin Enterprises and is provided under the terms of the Free Software Foundation's GNU General Public License. This license allows you to make and distribute copies of the Ghostscript files provided with MATLAB, namely the executable file `gs` and all other files found in the `ghostscript` directory, provided that you comply with the terms of the GNU General Public License. You will find a copy of this license in the file `gsrights`, which is part of the MATLAB distribution. This file, and the rights described therein, do *not* apply to the whole of, or any other part of, the MATLAB, Simulink, or toolbox programs.

The MathWorks will provide you with source code for Ghostscript if you so request. Ghostscript (including source code) is also available directly from the Free Software Foundation and from many sources on the Internet.

This table summarizes the Ghostscript device drivers provided with MATLAB.

Device	Description
`-dlaserjet`	HP LaserJet
`-dljetplus`	HP LaserJet+
`-dljet2p`	HP LaserJet IIP

Device	Description
`-dljet3`	HP LaserJet III
`-ddeskjet`	HP DeskJet and DeskJet Plus
`-ddjet500`	HP Deskjet 500
`-dcdjmono`	HP DeskJet 500C printing black only
`-dcdjcolor`	HP DeskJet 500C with 24 bit/pixel color and high-quality color (Floyd-Steinberg) dithering (UNIX only)
`-dcdj500`	HP DeskJet 500C (UNIX only)
`-dcdj550`	HP Deskjet 550C (UNIX only)
`-dpaintjet`	HP PaintJet color printer
`-dpjxl`	HP PaintJet XL color printer
`-dpjetxl`	HP PaintJet XL color printer
`-dpjxl300`	HP PaintJet XL300 color printer (UNIX only)
`-ddnj650c`	HP DesignJet 650C (UNIX only)
`-dbj10e`	Canon BubbleJet BJ10e
`-dbj200`	Canon BubbleJet BJ200
`-dbjc600`	Canon Color BubbleJet BJC-600 and BJC-4000
`-dln03`	DEC LN03 printer
`-depson`	Epson-compatible dot matrix printers (9- or 24-pin)
`-depsonc`	Epson LQ-2550 and Fujitsu 3400/2400/1200
`-deps9high`	Epson-compatible 9-pin, interleaved lines (triple resolution)
`-dbmp256`	8-bit (256-color) BMP file format
`-dbmp16m`	24-bit BMP file format

Device	Description
`-dpcxmono`	Monochrome PCX file format
`-dpcx16`	Older color PCX file format (EGA/VGA, 16-color)
`-dpcx256`	Newer color PCX file format (256-color)
`-dpcx24b`	24-bit color PCX file format, three 8-bit planes
`-dpbm`	Portable Bitmap (plain format)
`-dpbmraw`	Portable Bitmap (raw format)
`-dpgm`	Portable Graymap (plain format)
`-dpgmraw`	Portable Graymap (raw format)
`-dppm`	Portable Pixmap (plain format)
`-dppmraw`	Portable Pixmap (raw format)

Specifying Command Line Options

The `print` command accepts a number of different options that control various aspects of the output. Some of these options are valid only with certain drivers or on certain platforms.

This table summarizes the available printing options. These options are discussed in the sections that follow.

Option	Description
`-tiff`	Add TIFF preview to Encapsulated PostScript
`-loose`	Use loose bounding box for Encapsulated PostScript
`-cmyk`	Use CMYK colors in PostScript instead of RGB
`-append`	Append to existing PostScript file without overwriting
`-r`*`number`*	Specify resolution in dots per inch
`-adobecset`	Use PostScript default character set encoding
`-f`*`handle`*	Specify handle of figure to print
`-s`*`windowtitle`*	Specify name of Simulink system window to print
`-P`*`printer`*	Specify printer to use (UNIX only)
`-painters`	Render using painter's algorithm
`-zbuffer`	Render using Z-buffer
`-noui`	Suppress printing of user interface controls

Tiff Preview for EPS (-tiff)

MATLAB does not automatically include a preview image with EPS files. When you import an EPS file without a preview image into a file from another application, the image prints properly but may appear on screen as a gray box.

If you want to include a preview image with the EPS file, use the `-tiff` option. When you use this option, MATLAB creates a preview image in TIFF format.

For example, this command creates an EPS file named `figure1.eps` that includes a preview image:

```
print –deps –tiff figure1
```

Note that this image is visible only within an application that recognizes TIFF previews.

Specifying the Bounding Box (-loose)

The size and placement of the PostScript image on the page of the printed document may not exactly match its appearance on screen, because the figure includes some white space around the figure but the PostScript itself does not. If you need the screen placement to match the printed document, use the `–loose` option. This option instructs MATLAB to create the PostScript file with a loose bounding box (that is, including white space around the figure).

CMYK Color Separations (–cmyk)

By default, MATLAB produces color output based on red, green, blue (RGB) color values. If you plan to publish MATLAB figures using four-color separations, you may want to use cyan, magenta, yellow, black (CMYK) color values rather than RGB.

The `–cmyk` option automatically converts RGB values to CMYK values. This option applies only to the PostScript and Encapsulated PostScript drivers. When you print the figure, the PostScript interpreter that renders the file must include the CMYK Extension Set. This set is available on all color Level 1 PostScript printers, most newer black and white Level 1 PostScript printers, and all Level 2 PostScript printers.

Appending to an Existing File (–append)

To include more than one figure in a single output file, print the first figure to a file, and then, for subsequent figures, use the `–append` option and specify the same file. For example, these commands create a file named `figs.ps`, which contains two different figures:

```
print –dps –f1 figs
print –dps –f2 –append figs
```

When you print the resulting file, each figure will appear on a separate page.

The –append option is not valid for Encapsulated PostScript files.

Specifying Resolution (–r)

When you print a figure in a PS, EPS, TIFF, JPEG, or PNG format, you can specify the resolution of the output. This is most useful when rendering with z-buffer. The default resolution is 150 dpi. To specify a different resolution, use the –r option. The syntax for this option is:

```
print –rnumber
```

For example, this command prints the current figure at 300 dpi:

```
print –r300
```

If *number* is 0, MATLAB prints the figure at screen resolution. (On most systems, screen resolution is between 72 and 100 dpi.)

For more information about resolution and its relationship to the rendering method used, see "Selecting the Rendering Method for Printing".

Default Character-Set Encoding (–adobecset)

Some early PostScript Level 1 printers do not support the PostScript operator `ISOLatin1Encoding` that is used in MATLAB PostScript files. If your printer does not support this operator, you may notice problems in the text of MATLAB printouts. If this happens, use the –adobecset option to specify default character-set encoding. This encoding is supported by all PostScript printers.

Specifying the Figure or Model to Print (–f, –s)

By default, MATLAB takes the current figure (i.e., the value returned by `gcf`) as the window to print. To print a figure other than the current figure, use the –f option. Note that you must use this option if the figure's handle is hidden (i.e., its `HandleVisibility` property is set to `off`).

The syntax is:

```
print –fhandle
```

For example, this command prints the figure whose handle is 2, regardless of which figure is the current figure:

```
print –f2
```

The handle of a figure corresponds to the title of the window, so in the example above, MATLAB prints the figure in the window titled “Figure No. 2.”

You can also pass the handle as a variable to the function form of `print`. For example:

```
h = figure; plot(1:4,5:8)
print(h)
```

To print the block diagram displayed in a Simulink window, use the –s option. The syntax is:

```
print –swindowtitle
```

For example, this command prints the Simulink window titled “f14”:

```
print –sf14
```

If the window title includes any spaces, you can call the function form rather than the command form of `print`. For example, this command prints a Simulink window title “Thruster Control” to a file named `thrstcon.ps`, using the MATLAB Level 1 black and white PostScript driver:

```
print('–sThruster Control','–dps','thrstcon.ps')
```

You can omit the window title if you want to print the current system. Just use:

```
print –s
```

For information about issues specific to printing Simulink windows, see the Simulink documentation.

Specifying the Printer to Use (–P) UNIX only

In general, MATLAB sends the output from a `print` command to the default printer on your system. If you want to send the output to a different printer, you can use the –P option. The syntax is:

```
print –Pprintername
```

For example, this command sends the output to a printer named “homer”:

```
print –Phomer
```

This option does not work on MS-Windows systems, which print to whatever printer you have set as your default printer.

Selecting an Output Format

This section provides information to help you select an output format. This section discusses using:

- PostScript
- HPGL
- Adobe Illustrator
- PC-specific options.

PostScript

MATLAB has several built-in drivers for generating PostScript output. When you select a PostScript driver, you can choose among these options:

- PostScript Level 1 or Level 2
- Black and white or color
- PostScript or Encapsulated PostScript

For example, if you want to create a Level 2 color Encapsulated Postscript file, use the `–depsc2` switch.

Level 1 or Level 2

Level 2 PostScript files generally are smaller and render more quickly than Level 1 files, so if your printer supports Level 2 PostScript, you should use one of the Level 2 drivers. If your printer does not support Level 2, or if you're not sure, use a Level 1 driver. Level 1 PostScript will produce good results on a Level 2 printer, but Level 2 PostScript will not print properly on a Level 1 printer.

Black and White or Color

If you are using a color printer, you should select a color driver. If you are using a black and white printer, you can use either a color driver or a black and white driver; however, a black and white driver will produce smaller output files and will render lines and text better. (Note that black and white drivers produce grayscale output. You do not need to use a color driver to produce different shades of gray.)

PostScript or Encapsulated PostScript

The type of PostScript device you select depends on whether you want to print the file directly or import it into another application (such as a word processing program). If you want to send the output directly to a printer, or save it to a file and then send that file to the printer, use a regular PostScript driver. If you want to import the output into another application, use an Encapsulated PostScript (EPS) driver.

If you select a regular PostScript driver, you can provide a filename (in which case MATLAB creates an output file but does not send it to the printer) or you can omit the filename (in which case MATLAB sends the output to the printer and deletes the temporary file it creates).

If you select an EPS driver, MATLAB always creates a file; MATLAB does not print EPS directly. If you do not specify a filename, MATLAB creates a file named after the figure window used to create the file. For example, if the current figure window is titled "Figure 2," and you enter this command:

```
print –deps
```

MATLAB displays this message:

```
Encapsulated PostScript files cannot be sent to the printer.
File saved to disk under name 'figure2.eps'
```

HPGL Compatible Plotters (–dhpgl)

MATLAB provides HPGL support for the HP 7475A plotter and other plotters that are fully compatible with the HP 7475A. To specify the HPGL format, use the `–dhpgl` option.

If you specify this option and do not provide a filename, MATLAB sends the output directly to the plotter. If you provide a filename, the `print` command creates a file called *`filename`*`.hgl` for later output to a plotter. HPGL files can also be imported into documents of other applications, such as Microsoft Word.

When plotting a figure, it is especially important that the background color be white, because this driver does not do background fills. If the background color is black, make sure the value of the `InvertHardCopy` property is `on`. When this property is `on`, MATLAB inverts the colors of the figure for printing, so that black backgrounds print as white.

Color Selection

The HP 7475A plotter supports six pens, none of which can be white. If MATLAB tries to draw in white while rendering in HPGL mode, the driver ignores all drawing commands until a different color is chosen.

Pen 1 is assumed to be black, and is used for drawing axes. The remaining colors are the first five colors in the `ColorOrder` property of the current axes object. If `ColorOrder` specifies fewer than five colors, the unspecified pens are not used.

For 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

The HPGL driver has these limitations:

- Display colors and plotted colors sometimes differ.
- Areas (faces on mesh and surface plots, patches, blocks, and arrowheads) are not filled.
- There is no hidden line or surface removal.
- Text is printed in the plotter's default font.
- Line width is determined by pen width.
- Images and uicontrols cannot be plotted.
- Interpolated edge lines between two vertices are drawn with the pen whose color best matches the average color of the two vertices.
- Figures cannot be rendered using Z-buffer; this driver always uses painter's algorithm.

Adobe Illustrator 88 (–dill)

MATLAB provides the capability to generate illustrations that can be viewed and modified by Adobe Illustrator 88 or any other application that supports a compatible file format. Regardless of where an illustration was initially created, the MATLAB output file can be further processed with Illustrator running on any platform.

By default these illustrations are always in color and appear in portrait orientation. The Illustrator `group` command is used to give the illustrations a hierarchy similar to that of the Handle Graphics or Simulink graphic represented.

Creating Adobe Illustrator 88 files

The syntax of the command is:

```
print –dill filename
```

If you do not provide a filename, MATLAB gives the file a default name based on the figure window used to create the file.

To view the output, open the saved file within Illustrator. It will have no template.

Limitations

The Illustrator driver has these limitations:

- Interpolated patches and surfaces cannot be created. The color of each polygon will be determined by the average of the `CData` values for all of the polygon's vertices.
- Images cannot be rendered.
- No fonts are downloaded to the Illustrator file. Any fonts used must be available to Illustrator when the file is viewed.
- The file must be opened in Illustrator before it can be printed.

PC-Specific Output Options

On the PC, MATLAB uses two different printing mechanisms, depending on whether you print through Windows print drivers or with MATLAB's own built-in print drivers. By default, MATLAB uses Windows print drivers. To print using one of MATLAB's built-in drivers, you must either edit `printopt` to change the default device or else use the `print` command with the appropriate command-line switches for MATLAB's built-in drivers or for printing through Windows drivers (which are described in the following section).

Choosing Between Windows Drivers and MATLAB Drivers

When you use Windows drivers, printing is managed through the Windows Print Manager, which enables you to monitor printer queues and control various aspects of the printing process. When you print through MATLAB's drivers, MATLAB generates the output and copies it to a port, bypassing the Print Manager.

By default, MATLAB prints using Windows drivers. However, you may find the MATLAB drivers preferable in certain situations:

- If you are creating a file to import into a document, MATLAB has several Encapsulated PostScript drivers that create high-quality graphics for importing into word processing and page layout files.
- If you need to print to a printer for which you do not have the right Windows driver, you may be able to use one of the MATLAB drivers as a substitute.
- If you are having problems with a Windows driver, you can use a MATLAB driver instead.

This table summarizes the command-line device options specific to Windows

Device	Description
`-dwin`	Change lines and text to black and white and then use Windows printing services
`-dwinc`	Maintain line and text color as it appear on screen and then use Windows printing services
`-dmeta`	Windows Enhanced Metafile format copied to clipboard or file

Device	Description
`–dbitmap`	Windows Bitmap (BMP) format copied to clipboard or file
`–dsetup`	Display the **Print Setup** dialog box, but do not print
`–v`	Verbose mode to display the **Print** dialog box (suppressed by default)

Printing Lines and Text in Color or Black and White

If you use the `print` command and do not specify a device option and your device setting defined in `printopt` is either `–dwin` or `–dwinc`, MATLAB determines whether to print lines and text in black and white or color based on:

- The settings of the **Page Setup** dialog
- The printer's color property setting, which is accessed via the **Print Setup** dialog **Property** button

MATLAB uses the setting of the printer's color property unless you have set a value in the **Page Setup** dialog, which overrides the setting of the printer's color property. Note that device options specified with the print command override both the **Page Setup** and color property settings. The order of precedence is then:

- Options specified with the print command take highest precedence, then
- Options specified with the **Page Setup** dialog, then
- Options specified with the printer's color property

Output Target

Note that when you print using the `–dwin` or `–dwinc` device type, the output is always directed to a printer. Therefore, you should not specify a filename. If you do specify a filename, MATLAB will create the file using one of its built-in PostScript drivers rather than a Windows driver.

When you print with one of MATLAB's built-in drivers, MATLAB generates output in the appropriate format and then either saves the output to a file (if you provided a filename) or else sends the output to the printer (if you did not give a filename). If the output is directed to a printer, MATLAB creates a

temporary file and then executes the MS-DOS command stored in the `pcmd` string returned by `printopt`. The file is then deleted.

Ghostscript Drivers

Using Ghostscript drivers is similar to using the built-in MATLAB drivers. When you specify a Ghostscript driver, MATLAB generates PostScript, which Ghostscript then converts to the selected format. MATLAB then executes the command stored in `pcmd` to print the file, or else saves the output to the specified filename.

Troubleshooting

Occasionally, you may run into problems when printing with a Windows driver, because of a bug in the driver or an incompatibility between the driver and MATLAB. If you do have a problem printing with a Windows driver, try one of these options:

- Use a different Windows driver. There may be a newer version of the driver available from the manufacturer, or there may be a driver available from a different vendor. You may also be able to use a driver for a different printer, such as an earlier model from the same manufacturer.
- Use a built-in MATLAB driver or a Ghostscript driver. For example, if you are having trouble printing to an HP LaserJet printer using a Windows driver, you can use one of the Ghostscript LaserJet drivers instead. If your printer supports PostScript, use one of MATLAB's built-in PostScript drivers.

Specifying Fonts and Character Sets

MATLAB figures support several kinds of text objects, such as titles, axis labels, and tick labels. This section discusses how to control the font and character set for text objects in figures so that the printed output uses the fonts you want.

Font characteristics are properties of axes, uicontrols, and text objects. For each of these objects, you can set these properties:

- `FontName`
- `FontSize`
- `FontUnits`
- `FontWeight`
- `FontAngle`

For example, to specify 10-point Helvetica-BoldOblique for the current axes:

```
set(gca,'FontName','Helvetica','FontSize',10,'FontUnits', ...
'points','FontWeight','bold','FontAngle','oblique')
```

Note that when MATLAB generates hardcopy output, it does not attempt to determine what fonts are available on the hardcopy device before it sends output to the device. If you specify a font that is not available on your printer, the printer will substitute another font. A PostScript printer will substitute Courier for any unavailable font.

MATLAB can generate PostScript output using the fonts listed below. These are the actual names you should use when you specify the FontName property for a text object:

- `AvantGarde`
- `Bookman`
- `Courier`
- `Helvetica`
- `Helvetica-Narrow`
- `NewCenturySchlbk`
- `Palatino`
- `Symbol`
- `Times-Roman`

- `ZapfChancery`
- `ZapfDingbats`

If you use a font not on this list, MATLAB's PostScript driver substitutes Courier. This substitution affects the Ghostscript drivers as well, because they work by converting MATLAB's PostScript output.

Font properties for the axes object itself affect the *x*-, *y*-, and *z*-tick labels. Axis labels (`XLabel`, `YLabel`, and `ZLabel`) and `Titles` also use the axes font characteristics; however, you can set the font characteristics for these text objects explicitly to override the axes font values. For example, to change the font size of the Title, you could enter:

```
h = get(gca,'Title');
set(h,'FontSize',18);
```

The character set used for a text object is determined by its font. On most platforms, most fonts use the primary character set encoding for the platform. For PostScript output, you can also specify default PostScript character-set encoding by using the `–adobecset` option, as described on page 10-260.

PC

On the PC, the valid fonts and character sets depend on whether you print using Windows drivers or MATLAB's built-in drivers.

The built-in MATLAB drivers support only fonts that are compatible with the Windows Latin-1 character set. If you use a built-in MATLAB driver, you should choose fonts that match the standard set of supported PostScript fonts (such as Times or Helvetica). TrueType fonts are acceptable as long as they meet this requirement. For example, the TrueType Symbol font works. MATLAB also accepts the TrueType fonts Arial, New Times Roman, and New Courier, and maps them to their PostScript equivalents (Helvetica, Times Roman, and Courier, respectively).

The native Windows drivers support Windows Latin-1 as well as a wide variety of other Windows character sets. If you print using the Windows drivers, Windows requires that you use TrueType fonts for text to be printed correctly. (You can tell if a font is TrueType by looking at the Fonts Control Panel. The icon for a TrueType font has "TT" on it, and the filename extension is "TTF".)

UNIX

On UNIX systems, MATLAB supports the ISO Latin-1 primary character set. For example, suppose a text object is created with these commands:

```
h = text(0.1,0.1,'some text');
set(h,'FontName','Times');
```

There might be several X Window System fonts with the name “Times.” MATLAB tries to find a font that supports the primary ISO Latin-1 character set. For backward compatibility, this preference is ranked below any other specifications you provide, such as the font size, style, and so forth. ISO Latin-1 fonts have an X font specification that ends in `iso8859-1`, which is the formal name of the ISO Latin-1 character set. Here is an example of such a font specification:

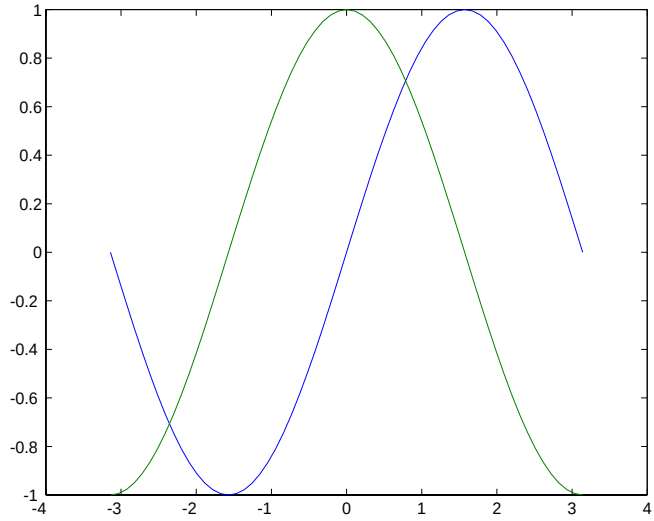
```
-misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso8859-1
```

You can use `xlsfonts` at the UNIX prompt to list the set of fonts available on your system.

Specifying Line Styles

When displaying on a color screen or printing to a color printer, MATLAB usually distinguishes different lines in a figure by their colors. For example, these commands plot the sine and cosine functions; MATLAB sets the colors of the lines according to the value of the `ColorOrder` property:

```
x = –pi:pi/30:pi;
plot(x,sin(x),x,cos(x))
```



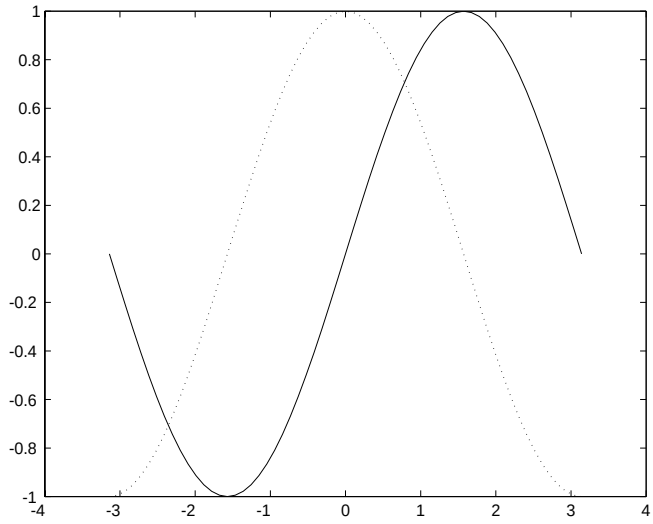
In addition to color, you can distinguish lines by line style or marker symbol. If you want to print the figure on a black and white printer, keep in mind that all lines will print as black (or white, if the background is black and `InvertHardCopy` is `off`). If you want MATLAB to dither the lines to attempt to render them as different shades of gray, you can use a color driver; however, lines are generally too thin to be dithered effectively. A better approach is to vary the line style or marker symbol. Many of the plotting functions provide a mechanism for setting the line style and marker symbol for each line being plotted.

You can also control the line styles by setting the axes `LineStyleOrder` and `ColorOrder` properties. To distinguish lines, MATLAB cycles first through the `ColorOrder` values and then the `LineStyleOrder` values. The factory default

for `ColorOrder` is a set of six colors, while the factory default for `LineStyleOrder` is a single style (a solid line). This means that MATLAB will use different colors but the same line style for all lines, unless you specify otherwise.

If you print to a black and white printer, you may want to change `ColorOrder` to a single color, and `LineStyleOrder` to multiple styles. This will cause MATLAB to use the same color for each line, but different styles. These values must be set before the axes object is created. For example, this code creates a new figure, sets the appropriate axes properties, and then creates the plot:

```
x = –pi:pi/30:pi;
figure('DefaultAxesColorOrder',[0 0 0], ...
'DefaultAxesLineStyleOrder','-|:|--|-.')
plot(x,sin(x),x,cos(x))
```



Windows 95 Limitation

Microsoft Windows 95 does not support broken line styles for lines whose width is greater than 1 pixel. Unfortunately, most printers produce lines more than 1 pixel thick, so in most cases, Windows 95 drivers produce solid lines, regardless of the setting of `LineStyleOrder`.

There are various ways you can work around this problem:

- Set up MATLAB to use lines 1 pixel wide, by adding this line to the `[MATLAB Settings]` section of your `MATLAB.INI` file:

  ```
  ThinLineStyles=1
  ```

  This will result in very thin lines, but the lines will print with the specified styles.
- Set the figure's `Renderer` property to `zbuffer`:

  ```
  set(gcf,'Renderer','zbuffer')
  ```

  This will result in the printed output matching the screen display. See "Selecting the Rendering Method" for more information about z-buffer.
- Use a PostScript or Ghostscript device. These drivers bypass the Windows Print Manager.

Selecting the Rendering Method for Printing

MATLAB uses two different methods to render figures for printing:

- Painters
- Z-buffer

Painters draws figures using vector graphics, while z-buffer uses raster (bitmap) graphics. Note that MATLAB uses z-buffer to print figures rendered on the screen with OpenGL.

In general, painters produces higher-resolution results than z-buffer. However, z-buffer works in situations where painter's algorithm either produces inaccurate results or does not work at all.

Automatic Selection of Rendering Method

By default, MATLAB automatically selects the best method, based on the complexity of the figure and the settings of various Handle Graphics properties. In general MATLAB uses:

- Painter's for line plots, area plots (bar graphs, histograms, etc.), and simple surface plots.
- Z-buffer for complex surface plots using interpolated shading and any figure using lighting.

Specifying the Rendering Method

You can specify the rendering method by setting the figure `Renderer` property. When the `RendererMode` property is set to `auto` (the factory default), MATLAB selects the rendering method for displaying and for printing. When `RendererMode` is set to `manual`, MATLAB uses the method specified by the `Renderer` property for both displaying and printing. (Setting the `Renderer` property automatically sets the `RendererMode` property to `manual`.)

The rendering method used for printing the figure is not always the same method used to display the figure.

In some cases, you may want to override MATLAB's renderer selection when you print, without changing the `Renderer` property. You can specify the rendering method to use for printing by using the `–zbuffer` or `–painters` option with the `print` command.

For example, these commands create a figure, display it using painter's algorithm, and print it using Z-buffer:

```
surf(peaks(32));
set(gcf,'Renderer','painters')
print –zbuffer
```

Limitations of Each Method

For many figures, it is possible to use either rendering method. There are certain situations, however, where painter's algorithm does not work or produces unacceptable results. For example:

- If the figure uses truecolor for patch or surface objects, it cannot be rendered with painter's algorithm. If you set `Renderer` to `painters`, MATLAB issues a warning and the graphics objects do not display or print.
- If the figure includes any lights, the lighting cannot be rendered with painter's algorithm. If you set `Renderer` to `painters`, the lighting disappears.

Note that in each case the figure retains all the appropriate data, so if you set `Renderer` back to `zbuffer` or set `RendererMode` to `auto`, the missing objects reappear.

In general, if you find that your printed output does not match what you see on screen, you should set `Renderer` to `zbuffer`, or use the `–zbuffer` switch when you print.

However, you cannot use z-buffer rendering if your device type is HPGL or Adobe Illustrator. If you attempt to print to one of these formats and `Renderer` is set to `zbuffer` (or if you use the `–zbuffer` option), MATLAB uses painter's algorithm instead.

Size of Output Files

When you print a figure rendered with painter's algorithm, the resolution has little effect on the size of the output file or the amount of memory needed for printing. Therefore, the default resolution is quite high (864 dpi for MATLAB's built-in PostScript drivers).

When you print a figure rendered using z-buffer, certain factors directly influence the size of the output file or amount of memory needed for printing:

- Resolution of the output
- Size of the printed graphic
- Use of a color or black and white (grayscale) driver

These relationships exist because Z-buffer figures are rendered as bitmaps, and the number of pixels in a bitmap is a function of the resolution and of the size of the graphic. For example, if a figure is 2 inches by 3 inches, it will consist of 60,000 pixels at 100 dpi. Increasing either the resolution or the size increases the number of pixels proportionately. For example:

- If you keep the figure the same size but increase resolution to 200 dpi, the number of pixels is 240,000.
- If you keep the resolution at 100 dpi but enlarge the figure to 4 inches by 6 inches, the number of pixels is also 240,000.

Note that the size of the actual output is what matters, not the size on the screen (although, if `PaperPositionMode` is set to `auto`, the size on screen and on paper are the same).

For figures rendered using Z-buffer, the default resolution is 150 dpi. To set the resolution to a different value, use the `-r` option. The syntax is:

```
print -rnumber
```

`number` is the number of dots per inch. For example, to specify a resolution of 100 dpi:

```
print -r100
```

To specify printing at screen resolution, set *`number`* to 0 (zero):

```
print -r0
```

In addition to the figure size and resolution, the choice of color or black and white also affects the size of the file, because the amount of information stored for each pixel is larger for color than for black and white. Color files are three times as large as black and white files, so be sure to use a black and white driver unless you want to print to a color device.

Because of these issues, you must make trade-offs between resolution, size, color, and printing resources, when printing a figure rendered using Z-buffer. For example, you can specify any resolution, but you may find at higher resolutions the resulting files are too big and require too much memory to

print. However, you are likely to find that much lower resolutions produce acceptable results.

Changing Background Colors

By default, MATLAB figures display on screen as colored lines and surfaces on a white background. When you print a figure on a color device, the colors remain unchanged. If you print to a black and white device, surface colors are dithered to render them as shades of gray, except for lines and text, which are changed to black because these objects are too thin to be dithered effectively.

If you want MATLAB to dither lines, use a color driver rather than a black and white driver. For example, if you are printing on a black and white PostScript printer, you could use the `–dpsc` option. Note, however, that you may not be able to distinguish between different colored lines on the basis of the dithering.

If you prefer, you can display figures on screen as colored lines and surfaces on a black background, by typing:

```
colordef black
```

When you print a figure with a black background, MATLAB inverts the colors for printing: anything black (including the background) is changed to white, and anything white (such as lines, surfaces, or text) is changed to black. These changes are made so the printer will use less toner and produce better looking output.

If you do not want MATLAB to invert the colors when you print the figure, set the figure's `InvertHardCopy` property to `off`. For example:

```
set(gcf,'InvertHardCopy','off')
```

Note that MATLAB does not invert image or uicontrol objects when you print them, regardless of the value of `InvertHardCopy`.

Troubleshooting MS-Windows Printing

If you encounter problems such as segmentation violations, general protection faults, application errors, or the output does not appear as you expect when using MS-Windows printer drivers, try the following:

- If your printer is PostScript compatible, print with one of MATLAB's built-in PostScript drivers. There are four PostScript device options that you can use with the `print` command: `–dps`, `–dpsc`, `–dps2`, and `–dpsc2`.
- The behavior you are experiencing may occur only with certain versions of the print driver. Contact the print driver vendor for information on how to obtain and install a different driver. If you are using Windows 95, try installing the drivers that ship with the Windows 95 CD-ROM.
- Try printing with one of MATLAB's built-in GhostScript devices. These devices use GhostScript to convert PostScript files into other formats, such as HP LaserJet, PCX, Canon BubbleJet, and so on.
- Use Windows Metafiles as the copy format when using the **Edit-->CopyFigure** menu item on the figure window menu or the `print –dmeta` option at the command line. You can then import the file into another application for printing.

  You can set copy options in the figure's **File-->Preferences...-->Copying Options** dialog box. The Windows Metafile clipboard format produces a better quality image than Windows Bitmap.

See "Importing MATLAB Graphics into Other Applications" for information on saving MATLAB figures in specific graphic file formats.

Saving MATLAB Graphics in File Format

In addition to options for printing directly to hardcopy devices, MATLAB provides the ability to produce files in various graphics formats for importing into other applications.

Creating Graphics Files

There are several ways to create graphics files in MATLAB.

- Use the `print` command with an appropriate device option; for example, Encapsulated PostScript (`–deps`) or TIFF (`–dtiff`). You can also specify a resolution in dots per inch using the `–r` option.
- Use the `capture` command to create an image of the figure, and then use `imwrite` to write the file. This technique is limited to screen resolution.
- On the PC, copy the figure to the clipboard as a Windows Bitmap or Windows Enhanced Metafile.

Using the print Command

When you use the `print` command and specify a filename, MATLAB creates the file but does not send it to the printer. Select a device option that creates a format that you can import into the target application.

You can use MATLAB's built-in drivers to produce graphics files in Encapsulated PostScript, Adobe Illustrator 88, and HPGL formats. To produce a file in one of these formats, specify the appropriate device option, and provide a name for the file. For example, this command produces an HPGL file named `surfplot.hgl`:

```
print –dhpgl surfplot
```

Additional formats are available only on certain platforms. On PC and UNIX systems, you can o produce standard graphics file formats such as PostScript, TIFF, JPEG, and PNG . On the PC, you can create files in Windows Bitmap and Windows Enhanced Metafile format by using the `–dbitmap` option or the `–dmeta` option and specifying a filename. (If you omit the filename, the Metafile or Bitmap is placed in the clipboard.)

For example, this command creates a Windows Bitmap file named `surfplot.bmp`:

```
print –dbitmap surfplot
```

This command creates a Metafile of figure 2 and places in the clipboard:

```
print -f2 -dmeta
```

MS-Windows Copy Options

On the PC, you can import a MATLAB graphic into another application by copying the figure to the clipboard in Windows Bitmap or Windows Enhanced Metafile format, and then pasting the graphic into the target application.

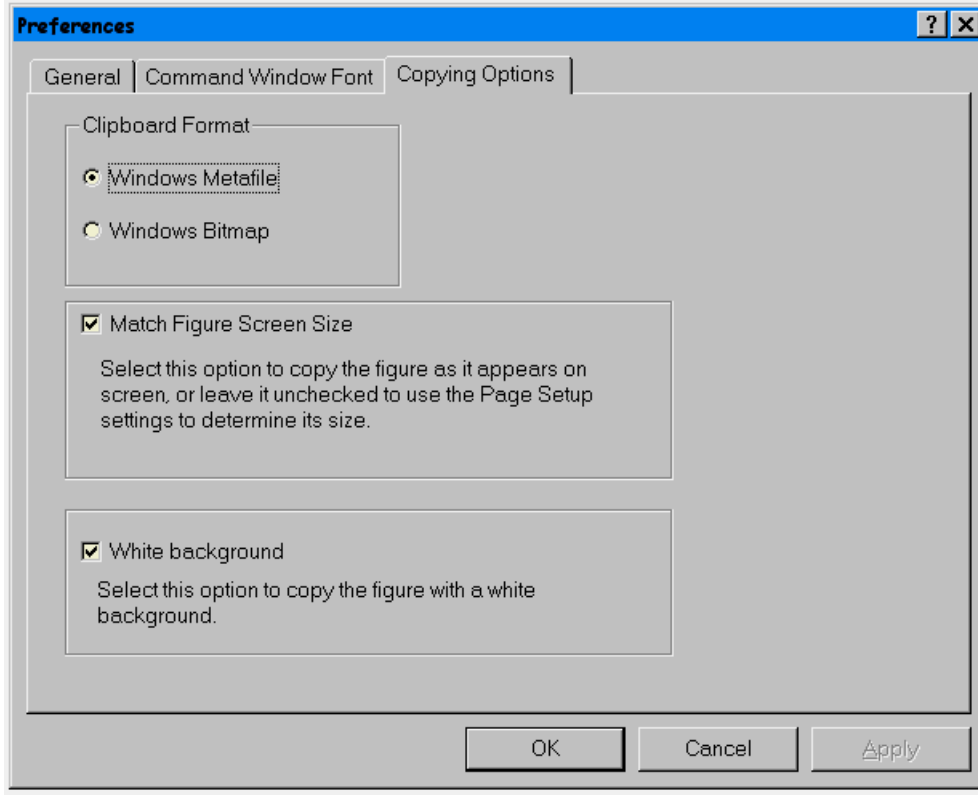
There are two ways to copy a figure to the clipboard:

- Select the **Copy Figure** command from the **Edit** menu of the figure window. The format of the output is determined by preferences you can set with the **Copy Options** dialog box.
- At the command line, use the `print` command with the `–dbitmap` or `–dmeta` option. Do not provide a filename. The figure is then copied to the clipboard as a Windows Bitmap or Metafile, depending on which option you use.

You then import the graphic into another application by using the **Paste** command.

Specifying Copy Options

You can set copy options using the **Copy Options** dialog box, which is located in the figure window's **Edit** menu.



The **Copy Options** dialog box contains three sections. The second two apply only to Windows Metafiles:

- **Clipboard Format**
- **Match Figure Screen Size**
- **White Background**

Choosing the Clipboard Format

The Windows Bitmap and Enhanced Metafile formats are fundamentally different in the way they represent the figure. The Bitmap format creates a bitmapped copy of the figure window, while the Metafile format uses a vectorized approach. In general, the bitmap format is of lower resolution than the Metafile format.

The Windows Enhanced Metafile format is a device-independent format for sharing graphics between Windows applications. This format is capable of producing high-quality graphics, and is preferred over the bitmap format on Windows systems.

Match Figure Screen Size

When the clipboard format is Metafile, you can select two ways to specify the size of the graphic that is copied to the clipboard:

- Check **Match Figure Screen Size** to match the size of the graphic copied into the clipboard to the size of the figure on the screen. Select this option if you want to see the figure in actual size on the screen.
- Leave **Match Figure Screen Size** unchecked to define the size of the graphic using the paper position specified in the **Page Setup** dialog box. If you select this option, you may also want to check the **Match Axes Limits and Ticks** option in the **Page Setup** dialog box to ensure the axis limits and tick mark placement remain the same in the copied graphic as they are on screen.

White Background

When the clipboard format is Metafile, you can choose to copy all graphics with a white background color, regardless of the color on screen. This option is useful when you are printing to a black and white printer and do not want to include the default gray figure background color or if you are using a white on black color scheme (see `colordef` for information about setting the color scheme).

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

- Create the figure you want to import into the application
- Save the MATLAB figure in a standard graphics file format that is appropriate for the type of figure and target application.
- Import the graphics file into the application.

The sections in this topic area provide background information about the characteristics of the various graphic file formats. This information helps you select the appropriate format for your application. If you would like to see examples of methods that work reliably in most applications, go directly to one of these examples:

- Importing graphs into word processor documents.
- Importing bit-mapped images into word processor documents.

Selecting the Graphics File Format

MATLAB provides many different options and formats for graphical output. The best format to use depends on your platform and which applications you want to import graphics into.

When deciding which format to use, you should consider:

- What formats does the target application support?
- What format is appropriate for the particular MATLAB figure?
- Do you need color or black and white?
- What level of quality do you need?
- Do you need to be able to edit the graphic in the target application?
- Do you need to be able to use the graphic on different platforms?

Most applications support a variety of formats, so the most important factor is whether to use a vector or bitmap format.

Vector Format

Vector formats store graphics as geometric objects defined by drawing commands. These commands are interpreted by the output device (e.g., a printer) resulting in high quality lines, surfaces, and text. You can resize vector graphics in your target application without losing quality. The vector formats that MATLAB supports are:

- Encapsulated PostScript
- Adobe Illustrator 88
- HPGL
- Windows Enhanced Metafile

The other graphics formats that MATLAB supports are all bitmap formats.

Vector formats are preferable for most 2-D plots and surface plots that are not of great complexity (the complexity is determined by the number of polygon, number of polygons with interpolated shading, number of markers, presence of truecolor images, and other factors). If MATLAB exports the figure to a file using the z-buffer renderer (such as with any figure that uses lighting), MATLAB automatically renders the figure as a bitmap. While you can still export the figure as one of the vector formats, the resulting files actually contains bitmaps and can be quite large. In this case, it is better to use a bitmap format such as TIFF.

Encapsulated PostScript

For many applications, the best format to use is Encapsulated PostScript (EPS). This format provides very high quality output because it is a vector format. In addition, EPS is portable to every platform MATLAB runs on.

The main drawback to the EPS format is that when you print a document in which the EPS graphic is embedded, you must use a PostScript printer, or the graphic will not print. Also, EPS graphic files are not visible in most applications and appear as gray boxes on screen. You can, however, include a TIFF preview in the EPS file, which is visible in most word processing and desktop publishing applications.

Bitmap Format

Bitmap formats store graphics as matrices of pixels. Bitmaps behave differently from vector-format graphics file. In general, resizing a bitmap

causes round-off errors that result in jagged edges and degradation of picture quality. This degradation is particularly obvious in text. Examples of commonly used bitmap formats include:

- TIFF
- JPEG
- BMP
- PNG

When working with bitmap formats, keep in mind:

- You should avoid scaling a bitmap picture once imported into another applications.
- The size of the file created is proportional to the size of the figure being printed (not the figure content) and the resolution at which you print it.
- If you are including a bitmap picture in a document you are going to print on paper, then you need to use a resolution higher than screen resolution.

Specifying the Size of a Bitmapped Graphic

The best approach when using bitmap formats is to render the figure to the desired size in MATLAB and avoid any scaling after the file has been created. You can do this by setting the `PaperPosition` property in the **Page Setup** dialog box.

See "Example – Importing a Bitmap Graphic" for information on how to import a bitmap into a word processor document.

See "Selecting the Rendering Method for Printing" for related information.

Additional Considerations

This section describes additional issues you should consider when selecting graphic formats.

Editing Graphics Files Outside of MATLAB

If you need to be able to edit the graphic once it is imported into the new application, there are a few options:

- If the target application is Adobe Illustrator, create the file using the `print –dill` command. The resulting file is editable in Illustrator.
- On the PC, the Enhanced Metafile format can be edited in many applications, such as Powerpoint.
- Some painting programs can edit bitmaps in certain formats. For example, the Paintbrush application that comes with Windows can edit PCX files.

Portability of Graphics File Formats

Portability is an issue if you use more than one computer platform. If you use the target application on more than one platform, you need to use a format that is not platform-specific. TIFF and EPS formats are good choices, because they are supported on all platforms that MATLAB runs on.

Using the Clipboard

In terms of convenience, copying to the clipboard and pasting into the target application is generally the simplest method. The disadvantages of this approach are that you are limited to working on a single platform, no file is created and you cannot specify `print` command options.

Creating Graphics File Formats Not Supported by MATLAB

Note that if you need output in a graphics format that MATLAB does not produce, you may be able to use a format conversion application to convert a MATLAB-produced graphic to another format. For example, MATLAB does not produce GIF files (due to patent restrictions), but there are many applications that can convert TIFF files to GIF.

Creating Graphic Files for Use in HTML Documents

MATLAB enables you to exports figures as JPEG files. You can include references to these files directly from HTML documents since WEB browsers can display this file format. You can also create PNG files, which may be supported by your WEB viewer.

Application-Specific Issues

This section discusses issues related to importing MATLAB graphics into several commonly used applications. These applications are:

- Microsoft Word
- Corel Draw
- Scientific Word
- LaTeX

Microsoft Word

When you import a graphic into a Microsoft Word document, first create a frame in the document and import the graphic into it. Importing into a frame will enable you to reposition the graphic by dragging it. For more detailed information about importing graphics into Word, see one of these examples:

- Importing graphs into word processor documents.
- Importing bit-mapped images into word processor documents.

Corel Draw

You can import Windows Enhanced Metafiles and PCX files into Corel Draw. Note that the graphic appears to be black and white until you make the picture full screen.

Scientific Word

You can import a MATLAB figure into Scientific Word by creating an Encapsulated PostScript file. Note that you cannot control the size of the graphic in Scientific Word, so be sure to make the image the size you want when you create it in MATLAB. The next section contains an example that illustrates how to use the **Page Setup** dialog box to specify the size of the graphic when using EPS files.

LaTeX

You can import a MATLAB figure into LaTeX by creating an Encapsulated PostScript file. The general syntax for including a PostScript figure in LaTeX is:

```
\begin{figure}[h]
\centerline{\psfig{figure=file.ps,height=height,angle=angle}
\caption{caption}
\end{figure}
```

(The items in italics are place holders for the actual values you specify.)

You can specify the height in any LaTeX compatible dimension. To set the height to 3.5 inches, use the command:

```
height=3.5in
```

You can use the `angle` command to rotate the graph. For instance, to rotate the graph 90 degrees, use the command:

```
angle=90
```

Including Graphics in Word Processor Documents

This sections provides examples illustrating how to include MATLAB graphics in word processor documents. The basic approach involves these steps:

- Create the graphic and make all modifications to it in MATLAB, rather than editing the graphic in the target application. This includes specifying the figure size, font size, line width, adding annotation, and so on.
- Export the figure to a standard graphic file format supported by the word processor application.
- Import the graphics file into the word processor document.

While MATLAB can generate a variety of graphic file formats, these examples focus on two formats that are both widely supported by word processors and produce high quality results: EPS and TIFF. (See "Selecting the Graphics File Format" for information on choosing a format.)

Example – Importing a Graph

Suppose you want to use the following plot in a document.

```
t = 0:pi/100:2*pi; y1 = sin(t); y2 = sin(t-0.25); y3 = sin(t-0.5);
h = plot(t,y1,'-',t,y2,'--',t,y3,':');
```

The best graphics format to select for this plot is a vector format so EPS is used.

Adjusting Line Width and Font Size

The resolution of a computer screen is considerably lower than that of a printed document. Therefore, you may find it useful to adjust certain characteristics to improve the quality of the printed document. This example adjusts the width of the lines plotted and the size of the axis tick label font (`LineWidth`, `FontSize`).

```
set(h,'LineWidth',1)
set(gca,'FontSize',8)
```

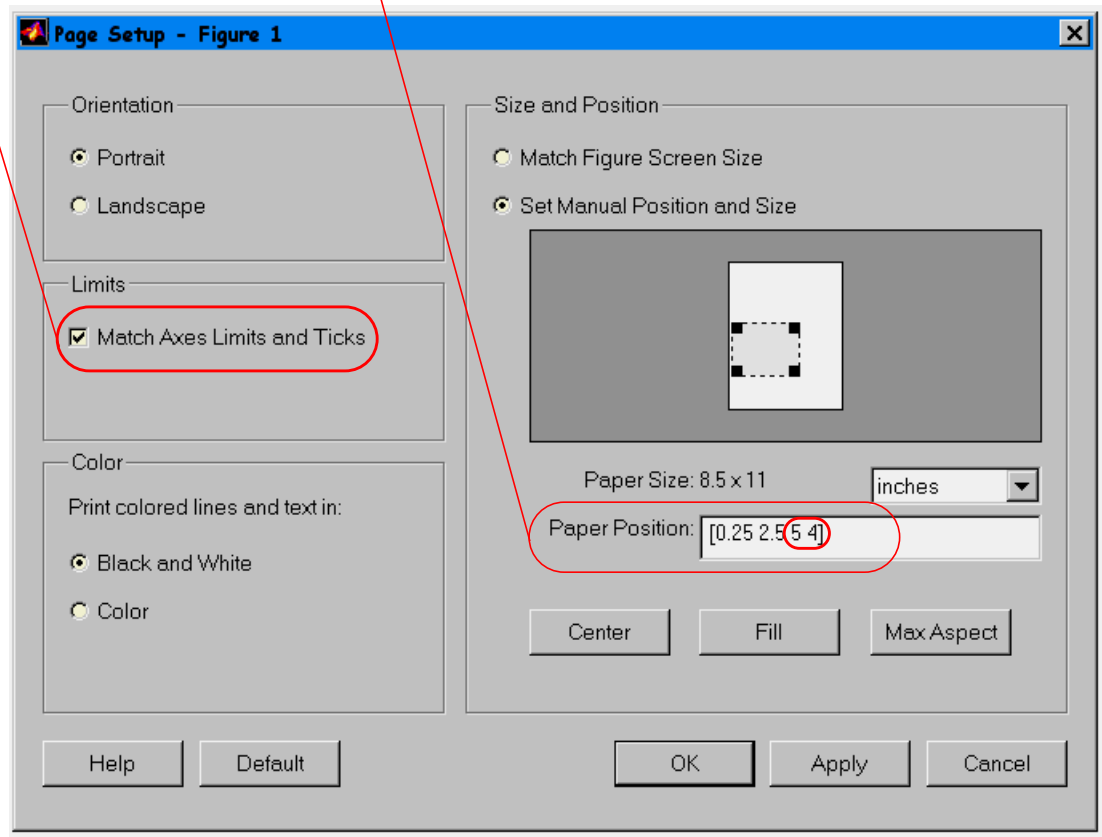
Specifying the Figure Size in Your Document

This example creates a figure that is five inches wide and four inches tall in the word processor document. To do this, set the `PaperPosition` property from the **Page Setup** dialog box. Also, to preserve the axis tick marks spacing used in

the original figure, check the **Match Axes Limits and Ticks** option in the **Limits** section of the dialog box.

Check this box to keep the axis limits the same as on the screen.

Change the width and height of the figure to 5 by 4.



Export the Figure to a File

To apply the **Page Setup** dialog box settings, click **OK** and return to the MATLAB command window, then issue the `print` command to create a color EPS file with a TIFF preview.

```
print –depsc –tiff filename
```

MATLAB adds the `.eps` extension to the file name specified: *`filename`*`.eps`.

You can now import the file into the word processor application. For example, in Microsoft Word, insert a frame and then insert the picture into the frame. Avoid using the Word picture editor once the graphic is imported.

Example – Importing a Bitmap Graphic

This example illustrates how to export a MATLAB graphic to a TIFF file that has a specified size and resolution. TIFF is a bitmapped format, therefore:

- You should not resize the picture once created.
- Bitmaps created at screen resolution may look unacceptably jagged when included in a document that is output to the higher resolution of graphics printed on paper.

The best approach is to specify the desired figure size in MATLAB and to create the TIFF file at a resolution greater than screen resolution.

Size and Resolution Considerations

The size of the bitmap file is affected by both the size of the graphic and the resolution. Therefore, printing to a smaller paper position size results in a smaller file. You may need to determine your resolution requirements through experimentation. However, the default resolution of 150 dpi is probably adequate for typical laser-printer output. If you are preparing graphics for high quality printing, such as a text book or color brochures, you may want to use 200 or 300 dpi.

Background Information

These sections provide relevant background information for this example:

- Selecting the Rendering Method for Printing
- Selecting the Graphics File Format

The Figure

This example uses the `ezsurf` function to evaluate a mathematical expression.

```
ezsurf('exp(-s)*cos(t)','exp(-s)*sin(t)','t',[0,8,0,6*pi])
```

Increasing Edge Line Width

To prevent the edges of the surface faces from appearing broken in the bitmapped image, increase the line width to 1 point from the default of .5 (`findobj`, `LineWidth`).

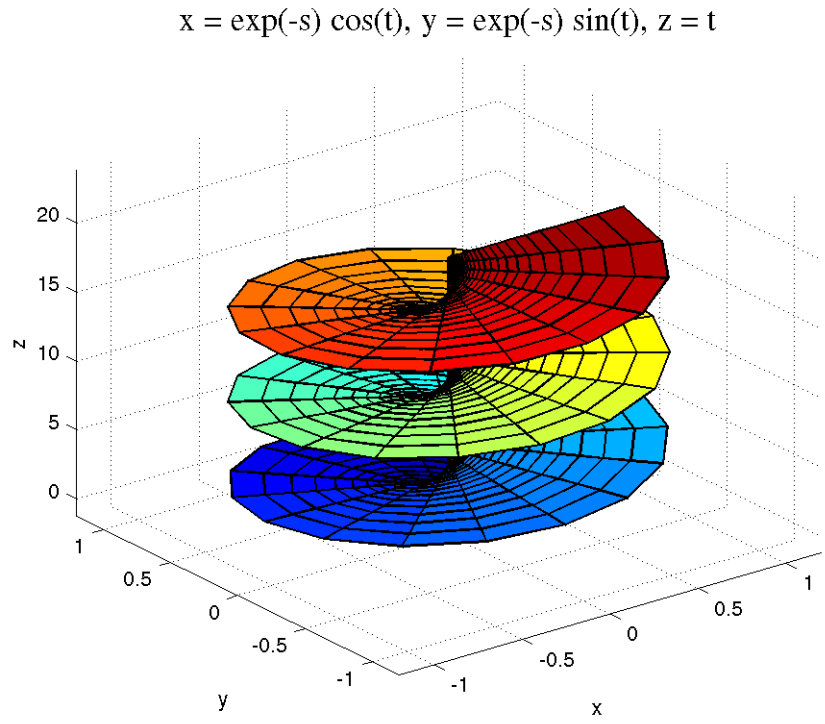
```
h = findobj('Type','surface');
set(h,'LineWidth',1)
```

Adjusting Fonts

To create a desired presentation in the printed document, this example changes the font size and the font type used in the title.

```
set(gca,'FontSize',8)
set(get(gca,'Xlabel'),'FontSize',8)
set(get(gca,'Ylabel'),'FontSize',8)
set(get(gca,'Zlabel'),'FontSize',8)
set(get(gca,'Title'),'FontSize',12,'FontName','times')
```

The following picture illustrates the results.



Specifying the Figure Size in Your Document

This example creates a figure in the word processor application that is five inches wide and four inches tall by setting the `PaperPosition` property from the **Page Setup** dialog box. Also, to preserve the axis tick marks spacing used in the original figure, check the **Match Axes Limits and Ticks** option in the **Limits** section of the dialog box. See a picture of the dialog box with these settings in the previous example.

Export the Figure to a File

To apply the **Page Setup** dialog box settings, click **OK** and return to the MATLAB command window. Issue the `print` command to create a TIFF file with a resolution of 200 dpi.

```
print –dtiff –r200 filename
```

MATLAB adds the `.tif` extension to the file name specified: *`filename`*`.tif`.

You can now import the file into the word processor application. For example, in Microsoft Word, insert a frame and then insert the picture into the frame. Avoid scaling or using the Word picture editor once the graphic is imported.

Setting Figure Printing Properties

Figure objects have several properties that control the size and aspect ratio of the printed graphic. You can set these properties from the command line or M-files using the `set` command or define default values for these properties if you want to change MATLAB's defaults. You can also set these properties from the printing dialog boxes, which is generally the easiest approach. The following table lists these properties and the dialog box where you can set them.

Property	Purpose	Set In Dialog
`InvertHardcopy`	Change figure colors for printing	**Copy Options**
`PaperOrientation`	Horizontal or vertical paper orientation	**Page Setup** or **Print Setup**
`PaperPosition`	Location and size of figure on printed page	**Page Setup**
`PaperPositionMode`	Use the `PaperPosition` values or the actual screen size of the figure	**Page Setup**, check **MatchFigure Screen Size** for `auto` and **Set Manual Position and Size** for `manual`
`PaperSize`	Size of `PaperType`	**Page Setup** (read only)
`PaperType`	Standard paper sizes	**Print Setup** (sets paper size)
`PaperUnits`	Units used by `PaperPosition` and `PaperSize`	**Page Setup**

Positioning the Figure on the Printed Page

The `PaperPosition` property is a four-element row vector that specifies the dimensions and position of the printed output. The form of the vector is:

```
[distance_from_left distance_from_bottom width height]
```

- *distance_from_*`left` – specifies the distance from the left edge of the paper to the left edge of the graphic.
- *distance_from_bottom* – specifies the distance from the bottom of the paper to the bottom of the graphic.
- *width* and *height* – specify the graphic's width and height.

Note that the window border is not included in the printed figure.

Resizing the Figure

By default, the value of `PaperPosition` does not change when you resize or reshape a figure. This means that the size of the printed output may not match the screen display. If you want the printed figure to match its size and shape on the screen, you can do one of the following:

- Choose **Page Setup** from the **File** menu. In the dialog box, check the box labeled **Match Figure Screen Size** in the **Size and Position** section.
- From the command line, enter:

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

In general, it is easiest to use the **Page Setup** dialog box to size and position the graphic on the printed page.

Automatic PaperPosition – WYSIWYG Printing

When `PaperPositionMode` is set to `auto`, the width and height of the printed figure are determined by the figure's dimensions on the screen, and the figure position is adjusted to center the figure on the page.

Note that when `PaperPositionMode` is `auto`, MATLAB sets the value of the `PaperPosition` property when you resize the figure. Therefore, if you change `PaperPositionMode` back to `manual` and then print the figure, the output will still be the same size as the figure is on screen, unless you also set `PaperPosition` to `default`.

If you want the default value of `PaperPositionMode` to be `auto`, enter this line in your `startup.m` file:

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

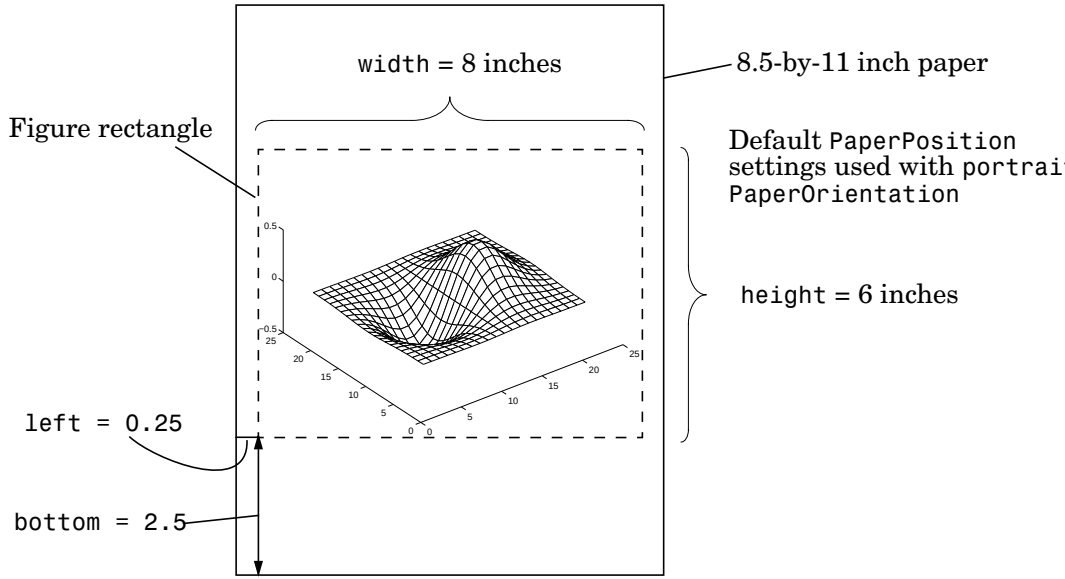
Setting `PaperPositionMode` to `auto` is especially important if you want to print out figures that include uicontrols or images. If `PaperPositionMode` is `manual`, these objects are likely to be distorted when you print them.

Factory Default PaperPosition

The factory default `PaperPosition`,

```
[0.25 2.5 8.0 6.0]
```

is designed to print the figure on 8.5-by-11 inch paper in portrait orientation. It results in a printed figure that is about as large as can fit on the paper (with a one-quarter inch border on the left and right sides) and is centered on the paper. The width of eight inches and the height of six inches give an aspect ratio (ratio of `width` to `height`) that is the same as the default figure aspect ratio on the screen.



Example – Readjusting PaperPosition

This example shows you how to recalculate paper position to achieve a specific layout.

Problem

You are working with figure windows of varying sizes and shapes and you want to determine the values to use for the `PaperPosition` property to print each one:

- As large as possible
- With the same aspect ratio as the figure on the screen
- With a minimum of a one-quarter inch border
- Centered on the page

Solution

To solve this problem in the general case, you need to know the values of the figure's `Position` and `PaperSize` properties, being careful to use the same units for all dimensions. With this information you can decide how to orient the paper and/or scale the figure size to fit properly.

Suppose a particular figure is five inches wide and three and one-half inches high and you want to print it on the default size paper at your site. You need to calculate a new value for `PaperPosition` based on these sizes.

Obtain the size of the figure and the paper, in inches.

```
set(gcf,'Units','inches','PaperUnits','inches')
figpos = get(gcf,'Position');
psize = get(gcf,'PaperSize');
```

Since the figure is wider than it is high, the width limits the maximum size that fits on the page. Furthermore, since the figure is smaller than the paper, you can set the `PaperPosition` width to the paper width minus a one-quarter inch border on both sides.

```
newpp(3) = psize(1)-.5;
```

Calculate the `PaperPosition` height in terms of the width, maintaining the correct aspect ratio.

```
newpp(4) = newpp(3)*figpos(4)/figpos(3);
```

To determine the offset from the bottom of the paper, subtract the new `PaperPosition` height (i.e., the printed height of the figure) from the paper

height, taking one-half this value to center the figure on the page from top to bottom.

```
newpp(2) = (psize(2)-newpp(4))/2;
```

The offset from the left is simply the border width.

```
newpp(1) = .25;
```

You have now fully specified the `PaperPosition` for this particular figure,

```
newpp =

    0.2500    2.7000    8.0000    5.6000
```

and can set the property for printing.

```
set(gcf,'PaperPosition',newpp)
```

Specifying Paper Orientation

By default, figures print in portrait mode. If you want to print in landscape mode, set the `PaperOrientation` property to `landscape`. You can do this in the **Page Setup** dialog box, or by entering this command at the command line:

```
set(gcf,'PaperOrientation','landscape')
```

You may also need to set the `PaperPosition` property so that the figure fits on the page.

If you set the paper orientation from the **Print Setup** dialog box or use the `orient` command, MATLAB sets the orientation of the printed output, by setting both the `PaperOrientation` and `PaperPosition` properties of the figure.

Specifying Paper Size

You can set the `PaperType` property to set the size of the paper you are printing on. The values for `PaperType` are standard paper sizes such as `usletter` and `a4`. When you set `PaperType`, MATLAB also sets `PaperSize`, which is a read-only property that indicates the actual dimensions for the current `PaperType`, in the current `PaperUnits`.

If you are printing on a paper size that MATLAB does not recognize, set the `PaperPosition` property to position the figure as you want it to be on the real

paper size. MATLAB will use the values you set, even if the size specified by `PaperPosition` is larger than the current value of `PaperSize`.

For example, suppose you want to print on paper that is 30 inches by 40 inches. You could set `PaperPosition` to `[.5 .5 29 39]`, either at the command line or in the **Page Setup** dialog box.

Changing the Default Paper Type

You can change the default value of the figure `PaperType` property by setting a default value on the root level:

```
set(0,'DefaultFigurePaperType','paper_type')
```

where *`paper_type`* is one of the values listed with the `PaperType` property description. This command changes the default for the duration of your MATLAB session. If you want the new default to be used whenever you start MATLAB, add this command to your `startup` M-file. Type

```
doc startup
```

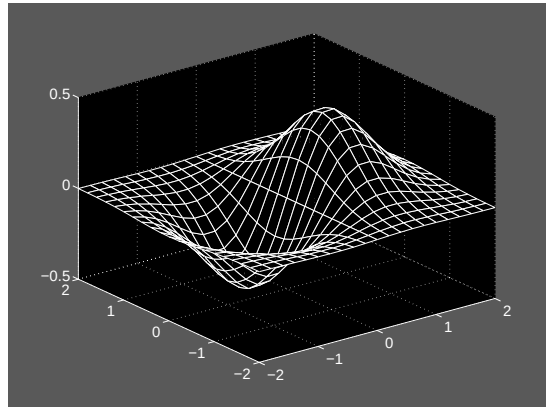
for more information.

Reversing Figure Colors

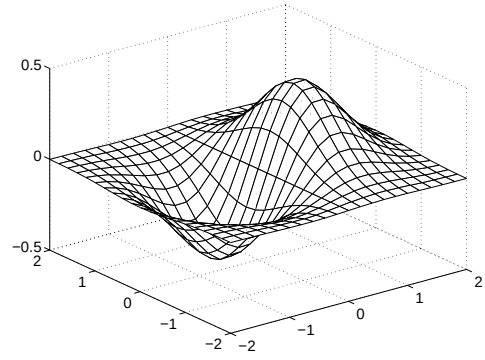
With the `colordef` command, it is possible to configure the default figure background color to black and the axis lines and labels to white. This color scheme provides good contrast on the computer screen, but is less desirable when printed on white paper by a black and white device such as a laser printer.

The figure `InvertHardcopy` property provides a simple way to convert the printed output to a white background. When `InvertHardcopy` is `on` (the default), MATLAB automatically inverts the color scheme to black-on-white output.

This mesh plot of a surface has a white `EdgeColor`. The white-on-black coloring produces a large black area on the printed page that results in poor discrimination between the mesh and background, particularly on low resolution printers.



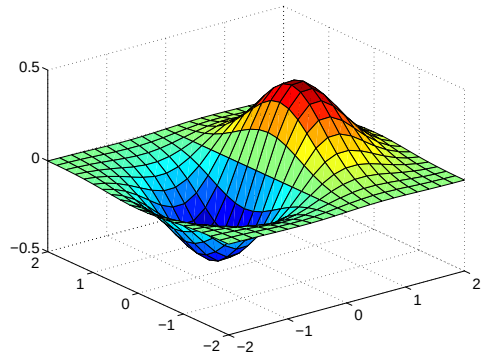
With `InvertHardcopy` enabled, MATLAB automatically produces output more suited to printing in black and white. Here is the same mesh plot after reversing the color scheme.



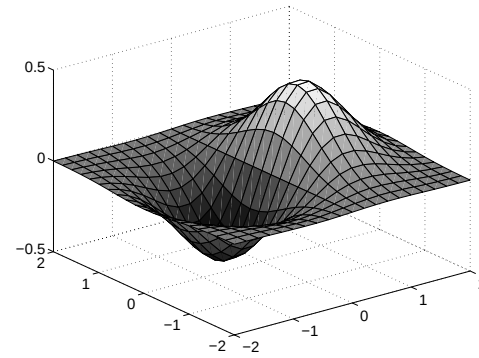
What Happens to Colors When Printing in B&W?

MATLAB changes the figure background color to white (and also changes the axes color unless it is set to `none`). Colors print in shades of gray on devices capable of printing grays.

For example, this surface uses the `jet` colormap and is printed with `InvertHardcopy` set to `on`.



In cases where you want to print colored objects in grayscale, you should use a colormap that varies continuously from dark to light, such as `gray`, `copper`, or `bone`. The same surface using the `gray` colormap prints in predictable shading.



12

Handle Graphics

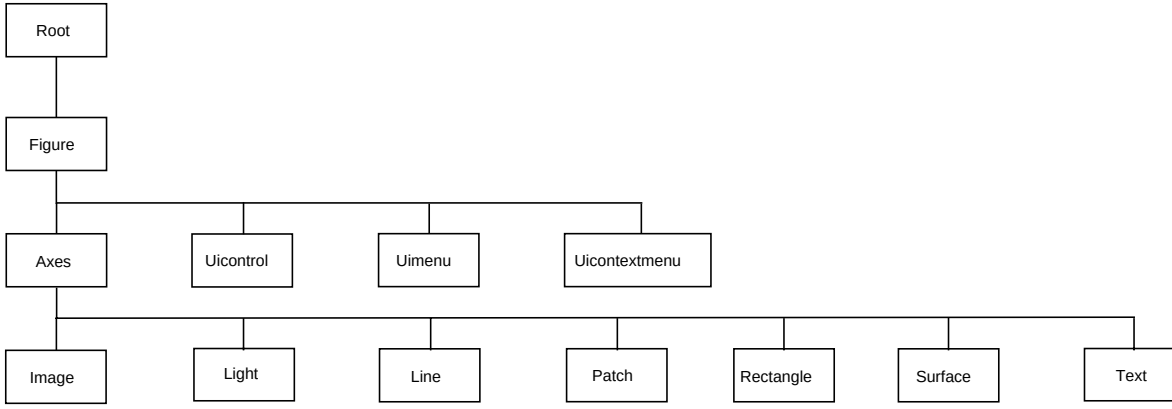
Handle Graphics is an object-oriented graphics system that provides the components necessary to create computer graphics. It supports drawing commands to create lines, text, meshes, and polygons as well as interactive devices such as menus, pushbuttons, and dialog boxes.

With Handle Graphics, you can directly manipulate the lines, surfaces, and other graphics elements that MATLAB's high-level routines use to produce various types of graphs. You can use Handle Graphics from the MATLAB command line to modify the display or in M-files to create customized graphics functions.

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.

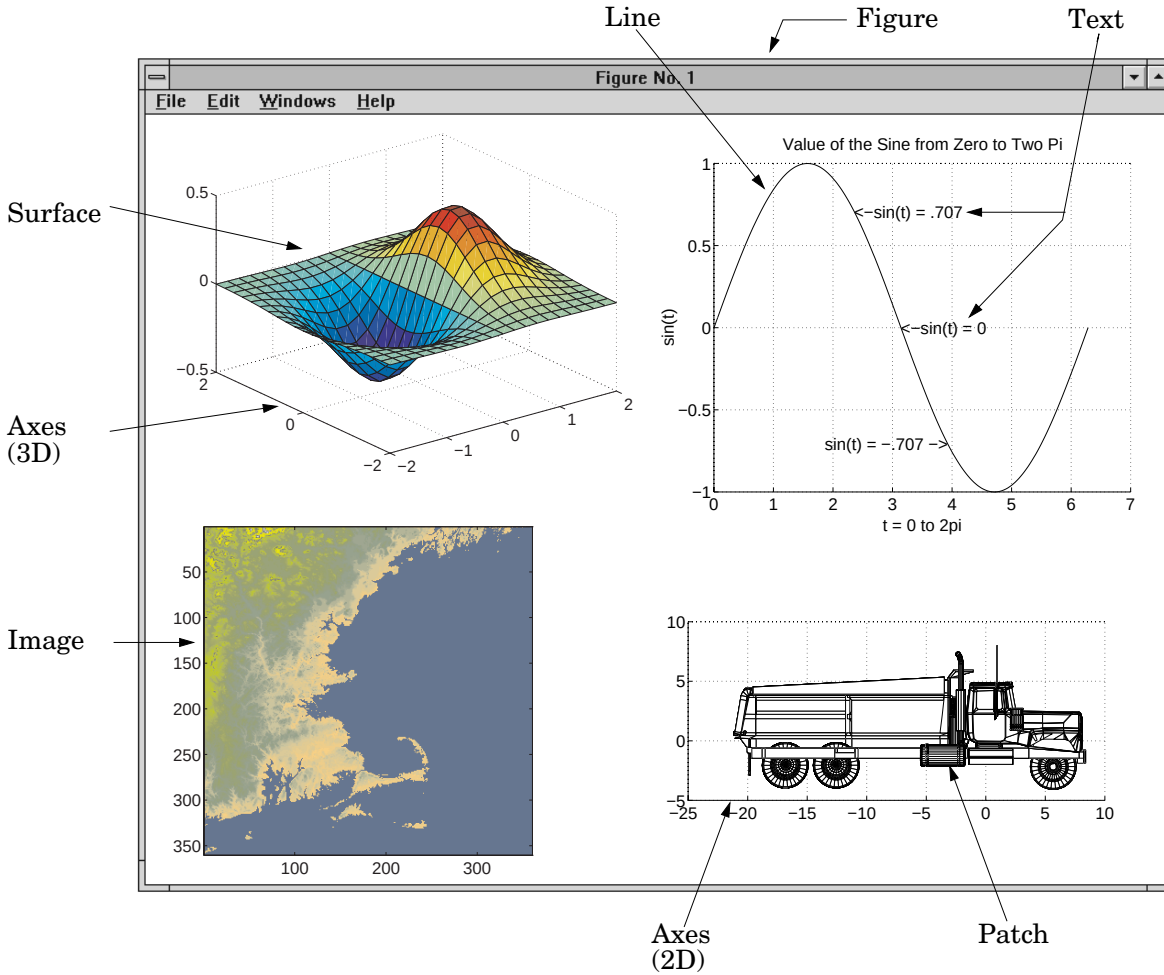
These objects are organized into a tree-structured hierarchy.



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.

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). 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

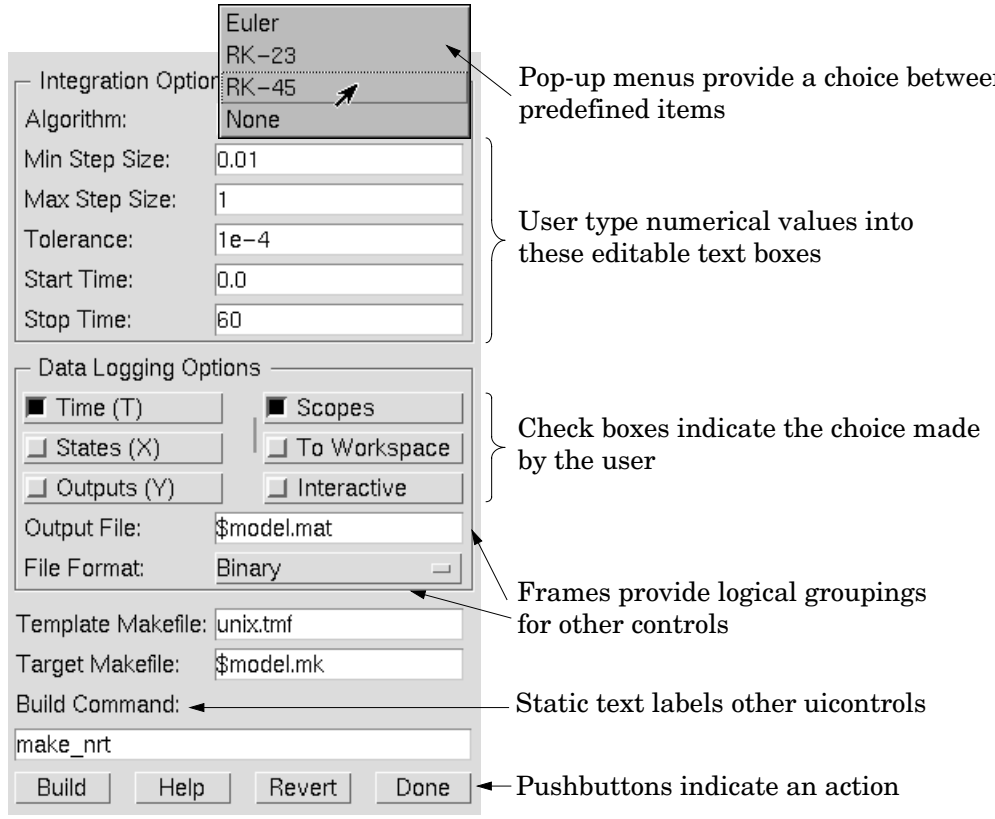
Figure objects are the individual windows on the root screen where MATLAB displays graphics. MATLAB places no limits on the number of figure windows you can create (your computer may, however). All figures are children of the root and all other graphics objects are descendants of figures.

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.

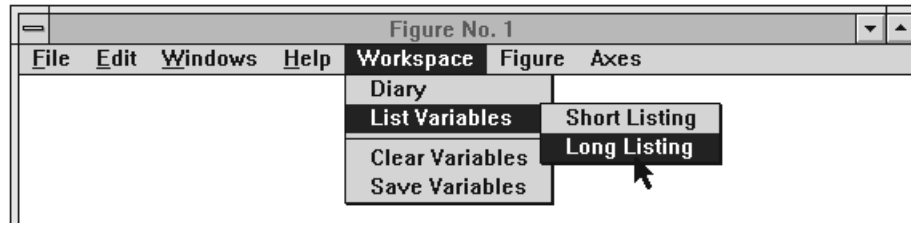
You can use uicontrols in combinations to construct control panels and dialog boxes. Pop-up menus, editable text boxes, check boxes, pushbuttons, static text, and frames compose this particular example.



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 uimenus 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 uimenus defined (titled **Workspace**, **Figure**, and **Axes**). Two levels of submenus are visible under **Workspace** top-level uimenu.



Uimenus 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 uimenus are not children of axes).

Image

A MATLAB image consists of a data matrix and possibly a colormap. There are three basic image types that differ in the way that data matrix elements are interpreted as pixel colors – indexed, intensity, and truecolor. Since images are strictly 2-D, you can view them only at the default 2-D view.

Light

Light objects define light sources that affect all patch and surface objects within the axes. You cannot see lights, but you can set properties that control the style of light source, color, location, and other properties common to all graphics objects.

Line

Line objects are the basic graphics primitives used to create most 2-D and some 3-D plots. High-level functions `plot`, `plot3`, and `loglog` (and others) create line objects. The coordinate system of the 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 ot 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
`uicontrol`	Programmable user-interface device, such as pushbutton, slider, or listbox.
`uimenu`	Programmable menu appearing at the top of a figure window.

All object creation functions have a similar format:

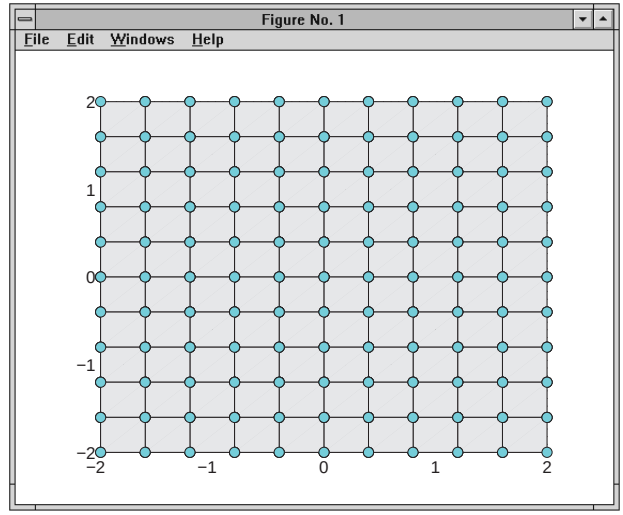
```
handle = function('propertyname',propertyvalue,...)
```

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:

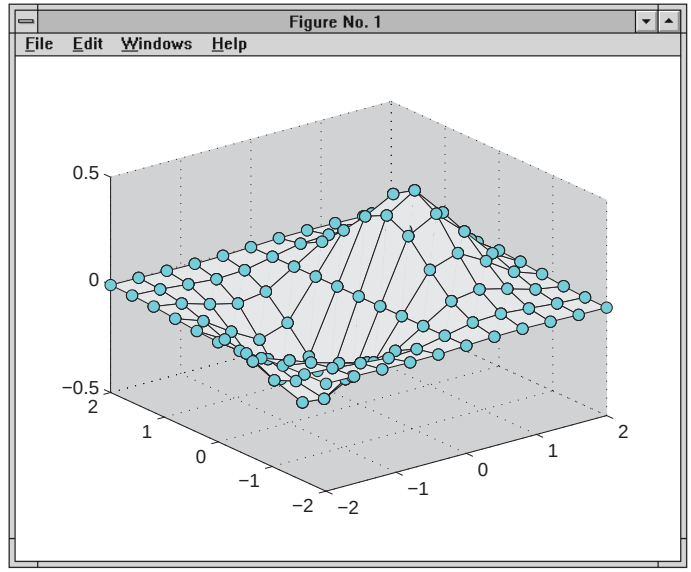
```
[x,y] = meshgrid([-2:.4:2]);
Z = x.*exp(-x.^2-y.^2);
fh = figure('Position',[350 275 400 300],'Color','w');
ah = axes('Color',[.8 .8 .8],'XTick',[-2 -1 0 1 2],...
          'YTick',[-2 -1 0 1 2]);
sh = surface('XData',x,'YData',y,'ZData',Z,...
             'FaceColor',get(ah,'Color')+.1,...
             '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 Vs. Low-Level

MATLAB's high-level graphics routines (e.g., `plot` or `surf`) call the appropriate object creation function to draw graphics objects. However, high-level routines also clear the axes or create a new figure, depending on the settings of the axes and figure `NextPlot` properties.

In contrast, object creation functions simply create their respective graphics objects and place them in the current parent object. They do not respect the setting of the figure 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 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 `XMinorTickMode`. While this makes property names easier to read, MATLAB does not check for uppercase letters. In addition, you need use only enough letters to identify the name uniquely, so you can abbreviate most property names.

In M-files, however, using the full property name can prevent problems with futures releases of MATLAB if a shortened name is no longer unique because of the addition of new properties.

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.

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_hanlde,'PropertyName');
```

Property names are always quoted strings. Property values depend on the particular property.

See Accessing Object Handles 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,'Factory<ObjectTypePropertyName>')
```

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 Defalut Values

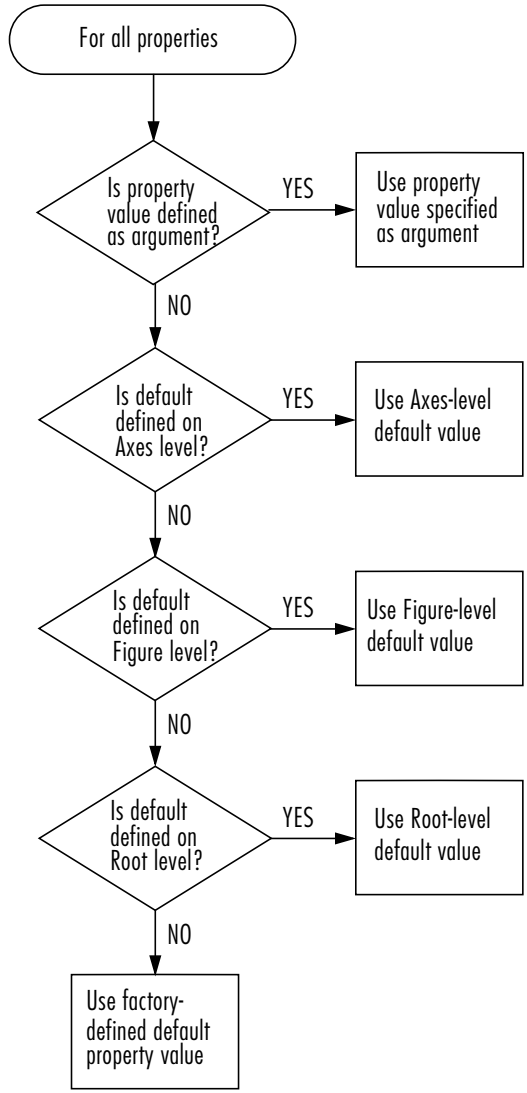
MATLAB’s search for a default value begins with the current object and continues through the object’s ancestors until it finds a user-defined default value or until it reaches the factory-defined value. Therefore, a search for property values is always satisfied.

The closer to the root of the hierarchy you define the default, the broader 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.


For all properties
Is property value defined as argument?
YES
Use property value specified as argument
NO
Is default defined on Axes level?
YES
Use Axes-level default value
NO
Is default defined on Figure level?
YES
Use Figure-level default value
NO
Is default defined on Root level?
YES
Use Root-level default value
NO
Use factory-defined default property value

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

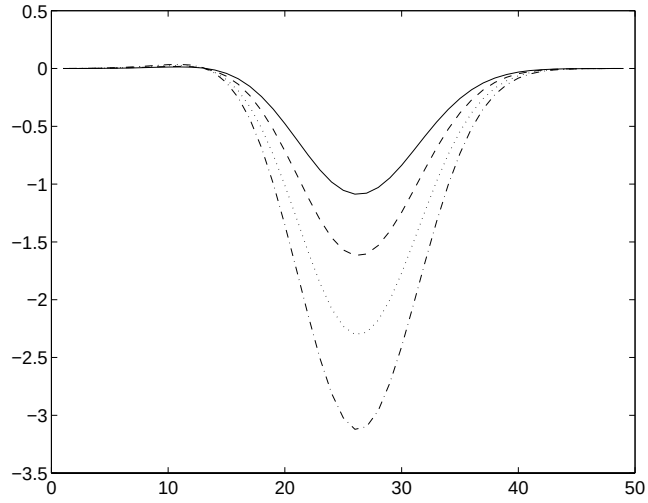
This example creates a figure with a white plot (axes) background color, then sets default values for axes objects on the root level.

```
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 linestyles defined for `LineStyleOrder`.

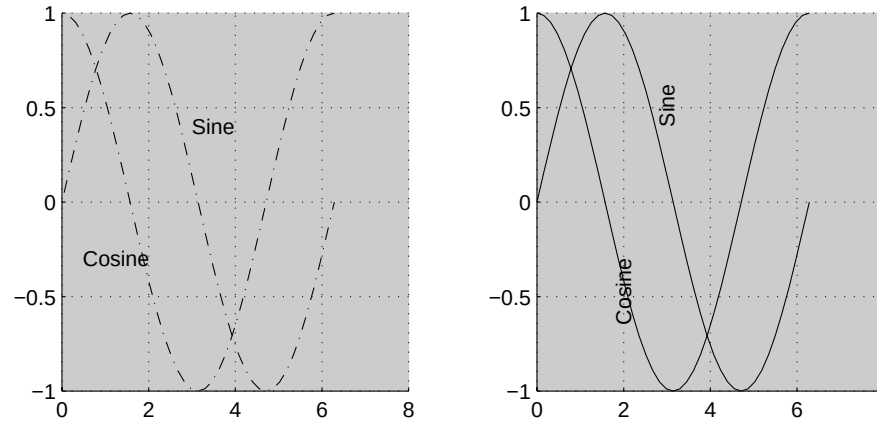


Second Example

This example sets default values on more than one level in the hierarchy. These statements create two axes in one figure window, setting default values on the figure level and the axes level.

```
t = 0:pi/20:2*pi;
s = sin(t);
c = cos(t);
% Set default value for axes Color property
figh = figure('Position',[30 100 800 350],...
              'DefaultAxesColor',[.8 .8 .8]);

axh1 = subplot(1,2,1); grid on
% Set default value for line LineStyle property in first axes
set(axh1,'DefaultLineLineStyle','-.')
line('XData',t,'YData',s)
line('XData',t,'YData',c)
text('Position',[3 .4],'String','Sine')
text('Position',[2 -.3],'String','Cosine',...
     'HorizontalAlignment','right')

axh2 = subplot(1,2,2); grid on
% Set default value for text Rotation property in second axes
set(axh2,'DefaultTextRotation',90)
line('XData',t,'YData',s)
line('XData',t,'YData',c)
text('Position',[3 .4],'String','Sine')
text('Position',[2 -.3],'String','Cosine',...
     'HorizontalAlignment','right')
```

Issuing the same `line` and `text` statements to each subplot region results in a different display, reflecting different default settings.



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 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

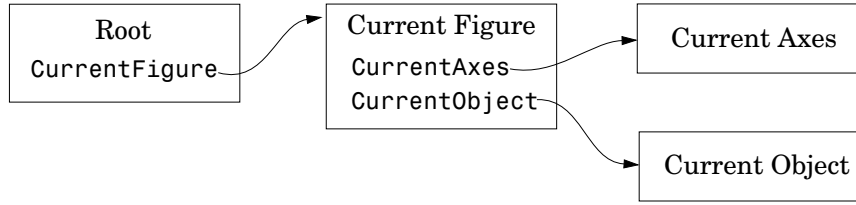
The figure `IntegerHandle` property controls which type of handle the figure receives.

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.

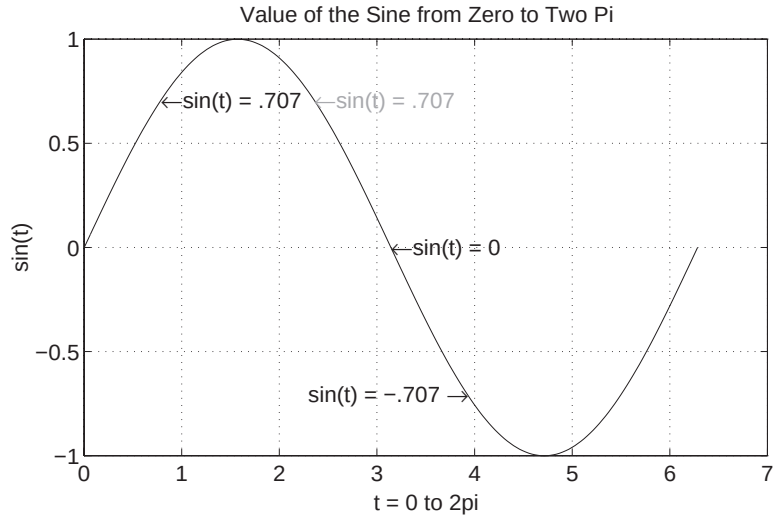
See “Controlling Graphics Output” for information on how to prevent users from accessing the handles of graphics objects that you want to protect.

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','\leftarrowsin(t) = .707');
```

Next move the object to the new position, defining the text `Position` in axes units.

```
set(text_handle,'Position',[3*pi/4,sin(3*pi/4),0])
```

`findobj` also lets you restrict the search by specifying a starting point in the hierarchy, instead of beginning with the root object. This results in faster searches if there are many objects in the hierarchy. In the previous example, you know the text object of interest is in the current axes so you can type:

```
text_handle = findobj(gca,'String','\leftarrowsin(t) = .707');
```

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.

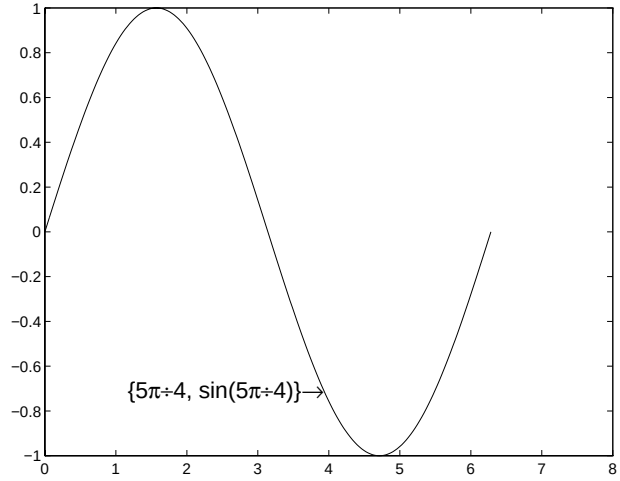
Example – Copying Objects

Suppose you are plotting a variety of data and want to label the point having the x- and y-coordinates determined by $5\pi \div 4$, $\sin\langle 5\pi \div 4\rangle$ 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 $\{5\pi \div 4, \sin\langle 5\pi \div 4\rangle\}$ 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`).

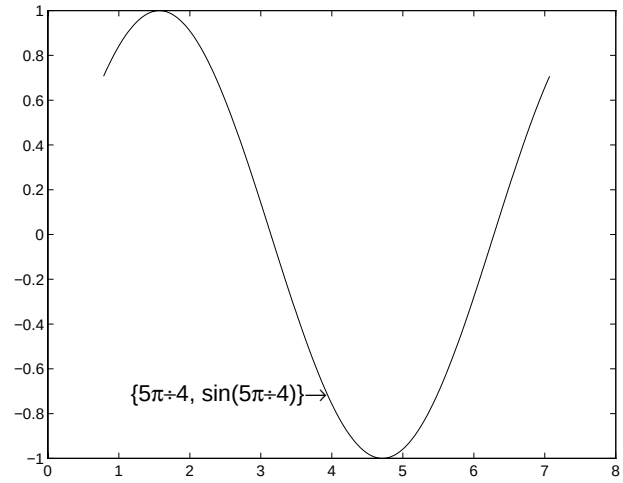


To label the same point with the same string in another plot, copy the text using `copyobj`. Since the last statement did not save the handle to the text object, you can find it using `findobj` and the `'String'` property.

```
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).
```



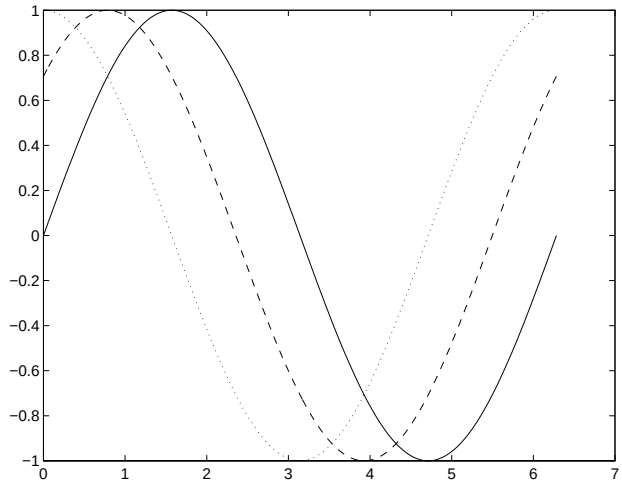
This particular example takes advantage of the fact that text objects define their location in the axes' data space. Therefore the text `Position` property did not need to change from one plot to another.

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's drawing behavior is essential to creating a program that behaves predictably.

This table summarizes the possible values for the `NextPlot` property:

NextPlot	Figure	Axes
`add`	Add new graphics objects without clearing or resetting the current figure. (Default setting)	Add new graphics objects without clearing or resetting the current axes.
`replacechildren`	Remove all child objects, but do not reset figure properties. Equivalent to `clf`.	Remove all child objects, but do not reset axes properties. Equivalent to `cla`.
`replace`	Remove all child objects and reset figure properties to their defaults. Equivalent to `clf reset`.	Remove all child objects and reset axes properties to their defaults. Equivalent to `cla reset`. (Default setting)

Note that a `reset` returns all properties, except `Position` and `Units`, to their default values.

The `hold` command provides convenient access to the `NextPlot` properties. The statement

```
hold on
```

sets both figure and axes `NextPlot` to `add`.

The statement

```
hold off
```

sets the axes `NextPlot` property to `replace`.

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's Default Behavior

Consider the default situation where the figure `NextPlot` property is `add` and the axes `NextPlot` property is `replace`. When you call `newplot`, it:

1. Checks the value of the current figure's `NextPlot` property (which is `add`) and determines MATLAB can draw into the current figure with no further action (if there is no current figure, `newplot` creates one, but does not recheck its `NextPlot` property).

2. Checks the value of the current axes' `NextPlot` property (which is `replace`), deletes all graphics objects from the axes, reset all axes properties (except `Position` and `Units`) to their defaults, and returns the handle of the current axes.

Example – Using newplot

To illustrate the use of `newplot`, this example creates a function that is similar to the built-in `plot` function, except it automatically cycles through different line styles instead of using different colors for multiline plots:

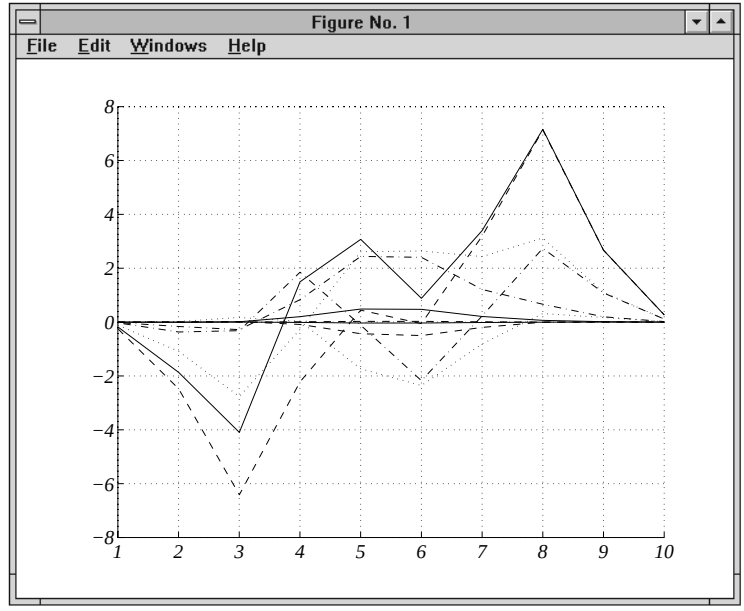
```
function my_plot(x,y)
cax = newplot; % newplot returns handle of current axes
LSO = ['- ';'--';': ';'-.'];
set(cax,'FontName','Times','FontAngle','italic')
set(get(cax,'Parent'),'MenuBar','none') %
line_handles = line(x,y,'Color','b');
style = 1;
for i = 1:length(line_handles)
    if style > length(LSO), 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.

MATLAB's default settings for the NextPlot properties facilitate writing M-files that adhere to MATLAB's standard behavior: reuse the figure window, but clear and reset the axes with each new graph. Other values for these properties allow you to implement different behaviors.

Replacing Only the Children Objects —replacechildren

The replacechildren value for NextPlot causes newplot to remove child objects from the figure or axes, but does not reset any property values (except the list of handles contained in the Children property).

This can be useful after setting properties you want to use for subsequent graphs without having to reset properties. For example, if you type on the command line

```
set(gca,'ColorOrder',[0 0 1],'LineStyleOrder','-|--|:|-.',...
        'NextPlot','replacechildren')
plot(x,y)
```

plot produces the same output as the M-file my_plot in the previous section, but only within the current axes. Calling plot still erases the existing graph (i.e., deletes the axes children), but it does not reset axes properties. The values specified for the ColorOrder and LineStyleOrder properties remain in effect.

Testing for Hold State

There are situations in which your M-file should change the visual appearance of the axes to accommodate new graphics objects. For example, if you want the M-file my_plot from the previous example to accept 3-D data, it makes sense to set the view to 3-D when the input data has *z*-coordinates.

However, to be consistent with the behavior of MATLAB's high-level routines, it is a good practice to test if hold is on before changing parent axes or figure properties. When hold is on, the axes and figure NextPlot properties are both set to add.

The M-file, `my_plot3`, accepts 3-D data and also checks the hold state, using `ishold`, to determine if it should change the view.

```
function my_plot3(x,y,z)
cax = newplot;
hold_state = ishold; % 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 MATLAB's handle access functions, while ensuring that users typing at the command line do not inadvertently disturb a protected object.
- `off` – The object's handle is hidden from all functions executing on the command line and in callback routines. This setting is useful when you want to protect objects from possibly damaging user commands.

For example, if a GUI accepts user input in the form of text strings, which are then evaluated (using the `eval` function) from within the callback routine, a string such as `'close all'` could destroy the GUI. To protect against this situation, you can temporarily set `HandleVisibility` to `off` on key objects.

```
user_input = get(editbox_handle,'String');
set(gui_handles,'HandleVisibility','off')
eval(user_input)
set(gui_handles,'HandleVisibility','commandline')
```

Values Returned by gca and gcf. When a protected figure is topmost on the screen, but has nonprotected figures stacked beneath it, `gcf` returns the topmost unprotected figure in the stack. The same is true for `gca`. If no unprotected figures or axes exist, calling `gcf` or `gca` causes MATLAB to create one in order to return its handle.

Accessing Protected Objects

The root `ShowHiddenHandles` property enables and disables handle visibility control. By default, `ShowHiddenHandles` is `off`, which means MATLAB obeys the setting of the `HandleVisibility` property. When set to `on`, all handles are visible from the command line and within callback routines. This can be useful

when you want access to all graphics objects that exist at a given time, including the handles of axes text labels, which are normally hidden.

The `close` function also allows access to nonvisible figures using the `hidden` option. For example,

```
close('hidden')
```

closes the topmost figure on the screen, even if it is protected. Combining `all` and `hidden` options,

```
close('all','hidden')
```

closes all figures.

The Close Request Function

MATLAB executes a callback routine defined by the figure's `CloseRequestFcn` whenever you:

- Issue a `close` command on a figure.
- Quit MATLAB while there are visible figures. (If a figure's `Visible` property is set to `off`, MATLAB does not execute its close request function when you quit MATLAB; the figure is just deleted).
- Close a figure from the windowing system using a close box or a close menu item.

The close request function enables you to prevent or delay the closing of a figure or the termination of a MATLAB session. This is useful to perform such actions as:

- Displaying a dialog box requiring the user to confirm the action.
- Saving data before closing.
- Preventing unintentional command-line deletion of a graphical user interface built with MATLAB.

The default callback routine for the CloseRequestFcn is an M-file called closereq. It contains the statements:

```
shh=get(0,'ShowHiddenHandles');
set(0,'ShowHiddenHandles','on');
delete(get(0,'CurrentFigure'));
set(0,'ShowHiddenHandles',shh);
```

This callback disables HandleVisibility control by setting the root ShowHiddenHandles property to on, which makes all figure handles visible.

Quitting MATLAB

When you quit MATLAB, the current figure's CloseRequestFcn is called, and if the figure is deleted, the next figure in the root's list of children (i.e., the root's Children property) becomes the current figure, and its CloseRequestFcn is in turn executed, and so on. 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 MATLAB's state in the beginning of your M-file. For example, you can begin an M-file with

```
cax = newplot;
cfig = get(cax,'Parent');
hold_state = ishold;
```

rather than querying this information each time you need it. Remember that utility commands like `ishold` obtain the values they return whenever called. (The `ishold` command issues a number of `get` commands and string compares (`strcmp`) to determine the hold state.)

If you are temporarily going to alter the hold state within the M-file, you should save the current values of the `NextPlot` properties so you can reset them later.

```
ax_nextplot = lower(get(cax,'NextPlot'));
fig_nextplot = lower(get(cfig,'NextPlot'));
.
.
.
set(cax,'NextPlot',ax_nextplot)
set(cfig,'NextPlot',fig_nextplot)
```

Properties Changed by Built-In Functions

To achieve their intended effect, many built-in functions change axes properties, which can then affect the workings of your M-file. This table lists MATLAB's built-in graphics functions and the properties they change. Note that these properties change only if `hold` is `off`.

Function	Axes Property: Set To
`fill`	`Box: on` `CameraPosition`: 2-D view `CameraTarget`: 2-D view `CameraUpVector`: 2-D view `CameraViewAngle`: 2-D view
`fill3`	`CameraPosition`: 3-D view `CameraTarget`: 3-D view `CameraUpVector`: 3-D view `CameraViewAngle`: 3-D view `XScale`: linear `YScale`: linear `ZScale`: linear

`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`

Function	Axes Property: Set To

`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

`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`

13

Figure Properties

Introduction

Figure graphics objects are the windows in which MATLAB displays graphical output. Figure properties allow you to control many aspects of these windows, such as their size and position on the screen, the coloring of graphics objects displayed within them, and the scaling of printed pictures.

This section discusses some of the features that are implemented through figure properties and provides examples of how to use these features. The table listing all figure properties provides an overview of the characteristics affected by figure properties.

Positioning Figures

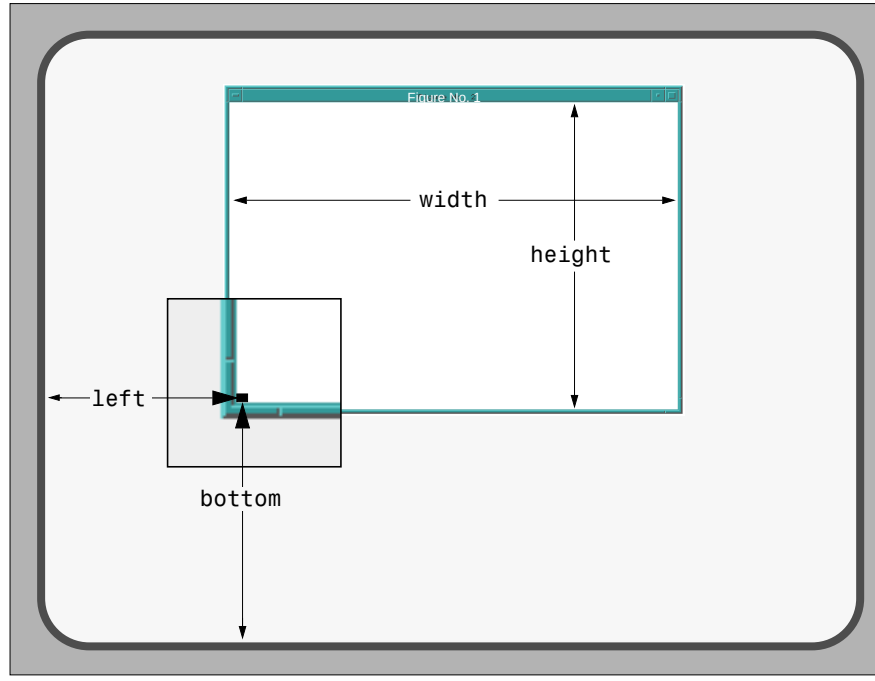
The figure `Position` property controls the size and location of the figure window on the root screen. At startup, MATLAB determines the size of your computer screen and defines a default value for `Position`. This default creates figures about one-quarter of the screen's size and places them centered 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:

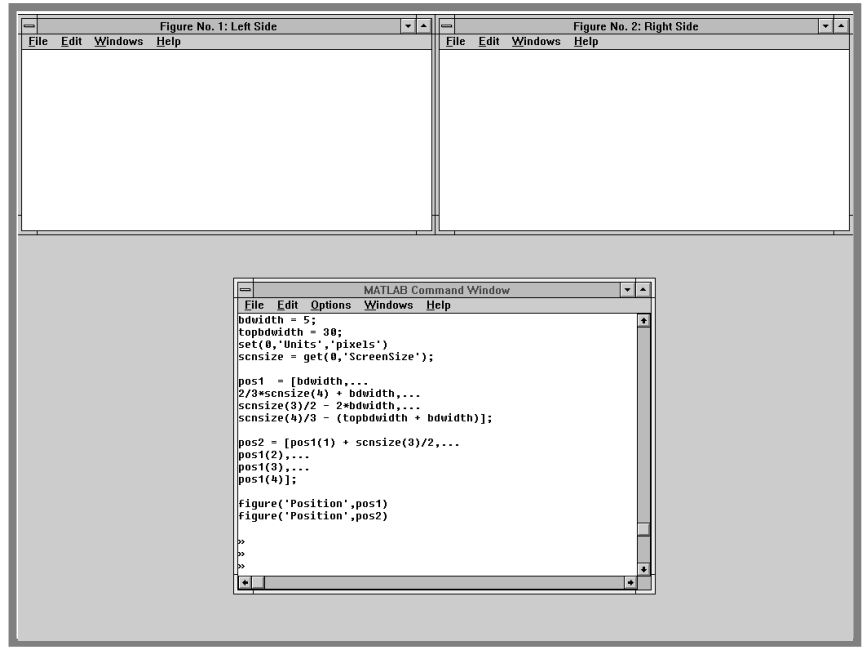


Figure No. 1: Left Side
File Edit Windows Help
Figure No. 2: Right Side
File Edit Windows Help
MATLAB Command Window
File Edit Options Windows Help
bdwidth = 5;
topbdwidth = 30;
set(0,'Units','pixels')
scnsize = get(0,'ScreenSize');
pos1 = [bdwidth,...
2/3*scnsize(4) + bdwidth,...
scnsize(3)/2 - 2*bdwidth,...
scnsize(4)/3 - (topbdwidth + bdwidth)];
pos2 = [pos1(1) + scnsize(3)/2,...
pos1(2),...
pos1(3),...
pos1(4)];
figure('Position',pos1)
figure('Position',pos2)
»
»
»

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 this 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.

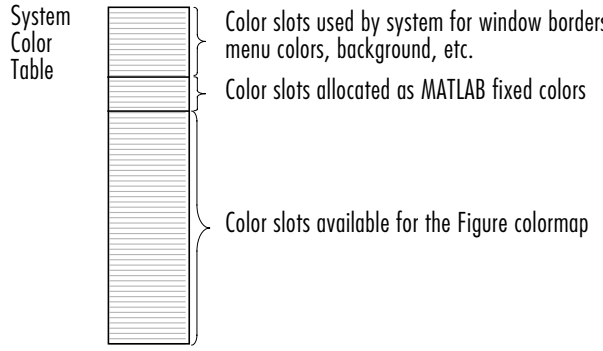
Indexed Color Displays

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.

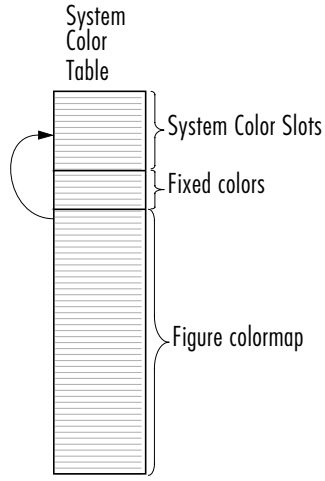
More Details

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).

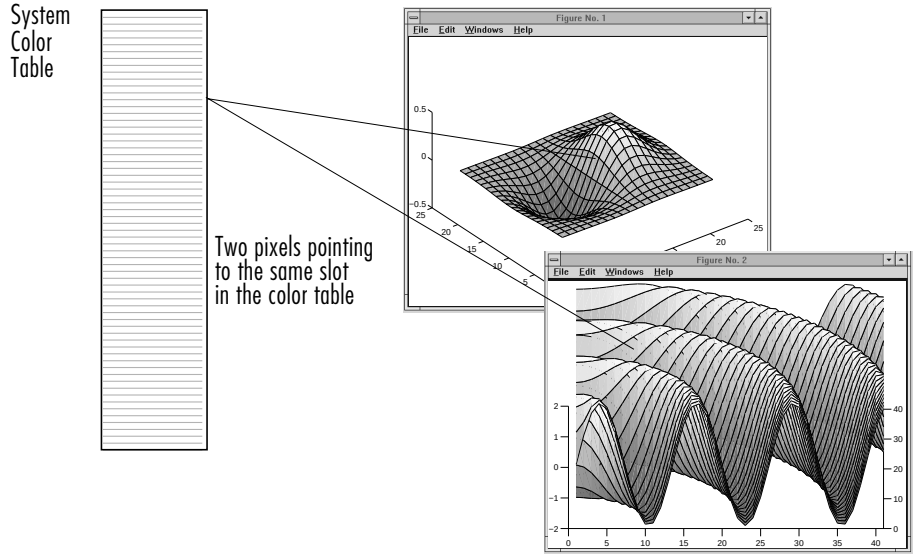
Nonactive Figures and Shared Colors

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 rerendering (i.e., recomputing color values and reassigning pixels to these colors) of the figure whose colormap you are altering.

If you want to change a figure's colormap rapidly, you should disable color sharing:

```
set(fig_handle,'ShareColors','off')
```

Note that the new colormap must be the same size as the original one to avoid rerendering the figure. Look at the `spinmap` M-file for an example of this technique.

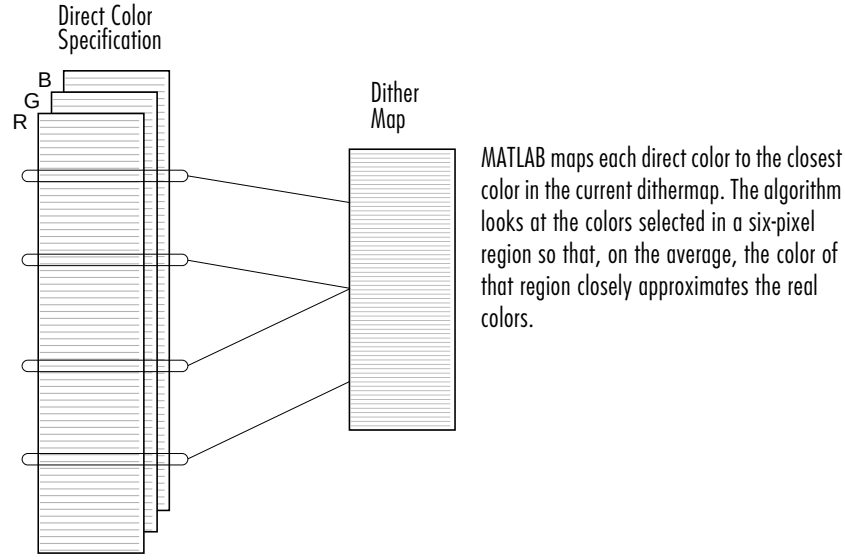
Dithering Truecolor on Indexed Color Systems

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. Index color systems interpret truecolor specifications by mapping each color to the closest color in the dithermap, which is assigned to the `Dithermap` property. MATLAB uses the Floyd-Steinberg algorithm to perform the mapping.



The dithermap is a colormap that replaces the figure colormap (which is not used in this case). The default dithermap contains a sampling of colors from the entire spectrum. This produces reasonably good quality with any object coloring. However, if the figure contains objects of primarily one color, a dithermap concentrated in the same color produces better color resolution.

Auto Dither Mode

When you set `DithermapMode` to `auto`, MATLAB automatically creates a dithermap based on the colors in the figure. MATLAB produces an appropriate dithermap using the minimum variance quantization algorithm; however, the process is time consuming. Also, MATLAB regenerates the dithermap each time it re-renders the figure.

To avoid excessive rendering time, you should reset `DithermapMode` to `manual` after MATLAB generates the dithermap. MATLAB then uses this dithermap without regenerating it until you once again set `DithermapMode` to `auto`. You do not need to regenerate the dithermap unless you change the colors used in the figure.

You can save a dithermap by assigning the `Dithermap` property to a variable and saving it as a MAT-file.

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

MATLAB creates a dithermap, which you can then save.

```
dmap = get(gcf,'Dithermap');
save DitherMaps dmap
```

Dithermap Size

To obtain the highest color resolution, the default dithermap is as large as the system allows. This is usually less than 256 colors because a certain number of slots are reserved for system colors. Also, MATLAB fixed colors are not overwritten by the dithermap.

Effects of Dithering

Dithering reduces the resolution of the displayed graphics because the colors are mapped in groups of six pixels. For example, suppose the color of one pixel is defined as orange, but the dithermap does not have this color. MATLAB selects combinations of colors from the dithermap that, taken together as a six-pixel group, approximate the color orange.

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. Double buffering works only when the figure `Renderer` property is set to `painters`.

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 `RendererMode` property to `auto` (the default).

Printing from Z-Buffer. You can select the resolution of the PostScript file produced by the `print` command using the `–r` option. By default, MATLAB prints Z-buffered figures at a medium resolution of 150 dpi (the default with `Renderer` set to `painters` is 864 dpi).

The size of the file generated from a Z-buffer figure does not depend on its contents, just the size of the figure. To decrease the file size, make the `PaperPosition` property smaller before printing (or set `PaperPositionMode` to `auto` and resize the figure window). See Printing MATLAB Graphics for more information.

OpenGL

OpenGL is available on many computer systems. It is generally faster than either painters or zbuffer and in some cases enables MATLAB to uses 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.

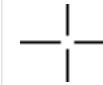
Printing from OpenGL. When printing a figure rendered with OpenGL, MATLAB switches to the zbuffer renderer.

Specifying the Figure Pointer

MATLAB indicates the position of the pointer within the figure window using a graphical symbol. You can select a pointer from 15 predefined symbols (see table) or you can define your own symbol. By convention, each of the predefined symbols has a purpose associated with it (although MATLAB enforces no rules for the use of any symbols).

You specify the pointer symbol by setting the value of the figure `Pointer` property. This table shows the predefined symbols, the associated specifier, and describes 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`	
Use as popular symbol	`fleur`	

Purpose	Specifier	Typical Symbol
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	`custom`	

Defining Custom Pointers

When you set the `Pointer` property to `custom`, MATLAB displays the pointer you define using the `PointerShapeCData` and the `PointerShapeHotSpot` properties. Custom pointers are 16-by-16 pixels, where each pixel can be either black, white, or transparent.

Specify the pointer by creating a 16-by-16 matrix containing elements that are:

- `1`s where you want the pixel black
- `2`s where you want the pixel white
- `Nan`s where you want the pixel transparent

Assign the matrix to the figure `PointerShapeCData` property. MATLAB displays the defined pointer whenever the pointer is in the figure window.

The `PointerShapeHotSpot` property specifies the pixel that indicates the pointer location. MATLAB then stores this location in the root `PointerLocation` property. Set the `PointerShapeHotSpot` property to a two-element vector specifying the row and column indices in the `PointerShapeCData` matrix that 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

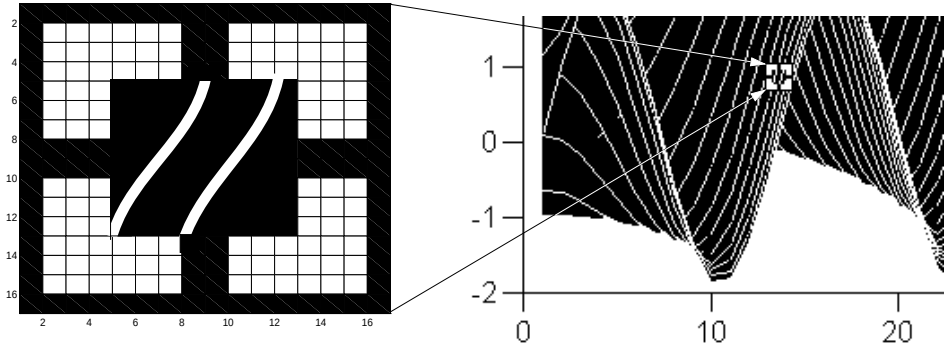
One way to create a custom pointer is to assign values to a 16-by-16 matrix by hand, as illustrated in the following example.

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:

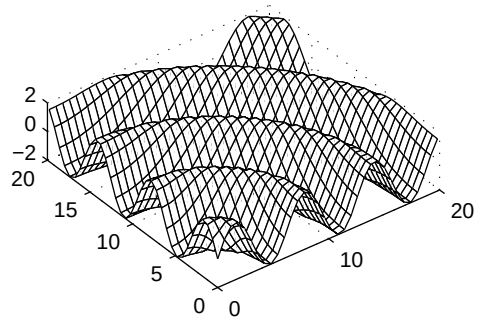


Creating Pointers from Functions. You can use a mathematical function to define the `PointerShapeCData` matrix. For example, evaluating the function,

$$2(\sin(\sqrt{x^2 + y^2}))$$

```
g = 0:.2:20;
[X,Y] = meshgrid(g);
Z = 2*sin(sqrt(X.^2 + Y.^2));
mesh(Z);
```

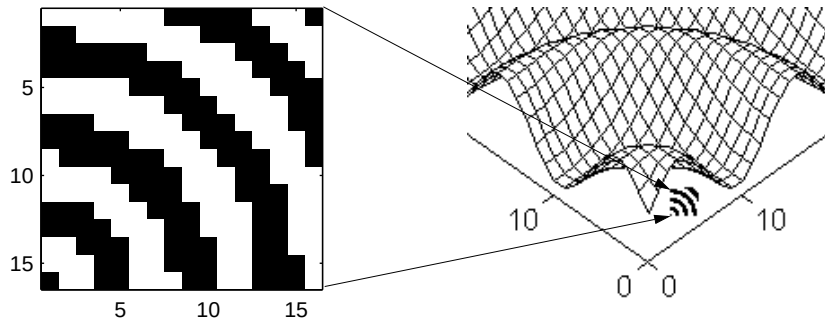
produces an interesting surface.



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.


5
10
15
5
10
15
10
10
0
0

Interactive Graphics

Figure objects contain a number of properties designed to facilitate user interaction with the figure. These properties fall into two categories.

Properties related to callback routine execution:

- `BusyAction`
- `ButtonDownFcn`
- `DestroyFcn`
- `KeyPressFcn`
- `Interruptible`
- `ResizeFcn`
- `WindowButtonDownFcn`, `WindowButtonMotionFcn`, and `WindowButtonUpFcn`

Properties that contain information about MATLAB's state:

- `CurrentAxes`
- `CurrentCharacter`
- `CurrentMenu`
- `CurrentObject`
- `CurrentPoint`
- `CurrentProperty`
- `SelectionType`

The manual, *Building GUIs with MATLAB,* provides information on creating programs that incorporate interactive graphics.

14

Axes Properties

Axes Properties

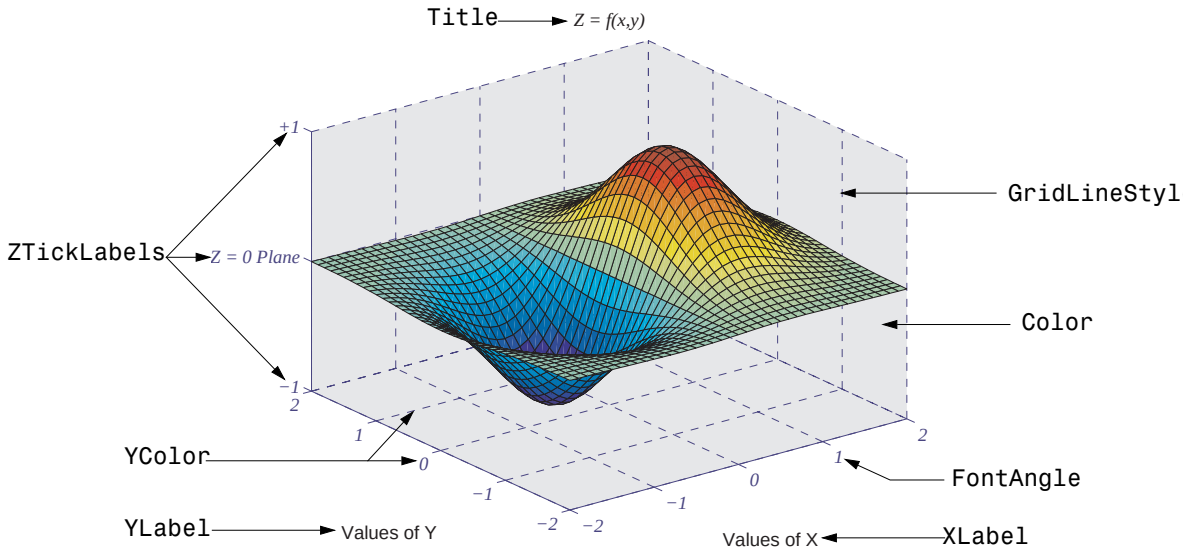
Axes are the parents of image, light, line, patch, rectangle, surface, and text graphics objects. These objects are the entities used to draw graphs of numerical data and pictures of real-world objects, such as airplanes or automobiles. Axes orient and scale their child objects to produce a particular 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 uses these features.

The table listing all axes properties provides an overview of the characteristics affected by axes properties.

Labeling and Appearance Properties

MATLAB provides a number of properties for labeling and controlling the appearance of axes. For example, this surface plot shows some of the labeling possibilities and indicates the controlling property.



Creating Axes with Specific Characteristics

To create this axes, specify values for the indicated properties.

```
h = axes('Color',[.9 .9 .9],...
         'GridLineStyle','--',...
         'ZTickLabels','-1|Z = 0 Plane|+1',...
         'FontName','times',...
         'FontAngle','italic',...
         'FontSize',14,...
         'XColor',[0 0 .7],...
         'YColor',[0 0 .7],...
         'ZColor',[0 0 .7]);
```

Axis Labels

The individual axis labels are text objects whose handles are normally hidden from the command line (their `HandleVisibility` property is set to `callback`). You can use the `xlabel`, `ylabel`, `zlabel`, and `title` functions to create axis labels. However, these functions affect only the current axes. If you are labeling axes other than the current axes by referencing the axes handle, then you must obtain the text object handle from the corresponding axes property. 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

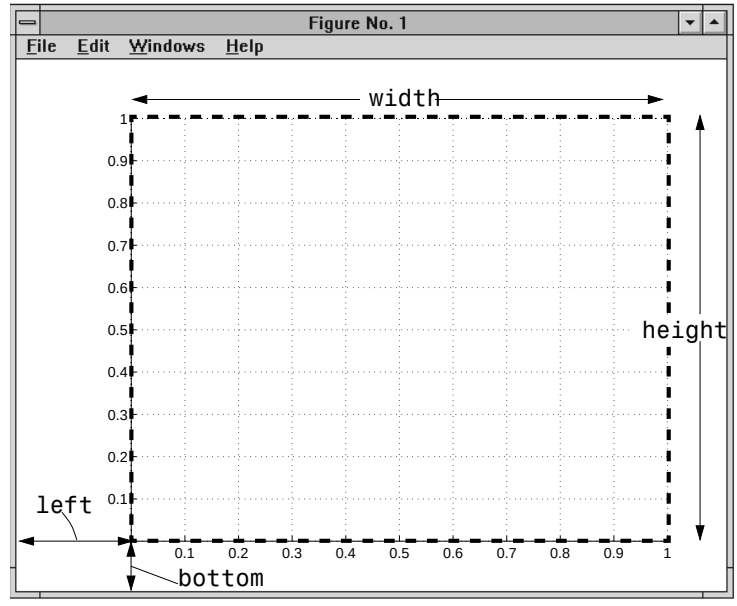
The axes `Position` property controls the size and location of an axes within a figure. The default axes has the same aspect ratio (ratio of width to height) as the default figure and fills most of the figure, leaving a border around the edges. However, you can define the axes position as any rectangle and place it wherever you want within a figure.

The Position Vector

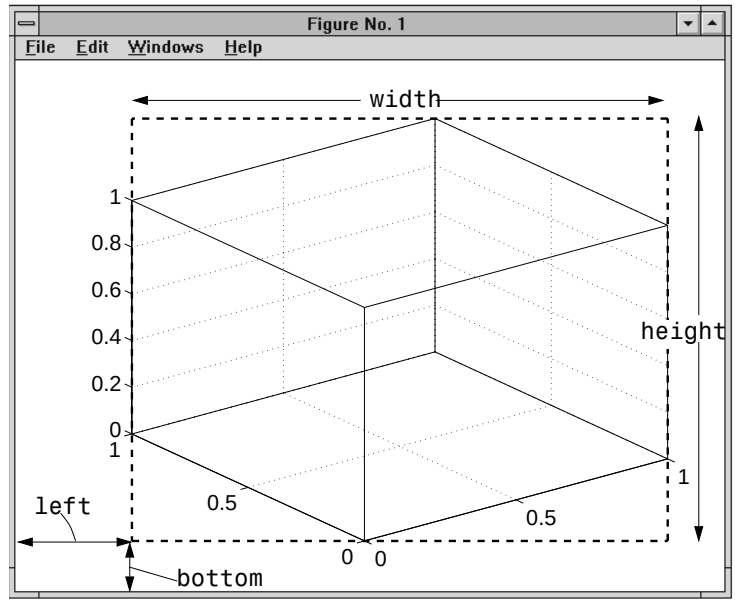
MATLAB defines the axes `Position` property as a vector.

```
[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.

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.

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])
```

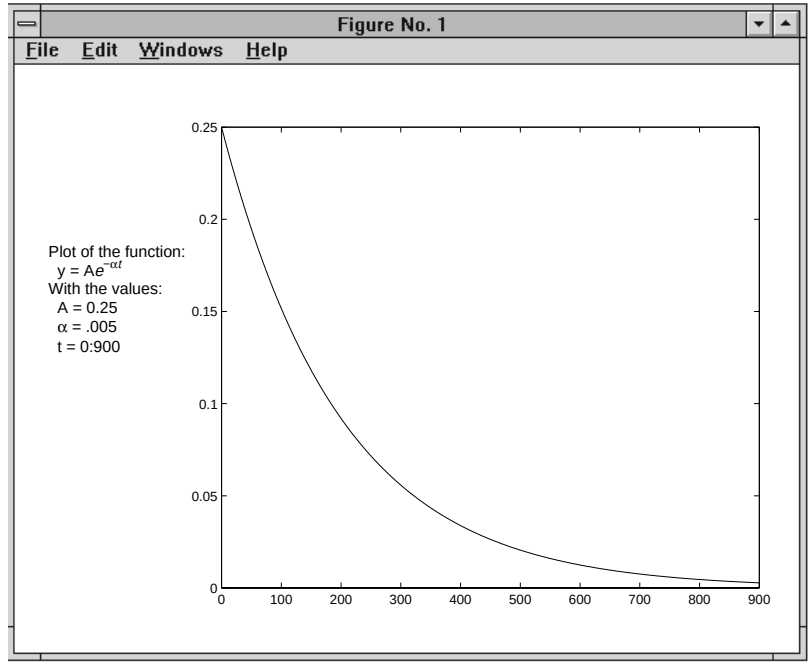
Since the axes units are normalized to the figure, specifying the `Position` as `[0 0 1 1]` creates an axes that encompasses the entire window.

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.

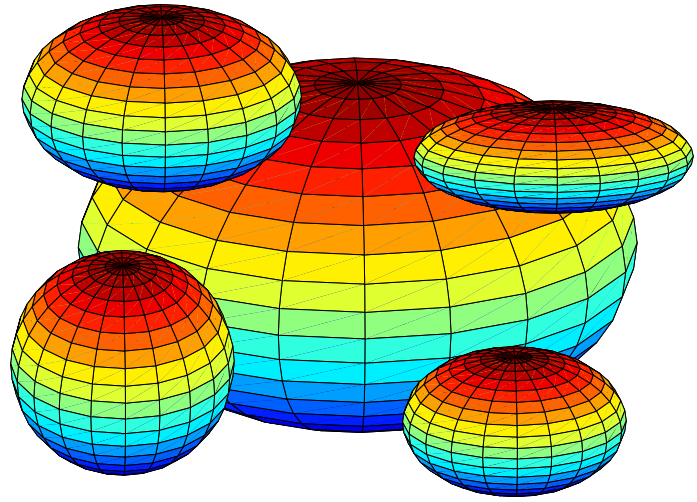
```
str(1) = {'Plot of the function:'};
str(2) = {' y = A{\ite}^{-\alpha{\itt}}'};
str(3) = {'With the values:'};
str(3) = {' A = 0.25'};
str(4) = {' \alpha = .005'};
str(5) = {' t = 0:900'};
set(gcf,'CurrentAxes',h)
text(.025,.6,str,'FontSize',12)
```



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
`XLim,YLim,ZLim`	Sets the axis range.
`XLimMode`, `YLimMode`, `ZLimMode`	Specifies whether axis limits are determined automatically by MATLAB or specified manually by the user.
`XTick`, `YTick`, `ZTick`	Sets the location of the tick marks along the axis.
`XTickMode`, `YTickMode`, `ZTickMode`	Specifies whether tick mark locations are determined automatically by MATLAB or specified manually by the user.
`XTickLabel`, `YTickLabel`, `ZTickLabel`	Specifies the labels for the axis tick marks.
`XTickLabelMode`, `YTickLabelMode`, `ZTickLabelMode`	Specifies whether tick mark labels are determined automatically by MATLAB or specified manually by the user.
`XDir,YDir,ZDir`	Sets the direction of increasing axis values.

Setting Axis Limits

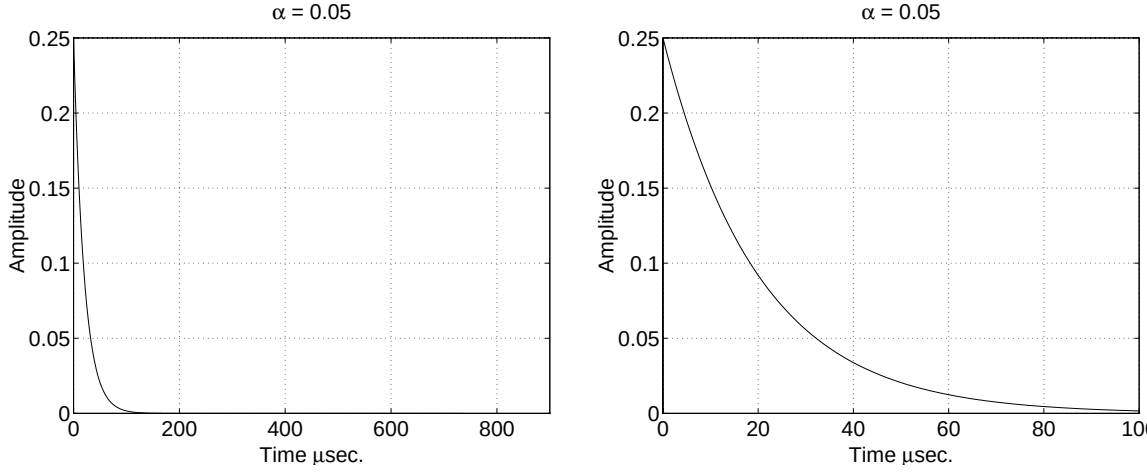
MATLAB determines the limits automatically for each axis based on the range of the data. You can override the selected limits by specifying the `XLim`, `YLim`, or `ZLim` property. For example, consider a plot of the function $Ae^{-\alpha t}$ evaluated with $A = 0.25$, $\alpha = 0.05$, and $t = 0$ to 900.

```
t = 0:900;
plot(t,0.25*exp(-0.05*t))
```

The plot on the left shows the results. MATLAB selects axis limits that encompass the range of data in both x and y. However, since the plot contains little information beyond $t = 100$, changing the x-axis limits improves the usefulness of the plot.

```
set(axhandle,'XLim',[0 100])
```

The plot on the right shows the result.



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(axhandle,'XLim',[0 Inf])
```

sets the `XLimMode` property to `auto` and allows MATLAB to determine the maximum value for `XLim`. Similarly, the statement,

```
set(axhandle,'XLim',[-Inf 800])
```

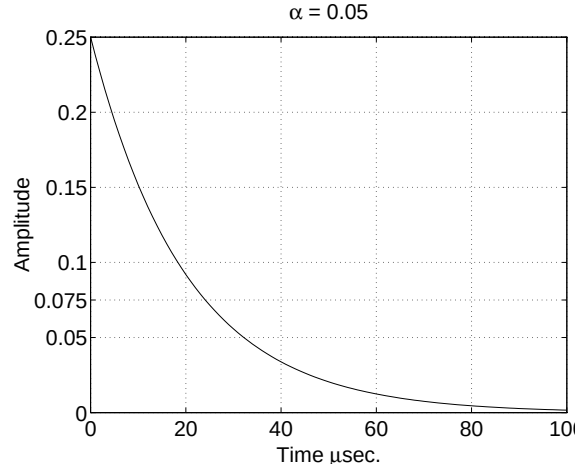
sets the `XLimMode` property to `auto` and allows MATLAB to determine the minimum value for `XLim`.

Setting Tick Mark Locations

MATLAB selects the tick mark location based on the data range to produce equally spaced ticks (for linear graphs). You can specify alternative locations for the tick marks by setting the `XTick`, `YTick`, and `ZTick` properties.

For example, if the value 0.075 is of interest for the amplitude of the function $Ae^{-\alpha t}$, specify tick marks to include that value.

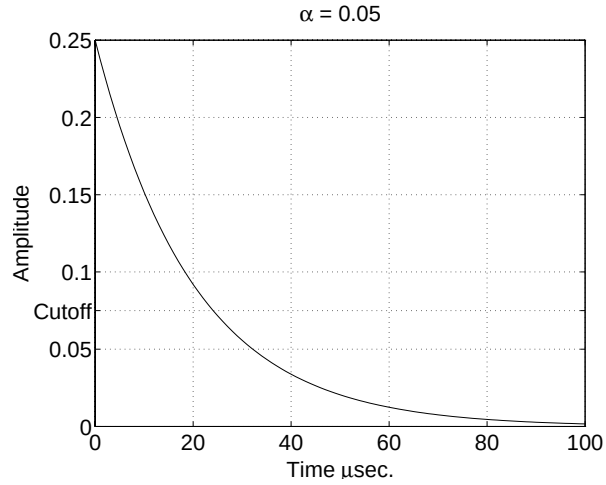
```
set(gca,'YTick',[0 0.05 0.075 0.1 0.15 0.2 0.25])
```



You can change tick labeling from numbers to strings using the `XTickLabel`, `YTickLabel`, and `ZTickLabel` properties.

For example, to label the *y*-axis value of 0.075 with the string `Cutoff`, you can specify all *y*-axis labels as a string, separating each label with the "|" character:

```
set(gca,'YTickLabel','0|0.05|Cutoff|0.1|0.15|0.2|0.25')
```



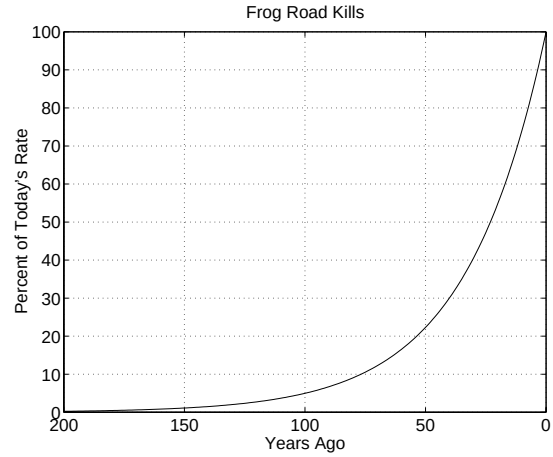
Changing Axis Direction

The `XDir`, `YDir`, and `ZDir` properties control the direction of increasing values on the respective axis. In the default 2-D view, the *x*-axis values increase from left to right and the *y*-axis values increase from bottom to top. The *z*-axis points out of the screen.

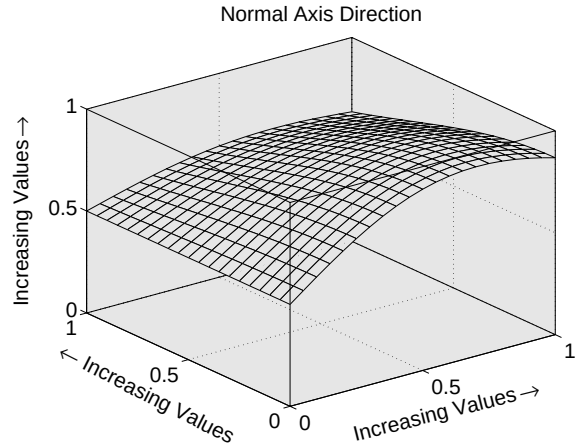
You can change the direction of increasing values by setting the associated property to `reverse`. For example, setting `XDir` to reverse,

```
set(gca,'XDir','reverse')
```

produces a plot whose *x*-axis decreases from left to right.



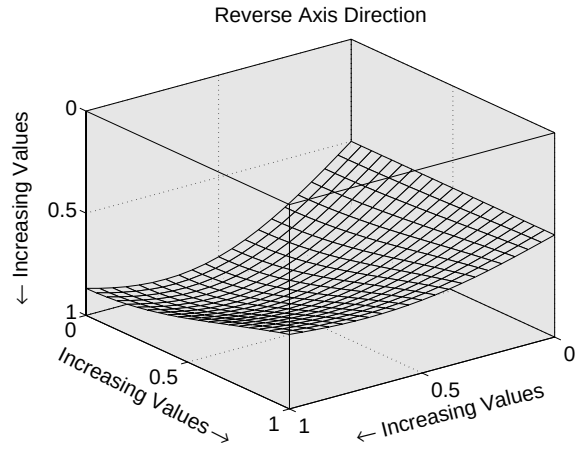
In the 3-D view, the *y*-axis increases from front to back and the *z*-axis increases from bottom to top.



Setting the *x*-, *y*-, and *z*-directions to reverse,

```
set(gca,'XDir','rev','YDir','rev','ZDir','rev')
```

yields:


Reverse Axis Direction
0
0.5
1
← Increasing Values
0
0.5
1
Increasing Values →
0
0.5
1
← Increasing Values

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:

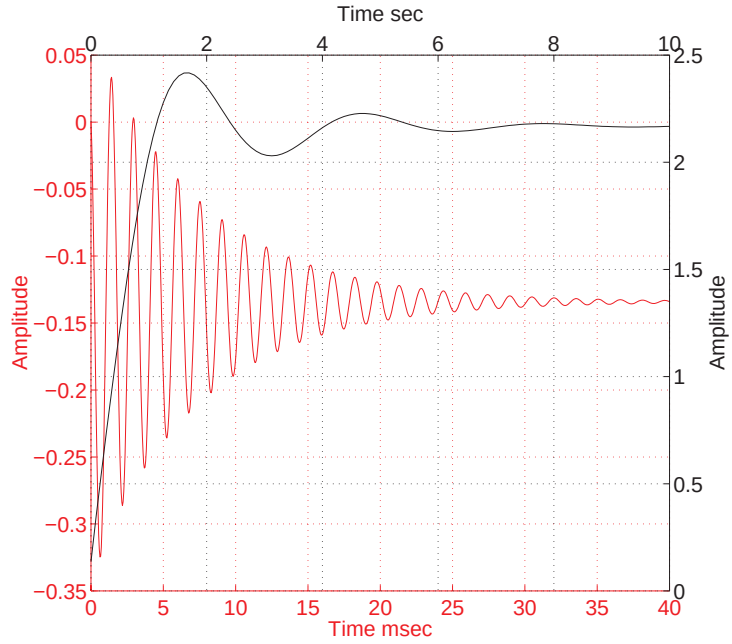
```
hl1 = line(x1,y1,'Color','r');
ax1 = gca;
set(ax1,'XColor','r','YColor','r')
```

Next, create another axes at the same location as the first, placing the *x*-axis on top and the *y*-axis on the right. Set the axes `Color` to `none` to allow the first axes to be visible and color code the *x*- and *y*-axis to match the data.

```
ax2 = axes('Position',get(ax1,'Position'),...
           'XAxisLocation','top',...
           'YAxisLocation','right',...
           'Color','none',...
           'XColor','k','YColor','k');
```

Draw the second set of data in the same color as the *x*- and *y*-axis.

```
hl2 = line(x2,y2,'Color','k','Parent',ax2);
```



Creating Coincident Grids

Since the two axes are completely independent, MATLAB determines tick mark locations according to the data plotted in each. It is unlikely the gridlines will coincide. This produces a somewhat confusing looking graph, even though the two grids are drawn in different colors. However, if you manually specify tick mark locations, you can make the grids coincide.

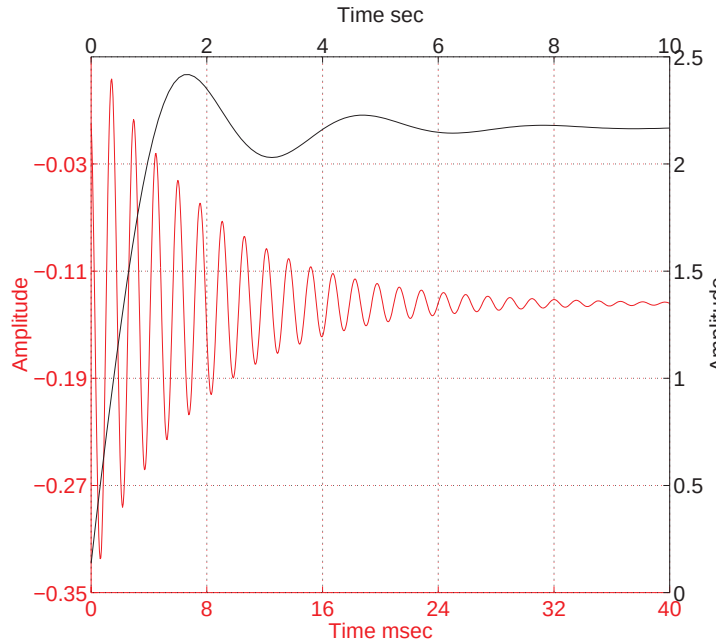
The key is to specify the same number of tick marks along corresponding axis lines (it is also necessary for both axes to be the same size). The following graph of the same data uses six tick marks per axis, equally spaced within the original limits. To calculate the tick mark location, obtain the limits 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',[xlimits(1):xinc:xlimits(2)],...
        'YTick',[ylimits(1):yinc:ylimits(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

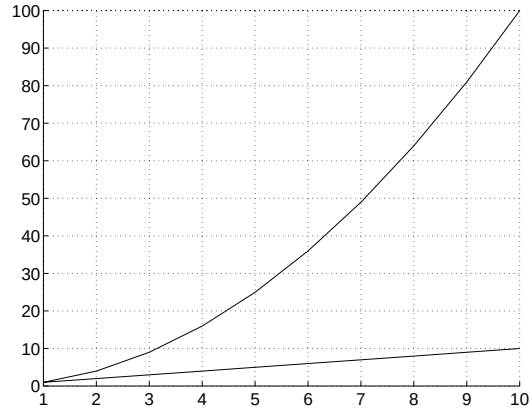
`XTickMode`
`YTickMode`
`ZTickMode`
	Tick mark spacing along the respective *x*, *y*, and *z* axes

`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]`.



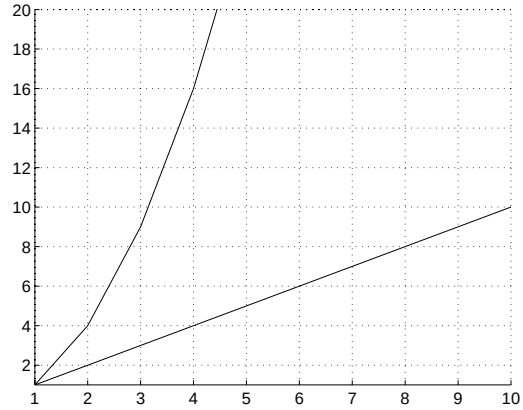
This is because `YLimMode` is `auto`, which always causes MATLAB to recompute the axis limits.

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.


20
18
16
14
12
10
8
6
4
2
1
2
3
4
5
6
7
8
9
10

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 autocycle order
`LineStyleOrder`	Line styles autocycle order (not a color, but related to `ColorOrder`)

Specifying Axes Colors

The default axes background color is set up by the `colordef` command, which is called in your startup file. However, you can easily define your own color scheme.

See "Reversing Figure Color" 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')
```

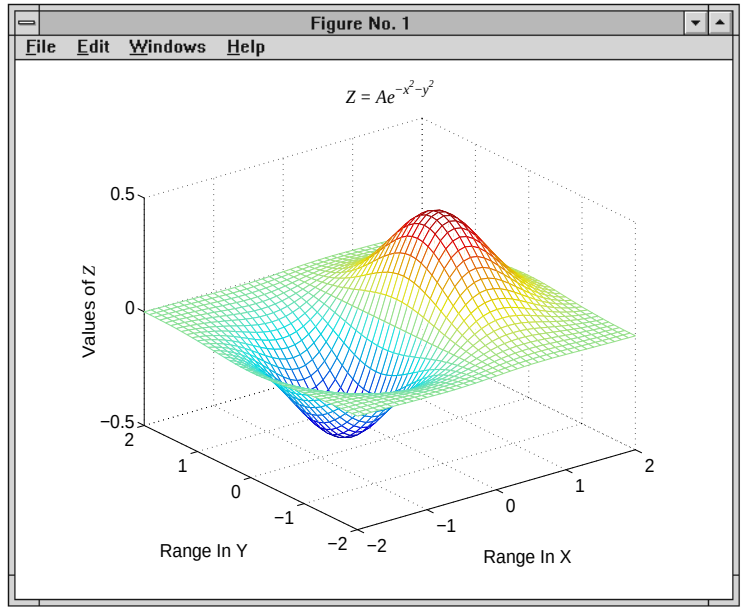
Next, change the color of the text objects used for the title and axis labels.

```
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:

```
colormap_index = fix((CData-cmin)/(cmax-cmin)*cm_length))+1
```

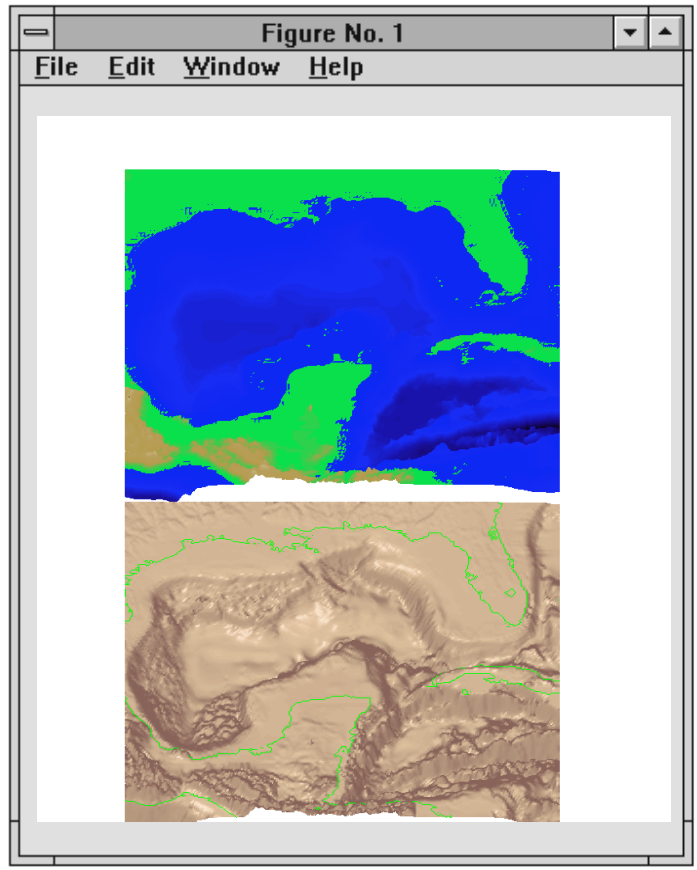
`cm_length` is the length of the colormap. When `CLimMode` is `auto`, MATLAB sets `CLim` to the range of the `CData` of all graphics objects within the axes. However, you can set `CLim` to span any range of values. This allows individual axes within a single figure to use different portions of the figure's 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]);
surfl(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)))
```

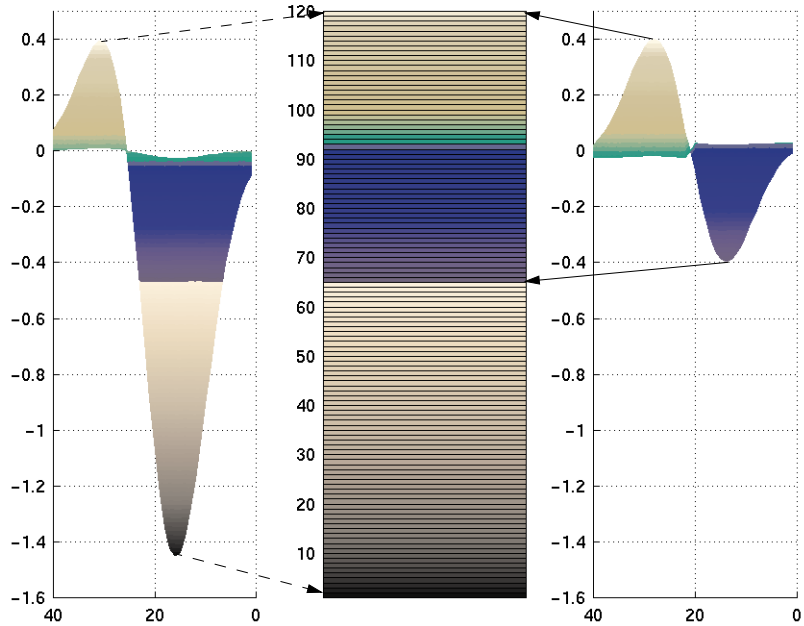
How the Function Works

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.



Virtual surface mapped to entire 120 slot colormap

Figure Colormap

Real surface using only color slots 65 to 120

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's startup.

Defining Your Own ColorOrder

You can redefine `ColorOrder` to be any *m*-by-3 matrix of RGB values, where *m* is the number of colors. However, high-level functions like `plot` and `plot3` reset most axes properties (including `ColorOrder`) to 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

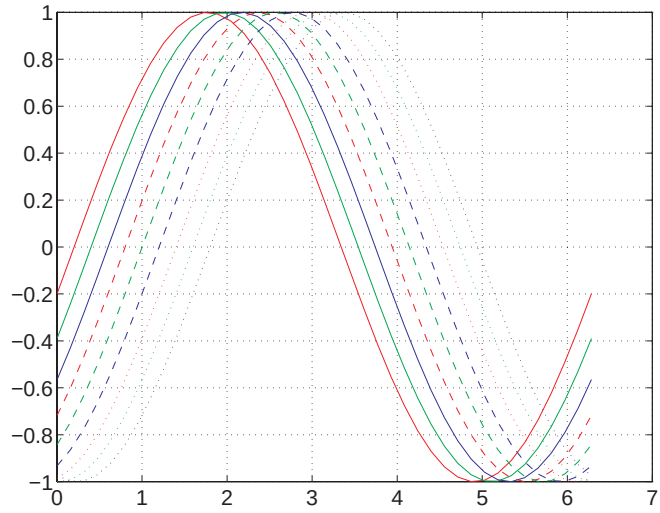
The axes `LineStyleOrder` property is analogous to the `ColorOrder` property. It specifies the line styles to use for multiline plots created with the `plot` and `plot3` functions. MATLAB increments the line style only after using all of the colors in the `ColorOrder` property. It then uses all the colors 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.

Index

A

D

E

F

G

J

L

M

N

O

P

Q

R

S

T

X

Y

Z